THE UNIVERSITY OF WARWICK

High Performance Computing *Course Notes*

Message Passing Programming II

Dr Ligang He

MPI functions

MPI is a complex system comprising of numerous functions with various parameters and variants

Six of them are indispensable, but can write a large number of useful programs already

Other functions add flexibility (datatype), robustness (non-blocking send/receive), efficiency (ready-mode communication), modularity (communicators, groups) or convenience (collective operations, topology).

In the lectures, we are going to cover most commonly encountered functions

Modularity

- MPI supports modular programming via communicators
- Provides information hiding by encapsulating local communications and having local namespaces for processes
- All MPI communication operations specify a communicator (process group that is engaged in the communication)

Creating new communicators – Approach 1

```
MPI_Comm world, workers;
MPI_Group world_group, worker_group;
int ranks[1];
MPI_Init(&argc, &argv);
world=MPI_COMM_WORLD;
MPI_Comm_size(world, &numprocs);
MPI_Comm_rank(world, &myid);
server=numprocs-1;
MPI_Comm_group(world, &world_group);
ranks[0]=server;
MPI_Group_excl(world_group, 1, ranks, &worker_group);
MPI_Comm_create(world, worker_group, &workers);
MPI_Group_free(&world_group);
MPI_Comm_free(&workers);
```

int MPI_Group_excl (MPI_Group group, int n, int *ranks, MPI_Group *newgroup)

Creating new communicators – functions

int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)

int MPI_Group_excl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)

int MPI_Group_incl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)

int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)

int MPI_Group_free(MPI_Group *group)

int MPI_Comm_free(MPI_Comm *comm)

Creating new communicators – Approach 2

MPI_Comm_split (comm, colour, key, newcomm)

Creates one or more new communicators from the original comm

comm communicator (handle)

colour control of subset assignment (processes with same colour are in same new communicator)

key control of rank assignment

newcomm new communicator

Is a collective communication operation (must be executed by all processes in the comm)

Is used to (re-) allocate processes to communicator (groups)

Creating new communicators – Approach 2

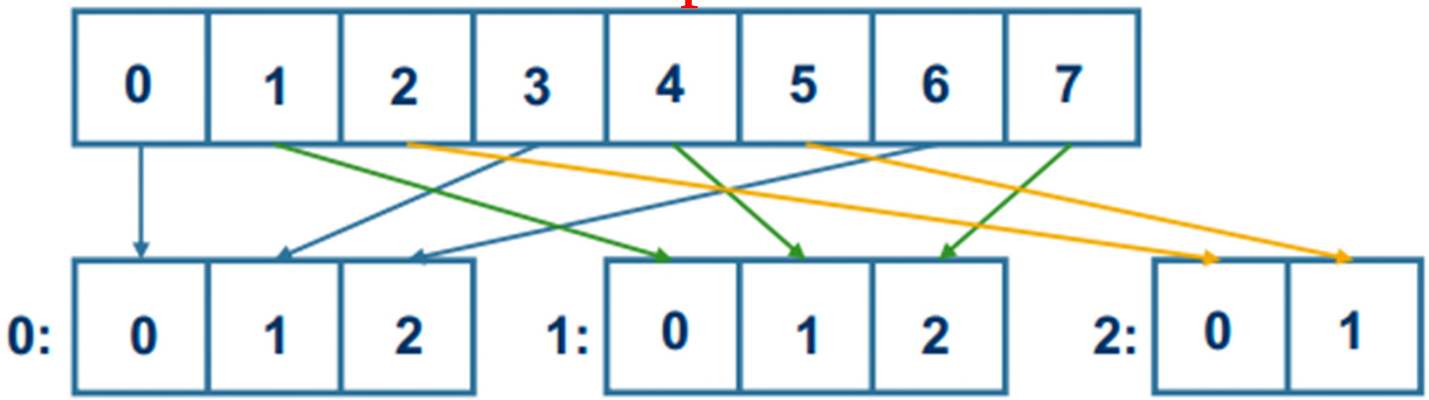
MPI_Comm_split (comm, colour, key, newcomm)

- The number of communicators created is the same as the number of different values of *colour*
- The processes with the same value of *colour* will be put in the same communicator
- These processes are assigned new ID (starting at zero) with the order determined by the value of *key*

Creating new communicators – Approach 2

MPI_Comm_split (comm, colour, key, newcomm)

```
MPI_Comm comm, newcomm; int myid, color;
MPI_Comm_rank(comm, &myid); //id of current process
color = myid%3;
MPI_Comm_split(comm, colour, myid, *newcomm);
```

Communications

Point-to-point communications: involving exact two processes, one sender and one receiver

For example, MPI_Send() and MPI_Recv()

Collective communications: involving a group of processes

Collective operations

- **Coordinated communication operations involving multiple processes**
- **Programmer could do this by hand (tedious), MPI**
- **provides a specialized *collective communications***

***barrier* – synchronize all processes**

***reduction* operations – sums, multiplies etc. distributed data**

***broadcast* – sends data from one to all processes**

***gather* – gathers data from all processes to one process**

***scatter* – scatters data from one process to all processes**

All executed collectively (by all processes in the group, at the same time, with the same parameters)

Collective operations

MPI_Barrier (comm)

Global synchronization

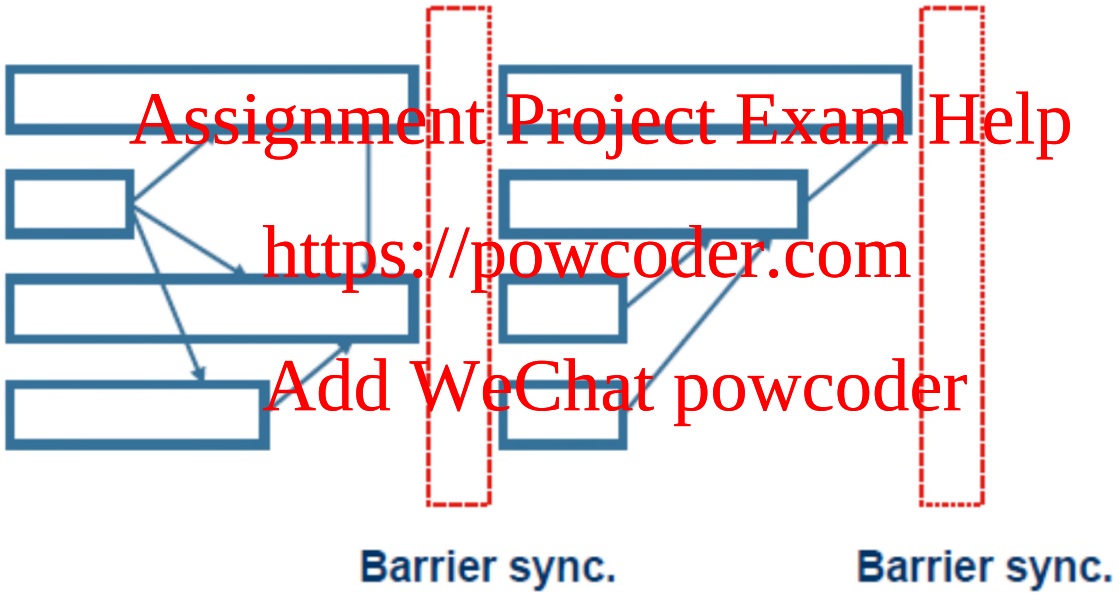
***comm* is the communicator handle**

No processes return from function until all processes have called it

Good way of separating one phase from another

Barrier synchronizations

Barrier synchronizations



Implementation of Barrier

- Depending on the versions of MPI implementations
- A possible implementation is to pass a token message around processes from process 0 to n-1,

- e.g., When a process calls MPI_Barrier
  - If it is Process 0, it sends a token to process 1, and then waits for the token to be received from process n-1
  - For another process i, it waits to receive a token from process i-1 and then send the token to process i+1

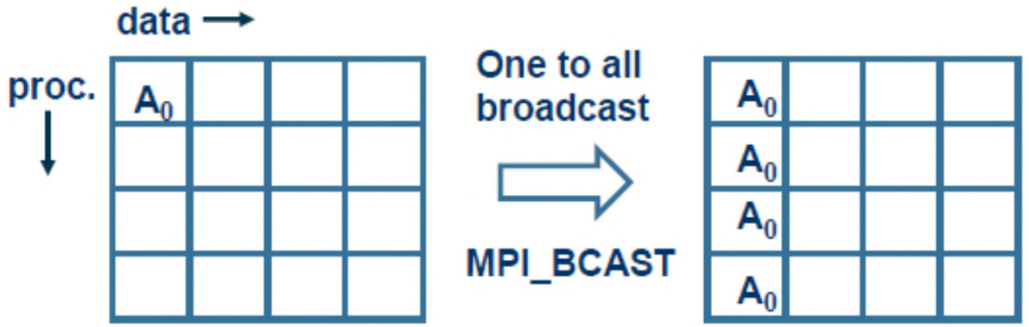
- No process can continue until the token has been passed back to Process 0.

Collective operations

MPI_Bcast (buf, count, type, root, comm)

Broadcast data from root to all processes

buf	address of receive buffer or sender's buffer (root)
count	no. of entries in buffer (>=0)
type	datatype of buffer elements
root	process id of root process
comm	communicator



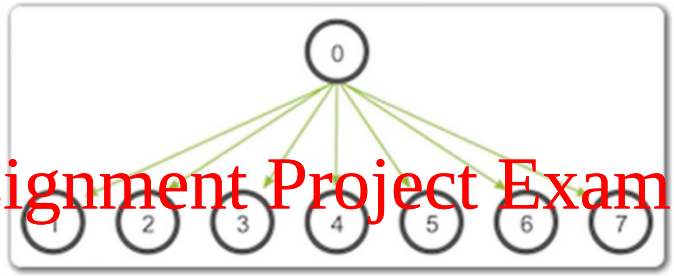
Example of MPI_Bcast

Broadcast 100 ints from process 0 to every process in the group


```
MPI_Comm comm;
int array[100];
int root = 0;
...
MPI_Bcast (array, 100, MPI_INT, root, comm);
```

Implementation of MPI_Bcast

Naïve Implementation:



Smarter Implementation:

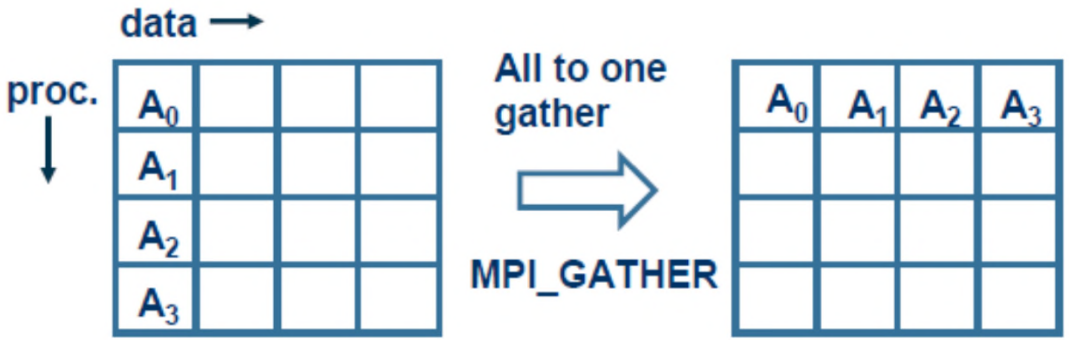
The number of messages being transported is reduced.

Collective operations

MPI_Gather (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

Collective data movement function

sendbuf	address of input buffer
sendcount	no. of elements sent from each (>=0)
sendtype	datatype of input buffer elements
recvbuf	address of output buffer (var param)
recvcount	no. of elements received from each
recvtype	datatype of output buffer elements
root	process id of root process
comm	communicator



Collective operations

MPI_Gather (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

Collective data movement function

sendbuf ***address of send buffer***

sendcount ***no. of elements sent from each (>=0)***

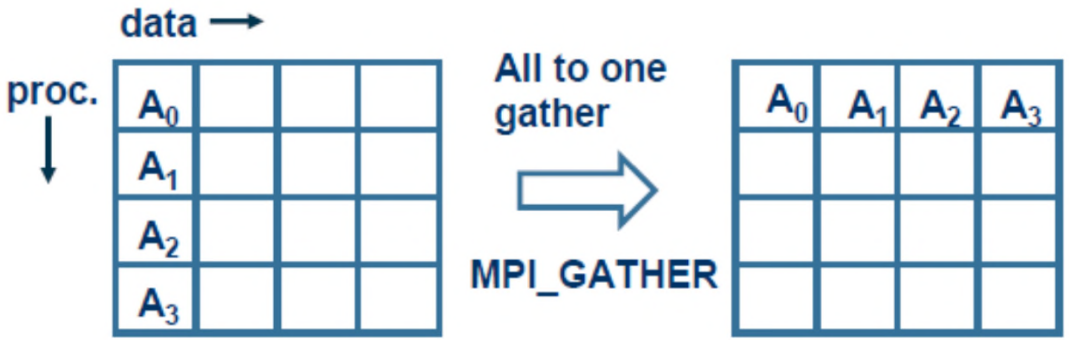
sendtype ***datatype of send buffer elements***

recvbuf **address of recv buffer (var param)** ***recvcount*** **no. of elements received from each**

recvtype **datatype of recv buffer elements**

root **process id of root process**

comm **communicator**

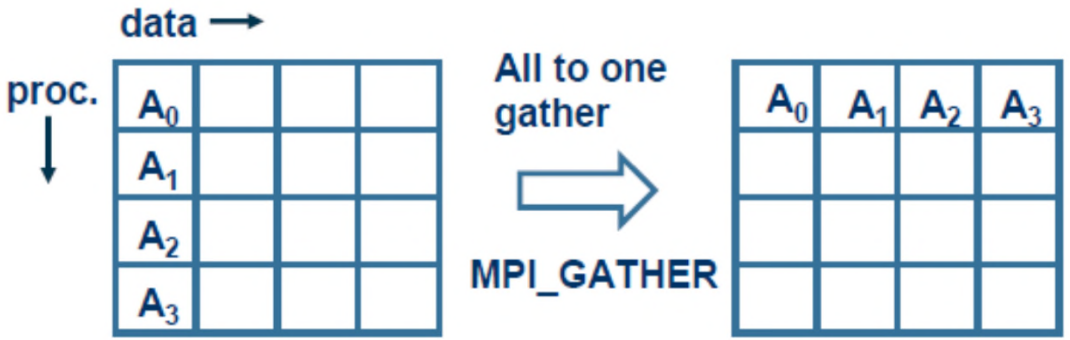


Collective operations

MPI_Gather (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

Collective data movement function

sendbuf	***address of send buffer***
sendcount	***no. of elements sent from each (>=0)***
sendtype	***datatype of send buffer elements***
recvbuf	**address of recv buffer (var param)**
recvcount	**no. of elements received from each**
recvtype	**datatype of recv buffer elements**
root	**process id of root process**
comm	**communicator**

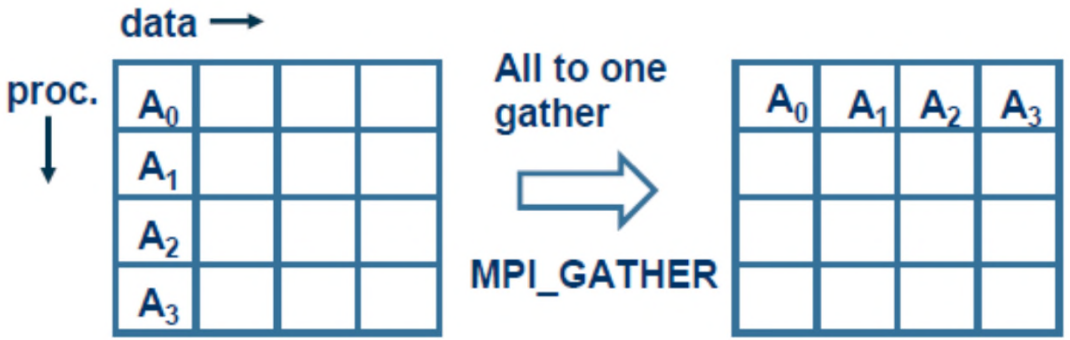


Collective operations

MPI_Gather (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

Collective data movement function

sendbuf	***address of send buffer***
sendcount	***no. of elements sent from each (>=0)***
sendtype	***datatype of send buffer elements***
recvbuf	**address of recv buffer (var param)**
recvcount	**no. of elements received from each**
recvtype	**datatype of recv buffer elements**
root	**process id of root process**
comm	**communicator**

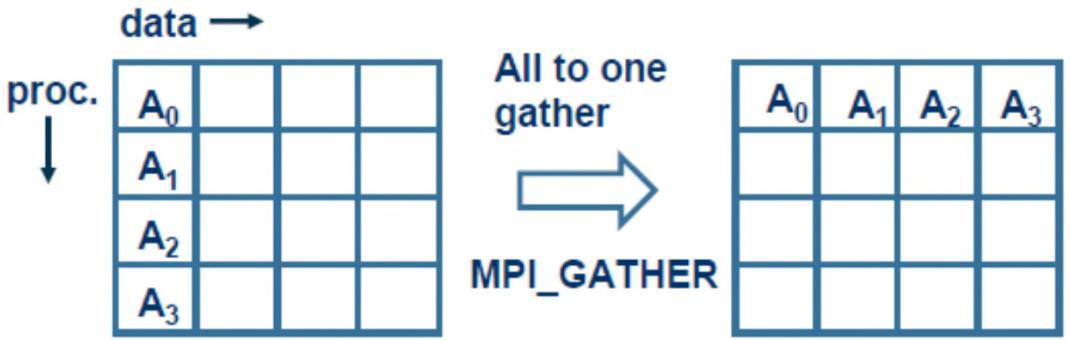


Collective operations

MPI_Gather (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

Collective data movement function

sendbuf	***address of send buffer***
sendcount	***no. of elements sent from each (>=0)***
sendtype	***datatype of send buffer elements***
recvbuf	**address of recv buffer (var param) //note the size**
recvcount	**no. of elements received from each**
recvtype	**datatype of recv buffer elements**
root	**process id of root process**
comm	**communicator**

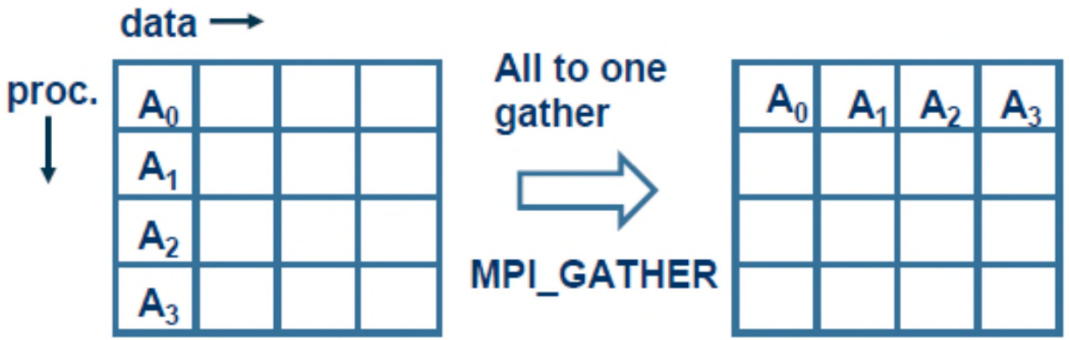


Collective operations

MPI_Gather (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

Collective data movement function

sendbuf	*address of send buffer*
sendcount	*no. of elements sent from each (>=0)*
sendtype	*datatype of send buffer elements*
recvbuf	address of output buffer (var param)
recvcount	no. of elements received from each
recvtype	datatype of output buffer elements
root	process id of root process
comm	communicator



MPI_Gather

- Collective: All processes call MPI_Gather at the same time.

MPI_Gather (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

- Different processes interpret the parameters in different ways

  - If the calling process is root, it looks at these parameters

MPI_Gather (recvbuf, recvcount, recvtype, root, comm)

  - If the calling process is non-root, it looks at these parameters

MPI_Gather (sendbuf, sendcount, sendtype, root, comm)

One more thing about root: root moves data from sendbuff to recvbuff.

MPI_Gather example

Gather 100 ints from every process in group to root

```
MPI_Comm comm;
Int gsize, sendarray[100];
Int root, myrank, *rbuf;
…
MPI_Comm_rank( comm, myrank );                    // find proc. id
If (myrank == root){
        MPI_Comm_size(comm, &gsize);             // find group size
        rbuf = (int *) malloc(gsize*100*sizeof(MPI_INT)); // calc. receive buffer
}
// MPI_Gather is run by all processes at the same time
MPI_Gather(sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```

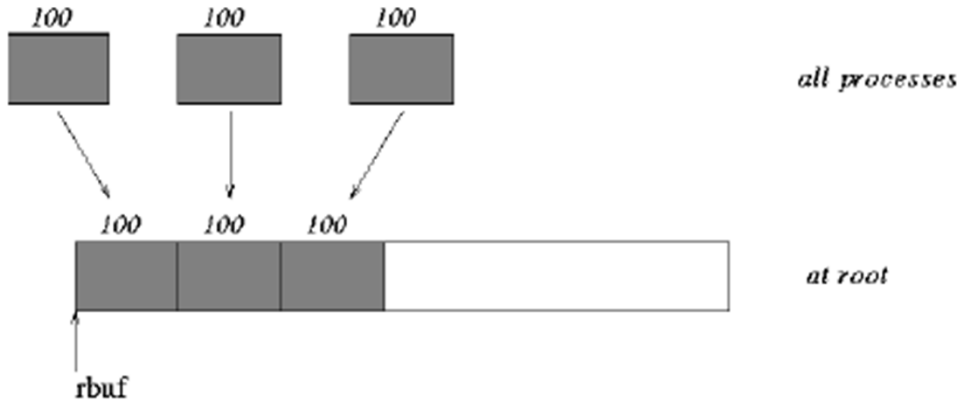
How to use MPI_Send and MPI_Recv to achieve the equivalent outcome as MPI_Gather?

All processes perform

MPI_Send(sendbuf, sendcount, sendtype, root, …),

The root process does MPI_Recv for each process i (i=0, …, n-1)

*MPI_Recv(recvbuf + i*recvcount*sizeof(recvtype), recvcount, recvtype, i, …),*

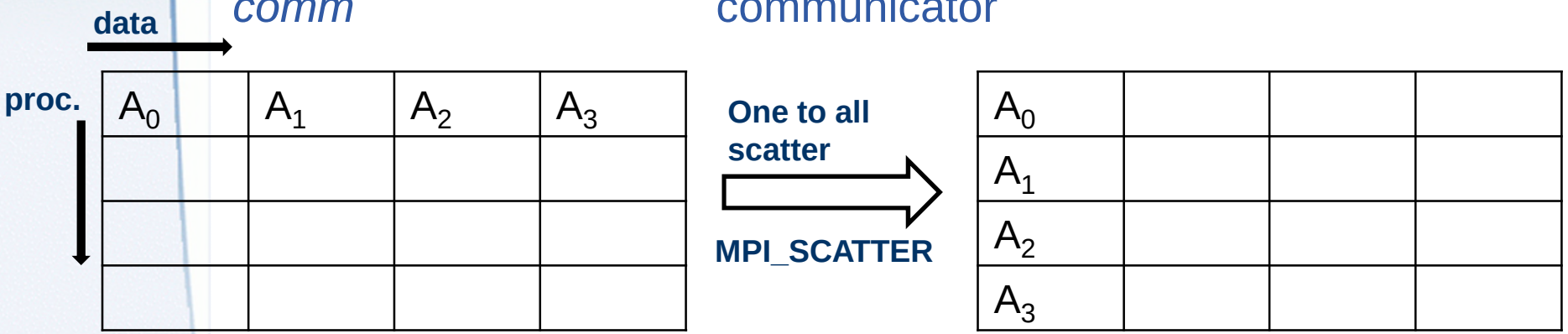


Collective operations

MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

Collective data movement function

sendbuf	address of send buffer // note buff size
sendcount	no. of elements sent to each (>=0)
sendtype	datatype of send buffer elements
recvbuf	address of recv buffer
recvcount	no. of elements received by each
recvtype	datatype of recv buffer elements
root	process id of root process
comm	communicator



Example of MPI_Scatter

MPI_Scatter is reverse of MPI_Gather

```
MPI_Comm comm;
int gsize, *sendbuf;
int root, rbuff[100];
...
MPI_Comm_rank( comm, myrank);          // find proc. id
If (myrank == root) {
MPI_Comm_size (comm, &gsize);
}
sendbuf = (int *) malloc (gsize*100*sizeof(int));
...MPI_Scatter (sendbuf, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```


It is as if the root sends the data in sendbuf using

MPI_Send(sendbuf+i*sendcountsizeof*(MPI_INT), sendcount, sendtype, pid_i , …)**

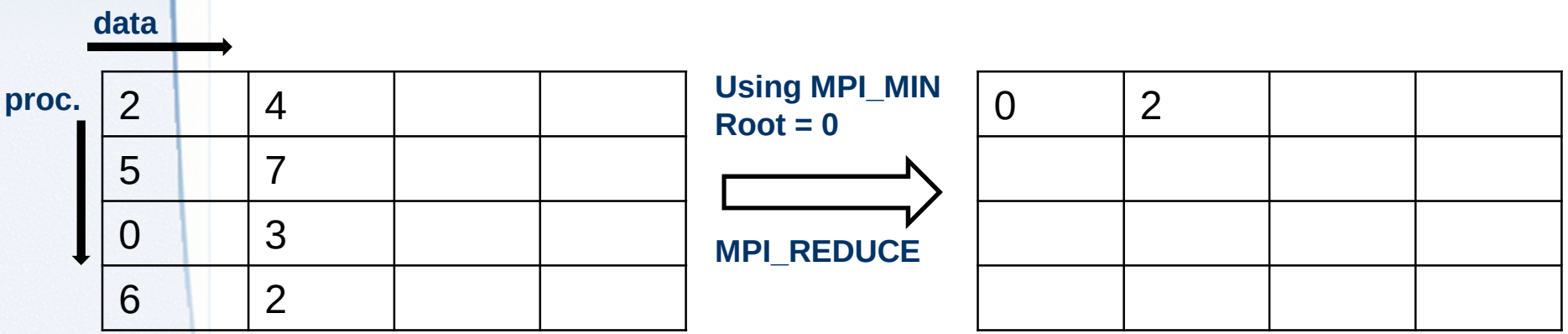
pid_i **is the process id of the i-th process**

Collective operations

MPI_Reduce(sendbuf, recvbuf, count, type, op, root, comm)

root* performs *op* over the data in the *sendbuf* and put the result in the *recvbuf* in *root

sendbuf	address of send buffer
recvbuf	address of recv buffer
count	no. of elements in send buffer (>=0)
type	datatype of send buffer elements
op	*operation*
root	process id of root process
comm	communicator

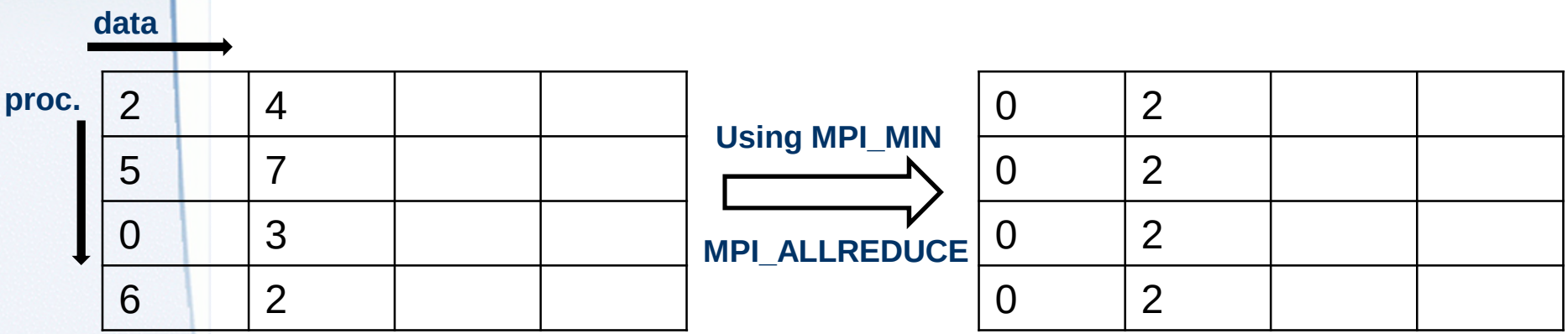


Collective operations

MPI_Allreduce(sendbuf, recvbuf, count, type, op, root, comm)

***root* performs *op* over the data in the *sendbuf* and put the result in the *recvbuf* in all processes**

sendbuf	address of send buffer
recvbuf	address of recv buffer
count	no. of elements in send buffer (>=0)
type	datatype of send buffer elements
op	*operation*
root	process id of root process
comm	communicator

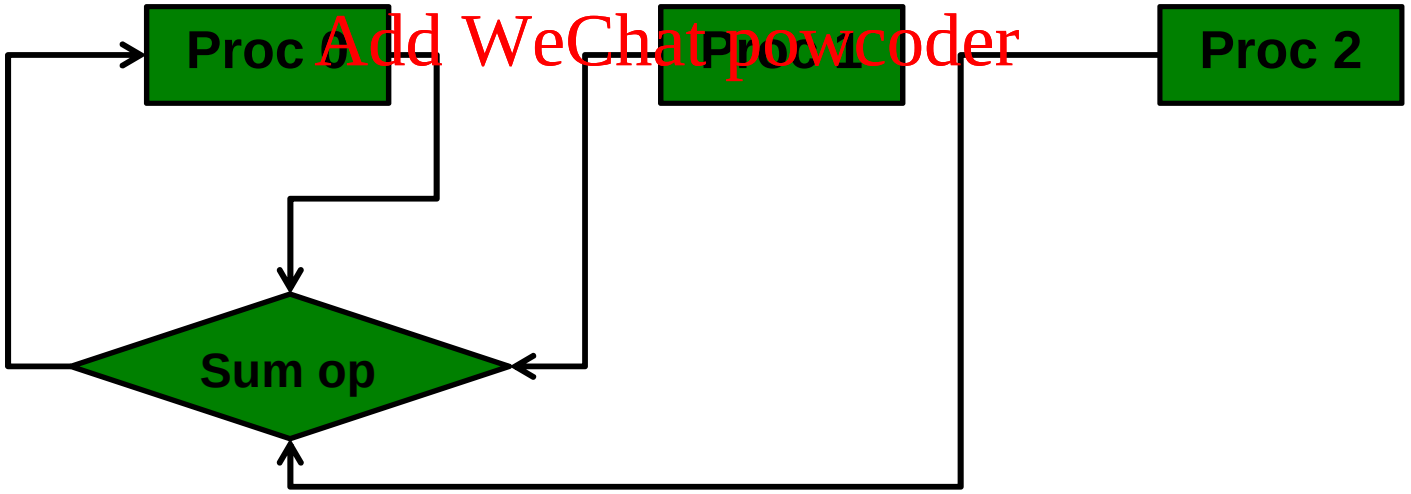


Operations performed in MPI_Reduce

sendbuf and recvbuf are arrays

recvbuf[0] = sum(proc1.sendbuf[0], proc2.sendbuff[0])

recvbuf[1] = sum(proc1.sendbuf[1], proc2.sendbuff[1])



Blocking and non-blocking communications

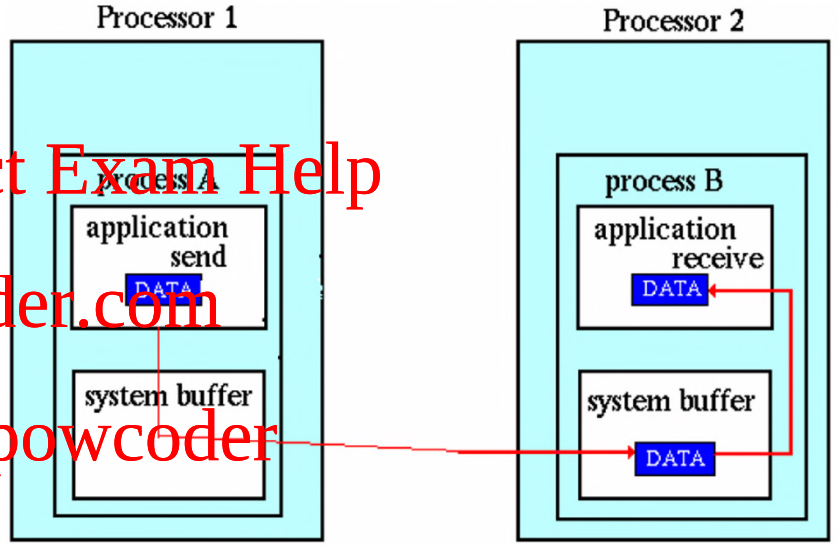
- **Blocking send**

  - **The sender doesn't return until the *application buffer* can be re-used (which often means that the data have been copied from application buffer to *system buffer*) //note: it doesn't mean that the data will be received**

MPI_Send(buf, count, datatype, dest, tag, comm)

Buffering in MPI communications

- **Application buffer: Specified by the first parameter in MPI_Send/Recv functions**



Path of a message buffered at the receiving process

- **System buffer:**
  - **Hidden from the programmer and managed by the MPI library**
  - **is limited and can be easy to exhaust**

MPI_Send(buf, count, datatype, dest, tag, comm)

Blocking and non-blocking communications

- **Blocking send**
  - The sender doesn't return until the application buffer can be re-used (which often means that the data have been copied from application buffer to system buffer) //note: it doesn't mean that the data will be received

MPI_Send(buf, count, datatype, dest, tag, comm)

- **Blocking receive:**

  - The receiver doesn't return until the data have been ready to use by the receiver (which often means that the data have been copied from system buffer to application buffer)

- **Non-blocking send/receive**
  - The calling process returns immediately
  - Just request the MPI library to perform the communication; no gaurantee when this will happen
  - Unsafe to modify the application buffer until you can make sure the requested operation has been performed (MPI provides routines to test this)
  - Can be used to overlap computation with communication and have possible performance gains

MPI_Isend(buf, count, datatype, dest, tag, comm, request)

Testing non-blocking communications

- Completion tests come in two types:
  - WAIT type
  - Test type
- WAIT type: the WAIT type routines block until the communication has been completed.
  - A non-blocking communication immediately followed by a WAIT-type test is equivalent to the corresponding blocking communication
- TEST type: these TEST type routines return immediately with a TRUE or FALSE value
  - The process can perform some other tasks if the communication has not been completed

Testing non-blocking communications for completion

The WAIT-type test is:

MPI_Wait(request, status)

This routine blocks until the communication specified by the request handle has completed. The request handle will have been returned by an earlier call to a non-blocking communication routine.

The TEST-type test is:

MPI_Test (request, flag, status)

In this case the communication specified by the handle request is simply queried to see if the communication has completed and the result of the query (TRUE or FALSE) is returned into flag.

Testing non-blocking communications for completion

Wait for all communications to complete

MPI_Waitall (count, array_of_requests, array_of_statuses)

This routine blocks until all the communications specified by the request handles, array_of_requests, have completed. The statuses of the communications are returned in the array array_of_statuses and each can be queried in the usual way for the source and tag if required

Test if all communications have completed

MPI_Testall (count, array_of_requests, flag, array_of_statuses)

If all the communications have completed, flag is set to TRUE, and information about each of the communications is returned in array_of_statuses. Otherwise flag is set to FALSE and array_of_statuses is undefined.

Testing non-blocking communications for completion

Query a number of communications at a time to find out if any of them have completed

Wait: MPI_Waitany (count, array_of_requests, index, status)

→MPI_WAITANY blocks until one or more of the communications associated with the array of request handles, array_of_requests, has completed.

→The index of the completed communication in the array_of_requests handles is returned in index, and its status is returned in status.

→Should more than one communication have completed, the choice of which is returned is arbitrary.

Test: MPI_Testany (count, array_of_requests, index, flag, status)

→ The result of the test (TRUE or FALSE) is returned immediately in flag.