MPI Programming 2

Faculty Development Program

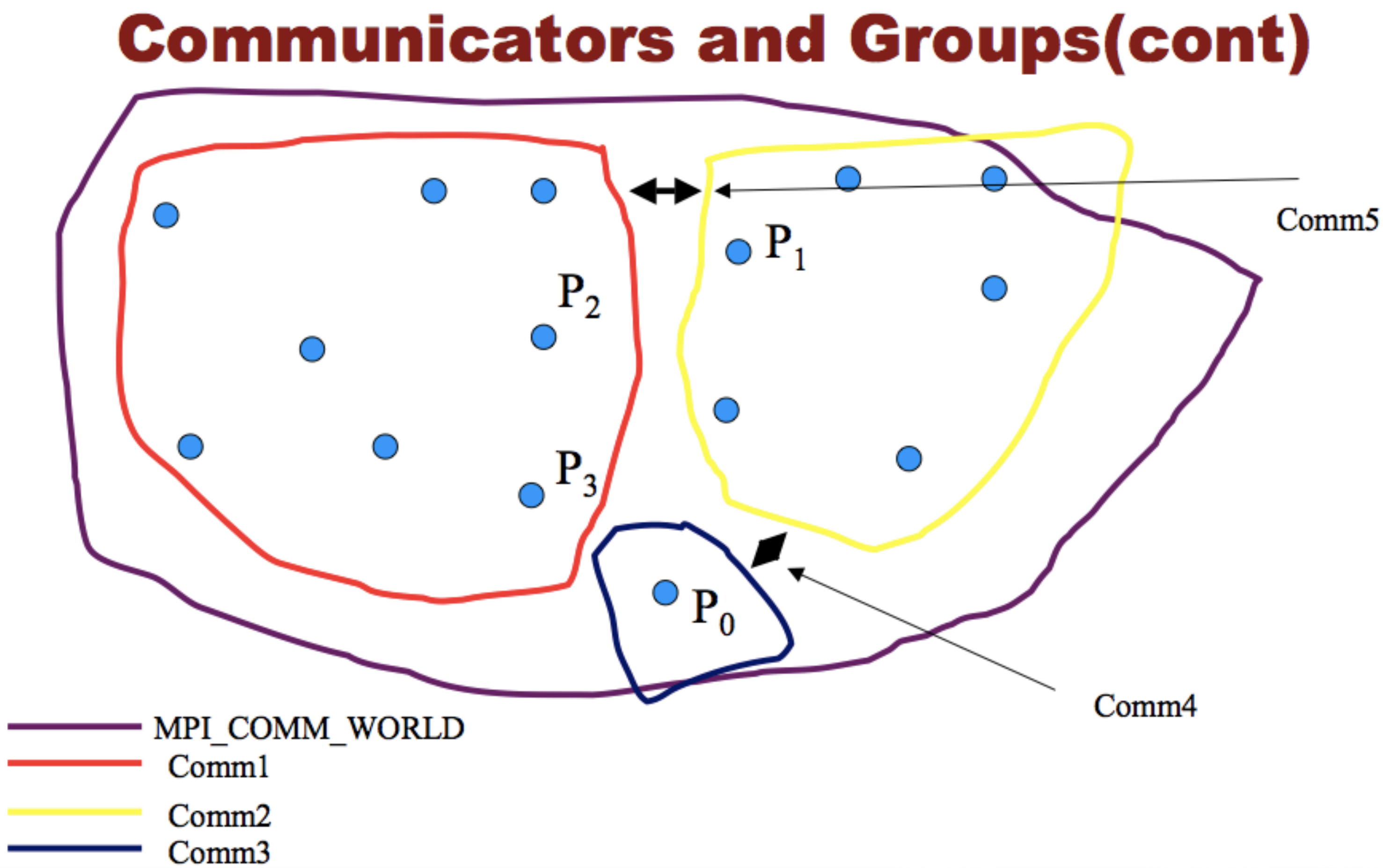
RSET, 17th August 2017

© Fr Dr Jaison Mulerikkal CMI

Collective Communications

- A collective communication always involves every process in the specified communicator.
- To perform a collective communication on a subset of the processes in a communicator, a new communicator has to be created

Communicators



Communicators and Groups

- A communicator can be thought of as a handle to an object (group attribute) that describes a group of processes
- An intracommunicator is used for communication within a single group
- An intercommunicator is used for communication between 2 disjoint groups

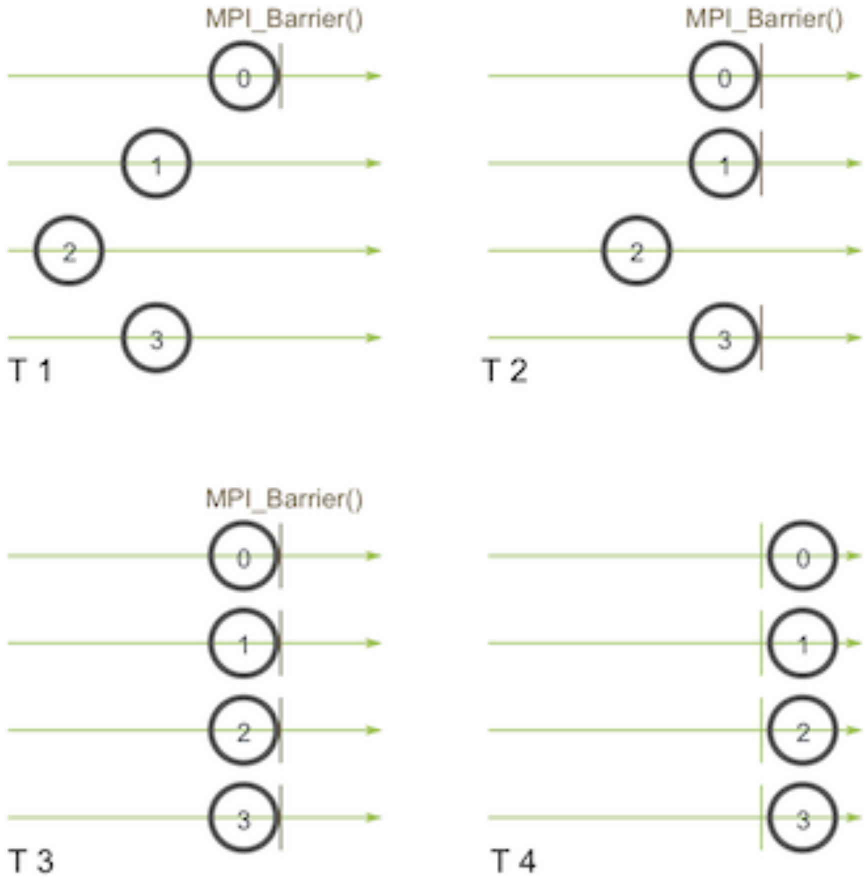
Collective Communications

- Collective communications cannot interfere with point-to-point communications and vice versa
- Collective and point-to-point communication are transparent to one another. For example, a collective communication cannot be picked up by a point-to-point receive.
- There is no concept of tags.

Barrier Synchronisation

- MPI_BARRIER (COMM)
- This is the simplest of all the collective operations and involves no data at all.
- MPI_BARRIER blocks the calling process until all other group members have called it.

Barrier synchronisation



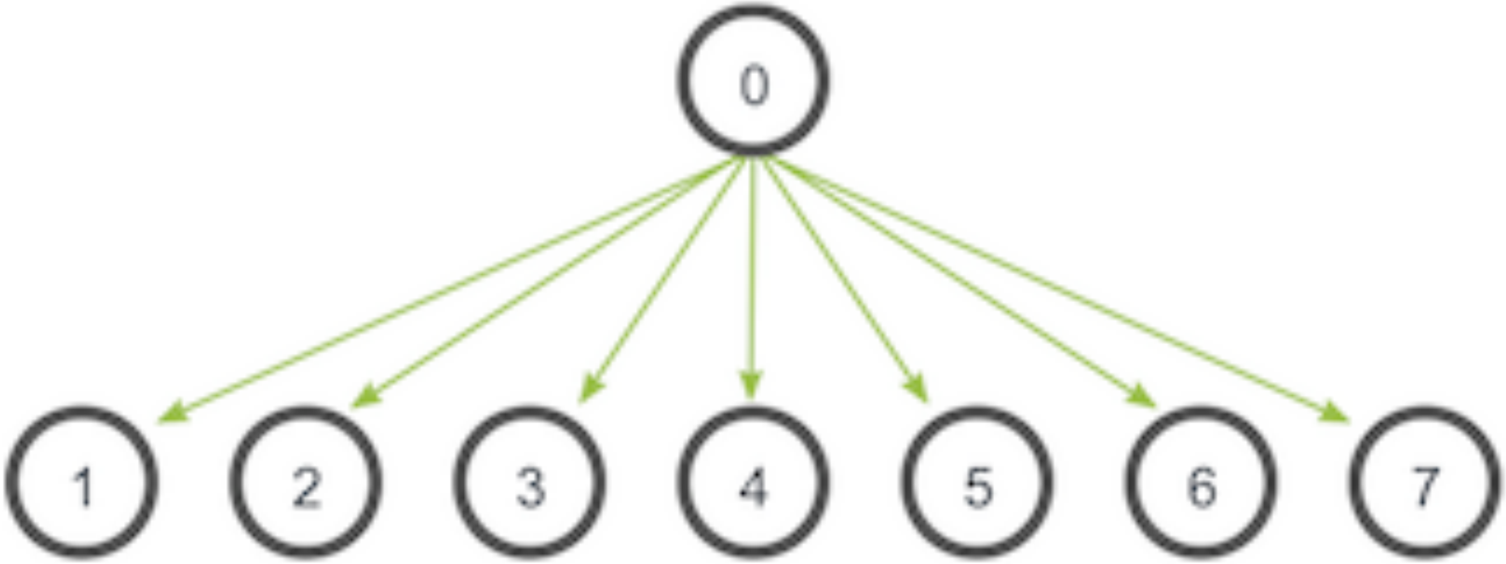
Barrier synchronisation

- In one phase of a computation, all processes participate in writing a file. The file is to be used as input data for the next phase of the computation.
- Therefore no process should proceed to the second phase until all processes have completed phase one.

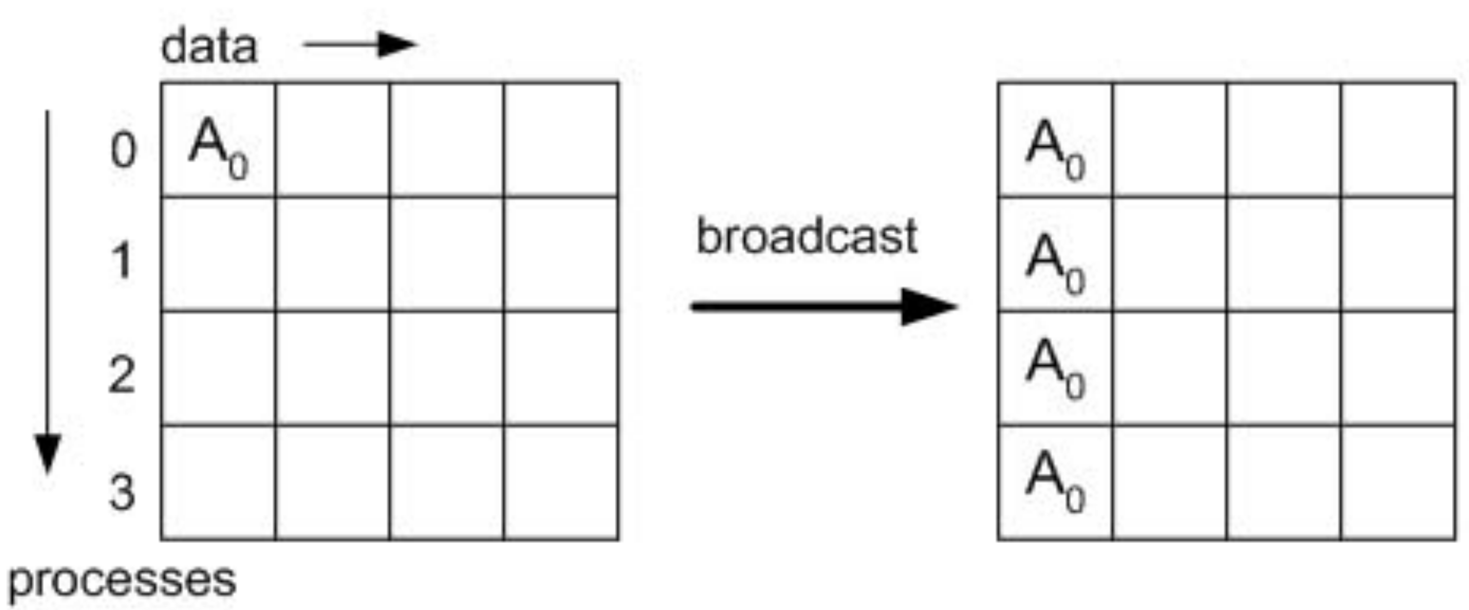
Broadcast

- MPI_BCAST distributes data from one process (the root) to all others in a communicator.
  - MPI_BCAST (buffer, count, datatype, root, comm)
- 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.

Broadcast



Broadcast



Example

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

int main(int argc, char** argv) {
        int rank;
        int buf;
        MPI_Status status;
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);

        if(rank == 0) {
                buf = 777;
                MPI_Bcast(&buf, 1, MPI_INT, 0, MPI_COMM_WORLD);
        }
        else {
                MPI_Recv(&buf, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
                printf("rank %d receiving received %d\n", rank, buf);
        }

        MPI_Finalize();
        return 0;
}
```

Example

- Previous one is wrong!

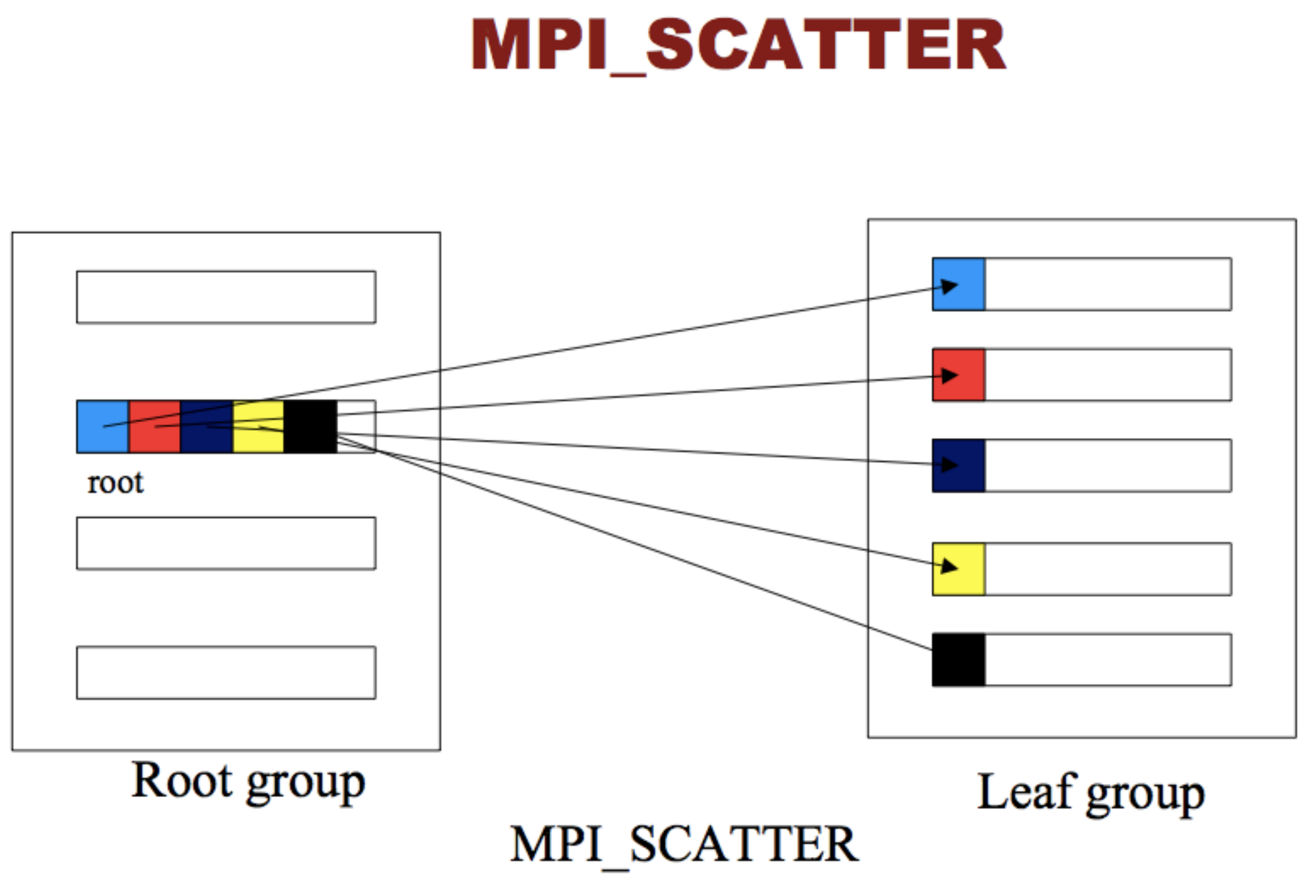
Example: Correct one!

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

int main(int argc, char** argv) {
        int rank;
        int buf;
        const int root=0;

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);

        if(rank == root) {
           buf = 777;
        }

        printf("[%d]: Before Bcast, buf is %d\n", rank, buf);

        /* everyone calls bcast, data is taken from root and ends up in everyone's buf */
        MPI_Bcast(&buf, 1, MPI_INT, root, MPI_COMM_WORLD);

        printf("[%d]: After Bcast, buf is %d\n", rank, buf);

        MPI_Finalize();
        return 0;
}
```

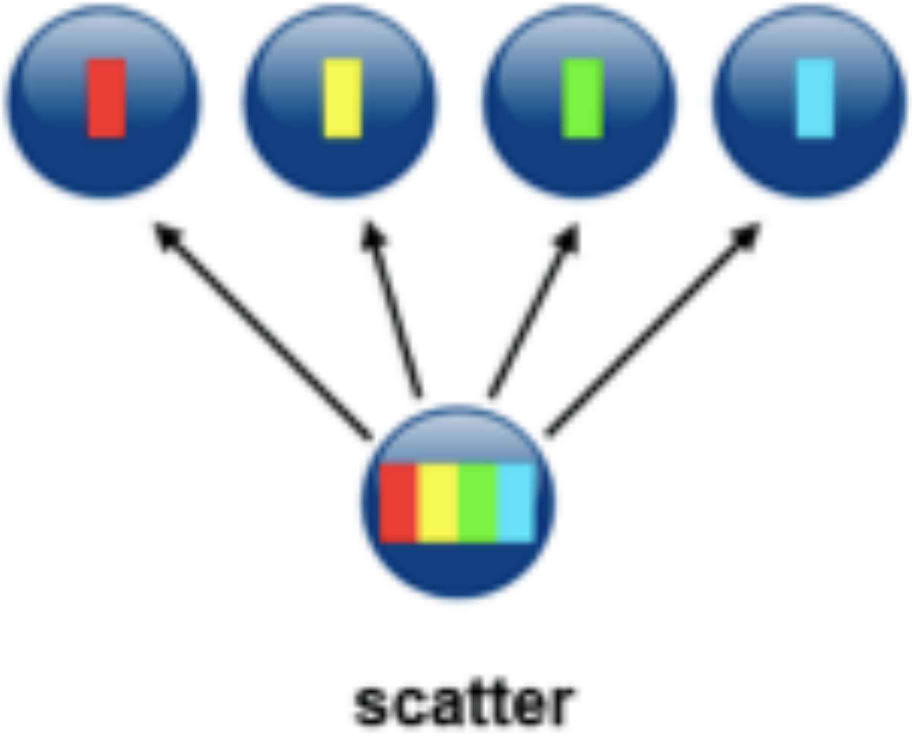
Scatter & Gather

- MPI_SCATTER, MPI_GATHER
- These routines also 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.

MPI Scatter

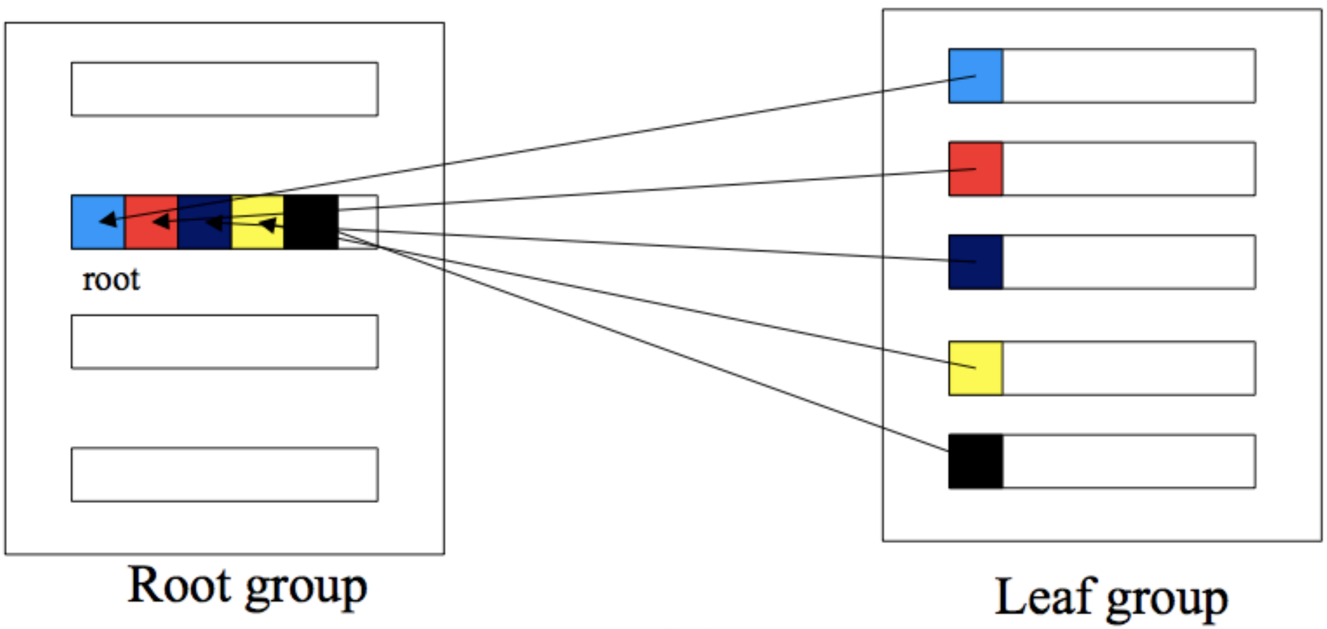


MPI Scatter



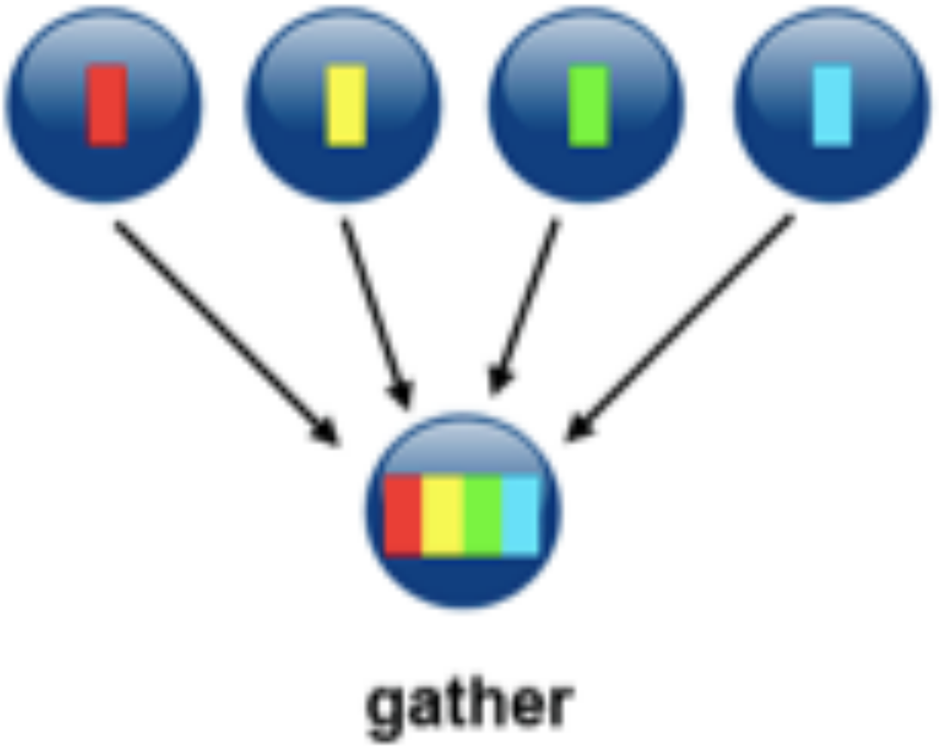
MPI Gather

MPI_GATHER



MPI_GATHER

MPI Gather

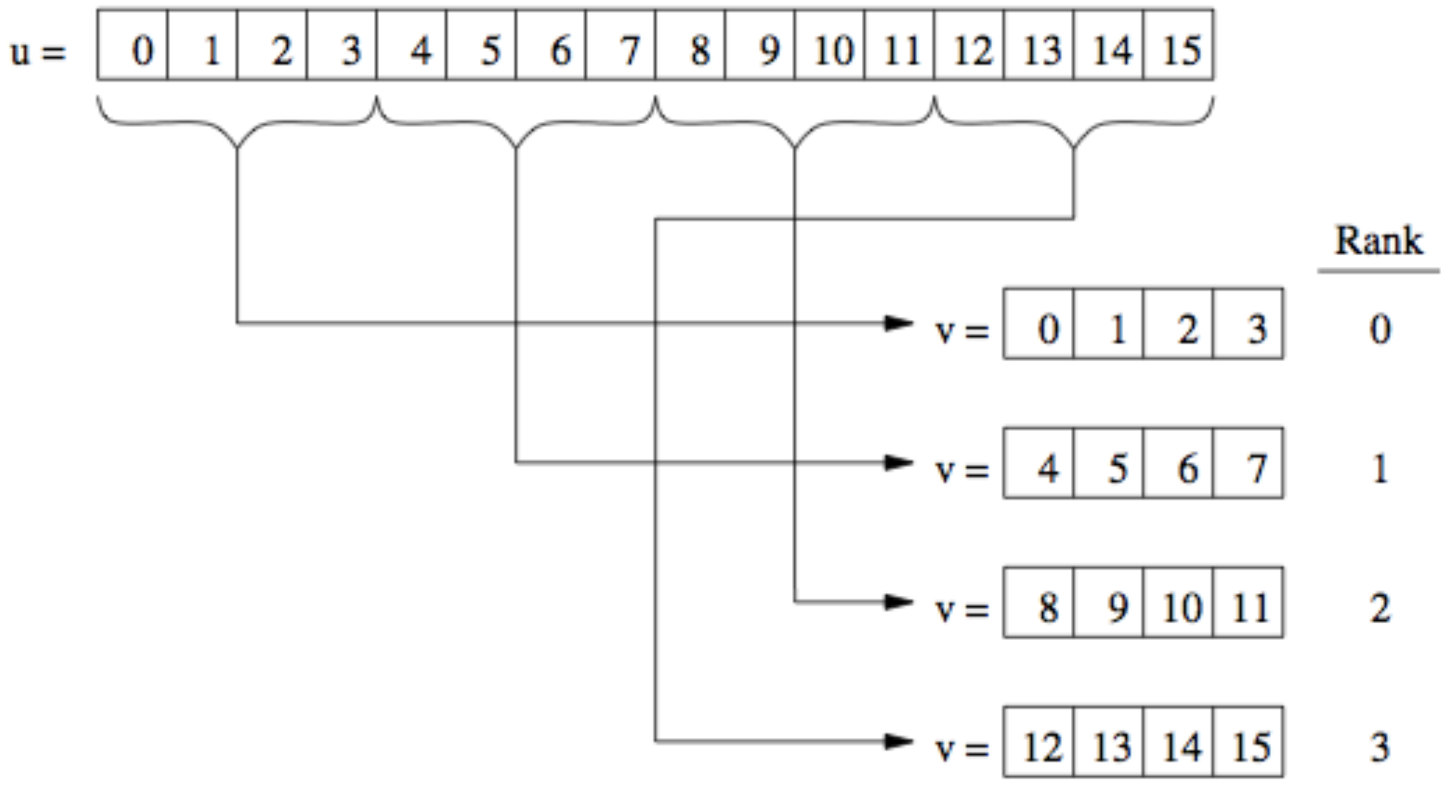


MPI Scatter

- MPI_SCATTER (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)
- 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 root argument is the rank of the root process.

MPI Scatter

```
MPI_Scatter(u, 4, MPI_INT, v, 4, MPI_INT, 0, MPI_COMM_WORLD);
```



MPI Scatter

The calling sequence for `MPI_Scatter()` is

```
int MPI_Scatter(
    void *sendbuf,            // pointer to send buffer
    int sendcount,            // items to send per process
    MPI_Datatype sendtype,    // type of send buffer data
    void *recvbuf,            // pointer to receive buffer
    int recvcount,            // number of items to receive
    MPI_Datatype recvtype,    // type of receive buffer data
    int root,                 // rank of sending process
    MPI_Comm comm)            // MPI communicator to use
```

MPI Scatter

- The contents of sendbuf on process with rank root are distributed in groups of sendcount elements of type sendtype to all processes.
- sendbuf should contain at least sendcount× number_of_processes data items of type sendtype where number_of_processes is the number of processes returned by MPI_Comm_size().
- Each process, including the root, receives recvcount elements of type recvtype into recvbuf.

MPI Scatter

MPI_Scatter

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

int main( int argc, char **argv )
{
    int isend[3], irecv;
    int rank, size, i;

    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );

    if(rank == 0){
      for(i=0;i<size;i++) isend[i] = i+1;
    }

    MPI_Scatter(&isend, 1, MPI_INT, &irecv, 1, MPI_INT, 0, MPI_COMM_WORLD);

    printf("rank = %d: irecv = %d
", rank,irecv);

    MPI_Finalize();
    return 0;
}
```

MPI Gather

```
int MPI_Gather(
  void *sendbuf,
  int sendcnt,
  MPI_Datatype sendtype,
  void *recvbuf,
  int recvcnt,
  MPI_Datatype recvtype,
  int root,
  MPI_Comm comm
);
```

MPI Gather

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

MPI Gather

- Sendbuf
  - starting address of send buffer (choice)
- Sendcount
  - number of elements in send buffer (integer)
- Sendtype
  - -data type of send buffer elements (handle)
- recvcount
  - number of elements for any single receive (integer, significant only at root)

MPI Gather

- Recvtype
  - data type of recv buffer elements (significant only at root)
- root - rank of receiving process (integer)
- comm - communicator (handle)

Scatter Gather Example

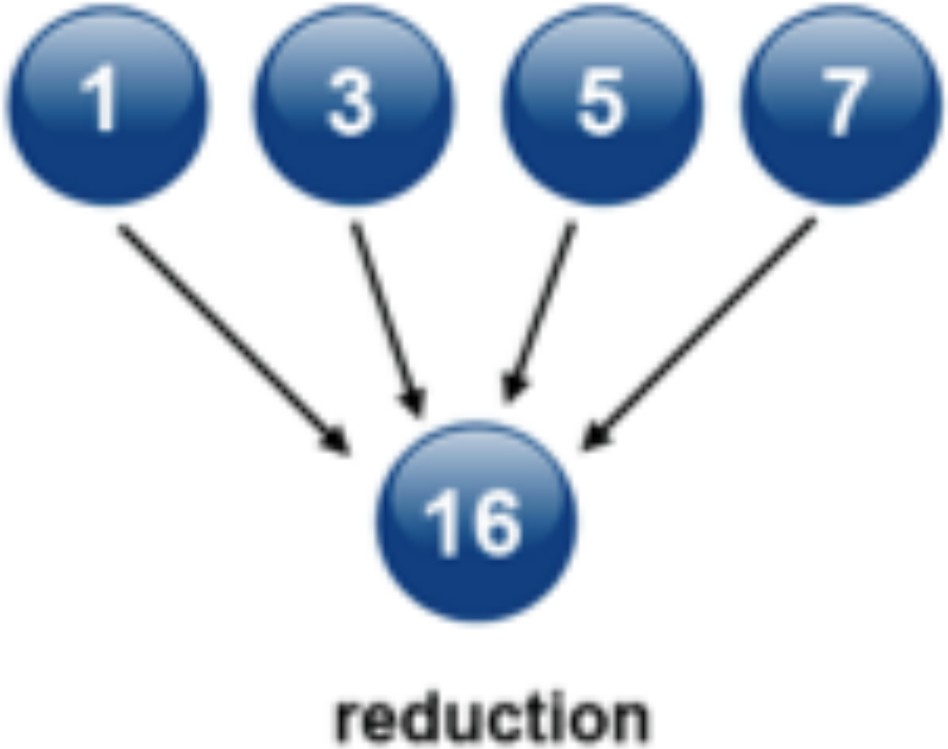
```
// Create a buffer that will hold a subset of the random numbers
float *sub_rand_nums = malloc(sizeof(float) * elements_per_proc);

// Scatter the random numbers to all processes
MPI_Scatter(rand_nums, elements_per_proc, MPI_FLOAT, sub_rand_nums,
            elements_per_proc, MPI_FLOAT, 0, MPI_COMM_WORLD);

// Compute the average of your subset
float sub_avg = compute_avg(sub_rand_nums, elements_per_proc);
// Gather all partial averages down to the root process
float *sub_avgs = NULL;
if (world_rank == 0) {
  sub_avgs = malloc(sizeof(float) * world_size);
}
MPI_Gather(&sub_avg, 1, MPI_FLOAT, sub_avgs, 1, MPI_FLOAT, 0,
           MPI_COMM_WORLD);
```

Ref: http://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/

MPI Reduce

- MPI_REDUCE combines data from all processes in communicator and returns it to one process.
- In many numerical algorithms, SEND/RECEIVE can be replaced by BCAST/REDUCE, improving both simplicity and efficiency.

MPI Reduce



MPI Reduce Example

```
if (world_rank == 0) {
  rand_nums = create_rand_nums(elements_per_proc * world_size);
}

// Create a buffer that will hold a subset of the random numbers
float *sub_rand_nums = malloc(sizeof(float) * elements_per_proc);

// Scatter the random numbers to all processes
MPI_Scatter(rand_nums, elements_per_proc, MPI_FLOAT, sub_rand_nums,
            elements_per_proc, MPI_FLOAT, 0, MPI_COMM_WORLD);

// Compute the average of your subset
float sub_avg = compute_avg(sub_rand_nums, elements_per_proc);
// Gather all partial averages down to the root process
float *sub_avgs = NULL;
if (world_rank == 0) {
  sub_avgs = malloc(sizeof(float) * world_size);
}
MPI_Gather(&sub_avg, 1, MPI_FLOAT, sub_avgs, 1, MPI_FLOAT, 0,
           MPI_COMM_WORLD);

// Compute the total average of all numbers.
if (world_rank == 0) {
  float avg = compute_avg(sub_avgs, world_size);
}
```

MPI All to All

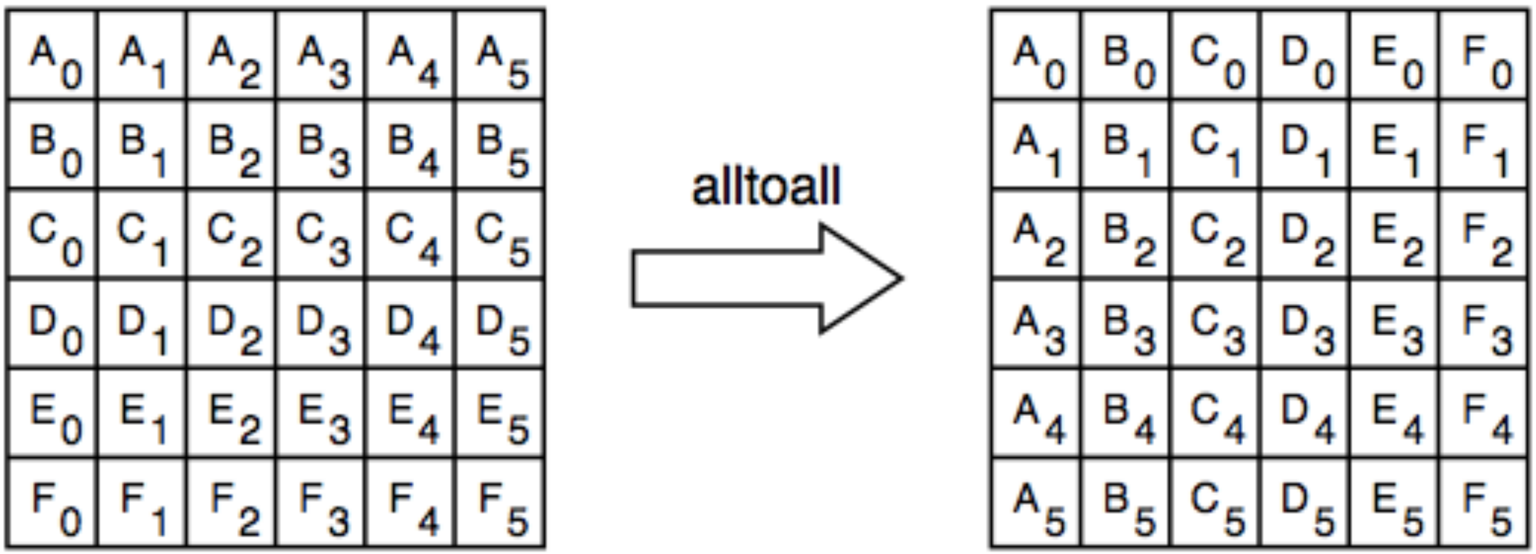
Suppose there are four processes including the root, each with arrays as shown below on the left. After the all-to-all operation

```
MPI_Alltoall(u, 2, MPI_INT, v, 2, MPI_INT, MPI_COMM_WORLD);
```

the data will be distributed as shown below on the right:

array u								Rank	array v							
10	11	12	13	14	15	16	17	0	10	11	20	21	30	31	40	41
20	21	22	23	24	25	26	27	1	12	13	22	23	32	33	42	43
30	31	32	33	34	35	36	37	2	14	15	24	25	34	35	44	45
40	41	42	43	44	45	46	47	3	16	17	26	27	36	37	46	47

MPI All to All



MPI All to All

The calling sequence for `MPI_Alltoall()` is

```
int MPI_Scatter(
  void *sendbuf,             // pointer to send buffer
  int sendcount,             // items to send per process
  MPI_Datatype sendtype,     // type of send buffer data
  void *recvbuf,             // pointer to receive buffer
  int recvcount,             // number of items to receive
  MPI_Datatype recvtype,     // type of receive buffer data
  MPI_Comm comm)             // MPI communicator to use
```

- The contents of `sendbuf` on each process are distributed in groups of `sendcount` elements of type `sendtype` to all processes.
- `sendbuf` should contain at least `sendcount` × `number_of_processes` data items of type `sendtype`.
- Each process receives `recvcount` elements of type `recvtype` into `recvbuf`.
- All arguments are significant on all processes.

Thank you

LEARN • SERVE • EXCEL
RAJAGIRI