MPI – Message Passing Interface

CHAPTER 4

Outline

1. MPI Overview
2. Basic Structure of a MPI program
3. Messages and Point-to-Point Communication
4. Non-blocking Communication
5. **Collective Communication**
6. Derived Data Types

Collective Communication

- Communication involving a group of processes
- Called by all processes in a communicator
- Examples:
  - Barrier synchronization
  - Broadcast, scatter, gather.
  - Global sum, global maximum, etc.
  - Neigbour communication in a virtual grid (New in MPI-3.0)

Collective Communication

Characteristics of Collective Communication

•Collective action over a communicator

•The collective communications do not interfere with the communications point-to-point and vice-versa.

•All processes of the communicator must call the collective routine.

•They do not use tags

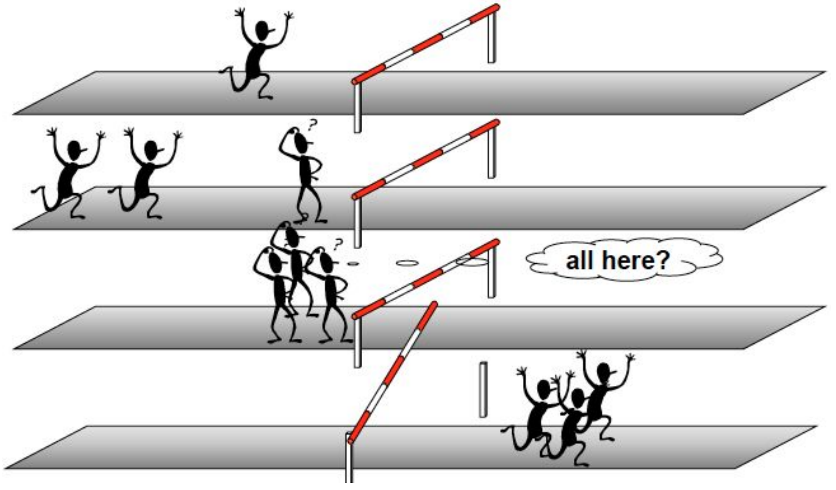
•Receive buffers must have the same size as send buffers.

Collective Communication

Barrier Synchronization

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

The synchronization by barrier blocks to all the processes in the communicator *comm* until all the processes of the group have done the call MPI_Barrier, producing the synchronization.



Collective Communication

Broadcast

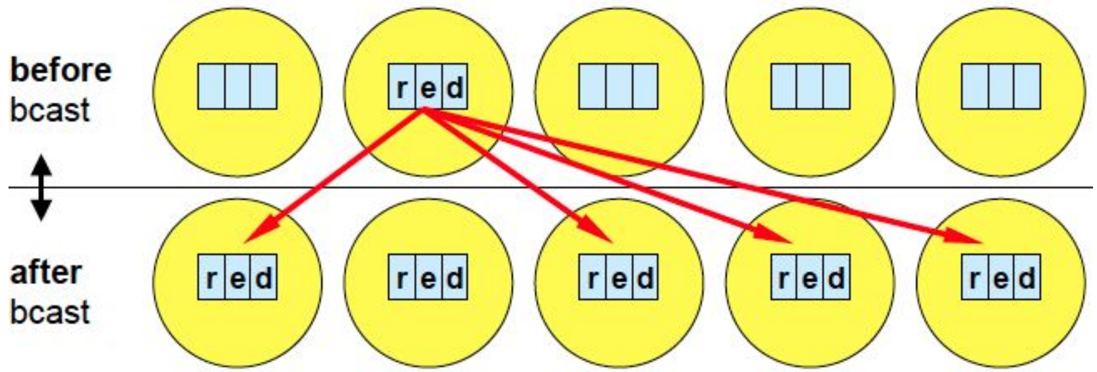
hpc-course/Examples/Broadcast.c

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

•The function **MPI_BCast** sends a message from the process root (with rank root) to the rest of processes of the group (itself included).

•All the processes that call the broadcast, have to invoke to this function with the same parameters ***comm*** and ***root***.

•When it goes back of the function, the content of the buffer of the process root has been copied in the buffer of all the processes.



root=1 (identical for all processes)

Collective Communication

Activity5: Vector Broadcast

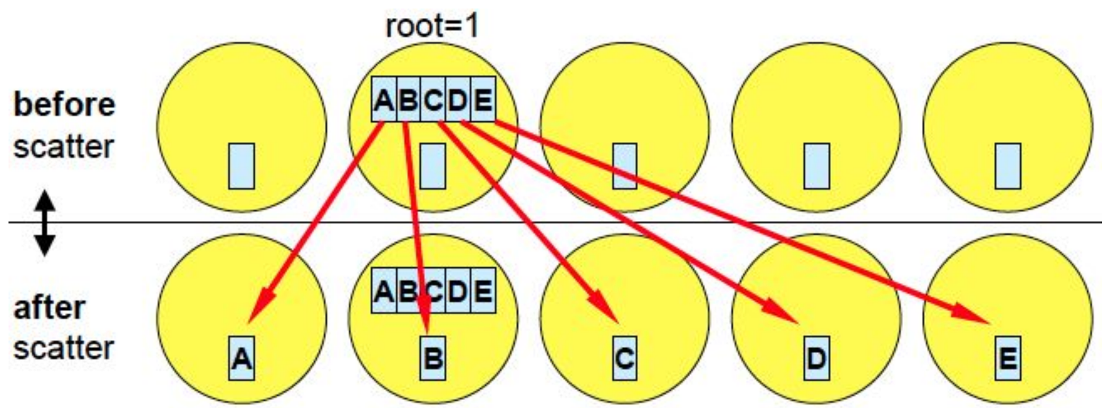
- Write a program to send an array to all the processors
  - process with rank 3 initializes a random vector with length=20
  - Each processor receives the vector and print the vector to the screen with the processes rank

Collective Communication

Scatter

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

- Communication **one-to-all**. The *root* process sends *sendcount* elements of information to each one of the processes in *comm* (itself included).
- **MPI_SCATTERV** Extends the previous functionality, varying the number of data sent to each process (sendcounts is a vector)

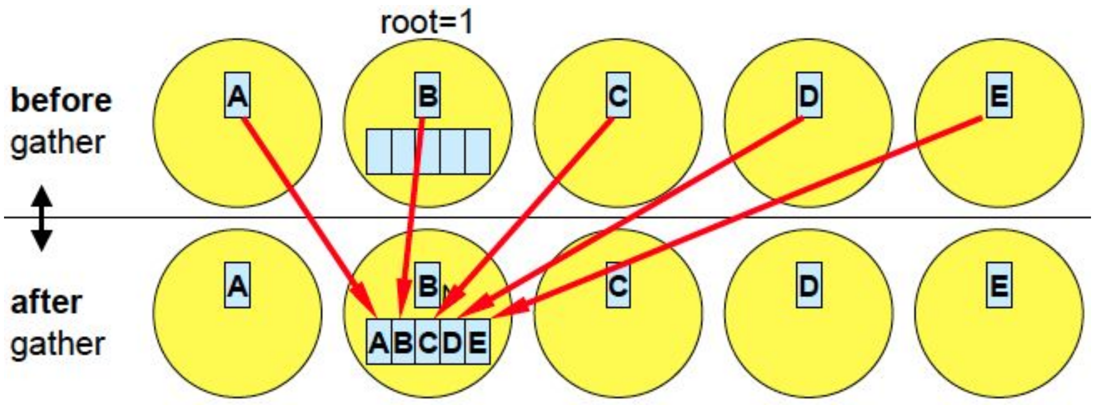


Collective Communication

Gather

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

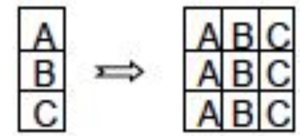
- Communication **all-to-one**. Each process (root included) sends the contents of its buffer *sendbuf*, to the *root* process. The *root* process receives all the messages and stores in *recvbuf* ordered by process rank.
- **MPI_GATHERV** extends the previous functionality, varying the number of data received from each process (recvcounts is a vector)



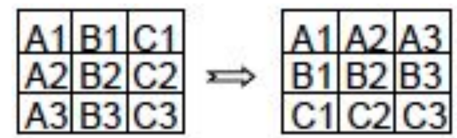
Collective Communication

Other Colletive Operations

- MPI_Allgather
  - similar to MPI_Gather, but all processes receive the result vector



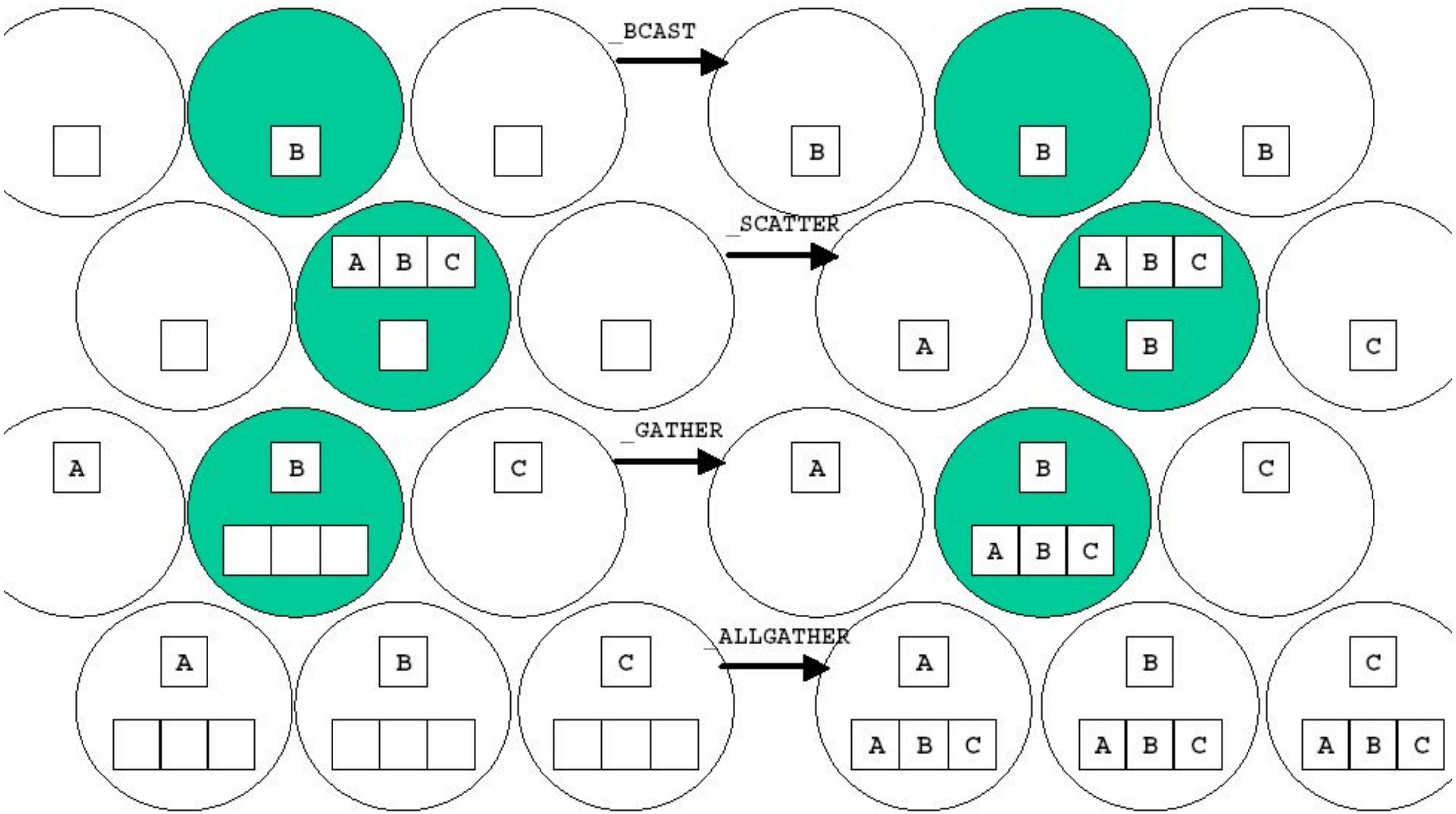
- MPI_Alltoall
  - each process sends messages to all processes
  - the most expensive routine



- MPI_******v (Gatherv, Scatterv, Allgaterv, Alltoallv, Alltoallw)
  - Each message has a different count and displacement
  - Array of counts and array of displacements (Alltoallw: also arrays of types)
  - Interface does not scale to thousands of MPI processes!
  - **Recommendation:** One should try to use data structures with same communication size on all ranks.

Collective Communication

Colletive Operations Summary



Collective Communication

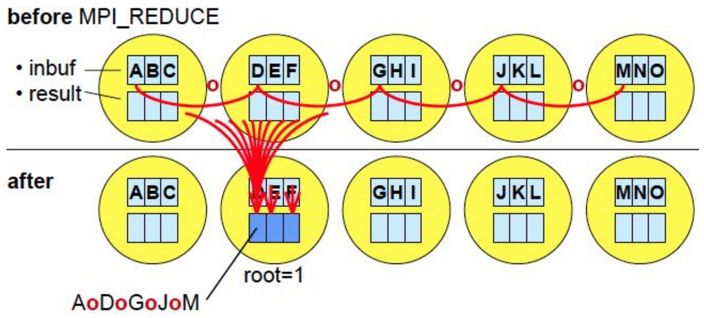
Global Reduction Operations

- Collect values from all processes and reduce to a scalar.
- only for associative operation

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

where:

- **sendbuf**: source address
- **recvbuf**: result address
- **count**: number of elements to send / receive
- **datatype**: type of each element
- **op**: reduction operation
- **root**: process receiving and reducing
- **comm**: communicator



Collective Communication

Predefined Reduction Operation Handles

Operation handle	Function
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND

Operation handle	Function
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical XOR
MPI_BXOR	Bitwise XOR
MPI_MAXLOC	Maximum and location
MPI_MINLOC	Minimum and location

Besides it exists a mechanism so that the user can build his own functions for the “Reduces”.

Collective Communication

Global Reduction Operations examples

Example 1 – one value reduction

Example 2 – vector reduction

Example 3 – MPI_MAXLOC, MPI_MINLOC reduction

Basic Structure of an MPI program

Activity6: MPI Reduce for Vectors

Compute

$$\|A\|_1 = \max_{1 \le j \le n} \sum_{i=1}^{m} |a_{ij}| \text{ for an } m \times n \text{ matrix } A.$$

Suppose there are m processes and the ith process has a vector arow(1:n) containing the ith row of A.

Use MPI_REDUCE to sum:

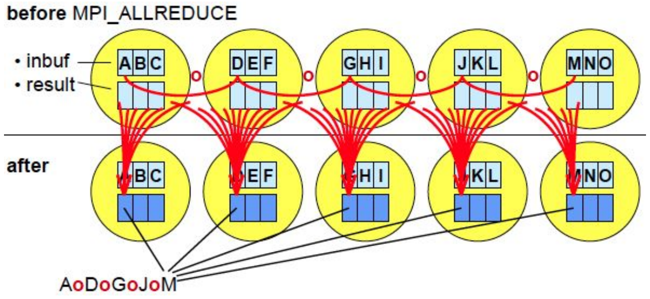
the first element of each row vector into colsum(1)

second element of each row vector into colsum(2), etc.

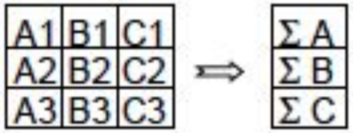
Collective Communication

Variants of Reduction Operations

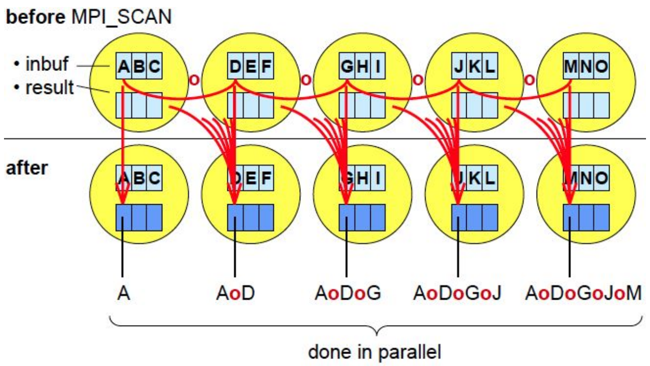
- **MPI_ALLREDUCE**
  - no root
  - Returns the result in all processes



- **MPI_REDUCE_SCATTER**
  - Result vector of the reduction operations is scattered to the processes into the real result buffers



- **MPI_SCAN**
  - Prefix reduction
  - Result at process with rank i:=reduction of inbuf-values **from rank 0 to rank i**
- **MPI_EXSCAN**
  - Result at process with rank i:=reduction of inbuf-values from **rank 0 to rank i-1**



Basic Structure of an MPI program

Activity7: MPI Reduce for Vectors

Normalize the vector *x*:

$$x \,/\, \|x\|_\infty$$

$$\|x\|_\infty = \max_i |x_i|$$

Suppose there are m processes and the ***ith*** process receives the ***istart*** and ***iend*** positions to process.

1.Use MPI_REDUCE to receive $\|x\|_\infty$
and MPI_BCAST to send again to all the processes
Send each part of normalized x back to Process 0.

2.Use MPI_ALLREDUCE to combine and broadcast to all processes
Normalize part of x on each process
Send each part of normalized x back to Process 0.

3.Capture de wall_clock and analyze the differences

Collective Communication

Activity 8: Global SCAN

•Write a program so that each process computes a partial sum.

•Write in a way that each process prints out its partial result in the correct order:

rank=0 sum=0
rank=1 sum=1
rank=2 sum=3
rank=3 sum=6
rank=4 sum=10

This can be done, e.g., by sending a token (empty message) from process 0 to process 1, from 1 to 2, and so on

•Compute the sum in each slice with the global reduction.

Outline

1. MPI Overview
2. Basic Structure of a MPI program
3. Messages and Point-to-Point Communication
4. Non-blocking Communication
5. Collective Communication
6. **Derived Data Types**

Derived Data Types

- Up to here, all communication functions have involved contiguous buffers containing to sequences of elements of the same type. This is too constraining:
  - one often wants to pass messages that contain values with different Datatypes.
  - one often wants to send non-contiguous data
- Different solutions:
  - Send successive messages with different basic types. This is expensive and proclivity to errors
  - Sub-vectors or non-consecutive positions on memory
  - Pack distinct types in one message and unpack them after the reception. It requires additional operations of copy in both sides of the communication.
  - Create your derived MPI Datatype to send all the data

Derived Data Types

How to decide which method to use?

- If data are the same type or in an array: **use count**
- If data are from different types but only sent few times: **use packing/unpacking**
- If data are from different types and must be sent repeatedly: **use derived Datatype**

Derived Data Types

Packed Data

The function **MPI_PACK** allows to store non-contiguous data of different types in contiguous memory positions.

```
int MPI_Pack(void* inbuf, int incount, MPI_Datatype datatype, void
*outbuf, int outsize, int *position, MPI_Comm comm)
```

- ***inbuf*** the variable to pack into the buffer ***outbuf***
- ***incount*** size of ***inbuf***
- ***datatype*** a basic datatype
- ***outbuf*** buffer with packed data
- ***outsize*** size of the packed data
- ***position*** the current position in the ***outbuf***. It is increased automatically with the packed data
- ***comm*** the communicator

Any of the MPI communication subroutines can be used for sending packed data using the handle MPI_PACKED

Derived Data Types

Unpack Data

The function **MPI_Unpack** allows to unpack the packed data in the source.

```
int MPI_Unpack(void* inbuf, int incount, int *position,
               void *outbuf, int outcount, MPI_*Datatype datatype,
               MPI_Comm comm)
```

- ***inbuf*** buffer with packed data
- ***incount*** size of ***inbuf***
- ***position*** the position in the ***inbuf*** to start to unpack
- ***outbuf*** buffer with unpacked data
- ***outcount*** size of the unpacked data
- ***datatype*** a basic datatype for the unpacked data
- ***comm*** the communicator

Derived Data Types

User defined Data Types

Define your personal Datatype to send messages with different types of data.

```
int MPI_Type_create_struct(int count, int array_of_blocklengths[],
     const MPI_Aint array_of_displacements[],
     const MPI_Datatype array_of_types[],
    MPI_Datatype *newtype)
```

- ***array_of_blocklengths[]*** an array with the length of the different blocks of the struct
- ***array_of_displacements[]*** an array of the relative displacements of each block inside the struct
- ***array_of_types[]*** an array with the type of each block
- ***newtype*** an MPI_Datatype variable, wich will contain the defined *struct* type

Derived Data Types

- To compute displacements
  <u>array_of_displacements[i]</u> := address(block_i) - address(block_0)

```
int MPI_Get_address (void *location, MPI_Aint *address)
```

  - ***<u>Location</u>*** location in the buffer to obtain the address
  - ***<u>address</u>*** the out obtained address

- Before using a derived Datatype, it must be commited

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

- After using it, it must be freed

```
int MPI_Type_free(MPI_Datatype *datatype)
```

Derived Data Types

- To compute displacements
  array_of_displacements[i] ***:= address(block_i) - address(block_0)***

```
int MPI_Get_address (void *location, MPI_Aint *address)
```

- ***Location*** location in the buffer to obtain the address
- ***address*** the out obtained address

- Before using a derived Datatype, it must be commited

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

- After using it, it must be freed

```
int MPI_Type_free(MPI_Datatype *datatype)
```

Derived Data Types

Grouping data example

Example 1 – <u>Grouping Data with **count**</u>

Example 2 – <u>Grouping Data with **Pack**</u>

Example 3 – <u>Grouping Data with **derived Datatype**</u>

Thanks for your atention!

MPI – Introduction to the Message Passing Interface

CHAPTER 4