Message Passing Interface (MPI) - Collective Communication

Usama Bin Amir

Department of Cyber Security,
Air University, Islamabad

Collective Communications

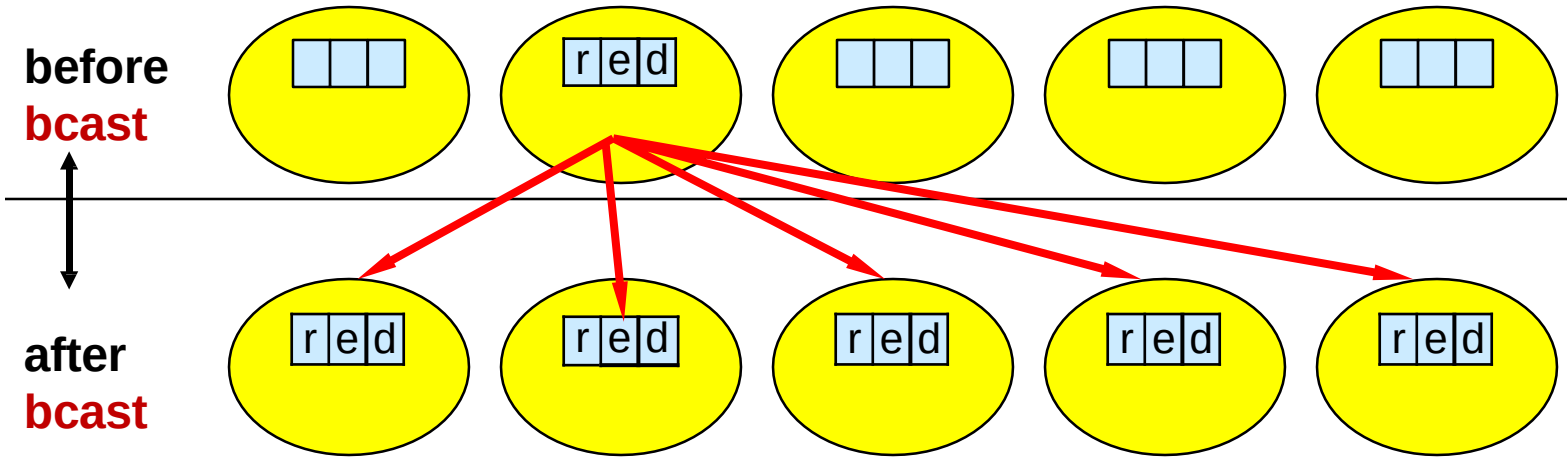
- **Processes** may **need** to **communicate** with **everyone else**

- **Three Main Classes:**
  1. **Communications**: ***Broadcast***, ***Gather***, ***Scatter***
  2. **Synchronization**: ***Barriers***
  3. **Reductions**: ***sum***, ***max***, etc.

- **Properties:**
  - Must be **executed** by **all processes** (of the communicator)
  - All **processes** in group **call same operation** at (roughly) the **same time**
  - **All collective operations** are **blocking operations**

Broadcast

- A **one-to-many** **communication**



e.g., root=1

- **rank** of the **sending process** (i.e., root process)
- must be **given identically** by **all processes**

Broadcasting with MPI_Bcast

```
int MPI_Bcast (void *buf, int count, MPI_Datatype dtype, int root,
               MPI_Comm comm)
```

IN buf:	*Adress of send/receive buffer*
IN count:	*Number of elements*
IN dtype:	*Data type*
IN root:	*Sender*
IN comm:	*Communicator*

- The **contents** of the **send buffer** is **copied** from a **sender** (i.e., root process) **to all other processes**, (***including itself***)

- The **type signature** (***number of elements***, ***data type***) on **any process must be same** (**as on** the **root process**)

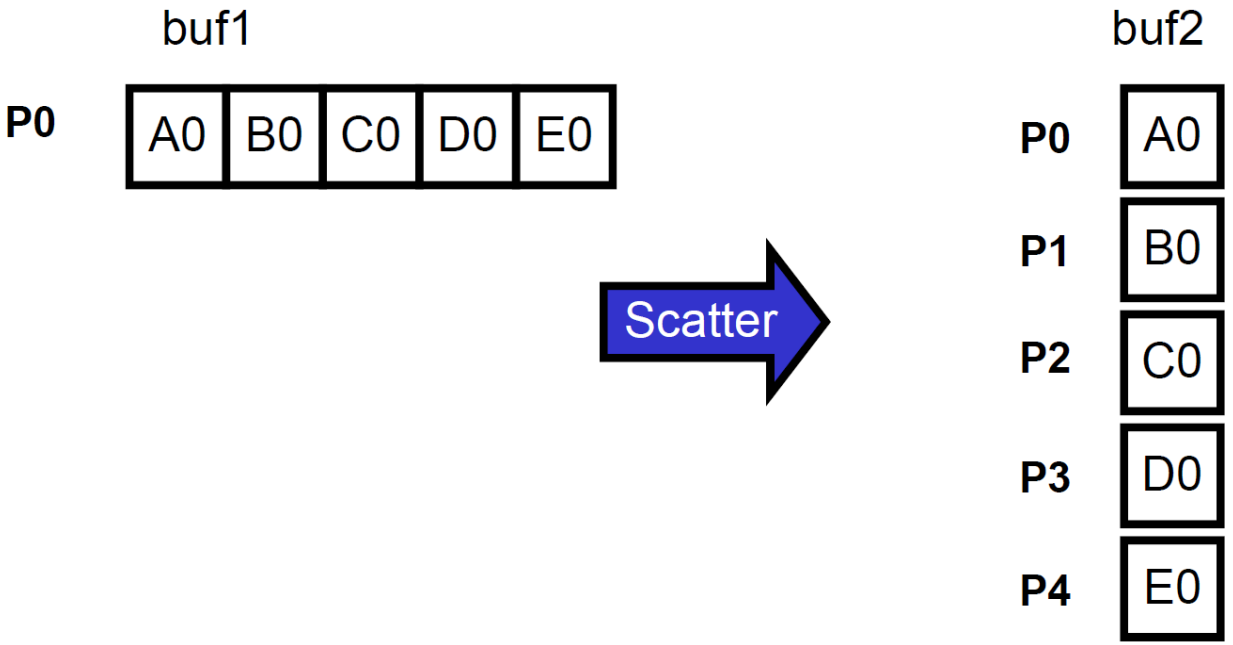
Broadcasting with MPI_Bcast

Demo: BroadCast.c

```
#include <stdio.h>
#include <string.h>
#include "mpi.h"
int main(int argc, char **argv)
{
  char message[20];
  int  i, rank, size;
  MPI_Status status;
  int root = 0;

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

  if (rank == root)
  {
    strcpy(message, "Hello, world");
  }
  MPI_Bcast(message, 13, MPI_CHAR, root, MPI_COMM_WORLD);
  printf( "Message from process %d : %s\n", rank, message);

  MPI_Finalize();
}
```

MPI_Scatter

- **MPI_Scatter is a collective routine that is similar to MPI_Bcast**
- **It sends *chunks of an array* to different processes**




```
int MPI_Scatter (buf1, 1, MPI_INT,
                 buf2, 1, MPI_INT,
                 P0, MPI_COMM_WORLD)
```

MPI_Scatter

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

IN sendbuf:	*Send buffer*
IN sendcount:	*Number of elements sent to each process*
IN sendtype:	*Data type*
OUT recvbuf:	*Receive buffer*
IN recvcount:	*Number of elements to be received by a process*
IN recvtype:	*Data type*
IN root:	*Sender*
IN comm:	*Communicator*

The **root sends a part** of its **send buffer** to **each process**

- **Process *k* receives *sendcount*** elements **starting with *sendbuf+k*sendcount***

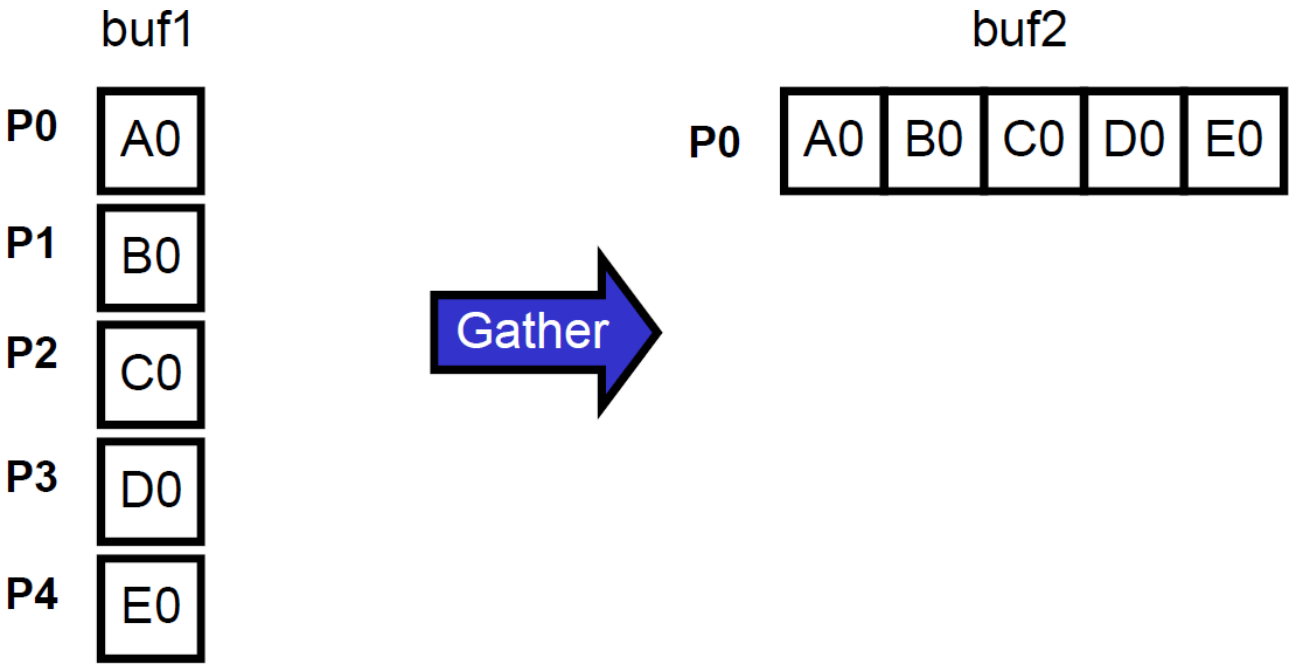
MPI_Scatter - Example

Demo:
Scatter.c

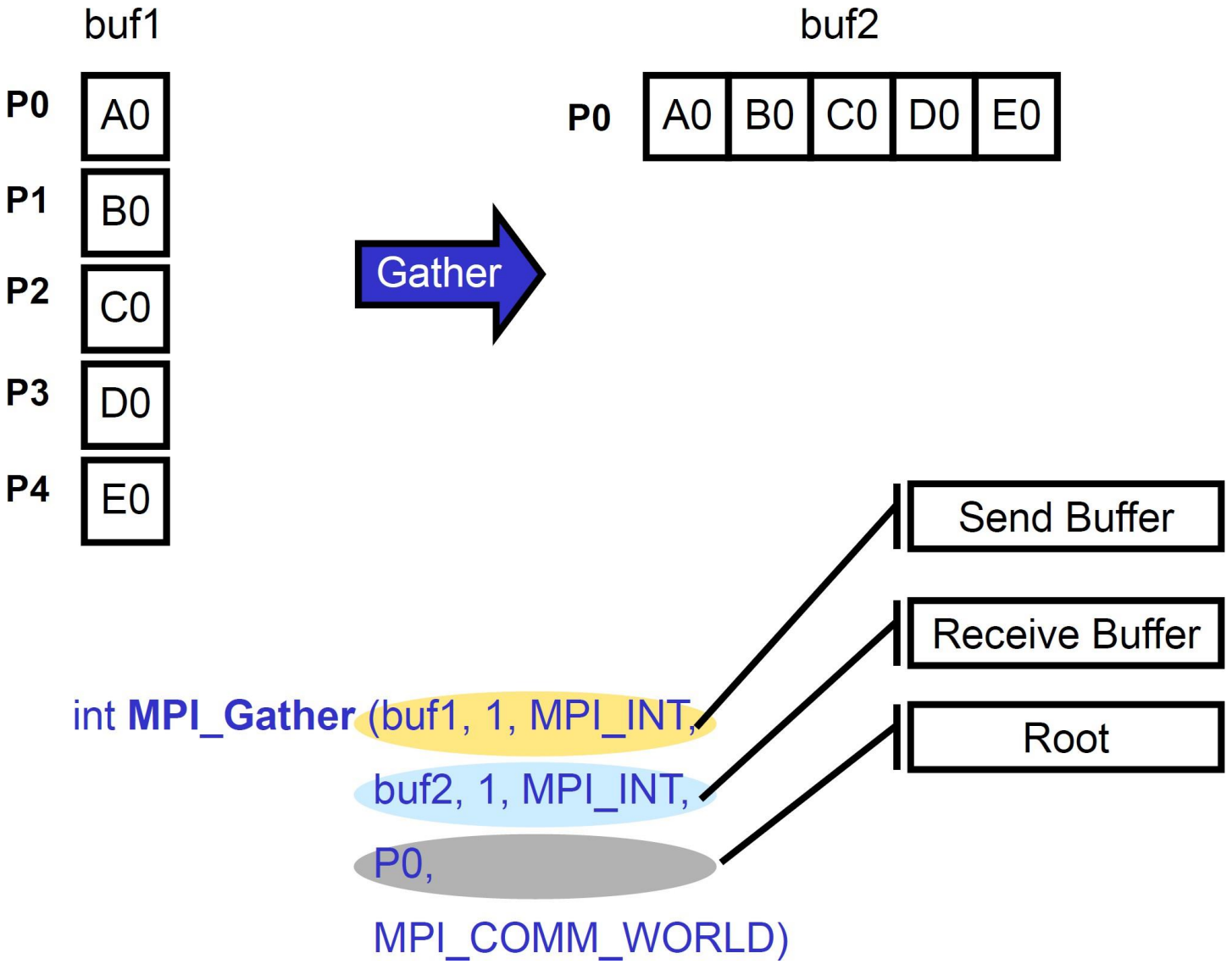
```
#include <mpi.h>

int main( int argc, char* argv[] ){
    int i, rank, nproc, *isend, irecv;
    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &nproc );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );

    if(rank == 0) {
        isend = (int*) malloc(sizeof(int) * nproc);
        for(i=0; i<nproc;)
            isend[i] = ++i;
    }
    MPI_Scatter(isend, 1, MPI_INT, irecv, 1, MPI_INT, 0,
                MPI_COMM_WORLD);

   printf("RANK: %d -> irecv = %d\n", rank, irecv);
   MPI_Finalize();
}
```

MPI_Gather

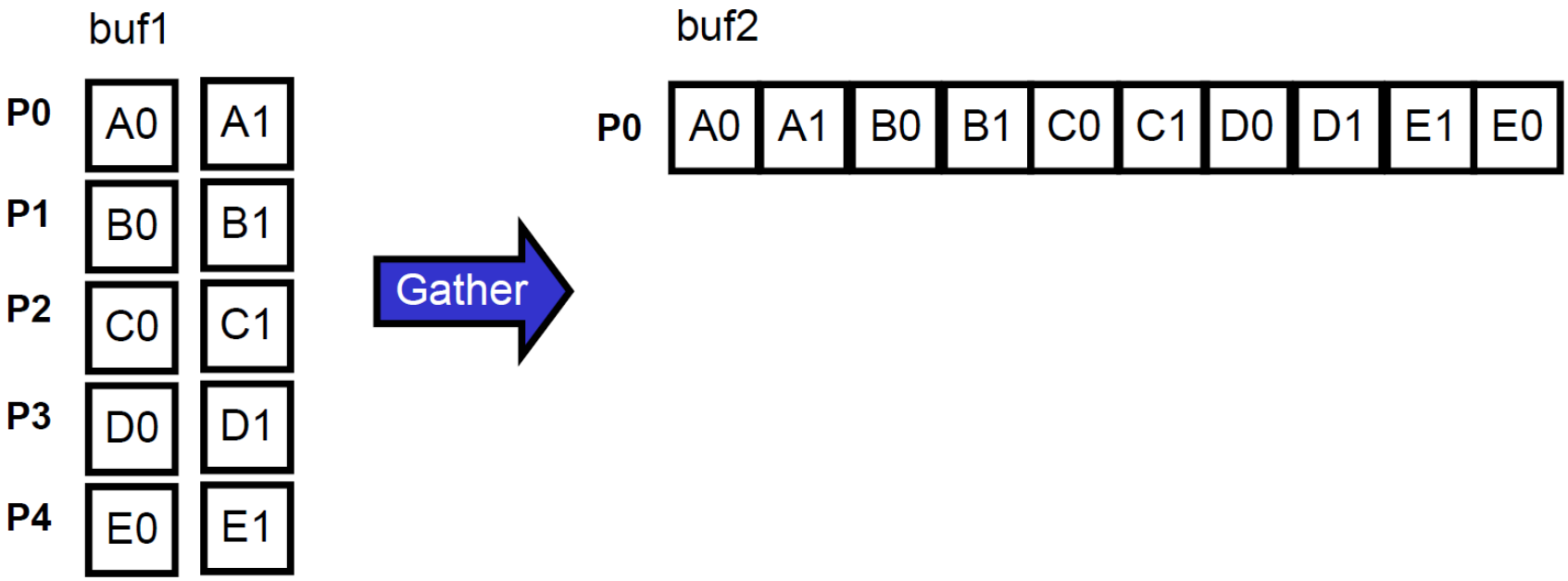
- **MPI_Gather** is the **inverse** of **MPI_Scatter**
- It **takes elements** from **many processes** and **gathers** them to one **single process**



MPI_Gather


buf1
P0
A0
P1
B0
P2
C0
P3
D0
P4
E0
Gather
buf2
P0
A0
B0
C0
D0
E0
Send Buffer
Receive Buffer
Root
int MPI_Gather (buf1, 1, MPI_INT,
buf2, 1, MPI_INT,
P0,
MPI_COMM_WORLD)

MPI_Gather




```
int MPI_Gather (buf1, 2, MPI_INT,
                buf2, 2, MPI_INT,
                P0, MPI_COMM_WORLD)
```

MPI_Gather

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

IN sendbuf:	*Send buffer*
IN sendcount:	*Number of elements to be sent to the root*
IN sendtype:	*Data type*
OUT recvbuf:	*Receive buffer*
IN recvcount:	*Number of elements to be received from each process.*
IN recvtype:	*Data type*
IN root:	*Receiver*
IN comm:	*Communicator*

- The **root receives data** from **all processes** (from send buffers)
- It **stores** the **data** in the **receive buffer** **ordered by** the **process number** of the **senders**

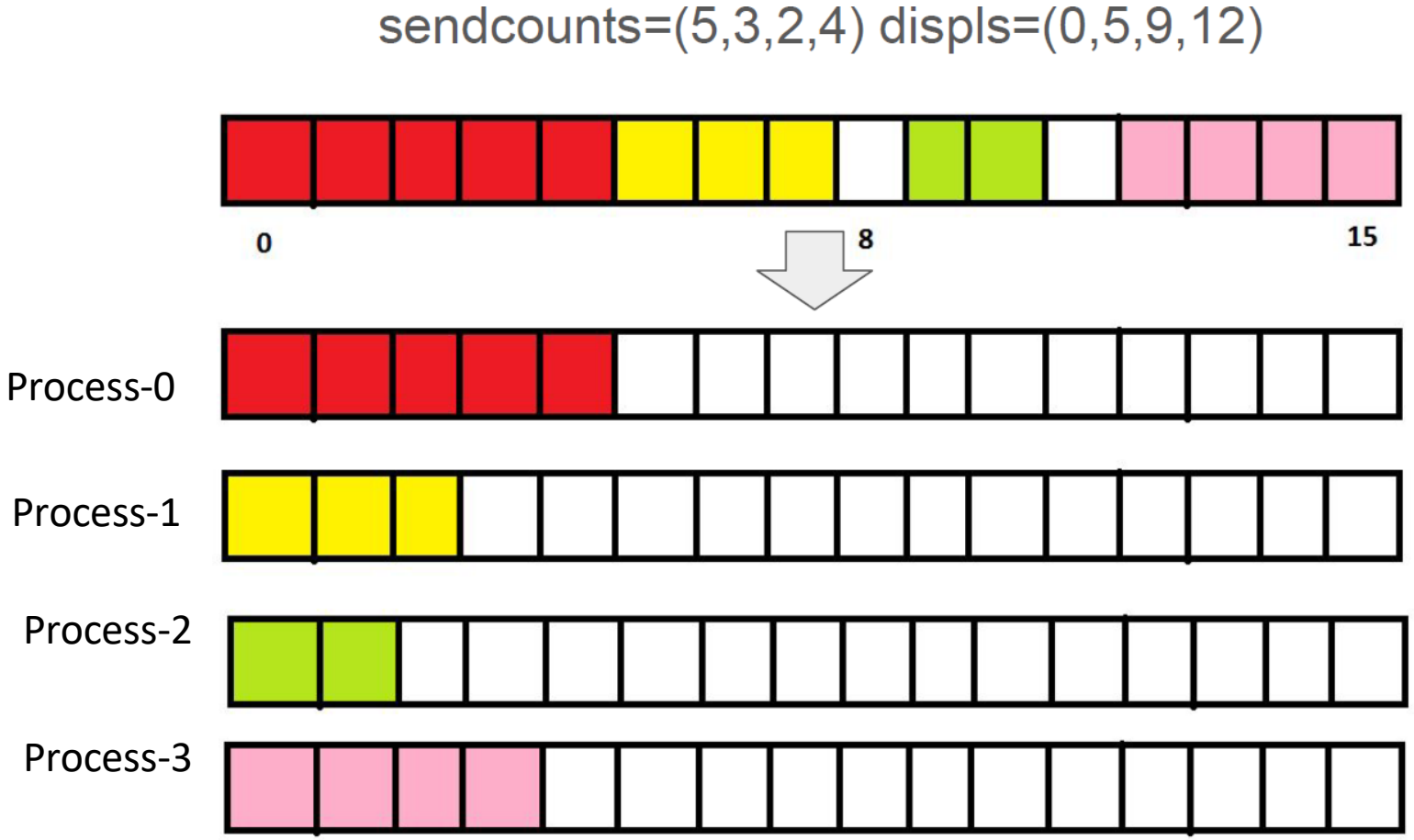
MPI_Gather - Example

Demo: Gather.c

```
int main(int argc,char*argv[])
{
        int i,rank,nproc;
        int sendBuf[3];
        int recvBuf[12]; //for 4 processes, 3 ints for each
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD,&nproc);
        MPI_Comm_rank(MPI_COMM_WORLD,&rank);

        //Each process prepare send buff data
        for (i=0;i<3;i++)
                sendBuf[i]=rank;

        //Everyone executes this, process 0 gathers from all
        MPI_Gather(sendBuf,3,MPI_INT,recvBuf,3,MPI_INT,0,MPI_COMM_WORLD);

        if(rank==0) {
                for(i=0;i<12;i++) {
                        printf("\nProcess number %d send data : %d",i/3, recvBuf[i]);
                }
        }

        MPI_Finalize();
        printf("\nBye from %d\n",rank);
return 0;
}
```

MPI_Scatterv

- **MPI_Scatterv is a collective routine that is similar to MPI_Scatter**
- **It sends variable *chunks of an array* to different processes**



Credits: https://www.cineca.it/

MPI_Scatterv

Demo: ScatterV.c

```
int MPI_Scatterv(const void *sendbuf, const int sendcounts[],
    const int displs[], MPI_Datatype sendtype,
    void *recvbuf, int recvcount, MPI_Datatype recvtype,
    int root, MPI_Comm comm)
```

sendbuf: **address** of **send buffer** (significant only at root)

sendcounts: **integer array** (of **length group size**) specifying the **number** of **elements** to **send** to **each process**

displs: **integer array** (of length group size). Entry ***i*** **specifies** the **displacement** (relative to **sendbuf** from which to **take** the **outgoing data** to **process *i***

sendtype: **data-type** of **send buffer** elements

recvcount: **number** of **elements** in **receive buffer** (integer)

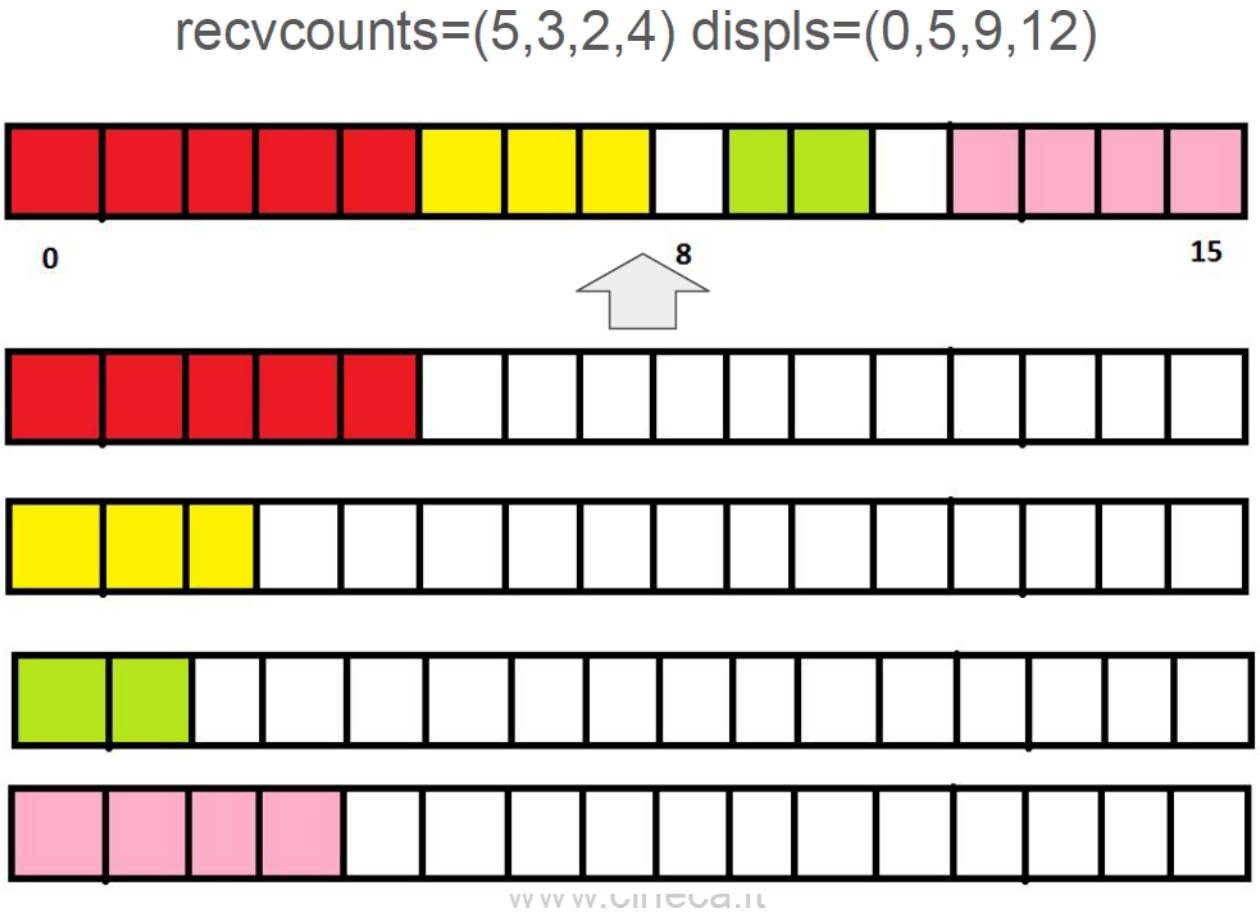
recvtype: **data-type** of **receive buffer** elements

root: **rank** of **sending process** (integer)

comm: **communicator** (handle)

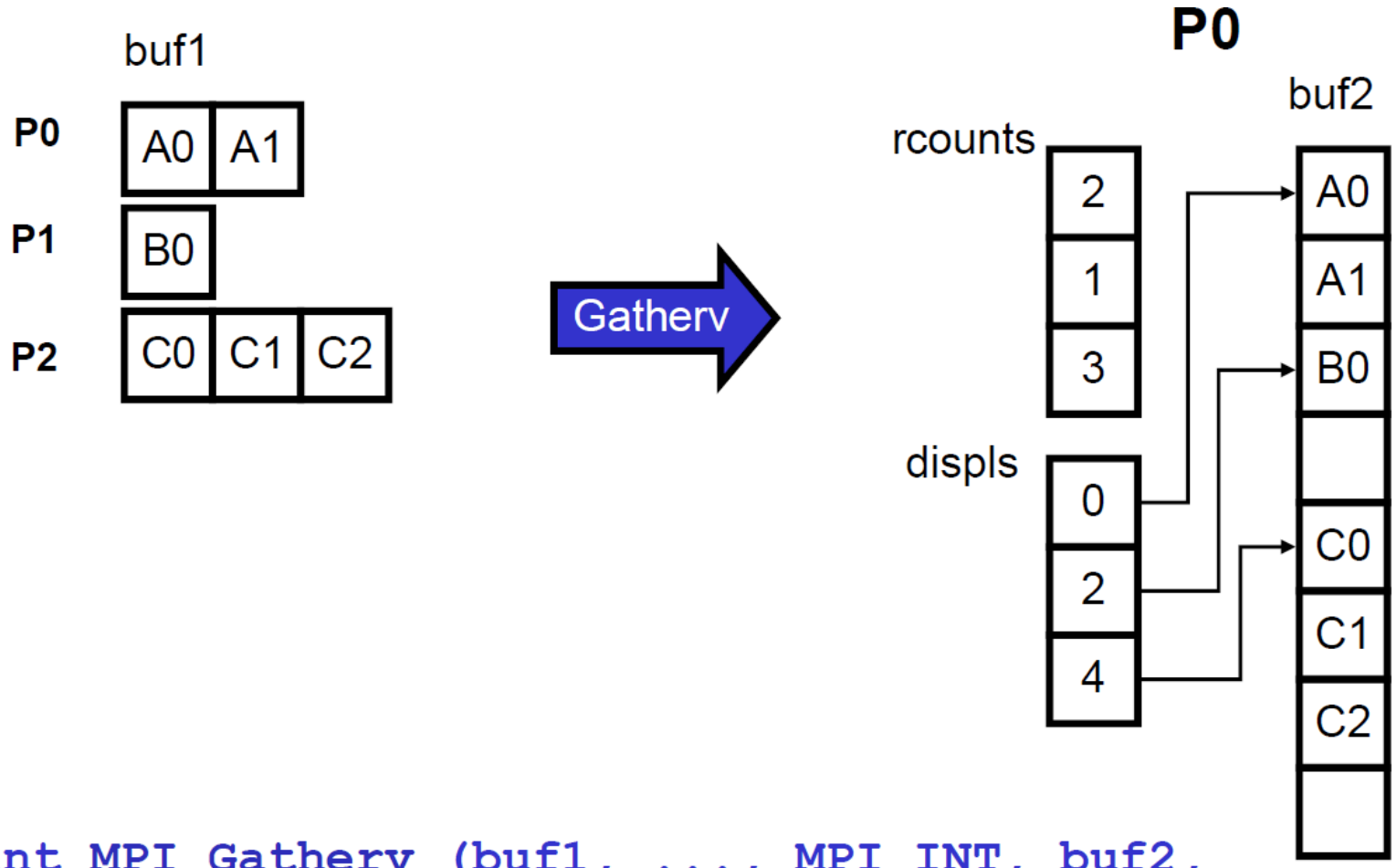
MPI_Gatherv

- Different number of elements can be received by the root process
- Individual messages are stored according to *displs* in the receive buffer



Credits: https://www.cineca.it/

MPI_Gatherv




```
int MPI_Gatherv (buf1, ..., MPI_INT, buf2,
                 rcounts, displs, MPI_INT, P0,
                 MPI_COMM_WORLD)
```

MPI_Gatherv

Demo: GatherV.c

```
int MPI_Gatherv(const void *sendbuf, int sendcount,
    MPI_Datatype sendtype, void *recvbuf, const int recvcounts[],
    const int displs[], MPI_Datatype recvtype,
    int root, MPI_Comm comm)
```

sendbuf: **address** of **send buffer**

sendcounts: **number** of **elements** in **send buffer** (integer)

sendtype: **data-type** of **send buffer** elements

recvbuf: **address** of the **receive buff (significant at root)**

recvcounts: **integer array** (of **length group size**) containing the **number** of **elements** that are to be **received from each process (on root)**

displs: **integer array** (of length group size). Entry ***i*** **specifies** the **displacement relative** to **recvbuf** at which to place data from ***process i*** (**significant** only at **root**)

recvtype: **data-type** of **receive buffer** elements (handle)

root: **rank** of receiving process (**root)**

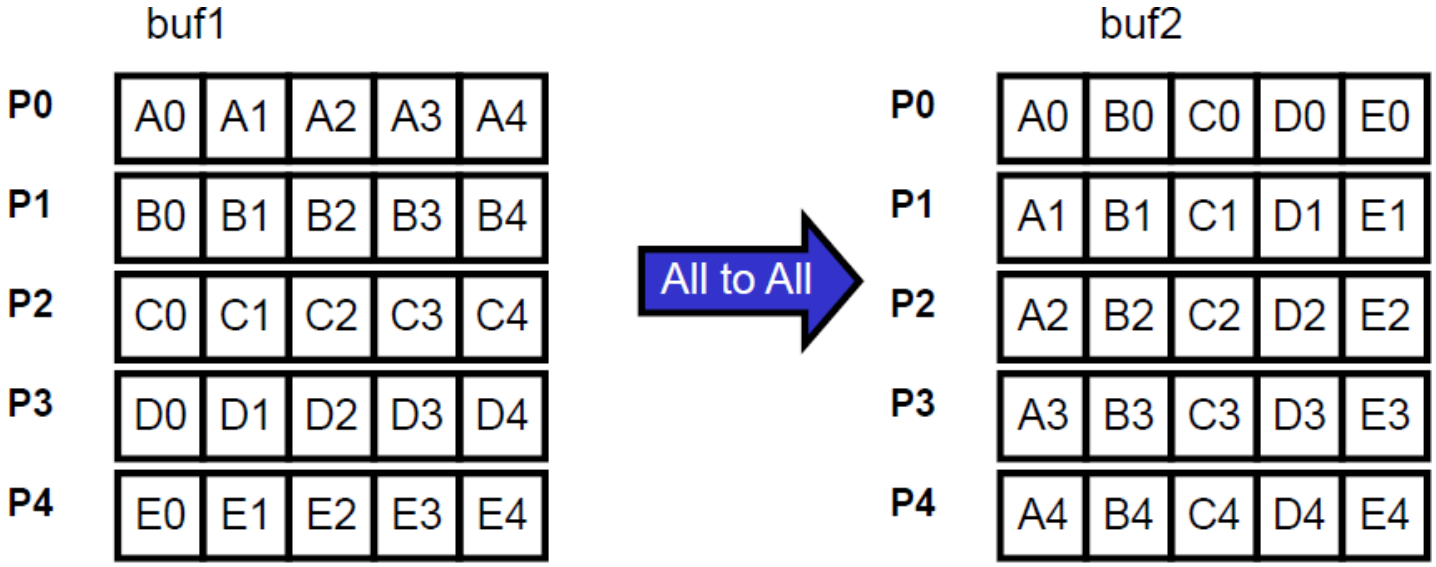
comm: **communicator** (handle)

Home Tasks

- **MPI_Allgather**
  - **Similar** to **MPI_Gather**, but the **result** is **available** to all **processes**
- **MPI_Allgatherv**
  - **Similar** to **MPI_Gatherv**, but the **result** is **available** to all **processes**
- **MPI_Alltoall**
  - **Similar** to **MPI_Allgather**, each process performs a **scater** followed by **gather** process
- **MPI_Alltoallv**
  - **Similar** to **MPI_Alltoall**, but **messages to different processes** can have **different length**

MPI_Alltoall

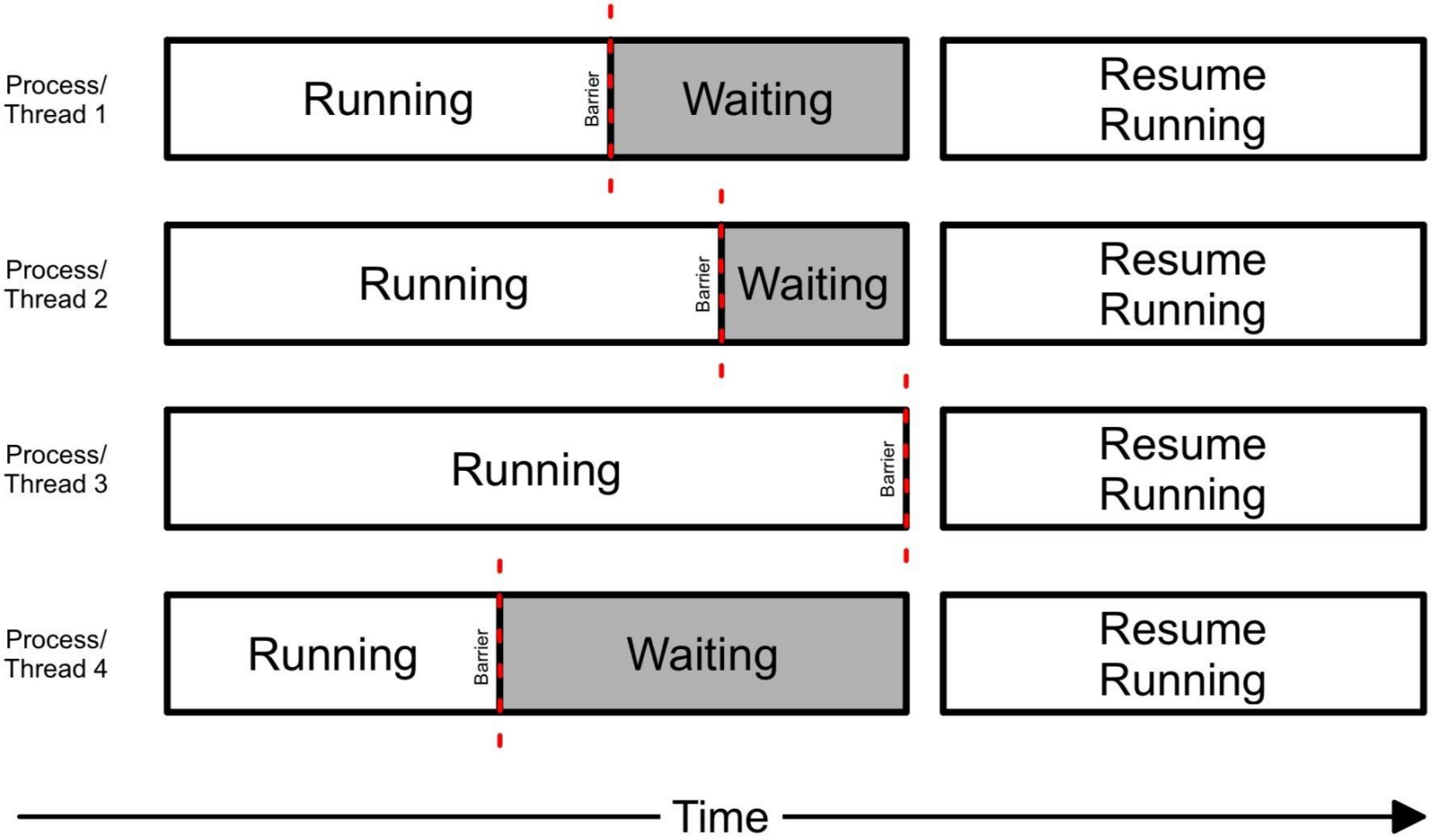
MPI_Alltoall makes a **redistribution** of **contents** such that **each process know** the **buffer of all others**. It is a **way to implement** the **matrix data transposition**




```
int MPI_Alltoall (buf1, 1, MPI_INT,
                  buf2, 1, MPI_INT,
                  MPI_COMM_WORLD)
```

Synchronization

Barrier Synchronization



Credits: https://medium.com/@jaydesai36/barrier-synchronization-in-threads-3c56f947047

MPI_BARRIER

Demo:
Barrier.c

int MPI_Barrier (MPI_Comm comm)

IN comm: *Communicator*

It **synchronizes** **ALL Processes** (by **blocking Processes**) in **communicator** **until all processes** have **called MPI_Barrier**.

Reductions

Reductions

The **communicated data** of the **processes** are **combined** via a **specified operation**, e.g. **'+'**

Two different variants:

- **Result** is **only available** at the **root process**
- **Result** is **available** at **all processes**

Input values (at each process):

- **Scalar variable:** **operation combines all values** of the processes
- **Array:** The **elements** of the **arrays** are **combined** in an **element-wise fashion**. The **result is an array**.

MPI_Reduce

```
int MPI_Reduce (void* sbuf, void* rbuf, int count, MPI_Datatype
                dtype,MPI_Op op, int root, MPI_Comm comm)
```

IN sbuf:	*Send buffer*
OUT rbuf:	*Receive buffer*
IN count:	*Number of elements in the Send buffer*
IN dtype:	*Data type*
IN op:	*Operation*
IN root:	*Root process*
IN comm:	*Communicator*

- This operation **combines** the **elements** in the **send buffer** and **delivers** the **result to root**.
- **Count**, **op**, and **root** **have to be equal in all processes**

MPI_Reduce - Example

```
#include <mpi.h>

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

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

    isend = rank + 1;
    MPI_Reduce(&isend, &irecv, 1, MPI_INT, MPI_SUM,0
                        MPI_COMM_WORLD);
    if(rank == 0)
        printf("irecv = %d\n", irecv);
    MPI_Finalize();
}
```

Demo: Reduction.c

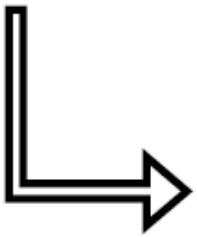
Scalar reduction

```
s=0
for (i=1; i<n; i++)
  s=s+a[i]
```

```
s=0
for (i=0; i<local_n; i++){
  s=s+a[i]
}
MPI_Reduce(&s, &s1, 1,
           MPI_INT, MPI_SUM, P0,
           MPI_COMM_WORLD)
if(rank==P0)
  s=s1
```

Array reduction

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    b[i]=b[i]+a[i][j]
```



```
for (i=0; i<n; i++)
 for (j=0;j<local_n; j++)
  b[i]=b[i]+a[i][j]

  MPI_Reduce(b, b1, n,
             MPI_INT, MPI_SUM, P0,
             MPI_COMM_WORLD)
```

Reduction Operations

Reduction Operations:

Predefined operation handle	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 of the maximum
MPI_MINLOC	Minimum and location of the minimum

Data types:

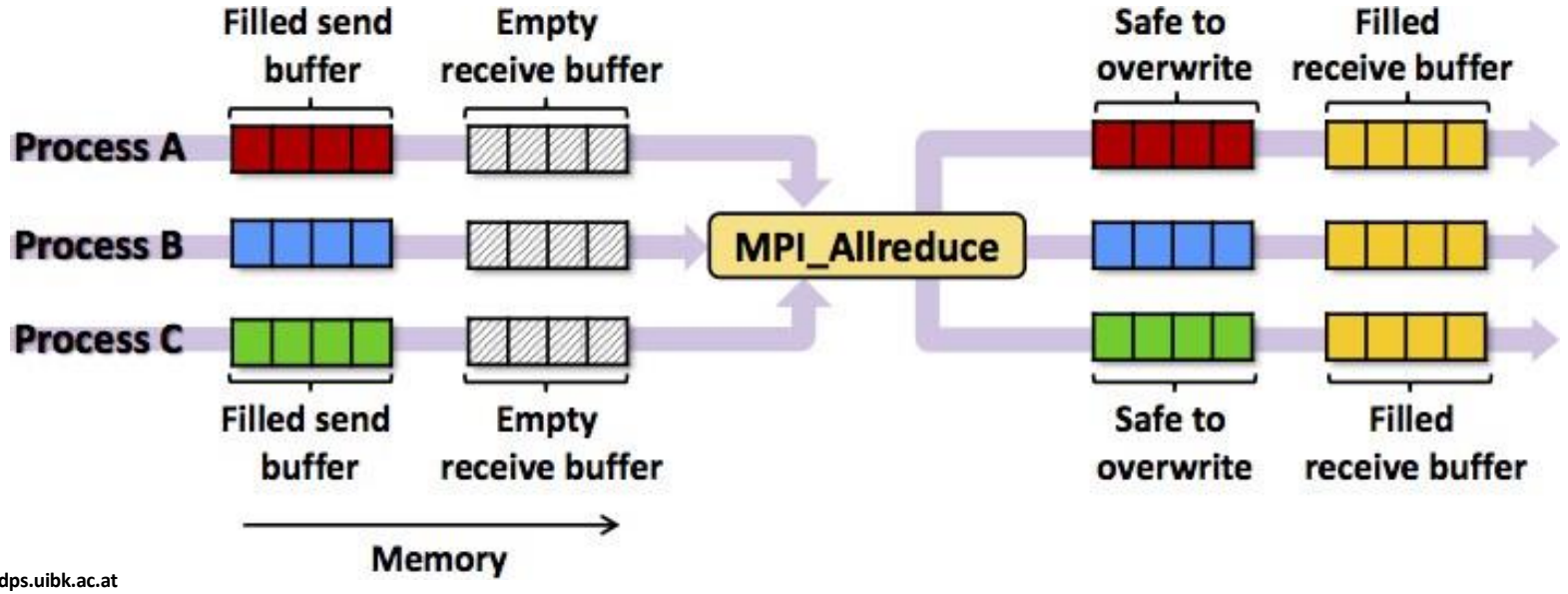
– **Operations** are **defined** for **appropriate data types**

MPI_Allreduce

```
int MPI_Allreduce (void* sbuf, void* rbuf, int count, MPI_Datatype dtype,
                   MPI_Op op, MPI_Comm comm)
```

IN sbuf:	*Send buffer*
OUT rbuf:	*Receive buffer*
IN count:	*Number of elements*
IN dtype:	*Data type*
IN op:	*Operation*
IN comm:	*Communicator*

Similar to **MPI_Reduce, returns** the **result** value to **all processes**



Credits: https://dps.uibk.ac.at

Any Questions