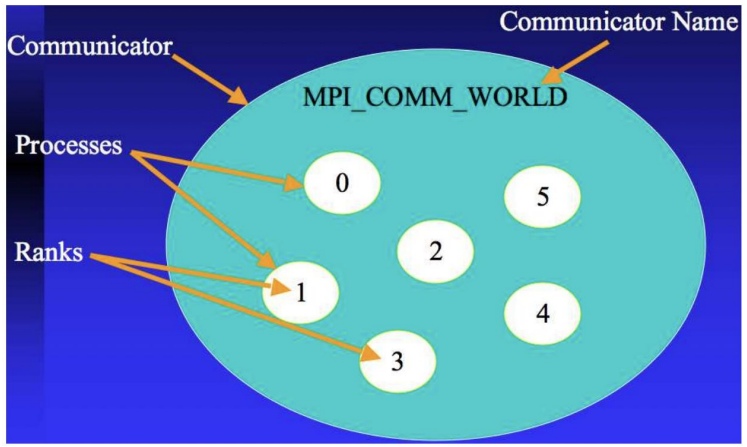
DISTRIBUTED AND CLOUD COMPUTING

LAB 2: MPI COMMUNICATION MODELS

RECAP: Message Passing Interface (MPI)

- **MPI:** a message passing **standard** to allow for distributed/parallel computation
- MPI implements an interface for ***parallel process communication***
  - Abstracts the low-level details of processes communication
  - Allows the programmer to focus on the problem at hand (the parallel application)!
- **MPI processes:** managed by MPI and run concurrently (at the same time)
- **MPI communicators:** group processes and assign them ranks
  - "MPI_COMM_WORLD" is the default communicator

Communicators



Boilerplate code

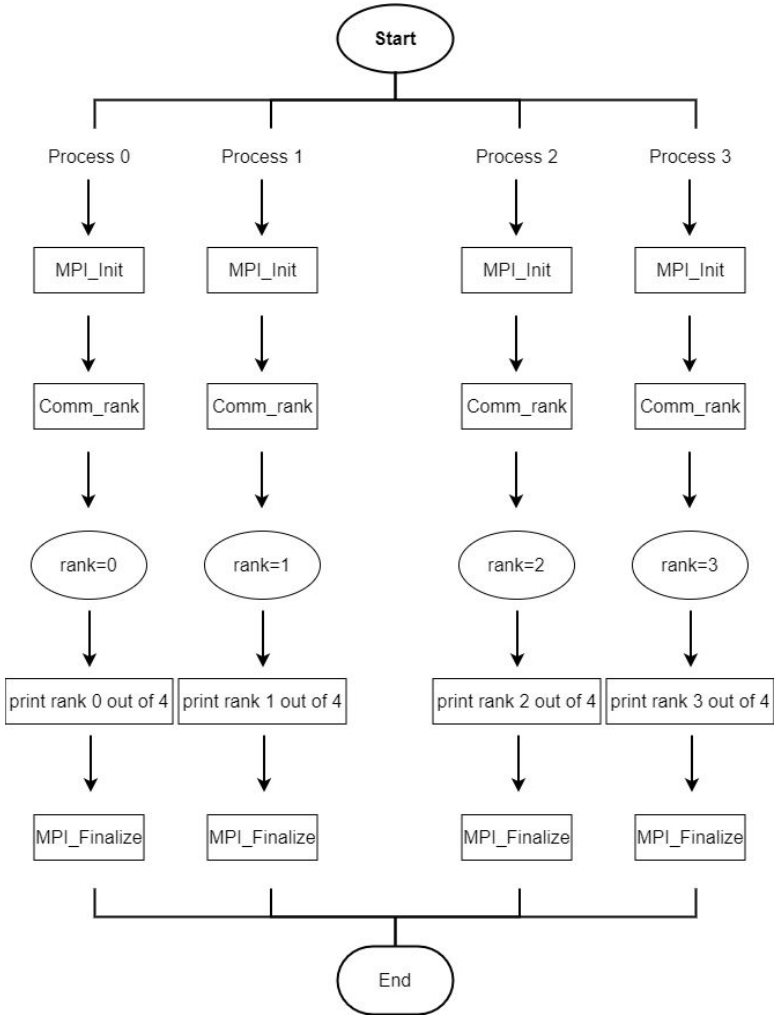
```mpi
#include <mpi.h>

int main(int argc, char* argv[])
{
    // Initialization
    MPI_Init(NULL, NULL);

    // APPLICATION LOGIC.

    // Finalize MPI.
    MPI_Finalize();
}
```

Useful functions

```mpi
// Get the number of processes
int world_size;
MPI_Comm_size(MPI_COMM_WORLD,
&world_size);

// Get the rank of the process
int world_rank;
MPI_Comm_rank(MPI_COMM_WORLD,  &world_rank);
```

RECAP: Message Passing Interface (MPI)

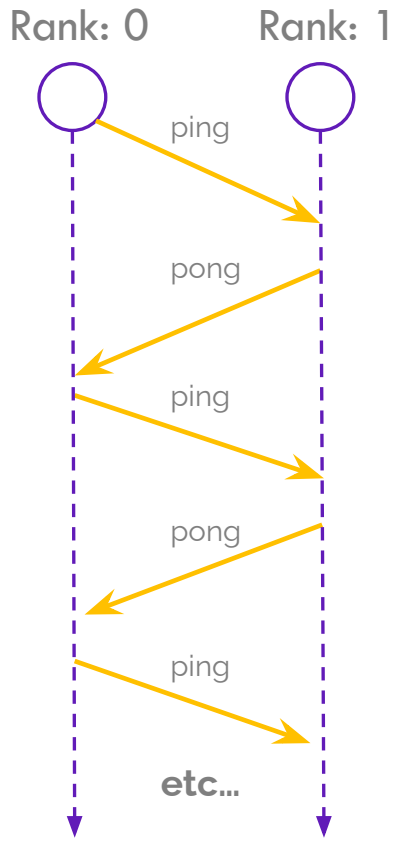
```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char* argv[]) {
    // Initialization
    MPI_Init(NULL, NULL);
    // Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    // Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    // Get the name of the processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);
    // Print off a hello world message
    printf("Hello world from processor %s, rank %d out of %d
        processors\n", processor_name, world_rank, world_size);
    // Finalize the MPI environment.
    MPI_Finalize();
}
```



https://github.com/wesleykendall/mpitutorial/blob/gh-pages/tutorials/mpi-hello-world/code/mpi_hello_world.c

RECAP: PING-PONG

- The hello-world example does not involve communication between processes
- Here we consider an example that allows two MPI processes to play ping-pong
- MPI processes send messages to each other



1. Recap of last week & MPI Datatypes

Example program: ping-pong
It implements message passing between two MPI processes

```
const int PING_PONG_LIMIT = 10;
// Some code is not shown here!!!

int ping_pong_count = 0;
int partner_rank = (world_rank + 1) % 2;
while (ping_pong_count < PING_PONG_LIMIT) {
    if (world_rank == ping_pong_count % 2) {
        // Increment the ping pong count before you send it
        ping_pong_count++;

        MPI_Send(&ping_pong_count, 1, MPI_INT, partner_rank, 0, MPI_COMM_WORLD);
        printf("%d sent and incremented ping_pong_count %d to %d\n",
               world_rank, ping_pong_count, partner_rank);
    } else {
        MPI_Recv(&ping_pong_count, 1, MPI_INT, partner_rank, 0,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("%d received ping_pong_count %d from %d\n",
               world_rank, ping_pong_count, partner_rank);
    }
}
```

Local rank

world_rank	partner_rank
1	0
0	1

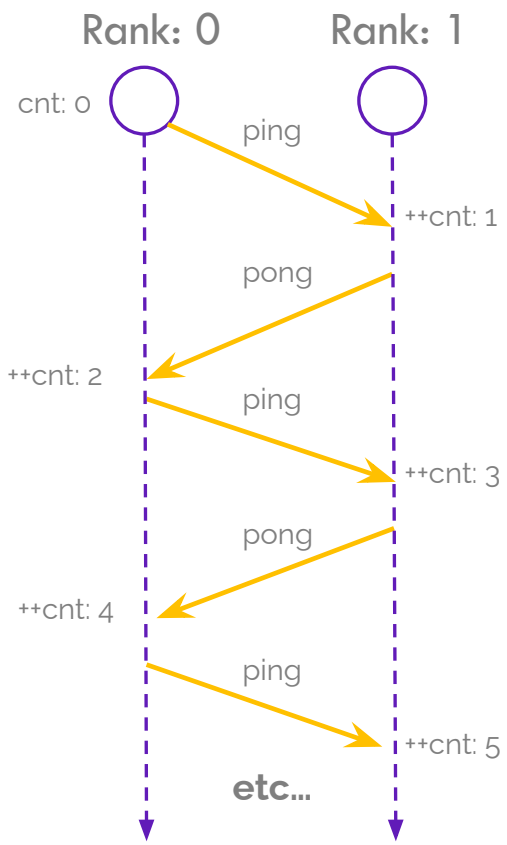
Send Buffer | SizeOf Buff | Data type | Rank of dest | Tag | Communicator

SEND

RECEIVE

https://github.com/wesleykendall/mpitutorial/tree/gh-pages/tutorials/mpi-send-and-receive/code/ping_pong.c

1. Recap of last week & MPI Datatypes




```
   ~ mpirun -np 2 ./mpi-ping-pong
0 sent and incremented ping_pong_count 1 to 1
1 received ping_pong_count 1 from 0
1 sent and incremented ping_pong_count 2 to 0
0 received ping_pong_count 2 from 1
0 sent and incremented ping_pong_count 3 to 1
0 received ping_pong_count 4 from 1
0 sent and incremented ping_pong_count 5 to 1
0 received ping_pong_count 6 from 1
0 sent and incremented ping_pong_count 7 to 1
0 received ping_pong_count 8 from 1
0 sent and incremented ping_pong_count 9 to 1
0 received ping_pong_count 10 from 1
1 received ping_pong_count 3 from 0
1 sent and incremented ping_pong_count 4 to 0
1 received ping_pong_count 5 from 0
1 sent and incremented ping_pong_count 6 to 0
1 received ping_pong_count 7 from 0
1 sent and incremented ping_pong_count 8 to 0
1 received ping_pong_count 9 from 0
1 sent and incremented ping_pong_count 10 to 0
```

2. MPI COMMUNICATION MODES

MPI COMMUNICATION MODE

- **Standard Mode, Buffered Mode, Synchronous Mode, Ready Mode**
- They have the same set of parameters
- **Differences:** The **method of sending** message and the **state of receiver**
- **Locality of mode:** whether the mode requires communicating with other processes.
  - **Local:** Completion of procedure depends only on local process
  - **Non-local:** Completion of procedure needs to execute some MPI procedure on another process

MPI COMMUNICATION MODE

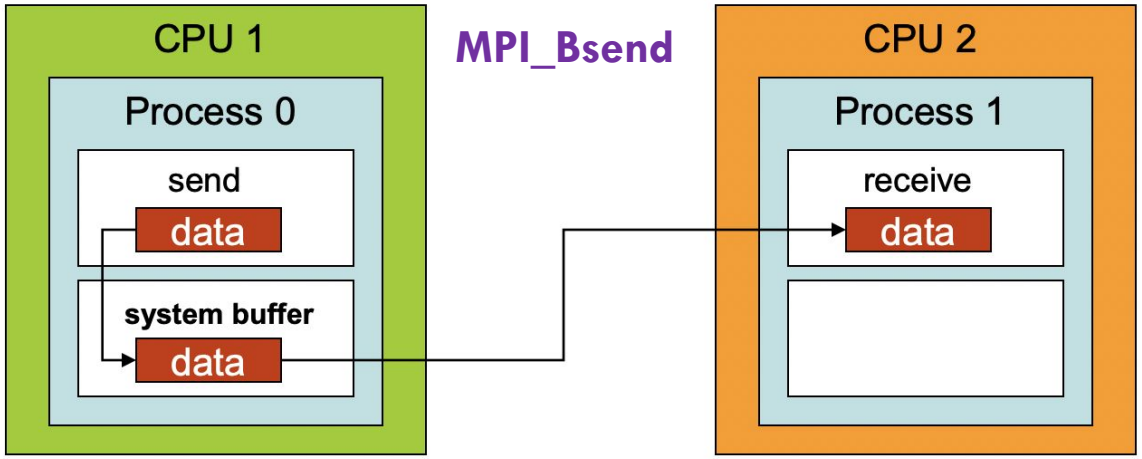
Standard mode:

- In standard mode, MPI determines if the data will be **buffered**
- **Buffered:** Copy the data into a buffer and return immediately
  - The sending will be done by MPI in background
- **Non-buffered:** Return when the data has completed sending
- Standard mode is **non-local**
  - **Non-buffered case** required processes to communicate

MPI COMMUNICATION MODE

Buffered mode:

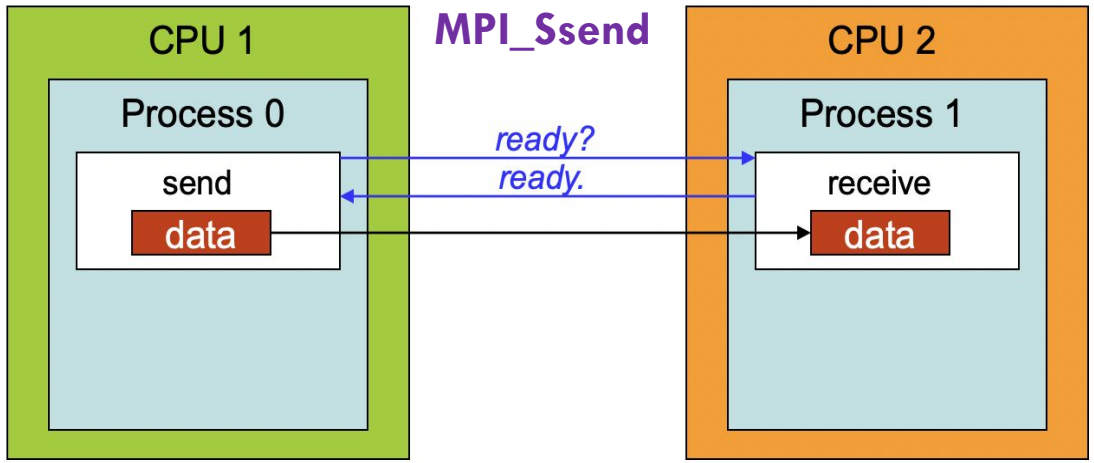
- MPI **ALWAYS** copies the data to a provided buffer and returns immediately
- The sending is done by MPI in the background
- Buffered mode is local



MPI COMMUNICATION MODE

Synchronous mode:

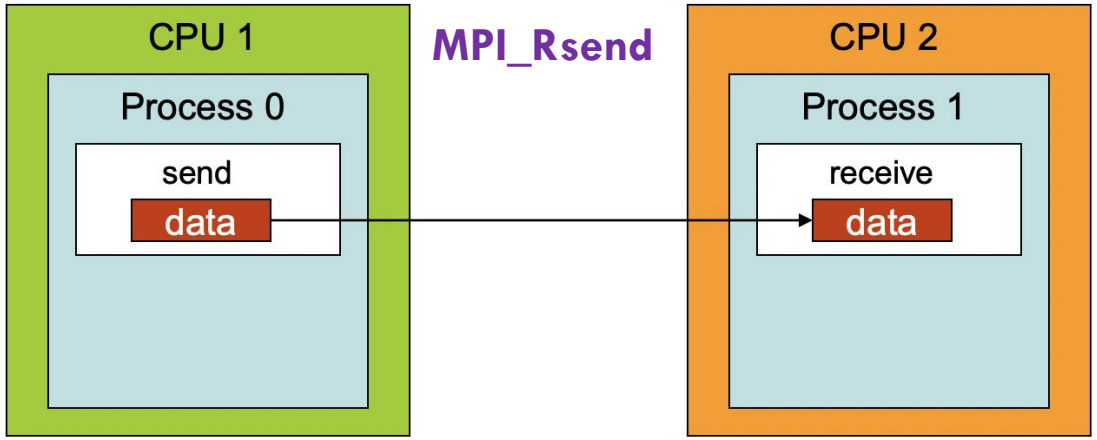
- Synchronous mode only returns when the recipient has started receiving message.
- Sending and receiving tasks must 'handshake'
- Handshake procedure ensures both processes are ready
- Synchronous mode is **non-local.**



MPI COMMUNICATION MODE

Ready mode:

- Ready mode **assumes the recipient is at ready state**
- Recipient **unable** to receive → **Error**
- Ready mode is **non-local**



MPI COMMUNICATION MODE

- This is mostly **encyclopedic** knowledge to provide intuition on MPI communication types
- In practise, we will **mostly use the standard type** communication

BLOCKING & NON-BLOCKING COMMUNICATION

Blocking communication:

- The function waits until operation is completed to return
- Suspends execution until the message buffer being sent/received is safe to use
  - **Example:** MPI_Send, MPI_Recv

Non-blocking communication:

- Function call returns immediately
- The actual operation is completed by MPI in background.
- User must ensure operation is completed before using received data
  - **Example:** MPI_Isend, MPI_Irecv

MPI: Available send & receive functions

SEND	Blocking	Nonblocking
Standard	`mpi_send`	`mpi_isend`
Ready	`mpi_rsend`	`mpi_irsend`
Synchronous	`mpi_ssend`	`mpi_issend`
Buffered	`mpi_bsend`	`mpi_ibsend`

RECEIVE	Blocking	Nonblocking
Standard	`mpi_recv`	`mpi_irecv`

MPI DATATYPES

- MPI supports various data types to be send among processes
- Complex MPI applications typically use **MPI_BYTE** to communicate with custom protocols
  - The bytes are then encoded back into their original structure based on the protocol

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_INT	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

3. COLLECTIVE COMMUNICATION

COLLECTIVE COMMUNICATION

- Collective communication refers to the communication among multiple (3 or more) processes.
- **One to many (1-N) | many to one (N-1) | many to many(N-N)**
  - In 1-N and N-1 modes: the “1” process is often called **‘root’**

Collective communication is the ‘bread and butter’ communication of distributed systems!

COLLECTIVE COMMUNICATION

Synchronization:

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

Broadcast message to all processes

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

Split data amongst all processes

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

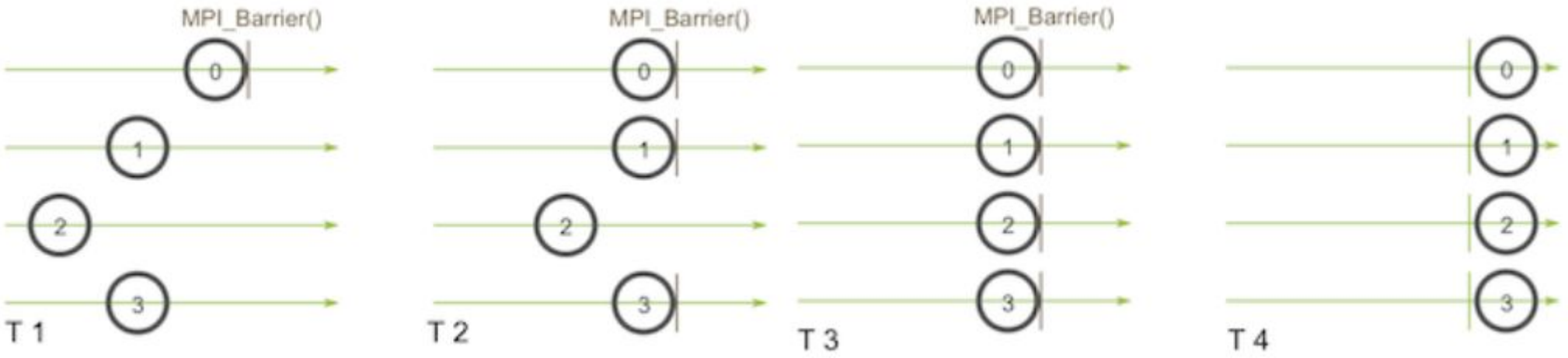
Receive messages from all processes:

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

COLLECTIVE COMMUNICATION

Barrier | `int MPI_Barrier(MPI_Comm comm)`

- **Blocks execution** of process in the given communicator **until all processes** (in that that communicator) **have reached their barrier**

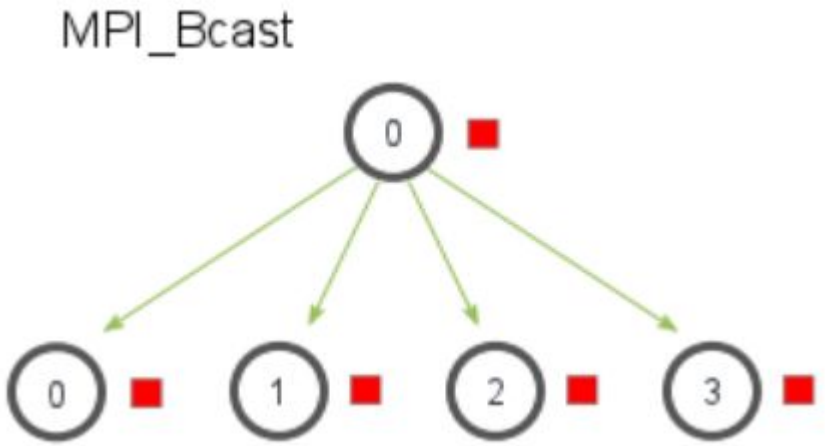


https://mpitutorial.com/tutorials/mpi-broadcast-and-collective-communication/

COLLECTIVE COMMUNICATION (1-N)

Broadcast | int MPI_Bcast(void* buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm)

- **Root** sends message **to all processes** in the communicator

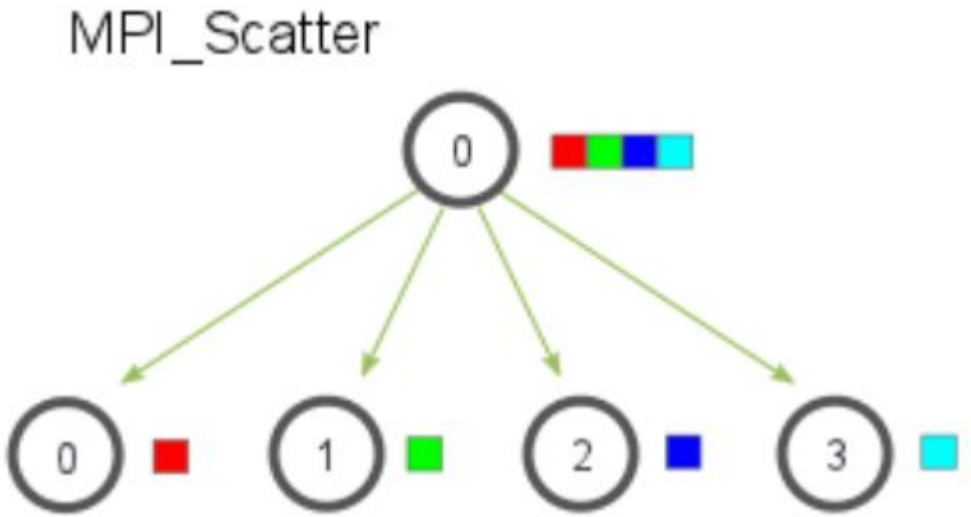


https://mpitutorial.com/tutorials/mpi-broadcast-and-collective-communication/

COLLECTIVE COMMUNICATION (1-N)

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

- **All processes send a message to root**

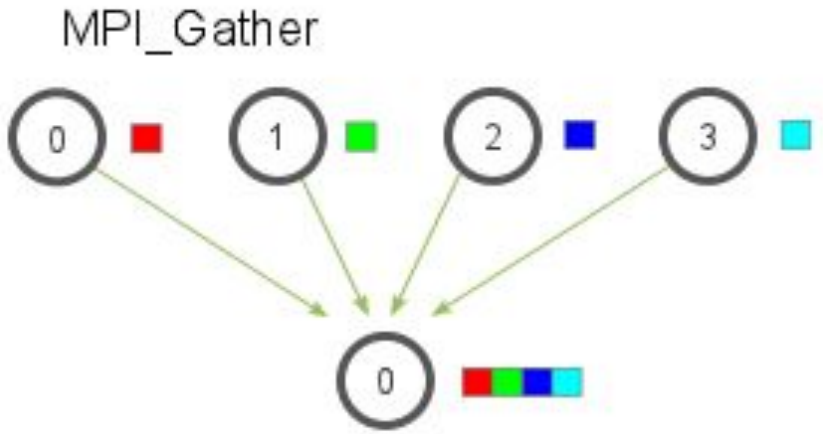


https://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/

COLLECTIVE COMMUNICATION (N-1)

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

- **Root receives a** message **from all processes** in the communicator **(including root!)**



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

COLLECTIVE COMMUNICATION (N-1)

Execute the Synchronize, Broadcast and Scatter_Gather examples available on blackboard and observe their behaviour!

```
mpicc source.c -o executable_name
mpirun –np <num_processes>  ./executable_name
```

TASK: DISTRIBUTED CALCULATOR

Using the examples we've seen today write a distributed calculator program!

- The root process (0) will broadcast number to 4 worker processes (including the root).
- Each worker will have a designated operation (0: addition, 1:subtraction, 2:multiplication, 3: division)
- After receiving the data from the root each worker will perform that operation
- Finally, the root processes will gather and print all results

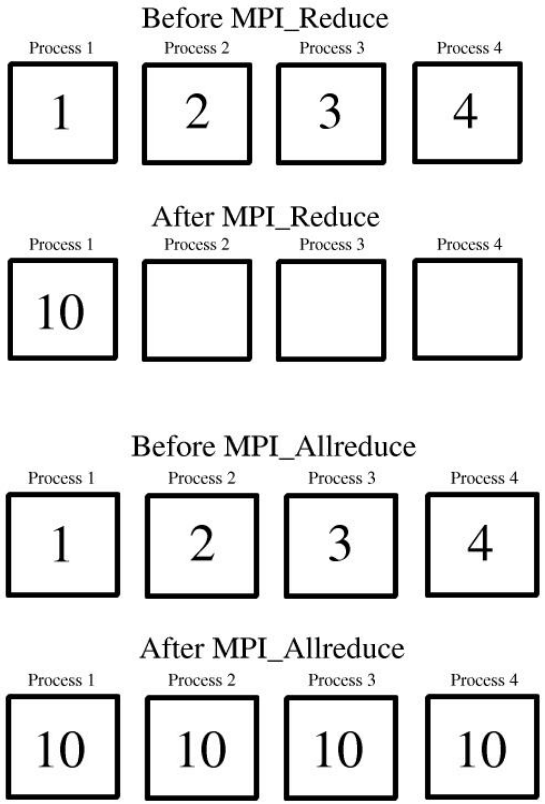
```
mpicc source.c -o executable_name
mpirun –np <num_processes>  ./executable_name
```

COLLECTIVE COMMUNICATION (N-N)

Reduce methods implement a distributed computation using data distributed over all processes

MPI_Reduce: result of computation is gathered by the ROOT

MPI_Allreduce: result of computation is broadcasted to all processes



BUILT-IN OPERATIONS IN MPI

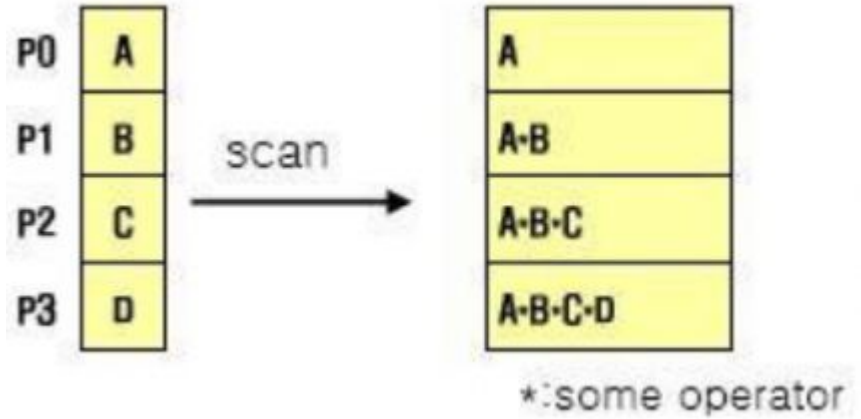
[MPI_MAX] maximum
[MPI_MIN] minimum
[MPI_SUM] sum
[MPI_PROD] product
[MPI_LAND] logical and
[MPI_BAND] bit-wise and
[MPI_LOR] logical or
[MPI_BOR] bit-wise or
[MPI_LXOR] logical xor
[MPI_BXOR] bit-wise xor
[MPI_MAXLOC] max value and location
[MPI_MINLOC] min value and location

https://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/

COLLECTIVE COMMUNICATION (N-N)

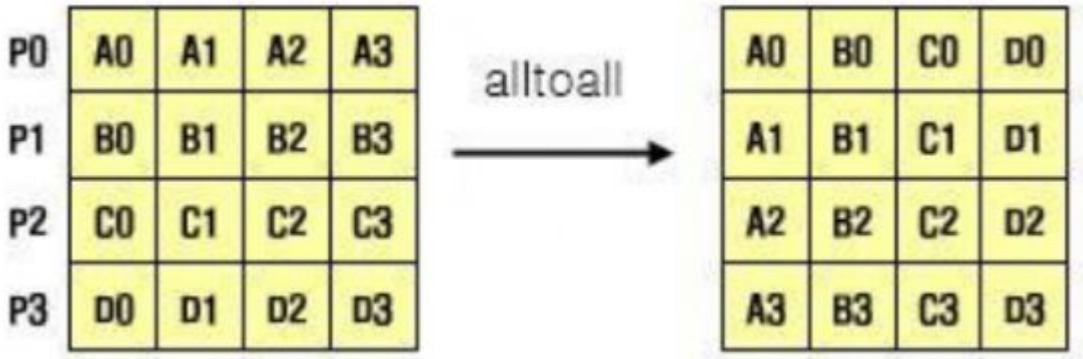
Scan: Each process get the first “rank” items

- process 1 gets the first item
- process 2 gets the first two items



AlltoAll: Each process gets **all** “rank” items

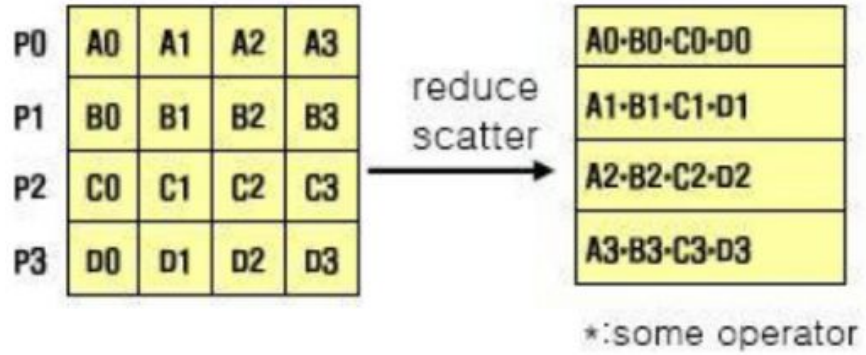
- process 1 gets all the items with index 1
- process 2 gets all items with index 2



COLLECTIVE COMMUNICATION (N-N)

MPI_Reduce_scatter: scatters data and perform reduction

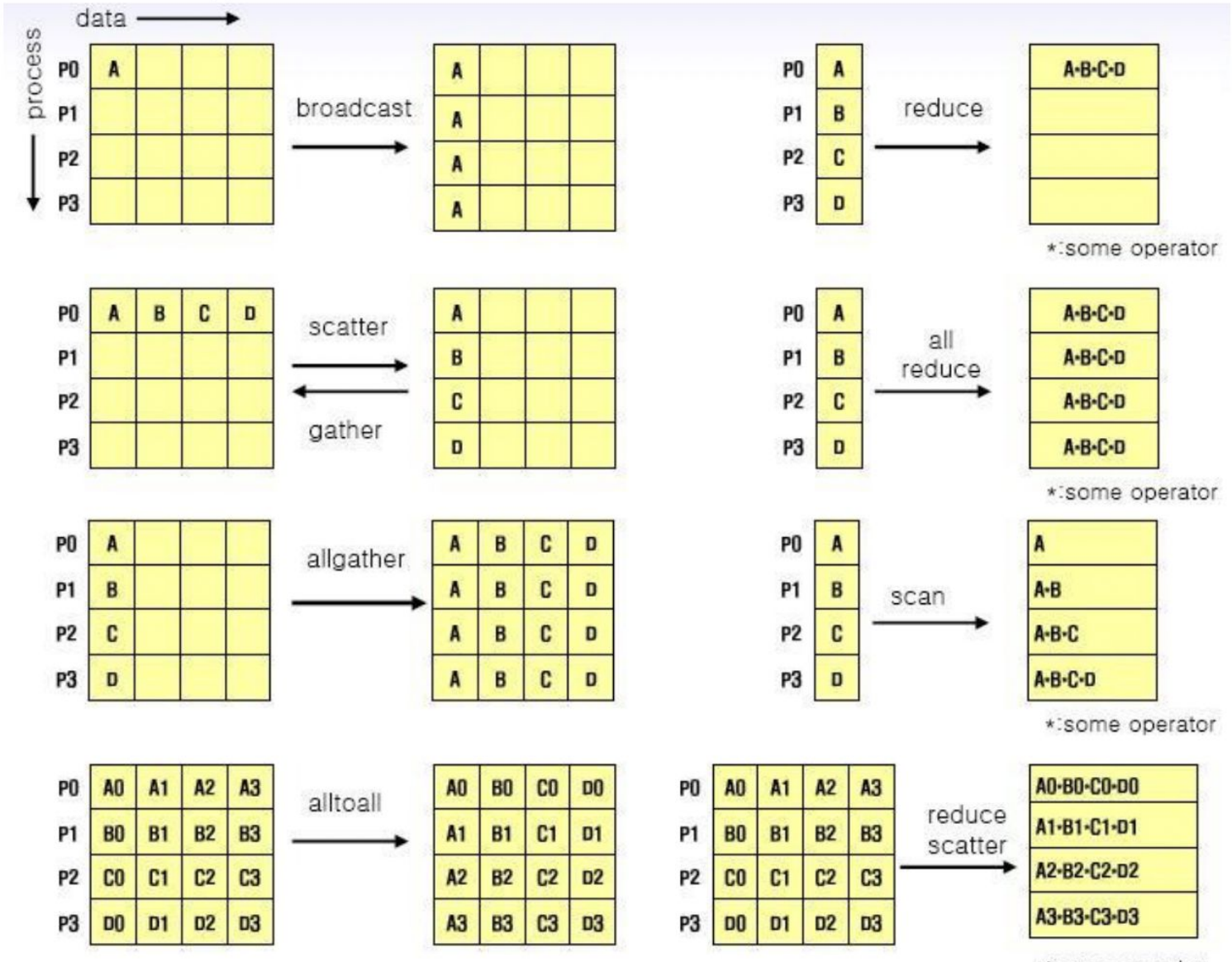
The 'scatter' operation is an **AllToAll operation!**



BUILT-IN OPERATIONS IN MPI

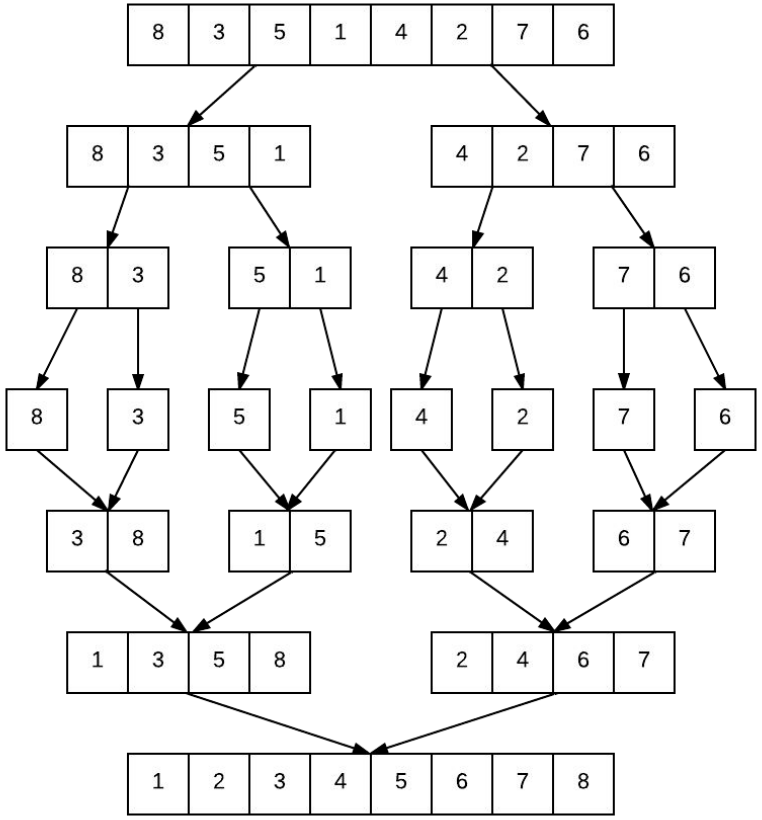
[MPI_MAX] maximum
[MPI_MIN] minimum
[MPI_SUM] sum
[MPI_PROD] product
[MPI_LAND] logical and
[MPI_BAND] bit-wise and
[MPI_LOR] logical or
[MPI_BOR] bit-wise or
[MPI_LXOR] logical xor
[MPI_BXOR] bit-wise xor
[MPI_MAXLOC] max value and location
[MPI_MINLOC] min value and location

Collective communication in one figure



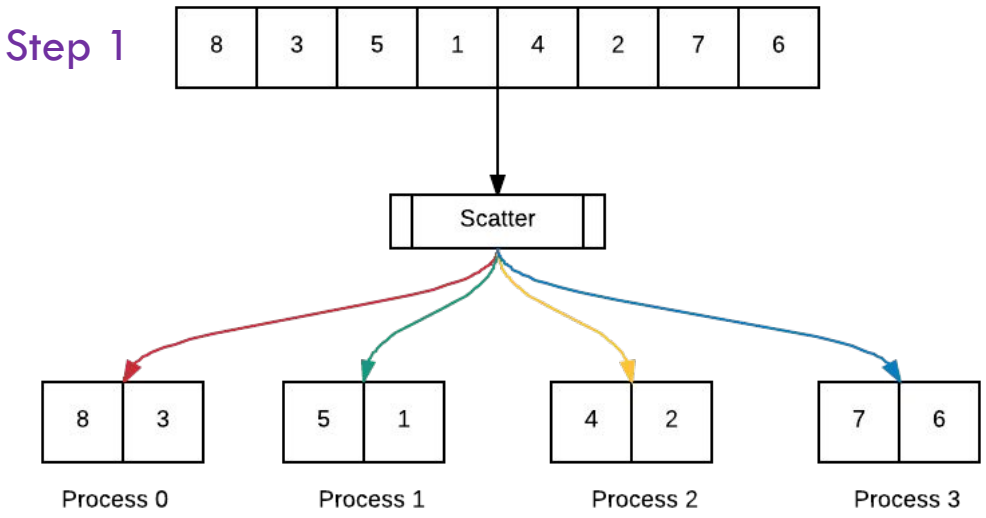
PARALLELIZE COMPUTATION – MERGE SORT

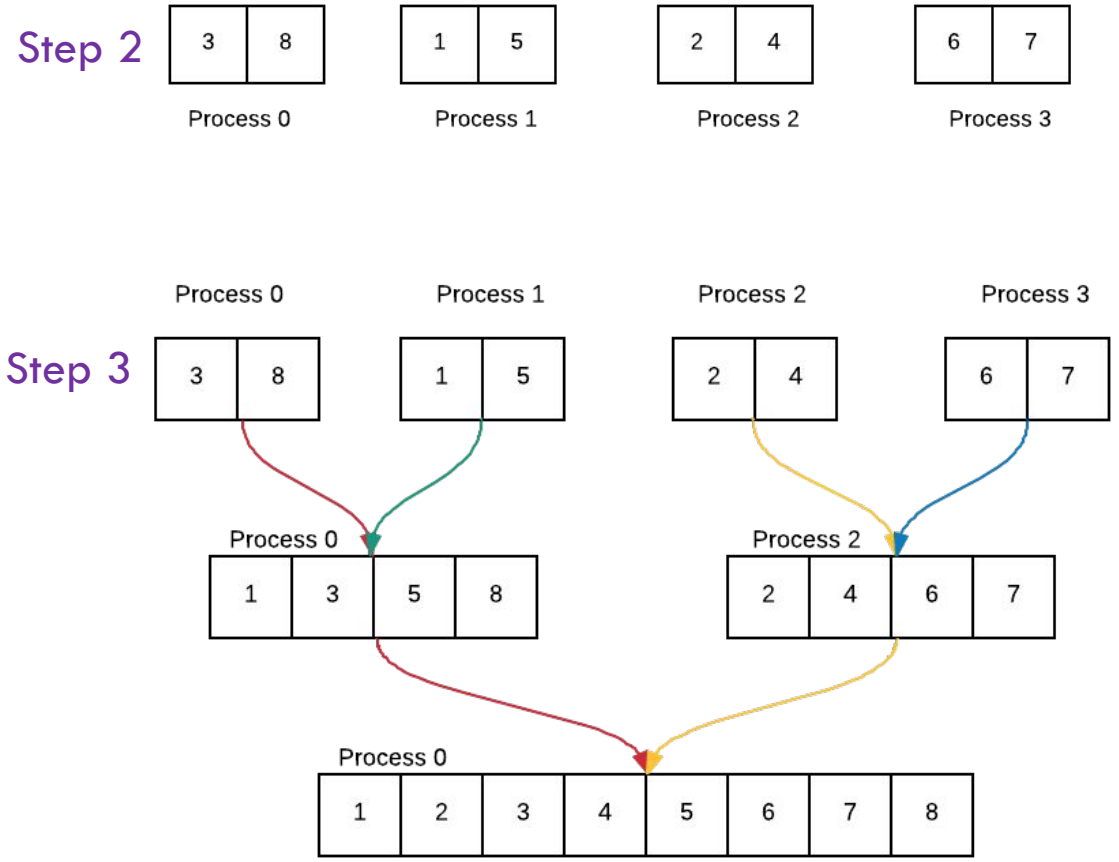
- Merge sort is a classic divide-and-conquer sorting algorithm
- Process: Divide list into unsorted sub-lists, then sort sub-lists, finally merge all sorted sub-lists.
- Good candidate for parallelize: Sorting of sub-lists are independent!



PARALLELIZE COMPUTATION – MERGE SORT

- Merge sort is a classic divide-and-conquer sorting algorithm!
- Process:
  - Divide list into unsorted sub-lists
  - sort sub-lists,
  - merge all sorted sub-lists.
- Good candidate for parallelization: Sorting of sub-lists are independent!





COLLECTIVE COMMUNICATION (N-1)

Execute the NtoN, Scan, AllToAll and Reduce Scatter examples available on blackboard and observe their behaviour!

```
mpicc source.c -o executable_name
mpirun –np <num_processes>  ./executable_name
```

TASK: DISTRIBUTED DOT PRODUCT

Using 3 workers perform a dot product operation on 2 vectors of length 9

Step 1: Root (0) scatters the vectors over the workers (including self)

Step 2: Workers perform dot product on sub-vectors

Step 3: Perform a reduction on dot products (MPI_SUM) and print result on root (0)

```
mpicc source.c -o executable_name
mpirun –np <num_processes>  ./executable_name
```