High Performance Computing and Parallel Programming

Teeraparb Chantavat

Institute for Fundamental Study
Naresuan University

Introduction to OpenMP with C Programming

Synchronization in OpenMP

What is Synchronization?

- Synchronization is a mechanism to ensure that multiple threads can safely access shared data without causing data races or inconsistencies.
- It is crucial for maintaining data integrity when threads perform read/write operations on shared resources.

Synchronization in OpenMP

Key Concepts

- **Parallel Programming**: Multiple threads execute code concurrently, potentially accessing shared resources.
- **Race Conditions**: When two or more threads access shared data simultaneously and the outcome depends on the timing of the accesses, leading to unpredictable results.
- **Synchronization**: Techniques used to control the access of multiple threads to shared resources, preventing race conditions.

Synchronization in OpenMP

Type of Synchronization in OpenMP

Critical Sections (#pragma omp critical):

- Ensures that only one thread executes the enclosed block of code at any time.
- Used when threads need exclusive access to a shared resource.

```
#pragma omp parallel for
for (int i = 0; i < 100; i++) {
    #pragma omp critical
    {
        sum += i;
    }
}
```

- The '#pragma omp critical' ensures that only one thread at a time updates the sum variable, preventing race conditions.

Synchronization in OpenMP

Type of Synchronization in OpenMP

Atomic Operations (#pragma omp atomic):

- Ensures that a specific memory operation (like incrementing a variable) is performed atomically, preventing race conditions on a single variable.

```
int sum = 0;

#pragma omp parallel for
for (int i = 0; i < 100; i++) {
    #pragma omp atomic
    sum += i;
}
```

- The '#pragma omp atomic' directive makes the increment operation atomic, ensuring that updates to sum happen safely without the need for a full critical section.

Synchronization in OpenMP

Type of Synchronization in OpenMP

Barrier Operations (#pragma omp barrier):

- Synchronizes all threads in a team; no thread proceeds beyond the barrier until all threads have reached it.

```
#pragma omp parallel
{
    int thread_id = omp_get_thread_num();
    printf("Thread %d: Executing code block 1\n", thread_id);
    #pragma omp barrier  // Synchronize all threads here
    printf("Thread %d: Executing code block 2\n", thread_id);
}
```

- The '#pragma omp barrier' forces all threads to wait at the barrier until every thread has completed the preceding code, ensuring synchronized execution before proceeding to the next.

Synchronization in OpenMP

Type of Synchronization in OpenMP

Single/Master (#pragma omp single / #pragma omp master):

- Single: Only one thread (the first to reach the directive) will execute the block.

```
#pragma omp parallel
{
    #pragma omp single
    {
        printf("Executed by a single thread: %d\n", omp_get_thread_num());
    }
    #pragma omp master
    {
        printf("Executed by the master thread: %d\n", omp_get_thread_num());
    }
}
```

See openmp_c/ex16_omp_atomic.c

See openmp_c/ex17_omp_single_master.c

Exercise: Bank Account

Implementing a Thread-Safe Bank Account with OpenMP

You are given transaction data in the following format:

- The data consists of two columns.
- The first column contains either a + or - sign, indicating a deposit (+) or withdrawal (-).
- The second column contains the amount of the deposit or withdrawal.

Write a C program that processes this transaction data using multiple threads. Each thread will manage a part of the transaction data and update a shared bank account balance.

Exercise: Bank Account

Implementing a Thread-Safe Bank Account with OpenMP

Expected Output: The output should include information similar to the following (order may vary due to parallel execution):

```
Thread 0: Deposit $100.00, Balance: $100.00
Thread 1: Withdrawal $50.00, Balance: $50.00
Thread 2: Deposit $25.00, Balance: $75.00
Thread 3: Withdrawal $10.00, Balance: $65.00
```

Read the transaction from the file `transactions.txt`.

Introduction to MPI with Python

Message Passing Interface (MPI)

What is MPI?

- MPI stands for Message Passing Interface.
- It is a standardized and portable **communication protocol** used to program parallel computers.
- MPI is used in **distributed memory systems**, where multiple processors work together by sending and receiving messages.

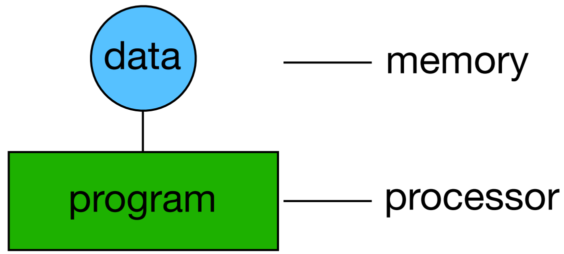
Message Passing Interface (MPI)

Why Use MPI?

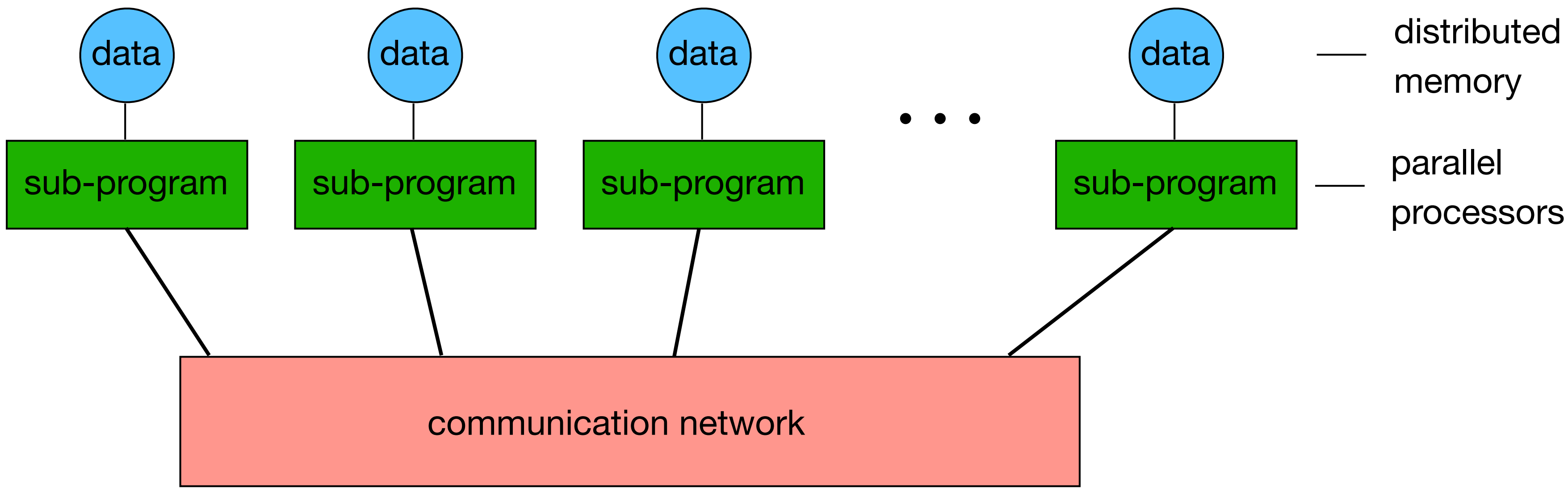
- **Scalability**: Efficient for large-scale systems with many processors.
- **Portability**: Can run on various architectures (clusters, supercomputers, multi-core systems).
- **Flexibility**: Supports both point-to-point and collective communication between processes.
- **Standardization**: MPI is a standard, ensuring code portability across different platforms.

The Message-Passing Programming Paradigm

- **Sequential Programming Paradigm**

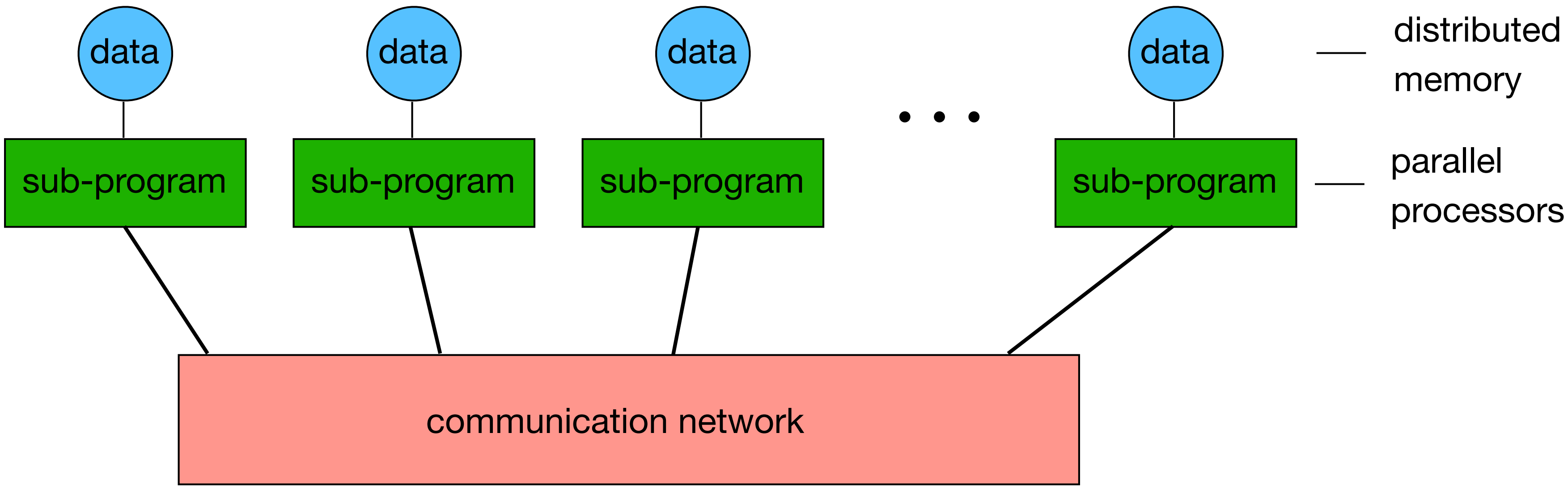


- **Message-Passing Programming Paradigm**



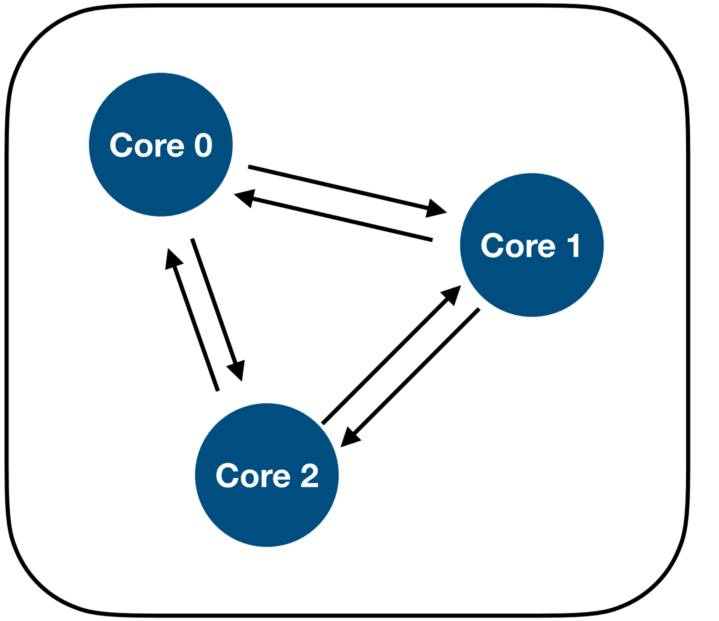
The Message-Passing Programming Paradigm

- Each processor in a message passing program
  - written in a conventional sequential language, e.g., C/C++, Fortran, or Python
  - typically the same on each processor (SPMD)
  - the variables of each sub-program have
  - communicate via
    - **the same name**
    - **but different locations (distributed memory) and different data!**
    - **i.e. all variables are private**
  - communicate via special send & receive routines ***(message passing)***



Communicators

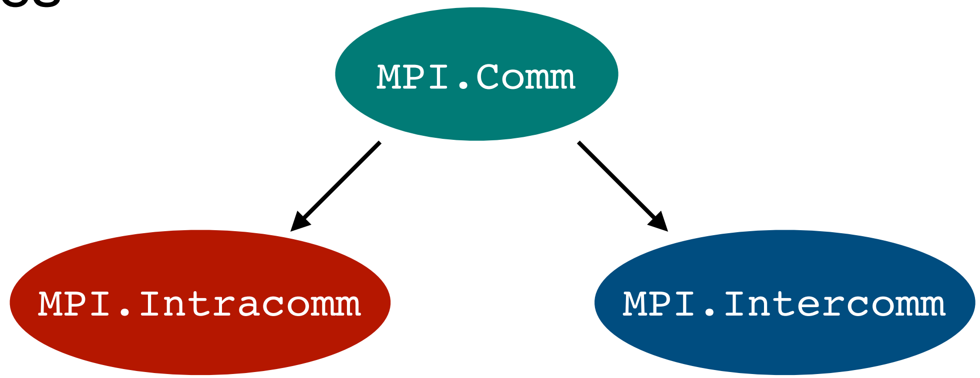
A **communicator** is a group of cores that can communicate to one another. Each cores is independent and can run its own process.



Communicator A **Communicator B**

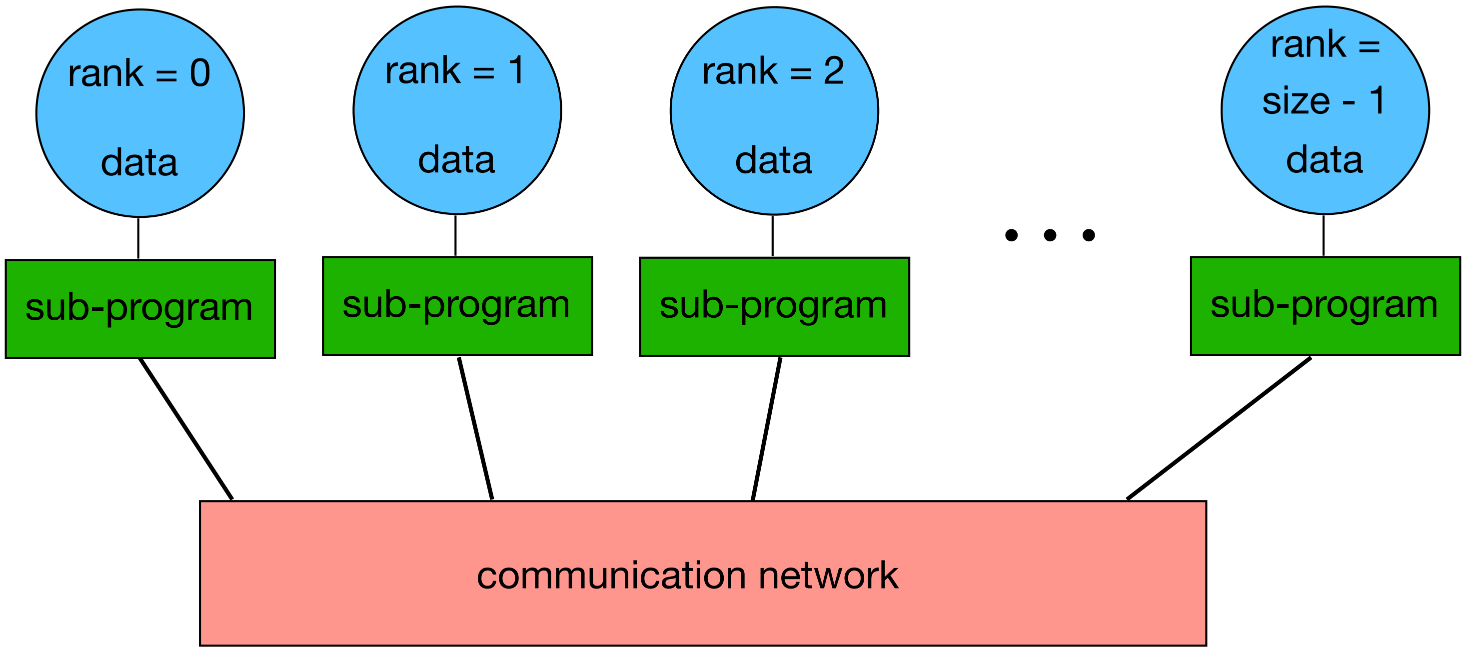
Communicators

- In *MPI for Python*, `MPI.Comm` is the base class of communicators.
- `MPI.Intracomm` is a sub-class of `MPI.Comm` mainly used for local nodes. **`MPI.COMM_WORLD`** and **`MPI.COMM_SELF`** are pre-defined instances of `MPI.Intracomm`.
- `MPI.Intercomm` is also a sub-class of `MPI.Comm` mainly used for remote nodes



Data and Work Distribution

- The value of **rank** is returned by special library routine.
- The system of **size** processes is started by special MPI initialization.
- All distribution decision are based on rank
- i.e., which process work on which data



See mpi_python/ex00_mpi_greeting.py

Point-to-point Communications

- Simplest form of message passing.
- One process (core) sends a message to another. Another process (core) will receive the message

`MPI.Comm.Send` `MPI.Comm.Recv`

```
MPI.Comm.Send(buf, dest, tag=0)
```

```
MPI.Comm.Recv(buf, source=ANY_SOURCE, tag=ANY_TAG,
              status=None)
```

See mpi_python/ex01_mpi_send_recv0.py

MPI DataType

- When sending an array of data, the type of MPI needs to know the type of data.
- Sometime the data type has to be specified in the data buffer.

MPI.CHAR	MPI.LONG
MPI.BYTE	MPI.FLOAT
MPI.SHORT	MPI.DOUBLE
MPI.INT	MPI.COMPLEX

See mpi_python/ex02_mpi_send_recv1.py

MPI for Python object

- On top of Standard MPI commands, `mpi4py` has commands for generic Python objects using all lowercases.

- Python objects are serialized / deserialized using `pickle` module

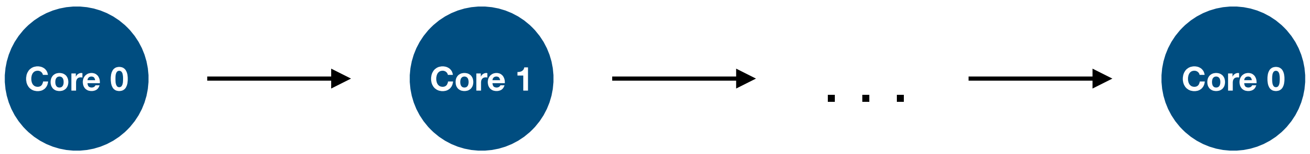
```
MPI.Comm.send(object, dest, tag=0)
```

```
MPI.Comm.recv(object, source=ANY_SOURCE, tag=ANY_TAG,
              status=None)
```

See mpi_python/ex03_mpi_send_recv2.py

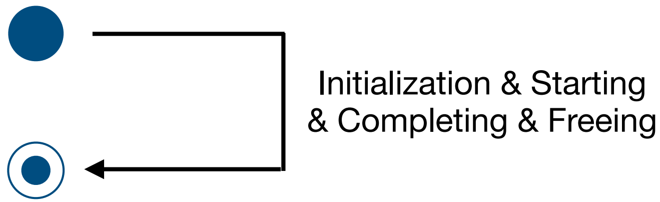
Exercise: Ring of Communication

Write an MPI program using `mpi4py` where each process sends a message (e.g., an integer or a string) to the next process in a circular manner. The message starts from process 0, is passed around all processes, and then returns back to process 0.

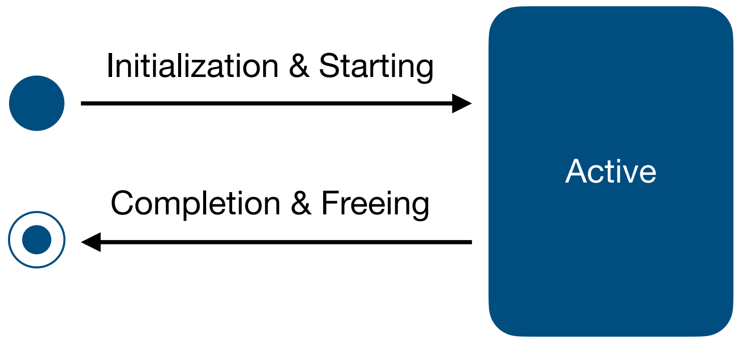


MPI Operations

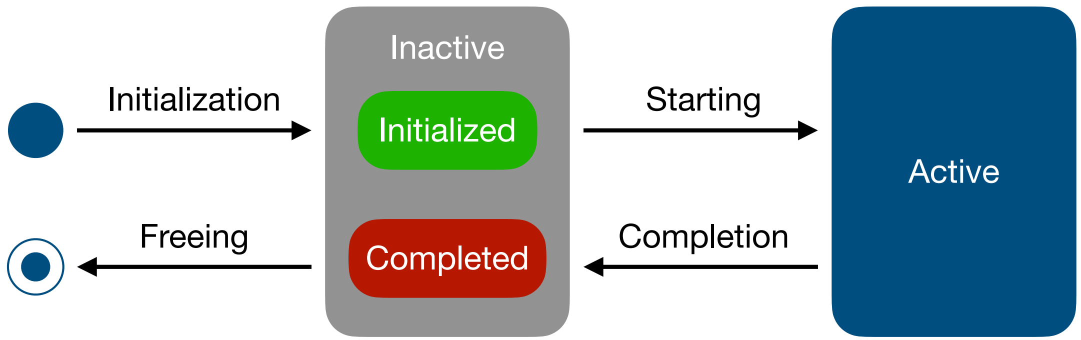
- **Blocking**
  Initialization & Starting & Completion & Freeing



- **Non-blocking**
  Initialization & Starting
  Completion & Freeing



- **Persistent**
  Initialization
  Starting
  Completion
  Freeing



See mpi_python/ex04_send_recv3.py

Exercise: Vector Addition

Implement a program that performs parallel vector addition using **non-blocking** communication. Each core will hold a vector and send their vector to the master core to get the sum. The final result will be distributed across all cores.

Steps

- Create vectors $\boldsymbol{A}_i$ of size N on for i = 0, …, M - 1, where M is the number of cores.
- Each core will send their vector to the master core (rank 0).
- The master core will send the total sum of all vectors back to all cores.
- Think about where we should put wait() command.

MPI Send Communication Modes

`MPI_Send` has 4 modes of communication while `MPI_Recv` has only 1 mode. Both can be blocking or non-blocking.

- **Standard Mode**
  command: `MPI.Comm.Send`
- **Synchronous Mode**
  command: `MPI.Comm.SSend`
- **Buffered (Asynchronous) Mode**
  command: `MPI.Comm.BSend`
- **Ready**
  command: `MPI.Comm.RSend`

Each of these modes has its own behavior regarding how data is sent and whether the sender or receiver has to be ready for the communication to proceed.

MPI Send Communication Modes

`MPI_Send` has 4 modes of communication while `MPI_Recv` has only 1 mode. Both can be blocking or non-blocking.

- **Standard**
  The MPI library decide whether or not the non-local buffered the outgoing data.
- **Buffered (Asynchronous)**
  The MPI library decide to use buffer for outgoing data if no matching receiver has been posted.
- **Synchronous**
  The outgoing data buffer can be reused once the receiving process starting receiving the data.
- **Ready**
  The outgoing data buffer can be reused once the receiving process has been posted.

MPI Send - Standard Mode

In **standard mode**, the message may or may not be buffered. This means that the send operation can return before the matching receive is posted. The completion of `Send` doesn't guarantee that the message has been received.

```
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = np.array([1, 2, 3, 4], dtype='i')
    comm.Send([data, MPI.INT], dest=1)
    print(f"Process {rank} sent data")
elif rank == 1:
    data = np.empty(4, dtype='i')
    comm.Recv([data, MPI.INT], source=0)
    print(f"Process {rank} received data: {data}")
```

MPI Send - Synchronous Mode

In **synchronous mode**, the send operation only completes when the matching receive has been posted and the message has started to be received. This mode ensures that the sender waits for the receiver to be ready, making it synchronous.

```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = np.array([5, 6, 7, 8], dtype='i')
    comm.Ssend([data, MPI.INT], dest=1)
    print(f"Process {rank} sent data synchronously")
elif rank == 1:
    data = np.empty(4, dtype='i')
    comm.Recv([data, MPI.INT], source=0)
    print(f"Process {rank} received data: {data}")
```

MPI Send - Buffered Mode

In **buffered mode**, the message is first buffered by the sender, allowing the send operation to complete even if no matching receive has been posted yet. The user must provide a buffer before using this mode. It is useful when you want to avoid blocking even when the receiver isn't ready.

```
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = np.array([9, 10, 11, 12], dtype='i')
    buffer = np.empty(100, dtype='b')
    MPI.Attach_buffer(buffer)
    comm.Bsend([data, MPI.INT], dest=1)
    print(f"Process {rank} sent data with buffered send")
    MPI.Detach_buffer()
elif rank == 1:
    data = np.empty(4, dtype='i')
    comm.Recv([data, MPI.INT], source=0)
    print(f"Process {rank} received data: {data}")
```

MPI Send - Ready Mode

In **ready mode**, the send operation can only be initiated if the matching receive has already been posted. If the receive isn't ready, the behavior is undefined (it may lead to errors). This mode is efficient when you know the receiver is ready beforehand.

```python
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = np.array([13, 14, 15, 16], dtype='i')
    comm.Barrier()
    comm.Rsend([data, MPI.INT], dest=1)
    print(f"Process {rank} sent data with ready send")
elif rank == 1:
    data = np.empty(4, dtype='i')
    comm.Barrier()
    comm.Recv([data, MPI.INT], source=0)
    print(f"Process {rank} received data: {data}")
```

MPI Send Communication Modes

Summary of the 4 Modes

Communication Mode	Behaviour
Standard	May or may not block, depends on system buffering. The send operation completes once the data is ready for transmission.
Synchronous	Blocks until the matching receive is posted and the message starts being received. Ensures sender and receiver are synchronized.
Buffered (Asynchronous)	Uses a buffer to allow the send operation to complete even if the receiver isn't ready. Requires a user-provided buffer.
Ready	Requires that the matching receive is already posted. Undefined behavior if the receiver isn't ready.

See mpi_python/ex05_send_ssend.py

See mpi_python/ex06_bsend_rsend.py

Blocking and non-Blocking

- **MPI_Send** is a blocking send operation, meaning that the function will not return until the data has been copied out of the send buffer and is safe for reuse or modification.
- **MPI_Isend** is a non-blocking send operation, meaning the function returns immediately, even if the data has not yet been sent. The user must call MPI_Wait or MPI_Test to ensure the send completes before modifying the buffer.

See mpi_python/ex07_mpi_send_isend.py

MPI Send Variants

Communication Mode	Blocking	Non-blocking	Persistent
Standard	`MPI.Comm.Send` `MPI.Comm.send`	`MPI.Comm.Isend` `MPI.Comm.isend`	`MPI.Comm.Send_init`
Synchronous	`MPI.Comm.Ssend` `MPI.Comm.ssend`	`MPI.Comm.Issend` `MPI.Comm.issend`	`MPI.Comm.Bsend_init`
Buffered (Asynchronous)	`MPI.Comm.Bsend` `MPI.Comm.bsend`	`MPI.Comm.Ibsend` `MPI.Comm.ibsend`	`MPI.Comm.Ssend_init`
Ready	`MPI.Comm.Rsend`	`MPI.Comm.Irsend`	`MPI.Comm.Rsend_init`

for generic Python objects

MPI Recv Variants

Communication Mode	Blocking	Non-blocking	Persistent
Standard	`MPI.Comm.Recv` `MPI.Comm.recv`	`MPI.Comm.Irecv` `MPI.Comm.irecv`	`MPI.Comm.Recv_Init`

for generic Python objects

See mpi_python/ex08_mpi_recv_irecv.py

MPI Sendrecv Variants

- The **send-receive** operation combine in one operation the sending of a message to one destination and the receiving of another message, from another process

Communication Mode	Blocking

Standard	`MPI.Comm.Sendrecv`
	`MPI.Comm.Sendrecv_replace`
	`MPI.Comm.sendrecv`

```
MPI.Comm.Sendrecv(sendbuf, dest, sendtag=0, recvbuf=None,
     source=ANY_SOURCE, recvtag=ANY_TAG, status=None)
```

```
MPI.Comm.Sendrecv_replace(buf, dest, sendtag=0,
     recvtag=ANY_TAG, status=None)
```

See mpi_python/ex09_mpi_sendrecv.py

See mpi_python/ex10_mpi_sendrecv2.py

See mpi_python/ex11_mpi_sendrecv_replace.py

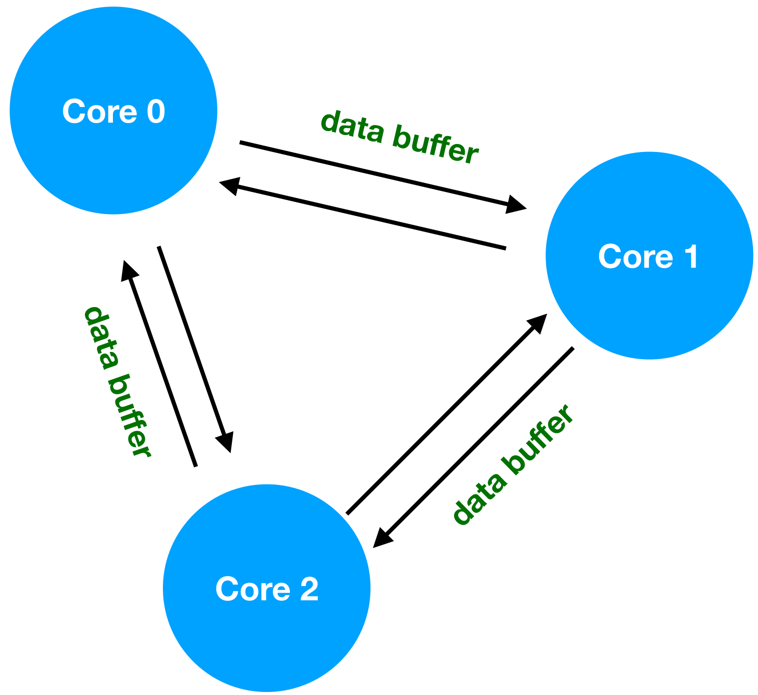
Collective Communication

What is Collective Communication?

- Collective communication in MPI involves communication operations that involve all **processes** in a communicator.
- Unlike point-to-point communication (which involves only two processes), collective operations ensure that **all processes participate**.

Collective Communication

- Collective communication routines are higher level routines.
- Several processes are involved at a time



Collective Communication

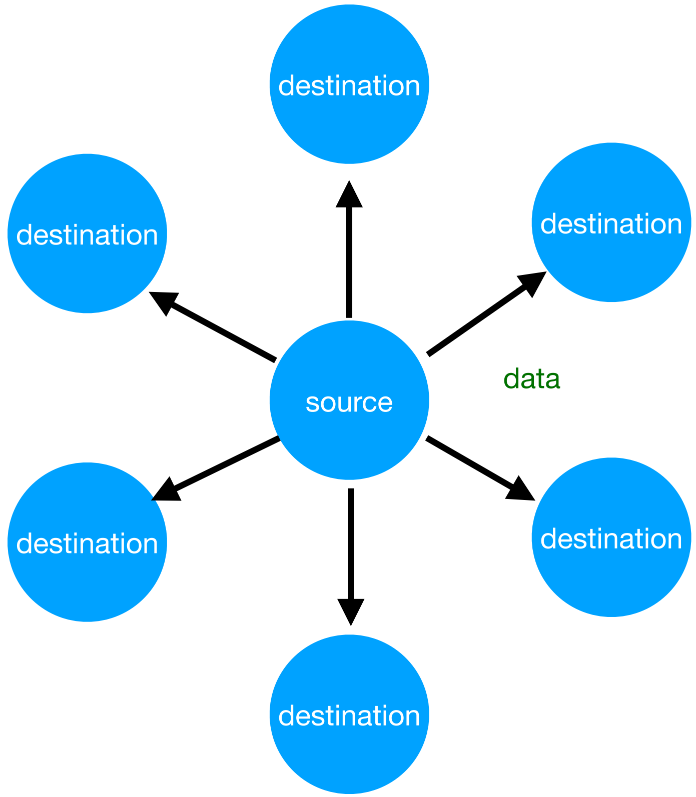
Types of Collective Communications:

- **Broadcast**: One process sends data to all other processes.
- **Scatter**: One process sends parts of the data equally to all other processes.
- **Gather**: All processes send data to one process (the root).
- **Allgather**: All processes send data to all other processes.
- **Reduce**: Combines values from all processes and sends the result to one process.
- **Allreduce**: Combines values from all processes and distributes the result to all processes.

Broadcast

- A one-to-many communication.
- All core receive the same copy of the data
- Not very efficient if we have only one communication channel.

```
MPI.Comm.Bcast(buf, root=0)
```



See mpi_python/ex12_mpi_bcast.py

MPI Bcast Variants

Communication Mode	Blocking	Non-blocking
Standard	`MPI.Comm.Bcast` `MPI.Comm.bcast`	`MPI.Comm.Ibcast`

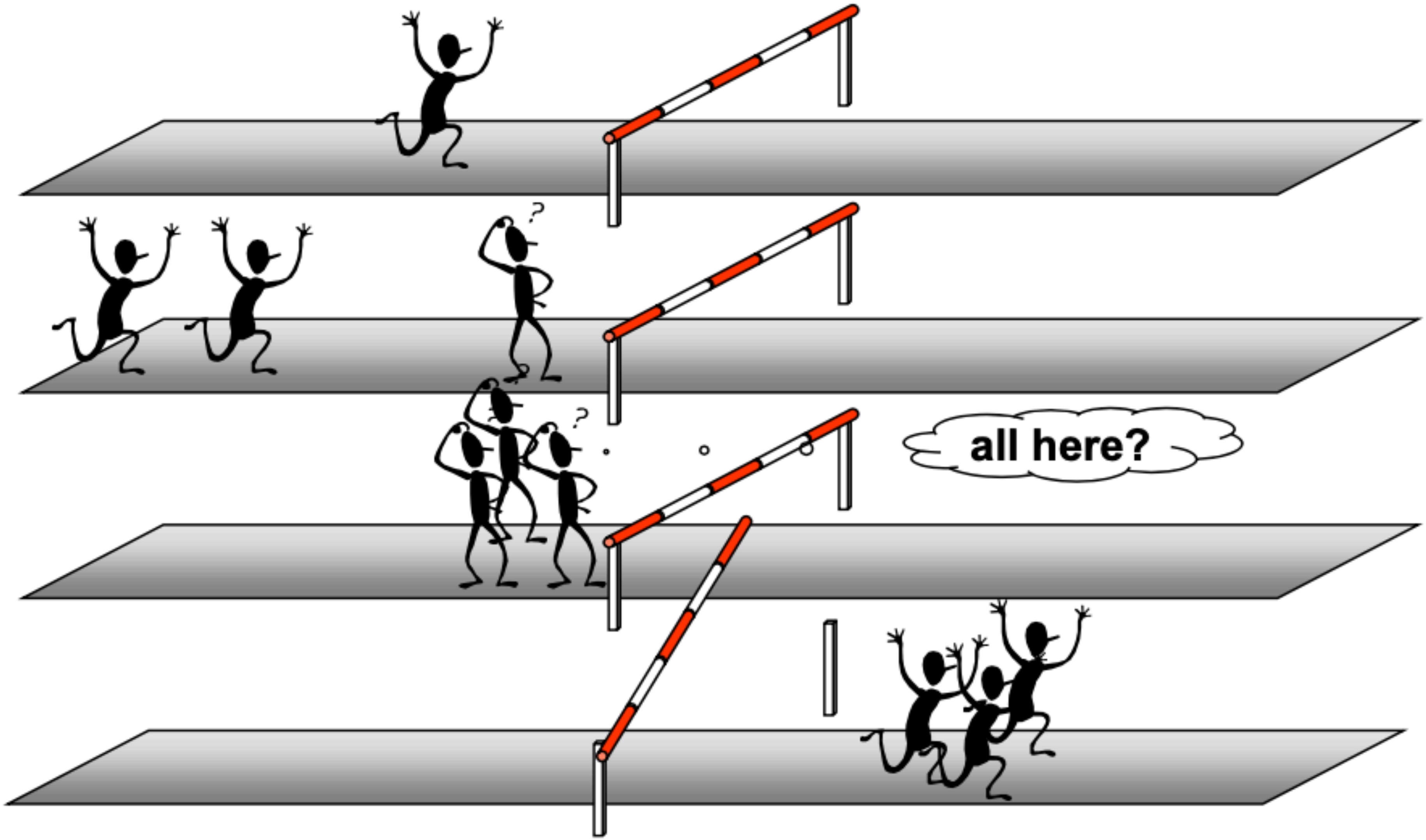
```
MPI.Comm.Ibcast(buf, root=0)
```

```
MPI.Comm.bcast(object, root=0)
```

See mpi_python/ex13_mpi_ibcast.py

Barrier

- Synchronize processes



MPI Barrier Variants

Communication Mode	Blocking	Non-blocking
Standard	`MPI.Comm.Barrier` `MPI.Comm.barrier`	`MPI.Comm.Ibarrier`

`MPI.Comm.Barrier()`

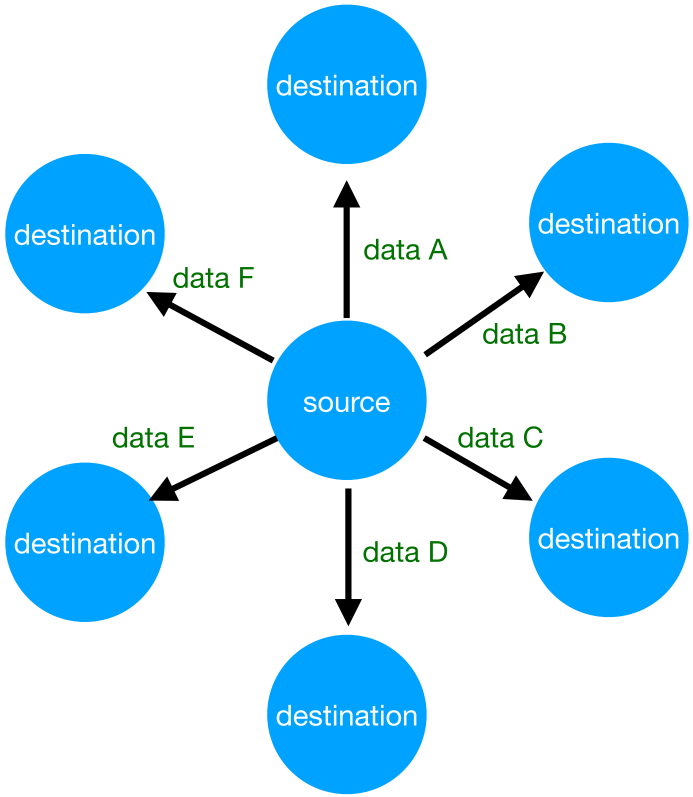
`MPI.Comm.Ibarrier()`

`MPI.Comm.barrier()`

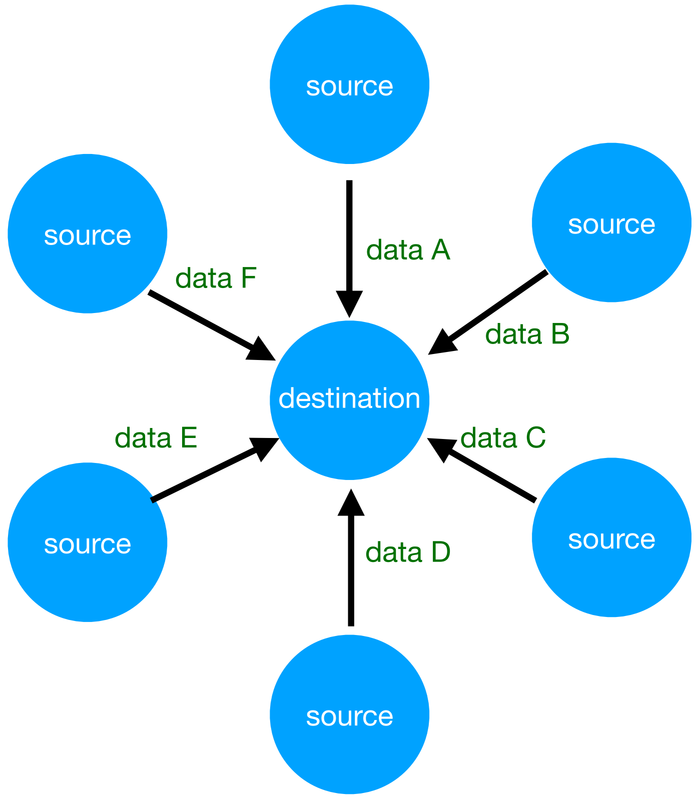
See mpi_python/ex14_mpi_ibarrier.py

Scatter and Gather

- Split the data equally to all other processes.



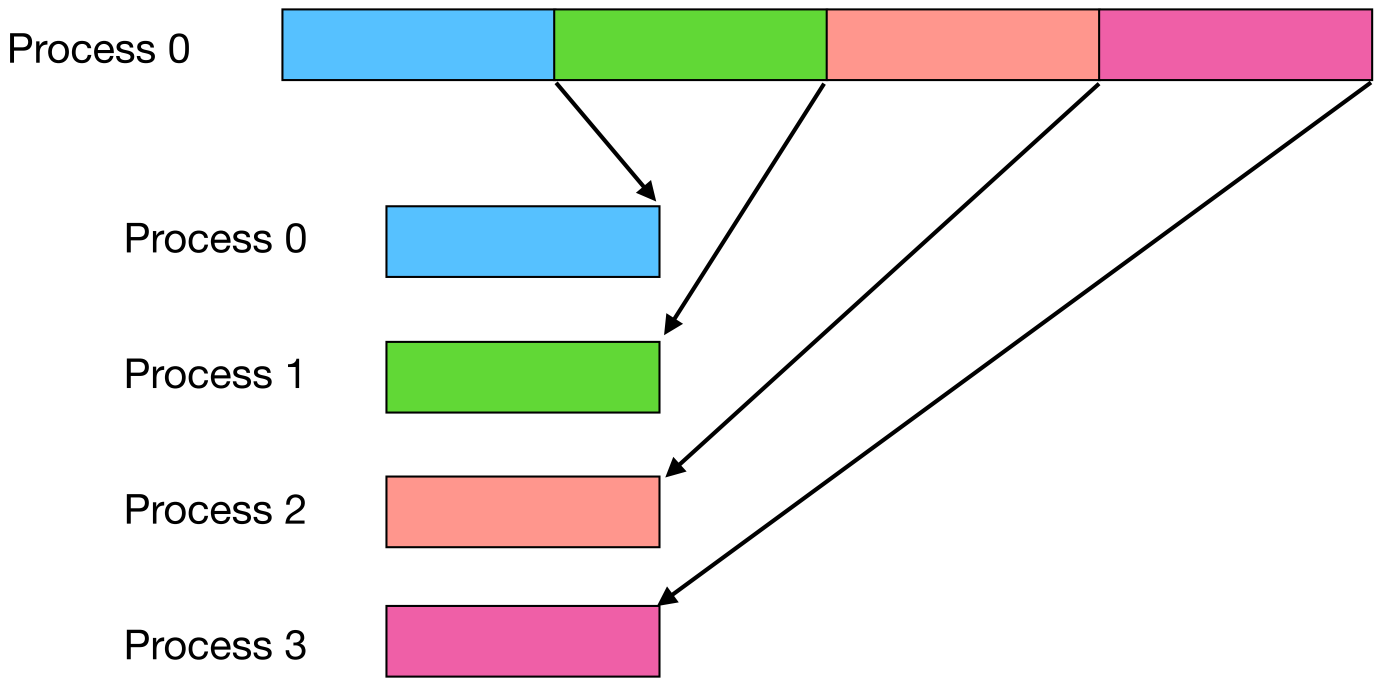
Scatter



Gather

Scatter and Gather

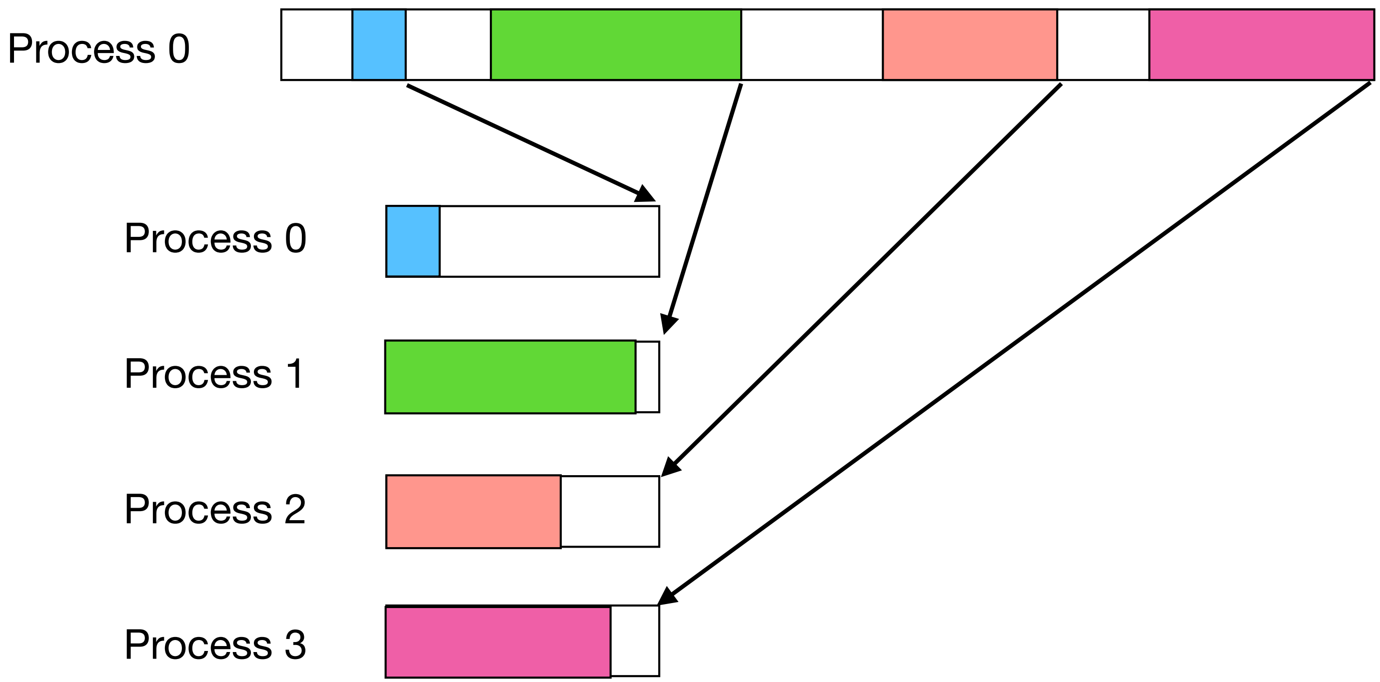
- Scatter (Gather) requires contiguous data and uniform data size



See mpi_python/ex15_mpi_scatter_gather.py

Scatter and Gather (Vector data)

- Scatterv (Gatherv) allows gaps between messages in source data
- Irregular message size are allowed.
- Data can be distributed to processes in any order



See mpi_python/ex16_mpi_scatterv_gatherv.py

MPI Scatter / Gather Variants

Communication Mode	Blocking	Non-blocking

Standard	`MPI.Comm.Scatter`
	`MPI.Comm.Scatterv`
	`MPI.Comm.scatter`
		`MPI.Comm.Iscatter`
		`MPI.Comm.Iscatterv`

Communication Mode	Blocking	Non-blocking

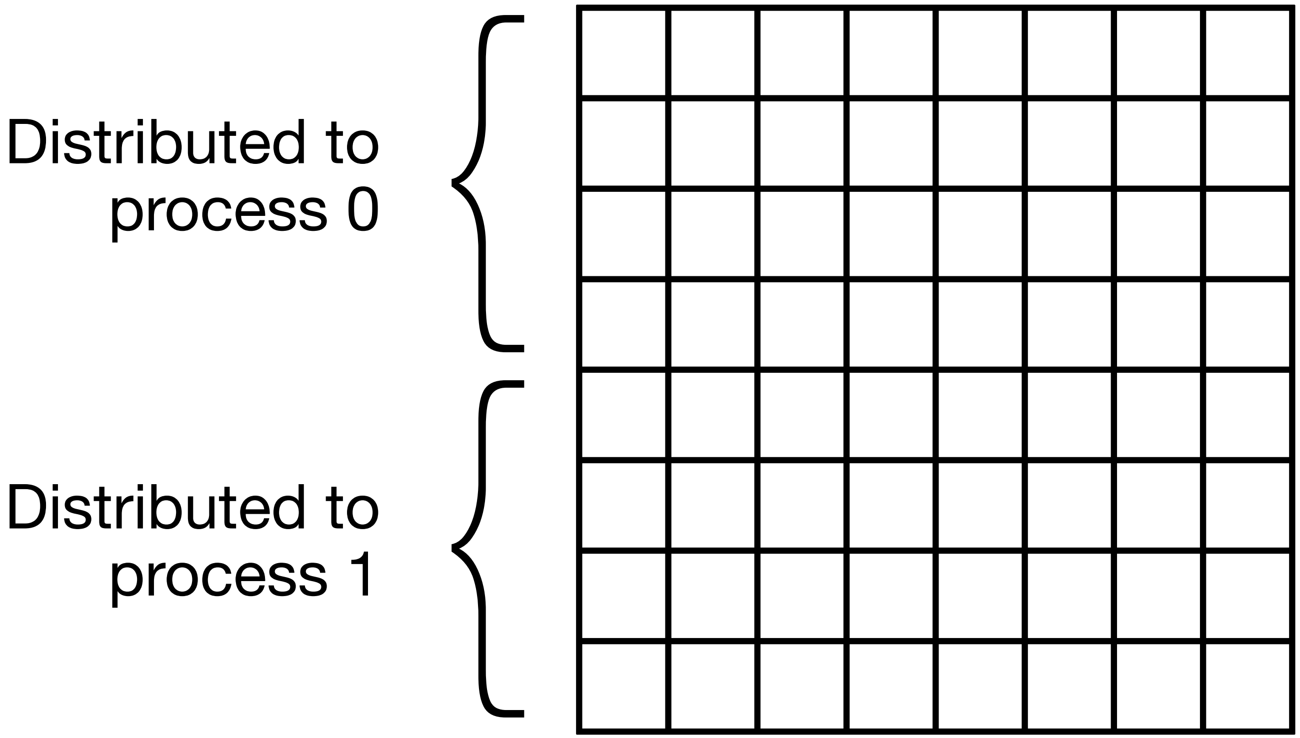
Standard	`MPI.Comm.Gather`
	`MPI.Comm.Gatherv`
	`MPI.Comm.gather`
		`MPI.Comm.Igather`
		`MPI.Comm.Igahterv`

See mpi_python/ex17_mpi_iscatter_igather.py

See mpi_python/ex18_mpi_iscatterv_igatherv.py

Exercise: Matrix Vector Multiplication

- The Matrix $\mathbf{A}$ is distribution across all processes (each process hold one or more row of the matrix). We will use Scatter (or its variants) to distribute part of the matrix to all processes; for example for 2 processes and an 8 x 8 matrix.



Exercise: Matrix Vector Multiplication

- Each process will have an identical vector $\mathbf{b}$. Use Bcast (or its variants) to broadcast the vector to all processes.
- Each process perform a matrix vector multiplication.

$$\mathbf{v} = \mathbf{A}\,\mathbf{b}$$

- The root process (rank 0) will Gather (or its variant) the result from other process and display the result of the matrix vector multiplication.

Allgather and Allgatherv

- Gathers data from all members of a group and sends the data to all members of the group.
- The Allgather (Allgatherv) function is similar to the MPI_Gather function, except that it sends the data to all processes instead of only to the root.

See mpi_python/ex19_mpi_allgather.py

MPI Allgather Variants

Communication Mode	Blocking	Non-blocking

Standard	`MPI.Comm.Allgather`
	`MPI.Comm.Allgatherv`
	`MPI.Comm.allgather`
		`MPI.Comm.Iallgather`
		`MPI.Comm.Iallgatherv`

```
MPI.Comm.Allgather(sendbuff, recvbuff)
```

```
MPI.Comm.Allgatherv(sendbuff, recvbuff)
```

```
MPI.Comm.allgather(sendobj)
```

All to all

- Gathers data from and scatters data to all members of a group.
- The **Alltoall** is an extension **Allgather** function.
- Each process sends distinct data to each of the receivers. The *j*th block that is sent from process *i* is received by process *j* and is placed in the *i*th block of the receive buffer.

All to all Variants

Communication Mode	Blocking	Non-blocking

Standard	`MPI.Comm.Alltoall`
	`MPI.Comm.Alltoallv`
	`MPI.Comm.Alltoallw`
	`MPI.Comm.alltoall`
		`MPI.Comm.Ialltoall`
		`MPI.Comm.Ialltoallv`
		`MPI.Comm.Ialltoallw`

```
MPI.Comm.Alltoall(sendbuff, recvbuff)
```

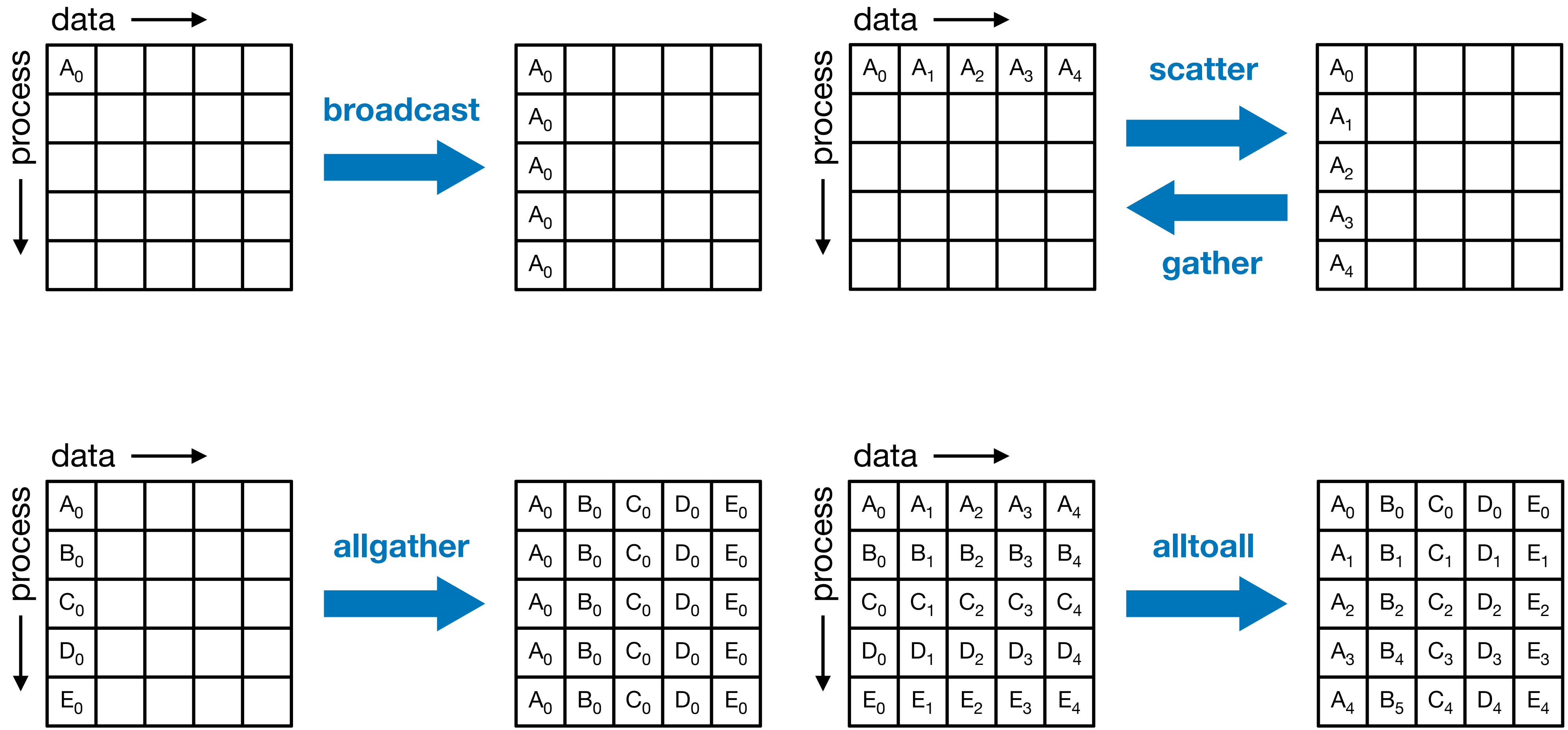
```
MPI.Comm.Alltoallv(sendbuff, recvbuff)
```

```
MPI.Comm.alltoall(sendobj)
```

See mpi_python/ex20_mpi_alltoall.py

See mpi_python/ex21_mpi_alltoall2.py

Summary of Collective Communication



MPI Reduce

What is MPI Reduce?

- Collects data from all processes and combines them using a reduction operation (e.g., sum, max, min).
- Only the root process receives the final result.
- Common operation are:
  - `MPI.SUM`: Adds all the values together.
  - `MPI.PROD`: Multiples all the values together.
  - `MPI.MIN`: Finds the maximum value.
  - `MPI.MAX`: Finds the minimum value.

MPI Operation Object

- Use with Reduce / Allreduce to perform reduction operation

`MPI.MAX`	Returns the maximum element
`MPI.MIN`	Returns the minimum element
`MPI.SUM`	Sums the elements
`MPI.PROD`	Multiplies all elements

Reduce / Allreduce

- Performs a global reduce operation across all members of a group.

Communication Mode	Blocking	Non-blocking

Standard	`MPI.Comm.Reduce`
	`MPI.Comm.Allreduce`
	`MPI.Comm.reduce`
	`MPI.Comm.allreduce`
		`MPI.Comm.Ireduce`
		`MPI.Comm.Iallreduce`

```
MPI.Comm.Reduce(sendbuff, recvbuff, op=MPI.SUM, root=0)
```

See mpi_python/ex22_mpi_reduce.py

See mpi_python/ex23_mpi_allreduce.py

MPI Reduce_scatter

- Performs a global reduce operation across all members of a group and scatter the result.

Communication Mode	Blocking	Non-blocking
Standard	`MPI.Comm.Reduce_scatter` `MPI.Comm.Reduce_scatter_block`	`MPI.Comm.Ireduce_scatter` `MPI.Comm.Ireduce_scatter_block`

```
MPI.Comm.Reduce_scatter(sendbuff, recvbuff, recvcounts=None,
op=SUM)
```

```
MPI.Comm.Reduce_scatter_block(sendbuff, recvbuff, op=SUM)
```

See mpi_python/ex24_mpi_reduce_scatter.py

See mpi_python/ex25_mpi_reduce_scatter_block.py

Introduction to MPI with C Programming

Message Passing Interface in C

MPI in C is based on the same principle as mpi4py and has a structure similarity.

```c
int main(void) {

    char    greeting[MAX_STRING];
    int     comm_sz;  /* Number of processes */
    int     my_rank;  /* For MPI functions, etc. */

    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

Need to initialize MPI where `mpi4py` does initialization automatically.

Equivalent to `MPI.Comm.Get_rank()`

`MPI.Comm.Get_size()`

in `mpi4py`

See mpi_c/ex01_mpi_hello.c

Message Passing Interface in C

```c
if (my_rank != 0) {
    sprintf(greeting, "Greetings from process %d of %d!",
            my_rank, comm_sz);
    MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
             MPI_COMM_WORLD);
}
```

Equivalent to `MPI.Comm.Send` in `mpi4py`

```c
printf("Greetings from process %d of %d!\n",
       my_rank, comm_sz);
for (int q = 1; q < comm_sz; q++) {
    MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
             0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("%s\n", greeting);
}
```

Equivalent to `MPI.Comm.Recv` in `mpi4py`

Message Passing Interface in C

```
    MPI_Finalize();
    return 0;
} /* main */
```

Need to finalize MPI where `mpi4py` does initialization automatically.

Summary

- In MPI in C, you have to initialize and finalize the MPI communication manually while `mpi4py` the processes are done automatically.
- All of the commands `mpi4py`, there is also an equivalent command in MPI in C programming language, but the input arguments may be slightly differ.

See mpi_c/ex02_mpi_send_recv1.c

See mpi_c/ex12_mpi_bcast.c

See mpi_c/ex22_mpi_reduce.c

Resources

OpenMP

- https://www.openmp.org

MPI

- https://learn.microsoft.com/en-us/message-passing-interface/microsoft-mpi

mpi4py

- https://mpi4py.readthedocs.io