Message-Passing Programming with MPI

Message-Passing Concepts

Reusing this material

Overview

- This lecture will cover
  - message passing model
  - SPMD
  - communication modes
  - collective communications

Programming Models

Serial Programming

Concepts

Arrays
Subroutines
Control flow
Variables
Human-readable
OO

Languages

Python
C/C++
Java
Fortran
struct
if/then/else

Implementations

gcc -O3
pgcc -fast
icc
crayftn
javac
craycc

Message-Passing Parallel Programming

Concepts

Processes
Send/Receive
SPMD
Collectives
Groups

Libraries

MPI
MPI_Init()

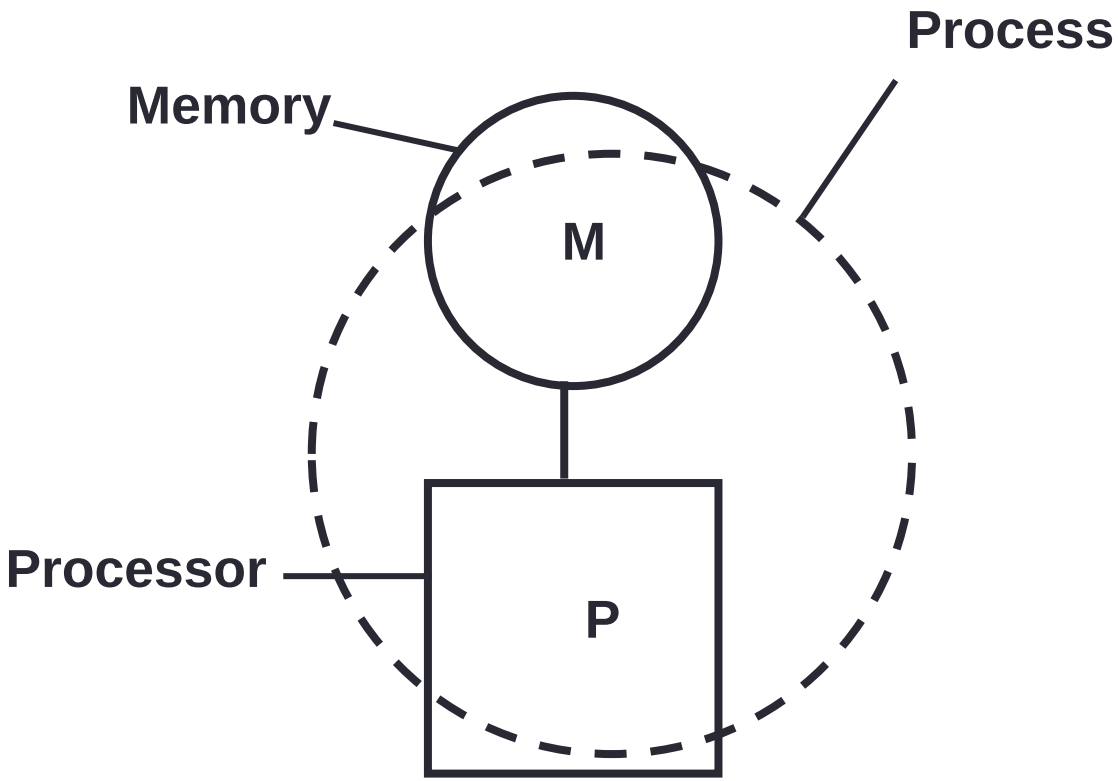
Implementations

Intel MPI
MPICH2
Cray MPI
OpenMPI
IBM MPI

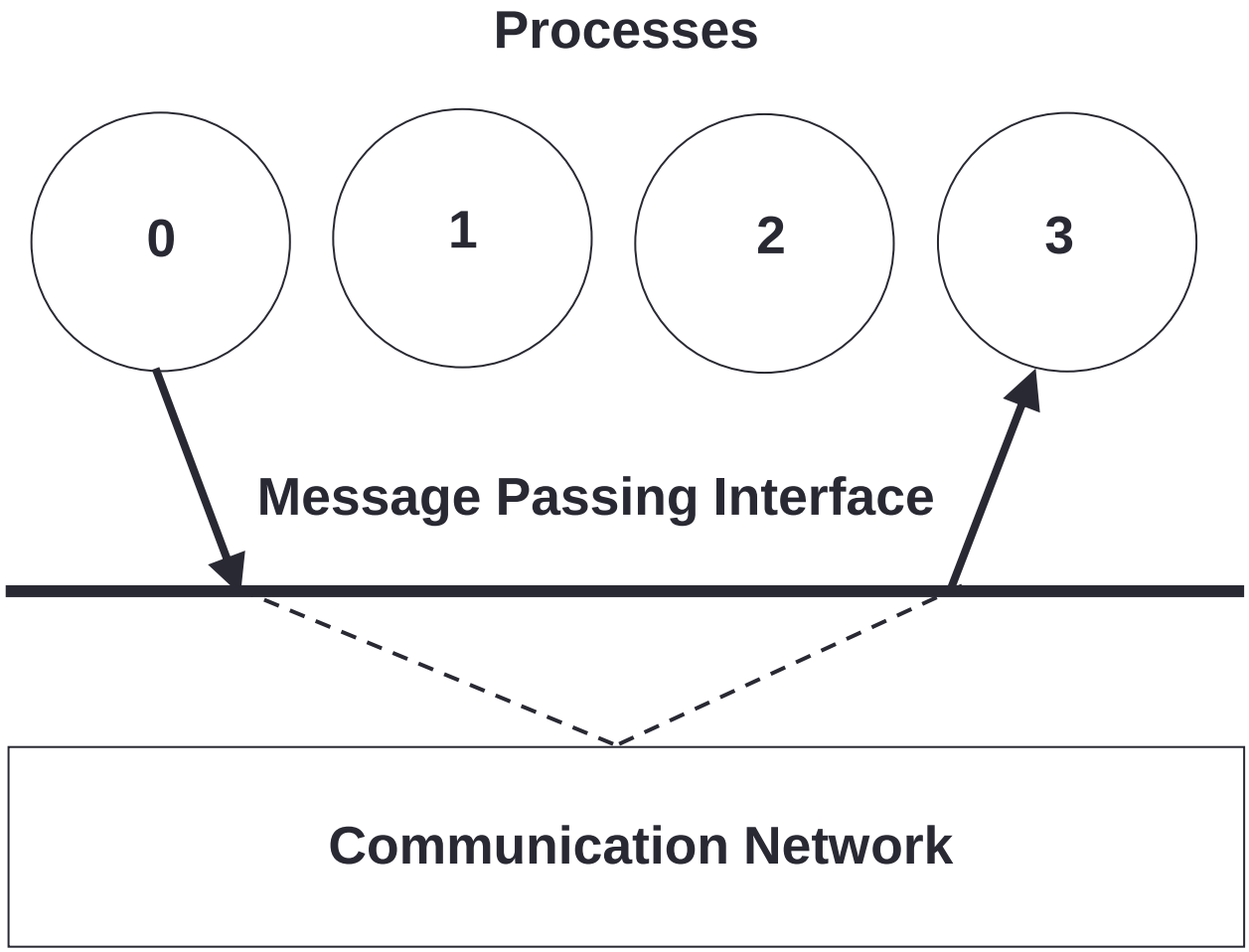
Message Passing Model

- The message passing model is based on the notion of processes
  - can think of a process as an instance of a running program, together with the program's data
- In the message passing model, parallelism is achieved by having many processes co-operate on the same task
- Each process has access only to its own data
  - ie all variables are private
- Processes communicate with each other by sending and receiving messages
  - typically library calls from a conventional sequential language

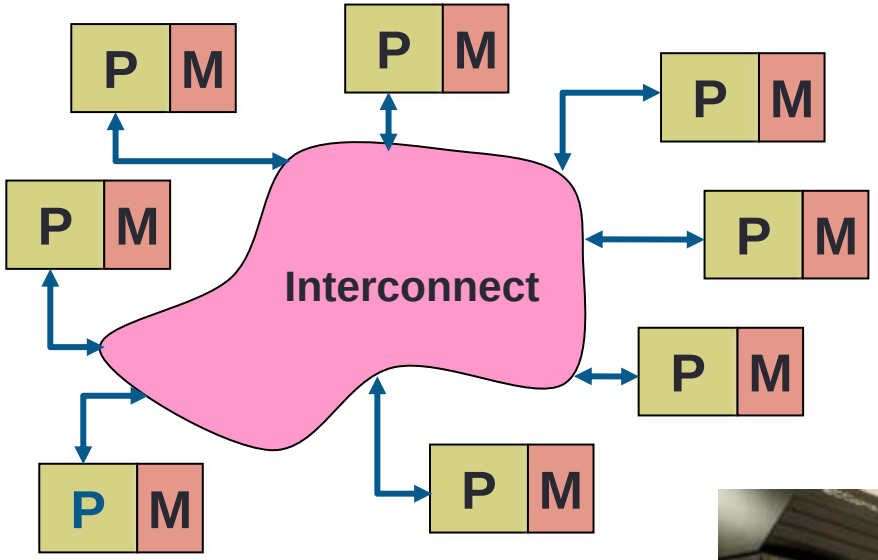
Sequential Paradigm



Parallel Paradigm



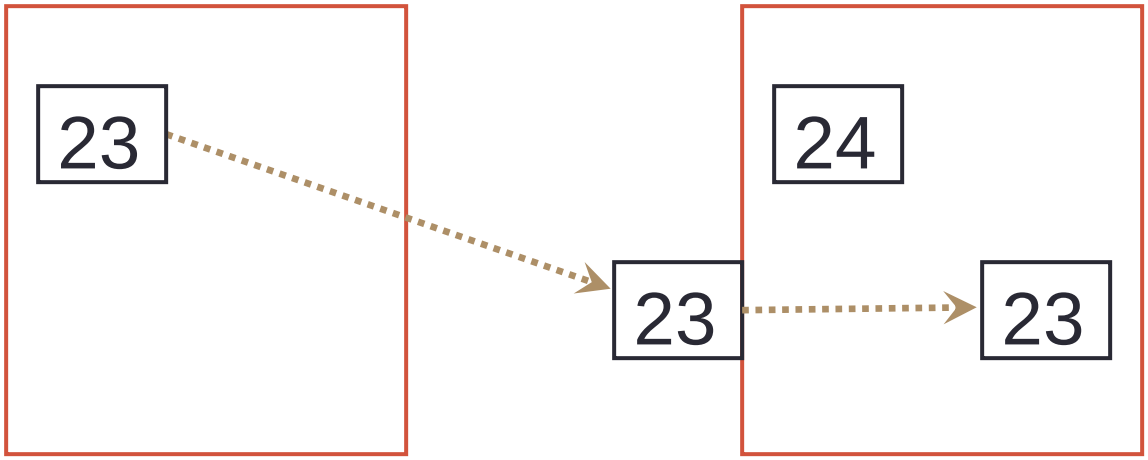
Distributed-Memory Architectures



Process Communication

	Process 1	Process 2
Program	`a=23`	`Recv(1,b)`
	`Send(2,a)`	`a=b+1`

Data



SPMD

- Most message passing programs use the Single-Program-Multiple-Data (SPMD) model
- All processes run (their own copy of) the same program
- Each process has a separate copy of the data
- To make this useful, each process has a unique identifier
- Processes can follow different control paths through the program, depending on their process ID
- Usually run one process per processor / core

Emulating General Message Passing (C)

```
main (int argc, char **argv)
{
            if (controller_process)
            {
                  Controller( /* Arguments */ );
            }
                  else
            {
                  Worker     ( /* Arguments */ );
            }
}
```

Emulating General Message Passing (F)

```
PROGRAM SPMD
    IF (controller_process) THEN
            CALL CONTROLLER ( ! Arguments ! )
    ELSE
            CALL WORKER     ( ! Arguments ! )
    ENDIF
END PROGRAM SPMD
```

Messages

- A message transfers a number of data items of a certain type from the memory of one process to the memory of another process

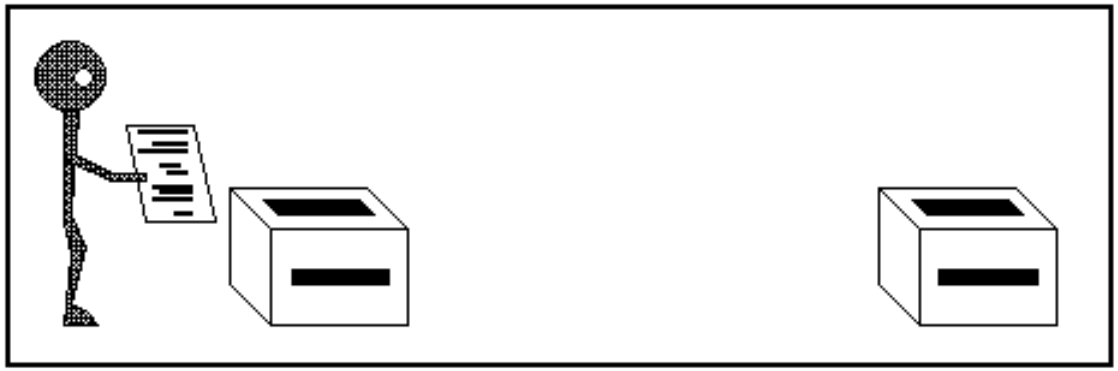
- A message typically contains
  - the ID of the sending processor
  - the ID of the receiving processor
  - the type of the data items
  - the number of data items
  - the data itself
  - a message type identifier

Communication modes

- Sending a message can either be synchronous or asynchronous
- A synchronous send is not completed until the message has started to be received
- An asynchronous send completes as soon as the message has gone
- Receives are usually synchronous - the receiving process must wait until the message arrives

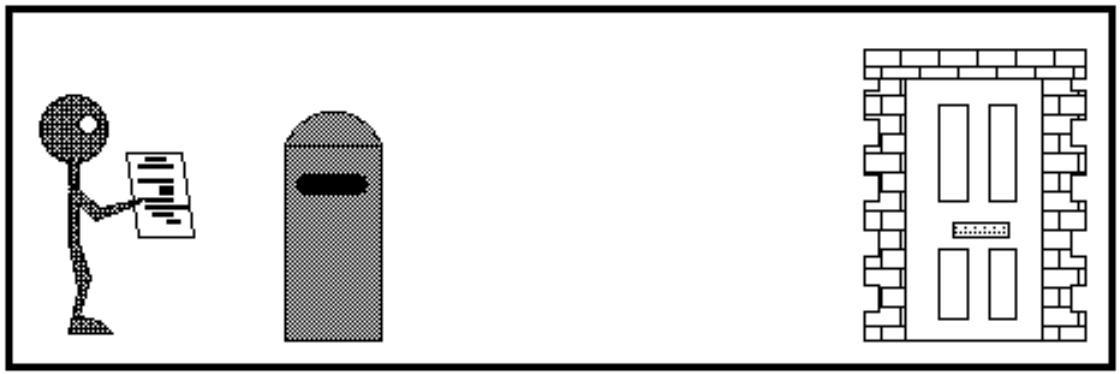
Synchronous send

- Analogy with faxing a letter.
- Know when letter has started to be received.



Asynchronous send

- Analogy with posting a letter.
- Only know when letter has been posted, not when it has been received.



Point-to-Point Communications

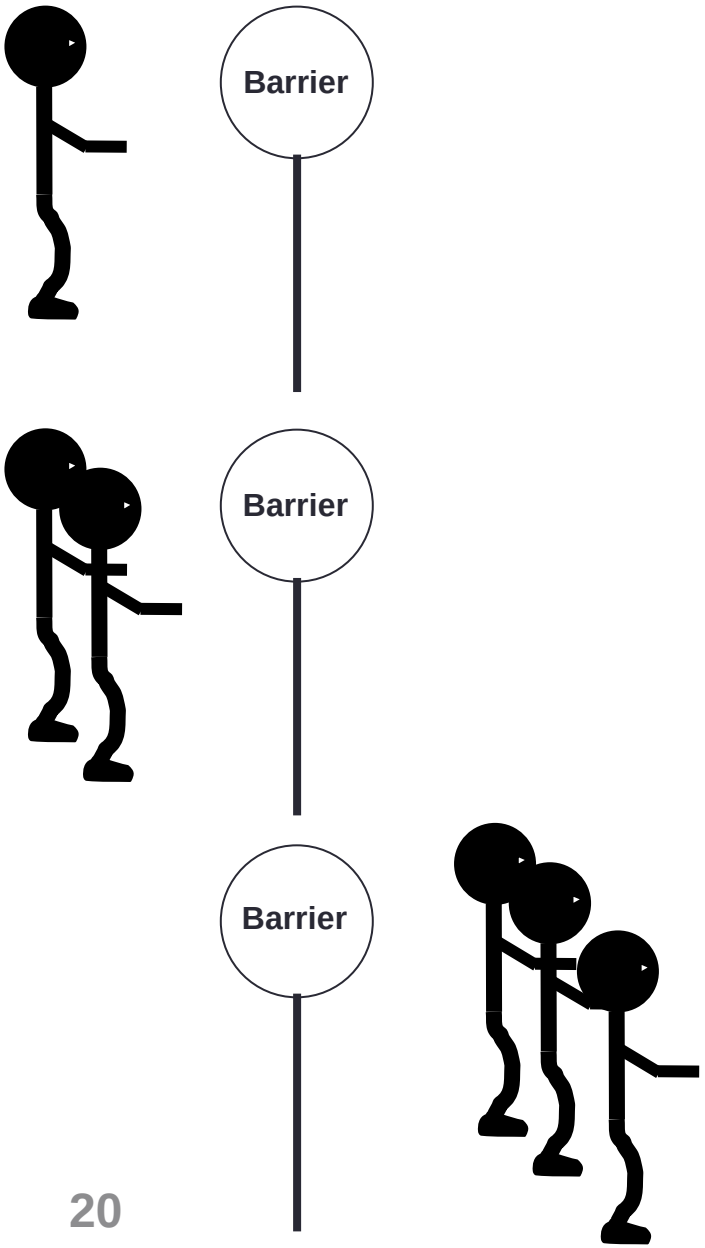
- We have considered two processes
  - one sender
  - one receiver
- This is called point-to-point communication
  - simplest form of message passing
  - relies on matching send and receive
- Close analogy to sending personal emails

Collective Communications

- A simple message communicates between two processes
- There are many instances where communication between groups of processes is required
- Can be built from simple messages, but often implemented separately, for efficiency

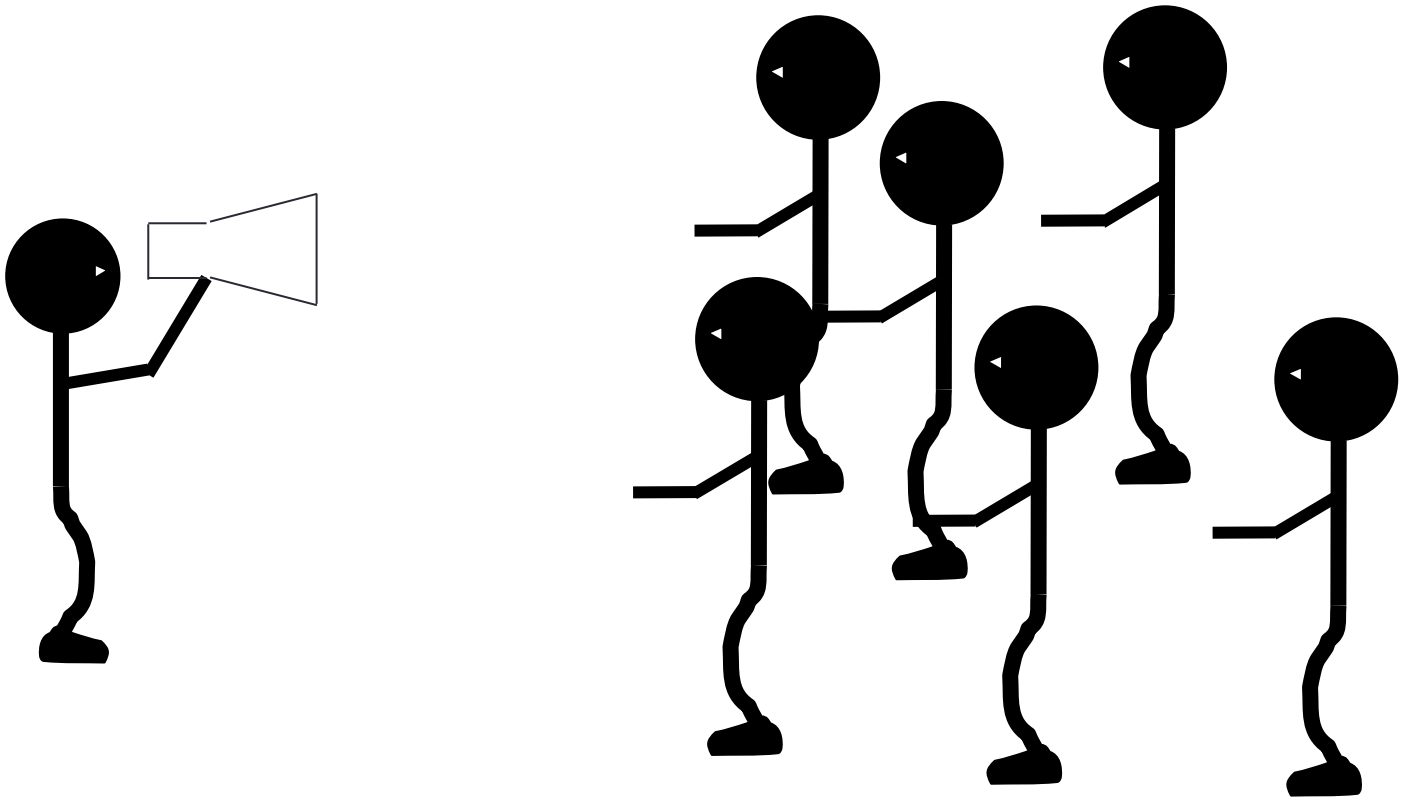
Barrier

- Global synchronisation



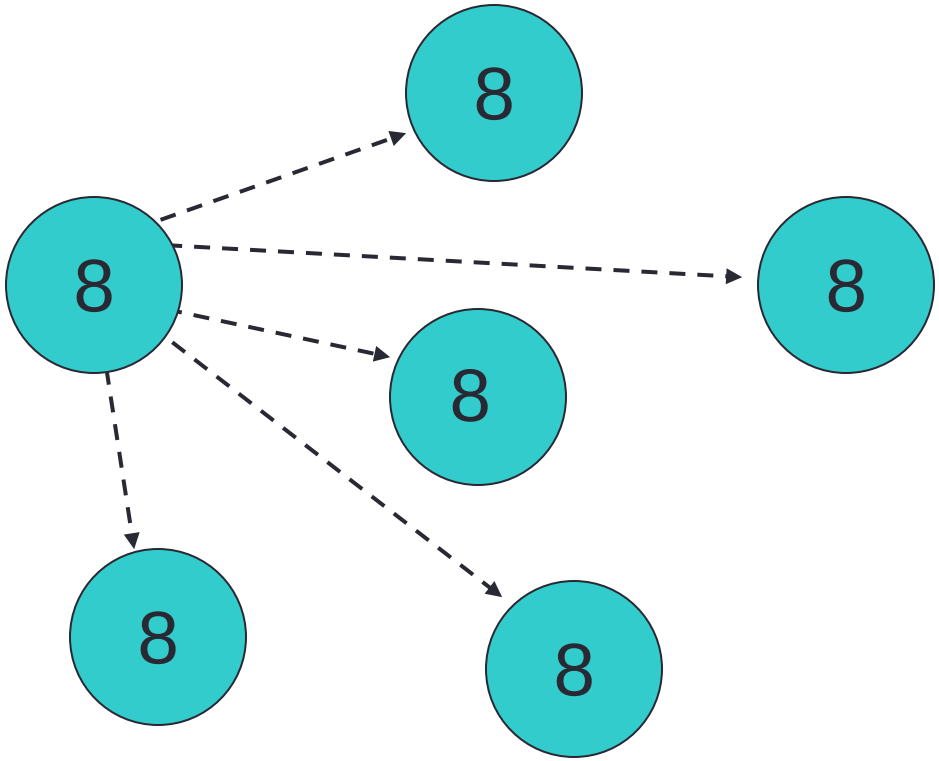
Broadcast

- One to all communication



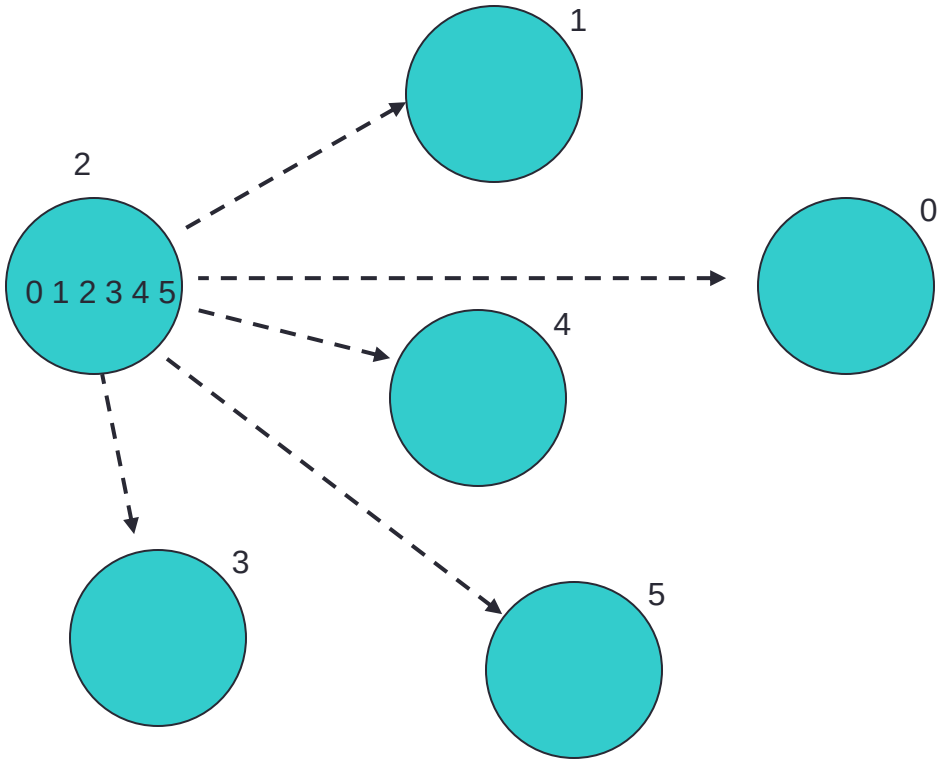
Broadcast

- From one process to all others



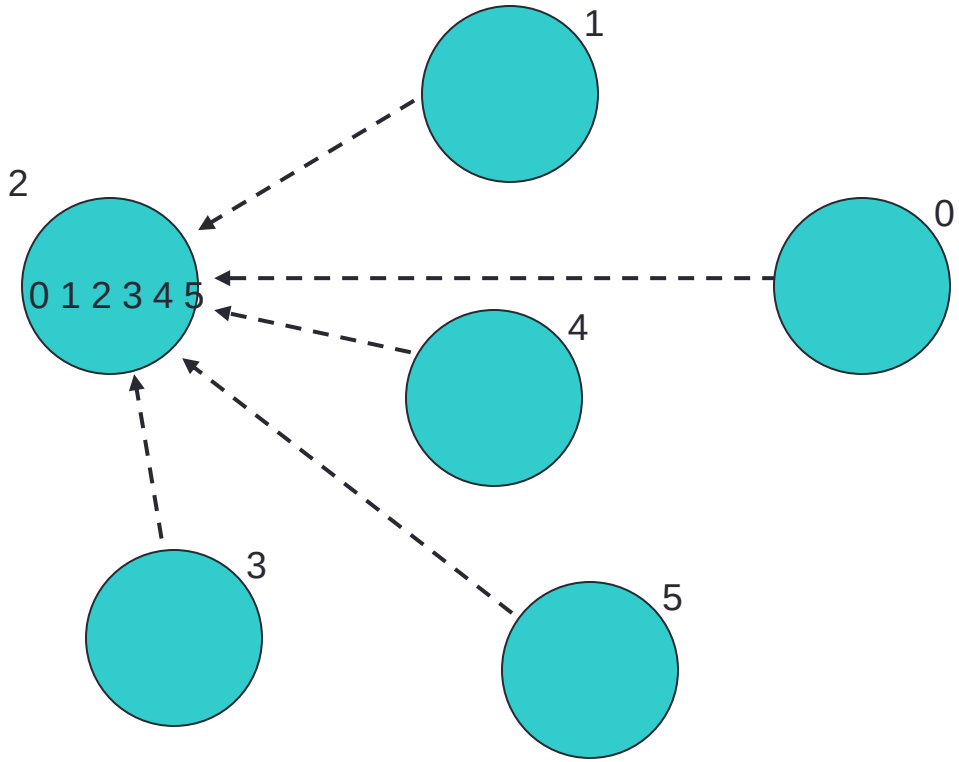
Scatter

- Information scattered to many processes



Gather

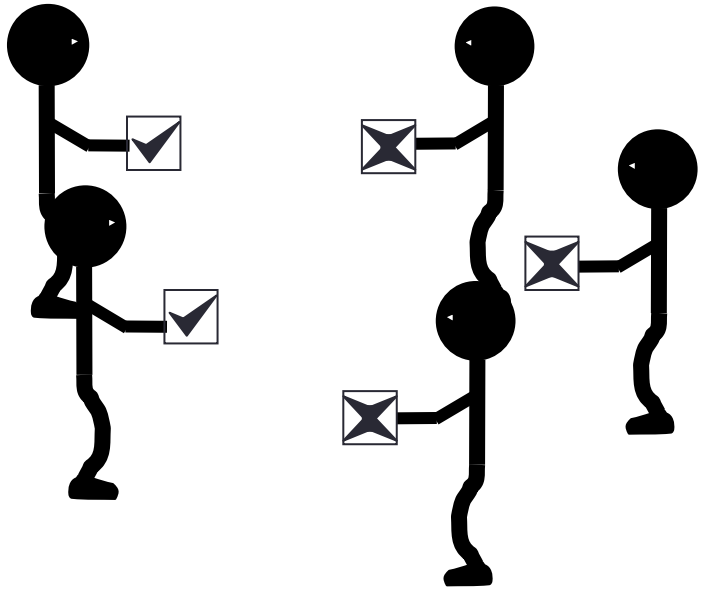
- Information gathered onto one process



Reduction Operations

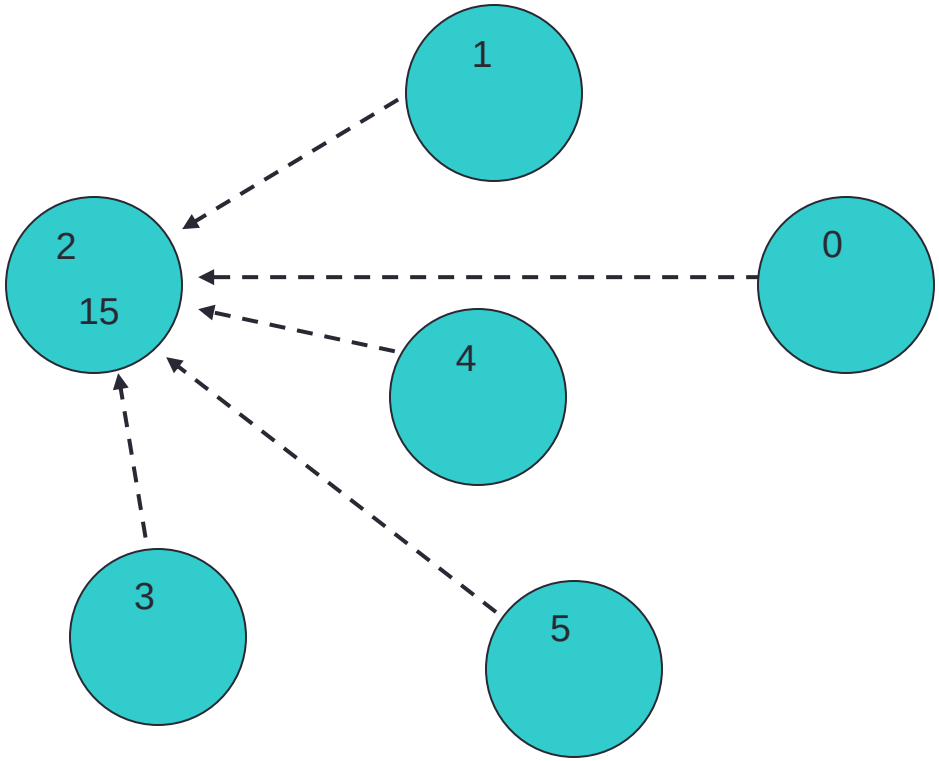
- Combine data from several processes to form a single result

Strike?



Reduction

- Form a global sum, product, max, min, etc.



Launching a Message-Passing Program

- Write a *single piece* of source code
  - with calls to message-passing functions such as send / receive
- Compile with a *standard compiler* and link to a *message-passing library* provided for you
  - both open-source and vendor-supplied libraries exist
- Run *multiple copies* of *same executable* on parallel machine
  - each copy is a separate *process*
  - each has its own private data completely distinct from others
  - each copy can be at a completely different line in the program
- Running is usually done via a launcher program
  - "please run *N* copies of my executable called *program.exe*"

Issues

- Sends and receives must match
  - danger of deadlock
  - program will stall (forever!)
- Possible to write very complicated programs, but …
  - most scientific codes have a simple structure
  - often results in simple communications patterns
- Use collective communications where possible
  - may be implemented in efficient ways

Summary (i)

- Messages are the *only* form of communication
  - all communication is therefore explicit
- Most systems use the SPMD model
  - all processes run exactly the same code
  - each has a unique ID
  - processes can take different branches in the same codes
- Basic communications form is point-to-point
  - collective communications implement more complicated patterns that often occur in many codes

Summary (ii)

- Message-Passing is a programming model
  - that is implemented by MPI
  - the Message-Passing Interface is a library of function/subroutine calls
- Essential to understand the basic concepts
  - private variables
  - explicit communications
  - SPMD
- Major difficulty is understanding the Message-Passing model
  - a very different model to sequential programming

```
if (x < 0)
   print("Error");
   exit;
```

epcc

THE UNIVERSITY OF EDINBURGH

HPC CARPENTRY

Hossein Kafiabad

School of Mathematics, University of Edinburgh

MPI Header

- C/C++:

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

- Fortran 77:

```
include 'mpif.h'
```

- Fortran 90:

```
use mpi
```

- Fortran 2008:

```
use mpi_f08
```

Python:
from mpi4py import MPI

but there is a caveat, see below

MPI Initialisation

- C:

```
int MPI_Init(int *argc, char ***argv)
```

- Fortran:

```
call MPI_INIT(IERROR)
INTEGER IERROR
```

- Must be the first MPI procedure called.
  - but multiple processes are already running before **MPI_Init**

Setting up and launching a Python mpi program

```
from mpi4py import MPI

## your code
```

```
import mpi4py.rc
mpi4py.rc.initialize = False
from mpi4py import MPI


MPI.Init()

## your code

MPI.Finalize()
```

```
mpiexec -n 2 python script.py

mpiexec --use-hwthread-cpus --oversubscribe -n 2 python script.py
```

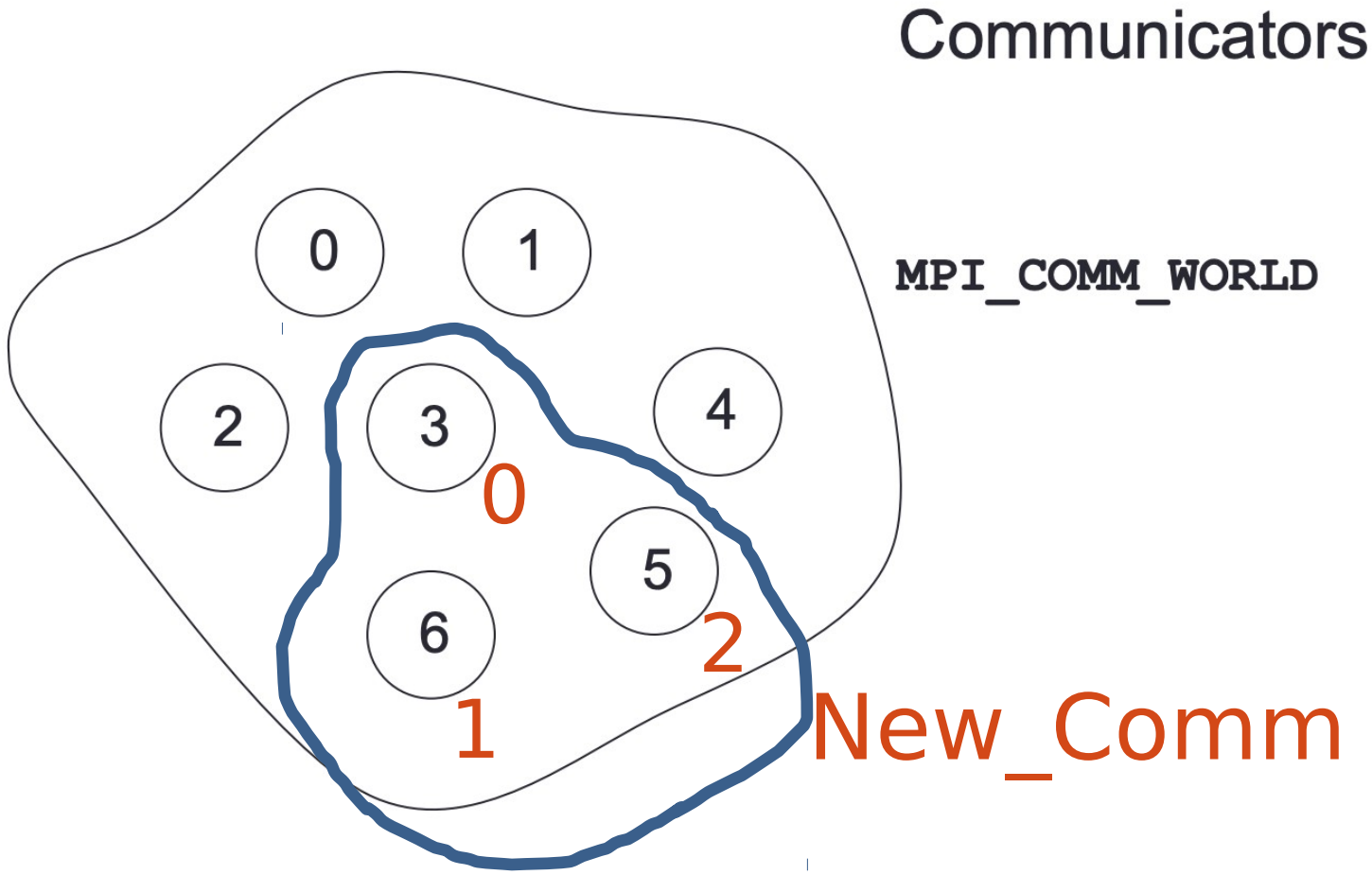
Example Launching Python with MPI

Code example ***hello_world.py.***

In folder ***MPI_lectures*** on Teams.

How many processes can we launch with the various options?

MPI Communicators



MPI Function Format

- C:

```
error = MPI_Xxxxx(parameter, ...);

MPI_Xxxxx(parameter, ...);
```

Python:
comm.Xxxx(parameter, ...) where comm is a communicator obj.
MPI_Xxxxx(parameter, ...) just as in C.

- Fortran:

```
CALL MPI_XXXXX(parameter, ..., IERROR)
```

  - **IERROR** optional in 2008 version *only*, otherwise *essential*

Rank

- How do you identify different processes in a communicator?
- The rank is not the physical processor number
  - numbering is always 0, 1, 2, …., N-1

```
MPI_Comm_rank(MPI_Comm comm, int *rank)
```

```
call MPI_COMM_RANK(COMM, RANK, IERROR)
     INTEGER COMM, RANK, IERROR
```

Python: comm.Get_rank( )

Size

- How many processes are contained within a communicator?

```
MPI_Comm_size(MPI_Comm comm, int *size)
```

```
call MPI_COMM_SIZE(COMM, SIZE, IERROR)
     INTEGER COMM, SIZE, IERROR
```

Python: comm.Get_size( )

Exiting MPI

- C:

```
int MPI_Finalize()
```

- Fortran:

```
call MPI_FINALIZE(IERROR)
INTEGER IERROR
```

Python: MPI_Finalize( )

Aborting MPI

- Aborting the execution from any processor (all the processors stop)
  - Good to use when a serious error occurs and there is no point in continuing the computation and wasting the allocation
  - Examples: there is inconsistency in data that is read, the numerical method is diverging, one of the processors found the final solution (e.g. in optimisation).
- C:

```
int MPI_Abort(MPI_Comm comm,int errorcode)
```

- Fortran:

call
```
MPI_ABORT(COMM, ERRORCODE, IERROR)
INTEGER COMM, ERRORCODE, IERROR
```

Python: comm.Abort( )

Collective Communication

- Collective action over a communicator: so a lot of processes are doing something together
- In contrast with point-to-point communication: just two processes send and receive messages (we will cover this next week)
- Synchronisation may or may not happen

Barrier

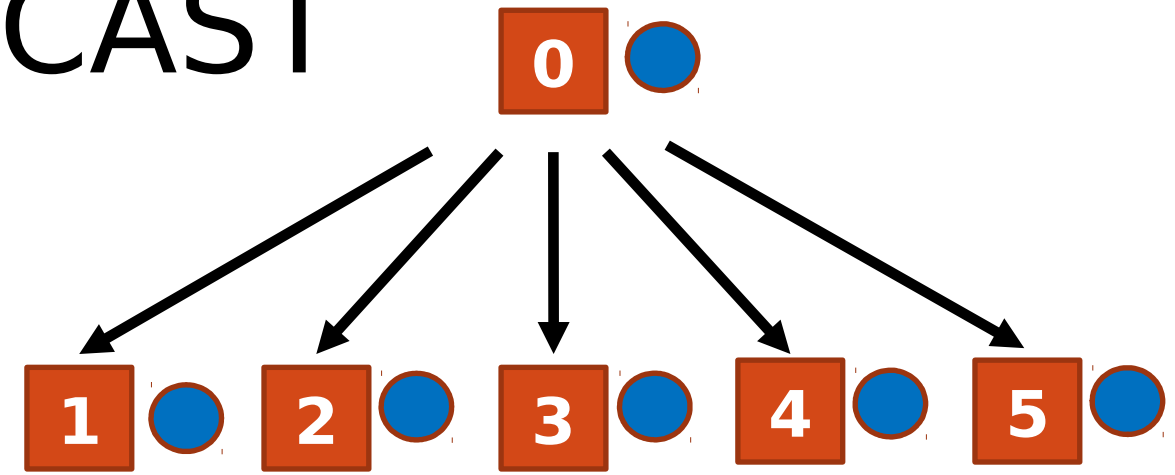
- C++: MPI_Barrier(MPI_Comm communicator)

- FORTRAN: MPI_BARRIER(COMM, IERROR)

  (going to drop the ‘call’ for FORTRAN functions)

Python: comm.Barrier( )

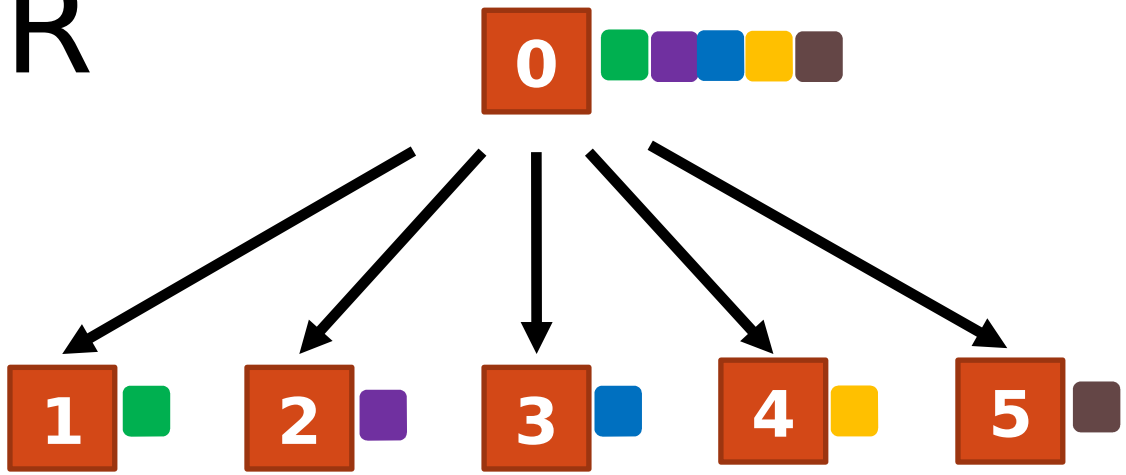
BROADCAST



- C++:
  MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
- FORTRAN:
  MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)

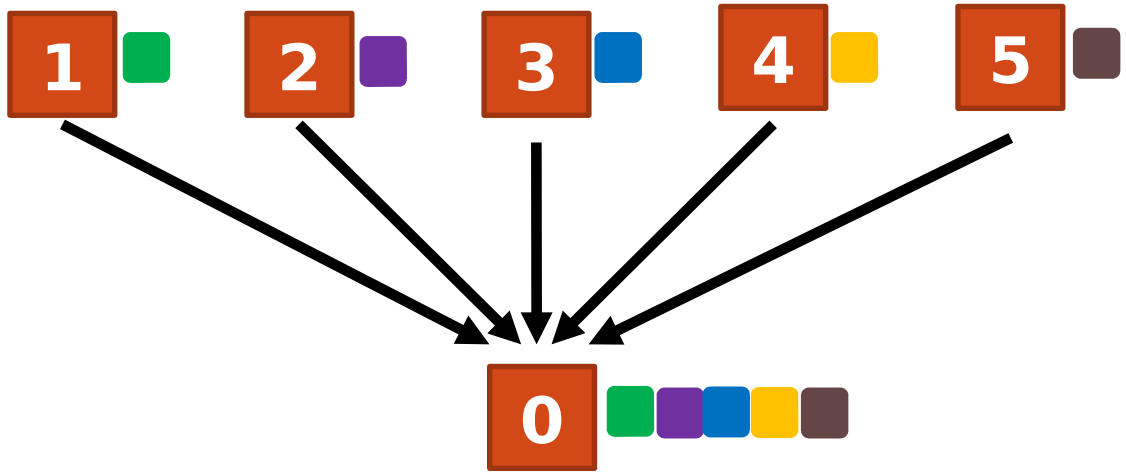
Python: comm.bcast(buffer, root) or comm.Bcast(buffer, root)

SCATTER



- C++:
  MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
- FORTRAN:
  MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR)
  Python:
  comm.scatter(buffer, root) or comm.Scatter(sendbuf, recvbuf, root)

GATHER

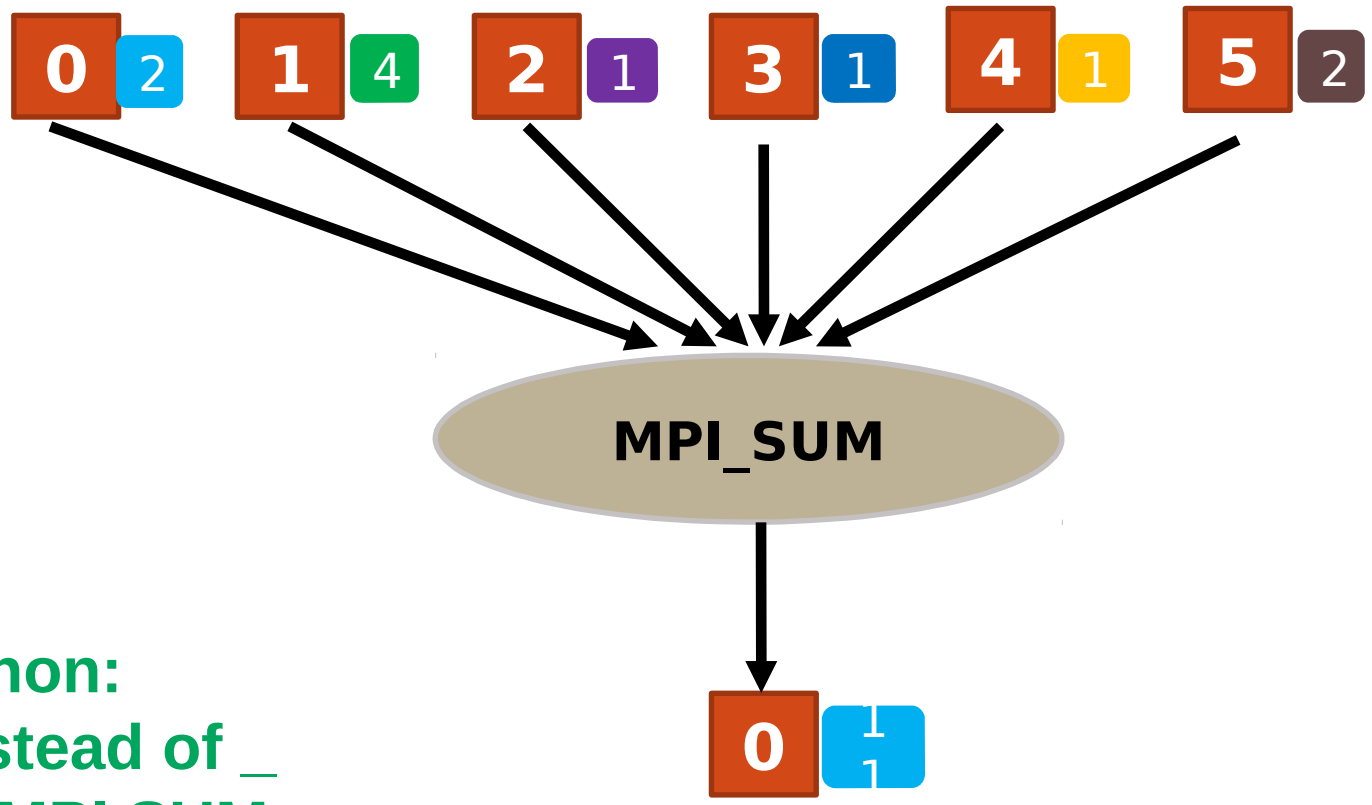


- C++:
  MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
- FORTRAN:
  MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR)

  Python:

  comm.gather(buffer, root) or comm.Gather(sendbuf, recvbuf, root)

REDUCE



- IMPORTRANT reduction operations:
  - MPI_SUM
  - MPI_MAX
  - MPI_MIN

Python:
. instead of _
eg. MPI.SUM

REDUCE

- C++:
  MPI_Reduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
- FORTRAN:
  MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR)

Python:
comm.reduce(buf, op, root) or
comm.Reduce(sendbuf, recvbuf, op, root)

Lowercase vs Uppercase Communication Methods

Lowercase:

- uses pickle under the hood.

- can transmit any Python object.

- easy to use:
  the received object is simply returned by the receiving mpi function and assigned to some generic Python variable.

- inefficient (due to pickle)

Example:

```
if rank == 0:
    data = [(i+1)**2 for i in range(size)]
else:
    data = None
data = comm.scatter(data, root=0)
```

Uppercase:

- can transmit data types that occupy contiguous memory, “buffer-like” objects.

- more difficult to use:
  the receiving end needs to prepare buffer of matching size and data type.

- more efficient (closely resembles Fortran and C/C++).

Example:

```
sendbuf = None
if rank == 0:
    sendbuf = np.empty([size, 100], dtype='i')
    sendbuf.T[:,:] = range(size)
recvbuf = np.empty(100, dtype='i')
comm.Scatter(sendbuf, recvbuf, root=0)
```

Contiguous Memory Object vs. Generic Python Object

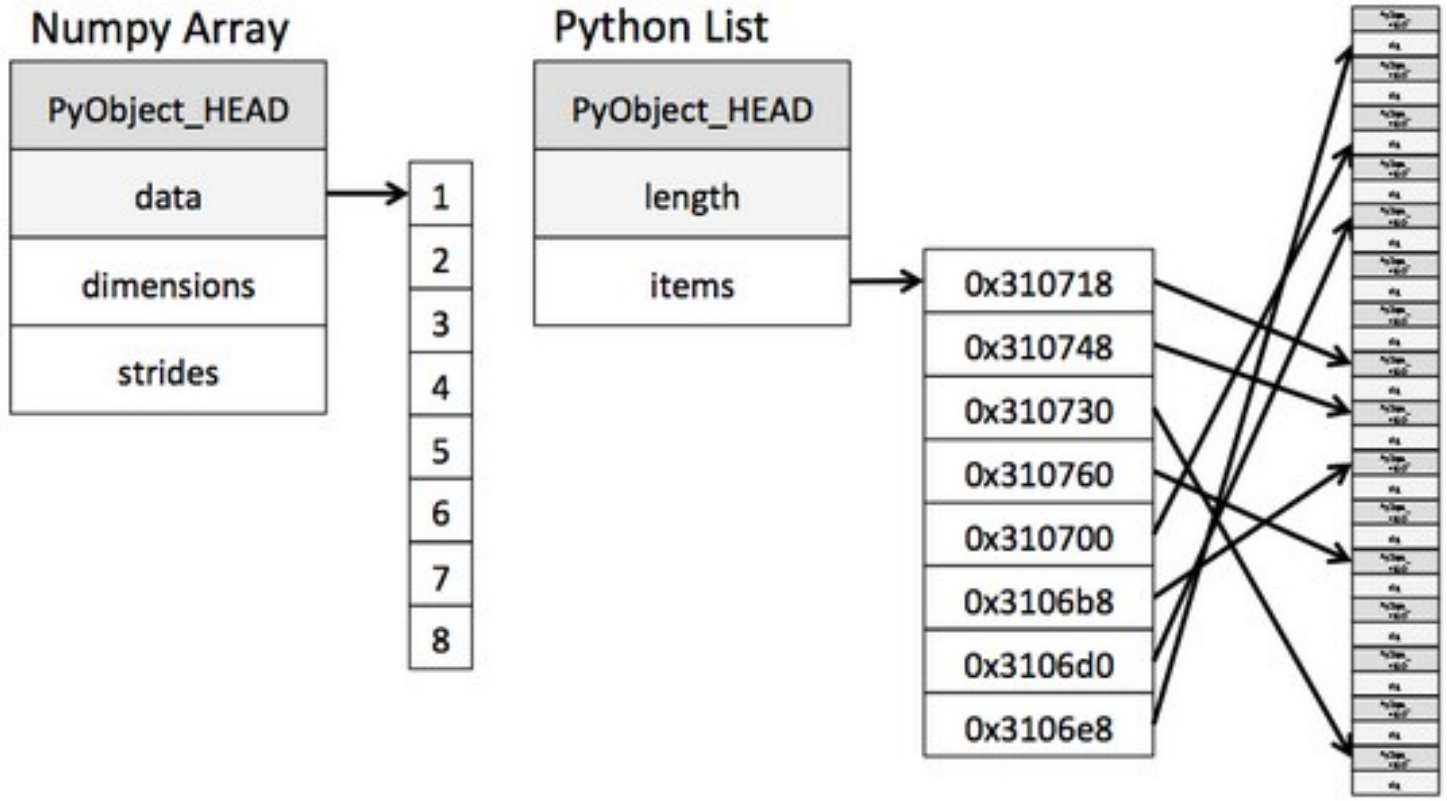


Image from
https://jakevdp.github.io/PythonDataScienceHandbook/02.01-understanding-data-types.html

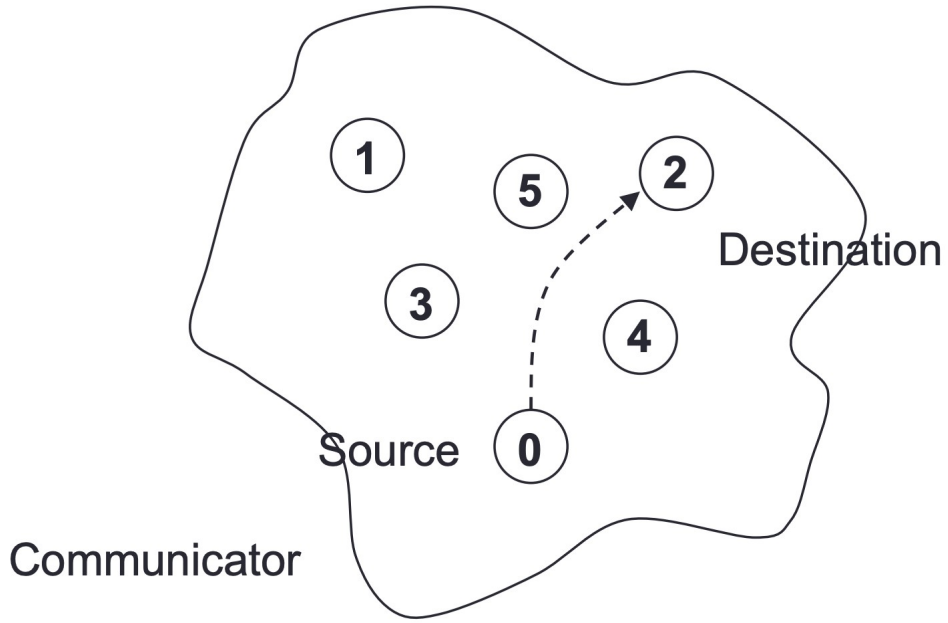
Example Collective Communication

Code example ***col_com.py***

In folder ***MPI_lectures*** on Teams.

POINT-TO-POINT COMMUNICATION

- Communication between two processes.
- Source process sends message to destination process.
- Communication takes place within a communicator.
- Destination process is identified by its rank in the communicator.



Send

- The sender (process) sends a message specified by
  - **Send buffer:** variable that contains the data
  - **Count**: how many (contiguous) memory blocks of that data type
  - **Destination:** which rank the data is going to be sent to
  - **Meta data (tag)** will talk about it later

- **C++:** int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
- **FORTRAN:**
  MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)

Python:

comm.send(buffer, dest, tag) or comm.Send(buffer, dest, tag)

Receive

- The receiver (process) receives a message **that is already sent** and is specified by
  - **Recv buffer:** variable that contains the data
  - **Count**: how many (contiguous) memory blocks of that data type are going to be stored
  - **Source:** which rank sent the data

- **C++:** int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
- **FORTRAN:**
  MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)

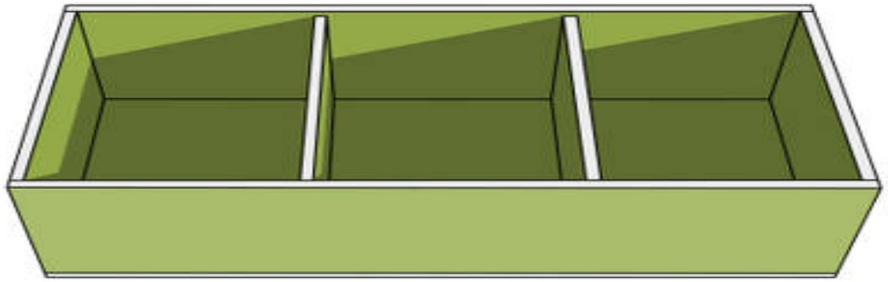
Python:

comm.recv(source, tag) or comm.Recv(buffer, source, tag)

Tags

- One (or several) sender(s) can send several messages to one receiver containing different data
- How distinguish between them: tags!

Sender



A B C

Reciever

Side Notes On P2P Communication

- **For a P2P communication to succeed:**
  - The length of the received message must be less than or equal to the length of the receive buffer.
  - Sender must specify a valid destination rank.
  - Receiver must specify a valid source rank.
  - The communicator must be the same.
  - Tags must match.
  - Message types must match.

Example P2P Communication

Code example ***p2p_com.py***

In folder ***MPI_lectures*** on Teams.

Many topics left to study...

blocking vs. nonblocking communication, topologies of processes, error management ....

but we should be fine for now :)

References

Official mpi4py documentation
https://mpi4py.readthedocs.io/en/stable/intro.html

EPCC Resource (for Fortran and C):
Writing Message Passing Parallel Programs with MPI
see Teams folder.

And of course....
https://stackoverflow.com