Introduction to MPI

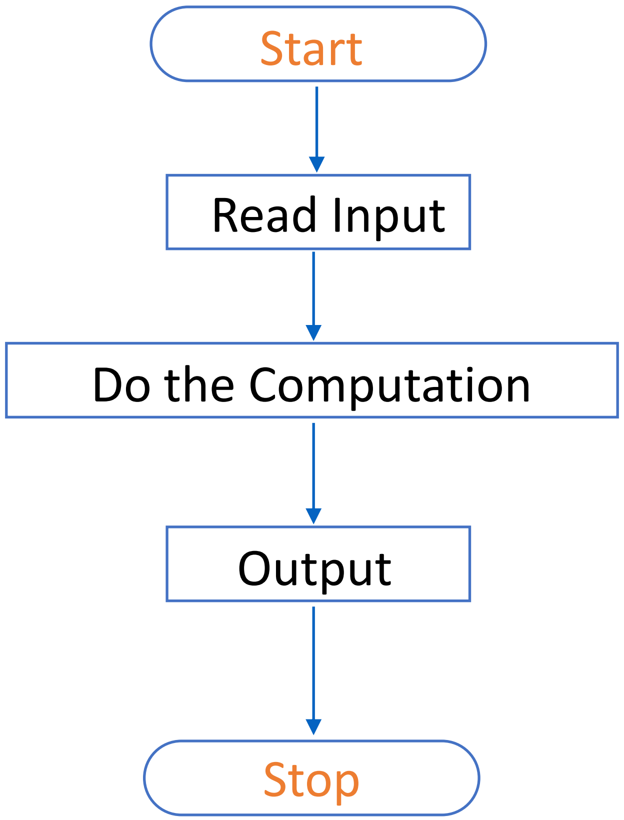
Sandeep Agrawal
C-DAC Pune

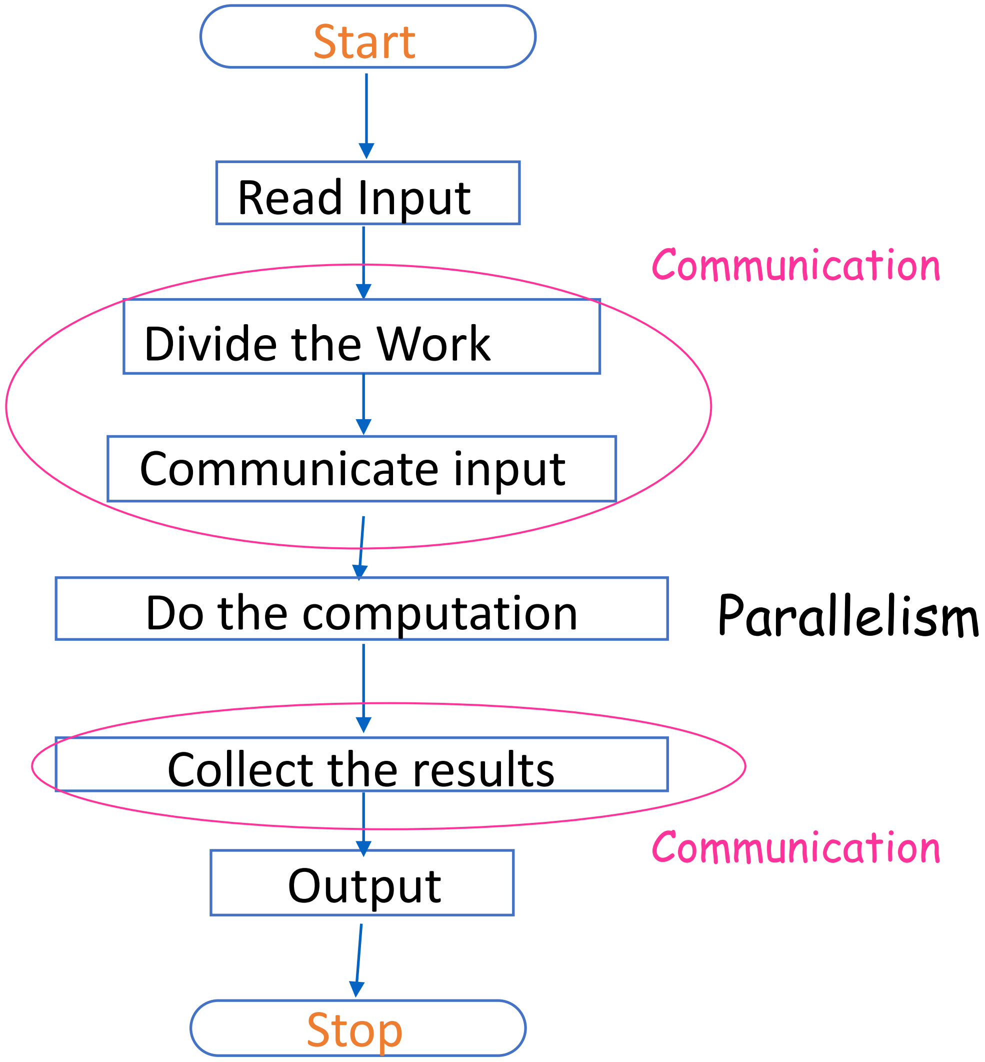
Plan

- General concepts
- About Message Passing Model
- What is MPI
- MPI Point-to-Point Communication

Serial vs Parallel Flow

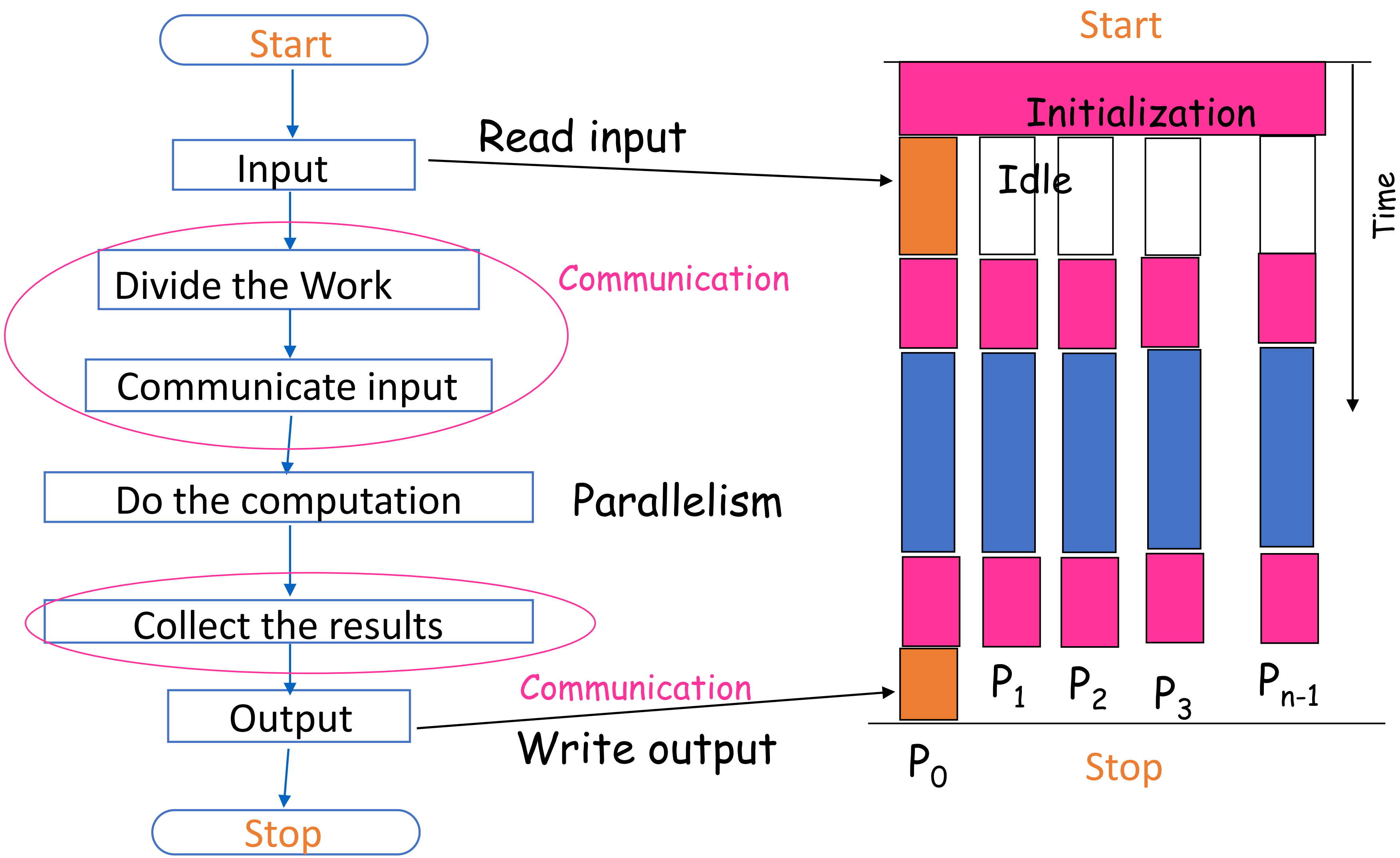
Compute $\sum x_i$ for million elements stored in a file





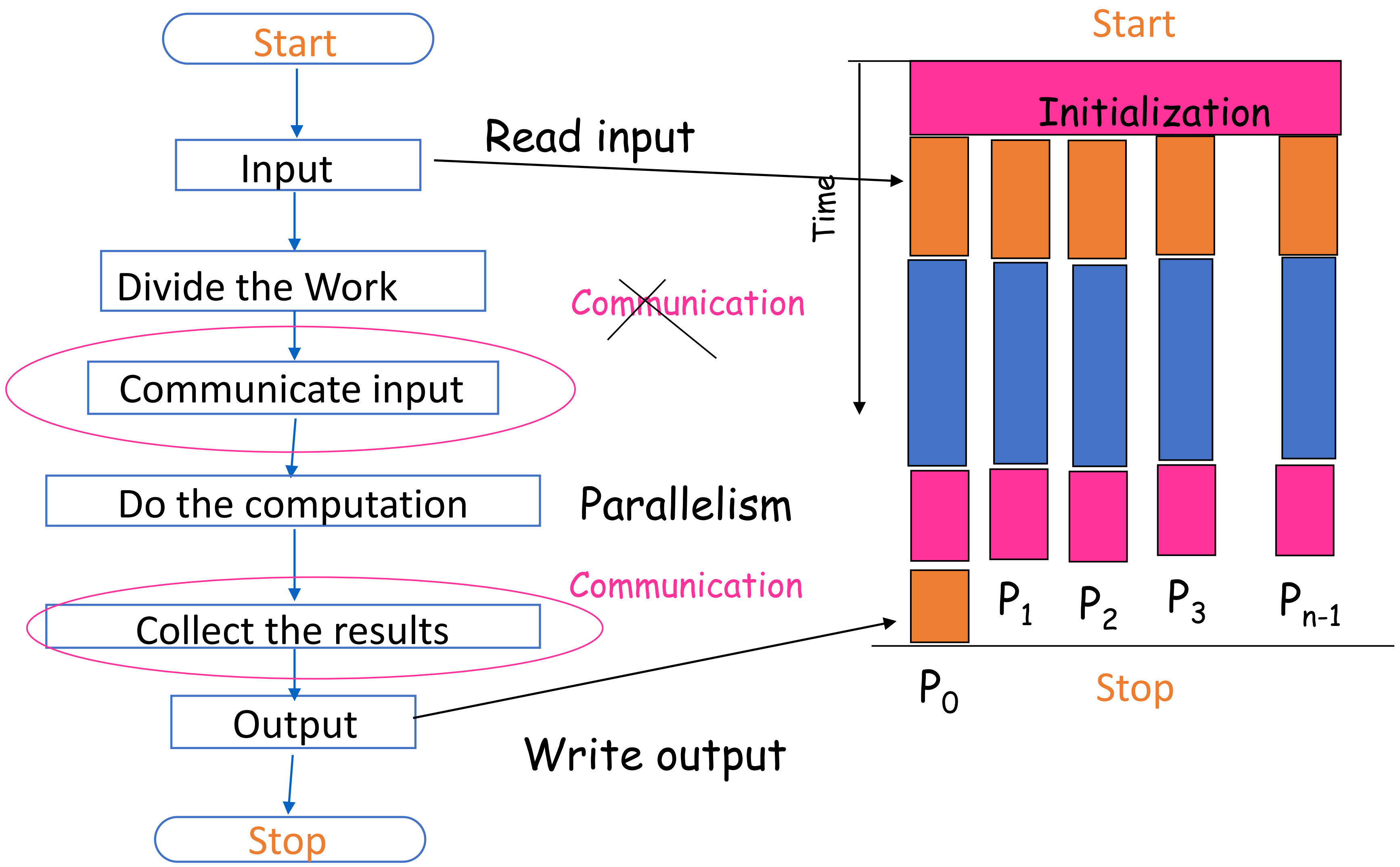
Serial vs Parallel Flow

Option 1: Read and Distribute inputs



Serial vs Parallel Flow

Option 2: Read inputs simultaneously



Performance Metrics

1. Speed up $= \dfrac{\text{Time for sequential code}}{\text{Time for parallel code}}$

$$S_P = \frac{Ts}{Tp} \qquad 1 \leq S_P \leq P$$

2. Efficiency $E_P = \dfrac{Sp}{p}$ $\qquad 0 < E_P < 1$

$E_P = 1 \Rightarrow S_P = P$ 100% efficient

Amdahl's Law

$$S = \frac{1}{f + (1-f) / P}$$

f = Sequential part of the code

Example f = 0.1 assume P = 10 processes

$$S = \frac{1}{0.1 + (0.9) / 10}$$

$$= \frac{1}{0.1 + (0.09)} \cong 5$$

As $P \longrightarrow \infty$ $S \longrightarrow 10$

Whatever we do, 10 is the maximum speedup possible

Communication Overheads

Latency

Startup time for each message transaction e.g. 1 μs

Bandwidth

The rate at which the messages (data size) are transmitted across the nodes
e.g. 100 Gbits / sec.

Plan

- General concepts
- **About Message Passing Model**
- What is MPI
- MPI Point-to-Point Communication

Characteristics of Message Passing Model

- Separate memory address spaces for every process
- Explicit interaction among processes
- Explicit data and work allocation by user to processes
- Asynchronous parallelism

How Message Passing Model Works

- A parallel computation consists of a number of processes
- Each process has purely local variables
- No mechanism for any process to directly access memory of another
- Sharing of data among processes is done by explicitly message passing
- Data transfer requires cooperative operations by concerned processes
- Different processes need not be running on different processors

Usefulness of Message Passing Model

- Extremely general model
- Essentially, any type of parallel computation can be cast in the message passing form
- Can be implemented on wide variety of platforms, from networks of workstations to even single processor machines
- Generally allows more control over data location and flow within a parallel application than in, for example the shared memory model
- High performance

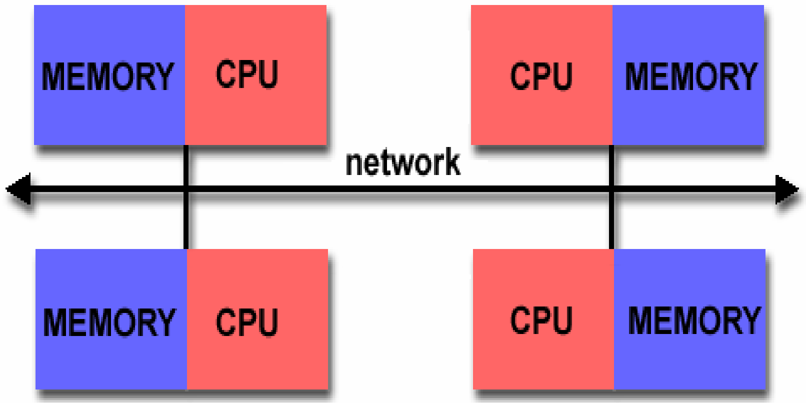
Plan

- General concepts
- About Message Passing Model
- **What is MPI**
- MPI Point-to-Point Communication

What is MPI ?

MPI stands for Message-Passing Interface

- MPI (Message-Passing Interface) is a message-passing library interface specification
- MPI addresses primarily the message-passing parallel programming model, in which data is moved from the address space of one process to that of another process through cooperative operations on each process.
- Extensions to the classical message-passing model are provided in collective operations, remote-memory access operations, dynamic process creation, threads and parallel I/O
- Every major HPC vendor have their own implementation of MPI
- However, programs written in message-passing style can run on
  - Distributed or shared-memory multi-processors
  - Networks of workstations
  - Single processor systems

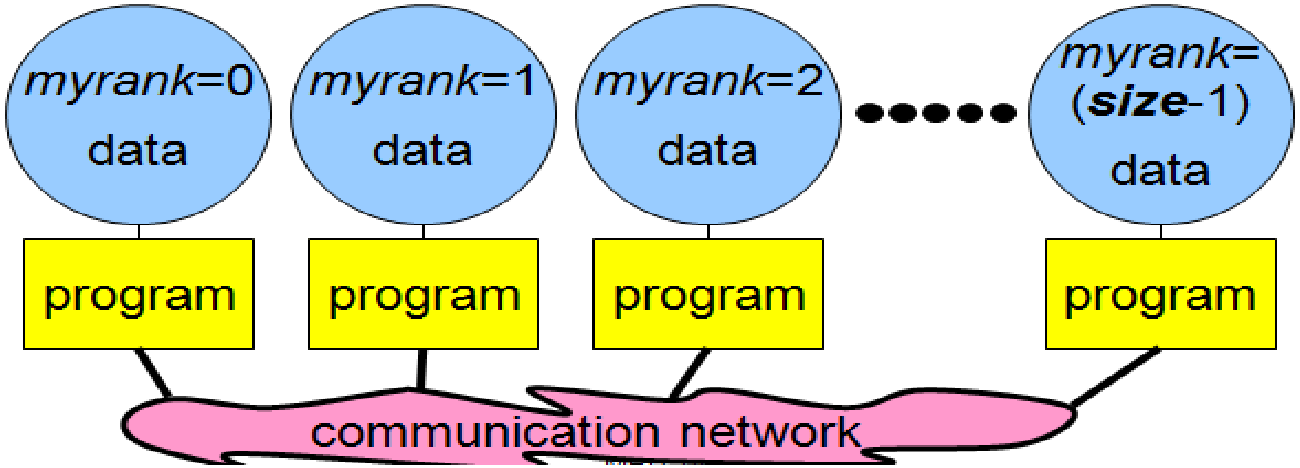


is MPI large or small ?

- MPI is Large (hundreds of functions)
    - Many features require extensive API
    - Complexity of use not related to number of functions
- MPI is small (6 basic functions)
    - All that’s needed to get started are only 6 functions
- MPI is just right !
    - Flexibility available when required
    - Can start with small subset

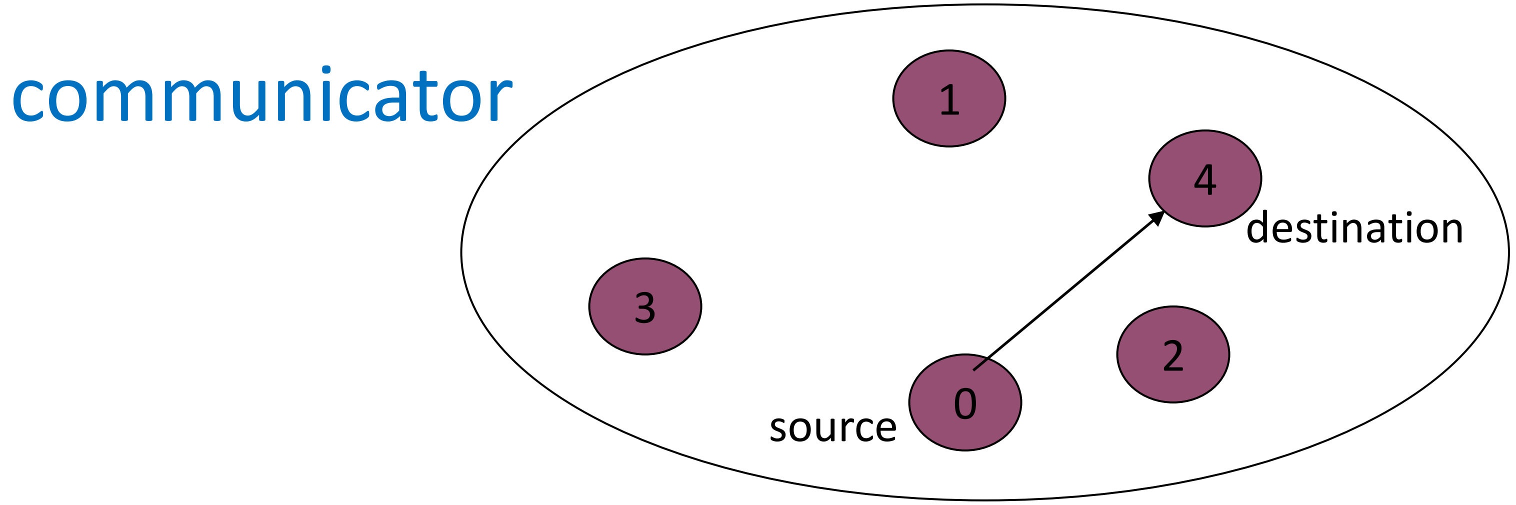
Data and Work Distribution

- Programmer imagines several processors, each with own memory, and writes a program to run on each processor
- To communicate amongst themselves MPI processes need unique identifiers: rank = identifying number
- all distribution decisions are based on the *rank*
  - like which process works on what part of data
  - which process works on what tasks



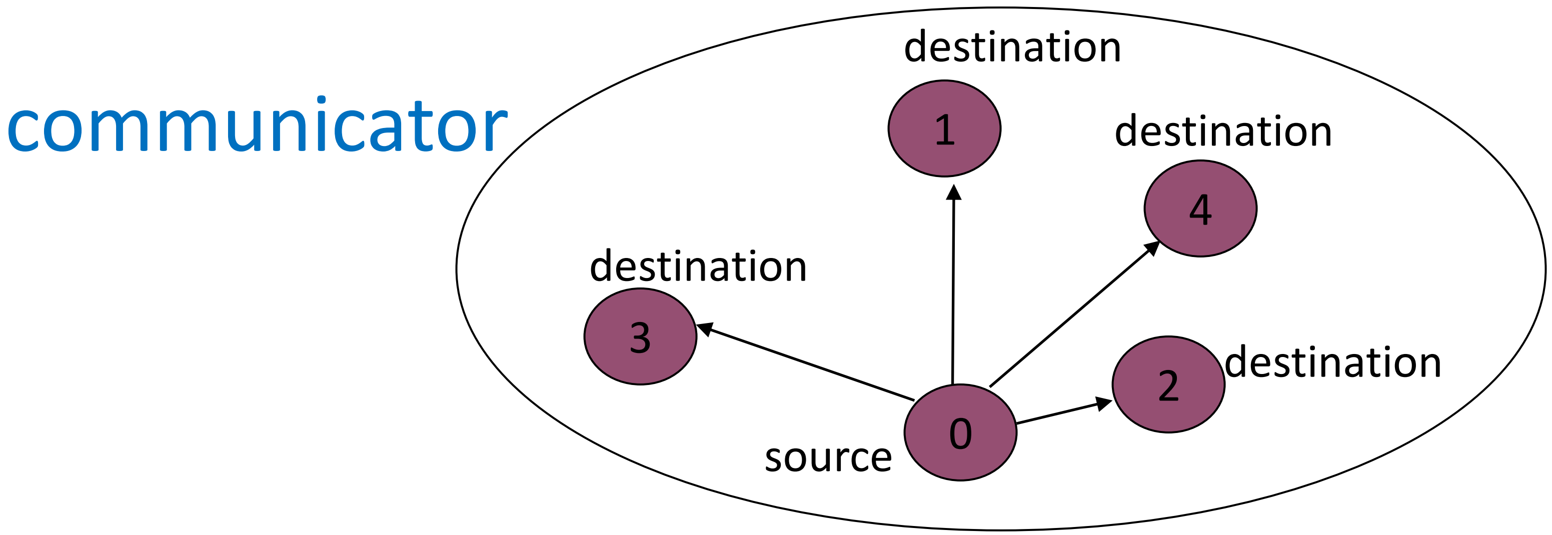
Point-to-Point Communication

- Communication between two processes
- Source process sends message to destination process
- Communication takes place within a communicator
- MPI_COMM_WORLD is default communicator



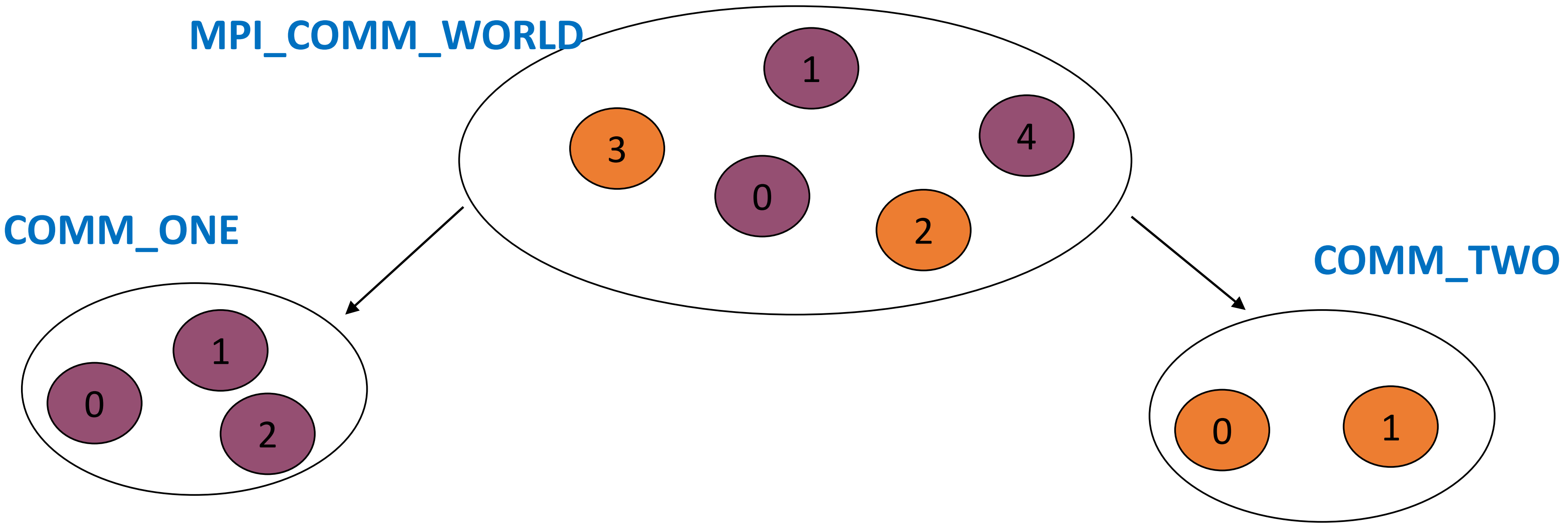
Collective Communication

- Communication between all processes
- A source process sends messages to or receives messages from all other processes
- Communication takes place within a communicator
- MPI_COMM_WORLD is default communicator



Communicators

- Is an object to handle a collection of processes
- Only processes within a communicator can talk among themselves
  - **ranks 0 to N-1**
- MPI_COMM_WORLD is default communicator containing all the processes
- User can create subsets of default communicators – overlapping or disjoint



Plan

- General concepts
- About Message Passing Model
- What is MPI
- **MPI Point-to-Point Communication**

Building blocks: Send and Recv

Basic operations in Message-passing programming paradigm are send and receive

Send (void *sendbuf, int num_elements, int dest)

Receive (void *recvbuf, int num_elements, int source)

Building blocks: Send and Recv (contd….)

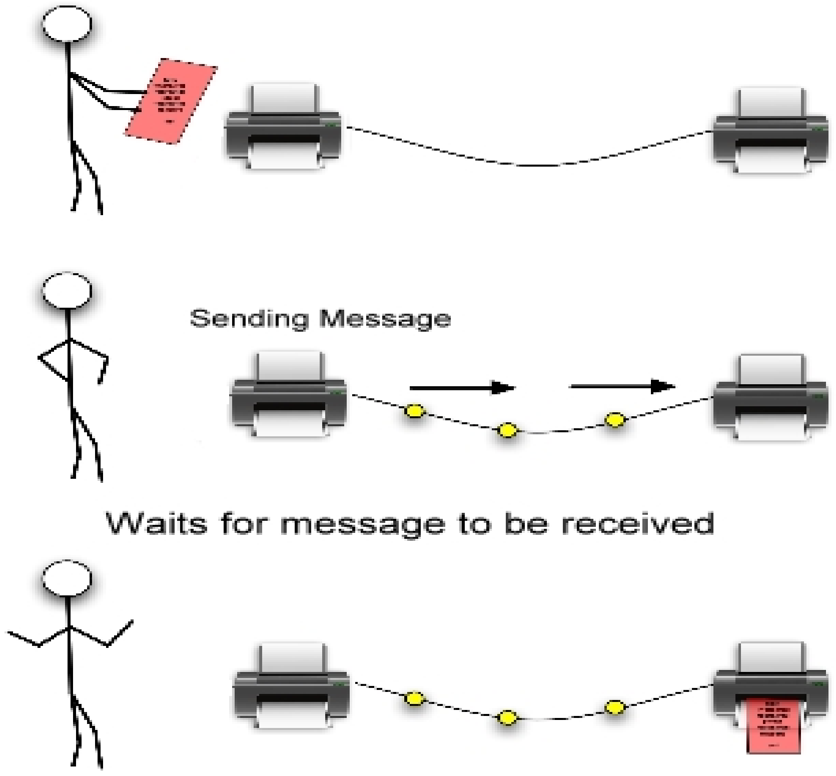
- “Completion” means that memory locations used in the message transfer can be safely accessed
  - **Sender side:** variable sent can be modified after completion
  - **Receiver side:** variable received can now be used

- MPI communication modes differ in what conditions on the receiving end are needed for completion

- Communication modes can be blocking or non-blocking
  - **Blocking:** return from function call implies completion
  - **Non-blocking:** routine returns immediately, completion to be tested for

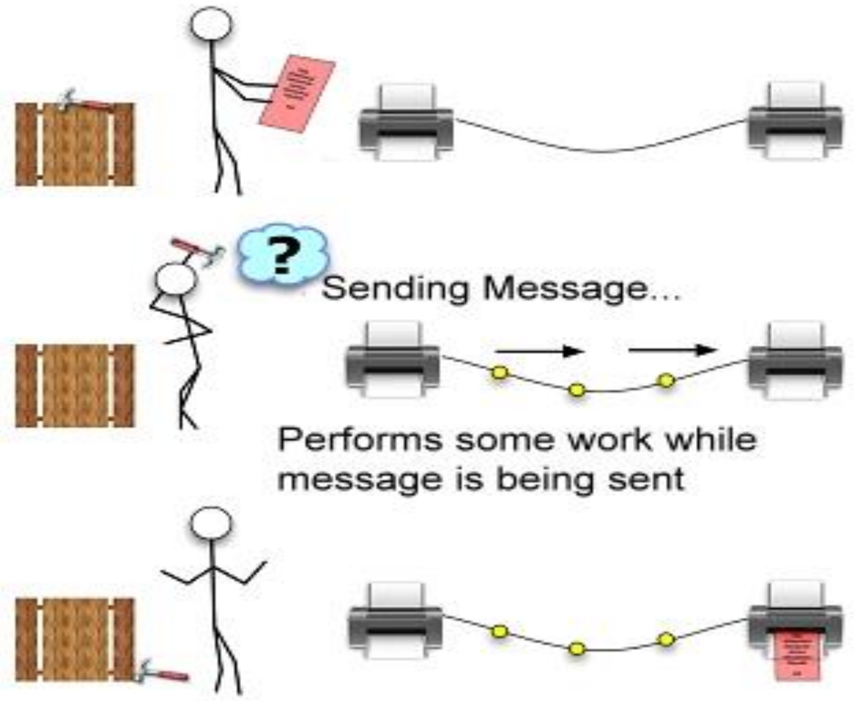
Blocking Operation

An operation that does not complete until the operation either succeeds or fails



Non-Blocking operation

An operation, such as sending or receiving a message, that returns immediately whether or not the operation was completed



Point to Point Communication

- Message is sent from a sending process to a receiving process. Only these two process need to know anything about the message.
- Message passing system provides following information to specify the message transfer
    - Which process is sending the message
    - Where is the data on the sending process
    - What kind of data is being sent
    - How much data is there
    - Which process is receiving the message
    - Where should the data be left on the receiving process
    - How much data is receiving process prepared to accept

General MPI Program Structure

MPI include file

.
.
.

Initialize MPI environment

.
.
.

Do work and make message passing calls

.
.
.

Terminate MPI Environment

Header files and calls format

- MPI constants, macros, definitions, function prototypes and handles are defined in a header file
- Required for all programs/routines which make MPI library calls

C (case sensitive):

```
# include "mpi.h"

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

Fortran (case unimportant):

```
include "mpif.h"

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

Starting With MPI Programming

- Six basic functions to start :

1. MPI_Init	Initialize MPI Environment
2. MPI_Finalize	Finish MPI Environment
3. MPI_Comm_rank	Get the process rank
4. MPI_Comm_size	Get the number of processes
5. MPI_Send	Send data to another process
6. MPI_Recv	Get data from another process

Initializing MPI

- MPI_Init is the first MPI routine called (only once)
- Initializes the MPI environment

C: int MPI_Init(int *argc, char ***argv)

Communicator Size

- How many processes are contained within a communicator?

C: MPI_Comm_size (MPI_Comm comm, int *size)

Process Rank

- Process ID number within the communicator
    - Starts with zero and goes to (n - 1) where n is the number of processes requested
- Used to identify the source and destination of messages, division of work among processes

C: MPI_Comm_rank(MPI_Comm comm, int *rank)

Exiting MPI

- Performs various clean-ups tasks to terminate the MPI environment.
- Always called at end of the computation.

C: MPI_Finalize()

Note : If any one process does not reach the finalization statement, the program will appear to hang.

Example program: hello_world.c

```
#include "mpi.h"
#include <stdio.h>
int main( argc, argv)
int argc; char **argv;
{
int rank, size;
MPI_Init( &argc, &argv );
MPI_Comm_rank(MPI_COMM_WORLD, &rank );
MPI_Comm_size(MPI_COMM_WORLD, &size );
/* Your code here */
printf("Hello world! I'm %d of %d\n", rank, size);
MPI_Finalize();
return 0;
}
```

Header File

Communicator

Initializing MPI

Rank

Size

Exiting MPI

How to Compile &Execute MPI Programs ?

To Compile : mpicc hello_world.c -o hello

mpif90 hello_world.f -o hello

To run with 4 processes : mpiexec -np 4 hello

Output

Hello world! I'm 2 of 4

Hello world! I'm 1 of 4

Hello world! I'm 3 of 4

Hello world! I'm 0 of 4

Note - Order of output is not specified by MPI

MPI Send

```
int MPI_Send( void *buf,          // Data To be sent
              int count,          // Total Data Elements to be sent
MPI_Datatype datatype,            // Datatype of the data to be sent
              int dest,           // Processor to which data is being sent
              int tag,            // To distinguish from diff types of msg
        MPI_Comm comm)            // Communicator
```

MPI Receive

```
int MPI_Recv(void *buf,        // Data To be Receive
             int count,        // Total Data Elements to be recv
MPI_Datatype datatype,         // Datatype of the data to be recv
             int source,       // Processor from where data is being sent
             int tag,          // To distinguish from diff types of msg
         MPI_Comm comm,        // Communicator
     MPI_Status *status)       // Status structure
```

Sending a Message

- int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
  - buf: starting address of the data to be sent
  - count: number of elements to be sent (not bytes)
  - datatype: MPI datatype of each element
  - dest: rank of destination process
  - tag: message identifier (set by user)
  - comm: MPI communicator of processors involved
- MPI_Send(data, 500, MPI_FLOAT, 5, 25, MPI_COMM_WORLD)

Receiving a Message

- int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
  - buf: starting address of buffer where the data is to be stored
  - count: number of elements to be received (not bytes)
  - datatype: MPI datatype of each element
  - source: rank of source process
  - tag: message identifier (set by user)
  - comm: MPI communicator of processors involved
  - status: structure of information about the message that is returned
- MPI_Recv(buffer, 500, MPI_FLOAT, 3, 25, MPI_COMM_WORLD, status)

Blocking Communication Functions

Mode	MPI Function
Standard send	MPI_Send
Synchronous send	MPI_Ssend
Buffered send	MPI_Bsend
Ready send	MPI_Rsend
Receive	MPI_Recv

Similar variants exist for non-blocking calls also

Wildcards

- Allow you to not necessarily specify a tag or source
  - Eg :MPI_ANY_SOURCE and MPI_ANY_TAG are wild cards

- Status structure is used to get wildcard values
  - The tag of a received message

    C : status.MPI_TAG

    Fortran : STATUS(MPI_TAG)
  - The source of a received message

    C : status.MPI_SOURCE
  - The error code of the MPI call

    C : status.MPI_ERROR

Message Datatype

- A message contains an array of elements or scalar element of some particular MPI datatype
- MPI datatypes:
  - Primitive/basic types e.g.

    C language - MPI_INT, MPI_DOUBLE, MPI_FLOAT, MPI_CHAR ...

    Fortran - MPI_INTEGER, MPI_DOUBLE_PRECISION, MPI_REAL, MPI_CHARACTER ...

  - Derived types

    - allows mixed data types, non-contiguous data to be sent in one message

    - derived types can be built from basic types

    - E.g.

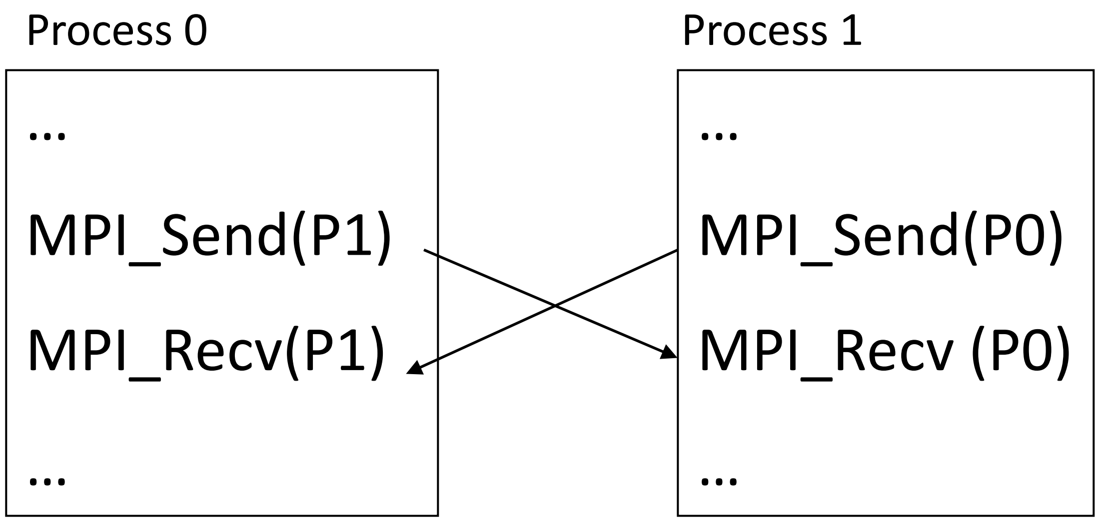
```
MPI_Datatype *mytype
MPI_Type_struct(...)
MPI_Type_commit(&mytype)
```

For a Communication to Succeed

- Sender must specify a valid destination rank
- Receiver must specify a valid source rank
- The communicator must be the same
- Tags must match
- Receiver’s buffer must be large enough
- User-specified buffer should be large enough (buffered send only)
- Receive posted before send (ready send only)

Deadlocks

- A deadlock occurs when two or more processors try to access the same set of resources
- Deadlocks are possible in blocking communication
  - Example: Two processors initiate a blocking send to each other without posting a receive



Thank YOU