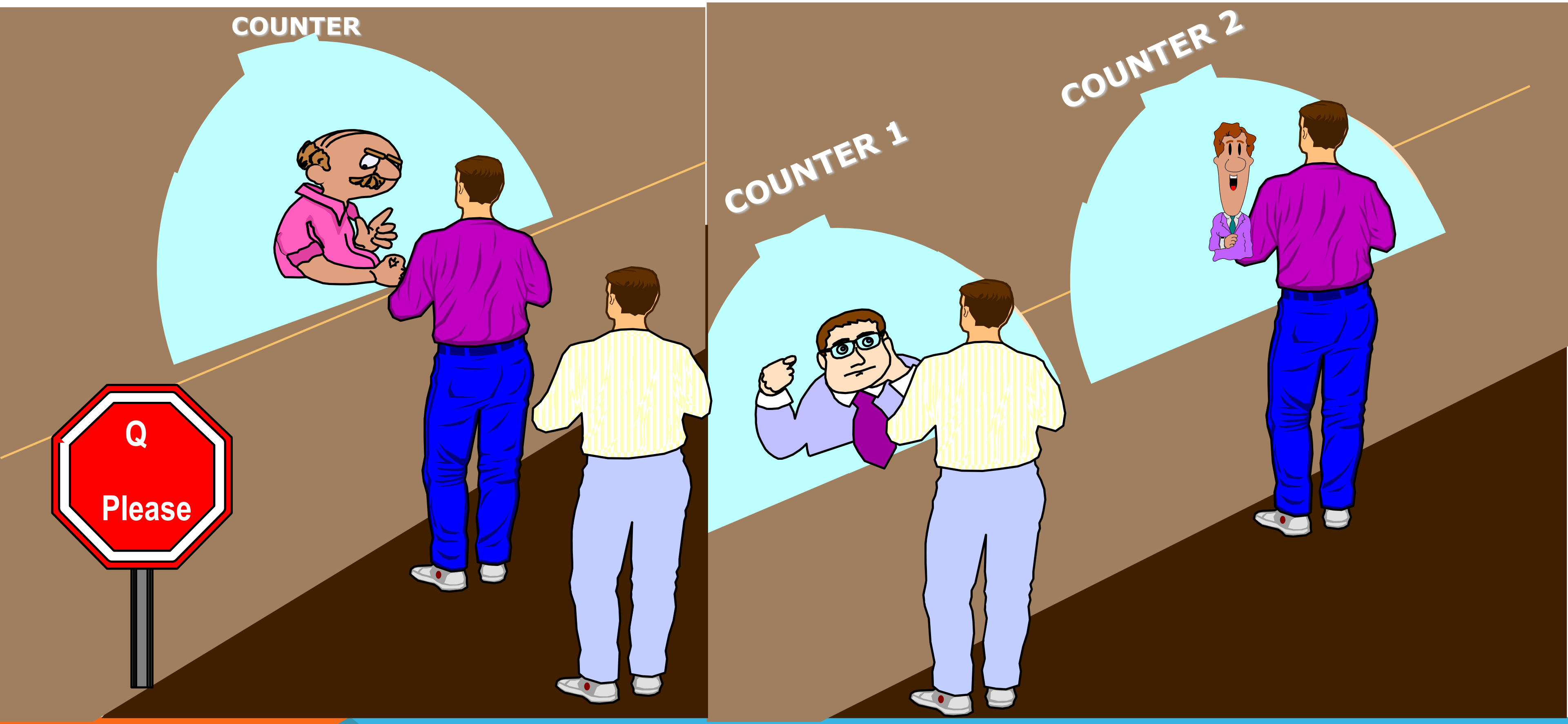
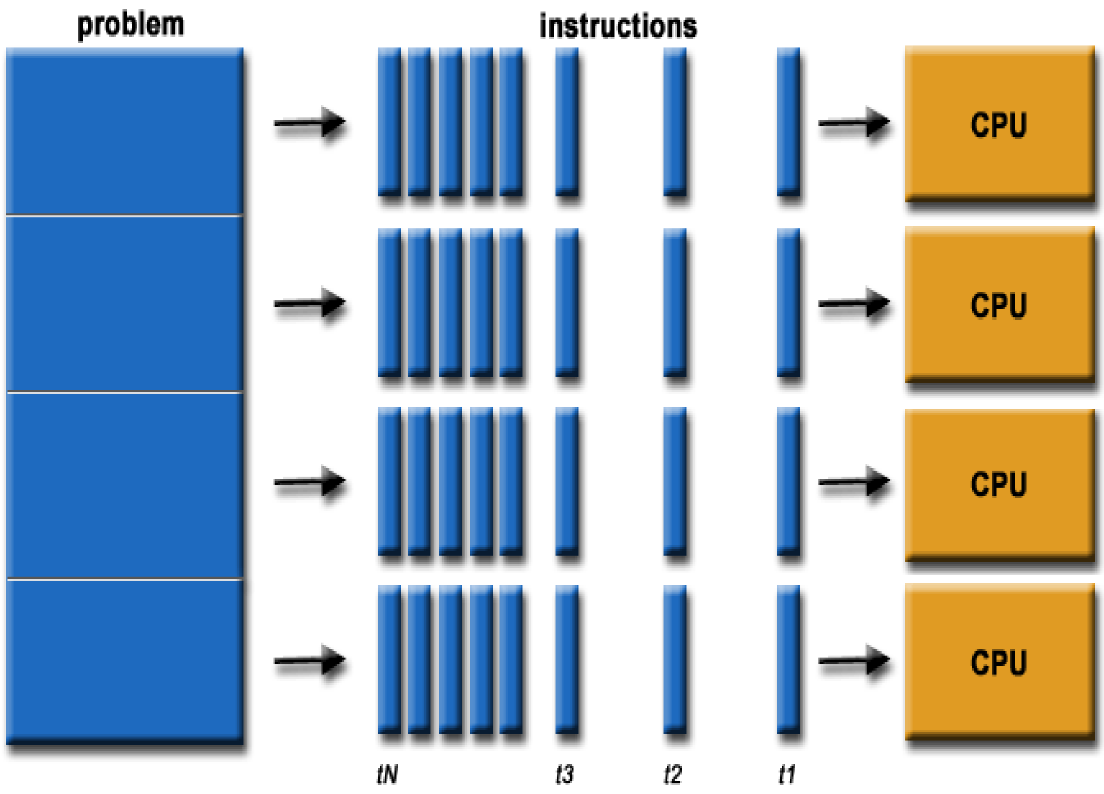
MPI POINT-TO-POINT COMMUNICATION

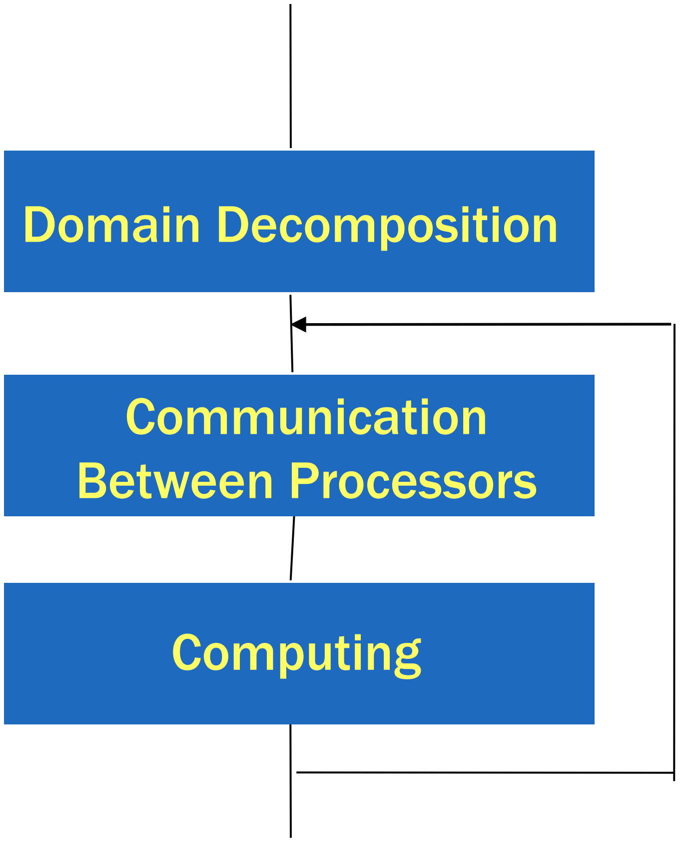
ARADDHANA DESHMUKH
C-DAC PUNE

SERIAL VS PARALLEL



PARALLEL COMPUTING





ADVANTAGES OF PARALLEL PROGRAMMING

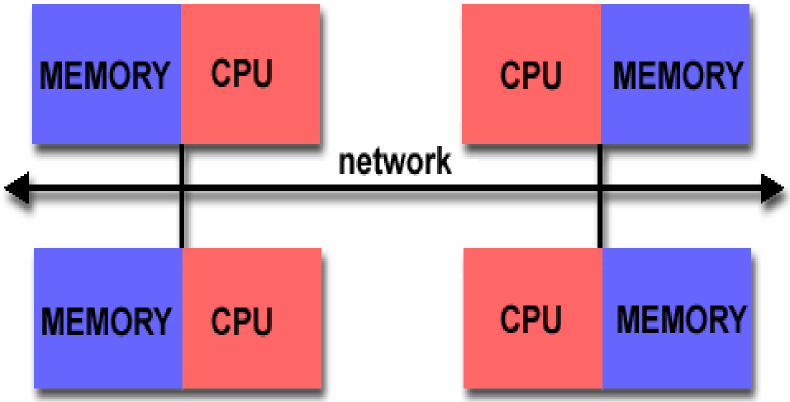
- **Need to solve larger problems:**
  - more memory intensive
  - more computation
  - more data intensive

- **Parallel programming provides :**
  - more CPU resources
  - more memory resources
  - solve problems that were not possible with serial program
  - solve problems more quickly

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 any architecture that supports such model
  - Distributed or shared-memory multi-processors
  - Networks of workstations
  - Single processor systems



IS MPI LARGE OR SMALL ?

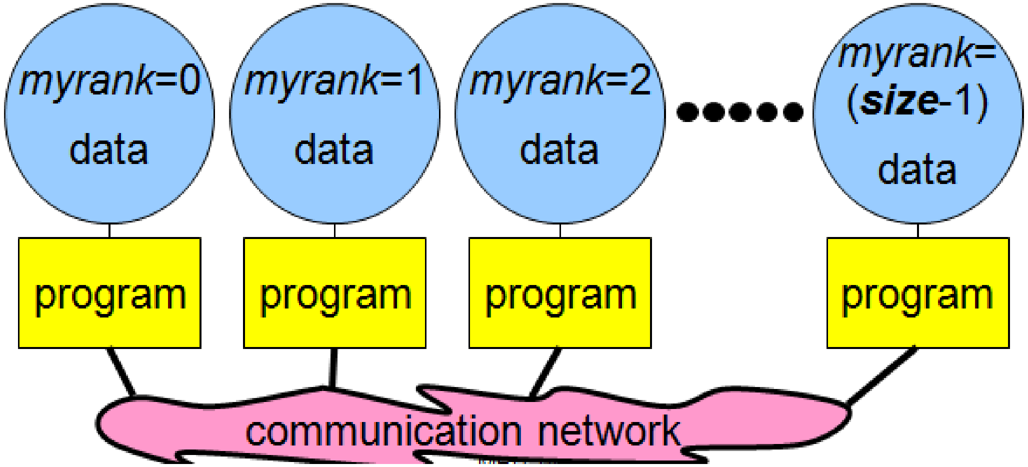
- **MPI is Large (more than 200 functions)**
  - Many feature requires extensive API
  - Complexity of use not related to number of functions

- **MPI is small (6 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

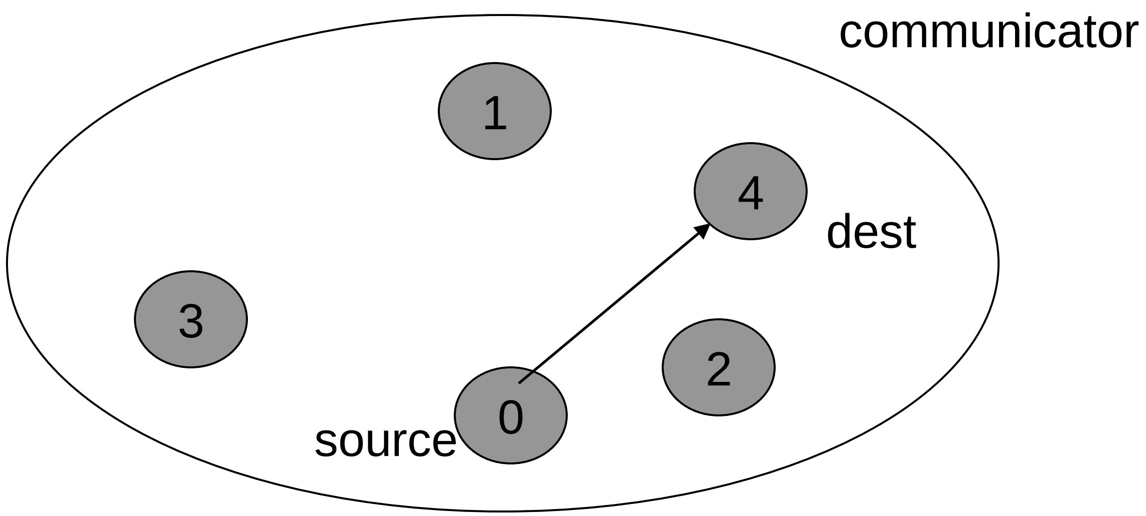
DATA AND WORK DISTRIBUTION

- Programmer imagines several processors, each with own memory, and writes a program to run on each processor
- To communicate together mpi-processes need unique identifiers: rank = identifying number
- all distribution decisions are based on the *rank*
  - i.e., which process works on which data



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
- MPI_COMM_WORLD is default communicator



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(s) are receiving the message
  - Where should the data be left on the receiving process
  - How much data is receiving process prepared to accept

BUILDING BLOCKS: SEND AND RECV

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

```
send(void *sendbuf,  int noelems,  int dest)

receive(void *recvbuf,  int noelems,  int source)
```

BUILDING BLOCKS: SEND AND RECV (CONTD....)

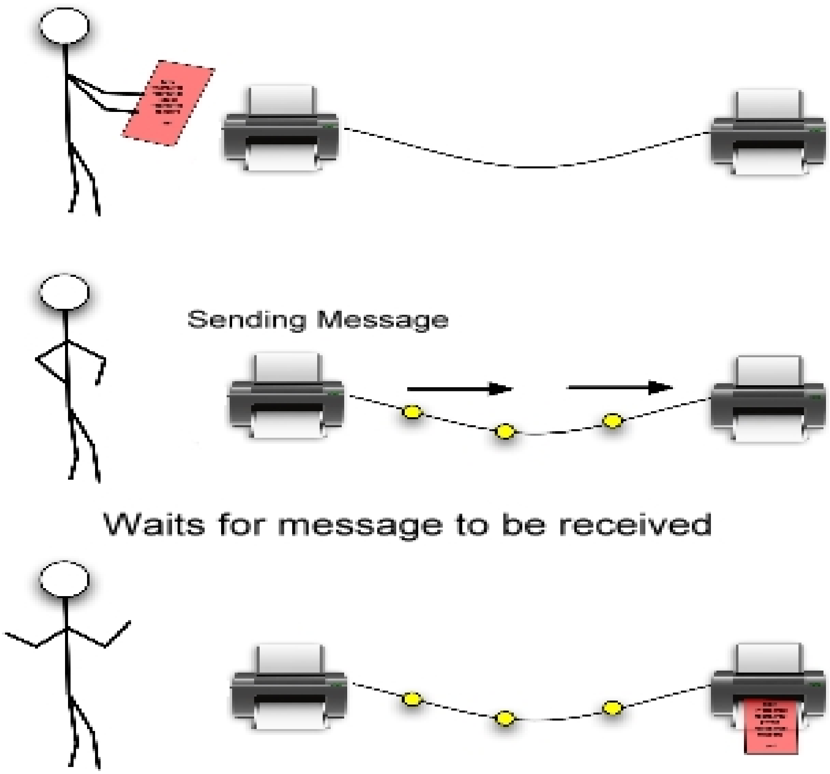
- "Completion" means that memory locations used in the message transfer can be safely accessed
  - send: variable sent can be reused after completion
  - receive: variable received can now be used

- 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

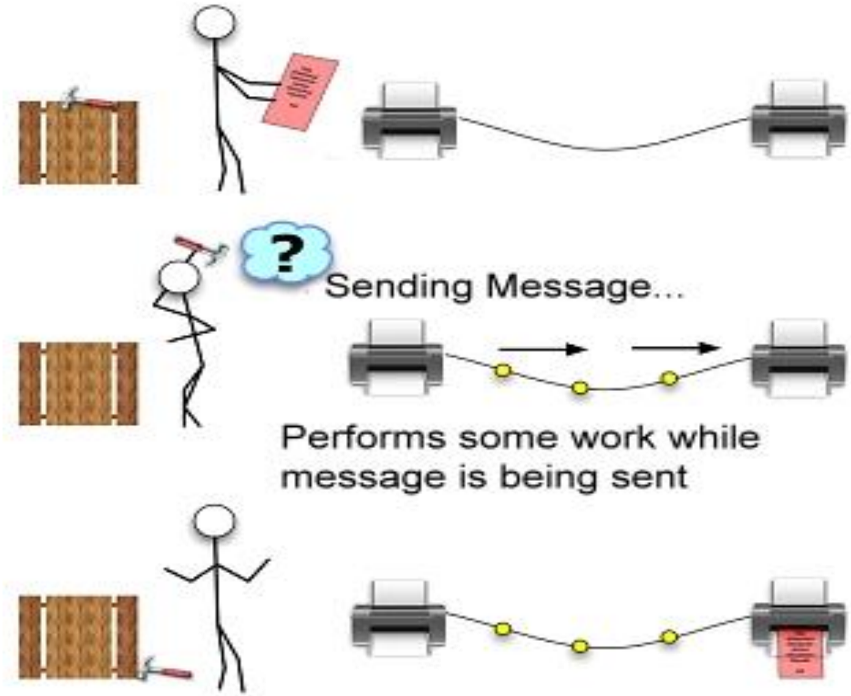
BLOCKING OPERATION

- 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.



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)
```

Fortran:

```
CALL MPI_INIT(IERROR)
INTEGER IERROR
```

COMMUNICATOR SIZE

- How many processes are contained within a communicator?

C:

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

Fortran:

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

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**

C:

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

Fortran:

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

EXITING MPI

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

C:

```
MPI_Finalize()
```

Fortran:

```
CALL MPI_FINALIZE(IERROR)
```

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

EXAMPLE PROGRAM 1: 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

EXAMPLE PROGRAM 1: HELLO_WORLD.F

```
program hello
include 'mpif.h'

integer rank, size, ierror, tag, status(MPI_STATUS_SIZE)

call MPI_INIT(ierror)
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)

print*, 'node', rank, ': Hello world '

call MPI_FINALIZE(ierror)

end
```

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)
```

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

MPI STATUS

- The status parameter returns additional information for some MPI routines
  - Additional Error status information
  - Additional information with wildcard parameters
- C declaration : a predefined struct
  - MPI_Status status;
- Fortran declaration : an array is used instead
  - INTEGER STATUS(MPI_STATUS_SIZE)

MPI STATUS

Accessing status information

- The tag of a received message
  - C : status.MPI_TAG
  - Fortran : STATUS(MPI_TAG)
- The source of a received message
  - C : status.MPI_SOURCE
  - Fortran : STATUS(MPI_SOURCE)
- The error code of the MPI call
  - C : status.MPI_ERROR
  - Fortran : STATUS(MPI_ERROR)
- Other uses...

MESSAGES

- A message contains an array of elements of some particular MPI datatype
- MPI datatypes:
  - Basic types
  - Derived types
- Derived types can be build up from basic types
- C types are different from Fortran types

MPI DATATYPES

C Data Types		Fortran Data Types	
MPI_CHAR	signed char	**MPI_CHARACTER**	character(1)
MPI_SHORT	signed short int		
MPI_INT	signed int	**MPI_INTEGER**	integer
MPI_LONG	signed long int		
MPI_UNSIGNED_CHAR	unsigned char		
MPI_UNSIGNED_SHORT	unsigned short int		
MPI_UNSIGNED	unsigned int		
MPI_UNSIGNED_LONG	unsigned long int		
MPI_FLOAT	float	**MPI_REAL**	real
MPI_DOUBLE	double	**MPI_DOUBLE_PRECISION**	double precision
MPI_LONG_DOUBLE	long double		
		MPI_COMPLEX	complex
		MPI_DOUBLE_COMPLEX	double complex
		MPI_LOGICAL	logical
MPI_BYTE	8 binary digits	**MPI_BYTE**	8 binary digits
MPI_PACKED	data packed or unpacked with MPI_Pack()/ MPI_Unpack	**MPI_PACKED**	data packed or unpacked with MPI_Pack()/ MPI_Unpack

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)**

STANDARD AND SYNCHRONOUS SEND

- **Standard send**
  - Completes once message has been sent
  - May or may not imply that message arrived
  - Don't make any assumptions (implementation dependent)
- **Synchronous send**
  - Use if need to know that message has been received
  - Sending and receiving process synchronize regardless of who is faster. Thus, processor idle time is possible
  - Large synchronization overhead
  - Safest communication method

READY AND BUFFERED SEND

- **Ready send**
  - Ready to receive notification must be posted; otherwise it exits with an error
  - Should not be used unless user is certain that corresponding receive is posted before the send
  - Lower synchronization overhead for sender as compared to synchronous send
- **Buffered send**
  - Data to be sent is copied to a user-specified buffer
  - Higher system overhead of copying data to and from buffer
  - Lower synchronization overhead for sender

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

NON-BLOCKING COMMUNICATIONS

- **Separates communication into three phases:**
  - Initiate non-blocking transfer
  - Do some other work not involving the data in transfer, i.e., overlap communication with calculation (latency hiding)
  - Wait for non-blocking communication to complete
- **Syntax of functions**
  - Similar to blocking functions' syntax
  - Each function has an "I" immediately following the "_". The rest of the name is the same
  - The last argument is a handle to an opaque request object that contains information about the message, i.e., its completion status

SENDING AND RECEIVING A MESSAGE

- int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request request)
- int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request request)
  - request: a request handle is allocated when a communication is initiated. Used to test if communication has completed.
  - Other parameters have the same definitions as for blocking functions

BLOCKING AND NON-BLOCKING

- Send and receive can be blocking or non-blocking
- A blocking send can be used with a non-blocking receive, and vice-versa
- Non-blocking sends can use any mode – synchronous, buffered, standard, or ready
  - No advantage for buffered or ready
- Characteristics of non-blocking communications
  - No possibility of deadlocks
  - Decrease in synchronization overhead
  - Increase or decrease in system overhead
  - Extra computation and code to test and wait for completion
  - Must not access buffer before completion

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)

DETECTING COMPLETIONS – MPI_TEST

- **MPI_Test tests for the completion of a send or receive.**
- **int MPI_Test (MPI_Request *request, int *flag, MPI_Status *status)**
  - request, status as for MPI_Wait.
  - does not block.
  - flag indicates whether operation is complete or not.
  - enables code which can repeatedly check for communication completion.

RECEIVE INFORMATION

- **Information of data is returned from MPI_Recv (or MPI_Irecv) as `status`**

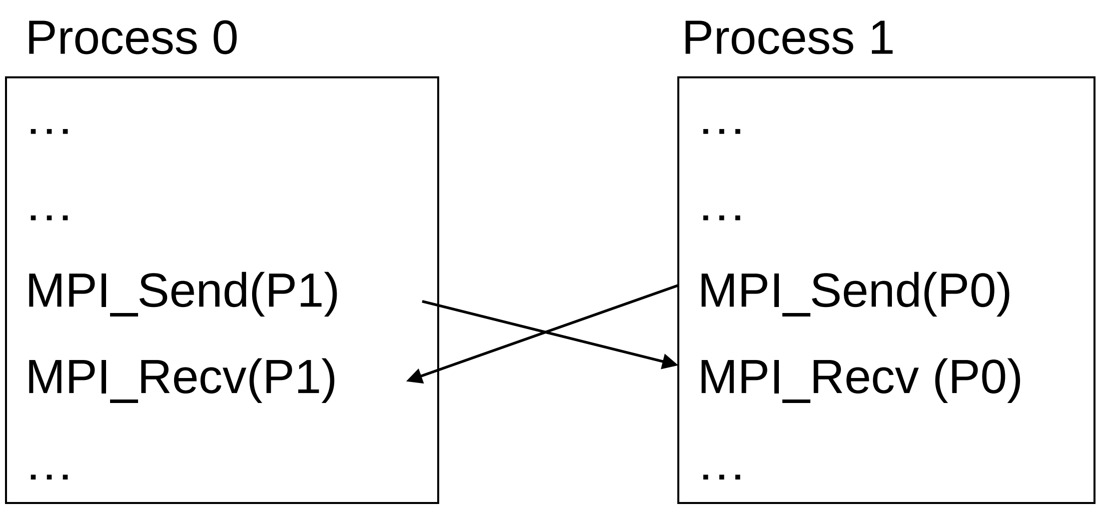
- **Information includes:**
  - Source: status.MPI_SOURCE
  - Tag: status.MPI_TAG
  - Error: status.MPI_ERROR
  - Count: message received may not fill receive buffer. Use following function to find number of elements actually received:

    int MPI_Get_count(MPI_Status status, MPI_Datatype datatype, int *count)

- **Message order preservation: messages do not overtake each other. Messages are received in the order sent.**

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



TIMERS

double MPI_Wtime(void)

- Time is measured in seconds
- Time to perform a task is measured by consulting the timer before and after

THANK YOU