Introduction to Parallel Programming with MPI

Mikhail Sekachev

Outline

- Message Passing Interface (MPI)
- Point to Point Communications

Today, 30-Jan-14

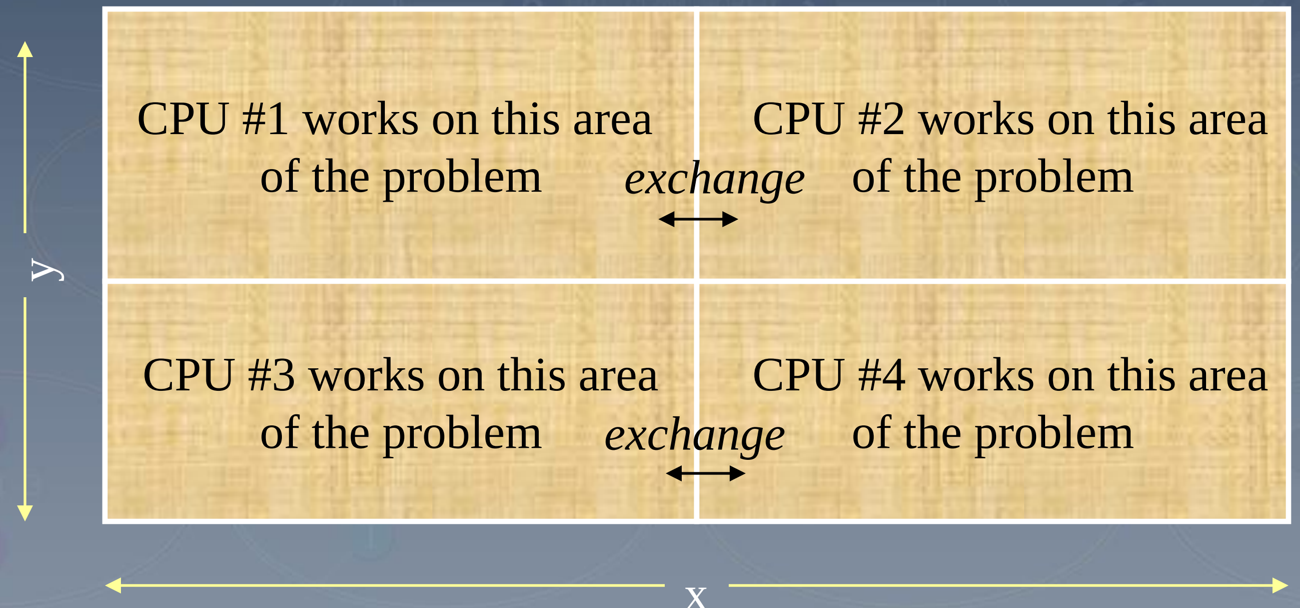
- Collective Communications
- Derived Datatypes

Tuesday, 4-Feb-14

What is Parallel Computing?

- *Parallel computing* - the use of multiple computers, processors or cores that work together on a common task.
  - Each processor works on a section of the problem
  - Processors are allowed to exchange information (data in local memory) with other processors

Grid of a problem to be solved



Good Old PC Cluster

Message Passing Interface

What is MPI?

- In 1992 the MPI Forum (40 organizations) established an MPI specification.
  - vendors, researchers, software library developers, and users
- By itself, MPI is NOT a library - but rather the specification of what such a library should be.
- As such, MPI is the first standardized vendor independent, message passing specification.
  - the syntax of MPI is standardized!
  - the functional behavior of MPI calls is standardized!
- Popular implementations: MPICH, OpenMPI (not to be confused with openMP), LAM, Cray MPT…

General MPI Program Structure

MPI Include file

Declarations, prototypes, etc.

Program begins

.
.
.

Serial code

Initialize MPI environment

Parallel code begins

.
.
.

Do work and make message passing calls

.
.
.

Terminate MPI environment

Parallel code ends

.
.
.

Serial code

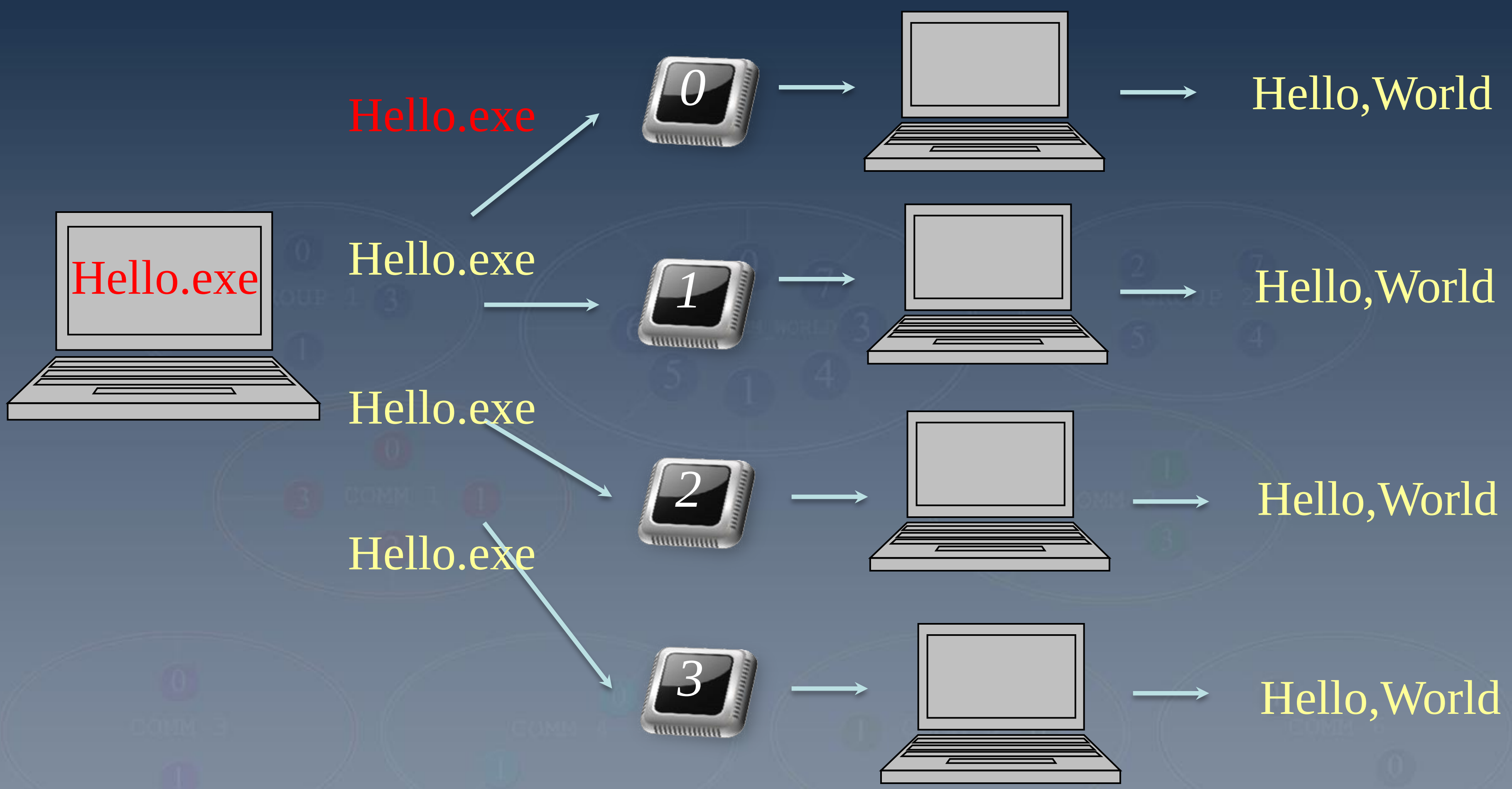
Program ends

First MPI Program: Hello,World!

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char** argv)
{
    MPI_Init(&argc, &argv);
    printf("Hello,World \n");
    MPI_Finalize();
}
```

- Every C program begins inside a function called `main`
- Every function starts with "{" and ends with "}"
- From `main` we can call other functions (including build-in functions)
- `#include` is a "preprocessor" directive that adds additional code from the header file called `stdio.h`

First MPI Program: Illustration



MPI Function Syntax

MPI syntax (C names are case sensitive; Fortran names are not):

C: `err = MPI_Xxxx(parameter,…)`
Fortran: `call MPI_XXXX(parameter,…, ierror)`

Errors:

C: Returned as `err`. `MPI_SUCCESS` if successful
Fortran: Returned as `ierror` parameter. `MPI_SUCCESS` if successful

SPMD Programming Paradigm

- Single Program, Multiple Data (SPMD) Programming Paradigm
- Same program runs on all processors, however, each processor operates on a different set of data
- The code generally contains an if-statement such as

```
if ( my_processor_id .eq. designated_id ) then
        -----/do work/-----
end
```

MPI Process Identifiers

- `MPI_Comm_rank`
  - Determines the rank of the calling process in the communicator
  - C : `int MPI_Comm_rank(MPI_Comm comm, int *rank)`
  - Fortran : `call MPI_COMM_RANK(mpi_comm, rank, ierror)`

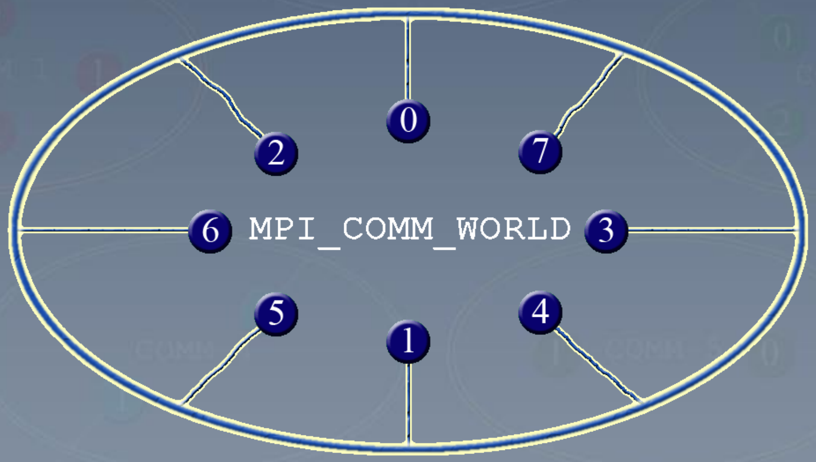
- `MPI_Comm_size`
  - Determines the size of the group associated with a communicator
  - C : `int MPI_Comm_size(MPI_Comm comm, int *size)`
  - Fortran : `call MPI_COMM_SIZE(mpi_comm, size, ierror)`

- `MPI_Comm` : Communicator

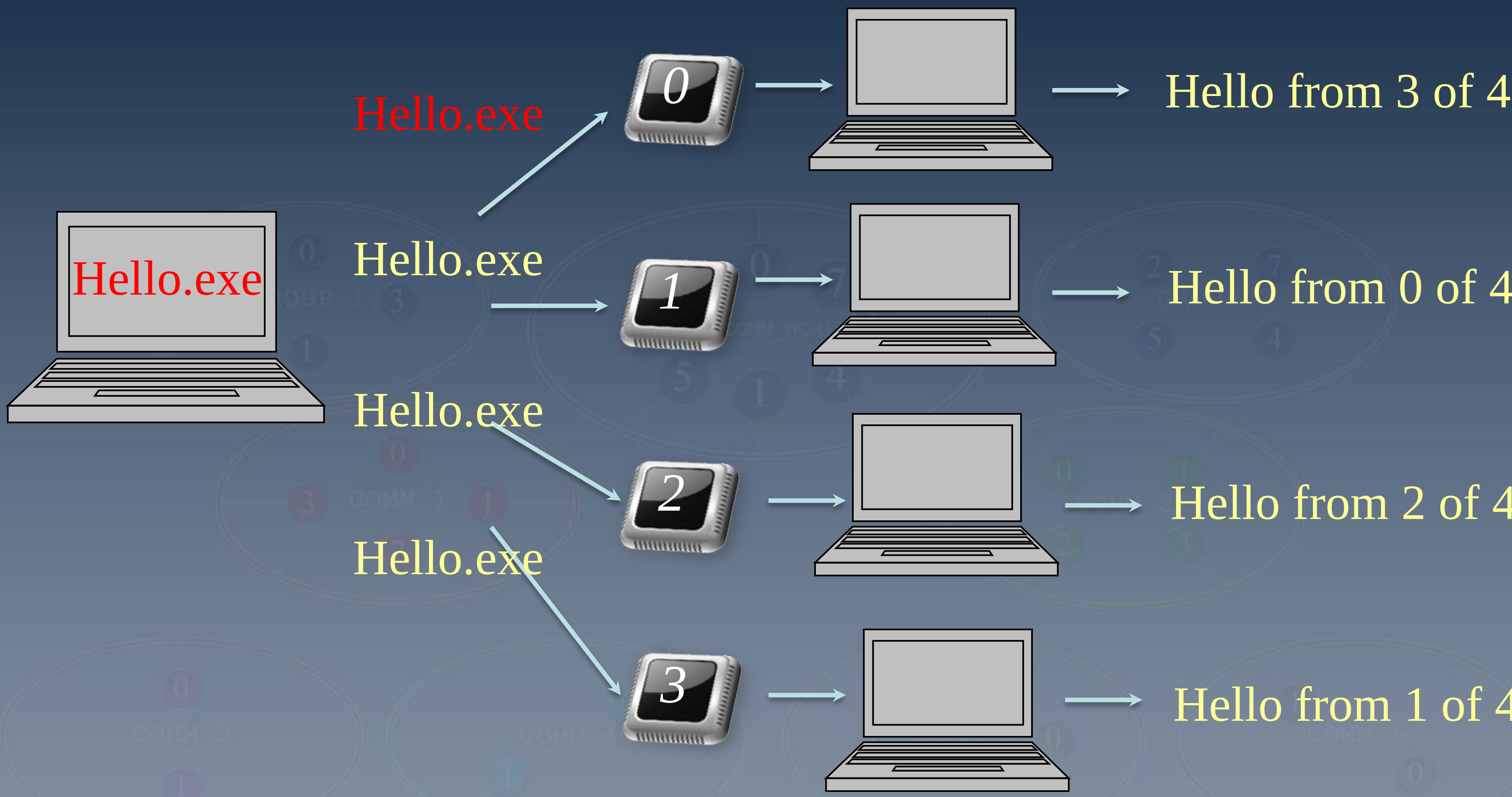
MPI Communicators: `MPI_COMM_WORLD`

- MPI uses objects called *communicators* to define which collection of processes may communicate with each other.
- All MPI communication calls require a *communicator argument* and MPI processes can only communicate if they share a communicator.

- `MPI_Init()` initializes a default communicator: `MPI_COMM_WORLD`
- `MPI_COMM_WORLD` contains all processes
- For simplicity, just use it wherever a communicator is required!



"Hello From…" program illustration



Example: “Hello From …” – C

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char** argv)
{
  int my_rank, num_procs;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

  MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

  printf("Hello from %d of %d.\n", my_rank, num_procs);

  MPI_Finalize();
}
```

MPI Environment Management Routines

- Most commonly used MPI environment management routines
  - initializing the MPI environment
  - querying the MPI environment
  - terminating the MPI environment

  `MPI_Init`

  `MPI_Comm_size`

  `MPI_Comm_rank`

  `MPI_Finalize`

- Other MPI environment management routines

  `MPI_Abort`

  `MPI_Get_processor_name`

  `MPI_Initialized`

  `MPI_Wtime`

  `MPI_Wtick`

MPI Basic Datatypes for C

MPI Datatypes	C Datatypes
MPI_CHAR	signed char
MPI_INT	signed int
MPI_LONG	signed long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	--------
MPI_SHORT	signed short int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED	unsigned int
MPI_PACKED	--------

Point to Point Communications

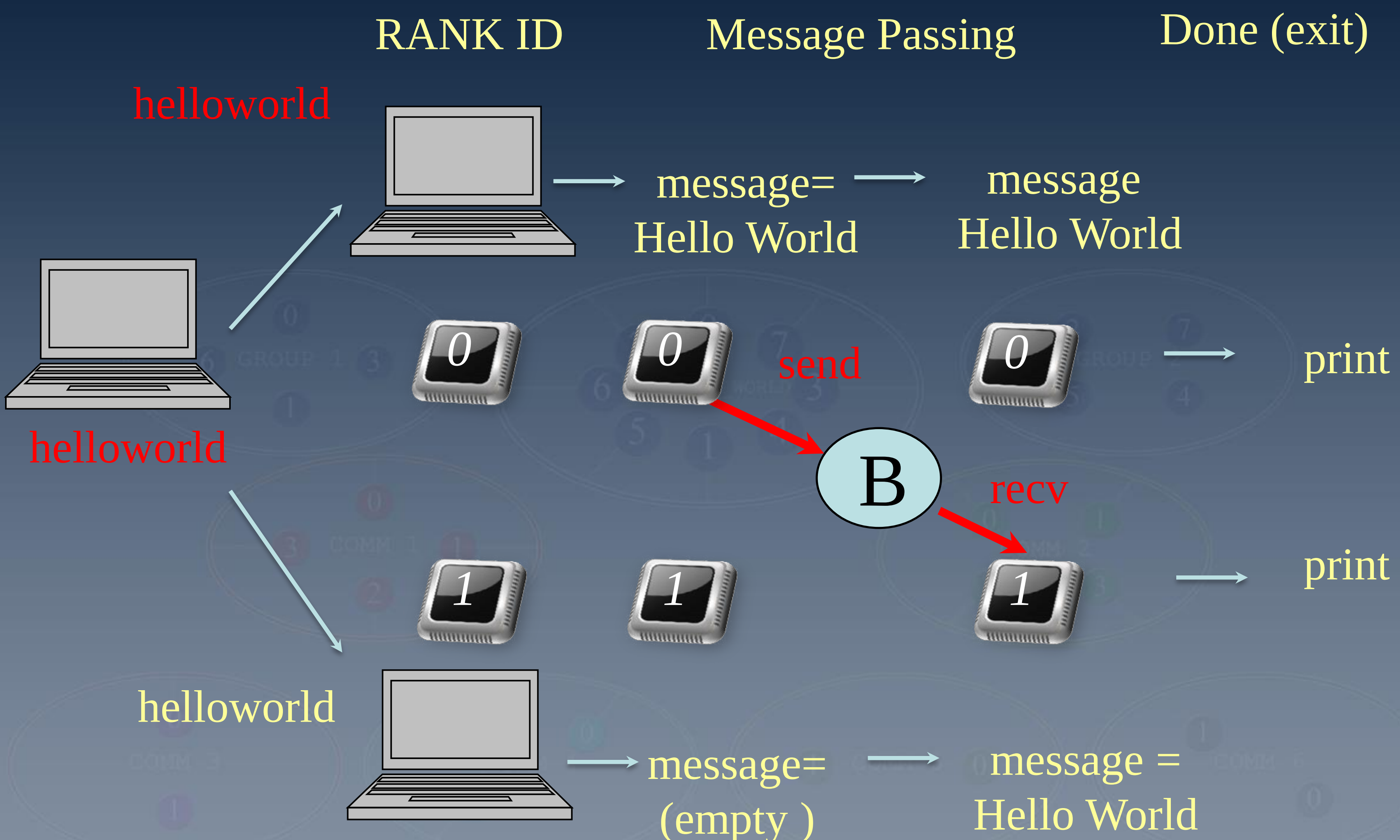
Overview

- MPI Point to point communication is message passing between two, and only two, different MPI tasks.
- One task is performing a send operation and the other task is performing a matching receive operation.
- MPI point-to-point routines can be either blocking or non-blocking
  - Blocking call stops the program until the message buffer is safe to use
  - Non-blocking call separates communication from computation
- MPI defines four communication modes for blocking and non-blocking send:
  - synchronous mode ("safest")
  - ready mode (lowest system overhead)
  - buffered mode (decouples sender from receiver)
  - standard mode (compromise)
- The receive call does not specify communication mode - it is simply blocking and non-blocking

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
- Message types must match
- Receiver's buffer must be large enough

Passing a Message: Illustration



Passing a Message: Hello World Again!

C Example

- ✓ 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
- ✓ Receiver's buffer must be large enough

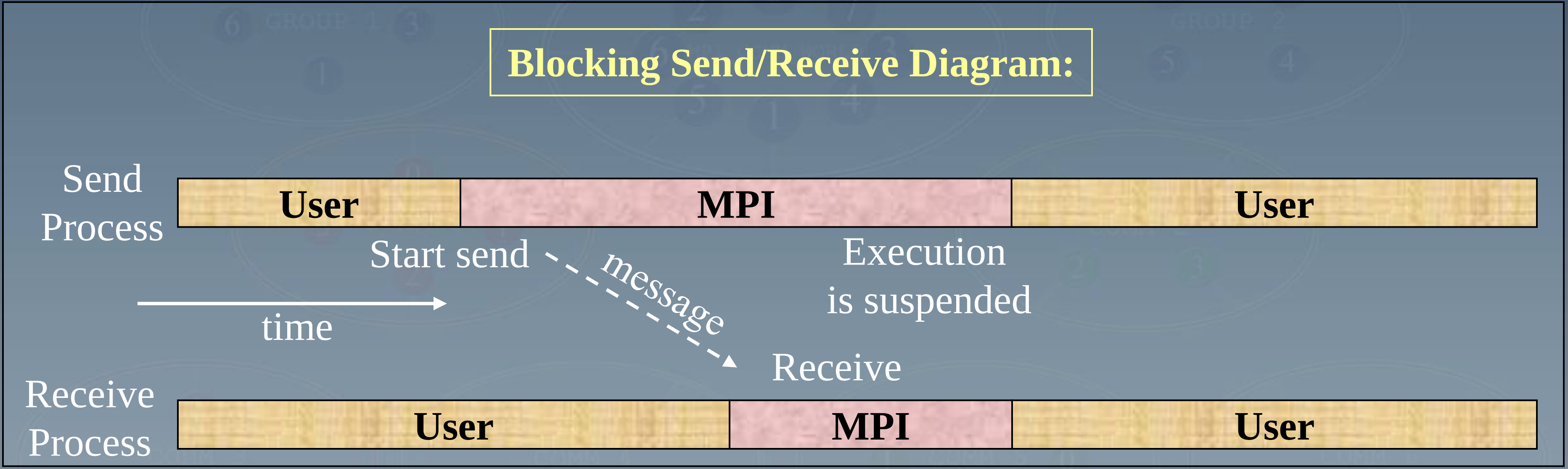
```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char ** argv)
  {
     int my_rank, ntag = 100;
     char message[12];
     MPI_Status status;
     MPI_Init(&argc, &argv);
     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

     if ( my_rank == 0 ) {
       message[12] = "Hello, world";
       MPI_Send(&message, 12,MPI_CHAR,1,ntag,MPI_COMM_WORLD);
       printf("Process %d : %s\n", my_rank, message);
     }
     else if ( my_rank == 1 ) {
       MPI_Recv(&message, 12,MPI_CHAR,0,ntag,MPI_COMM_WORLD, &status);
       printf("Process %d : %s\n", my_rank, message);
     }
     MPI_Finalize();
  }
```

Blocking Calls

- A blocking send or receive call suspends the execution of user's program until the message buffer being sent/received is safe to use.
- In case of a blocking send, this means the data to be sent have been copied out of the send buffer, but these data have not necessarily been received in the receiving task. The contents of the send buffer can be modified without affecting the message that was sent
- The blocking receive implies that the data in the receive buffer are valid.

Blocking Send and Receive

- A blocking MPI call means that the program execution will be suspended until the message buffer is safe to use. The MPI standards specify that a blocking SEND or RECV does not return until the send buffer is safe to reuse (for MPI_SEND), or the receive buffer is ready to use (for MPI_RECV).



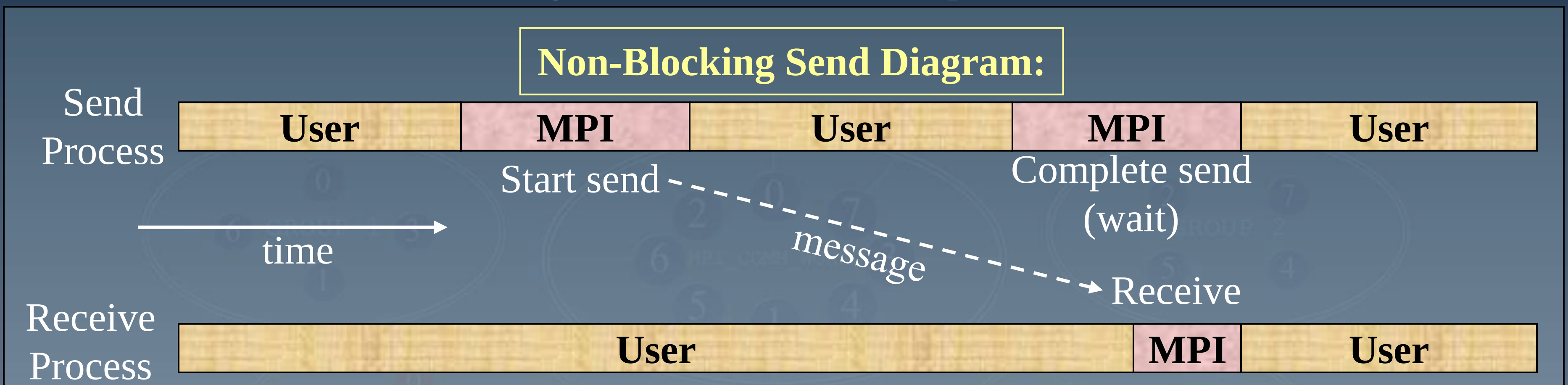
Non-Blocking Calls

- Non-blocking calls return immediately after initiating the communication.
- In order to reuse the send message buffer, the programmer must check for its status.
- In general, a blocking or non-blocking send can be paired to a blocking or non-blocking receive

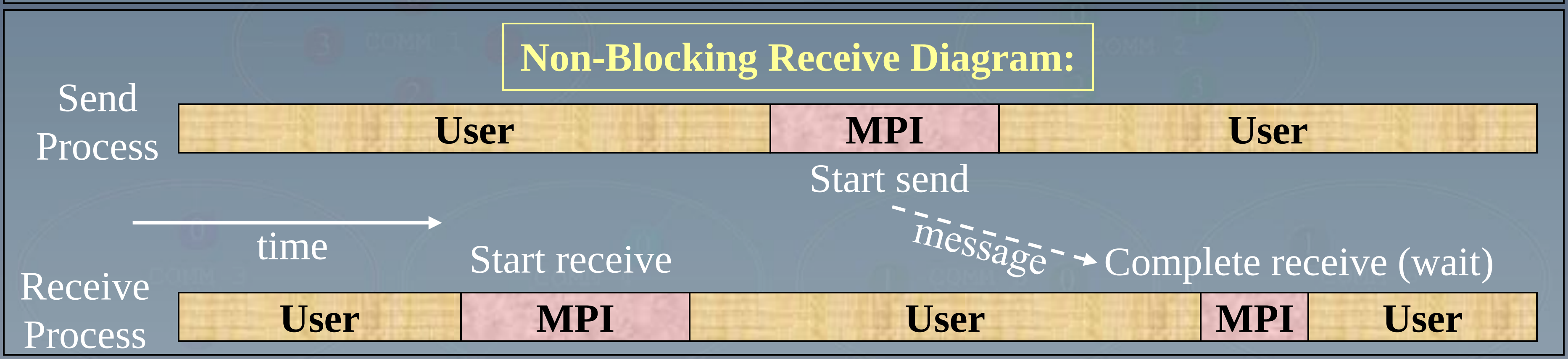
Joint Institute for Computational Sciences

Non-Blocking Send and Receive

- Separate Non-Blocking communication into three phases:
  - Initiate non-blocking communication.
  - Do some work (perhaps involving other communications?)
  - Wait for non-blocking communication to complete.



Non-Blocking Send Diagram:



Non-Blocking Receive Diagram:

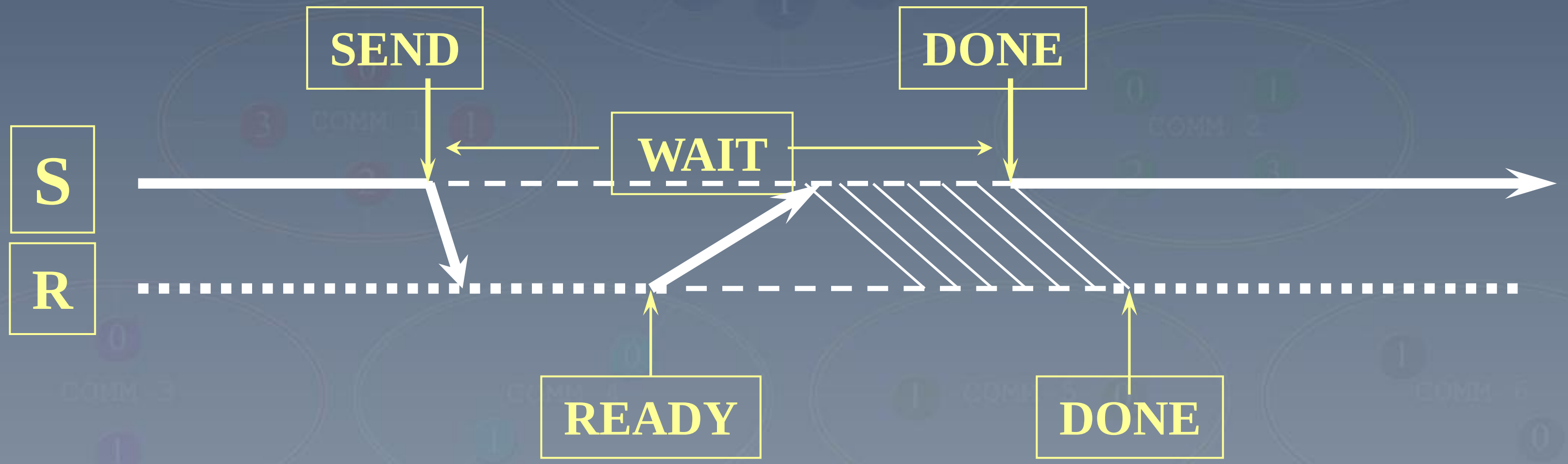
Communication Modes

- MPI has 8 different types of Send
- The non-blocking send has an extra argument of ***request handle***

	Blocking	**Non-Blocking**
Standard	MPI_Send	MPI_Isend
Synchronous	MPI_Ssend	MPI_Issend
Buffer	MPI_Bsend	MPI_Ibsend
Ready	MPI_Rsend	MPI_Iresend
	MPI_RECV	MPI_IRECV

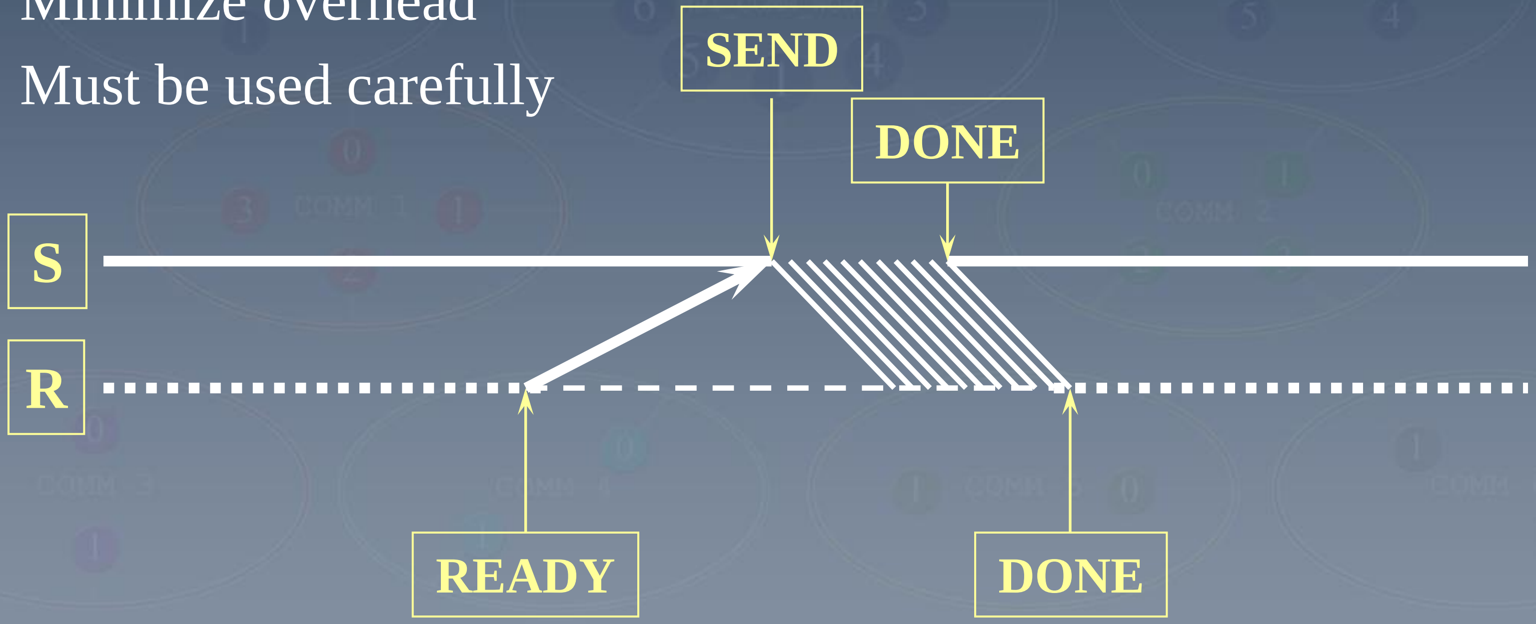
Blocking Synchronous Send: `MPI_SSEND`

- Can be started whether or not a matching receive was posted.
- However, the send will complete successfully only if a matching receive is posted.
- The sending task tells the receiver that a message is ready for it and waits for the receiver to acknowledge
- Synchronization overhead : handshake + waiting
- Safest , most portable



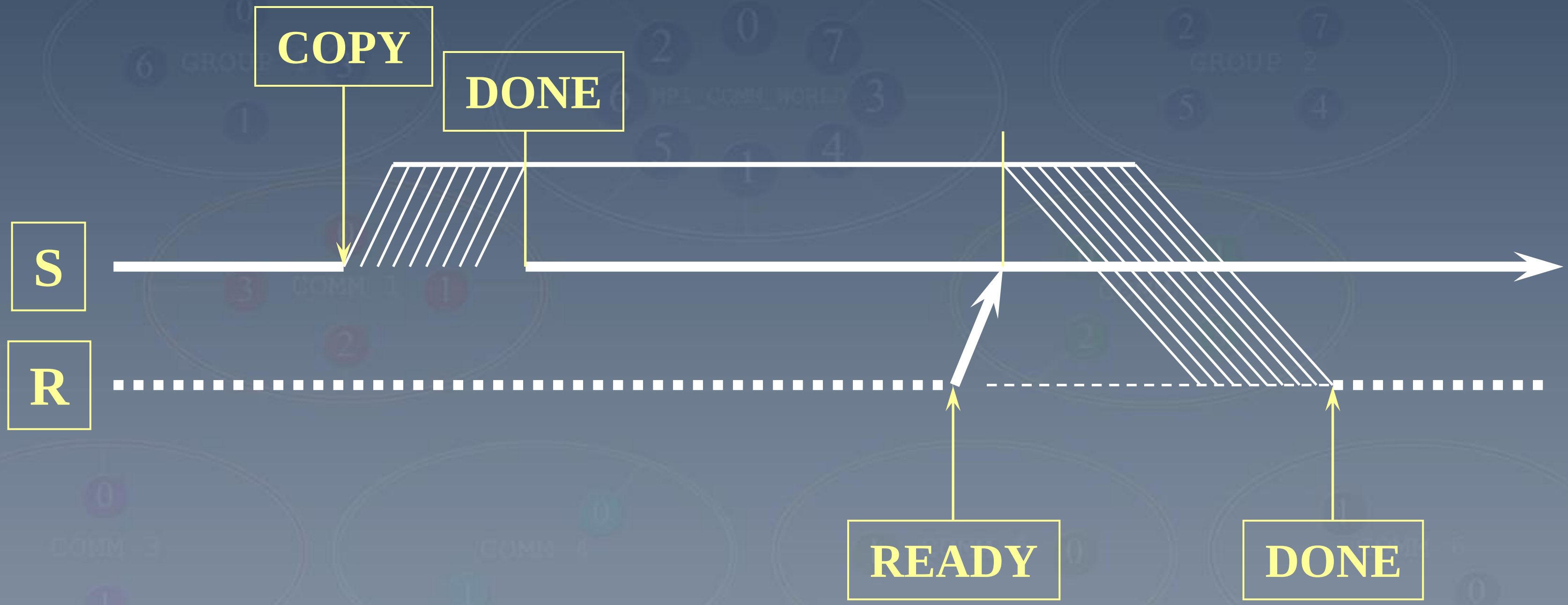
Blocking Ready Send: `MPI_RSEND`

- May be started *only* if the matching receive is already posted.
- Otherwise, the operation is erroneous and its outcome is undefined
- Allows the removal of a hand-shake operation
- The completion of the send operation does not depend on the status of a matching receive
- Minimize overhead
- Must be used carefully



Blocking Buffered Send: `MPI_BSEND`

- Can be started whether or not a matching receive was posted
- It may complete before a matching receive is posted.
- Buffer can be statically or dynamically allocated
- An error will occur if there is insufficient buffer space



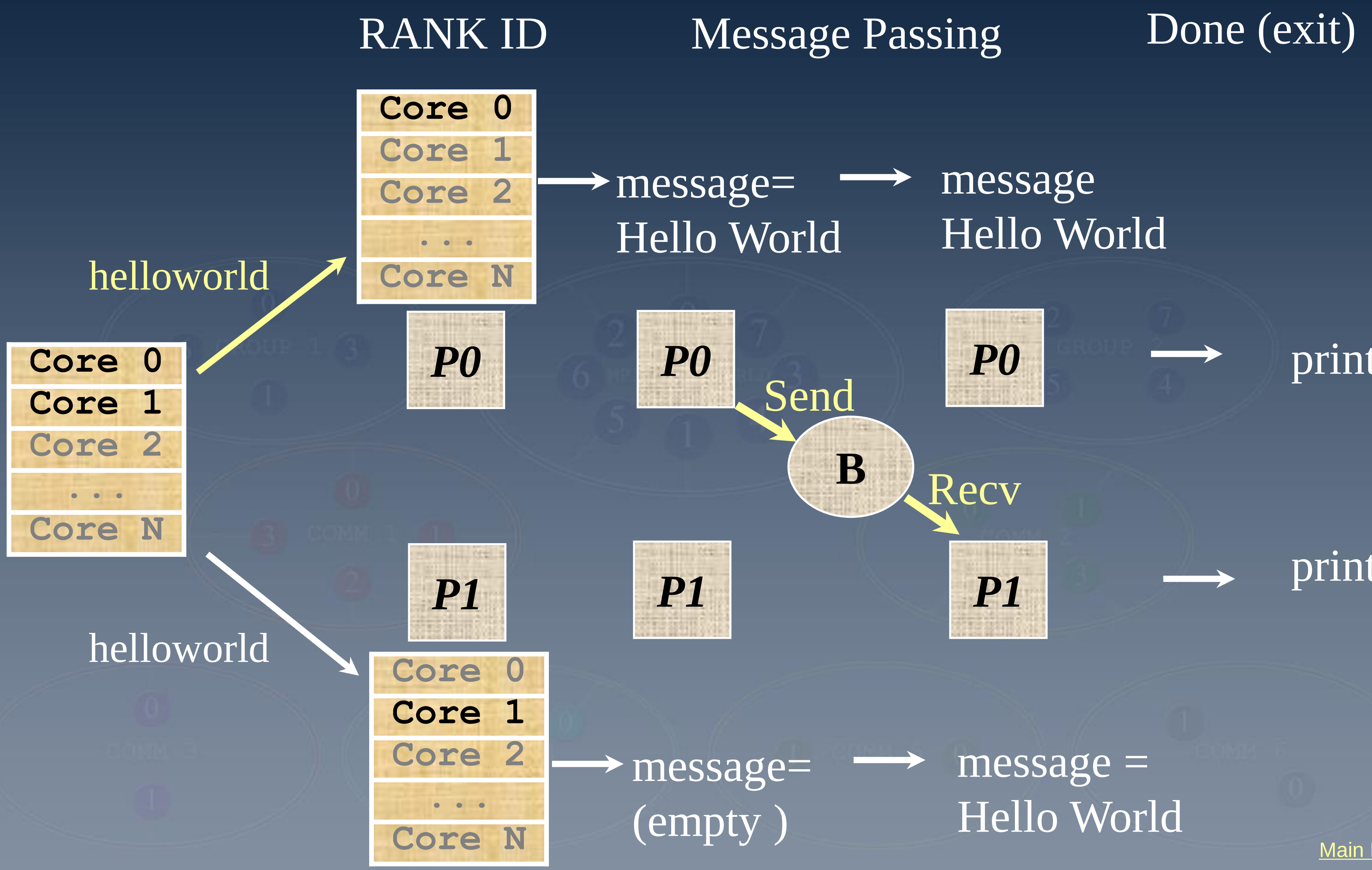
Blocking Standard Send: `MPI_Send`

- Either synchronous or buffered
- Implemented by vendors to give good performance for most programs.
- Simple and easy to use

Blocking Receive: `MPI_Recv`

- There is only one receive operation, which can match any of the send modes.
- Blocking receive returns only after the receive buffer contains the newly received message.
- Non-blocking receive can complete before the matching send has completed   (of course, it can complete only after the matching send has started)

Example: Passing a Message – Schematic



Example: Passing a Message – Hello World Again!

C Example

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char ** argv)
  {
     int my_rank, ntag = 100;
     char message[12];
     MPI_Status status;
     MPI_Init(&argc, &argv);
     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
     if ( my_rank == 0 ) {
       char message[12] = "Hello, world";
       MPI_Send(&message, 12,MPI_CHAR,1,ntag,MPI_COMM_WORLD);
       printf("Process %d : %s\n", my_rank, message);
     }
     else if ( my_rank == 1 )
     {
       MPI_Recv(&message, 12,MPI_CHAR,0,ntag,MPI_COMM_WORLD, &status);
       printf("Process %d : %s\n", my_rank, message);
     }
     MPI_Finalize();
  }
```

How many processes can you use to run this program?

1 : Fatal error in MPI_Send: Invalid rank, error stack

2 : Process 0 : Hello,World!
Process 1 : Hello,World!

3 : Application hangs!

Resources for Users: `man` pages and MPI web-sites

- There are man pages available for MPI which should be installed in your MANPATH. The following man pages have some introductory information about MPI.

```
% man MPI
% man cc
% man ftn
% man qsub
% man MPI_Init
% man MPI_Finalize
```

- MPI man pages are also available online. http://www.mcs.anl.gov/mpi/www/
- Main MPI web page at Argonne National Laboratory http://www-unix.mcs.anl.gov/mpi
- Set of guided exercises http://www-unix.mcs.anl.gov/mpi/tutorial/mpiexmpl
- MPI tutorial at Lawrence Livermore National Laboratory https://computing.llnl.gov/tutorials/mpi/
- MPI Forum home page contains the official copies of the MPI standard. http://www.mpi-forum.org/