TEXAS TECH UNIVERSITY

CS4379: Parallel and Concurrent Programming
CS5379: Parallel Processing

Lecture 22

Dr. Yong Chen

Associate Professor

Computer Science Department

Texas Tech University

Lecture Video

- Please view the lecture video either from Teams or from the below link:

  https://texastechuniversity.sharepoint.com/sites/CS4379-CS5379/Shared%20Documents/General/Lecture22.mp4

Course Info

- **Lecture Time**: TR, 12:30-1:50
- **Lecture Location**: ECE 217
- **Sessions**: CS4379-001, CS4379-002, CS5379-001, CS5379-D01
- **Instructor:** Yong Chen, Ph.D., Associate Professor
- **Email:** yong.chen@ttu.edu
- **Phone:** 806-834-0284
- **Office**: Engineering Center 315
- **Office Hours**: 2-4 p.m. on Wed., or by appointment
- **TA:** Mr. Ghazanfar Ali, Ghazanfar.Ali@ttu.edu
- **TA Office hours:** Tue. and Fri., 2-3 p.m., or by appointment
- **TA Office:** EC 201 A
- **More info:**
  - http://www.myweb.ttu.edu/yonchen
  - http://discl.cs.ttu.edu; http://cac.ttu.edu/; http://nsfcac.org

Outline

- Questions?

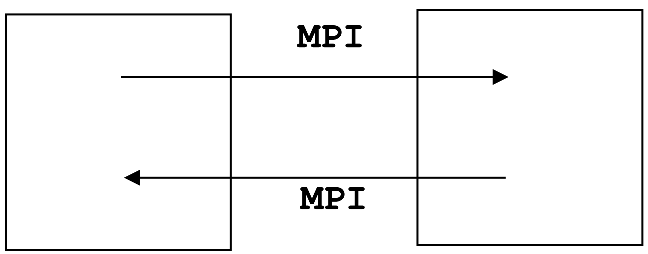
- Principles of Message Passing Programming
- MPI (Message Passing Interface) Basics
- Blocking v.s. Non-blocking

Principles of Message-Passing Programming

- The logical view of a distributed-address-space parallel computer supporting the message-passing paradigm consists of p processes, each with its own exclusive address space.
- Each data element must belong to some process' address space; hence, data must be explicitly partitioned and placed.
- All interactions (read-only or read and write) require cooperation of two (or more) processes - the process that has the data and the process that wants to access the data.
- Most message-passing programs are written using the single program multiple data (SPMD) model.

Principles of Message-Passing Programming

- Message passing model is for communication among processes (with separate address spaces)
- Interprocess communication consists of
  - Movement of data from one process's address space to another's
  - Synchronization



What is MPI?

- Message Passing Interface (MPI)
- A *message-passing library specification for parallel computers*
  - Not a language or compiler specification
  - Not a specific implementation or product
- Full-featured
- Designed for
  - Application developers
  - Library writers
  - Tool developers

Where Did MPI Come From?

- Early vendor systems (Intel’s NX, IBM’s EUI) were not portable
- Early portable systems (PVM, p4, Chameleon) were mainly research efforts
  - Did not address the full spectrum of message-passing issues
  - Lacked vendor support
  - Were not implemented at the most efficient level
- The MPI Forum organized in 1992 with broad participation by:
  - Vendors: IBM, Intel, TMC, SGI, Convex, Meiko
  - Portability library writers: PVM, p4
  - Users: application scientists and library writers
  - MPI-1 finished in 18 months

Important Considerations while Using MPI

- All parallelism is explicit: the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs.

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 );
    printf( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

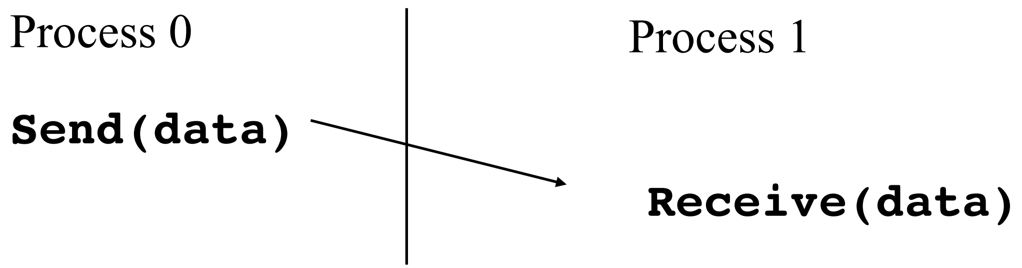
Hello World (Fortran)

```
program main
include 'mpif.h'
integer ierr, rank, size

call MPI_INIT( ierr )
call MPI_COMM_RANK( MPI_COMM_WORLD, rank, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, size, ierr )
print *, 'I am ', rank, ' of ', size
call MPI_FINALIZE( ierr )
end
```

MPI Basic Send/Receive

- We need to fill in the details in



- Things that need specifying:
  - How will "data" be described?
  - How will processes be identified?
  - How will the receiver recognize/screen messages?
  - What will it mean for these operations to complete?

Some Basic Concepts

- Processes can be collected into *groups*.
- Each message is sent in a *context*, and must be received in the same context.
- A group and context together form a *communicator*.
- A process is identified by its *rank* in the group associated with a communicator.
- There is a default communicator whose group contains all initial processes, called **MPI_COMM_WORLD**.

MPI Datatypes

- The data in a message to sent or received is described by a triple (address, count, datatype), where

- An MPI *datatype* is recursively defined as:
  - predefined, corresponding to a data type from the language (e.g., MPI_INT, MPI_DOUBLE_PRECISION)
  - a contiguous array of MPI datatypes
  - a strided block of datatypes
  - an indexed array of blocks of datatypes
  - an arbitrary structure of datatypes

- There are MPI functions to construct custom datatypes, such an array of (int, float) pairs, or a row of a matrix stored columnwise.

MPI Tags

- Messages are sent with an accompanying user-defined integer *tag*, to assist the receiving process in identifying the message.

- Messages can be screened at the receiving end by specifying a specific tag, or not screened by specifying **MPI_ANY_TAG** as the tag in a receive.

- Some non-MPI message-passing systems have called tags “message types”. MPI calls them tags to avoid confusion with datatypes.

MPI Basic (Blocking) Send

MPI_SEND (address, count, datatype, dest, tag, comm)

- The message buffer is described by (**address, count, datatype**).
- The target process is specified by **dest**, which is the rank of the target process in the communicator specified by **comm**.
- When this function returns, the data has been delivered to the system and the buffer can be reused. The message may not have been received by the target process.

MPI Basic (Blocking) Receive

MPI_RECV(address, count, datatype, source, tag, comm, status)

- Waits until a matching (on **source** and **tag**) message is received from the system, and the buffer can be used.
- **source** is rank in communicator specified by **comm**, or **MPI_ANY_SOURCE**.
- **status** contains further information
- receiving fewer than **count** occurrences of **datatype** is OK, but receiving more is an error.

Status Object

- The status object is used after completion of a receive to find the actual length, source, and tag of a message
- Status object is MPI-defined type and provides information about:
  - The source process for the message (status.source)
  - The message tag (status.tag)
- The number of elements received is given by:

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

status return status of receive operation (Status)
datatype datatype of each receive buffer element (handle)
count number of received elements (integer)(OUT)

MPI is Simple

- Many parallel programs can be written using just these six functions, only two of which are non-trivial:
  - **MPI_INIT – initialize the MPI library (must be the first routine called)**
  - **MPI_COMM_SIZE - get the size of a communicator**
  - **MPI_COMM_RANK – get the rank of the calling process in the communicator**
  - **MPI_SEND – send a message to another process**
  - **MPI_RECV – send a message to another process**
  - **MPI_FINALIZE – clean up all MPI state (must be the last MPI function called by a process)**

- For performance, however, you need to use other MPI features

Outline

- Questions?
- Principles of Message Passing Programming
- MPI (Message Passing Interface) Basics
- Blocking v.s. Non-blocking

Blocking v/s Non-blocking modes

- Blocking mode:
  - Return from the routine implies completion.
- Non-Blocking mode:
  - Routine returns immediately, doesn't imply completion, the completion needs to be tested for
  - Primarily used to overlap computation with communication and exploit possible performance gains

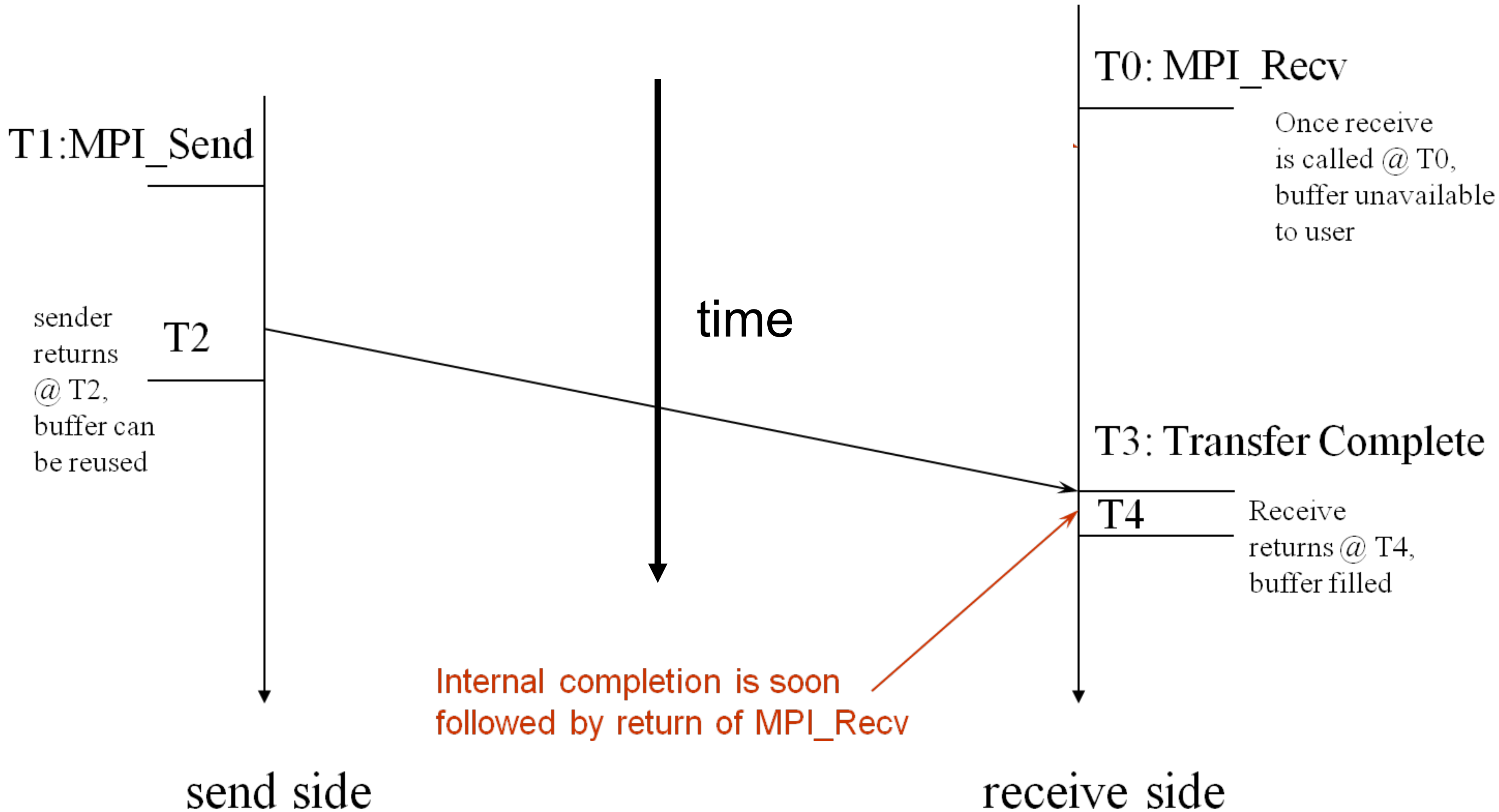
- "Completion" means that memory locations used in the message transfer can be safely accessed for reuse.
  - Safe means that modifications will not affect the data intended for the receive task.
  - For "send" completion implies variable sent can be reused/modified
    - Nothing is said whether the message has been delivered/received
  - For "receive" variable received can be read.

Blocking Communication

- In Blocking communication
  - MPI_SEND does not complete until buffer is emptied (available for reuse)
  - MPI_RECV does not complete until buffer is filled (available for use)
- A process sending data will be blocked until data in the send buffer is emptied
- A process receiving data will be blocked until the receive buffer is filled
- Completion of communication generally depends on the message size and the system buffer size
- Blocking communication is simple to use but can be prone to deadlocks

```
                      If (myrank .eq. 0) then
                            Call mpi_send(..)
                            Call mpi_recv(…)
Can have deadlocks →     Else
                             Call mpi_send(…) ← UNLESS you reverse send/recv
                             Call mpi_recv(….)
                          Endif
```

Blocking Send-Receive Diagram



Non-Blocking Communication

- Non-blocking operations return (immediately) "request handles" that can be waited on and queried
  - **MPI_ISEND**(start, count, datatype, dest, tag, comm, request)
  - **MPI_IRECV**(start, count, datatype, src, tag, comm, request)
  - **MPI_WAIT**(request, status) -> Blocking!

- Non-blocking operations allow overlapping computation and communication.
- One can also test without waiting using MPI_TEST
  - **MPI_TEST**(request, flag, status)
- Anywhere you use MPI_Send or MPI_Recv, you can use the pair of MPI_Isend/MPI_Wait or MPI_Irecv/MPI_Wait

Multiple Completions

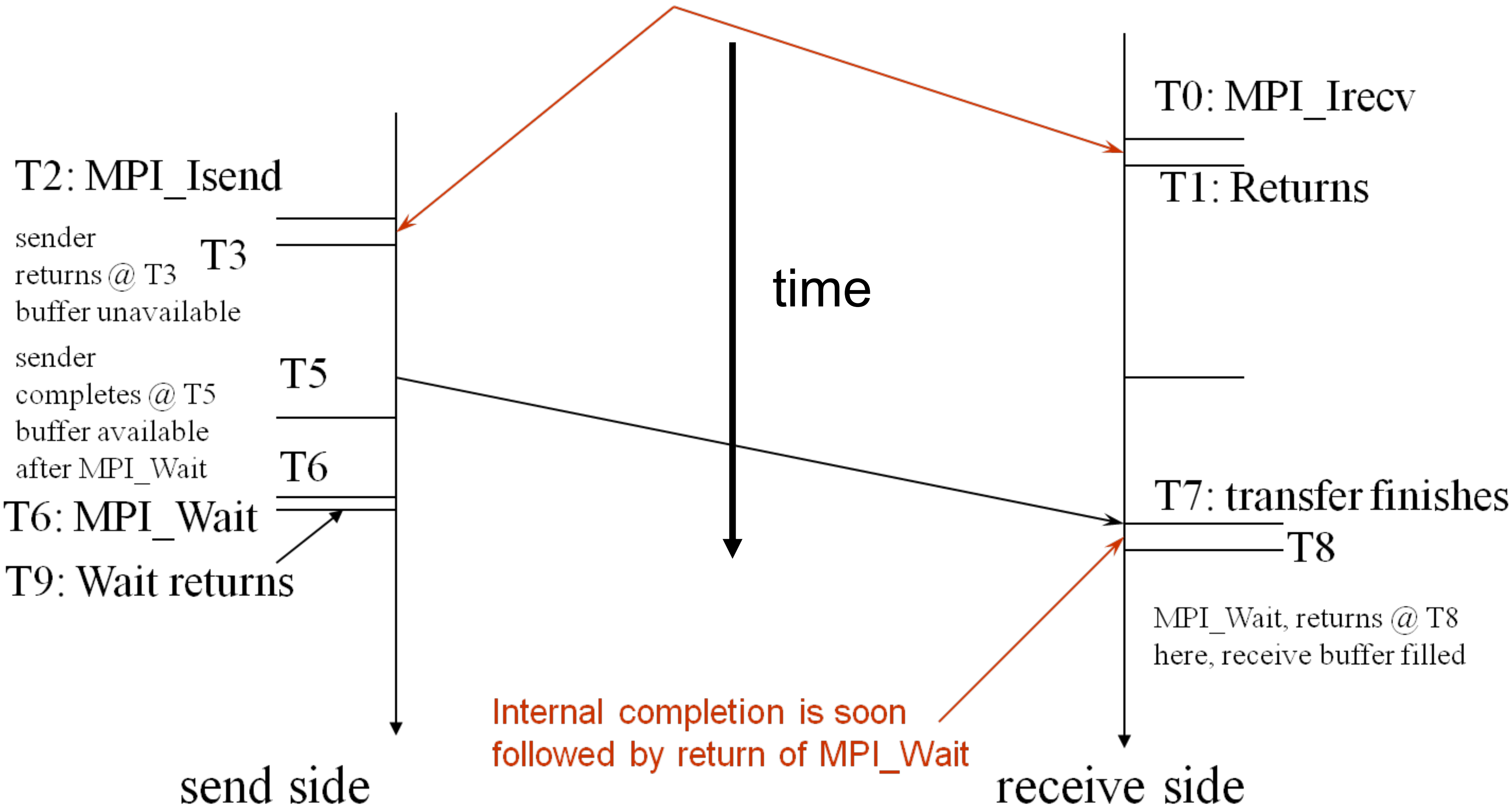
- It is sometimes desirable to wait on multiple requests:

```
MPI_Waitall(count, array_of_requests,
   array_of_statuses)
MPI_Waitany(count, array_of_requests,
   &index, &status)
MPI_Waitsome(count, array_of_requests,
   array_of_indices, array_of_statuses)
```

- There are corresponding versions of **test** for each of these.

Non-Blocking Send-Receive Diagram

High Performance Implementations
Offer Low Overhead for Non-blocking Calls



Non-Blocking Communication

- Avoid deadlocks
- Post all requests and let matches happen

```
If (myrank .eq. 0) then
        Call mpi_isend(..)
        Call mpi_irecv(…)
    Else
         Call mpi_isend(…)
         Call mpi_irecv(….)
    Endif
```

Message Completion and Buffering

- 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
- A send has completed when the user supplied buffer can be reused

```
*buf =3;
MPI_Send (buf, 1, MPI_INT …)
*buf = 4; /*OK, receiver will always receive 3*/
```

```
*buf =3;
MPI_Isend (buf, 1, MPI_INT …)
*buf = 4;  /*Not certain if receiver gets 3 or 4*/
MPI_Wait(…);
```

- Just because the send completes does not mean that the receive has completed
  - Message may be buffered by the system
  - Message may still be in transit

Readings

- Reference book ITPC – Chapter 6, 6.1-6.2

Questions?

Questions/Suggestions/Comments are always welcome!

Write me: yong.chen@ttu.edu
Call me: 806-834-0284
See me: ENGCTR 315

If you write me an email for this class, please start the email subject with [CS4379] or [CS5379].