TEXAS ADVANCED COMPUTING CENTER

WWW.TACC.UTEXAS.EDU

The University of Texas at Austin

# MPI Programming

IHPCSS

Parallel Programming: Classic Track

July 7-12, 2019

PRESENTED BY:

John Cazes

cazes@tacc.utexas.edu

Lecture and Lab slides available at: tinyurl.com/tacc-2019-ihpcss

# Outline

# Message Passing Interface – MPI

Allows the exchange of data between independent processes that reside either on the same node or on different nodes of a cluster.

Universality : MPI is a standard that is supported on all multi-node HPC platforms.

Expressivity : MPI has been found to be a useful and complete model in which to express parallel algorithms.

## Message Passing Interface – MPI

MPI - Message Passing Interface

- Library standard defined by committee of vendors, implementers, and parallel programmers
- Used to create parallel programs based on message passing
- Available on most HPC systems with C and Fortran bindings
- Used to communicate between processes both on-node and off-node

## Background on MPI

MPI 1.0 – May 1994

- Point-to-Point communications
- Collective operations
- Communicator groups
- Datatypes

MPI 2.0 – July 1997(2.1 – September 2008)

- One-sided communications
- Parallel I/O
- Inter communicators

MPI 3.0 – September 2012

- Non-blocking collectives
- Improved one-sided communications
- Support for Fortran08 bindings

# MPI Implementations

There are optimized versions supported by both vendors and opensource efforts.

Base implementations

- MPICH  https://www.mpich.org/
  - Intel MPI  https://software.intel.com/en-us/mpi-library
  - MVAPICH2  http://mvapich.cse.ohio-state.edu/
  - Cray MPI  https://pubs.cray.com/content/S-2529/17.05/xctm-series-programming-environment-user-guide-1705-s-2529/mpt
- OPENMPI  https://www.open-mpi.org/
  - Mellanox HPC-X  http://www.mellanox.com/page/hpcx_overview

## Key Concepts of MPI

MPI is an Application Programming Interface (API) standard

- Not a stand alone compiler
- Not a language
- Used to exchange data between programs/processes both on-node and off-node

Two models of programming

- SPMD – Single program/ multiple data
- MPMD – Multiple program / multiple data

# MPI Include and Module Files

MPI libraries need header information to define constants and interfaces

From include files:

- C: `#include <mpi.h>`
- Fortran: `include “mpif.h”`

Or module files for Fortran:

- Fortran: `use mpi`
- Fortran: `use mpi_f08` – supports Fortran08 bindings
  – requires explicit typing of arguments

Compiler/build wrappers are usually provided, which point to the correct path to the include files and libraries

```
mpicc –show  #Show actual compile command for C
mpif90 –show #Show actual compile command for Fortran
```

# Communicators

Communicators

- A parameter for most MPI calls
- A collection of processors working on some part of a parallel job
- MPI_COMM_WORLD is defined in the MPI include file as all of the processors in your job
- Can create subsets of MPI_COMM_WORLD
- Processors within a communicator are assigned numbers *0* to *n-1*

## Data Types

Data types

- When sending a message, it is given a data type
- Predefined types correspond to "normal" types
  - MPI_REAL , MPI_FLOAT -Fortran and C real
  - MPI_DOUBLE PRECISION  , MPI_DOUBLE - Fortran and C double
  - MPI_INTEGER and MPI_INT - Fortran and C integer
- Can create user-defined types

# Minimal MPI program

Every MPI program needs these…

C version

```
#include <mpi.h>                                    /* MPI include file */
...
ierr=MPI_Init(&argc, &argv);                          /* Initialize MPI */
ierr=MPI_Comm_size(MPI_COMM_WORLD,&nPEs);                /* Total tasks */
ierr=MPI_Comm_rank(MPI_COMM_WORLD,&iam);               /* Taskid (rank) */
...
ierr=MPI_Finalize();                                    /* Finalize MPI */
```

In C MPI routines are functions and return an error value

# Minimal MPI program

Every MPI program needs these…

Fortran version

```
include 'mpif.h'                                    ! MPI include file
…
call MPI_Init(ierr)                                 ! Initialize MPI
call MPI_Comm_size(MPI_COMM_WORLD, nPEs, ierr)      ! Total tasks
call MPI_Comm_rank(MPI_COMM_WORLD, iam, ierr)       ! Taskid (rank)
     ...
call MPI_Finalize(ierr)                             ! Finalize MPI
```

In Fortran, MPI routines are subroutines, and last parameter is an error value

## Basic Communications in MPI

- Data values are transferred from one processor to another
  - One process sends the data
  - Another receives the data
- Blocking
  - Call does not return until the message is sent or received
- Nonblocking
  - Call indicates a start of send or received, and another call is made to determine if finished

# The Six Basic MPI Calls

MPI is used to create parallel programs based on message passing

The same program is run on multiple processors

The 6 basic calls in MPI are:

```
1. call MPI_Init(ierr)
2. call MPI_Comm_rank(MPI_COMM_WORLD, myid, ierr)
3. call MPI_Comm_size(MPI_COMM_WORLD, numprocs, ierr)
4. call MPI_Send(buffer,count,MPI_TYPE,destination,
                 tag,MPI_COMM_WORLD,ierr)
5. call MPI_Recv(buffer,count,MPI_TYPE,source,tag,
                 MPI_COMM_WORLD,status,ierr )
6. call MPI_Finalize(ierr)
```

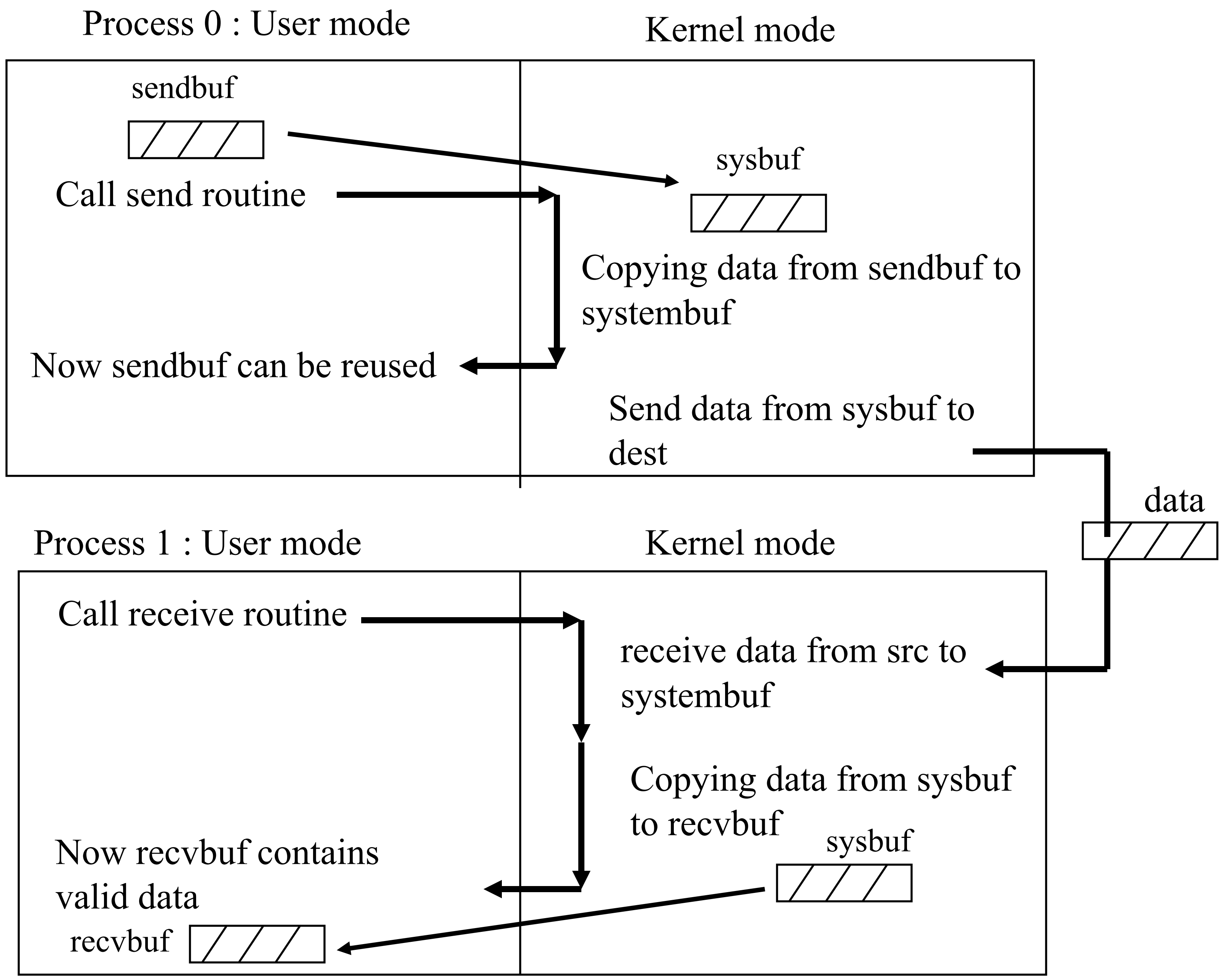
# Point to Point Communications

Sending process

- data is copied to the user buffer by the user
- User calls one of the MPI send routines
- System copies the data from the user buffer to the system buffer
- System sends the data from the system buffer to the destination processor

Receiving process

- User calls one of the MPI receive subroutines
- System receives the data from the source process, and copies it to the system buffer
- System copies the data from the system buffer to the user buffer
- User uses the data in the user buffer


Process 0 : User mode
Kernel mode
sendbuf
sysbuf
Call send routine
Copying data from sendbuf to systembuf
Now sendbuf can be reused
Send data from sysbuf to dest
data
Process 1 : User mode
Kernel mode
Call receive routine
receive data from src to systembuf
Copying data from sysbuf to recvbuf
sysbuf
Now recvbuf contains valid data
recvbuf

# Unidirectional Communication

Blocking send and blocking receive

C:

```
if (myrank == 0) {
    ierr = MPI_Send(…)
}
else if (myrank == 1) {
    ierr = MPI_Recv(….)
}
```

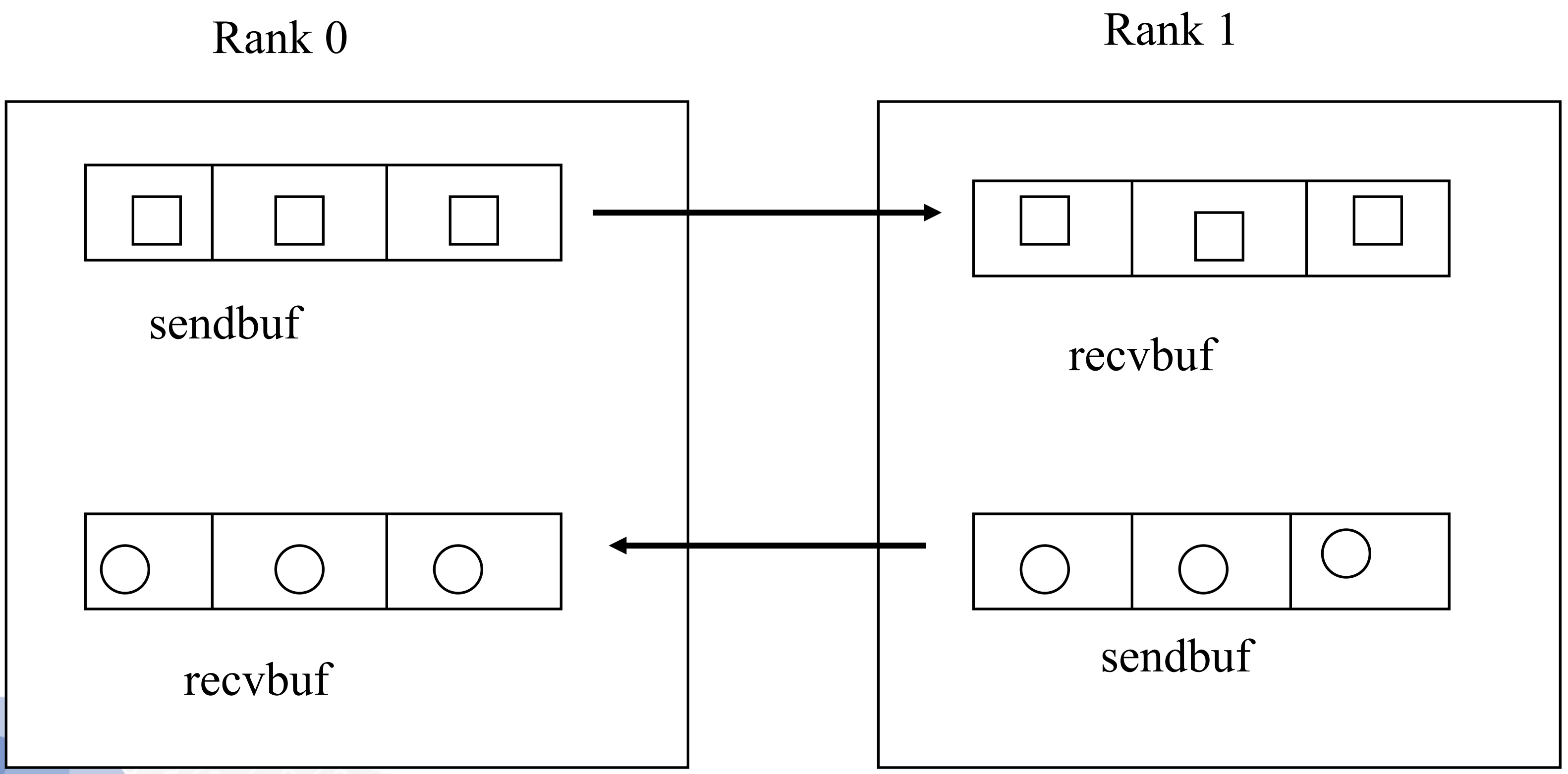
Fortran:

```
if (myrank == 0) then
    call MPI_Send(…)
elseif (myrank == 1) then
    call MPI_Recv(….)
endif
```

# Bidirectional Communications

## Deadlock

- Can occur due to incorrect order of send and receive
- Can occur due to limited size of the system buffer



## Deadlocked communication

Case 1 : both processes call recv first, then send

```
if (myrank == 0 ){
  MPI_Recv(….)
  MPI_Send (…)
}
else if (myrank == 1) {
  MPI_Recv(….)
  MPI_Send(….)
}
```

```
if (myrank == 0 ) then
  call MPI_Recv(….)
  call MPI_Send (…)
elseif (myrank == 1) then
  call MPI_Recv(….)
  call MPI_Send(….)
endif
```

The above will always lead to deadlock

# Deadlocked communication

Case 2 : both processes call send first, then recv

```
if (myrank == 0 ){
  MPI_Send(….)
  MPI_Recv (…)
}
else if (myrank == 1) {
  MPI_Send(….)
  MPI_Recv(….)
}
```

```
if (myrank == 0 ) then
  call MPI_Send(….)
  call MPI_Recv (…)
elseif (myrank == 1) then
  call MPI_Send(….)
  call MPI_Recv(….)
endif
```

This may not deadlock if the message size is small enough to fit in the receive buffer.

The size of the receive buffer may change with the total number of tasks or number of tasks per node.

Moral : There may be error in coding that only shows up for larger task counts or problem sizes.

## Send-Receive

MPI_Sendrecv: Sends and receives data in the same operation

- Call blocks until data is sent and received
- Usually used in shift or exchange operations

C

```
Ierr = MPI_Sendrecv(
  &sendbuffer, sendcount, sendtype, destination, sendtag,
  &recvbuffer, recvcount, recvtype, source,      recvtag,
  communicator, &status, ierr)
```

Fortran

```
call MPI_Sendrecv( &
   sendbuffer, sendcount, sendtype, destination, sendtag,
   recvbuffer, recvcount, recvtype, source,      recvtag,
   communicator, status, ierr)
```

# MPI_Sendrecv

| | |
|---|---|
| sendbuffer | data (address) |
| sendcount | Length of send array (in elements, 1 for scalars) |
| sendtype | Data Type: e.g. MPI_INT (C), MPI_INTEGER |
| destination | Rank (task #) of destination in communicator group |
| sendtag | Message identifier (arbitrary integer) |
| recvbuffer | data (address) |
| recvcount | Length of recv array (in elements, 1 for scalars) |
| recvtype | Data Type: e.g. MPI_INT (C), MPI_INTEGER |
| source | Rank (task #) of source in communicator group |
| sendtag | Message identifier (arbitrary integer) |
| communicator | Group of processors |
| Status | Status of the receiving operation |
| ierr | Error return (ONLY in Fortran) |

## Running Interactively

If you would like to follow along using the examples during the lecture, you may start an interactive session on Bridges or Comet.

Bridges:

```
# Monday
interact -p RM -N 1 -n 4 -t 4:00:00 -A ac560tp -R mpi
# Tuesday
interact -p RM -N 1 -n 4 -t 4:00:00 -A ac560tp -R mpi2
```

Comet:

```
srun -p compute -N 1 --ntasks-per-node=16 -t 4:00:00 \
--wait=0 --export=all --pty /bin/bash
```

## License