CME 213

SPRING 2012-2013

Eric Darve

MPI

Deadlocks

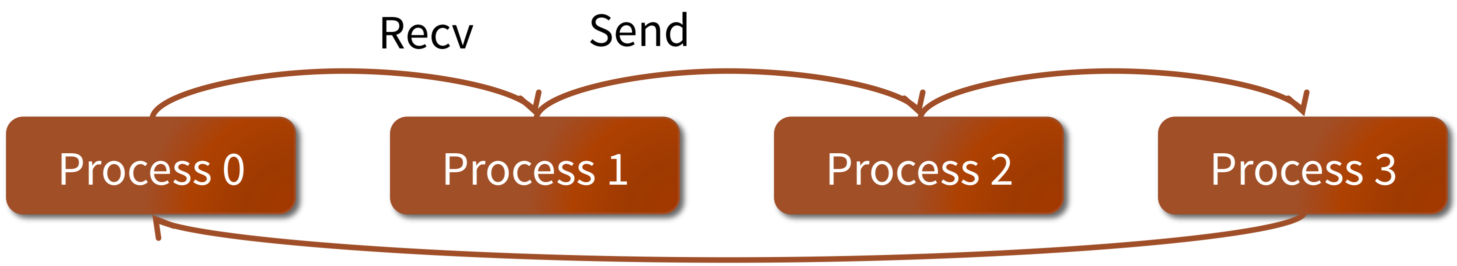
DEADLOCKS

Because we use blocking routines, deadlocks can occur:

Process 0	Process 1	Deadlock
Recv() Send()	Recv() Send()	Always
Send() Recv()	Send() Recv()	Depends on whether a buffer is used or not
Send() Recv()	Recv() Send()	Secure

- See MPI codes and diagram on next slide.
- Secure implementation: code is guaranteed to never deadlock; independent of whether buffers are used or not.

Ring communication




```
MPI_Sendrecv(void *sendbuf, int sendcount,
        MPI_Datatype sendtype, int dest, int sendtag,
        void *recvbuf, int recvcount,
        MPI_Datatype recvtype, int source, int recvtag,
        MPI_Comm comm, MPI_Status *status)
```

Non-Blocking communications

NON-BLOCKING VERSIONS OF SEND AND RECV

Replace: **MPI_send → MPI_Isend**

```
int MPI_Isend(void* buf, int count,
      MPI_Datatype datatype,
      int dest, int tag,
      MPI_Comm comm, MPI_Request *request)
```

MPI_Request* use to get information later on about the status of that operation.

What does I stand for?

Immediate

TESTING AND WAITING

There is a similar non-blocking receive:

```
int MPI_Irecv(void* buf, int count,
      MPI_Datatype datatype,
      int source, int tag,
      MPI_Comm comm, MPI_Request *request)
```

Test the status of the request using:

```
int MPI_Test(MPI_Request *request, int *flag,
      MPI_Status *status)
```

- Flag is 1 if request has been completed, 0 otherwise.

Wait until request completes:

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

Gather ring using non-blocking communication

step 1

$$\begin{tikzcd}
P_3\ \boxed{x_3} \arrow[d] & \boxed{x_2}\ P_2 \arrow[l] \\
P_0\ \boxed{x_0} \arrow[r] & \boxed{x_1}\ P_1 \arrow[u]
\end{tikzcd}$$

step 2

$$\begin{tikzcd}
P_3\ \boxed{x_2, x_3} \arrow[d] & \boxed{x_1, x_2}\ P_2 \arrow[l] \\
P_0\ \boxed{x_3, x_0} \arrow[r] & \boxed{x_0, x_1}\ P_1 \arrow[u]
\end{tikzcd}$$

step 3

$$\begin{tikzcd}
P_3\ \boxed{x_1, x_2, x_3} \arrow[d] & \boxed{x_0, x_1, x_2}\ P_2 \arrow[l] \\
P_0\ \boxed{x_2, x_3, x_0} \arrow[r] & \boxed{x_3, x_0, x_1}\ P_1 \arrow[u]
\end{tikzcd}$$

step 4

$$\begin{tikzcd}
P_3\ \boxed{x_0, x_1, x_2, x_3} \arrow[d] & \boxed{x_3, x_0, x_1, x_2}\ P_2 \arrow[l] \\
P_0\ \boxed{x_1, x_2, x_3, x_0} \arrow[r] & \boxed{x_2, x_3, x_0, x_1}\ P_1 \arrow[u]
\end{tikzcd}$$

See MPI code.

Communication modes

Stanford University

STANDARD MODE

- Standard: this the mode we have used up to now.
- This is in most cases sufficient.
- It provides good performance and relies on the MPI library for several optimizations.
- For example, MPI will decide whether or not buffers should be used.

SYNCHRONOUS MODE

- In synchronous mode, a send operation will be completed not before the corresponding receive operation has been started and the receiving process has started to receive the data sent.
- This leads to a form of synchronization between the sending and the receiving processes: the completion of a send operation in synchronous mode indicates that the receiver has started to store the message in its local receive buffer.
- Note: completion does not imply that the receiving node has finished receiving the data.

ALL OPTIONS ARE POSSIBLE

	Blocking	Non-blocking
Asynchronous	**MPI_Send, MPI_Recv** Blocking means that the buffer is usable when the subroutine returns. A system buffer may or may not be used by MPI.	**MPI_Isend, MPI Irecv** The subroutine returns immediately. Use Test() or Wait() to check status.
Synchronous	**MPI_Ssend** Returns when receive has been posted.	**MPI_Issend** Returns immediately. Test() and Wait() will consider that the communication is complete **only when the receive has been posted.**

Two more modes

Buffered mode:

- The user can allocate space for the MPI system buffer.
- This guarantees that a buffer is used.
- MPI_Bsend and MPI_Ibsend (non-blocking)

Ready mode:

- It can only be used if the user can guarantee that a matching receive has already been posted.
- The user is responsible for writing a correct program.
- Ready mode aims to minimize system overhead and synchronization overhead incurred by the sending task.
- MPI_Rsend and MPI_Irsend (non-blocking)

Which one should I choose?

- No perfect answer in general
- MPI_Ssend: in most cases, this call gives you the best performance. It allows MPI to completely avoid buffering. This requires that the processes are nearly synchronized, otherwise wait time results.
- MPI_Send: this allows the MPI implementation the maximum flexibility in choosing how to deliver your data. This is probably your best bet.
- If non-blocking is necessary (that is you are able to overlap communication and computation – this is not always possible), then consider using: MPI_Isend or MPI_Irecv.
- MPI_Bsend is essentially the same as MPI_Isend but you are forcing MPI to use a buffer.
- Other functions are less common.
- There is no MPI_Brecv: this would be a pointless function.

COLLECTIVE COMMUNICATION

NEED FOR COLLECTIVE COMMUNICATIONS

- There are many instances where collective communications are required, for example in a reduction.
- Since these are typical operations, MPI provides several functionalities that implement these operations.
- All these operations are blocking.

SINGLE BROADCAST

The simplest communication: one process sends a piece of data to all other processes.

```
int MPI_Bcast(void *message, int count,
      MPI_Datatype type, int root, MPI_Comm comm)
```

$$\begin{array}{l} P_1 : \boxed{x} \\ P_2 : \boxed{-} \\ \vdots \\ P_p : \boxed{-} \end{array} \quad \xRightarrow{broadcast} \quad \begin{array}{l} P_1 : \boxed{x} \\ P_2 : \boxed{x} \\ \vdots \\ P_p : \boxed{x} \end{array}$$

SINGLE ACCUMULATION

- Each process provides a block of data with the same type and size.
- When performing the operation, a reduction operation is applied element by element to the data blocks provided by the processes
- The resulting accumulated data block is collected at a specific root process.

```
int MPI_Reduce(void *sendbuf, void *recvbuf,
      int count, MPI_Datatype type, MPI_Op op,
      int root, MPI_Comm comm)
```

$$
\begin{array}{ll}
P_1 : \boxed{x_1} & \\
P_2 : \boxed{x_2} & \\
\vdots & \\
P_p : \boxed{x_p} &
\end{array}
\quad \overset{accumulation}{\Longrightarrow} \quad
\begin{array}{l}
P_1 : \boxed{x_1 + x_2 + \cdots + x_p} \\
P_2 : \boxed{x_2} \\
\vdots \\
P_p : \boxed{x_p}
\end{array}
$$

Representation	Operation
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bit-wise and
MPI_LOR	Logical or
MPI_BOR	Bit-wise or
MPI_LXOR	Logical exclusive or
MPI_BXOR	Bit-wise exclusive or
MPI_MAXLOC	Maximum value and corresponding index
MPI_MINLOC	Minimum value and corresponding index

CODE EXAMPLE

1. Computing π by throwing darts
2. Computing prime numbers in parallel.