Programming with Message Passing
PART I: Basics

HPC Fall 2012
Prof. Robert van Engelen
Overview

- Communicating processes
- MPMD and SPMD
- Point-to-point communications
  - Send and receive
  - Synchronous, blocking, and nonblocking message passing
  - Message selection
- Collective communications
  - broadcast, gather, scatter, barrier
- Further reading
Process Creation

- Processes communicate via message passing

How are processes created?

- **Static process creation**
  - All processes are specified before execution
  - Fixed number of processes executed
  - Example: `mpirun` command to start MPI program on \( n \) processors:
    \[
    \text{mpirun} \ -\text{np} \ n
    \]

- **Dynamic process creation**
  - Processes are created during the execution of other processes
  - Processes can fork new processes
  - Management (start/stop), synchronization, and communication are more difficult
MPMD Versus SPMD

Multiple Program Multiple Data (MPMD)

Single Program Multiple Data (SPMD)

Example: web server and web browsers

Example: MPI program
Basic Send and Receive

- Send and receive operations w/o source and destination process ID
  
  `send(&x)` send $x$ to any destination
  
  `recv(&y)` receive $y$ from any source

- Send and receive operations with source and destination process ID
  
  `send(&x, destID)` send $x$ to destination `destID`
  
  `recv(&y, srcID)` receive $y$ from source `srcID`

- Data type of $x$ and $y$ must match

- What about `rendezvous`?
  
  Should the sender wait until message is received by destination?
Synchronous and (non)Blocking Send Operations

- **Synchronous (also called blocking)**
  - Both sender and receiver wait until entire message is delivered

- **(Locally) blocking send**
  - Sender sends x and may continue operating on x
  - Copy of x is buffered (causing process to be temporarily suspended until copy is completed) or immediately transmitted (when x is small)
  - A receiver may accept message at any time

- **Nonblocking send**
  - Sender initiates a “send” of x and immediately continues
  - Sender cannot further operate on x (data x is in transfer state)
  - Receiver may accept message at any time
Blocking and Nonblocking Receive Operations

- **Blocking receive**
  - Receiver waits for data to be completely transferred

- **Nonblocking receive**
  - Receiver indicates it is ready to receive data into `y`
  - A *handle* is returned that allows the receiver to query the status of the received data for `y`

- Note: any type of send can be paired with any type of receive
Synchronous Send and Recv

send() occurs before recv()
P0 is suspended until a receiver is ready

recv() occurs before send()
P1 is suspended until a sender is ready
(non)Blocking Send andRecv

In a (locally) blocking `send()`, process P0 continues after the message is locally buffered or in transit to receiver, and it is safe for P0 to modify the data.

Blocking: P0 suspends until a `recv()` is posted.

In a nonblocking `send()`, process P0 immediately continues and executes while message is delivered (hides the messaging latency).

P0 cannot modify data in transit, explicitly probe message status or wait until message was received.
Deadlock

SPMD program

\[
\begin{align*}
\text{Process 1} & \\
A & := 0 \\
\text{for } i = 1..N/2 & \\
A & := A + f(i) \\
send A \text{ to } P2 & \\
receive B \text{ from } P2 & \\
A & := A + B
\end{align*}
\]

\[
\begin{align*}
\text{Process 2} & \\
A & := 0 \\
\text{for } i = 1..N/2 & \\
A & := A + f(i) \\
send A \text{ to } P1 & \\
receive B \text{ from } P1 & \\
A & := A + B
\end{align*}
\]

Deadlock with synchronous blocking send operations: both processors wait for data to be send to a receiver that is not ready to accept the message.

Note: nonblocking sends and sendrecv() operations (send-recev exchanges) are safe to use for this example.
Message Selection

Send and receive operations indicate source/destination process ID
- Id can be a wildcard

What if multiple messages are *asynchronously transmitted out-of-order* to a destination?
- Messages may be queuing up and end up being transmitted or accepted in different order, as if they “crossed” in transit
- Cannot rely on message ordering with blocking/nonblocking send, even when sends are initiated by one processes
- *Message tags* are used to match send and receive operations
  ```
  send(&x, destID, tag)
  recv(&y, srcID, tag)
  ```
  message is transferred when *tag* value matches
Broadcast

- **Multicast**: a root process sends a message to a specific subset of processes
- **Broadcast** = *multicast* within a process group
- First a group must be formed and root process selected
Scatter

- **Scatter**: a *root process* sends elements of an array $a[0,\ldots,n]$ to the enumerated processes $P_i$, $i=0,\ldots,n$
- First a group must be formed and root process selected
Gather

- **Gather**: a root process collects data from the enumerated processes $P_i, i=0,\ldots,n$ and puts them into the elements of an array $a[0,\ldots,n]$
- First a group must be formed and root process selected
Reduce

- **Reduce**: a root process collects data from the enumerated processes $P_i, i=0,\ldots,n$ and reduces it to a single value.
- First a group must be formed and root process selected.
AllGather and AllReduce

- AllGather and AllReduce: perform gather/reduce and broadcast result
- First a group must be formed and root process selected
Barrier

- **Barrier**: synchronization point

*Example barrier based on an allReduce (typically more efficient implementations are used)*
Processor Groups and Interconnect Topologies

- A processor group is a subset of all processors
  - Collective communications occur within a group
- A group (including the group of all processors) can be mapped to a virtual topology
  - When the virtual topology of a group is matched to a physical interconnect topology that is a close approximation of the virtual topology, message latencies are more predictable

*Group 1 with 1D Cartesian virtual topology*

*Group 2 with 2D Cartesian virtual topology*
Further Reading

- [PP2] pages 42-51