Programming with Message Passing
PART I

HPC Fall 2007
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 with 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:
    ```
    mpirun -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)

Example: web server and web browsers

Single Program Multiple Data (SPMD)

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 Id
  recv(&y, sourcelId) receive y from source Id

- 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
  - Sender waits until message x is buffered and/or transmitted (causing process to be suspended)
  - 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 (x is in transfer state)
  - Receiver may accept message at any time
Blocking and Nonblocking Receive Operations

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

- **Nonblocking receive**
  - Receiver indicates it is ready to receive
  - A *handle* is returned that allows the receiver to query the status of the message

- 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 \texttt{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.

P0 may suspend until a \texttt{recv()} is posted.

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

P0 cannot modify data and explicitly probes message status or waits until message was received.
**Deadlock**

**Process 1**

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

**Process 2**

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

**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.

*In this example, nonblocking sends and sendrecv() operations (exchanges) are safe*
Message Selection

- Send and receive operations indicate source/destination process ID
  - Id can be a wildcard
- What if multiple messages are asynchronously send 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, sourceld, 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 $P_i$, $i=0,...,n$ and puts them into the elements of an array $a[0,...,n]$.
- 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

interconnect topology
Further Reading

- [PP2] pages 42-51