• Skim
  – B. Barney (LLNL)
    » MPI tutorial and reference
Topics

• Short overview of basic issues in message passing

• MPI: A message-passing interface for distributed-memory parallel programming

• Collective communication operations
Basic Interprocess Communication

- Basic building block
  - message passing: send and receive operations between processes (address spaces)

How will this really be performed?
Synchronous Message Passing

- Communication upon synchronization
  - Hoare’s Communicating Sequential Processes (1978)

- BLOCKING send and receive operations
  - unbuffered communication
  - several steps in protocol
    » synchronization, data movement, completion
  - delays participating processes

```
<table>
<thead>
<tr>
<th>process P1</th>
<th>process P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>. . .</td>
<td>. . .</td>
</tr>
<tr>
<td>send m to P2</td>
<td>receive x from P1</td>
</tr>
<tr>
<td>. . .</td>
<td>. . .</td>
</tr>
</tbody>
</table>
```
Asynchronous Message Passing

• **Buffered communication**
  – send/receive via OS-maintained buffers
    » e.g. pipes or TCP connections
    » may increase concurrency (e.g. producer/consumer)
    » may increase transit time
  
  – send operation
    » send operation completes when message copied to buffer
    » generally non-blocking but may block if buffer full
  
  – receive operation – two flavors
    » **BLOCKING**
      • receive operation completes when message has been delivered
    » **NON-BLOCKING**
      • receive operation provides location for message
      • notified when receive complete (via flag or interrupt)
Asynchronous Message Passing

process P1

send m to P2

... (OS)
Buffering

(OS)
Buffering

process P2

... receive x from P1

...
Deadlock in message passing

• Can concurrent execution of P1 and P2 lead to deadlock?
  – assuming synchronous message passing?
  – assuming asynchronous message passing?

```plaintext
process P1

...  
send m1 to P2  
receive y from P2

...  

process P2

...  
send m2 to P1  
receive x from P1

...  
```
Non-determinism in Message Passing

- In what order should the receive operations be performed?

Two producers

process P1

... send m1 to P3 ...

process P2

... send m2 to P3 ...

One consumer

process P3

... receive x from ?

... receive y from ?

Here we want

receive x from any_process
receive y from any_process
Safe communication

• **MPI has four pairwise message passing modes**
  - **Synchronous**
    » unbuffered, but all send-receive pairs must synchronize
  - **Buffered (asynchronous)**
    » Programmer supplies (sufficient) buffer space
  - **Ready**
    » Receiver guaranteed to be ready to receive at the time of the send
  - **“Standard”**
    » OS Buffered for small messages, synchronous for large messages

• **Most programs rely on a certain amount of buffering in communication**
  - SPMD programming models: send, then receive
  - Nondeterminacy: receive from left, receive from right

• **Most programs use standard model**
  - Dangerous, as buffer size is system-dependent
Destination naming

• How are messages addressed to their receiver?
  – Static process to processor mapping
    » Fixed set of processes at compile time
    » *mapper* statically assigns processes to processors at run time.
    » Ex: Communicating Sequential Processes (CSP)
  
  – Semi-dynamic process to processor mapping (SPMD)
    » Unknown set of processes at compile time
    » Fixed set of processes at run time
    » fixed mapping over execution lifetime
    » Ex: MPI communicators
  
  – Dynamic process to processor mapping
    » Unknown set of processes at compile time
    » Processes may be created or moved dynamically at run time
    » Communication requires lookup
    » Ex: PVM, MPI-2
Data Representation

- In general, prefer to send an abstract data type (ADT) rather than single elements
  - ADTs represent abstractions suited to program
  - higher performance can be obtained for large messages
    » e.g. aggregate data types

- How are components of an ADT combined together?
  - data marshalling
    » packing components into a send buffer

- How is a message represented as a sequence of bits?
  - encoding must be suitable for source and destination
    » XDR (eXternal Data Representation)

- How is a message disassembled into an ADT?
  - data unmarshalling
    » extracting components from a receive buffer
Message Selection

- Receiving process may need to receive message from multiple potential senders

  - How to specify/distinguish message to be received?
    » sender selection (socket, MPI, CSP)
    » message data type selection (MPI, CSP)
    » condition selection (CSP)
    » message “tag” (MPI)

  - Specification of message to be received can decrease nondeterminacy
    » Non-deterministic reception order requires care with blocking sends/receives
Message Passing Interface (MPI)

- A library of communication operations for distributed-memory parallel programming
  - history
  - programming model
    » SPMD - single program with library calls
  - MPI functionality
    » send/receive, synchronization, collective communication
    » MPI 1 specifies 129 procedures
      • widely implemented and generally efficient
    » MPI 2 adds one-sided communication, dynamic processes, parallel I/O and more
      • One-sided communication: remote direct memory access – good for BSP.
      • Over 15 years from full specification to correct and (generally) efficient implementations
    » MPI-3
      • Tweaks and shared memory segments between MPI processes
  - portability
    » MPI is the most portable parallel programming paradigm – it runs on
      • shared and distributed memory
      • homogeneous and heterogeneous systems
      • variety of interconnection networks
    » BUT functional portability ≠ performance portability !
#include <mpi.h>

int main(int argc, char **argv) {
    int nproc, myid;

    MPI_Init (&argc, &argv);
    MPI_Comm_size (MPI_COMM_WORLD, &nproc);
    MPI_Comm_rank (MPI_COMM_WORLD, &myid);

    printf("Hello World! Here is process %d of %d.\n", myid, nproc);

    MPI_Finalize ();
}
MPI return codes

```c
#include <mpi.h>
#include <stdio.h>
#include <err.h>

int main(int argc, char **argv) {
    int nproc, myid, ierr;

    ierr = MPI_Init(&argc, &argv);
    if (ierr != MPI_SUCCESS) err(4, "Error %d in MPI_Init\n", ierr);

    ierr = MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    if (ierr != MPI_SUCCESS) err(4, "Error %d in MPI_Comm_size\n", ierr);

    ierr = MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    if (ierr != MPI_SUCCESS) err(4, "Error %d in MPI_Comm_rank\n", ierr);

    printf("Hello World! Here is process %d of %d.\n", myid, nproc);

    ierr = MPI_Finalize();
    if (ierr != MPI_SUCCESS) err(4, "Error %d in mpi_finalize\n", ierr);
}```
Point-to-point communication

• **Specification of message to receive**
  » communicator – identifies logical set of processors
    • intracommunicator vs. intercommunicator
  » sending process rank (= proc id)
  » tag
    – details of received message via status parameter
      » wildcard specifications may result in non-deterministic programs

• **Type Specification**
  – must provide types of transmitted values
    » predefined types & user-defined types
    » implicit conversions in heterogeneous systems

• **Protocol specification**
  – send
    » blocking / non-blocking / repeated / …
      • standard / buffered / synchronous / “ready”
Simple message exchange

- no deadlock
- two sequential transfers

#define MYTAG 123
#define WORLD MPI_COMM_WORLD

Process 0:

MPI_Send(A, 100, MPI_DOUBLE, 1, MYTAG, WORLD);
MPI_Recv(B, 100, MPI_DOUBLE, 1, MYTAG, WORLD);

Process 1:

MPI_Recv(B, 100, MPI_DOUBLE, 0, MYTAG, WORLD);
MPI_Send(A, 100, MPI_DOUBLE, 0, MYTAG, WORLD);
Non-blocking message exchange

• no deadlock

• possibility of concurrent transfer

```c
#define MYTAG 123
#define WORLD MPI_COMM_WORLD

MPI_Request request;
MPI_Status status;

Process 0:

MPI_Irecv(B, 100, MPI_DOUBLE, 1, MYTAG, WORLD, &request);
MPI_Send(A, 100, MPI_DOUBLE, 1, MYTAG, WORLD);
MPI_Wait(&request, &status);

Process 1:

MPI_Irecv(B, 100, MPI_DOUBLE, 0, MYTAG, WORLD, &request);
MPI_Send(A, 100, MPI_DOUBLE, 0, MYTAG, WORLD);
MPI_Wait(&request, &status);
```
Overlapping communication and computation

Process 0 and 1:

```c
#define MYTAG 123
#define WORLD MPI_COMM_WORLD

MPI_Request requests[2];
MPI_Status statuses[2];

// p is process id of the partner in a pairwise exchange
MPI_Irecv(B, 100, MPI_DOUBLE, p, 0, WORLD, &request[1]);
MPI_Isend(A, 100, MPI_DOUBLE, p, 0, WORLD, &request[0]);

.... do some useful work here ....

MPI_Waitall(2, requests, statuses);
```

- no deadlock
- concurrent transfer
- communication and computation may be overlapped on some machines
  - requires hardware communication support
Communicators

- **MPI_COMM_WORLD** is a communicator
  - group of processes numbered 0 ... p-1
  - set of private communication channels between them

- **Message sent with one communicator cannot be received in another communicator**
  - all communication is intra-communicator
  - enables development of safe libraries
  - restricting communication to subgroups is useful

- **Creating new communicators**
  - duplication
  - splitting

- **Intercommunicators**
  - orchestrate communication between two different communicators
Collective Communication

• **Operations involve all processes in an (intra)communicator**
  – encapsulate important communication patterns (cf. BSP)
    » broadcast
    » total exchange (transpose)
    » reduction + scan
    » barrier
  – operations do not necessarily imply a barrier synchronization
    » however, all processes must issue the same collective communication operations in the same order

• **Type specification**
  – predefined or user-defined types
  – predefined or user-defined associative operation for reduction & scan

• **Distinguished process**
  – for broadcast or reduction operations
Collective communication operations

• classified by
  – source of values
    » one/all processor(s)
  – target of result
    » one/all processors(s)
  – operation
    » broadcast
    » exchange
    » accumulate (reduce)
  – size of values
    » 1 or n

• duality of communication operations
  – communication patterns are related
  – broadcast & reduction are duals
  – exchange is its own dual

Ex:
source

target

operation

size of value

one-to-all broadcast (1)
Broadcast: single source, single value

one-to-all broadcast (1)

MPI_Bcast(...1...)

all-to-one sum (1)

MPI_Reduce(...1...)
Broadcast: single source, multiple values

one-to-all broadcast (n)

MPI_Bcast(...n...)

all-to-one sum (n)

MPI_Reduce(...n...)

\[ R_i = A_i \oplus B_i \oplus C_i \oplus D_i \]
Broadcast: multiple source, single value

all-to-all broadcast (1)

MPI_Allgater(...n...)

all-to-all sum (1)

MPI_Reduce_scatter(...n...)

\[ R_i = A_i \oplus B_i \oplus C_i \oplus D_i \]
Exchange: single source or single target

- **One-to-all exchange** \((n)\)
  
  \[
  \text{MPI} \_ \text{Scatter}( \ldots )
  \]

- **All-to-one exchange** \((1)\)
  
  \[
  \text{MPI} \_ \text{Gather}( \ldots )
  \]
Exchange: multiple source, multiple values

- **all-to-all exchange (n)**
  
  `MPI_Alltoall(...)`

  - BSP “total exchange” or transpose
Reductions: multiple source, multiple values

\[ R_i = A_i \oplus B_i \oplus C_i \oplus D_i \]

all-to-one sum (n)

MPI_Reduce(\ldots n\ldots)

all-to-all sum (1)

MPI_Reduce_scatter(\ldots n\ldots)

all-to-all sum (n)

MPI_Allreduce(\ldots n\ldots)