Overview: Message Passing

- MPI history
- message passing libraries: basic requirements and system building blocks
- message transfer semantics and calls
- communicators and collectives

Review of networks: what networks are there inside a node of Japan’s latest supercomputer

The Options

1. design special parallel programming language
   - Occam, Go
2. extend the syntax of an existing sequential language
   - CC++ (extension to C++), Fortran M, Linda (C or Fortran based)
3. use a standard sequential language with special library
   - most common, e.g. MPI, PVM, P4, TCGMSG

We will take option 3 and use MPI: see www.mpi-forum.org

MPI-1: Message Passing Interface

- parallel computer vendors initially developed own message-passing APIs
  - e.g. Fujitsu’s APLib for the AP1000 series (1991–1998)
  - issue: portability across machines difficult (especially with subtle differences in semantics)
- early work on a standard started in 1992 at Oak Ridge National and Rice Uni
- over 40 academic and government participants
- at that stage, there was a plethora of different message passing environments
- some deliberate exclusions from MPI-1:
  - IO and dynamic process creation
  - debugging and profiling tools were outside scope
- target was C and Fortran applications, although not strongly
- MPI-1 released in May 94
  - contains: point-to-point communications, collective operations, processor topology
- minor clarifications: MPI 1.1 (June 950, MPI 1.2 (July 97)

MPI-2

- initial discussions in Feb 94
- target areas: what was too hard for MPI1
- finished July 97
- just additions to MPI-1, not changes
  - dynamic process creation
  - one-sided communications
  - cooperative IO
  - C++/Fortran 90 binding
  - extended collective operations
  - minor other new functionality
- MPI 2.1 & 2.2 (Sept 2008 & Sept 2009) clarification and rationalization
- MPI 3.0, standard released Sept 2012, include issues like non-blocking collectives, one-sided operations, Fortran 2008 bindings. (not fault tolerance - too hard!)
Basic Requirement #1: Process Creation

Require a method for creating separate processes for execution on different CPUs

\[ \text{spawn(name of executable, where to run it)} \]

- **options:**
  - **static:** number of processes fixed throughout execution
  - **dynamic:** number of processes fluctuates during execution
- both require a means of identifying each process uniquely

Initial MPI standard (MPI-1) did not permit dynamic spawning of processes

number of processes defined by runtime environment, e.g. on the NCI system

\[ \text{mpirun -np 4 mpi job} \]

MPI provides a function to identify the number of processes and a unique identifier for each process (the rank of the process within the parallel group of processes):

```c
int MPI_Comm_size(MPI_Comm comm, int *size);
int MPI_Comm_rank(MPI_Comm comm, int *rank);
```

Basic Requirement #2: Data Transmission

Require method for sending/receiving messages between processes:

\[ \text{send(data, to where) and receive(data, from where)} \]

- **data**
  - usually a pointer and number of bytes
  - non-contiguous data must be packed
  - heterogeneous systems may require type conversion (e.g. big/little endian)
- **MPI send and receive calls:**

\[ \begin{align*}
\text{int MPI_Send(} & \text{void *buf, int count, MPI_Datatype datatype,}
\text{int dest, int tag, MPI_Comm comm);} \\
\text{int MPI_Recv(} & \text{void *buf, int count, MPI_Datatype datatype,}
\text{int source, int tag, MPI_Comm comm,}
\text{MPI_Status *status);}
\end{align*} \]

Rolling Your Own

UNIX provides you with all you need to build your own message passing library - all the useful stuff you learnt about in concurrent and distributed systems (COMP2310)!!

- **fork** - spawns an identical task to parent
- **ssh** - starts process on a remote machine
- **exec** - overwrites a process with a new process
- **sockets** - provide communication between machines
- **shmget** - provides communication within shared memory
- **xdr** - provides data conversion between machines

MPI implementations (MPICH, OpenMPI etc) use these utilities, e.g. on NCI Raijin system, CSIT labs

Message Transfer

Be careful of definitions:

- **synchronous:** the send only returns when message has been received, e.g. typical 3 message protocol:
  - request to send
  - receive the OK to send
  - send message
- **blocking:** the send returns when it is safe to re-use the sending buffer
  - locally blocking: returns after MPI copies the message into a local buffer
  - globally blocking: returns when receiver has collected the message (and hence has posted its receive call)

The receiver returns when message has been received.

- **non-blocking:** the send returns immediately even though the data may still be in the original buffer
  - another function call is used to check that the buffer is free to use again

The receive also returns immediately; another call is used to check for arrival.
Message Transfer

MPI Send and Receive Examples

1. consider a two-process MPI program, attempting send each other's a array:
   ```c
   char a[N]; int rank;
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   // initialize a, using rank
   MPI_Send(a, N, MPI_CHAR, 1-rank, 99, MPI_COMM_WORLD);
   MPI_Recv(a, N, MPI_CHAR, 1-rank, 99, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
   ```

   What would happen, if the send was locally blocking, globally-blocking, or non-blocking?

2. in the sockets API, what semantics does `send()` have? (similarly for Unix pipes, with `write()`)

Message Selection

- the sending process calls
  ```c
  int MPI_Send(void *buf, int count, MPI_Datatype datatype,
               int dest, int tag, MPI_Comm comm);
  ```

  - `dest`: where to send message, e.g. process 0,...,p − 1
  - a programmer-determined message tag can be used to create classes of messages

- the receiving process calls
  ```c
  int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
               int source, int tag, MPI_Comm comm,
               MPI_Status *status);
  ```

  - receive a message from given process (including MPI_ANY_SOURCE)
  - receive a message with given tag (including MPI_ANY_TAG)
  - buffer must be long enough for incoming message (!)

- both have blocking semantics (send is either local or global)

Communicators

- how to prevent conflict between messages that are internal to a library and those used by the application program?
- define a communicator: a group that processes can join

![Communicator Diagram]
Collective Operations

All the following can be constructed from simple sends and receives

- **synchronization**: barrier to inhibit further execution
  - e.g. use simple pingpong between two processors

- **broadcast**: send same message to many processors
  - must define processors in the group (specified by a communicator)
  - must define who sends and who received information
  - may or may not synchronize processors (implementation dependent)

- **scatter**: 1 process sends unique data to every other in group

- **gather**: reverse of above

- **reduction**: gather and combined with arithmetic/logical operation
  - result goes to just one process, or goes to all processes

However, MPI provides (usually highly optimized) calls to perform all of these, using communicators to specify which processes are involved.