MPI, the Message-Passing Interface: An Introduction

Timothy A. Chagnon and Tom Plick

November 1, 2007
MPI is a network protocol and API standard with several popular implementations for FORTRAN, C, C++

- Additional implementations exist for Python, Java, OCaml and others

An implementation consists of:

- library of calls to explicitly pass data between processes
  - `MPI_Send`, `MPI_Recv` etc.
- programs to spawn and manage multiple processes
  - `mpiexec`, `mpirun`

MPI provides many communication routines, but leaves it up to the programmer to specify the parallel execution and synchronization of distributed memory
History and Implementations

- MPI-1 in 1994
  - Started as MPI Forum at Supercomputing ’92
  - Broad collaboration from the parallel computing community
  - Defined most of the basic communication routines
  - Generally much more popular than MPI-2

- MPI-2 in 1997 added
  - Parallel file I/O
  - Remote memory operations
  - Dynamic process management

- First implemented as MPI-CH from Argonne Nat’l Laboratory
- LAM/MPI was created by the Ohio Supercomputing Center
- Open MPI is a new project derived from LAM/MPI

Single-program, multiple data

- SPMD: MPI favors this structure of parallelism
- Written as a single program with conditional execution based on process number
- Conditionals provide a combined SIMD/MIMD like environment
- Consider the following example of forking in C:

```c
int pid = fork();
if (pid){
    /* I am the parent process ... */
} else {
    /* I am the child process ... */
}
```
Starting an MPI program

- Issue the command `mpiexec -n 10 myprog`

- `mpiexec` starts 10 processes and provides each to find its `rank` (its ID from 0 to $n-1$)

```c
int n, myRank;
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &n);
MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
/* Continue based on myRank */ ...
```

- They can be spread across CPUs or hosts, as the MPI implementation allows
Basic Communications Routines

- Communication routines typically follow a common pattern

```
MPI_Send(&buf, n, type, dest, tag, communicator)
```

- **&buf** A pointer to the beginning of data to send
- **n** Number of elements to send
- **type** Datatype to send (MPI_INT, MPI_CHAR, etc.)
- **dest** Process rank to send data to
- **tag** User-defined tag to distinguish messages
  - Can be any integer
- **communicator** Communicator (aka Group) to use
  - Typically MPI_COMM_WORLD

[Leopold(2001)]
MPI Routines

- **Point-to-Point**
  - MPI_Send, MPI_Recv Blocking transfers
  - MPI_Isend, MPI_Irecv, Wait Non-blocking transfers

- **Collective Communication**
  - MPI_Bcast Send the same message to all group members
  - MPI_Scatter Split up message into blocks and send
  - MPI_Gather Collect part of message from each group member
  - MPI_Reduce Collect data and combine via an operation (e.g. sum)
  - MPI_Alltoall Combined Scatter and Gather all at once

- **Derived Data Types**
  - MPI_Type_vector Access data at a constant stride
  - MPI_Type_struct Combine datatypes from a struct
Communication Patterns

- Complete network on $n$ points
- Ring: process $i$ communicates with $i - 1$ and $i + 1$

- Mesh: a 2-D grid
- Binary tree
- Hypercube: each process communicates with $(\log n)$ others

Networks can also be embedded; for instance, we could have a ring of hypercubes, a mesh of rings, etc.

[Wilkinson and Allen (1998)]
Communicators

- Each process belongs to a group and communicates with its groupmates.
- A *communicator* allows a process to send a message to its groupmates and disregard other groups.
- `mpiexec` starts its processes in the communicator `MPI_COMM_WORLD`.
- A library might use other communicators, so as not to interfere with the user’s processes.
Work patterns

- Divide and conquer: Divide problem into \( n \) smaller problems; or, repeatedly halve the problem \((\log n)\) times, and assign each leaf to a process
  
  e.g. Parallel mergesort

- Work pool: A problem is divided into many small tasks; a process takes one, completes the task, takes another, etc.
  
  e.g. Calculating the Mandelbrot set: work pool consists of a set of points

- Pipeline: Dependencies between processes sequence the operations.
  
  e.g. A frequency filter: each process \( P_i \) accepts a signal and outputs the signal with a frequency \( f_i \) removed.

- Lock-step: There are several milestones that each process must attain before any can continue; use the MPI_BARRIER call, which blocks while necessary.
  
  e.g. Cellular automata: \( M_i \) is a function on entries of \( M_{i-1} \)
Performance Considerations

Generality, sometimes at the expense of performance...

In MPICH1, the original implementation of MPI:

- For broadcast, uses unicast messages, whereas IP multicast could be used (but requires further care)
- Portable, so suffers in comparison with a vendor-tuned implementation

Other trade-offs for implementors to consider:

- Communication: TCP/IP sockets are universal, but shared memory is faster
- Buffering: large buffers use memory; small buffers cause delay

The good news is that a programmer can test several versions of MPI without changing his or her code.
References

W. Gropp, E. Lusk, and A. Skjellum.
Using MPI (2nd ed.): portable parallel programming with the message-passing interface.
ISBN 0262571323.

C. Leopold.
Parallel and Distributed Computing: A Survey of Models, Paradigms and Approaches.
ISBN 0471358312.

B. Wilkinson and M. Allen.
ISBN 0136717101.