Message-Passing Libraries

CSE 791

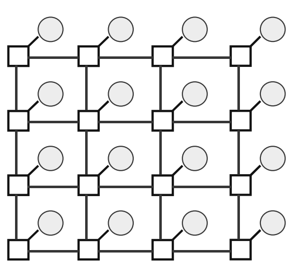
Fall 1999

The Duality

- Remember the duality between shared memory and message passing
- Shared memory programming is more natural, except…
  - ◆ scientific computing has long used grid-based decomposition
  - ◆ threads and synchronization don't come naturally to scientists
  - ◆ scalability

Message Passing

- Computers were built to reflect the computation structure (meshes)
- But they weren't the same
- machine-specific messaging
- need to know exactly who you're talking to
- ⇨long development time



Message Passing II

- The Attack of the Killer Micros was underway
- The result would be distributed systems
- We've lost the nice, regular structure of parallel machines
- What about clusters?

⇨A crisis was imminent

Problems with Message Passing

- Need to know exact location of each sender/recipient
- Need to write code to send/receive messages
- Need to manage queues of outstanding messages (or send synchronously)
- Need to manage heterogeneous data types

The Role of the Message-Passing Library

- Eliminate the need to know exact location of partner; use a local ID that is bound by the MP layer (indirect addressing)
- Use built in messaging routines
- Can tag messages to assist in management

Role of Message Passing Library (cont.)

- Automatically translate data formats (XDR)
- Allow group communication
- Automatic synchronization

Three Solutions

- Local Area Multicomputer (LAM) and Trollius -- Greg Burns, Cornell/OSC/ND
- Parallel Virtual Machine (PVM) -- Vaidy Sunderam (Emory), Al Geist (Oak Ridge), et al.
- Message Passing Interface Standard -- MPI Forum (http://www.mpi-forum.org)

LAM

- LAM provides a low-level message passing layer, designed to aggregate clusters of workstations
- Has both PVM and MPI interface layers available
- See http://www.mpi.nd.edu/lam

PVM

- The first large-scale solution with high-level messaging functions
- *NOT* a standard
- Essentially an interface and its implementation (LAM decoupled this)
- Includes dynamic process creation
- No scheduling/placement
- http://www.epm.ornl.gov/pvm

MPI

- Learned from what PVM did
- Developed standard interfaces (MPI!)
- No default implementation
  - LAM
  - MPICH
  - CHIMP
- No dynamic process creation
- No scheduling/placement

MPI (cont.)

- Many messaging systems now provide MPI implementations
  - Legion
  - specific implementations for fast message platforms
- MPI-2 does define dynamic process creation

Top 10 Reasons to Prefer MPI over PVM (Greg Burns)

1. MPI has more than one freely available, quality implementation.
   - There are at least LAM, MPICH and CHIMP. The choice of development tools is not coupled to the programming interface.
2. MPI defines a 3rd party profiling mechanism.
   - A tool builder can extract profile information from MPI applications by supplying the MPI standard profile interface in a separate library, without ever having access to the source code of the main implementation.

Top 10 (cont.)

3. MPI has full asynchronous communication.
   - Immediate send and receive operations can fully overlap computation.
4. MPI groups are solid, efficient, and deterministic.
   - Group membership is static. There are no race conditions caused by processes independently entering and leaving a group. New group formation is collective and group membership information is distributed, not centralized.

Top 10 (III)

5. MPI efficiently manages message buffers.
   - Messages are sent and received from user data structures, not from staging buffers within the communication library. Buffering may, in some cases, be totally avoided.
6. MPI synchronization protects the user from 3rd party software.
   - All communication within a particular group of processes is marked with an extra synchronization variable, allocated by the system. Independent software products within the same process do not have to worry about allocating message tags.

Top 10 (IV)

7. MPI can efficiently program MPP and clusters.
   - A virtual topology reflecting the communication pattern of the application can be associated with a group of processes. An MPP implementation of MPI could use that information to match processes to processors in a way that optimizes communication paths.
8. MPI is totally portable.
   - Recompile and run on any implementation. With virtual topologies and efficient buffer management, for example, an application moving from a cluster to an MPP could even expect good performance.

Top 10 (V)

9. MPI is formally specified.
   - Implementations have to live up to a published document of precise semantics.
10. MPI is a standard.
   - Its features and behaviour were arrived at by consensus in an open forum. It can change only by the same process.

How to Use MPI: Simple Master/Slave Skeleton

- Process 0 becomes the master
  - receives requests
  - distributes work to slaves
  - collects results
- All other processes are slaves
  - get work from master
  - return result
  - die when told to
- Skeleton because we don't do real work

Master/Slave Skeleton: Common Code

```
#include <mpi.h>

main(int argc, char *argv[])
{
   int                  myrank;

   MPI_Init(&argc, &argv);           /* initialize MPI */
   MPI_Comm_rank(MPI_COMM_WORLD,     /* always use this */
                &myrank);            /* process rank, 0 */
                                     /* thru N-1 */
    if (myrank == 0) {
       master();
    } else {
       slave()
    }
    MPI_Finalize();               /* cleanup MPI */
}
```


Master Code: Initialization

```
#define WORKTAG    1
#define DIETAG     2

master()
{
   int                  ntasks, rank, work;
   double               result;
   MPI_Status           status;

   MPI_Comm_size(MPI_COMM_WORLD, &ntasks);
        /* find out #processes in application */
```

Master Code: Seeding the Slaves

```
/*
 * Seed the slaves.
 */

for (rank = 1; rank < ntasks; ++rank) {
   work = /* get_next_work_request */;
   MPI_Send(&work,       /* message buffer */
            1,           /* one data item */
            MPI_INT,     /* data item is an integer */
            rank,        /* dest. process rank */
            WORKTAG,     /* user chosen message tag */
            MPI_COMM_WORLD);/* always use this */
}
```


Master Code: Receive Results Loop

```
/*
 * Receive a result from any slave and dispatch
 * a new work request until work requests have been
 * exhausted.
 */
work = /* get_next_work_request */;

while (/* valid new work request */) {
   MPI_Recv(&result,       /* message buffer */
            1,              /* one data item */
            MPI_DOUBLE,     /* of type double real */
            MPI_ANY_SOURCE, /* receive from anyone */
            MPI_ANY_TAG,    /* any type of message */
            MPI_COMM_WORLD, /* always use this */
            &status);       /* received message info */
```


Master Code: Main Loop Cont.

```
      MPI_Send(&work, 1, MPI_INT,
               status.MPI_SOURCE,
               WORKTAG, MPI_COMM_WORLD);

      work = /* get_next_work_request */;
   }
   /*
    * Receive results for outstanding work requests.
    */
   for (rank = 1; rank < ntasks; ++rank) {
      MPI_Recv(&result, 1, MPI_DOUBLE, MPI_ANY_SOURCE,
               MPI_ANY_TAG, MPI_COMM_WORLD, &status);
   }
```


Master Code: Cleanup

```
      /*
       * Tell all the slaves to exit.
       */
      for (rank = 1; rank < ntasks; ++rank) {
         MPI_Send(0, 0, MPI_INT, rank, DIETAG,
                  MPI_COMM_WORLD);
      }
   }  /* End of master() */
```

Slave Code: Initialization

```
#define WORKTAG    1
#define DIETAG     2

slave()
{
   double              result;
   int                 work;
   MPI_Status          status;
```


Slave Code: Main Loop

```
for (;;) {
   MPI_Recv(&work, 1, MPI_INT, 0, MPI_ANY_TAG,
                MPI_COMM_WORLD, &status);
   /*
    * Check the tag of the received message.
    */
   if (status.MPI_TAG == DIETAG) {
      return;
   }

   result = /* do the work */;

   MPI_Send(&result, 1, MPI_DOUBLE, 0, 0,
            MPI_COMM_WORLD);
   }
} /* End of Slave */
```