CMSC 714
Lecture 3
Message Passing with
PVM and MPI

Alan Sussman

Notes

- To access papers in ACM or IEEE digital library, must come from a UMD IP address
- Accounts handed out next week for deepthought2 cluster, used for all assignments
- First assignment (MPI) announced next week
- Check Readings page to see when you are assigned to send questions for a lecture
  - 2-4 questions on average, more is OK
  - by 6PM day before lecture

PVM

- Provide a simple, free, portable parallel environment
- Run on everything
  - Parallel Hardware: SMP, MPPs, Vector Machines
  - Network of Workstations: ATM, Ethernet,
    - UNIX machines and PCs running Win32 API
  - Works on a heterogenous collection of machines
    - handles type conversion as needed
- Provides two things
  - message passing library
    - point-to-point messages
    - synchronization: barriers, reductions
  - OS support
    - process creation (pvm_spawn)

PVM Environment (UNIX)

- One PVMD per machine
  - all processes communicate through pvmd (by default)
- Any number of application processes per node
PVM Message Passing

- All messages have tags
  - an integer to identify the message
  - defined by the user
- Messages are constructed, then sent
  - `pvm_pk(int,char,float)(*var, count, stride)`
  - `pvm_unpk(int,char,float)` to unpack
- All processes are named based on task ids (tids)
  - local/remote processes are the same
- Primary message passing functions
  - `pvm_send(tid, tag)`
  - `pvm_recv(tid, tag)`

PVM Process Control

- Creating a process
  - `pvm_spawn(task, argv, flag, where, ntask, tids)`
  - `task` is name of program to start
  - `flag` and `where` provide control of where tasks are started
  - `ntask` determines how many copies are started
  - program must be installed on each target machine
  - returns number of tasks actually started
- Ending a task
  - `pvm_exit`
  - does not exit the process, just the PVM machine
- Info functions
  - `pvm_mytid()` - get the process task id

PVM Group Operations

- Group is the unit of communication
  - a collection of one or more processes
  - processes join group with `pvm_joingroup("<group name>")`
  - each process in the group has a unique id
    - `pvm_gettid("<group name>")`
- Barrier
  - can involve a subset of the processes in the group
  - `pvm_barrier("<group name>", count)`
- Reduction Operations
  - `pvm_reduce( void (*func)(), void *data, int count, int datatype, int msgtag, char *group, int rootinst)`
    - result is returned to rootinst node
    - does not block
  - pre-defined funcs: PvmMin, PvmMax, PvmSum, PvmProduct

PVM Performance Issues

- Messages have to go through PVMD
  - can use `direct route` option to prevent this problem
- Packing messages
  - semantics imply a copy
  - extra function call to pack messages
- Heterogenous Support
  - information is sent in machine independent format
  - has a short circuit option for known homogenous comm.
    - passes data in native format then
Sample PVM Program

```c
int main(int argc, char **argv) {  
    if (argc==0) {  
        /* Initialize the message */  
        for (i=0; i<MESSAGESIZE; i++) {  
            message[i] = '1';  
        }  
    } else {  
        /* Now start passing the message back and forth */  
        for (i=0; i<ITERATIONS; i++) {  
            if (myGroupNum==0) {  
                pvm_initsend(PvmDataDefault);  
                pvm_pkint(message,MESSAGESIZE,1);  
                pvm_send(friendTid,msgid);  
                pvm_recv(friendTid,msgid);  
                pvm_upkint(message,MESSAGESIZE,1);  
                pvm_exit();  
            } else {  
                pvm_recv(friendTid,msgid);  
                pvm_upkint(message,MESSAGESIZE,1);  
                pvm_initsend(PvmDataDefault);  
                pvm_pkint(message,MESSAGESIZE,1);  
                pvm_send(friendTid,msgid);  
            }  
        }  
        // additional code...  
    }  
    exit(EXIT_SUCCESS);  
}
```

MPI

- **Goals:**
  - Standardize previous message passing:
    - PVM, P4, NX (Intel), MPL (IBM), ...
  - Support copy-free message passing
  - Portable to many platforms – defines an API, not an implementation

- **Features:**
  - point-to-point messaging
  - group/collective communications
  - profiling interface: every function has a name-shifted version

- **Buffering (in standard mode)**
  - no guarantee that there are buffers
  - possible that send will block until receive is called

- **Delivery Order**
  - two sends from same process to same dest. will arrive in order
  - no guarantee of fairness between processes on receive

 MPI Communicators

- Provide a named set of processes for communication
  - plus a context – system allocated unique tag

- All processes within a communicator can be named
  - a communicator is a group of processes and a context
  - numbered from 0...n-1

- Allows libraries to be constructed
  - application creates communicators
  - library uses it
  - prevents problems with posting wildcard receives
    - adds a communicator scope to each receive

- All programs start with MPI_COMM_WORLD
  - Functions for creating communicators from other communicators (split, duplicate, etc.)
  - Functions for finding out about processes within communicator (size, my_rank, ...)

Non-Blocking Point-to-point Functions

- **Two Parts**
  - post the operation
  - wait for results

- **Also includes a poll/test option**
  - checks if the operation has finished

- **Semantics**
  - must not alter buffer while operation is pending (wait returns or test returns true)
  - and data not valid for a receive until operation completes
Collective Communication

- Communicator specifies process group to participate
- Various operations, that may be optimized in an MPI implementation
  - Barrier synchronization
  - Broadcast
  - Gather/scatter (with one destination, or all in group)
  - Reduction operations – predefined and user-defined
    - Also with one destination or all in group
    - Scan – prefix reductions
- Collective operations may or may not synchronize
  - Up to the implementation, so application can’t make assumptions

MPI Calls

- Include <mpi.h> in your C/C++ program
- First call MPI_Init(&argc, &argv)
- MPI_Comm_rank(MPI_COMM_WORLD, &myrank)
  - myrank is set to id of this process (in range 0 to P-1)
- MPI_Wtime()
  - Returns wall time
- At the end, call MPI_Finalize()
  - No MPI calls allowed after this

MPI Communication

- Parameters of various calls (in later example)
  - var – a variable (pointer to memory)
  - num – number of elements in the variable to use
  - type {MPI_INT, MPI_REAL, MPI_BYTE, …}
  - root – rank of process at root of collective operation
  - src/dest – rank of source/destination process
  - status - variable of type MPI_Status;
- Calls (all return a code – check for MPI_Success)
  - MPI_Send(var, num, type, dest, tag, MPI_COMM_WORLD)
  - MPI_Recv(var, num, type, src, MPI_ANY_TAG, MPI_COMM_WORLD, &status)
  - MPI_Bcast(var, num, type, root, MPI_COMM_WORLD)
  - MPI_Barrier(MPI_COMM_WORLD)

MPI Misc.

- MPI Types
  - All messages are typed
    - base/primitive types are pre-defined:
      - int, double, real, {unsigned}{short, char, long}
    - can construct user-defined types
      - includes non-contiguous data types
- Processor Topologies
  - Allows construction of Cartesian & arbitrary graphs
  - May allow some systems to run faster
- Language bindings for C, Fortran, C++, …
- What’s not in MPI-1
  - process creation
  - I/O
  - one sided communication
Sample MPI Program

```c
#include "mpi.h"
int main(int argc, char **argv) {
    int myrank, friendRank;
    char message[MESSAGESIZE];
    int i, tag=MSG_TAG;
    MPI_Status status;

    /* Initialize, no spawning necessary */
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    if (myrank==0) { /* I am the first process */
        friendRank = 1;
    } else { /* I am the second process */
        friendRank = 0;
    }
    MPI_Barrier(MPI_COMM_WORLD);
    if (myrank==0) {
        /* Initialize the message */
        for (i=0 ; i<MESSAGESIZE ; i++) {
            if (myrank==0) {
                MPI_Send(message, MESSAGESIZE,
                        MPI_CHAR, friendRank, tag,
                        MPI_COMM_WORLD);
                MPI_Recv(message, MESSAGESIZE,
                          MPI_CHAR, friendRank, tag,
                          MPI_COMM_WORLD, &status);
            } else {
                MPI_Recv(message, MESSAGESIZE,
                          MPI_CHAR, friendRank, tag,
                          MPI_COMM_WORLD, &status);
                MPI_Send(message, MESSAGESIZE,
                          MPI_CHAR, friendRank, tag,
                          MPI_COMM_WORLD);
            }
        }
        MPI_Finalize();
        exit(0);
    }
}
```

For more details

  - current version is 3.4.6, available for download from netlib
  - book from MIT Press is *PVM: Parallel Virtual Machine A Users' Guide and Tutorial for Networked Parallel Computing*

- **MPI** – [http://www.mpi-forum.org](http://www.mpi-forum.org)
  - includes both 1.1 and 3.1 documentation (API)
  - books from MIT Press include *Using MPI* and *MPI: The Complete Reference*
  - multiple public domain implementations available
    - OpenMPI (formerly LAM) – large open source team – [http://www.open-mpi.org](http://www.open-mpi.org)
  - vendor implementations available too (IBM, Cray, …)
  - for deepthought2 cluster info, see [http://www.glue.umd.edu/hpcc/dt2.html](http://www.glue.umd.edu/hpcc/dt2.html)