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 emailed soon for UMIACS bug cluster and tau SUN SMP
- First assignment probably announced on Tuesday
Last time

- Coordination for parallel programs
  - synchronization
  - load balancing
- Control vs. data parallelism
- Metrics
  - Speedup - vs. best known serial algorithm
  - Scaled speedup
  - Amdahl’s law
  - Maximize computation to communication ratio
- Writing parallel programs
  - compiler converts old serial code
  - serial language plus communication library
  - new programming language
  - hybrid - old language with new constructs

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` controls how many copies are started
  - Program must be installed on 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) {
    int myGroupNum;
    int mytid;
    int tids[2];
    int message[MESSAGESIZE];
    int c,i,okSpawn;
    /* Initialize process and spawn if necessary */
    myGroupNum=pvm_joingroup("ping-pong");
    mytid=pvm_mytid();
    if (myGroupNum==0) { /* I am the first process */
        okSpawn=pvm_spawn(MYNAME,argv,0,"",1,&friendTid);
        if (okSpawn!=1) {
            printf("Can't spawn a copy of myself!
";
            pvm_exit();
            exit(1);
        }
        tids[0]=mytid;
        tids[1]=friendTid;
    } else { /*I am the second process */
        friendTid=pvm_parent();
        tids[0]=friendTid;
        tids[1]=mytid;
    }
    pvm_barrier("ping-pong",2);
    if (myGroupNum==0) {
        /* Initialize the message */
        for (i=0 ; i<MESSAGESIZE ; i++) {
            message[i]=i;
        }
        /* 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);
            } else {
                pvm_recv(friendTid,msgid);
                pvm_upkint(message,MESSAGESIZE,1);
                pvm_initsend(PvmDataDefault);
                pvm_pkint(message,MESSAGESIZE,1);
                pvm_send(friendTid,msgid);
            }
        }
        pvm_exit();
        exit(0);
    }
    pvm_exit();
    exit(0);
}
```
Notes

- **Accounts are for bug cluster**
  - login nodes are {brood01, ..., brood06}.umiacs.umd.edu
  - cluster nodes are {bug01, ..., bug64}
    - each a dual processor Pentium IV, 2GB memory
  - space shared, so use qsub, qstat, etc. to submit and check jobs - more info on cluster web pages linked from project spec

- **Project due Friday, Sept. 28, 6PM, via email**
- **No class Thursday**
- **Guest lecturer next Tuesday, Chau-Wen Tseng, on UPC and OpenMP**

---

**MPI**

- **Goals:**
  - Standardize previous message passing:
    - PVM, P4, NX, MPL, ...
  - Support copy-free message passing
  - Portable to many platforms

- **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
  - 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
  - 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
  - dest – rank of 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, dest, 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++) {
            message[i]='1';
        }
    }
    else {
        /* Initialize the message */
        for (i=0 ; i<MESSAGESIZE ; i++) {
            message[i]='1';
        }
    }
    /* Now start passing the message back and forth */
    for (i=0 ; i<ITERATIONS ; 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 3.4.5, 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 2.0 documentation (API)
  - books from MIT Press include *Using MPI* and *MPI: The Complete Reference*
  - multiple public domain implementations available
    - mpich – Argonne National Lab
    - OpenMPI (formerly LAM) – [http://www.open-mpi.org](http://www.open-mpi.org)
  - vendor implementations available too (IBM, Cray, Sun, …)
  - for UMIACS Linux cluster info, see [http://www.umiacs.umd.edu/research/parallel/](http://www.umiacs.umd.edu/research/parallel/) - start with Class guide link – you’ll be using the bug cluster