Lecture 13
MPI

EN 600.320/420

Instructor: Randal Burns

28 March 2012

Department of Computer Science, *Johns Hopkins University*

MPI

- MPI = Message Passing Interface
  - Message passing parallelism
  - Cluster computing (no shared memory)
  - Process (not thread oriented)
- Parallelism model
  - SPMD: by definition
  - Also implement: master/worker, loop parallelism
- MPI environment
  - Application programming interface
  - Implemented in libraries
  - Multi-language support (most frequently C/C++ and Fortran)

The (Not So?) Big Deal

- Process groups
  - Set of processes conducting the same task (SMPD group)
- Communication contexts
  - Scope delivery to process group
  - Even when same sender, receiver, and tag
  - Like namespaces for messaging
- So what
  - Can write reusable parallel code (libraries)
  - Can use parallel libraries together
  - Run time system can dynamically deliver messages without permanently allocating contexts between send/receive pairs

Managing the runtime environment

- Initialize the environment
  - `MPI_Init ( &argc, &argv )`
- Acquire information for process
  - `MPI_Comm_size ( MPI_COMM_WORLD, &num_procs )`
  - `MPI_Comm_rank ( MPI_COMM_WORLD, &ID )`
  - To differentiate process behavior in SMPD
- And cleanup
  - `MPI_Finalize()`
- Some MPI instances leave orphan processes around
  - `MPI_Abort()`
  - Don't rely on this

A Simple MPI Program

- Configure the MPI environment
- Discover yourself
- Take some differentiated activity

See mpimsg.c

- Idioms
  - SPMD: all processes run the same program
  - MPI_Rank: tell yourself apart from other and customize the local processes behaviours
    - Find neighbors, select data region, etc.

Point-to-Point Messaging

- Blocking I/O
  - Blocking provides built in synchronization
  - Blocking leads to deadlock
- Send and receive, let's do an example

See deadlock.c

What's in a message?

- First three arguments specify content

```
int MPI_Send (
    void* sendbuf,
    int count,
    MPI_Datatype datatype,
    . . . )
```

- All MPI data are arrays
  - Where is it?
  - How many?
  - What type?

MPI Datatypes

Table 3.1 Some Predefined MPI Datatypes

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG	signed long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Deadlock in MPI Messaging

- Synchronous: the caller waits on the message to be delivered prior to returning
  - *So why didn't our program deadlock?*

Deadlock in MPI Messaging

- Synchronous: the caller waits on the message to be delivered prior to returning
  - *So why didn't our program deadlock?*
- Blocking ***standard*** send may be implemented by the MPI runtime in a variety of ways
  - `MPI_Send( …, MPI_COMM_WORLD )`
  - Buffered at sender or receiver
  - Depending upon message size, number of processes
- Converting to a mandatory synchronous send reveals the deadlock
  - `MPI_Ssend( …, MPI_COMM_WORLD )`
  - But so could increasing the # of processors

Standard Mode

- MPI runtime chooses best behavior for messaging based on system/message parameters:
  - Amount of buffer space
  - Message size
  - Number of processors
- Preferred way to program??
  - Commonly used and realizes good performance
  - System take available optimizations
- Can lead to horrible errors
  - Because semantics/correctness changes based on job configuration. **Dangerous!**

Avoiding Deadlock

- Conditions for deadlock
  - Two processes
  - Two resources
  - Opposition
- More generally: cycles in a resource dependency graph
- Avoiding deadlock in MPI
  - Create cycle-free messaging disciplines
  - Synchronize actions

See passitforward.c

Messaging Topologies

- Order/pair sends and receives to avoid deadlocks
- For linear orderings and rings
  - Simplest and sufficient: (n-1) send/receive, 1 receive/send
  - More parallel, alternate send/receive and receive/send
- For more complex communication topologies?
- Messaging topology dictates parallelism
  - Important part of parallel design

How about asynchronous I/O?

- MPI has support for non-blocking I/O
  - Send/recv request (returns as soon as resources allocated)
  - `MPI_Isend( … )`
  - Do some useful work
  - `MPI_Wait( &request, &status ) //finalize`

- `MPI_Wait:` await the completion of operation
- `MPI_Test:` check the completion of operation and return immediately

- Program must leave buffer intact until completion!
  - Tie up memory in application space
  - Source of errors

Asychronous I/O Useful?

- Forces for:
  - Overlap communication with computation
- Forces against:
  - Ties up buffers
  - Complex code
  - Little overlap available for time-step synchronous programs

- Use as a last resort
  - Remember the runtime is trying to do this for you

Synchronization

- Implicit synchronization (blocking send/receives)
  - Most common model
  - Allows for fine-grained dependency resolution

- Explicit synchronization (barriers)
  - `MPI_Barrier ( MPI_COMM_WORLD )`
  - All processes must enter barrier before any continue
  - Coarse-grained stops all
  - Common when interacting with shared resources, e.g. parallel file systems or shared-memory (when available)

Barriers vs. Send/Receive

- Barriers are useful when awaiting a global condition:
  - Data ready
  - Previous pipeline complete
  - Library call finished
  - Checkpoint written
- But, not a good replacement for pairwise sends and receives
  - They allow nodes to complete whenever their local synchronization constraints are met
  - Barriers are global and create global stalls (Next Monday)