COMP 322: Principles of Parallel Programming

Lecture 21: Advanced MPI

Fall 2009

http://www.cs.rice.edu/~vsarkar/comp322

Vivek Sarkar
Department of Computer Science
Rice University
vsarkar@rice.edu
Acknowledgments for today’s lecture

• Slides for today’s lecture were taken from the “Advanced MPI” tutorial given at Supercomputing 2007 conference, William Gropp, Rusty Lusk, Rob Ross, Rajeev Thakur, November 2007
Our Approach in this Tutorial

- Example driven
  - Structured data (Life)
  - Unpredictable communication (pNeo)
  - Passive target RMA (global arrays and MPI mutex)
Conway’s Game of Life

• A cellular automata
  – Described in 1970 Scientific American
  – Many interesting behaviors; see:

• Program issues are very similar to those for codes that use regular meshes, such as PDE solvers
  – Allows us to concentrate on the MPI issues
Rules for Life

- Matrix values $A(i,j)$ initialized to 1 (live) or 0 (dead)
- In each iteration, $A(i,j)$ is set to
  - 1 (live) if either
    - the sum of the values of its 8 neighbors is 3, or
    - the value was already 1 and the sum of its 8 neighbors is 2 or 3
  - 0 (dead) otherwise
Implementing Life

- For the non-parallel version, we:
  - Allocate a 2D matrix to hold state
    - Actually two matrices, and we will swap them between steps
  - Initialize the matrix
    - Force boundaries to be “dead”
    - Randomly generate states inside
  - At each time step:
    - Calculate each new cell state based on previous cell states (including neighbors)
    - Store new states in second matrix
    - Swap new and old matrices
Steps in Designing the Parallel Version

- Start with the “global” array as the main object
  - Natural for output - result we're computing
- Describe decomposition in terms of global array
- Describe communication of data, still in terms of the global array
- Define the “local” arrays and the communication between them by referring to the global array
Step 1: Description of Decomposition

• By rows (1D or row-block)
  — Each process gets a group of adjacent rows
• Later we’ll show a 2D decomposition
Step 2: Communication

- “Stencil” requires read access to data from neighbor cells

- We allocate extra space on each process to store neighbor cells

- Use send/recv or RMA to update prior to computation
Step 3: Define the Local Arrays

- Correspondence between the local and global array
- “Global” array is an abstraction; there is no one global array allocated anywhere
- Instead, we compute parts of it (the local arrays) on each process
- Provide ways to output the global array by combining the values on each process (parallel I/O!)
Boundary Regions

- In order to calculate next state of cells in edge rows, need data from adjacent rows.
- Need to communicate these regions at each step.
  - First cut: use isend and irecv.
  - Revisit with RMA later.
Point-to-Point Exchange

- Duplicate communicator to ensure communications do not conflict
  - This is good practice when developing MPI codes, but is not required in this code
  - If this code were made into a component for use in other codes, the duplicate communicator would be required

- Non-blocking sends and receives allow implementation greater flexibility in passing messages
Exchanging Data with RMA
Revisiting Mesh Communication

- Recall how we designed the parallel implementation
  - Determine source and destination data
- Do not need full generality of send/receive
  - Each process can completely define what data needs to be moved to itself, relative to each processes local mesh
    - Each process can “get” data from its neighbors
  - Alternately, each can define what data is needed by the neighbor processes
    - Each process can “put” data to its neighbors
Remote Memory Access

- Separates data transfer from indication of completion (synchronization)
- In message-passing, they are combined

<table>
<thead>
<tr>
<th>Proc 0</th>
<th>Proc 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>store</td>
<td>send</td>
</tr>
<tr>
<td></td>
<td>receive</td>
</tr>
<tr>
<td>load</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Proc 0</th>
<th>Proc 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>fence</td>
<td>fence</td>
</tr>
<tr>
<td>put</td>
<td>put</td>
</tr>
<tr>
<td>fence</td>
<td>fence</td>
</tr>
<tr>
<td>load</td>
<td>load</td>
</tr>
<tr>
<td>or</td>
<td>or</td>
</tr>
</tbody>
</table>

or

<table>
<thead>
<tr>
<th>Proc 0</th>
<th>Proc 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>store</td>
<td>fence</td>
</tr>
<tr>
<td>fence</td>
<td>get</td>
</tr>
</tbody>
</table>
Remote Memory Access in MPI-2
(also called One-Sided Operations)

• **Goals of MPI-2 RMA Design**
  - Balancing efficiency and portability across a wide class of architectures
    - shared-memory multiprocessors
    - NUMA architectures
    - distributed-memory MPP's, clusters
    - Workstation networks
  - Retaining “look and feel” of MPI-1
  - Dealing with subtle memory behavior issues: cache coherence, sequential consistency
Remote Memory Access Windows and Window Objects

- Process 0
- Process 1
- Process 2
- Process 3

Get

Put

window

= address spaces

= window object
Basic RMA Functions for Communication

- **MPI_Win_create** exposes local memory to RMA operation by other processes in a communicator
  - *Collective operation*
  - *Creates window object*
- **MPI_Win_free** deallocates window object

- **MPI_Put** moves data from local memory to remote memory
- **MPI_Get** retrieves data from remote memory into local memory
- **MPI_Accumulate** updates remote memory using local values
- Data movement operations are non-blocking
- Subsequent synchronization on window object needed to ensure operation is complete
Why Use RMA?

- Potentially higher performance on some platforms, e.g., SMPs
- Details later
Advantages of RMA Operations

• Can do multiple data transfers with a single synchronization operation
  —like BSP model

• Bypass tag matching
  —effectively precomputed as part of remote offset

• Some irregular communication patterns can be more economically expressed

• Can be significantly faster than send/receive on systems with hardware support for remote memory access, such as shared memory systems
Irregular Communication Patterns with RMA

- If communication pattern is not known up front, the send-receive model requires an extra step to determine how many sends-receives to issue.
- RMA, however, can handle it easily because only the origin or target process needs to issue the put or get call.
- This makes dynamic communication easier to code in RMA.
RMA Window Objects

MPI_Win_create(base, size, disp_unit, info, comm, win)

- Exposes memory given by (base, size) to RMA operations by other processes in comm
- win is window object used in RMA operations
- disp_unit scales displacements:
  - 1 (no scaling) or sizeof(type), where window is an array of elements of type type
  - Allows use of array indices
  - Allows heterogeneity
RMA Communication Calls

- MPI_Put - stores into remote memory
- MPI_Get - reads from remote memory
- MPI_Accumulate - updates remote memory
- All are non-blocking: data transfer is described, maybe even initiated, but may continue after call returns
- Subsequent synchronization on window object is needed to ensure operations are complete
Put, Get, and Accumulate

- `MPI_Put(origin_addr, origin_count, origin_datatype, target_rank, target_offset, target_count, target_datatype, window)`
- `MPI_Get( ... )`
- `MPI_Accumulate( ..., op, ... )`
- `op` is as in `MPI_Reduce`, but no user-defined operations are allowed
The Synchronization Issue

- **Issue**: Which value is retrieved?
  - Some form of synchronization is required between local load/stores and remote get/put/accumulates
- **MPI** provides multiple forms
Synchronization with Fence

Simplest methods for synchronizing on window objects:

- MPI_Win_fence - like barrier, supports BSP model

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Win_fence(win)</td>
<td>MPI_Win_fence(win)</td>
</tr>
<tr>
<td>MPI_Put</td>
<td>MPI_Put</td>
</tr>
<tr>
<td>MPI_Put</td>
<td></td>
</tr>
<tr>
<td>MPI_Win_fence(win)</td>
<td>MPI_Win_fence(win)</td>
</tr>
</tbody>
</table>
Mesh Exchange Using MPI RMA

• Define the windows
  — *Why* - safety, options for performance (later)

• Define the data to move

• Mark the points where RMA can start and where it must complete (e.g., fence/put/put/fence)
Outline of 1D RMA Exchange

- Create Window object
- Computing target offsets
- Exchange operation
Computing the Offsets

- **Offset to top ghost row**
  
  $1$

- **Offset to bottom ghost row**
  
  $1 + (\# \text{ cells in a row}) \times (\# \text{ of rows} - 1)$
  
  $= 1 + (\text{cols} + 2) \times (e - s + 2)$
Fence Life Exchange Code Walkthrough

- Points to observe
  - `MPI_Win_fence` is used to separate RMA accesses from non-RMA accesses
    - Both starts and ends data movement phase
  - Any memory may be used
    - No special malloc or restrictions on arrays
  - Uses same exchange interface as the point-to-point version
  - Two `MPI_Win` objects are used, one for the current patch, one for next (and last) iteration's patch
    - For this example, we could use a single `MPI_Win` for the entire patch
    - To simplify the evolution of this code to a two-dimensional decomposition, we use two `MPI_Win` objects
More on Fence

• **MPI_Win_fence** is collective over the group of the window object

• **MPI_Win_fence** is used to *separate*, not just complete, RMA and local memory operations
  
  — That is why there are *two* fence calls

• Why?
  
  — **MPI RMA** is designed to be portable to a wide variety of machines, including those without cache coherent hardware (including some of the fastest machines made)
  
  — See performance tuning for more info
Scalable Synchronization with Post/Start/Complete/Wait

- Fence synchronization is not scalable because it is collective over the group in the window object.
- MPI provides a second synchronization mode: Scalable Synchronization
  - Uses four routines instead of the single MPI_Win_fence:
    - 2 routines to mark the begin and end of calls to RMA routines
      MPI_Win_start, MPI_Win_complete
    - 2 routines to mark the begin and end of access to the memory window
      MPI_Win_post, MPI_Win_wait
- P/S/C/W allows synchronization to be performed only among communicating processes
Synchronization with P/S/C/W

- Origin process calls MPI_Win_start and MPI_Win_complete
- Target process calls MPI_Win_post and MPI_Win_wait

```plaintext
Process 0

MPI_Win_start(target_grp)   MPI_Win_post(origin_grp)
MPI_Put
MPI_Put

MPI_Win_complete(target_grp) MPI_Win_wait(origin_grp)
```
• **Points to Observe**
  
  — Use of MPI group routines to describe neighboring processes
  — No change to MPI_Put calls
  - You can start with MPI_Win_fence, then switch to P/S/C/W calls if necessary to improve performance
pNeo - Modeling the Human Brain
Science Driver

- **Goal:** Understand conditions, causes, and possible corrections for epilepsy
- **Approach:** Study the onset and progression of epileptiform activity in the neocortex
- **Technique:** Create a model of neurons and their interconnection network, based on models combining wet lab measurements of resected tissue samples and *in vivo* studies
- **Computation:** Develop a simulation program that can be used for detailed parameter studies
Model Neurons

Excitatory and inhibitory signal wiring between neurons

Neurons in the focal neocortex

Compartmental neural models
Modeling Approach

- Individual neurons are modeled using electrical analogs to parameters measured in the laboratory.
- Differential equations describe evolution of the neuron state variables.
- Neuron spiking output is wired to thousands of cells in a neighborhood.
- Wiring diagram is based on wiring patterns observed in neocortex tissue samples.
- Computation is divided among available processors.

Schematic of a two dimensional patch of neurons showing communication neighborhood for one of the cells in the simulation and partitioning of the patch among processors.
Abstract pNeo for Tutorial Example

• “Simulate the simulation” of the evolution of neuron state instead of solving the differential equations
• Focus on how to code the interactions between cells in MPI
• Assume one cell per process for simplicity
  — Real code multiplexes many individual neurons onto one MPI process
What Happens In Real Life

- Each cell has a fixed number of connections to some other cells
- Cell “state” evolves continuously
- From time to time “spikes” arrive from connected cells.
- Spikes influence the evolution of cell state
- From time to time the cell state causes spikes to be sent to other connected cells
What Happens In Existing pNeo Code

• In pNeo, each cell is connected to about 1000 cells
  – Large runs have 73,000 cells
  – Brain has ~100 billion cells

• Connections are derived from neuro-anatomical data

• There is a global clock marking time steps

• The state evolves according to a set of differential equations

• About 10 or more time steps between spikes
  – I.e., communication is unpredictable and sparse

• Possible MPI-1 solutions
  – Redundant communication of communication pattern before communication itself, to tell each process how many receives to do
  – Redundant “no spikes this time step” messages

• MPI-2 solution: straightforward use of Put, Fence
What Happens in Tutorial Example

• There is a global clock marking time steps
• At the beginning of a time step, a cell notes spikes from connected cells (put by them in a previous time step).
• A dummy evolution algorithm is used in place of the differential equation solver.
• This evolution computes which new spikes are to be sent to connected cells.
• Those spikes are sent (put), and the time step ends.
• We show both a Fence and a Post/Start/Complete/Wait version.
Two Examples Using RMA

• **Global synchronization**
  - Global synchronization of all processes at each step
  - Illustrates Put, Get, Fence

• **Local synchronization**
  - Synchronization across connected cells, for improved scalability (synchronization is local)
  - Illustrates Start, Complete, Post, Wait
pNeo Code Walkthrough

• Points to observe
  — Data structures can be the same for multiple synchronization approaches

• Code is simple compared to what a send/receive version would look like
  — Processes do no need to know which other processes will send them spikes at each step
Passive Target RMA
Active vs. Passive Target RMA

• *Active target* RMA requires participation from the target process in the form of synchronization calls (fence or P/S/C/W)

• In *passive target* RMA, target process makes no synchronization call
Passive Target RMA

• We need to indicate the beginning and ending of RMA calls by the process performing the RMA
  — This process is called the origin process
  — The process being accessed is the target process

• For passive target, the begin/end calls are
  — MPI_Win_lock, MPI_Win_unlock
Synchronization for Passive Target RMA

- **MPI_Win_lock(locktype, rank, assert, win)**
  - **Locktype is**
    - **MPI_LOCK_EXCLUSIVE**
      One process at a time may access
      Use when modifying the window
    - **MPI_LOCK_SHARED**
      Multiple processes (as long as none hold **MPI_LOCK_EXCLUSIVE**)
      Consider using when using MPI_Get (only) on the window
      - **Assert is either 0 or MPI_MODE_NOCHECK**

- **MPI_Win_unlock(rank, win)**

- **Lock is not a real lock but means begin-RMA; unlock is end-RMA, not real unlock**
Put with Lock

if (rank == 0) {
    MPI_Win_lock(MPI_LOCK_EXCLUSIVE, 1, 0, win);
    MPI_Put(outbuf, n, MPI_INT, 1, 0, n, MPI_INT, win);
    MPI_Win_unlock(1, win);
}

• Only process performing MPI_Put makes MPI RMA calls
  —Process with memory need not make any MPI calls; it is “passive”
• Similarly for MPI_Get, MPI_Accumulate
Global Arrays

• Let's look at updating a single array, distributed across a group of processes
A Global Distributed Array

- Problem: Application needs a single, 1-dimensional array that any process can update or read
- Solution: Create a window object describing local parts of the array, and use MPI_Put and MPI_Get to access

- Each process has a local\([n]\)
- We must provide access to \(a[pn]\)
- We cannot use MPI_Win_fence; we must use MPI_Win_lock and MPI_Win_unlock
Creating the Global Array

volatile double *locala;
...
MPI_Alloc_mem(n * sizeof(double), MPI_INFO_NULL, &locala);
MPI_Win_create(locala, n * sizeof(double),
sizeof(double),
MPI_INFO_NULL, comm, &win);
• **MPI-2** allows “global” to be relative to a communicator, enabling hierarchical algorithms
  —i.e., “global” does not have to refer to MPI_COMM_WORLD
• **MPI_Alloc_mem** is required for greatest portability
  —Some MPI implementations may allow memory not allocated with MPI_Alloc_mem in passive target RMA operations
Accessing the Global Array From a Remote Process

- **To update:**
  
  ```c
  rank = i / n;
  offset = i % n;
  MPI_Win_lock(MPI_LOCK_EXCLUSIVE, rank, 0, win);
  MPI_Put(&value, 1, MPI_DOUBLE,
           rank, offset, 1, MPI_DOUBLE, win);
  MPI_Win_unlock(rank, win);
  ```

- **To read:**
  
  ```c
  rank = i / n;
  offset = i % n;
  MPI_Win_lock(MPI_LOCK_SHARED, rank, 0, win);
  MPI_Get(&value, 1, MPI_DOUBLE,
           rank, offset, 1, MPI_DOUBLE, win);
  MPI_Win_unlock(rank, win);
  ```
Accessing the Global Array From a Local Process

- The issues
  - Cache coherence (if no hardware)
  - Data in register

- To read:
```c
volatile double *locala;
rank = i / n;
offset = i % n;
MPI_Win_lock(MPI_LOCK_SHARED, rank, 0, win);
if (rank == myrank) {
    value = locala[offset];
} else {
    MPI_Get(&value, 1, MPI_DOUBLE, 
            rank, offset, 1, MPI_DOUBLE, win);
}
MPI_Win_unlock(rank, win);
```
Memory for Passive Target RMA

- Passive target operations are *harder* to implement
  - Hardware support helps

- MPI *allows* (but does not require) an implementation to require that windows objects used for passive target RMA use local windows allocated with MPI_Aloc_mem
Allocating Memory

- **MPI_Alloc_mem, MPI_Free_mem**
- Special Issue: Checking for no memory available:
  - e.g., the Alloc_mem equivalent of a null return from malloc
  - Default error behavior of MPI is to abort
- Solution:
  - Change the error handler on MPI_COMM_WORLD to MPI_ERRORS_RETURN, using MPI_COMM_SET_ERRHANDLER (in MPI-1, MPI_ERRHANDLER_SET)
  - Check error class with MPI_ERROR_CLASS
    - Error codes are not error classes
Mutex with Passive Target RMA

- `MPI_Win_lock/unlock` DO NOT define a critical section
- One has to implement a distributed locking algorithm using passive target RMA operations in order to achieve the equivalent of a mutex
- Example follows
Implementing Mutex

- Create "waitwin" window object
  - One process has N-byte array (byte per process)
- One access epoch to try to lock
  - Put "1" into corresponding byte
  - Get copy of all other values
- If all other values are zero, obtained lock
- Otherwise must wait
Attempting to lock

- Processes use one access epoch to attempt to obtain the lock
- Process 1 succeeds, but process 3 must wait

- No other 1s, so lock was obtained
- 1 in rank 1 position, so process must wait
Waiting for the lock

- Naïve approach: simply MPI_Get the other bytes over and over
  - Lots of extra remote memory access
  - Better approach is to somehow notify waiting processes
  - Using RMA, set up a second window object with a byte on each process, spin-wait on local memory
    - This approach is like MCS locks
    - Lots of wasted CPU cycles spinning

- Better approach: Using MPI-1 point-to-point, send a zero-byte message to the waiting process to notify it that it has the lock
  - Let MPI implementation handle checking for message arrival
Releasing the Lock

- Process 1 uses one access epoch to release the lock
- Because process 3 is waiting, process 1 must send a message to notify process 3 that it now owns the lock

Diagram:

- Process 0
  - waitwin[4]
  - 0, 0, 0, 1
- Process 1
  - Lock
  - Put(0 at byte 1)
  - Get(other 3 bytes)
  - Unlock
- Process 3
  - MPI_Recv(ANY_SRC)
  - MPI_Send(rank 3)

1 in rank 3 position, must notify of release

MPI_Recv completes, Process 3 has lock
Mutex Code Walkthrough

- Code allows any process to be the “home” of the array:

  Process 0           Process “homerank”           Process nprocs - 1

  waitlist[N]          ...                          ...  waitlistwin
  object

- mpimutex_t type, for reference:

  typedef struct mpimutex {
    int nprocs, myrank, homerank;
    MPI_Comm comm;
    MPI_Win waitlistwin;
    MPI_Datatype waitlisttype;
    unsigned char *waitlist;
  } *mpimutex_t;
Comments on Local Access

- **Volatile:**
  - Tells compiler that some other agent (such as another thread or process) may change the value
  - In practice, rarely necessary for arrays but *usually necessary for scalars*
  - Volatile is *not* just for MPI-2. Any shared-memory program needs to worry about this (even for cache-coherent shared-memory systems)

- Fortran users don't have volatile (yet):
  - But they can use the following evil trick ...
Tuning RMA
Performance Tuning RMA

- MPI provides *generality* and *correctness*
- Special cases may allow performance optimizations
  - MPI provides two ways to identify special cases:
    - Assertion flags for `MPI_Win_fence`, etc.
    - Info values for `MPI_Win_create` and `MPI_Alloc_mem`
Tuning Fence

- **Asserts for fence**
  - Note that these rely on understanding the “global/collective” use of the RMA calls in the code.
MPI_Win_fence Assert Values

- **MPI_MODE_NOSTORE**
  - No update to the local window was made by the local process (using assignments, e.g., stores) since the last call to MPI_Win_fence

- **MPI_MODE_NOPUT**
  - There will be no RMA (Put or Accumulate) to the local window before the next MPI_Win_fence

- **MPI_MODE_NOPRECEDE**
  - This MPI_Win_fence will not complete any RMA calls made by this process (no preceding RMA calls)

- **MPI_MODE_NOSUCCEED**
  - No RMA calls will be made on this window before the next MPI_Win_fence call (no succeeding (as in coming after) RMA calls)
MPI_Win_fence(MPI_MODE_NOPRECEDE, win);

MPI_Put(&matrix[myrows][0], cols+2, MPI_INT,
       exch_next, 0, cols+2, MPI_INT, win);

MPI_Put(&matrix[1][0], cols+2, MPI_INT, exch_prev,
       (nrows_prev+1)*(cols+2), cols+2, MPI_INT, win);

MPI_Win_fence(MPI_MODE_NOSTORE | MPI_MODE_NOPUT | MPI_MODE_NOSUCCEED, win);
Tuning P/S/C/W

• Asserts for MPI_Win_start and MPI_Win_post
  • Start
    — MPI_MODE_NOCHECK
      - Guarantees that the matching calls to MPI_Win_post have already been made
  • Post
    — MPI_MODE_NOSTORE, MPI_MODE_NOPUT
      - Same meaning as for MPI_Win_fence
    — MPI_MODE_NOCHECK
      - Nocheck means that the matching calls to MPI_Win_start have not yet occurred
MPI_Win_create

• If only active-target RMA will be used, pass an info object to MPI_Win_create with key “no_locks” set to “true”

  MPI_Info info;
  MPI_Info_create( &info );
  MPI_Info_set( info, "no_locks", "true" );
  MPI_Win_create( ..., info, ... );
  MPI_Info_free( &info );
Understanding the MPI-2 Completion Model

• Very relaxed
  — To give the implementer the greatest flexibility
  — Describing this relaxed model precisely is difficult
    - Implementer only needs to obey the rules
  — But it doesn’t matter; simple rules work for most programmers

• When does the data actually move?
Data Moves Early

Process 0

MPI_Win_lock
(win_lock returns)

MPI_Put

MPI_Put

MPI_Get

MPI_Win_unlock
(unlock returns)

Process 1

(lock granted)

(lock released)

(window updated)

(window updated)

(window accessed)
Data Moves Late

Process 0

`MPI_Win_lock` (save information)

`MPI_Put` (save information)

`MPI_Put` (save information)

`MPI_Get` (save information)

`MPI_Win_unlock` (unlock returns)

Process 1

(acquire lock, process requests, release lock)
Performance Tests

- “Halo” exchange or ghost-cell exchange operation
  - Each process exchanges data with its nearest neighbors
  - Part of mpptest benchmark
  - One-sided version uses all 3 synchronization methods

- Ran on
  - Sun Fire SMP at Univ. of Aachen, Germany
  - IBM p655+ SMP at San Diego Supercomputer Center
One-Sided Communication on Sun SMP with Sun MPI

Halo Performance on Sun

Bytes

uSec

sendrecv - 8
psendrecv - 8
putall - 8
putpscwalloc - 8
putlockshared - 8
putlocksharednb - 8
One-Sided Communication on IBM SMP with IBM MPI

![Halo Performance (IBM-7)](image-url)
MPI and Threads
MPI and Threads

- **MPI** describes parallelism between *processes*

- *Thread* parallelism provides a shared-memory model within a process

- **OpenMP** and pthreads are common
  - *OpenMP* provides convenient features for loop-level parallelism
MPI and Threads (contd.)

- MPI-2 defines four levels of thread safety
  - MPI_THREAD_SINGLE: only one thread
  - MPI_THREAD_FUNNELED: only one thread that makes MPI calls
  - MPI_THREAD_SERIALIZED: only one thread at a time makes MPI calls
  - MPI_THREAD_MULTIPLE: any thread can make MPI calls at any time

- User calls MPI_Init_thread to indicate the level of thread support required; implementation returns the level supported
Threads and MPI in MPI-2

• An implementation is not required to support levels higher than MPI_THREAD_SINGLE; that is, an implementation is not required to be thread safe

• A fully thread-compliant implementation will support MPI_THREAD_MULTIPLE

• A portable program that does not call MPI_Init_thread should assume that only MPI_THREAD_SINGLE is supported
For MPI_THREAD_MULTIPLE

• When multiple threads make MPI calls concurrently, the outcome will be as if the calls executed sequentially in some (any) order

• Blocking MPI calls will block only the calling thread and will not prevent other threads from running or executing MPI functions

• It is the user's responsibility to prevent races when threads in the same application post conflicting MPI calls

• User must ensure that collective operations on the same communicator, window, or file handle are correctly ordered among threads
Efficient Support for MPI_THREAD_MULTIPLE

• MPI-2 allows users to write multithreaded programs and call MPI functions from multiple threads (MPI_THREAD_MULTIPLE)

• Thread safety does not come for free, however

• Implementation must protect certain data structures or parts of code with mutexes or critical sections

• To measure the performance impact, we ran tests to measure communication performance when using multiple threads versus multiple processes
  — Details in Euro PVM/MPI 2007 paper
Tests with Multiple Threads versus Processes

```
\[ \text{Tests with Multiple Threads versus Processes} \]
```
Concurrent Bandwidth Test on Linux Cluster

MPICH2 version 1.0.5
Open MPI version 1.2.1
Concurrent Bandwidth Test on Sun and IBM SMPs
Concurrent Latency Test on Linux Cluster
Concurrent Latency Test on Sun and IBM SMPs
Common User Errors
Top MPI Errors

- MPI_Bcast not called collectively (e.g., sender bcasts, receivers use MPI_Recv)
- Failure to wait (or test for completion) on MPI_Request
- Reusing buffers on nonblocking operations
- Using a single process for all file I/O
- Using MPI_Pack/Unpack instead of Datatypes
- Unsafe use of blocking sends/receives
- Using MPI_COMM_WORLD instead of comm in libraries
- Not understanding implementation performance settings
- Failing to install and use the MPI implementation according to its documentation.
Conclusions
Designing Parallel Programs

• Common theme – think about the “global” object, then see how MPI can help you

• Also specify the largest amount of communication or I/O between “synchronization points”
  — Collective and noncontiguous I/O
  — RMA
Summary

- MPI-2 provides major extensions to the original message-passing model targeted by MPI-1
- MPI-2 can deliver to libraries and applications portability across a diverse set of environments
- Implementations are here now
- Sources:
  - The MPI standard documents are available at http://www.mpi-forum.org
  - Using MPI (Gropp, Lusk, and Skjellum) and Using MPI-2 (Gropp, Lusk, and Thakur), MIT Press
    - Using MPI also available in German from Oldenbourg
    - Using MPI-2 also available in Japanese, from Pearson Education Japan
Conclusions

- MPI is a proven, effective, portable parallel programming model
- MPI has succeeded because
  - features are orthogonal (complexity is the product of the number of features, not routines)
  - programmer can control memory motion (critical in high-performance computing)
  - complex programs are no harder than easy ones
  - open process for defining MPI led to a solid design
More Information on Software

- **MPICH2**

- **More Information on PnetCDF**
  - Parallel netCDF web site:
  - Parallel netCDF mailing list:
    - Mail to majordomo@mcs.anl.gov with the body “subscribe parallel-netcdf”
  - The SDM SciDAC web site:
    - http://sdm.lbl.gov/sdmcenter/

- **PETSc**
  - http://www.mcs.anl.gov/petsc

- **HDF5**
  - http://hdf.ncsa.uiuc.edu/HDF5/
MPICH2

- **Goals**: same as MPICH
  - Research project, to explore scalability and performance, incorporate and test research results
  - Software project, to encourage use of MPI-2

- **Scope**: all of MPI-2
  - I/O
  - Dynamic
  - One-sided
  - All the obscure parts, too
  - Useful optional features recommended by the Standard (full mpiexec, singleton-init, thread safety)
  - Other useful features (debugging, profiling libraries, tools)
**MPICH2**

- Incorporates latest research into MPI implementation
  - Collective operations
  - Optimizations for one-sided ops
  - Optimized datatype handling
  - I/O

- See recent Euro PVM/MPI and Cluster Proceedings

- In use by vendors
  - IBM on BG/L and BG/P
  - Cray on Red Storm, XT3, XT4
  - Intel, Microsoft, Myricom, ...
  - Having vendors adapt MPICH2 into their products has helped make it efficient and robust
The MPI Standard (1 & 2)
Tutorial Material on MPI, MPI-2

http://www.mcs.anl.gov/mpi/{usingmpi,usingmpi2}