Argonne NATIONAL LABORATORY

Programming in MPI for Performance and MPI at Exascale

Rajeev Thakur

Mathematics and Computer Science Division

Argonne National Laboratory

U.S. DEPARTMENT OF ENERGY

Outline

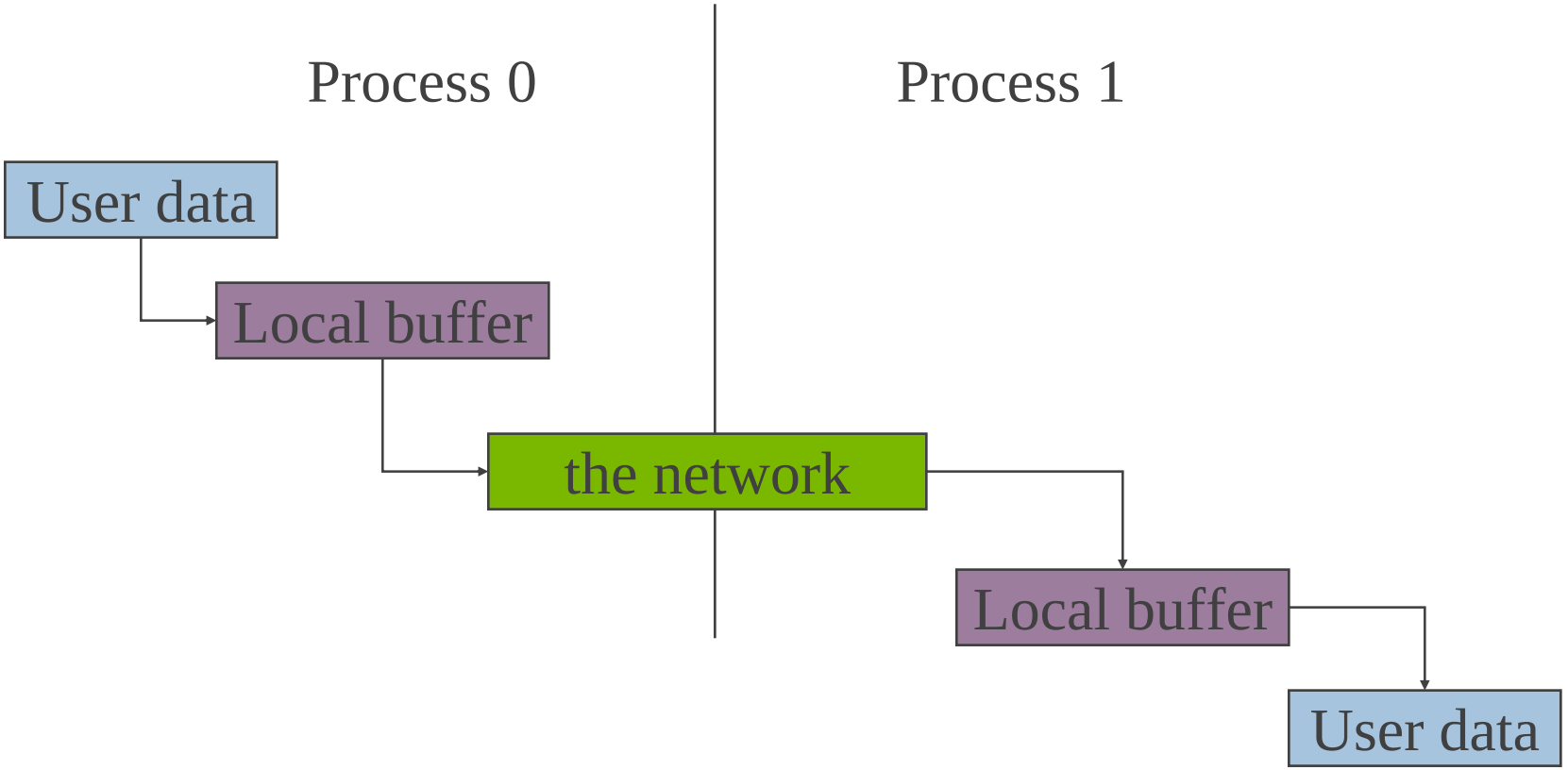
- Selected topics in MPI programming
- MPI profiling interface and tools
  - SLOG/Jumpshot: visualizing parallel performance
  - FPMPI: gathering summary statistics
  - Collchk: runtime checking of correct use of collective operations
- MPI and threads: hybrid programming
- One-sided communication
- Dynamic processes
- MPI at Exascale
- Recent Activities of the MPI Forum

Message Passing, Buffering, Deadlocks

- Message passing is a simple programming model, but there are some special issues
  - Buffering and deadlock
  - Deterministic execution
  - Performance

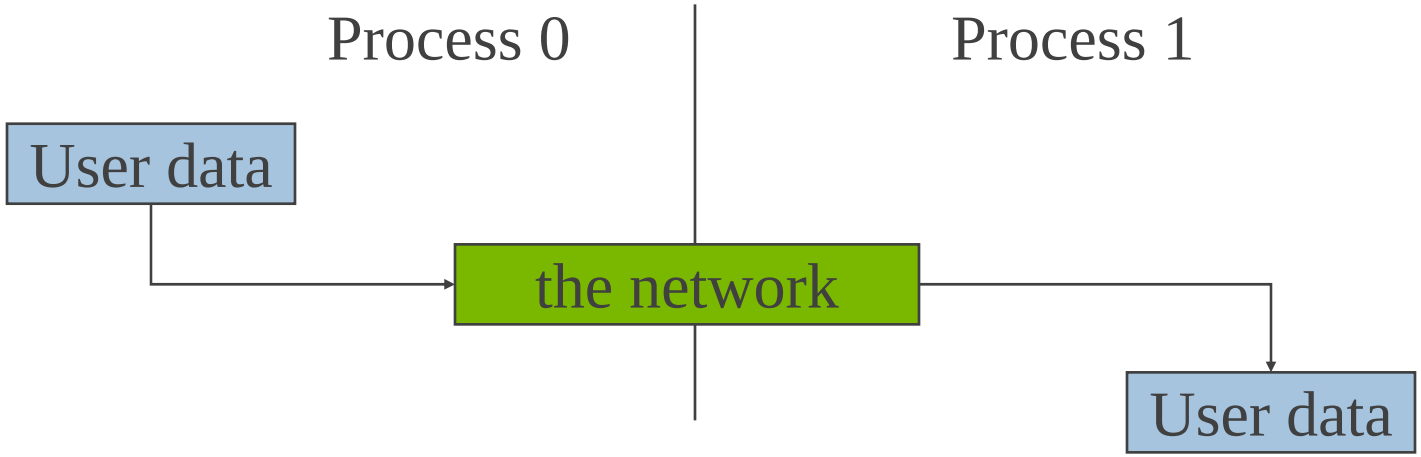
Buffers

- When you send data, where does it go? One possibility is:



Avoiding Buffering

- It is better to avoid copies:



This requires that **`MPI_Send`** wait on delivery, or that **`MPI_Send`** return before transfer is complete, and we wait later.

Sources of Deadlocks

- Send a large message from process 0 to process 1
  - If there is insufficient storage at the destination, the send must wait for the user to provide the memory space (through a receive)
- What happens with this code?

Process 0	Process 1
Send(1)	**Send(0)**
Recv(1)	**Recv(0)**

- This is called “unsafe” because it depends on the availability of system buffers in which to store the data sent until it can be received

Some Solutions to the "unsafe" Problem

- Order the operations more carefully:

Process 0	Process 1
`Send(1)`	`Recv(0)`
`Recv(1)`	`Send(0)`

- Supply receive buffer at same time as send:

Process 0	Process 1
`Sendrecv(1)`	`Sendrecv(0)`

More Solutions to the “unsafe” Problem

- Supply own space as buffer for send

Process 0	Process 1
`Bsend(1)`	`Bsend(0)`
`Recv(1)`	`Recv(0)`

- Use non-blocking operations:

Process 0	Process 1
`Isend(1)`	`Isend(0)`
`Irecv(1)`	`Irecv(0)`
`Waitall`	`Waitall`

Communication Modes

- MPI provides multiple *modes* for sending messages:
  - Synchronous mode (**`MPI_Ssend`**): the send does not complete until a matching receive has begun. (Unsafe programs deadlock.)
  - Buffered mode (**`MPI_Bsend`**): the user supplies a buffer to the system for its use. (User allocates enough memory to make an unsafe program safe.
  - Ready mode (**`MPI_Rsend`**): user guarantees that a matching receive has been posted.
    - Allows access to fast protocols
    - undefined behavior if matching receive not posted
- Non-blocking versions (**`MPI_Issend`**, etc.)
- **`MPI_Recv`** receives messages sent in any mode.

Buffered Mode

- When MPI_Isend is awkward to use (e.g. lots of small messages), the user can provide a buffer for the system to store messages that cannot immediately be sent.

```
int bufsize;
char *buf = malloc( bufsize );
MPI_Buffer_attach( buf, bufsize );
...
MPI_Bsend( ... same as MPI_Send ... )
...
MPI_Buffer_detach( &buf, &bufsize );
```

- MPI_Buffer_detach waits for completion.
- Performance depends on MPI implementation and size of message.

MPI_Sendrecv

- Allows simultaneous send and receive
- Everything else is general.
  - Send and receive datatypes (even type signatures) may be different
  - Can use Sendrecv with plain Send or Recv (or Irecv or Ssend_init, …)
  - More general than “send left”

Process 0	Process 1
`SendRecv(1)`	**`SendRecv(0)`**

Understanding Performance: Unexpected Hot Spots

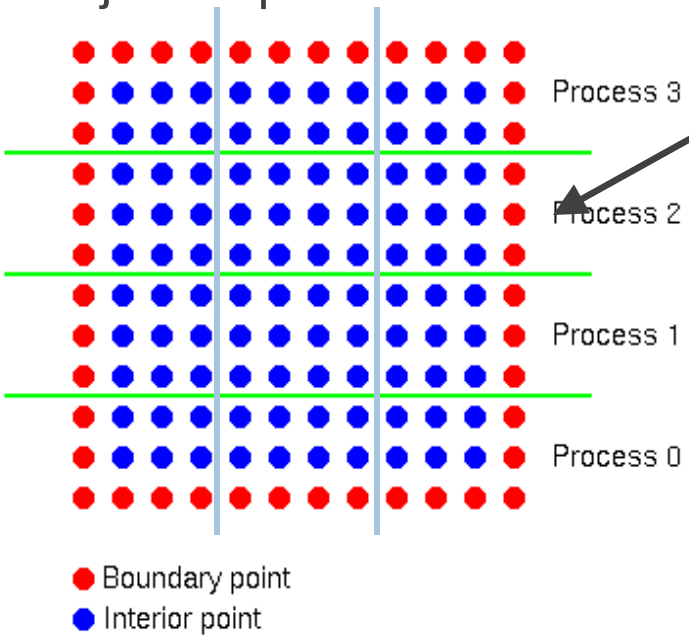
- Basic performance analysis looks at two-party exchanges
- Real applications involve many simultaneous communications
- Performance problems can arise even in common grid exchange patterns
- Message passing illustrates problems present even in shared memory
  - Blocking operations may cause unavoidable memory stalls

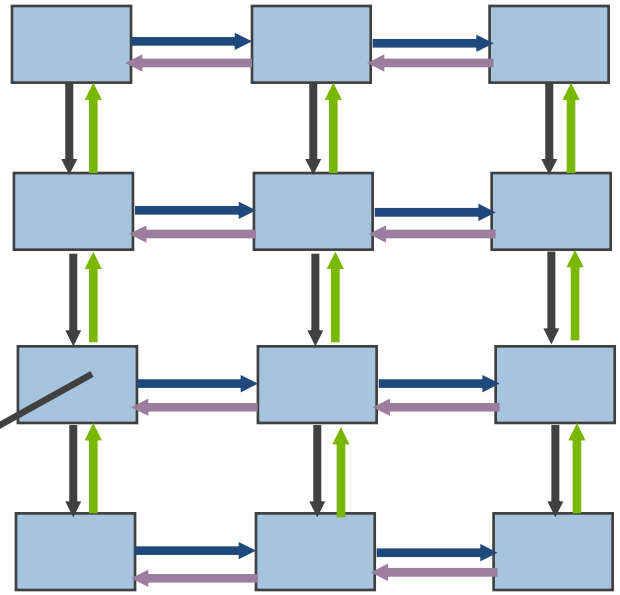
Basic MPI: Looking Closely at a Simple Communication Pattern

- Many programs rely on “halo exchange” (ghost cells, ghost points, stencils) as the core communication pattern
  - Many variations, depending on dimensions, stencil shape
  - Here we look carefully at a simple 2-D case
- Unexpected performance behavior
  - Even simple operations can give surprising performance behavior.
  - Examples arise even in common grid exchange patterns
  - Message passing illustrates problems present even in shared memory
    - Blocking operations may cause unavoidable stalls

Processor Parallelism

- Decomposition of a mesh into 1 patch per process
  - Update formula typically a(I,j) = f(a(i-1,j),a(i+1,j),a(I,j+1),a(I,j-1),…)
  - Requires access to “neighbors” in adjacent patches





Sample Code

- Do i=1,n_neighbors
  Call MPI_Send(edge, len, MPI_REAL, nbr(i), tag,
  comm, ierr)
  Enddo
  Do i=1,n_neighbors
  Call MPI_Recv(edge,len,MPI_REAL,nbr(i),tag,
  comm,status,ierr)
  Enddo
- What is wrong with this code?

Deadlocks!

- All of the sends may block, waiting for a matching receive (will for large enough messages)
- The variation of

  ```
  if (has down nbr)
      Call MPI_Send( … down … )
  if (has up nbr)
      Call MPI_Recv( … up … )
  …
  ```

  sequentializes (all except the bottom process blocks)

Sequentialization

Start	Start	Start	Start	Start	Start	Send	Recv
Send	Send	Send	Send	Send	Send		
					Send	Recv	
				Send	Recv		
			Send	Recv			
		Send	Recv				
	Send	Recv					
Send	Recv						

Fix 1: Use Irecv

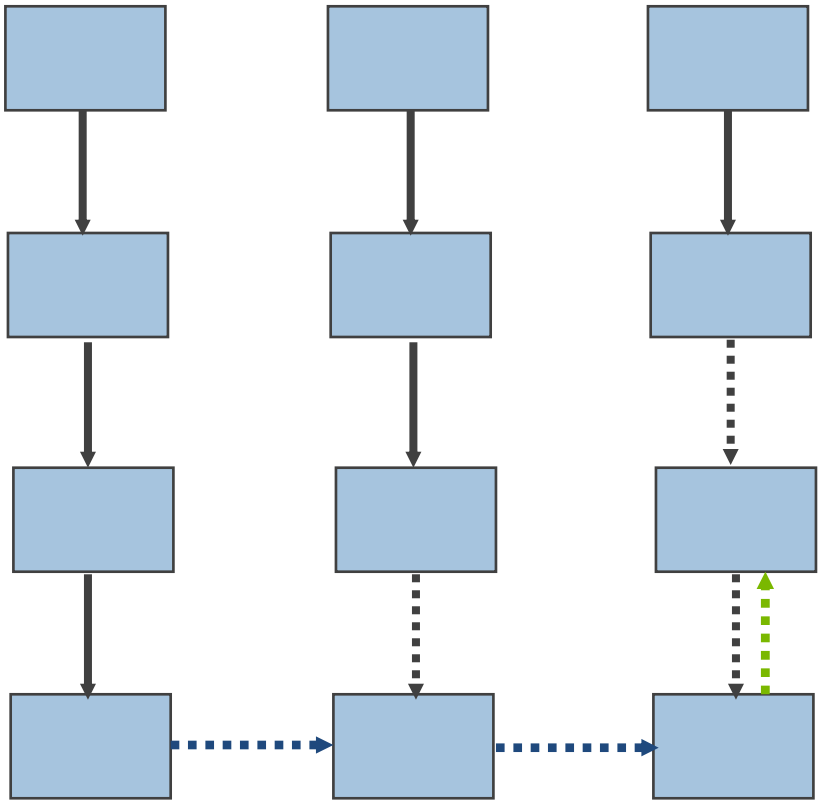
- Do i=1,n_neighbors
  Call MPI_Irecv(edge,len,MPI_REAL,nbr(i),tag,
  comm,requests(i),ierr)
  Enddo
  Do i=1,n_neighbors
  Call MPI_Send(edge, len, MPI_REAL, nbr(i), tag,
  comm, ierr)
  Enddo
  Call MPI_Waitall(n_neighbors, requests, statuses, ierr)
- Does not perform well in practice. Why?

Timing Model

- Sends interleave
- Sends block (data larger than buffering will allow)
- Sends control timing
- Receives do not interfere with Sends
- Exchange can be done in 4 steps (down, right, up, left)

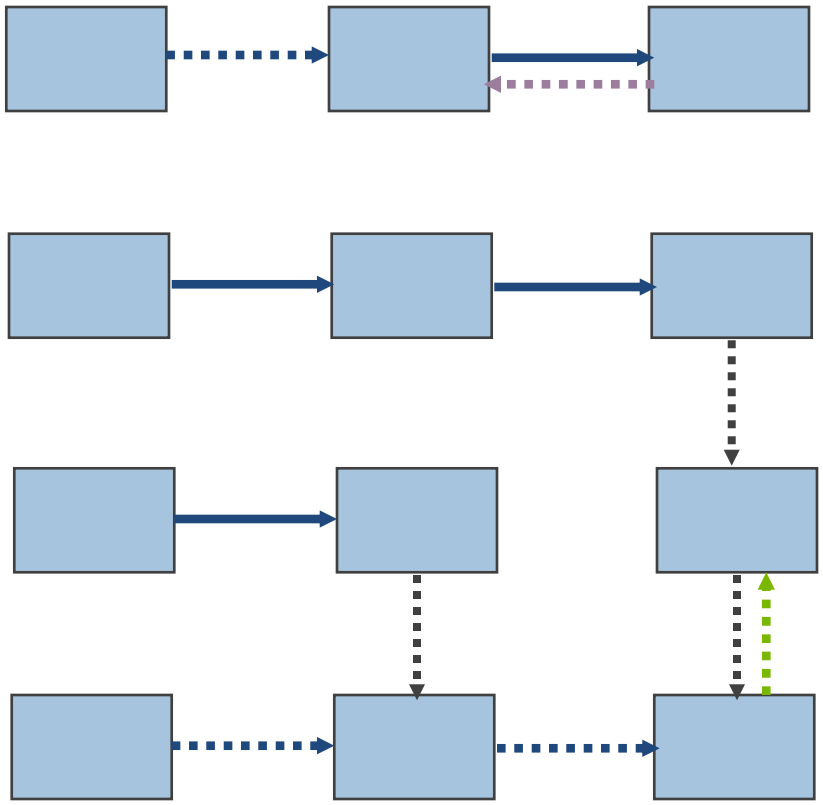
Mesh Exchange - Step 1

- Exchange data on a mesh



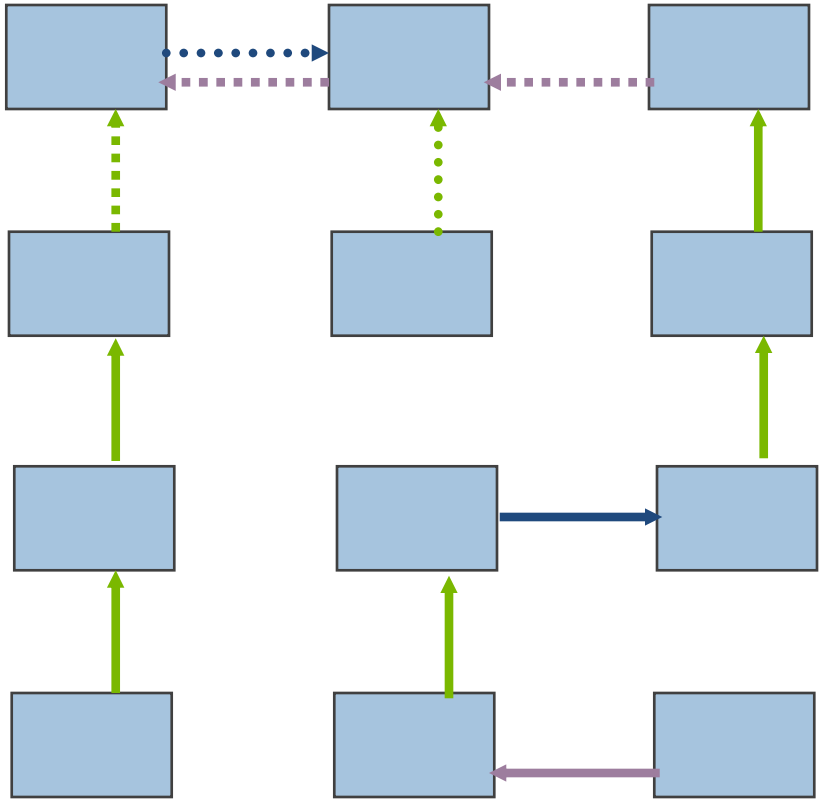
Mesh Exchange - Step 2

- Exchange data on a mesh



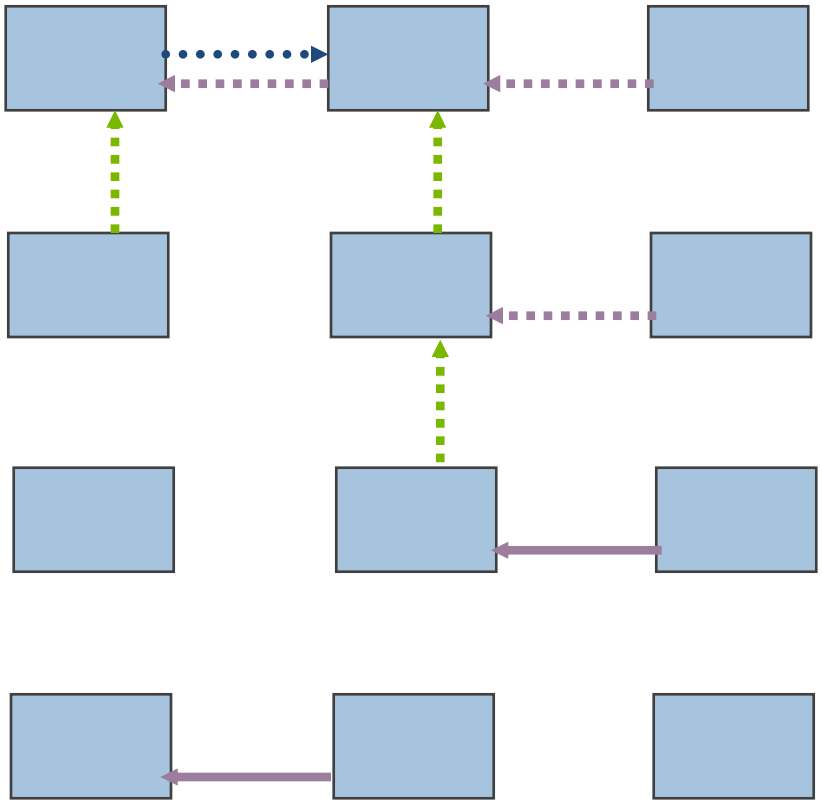
Mesh Exchange - Step 3

- Exchange data on a mesh



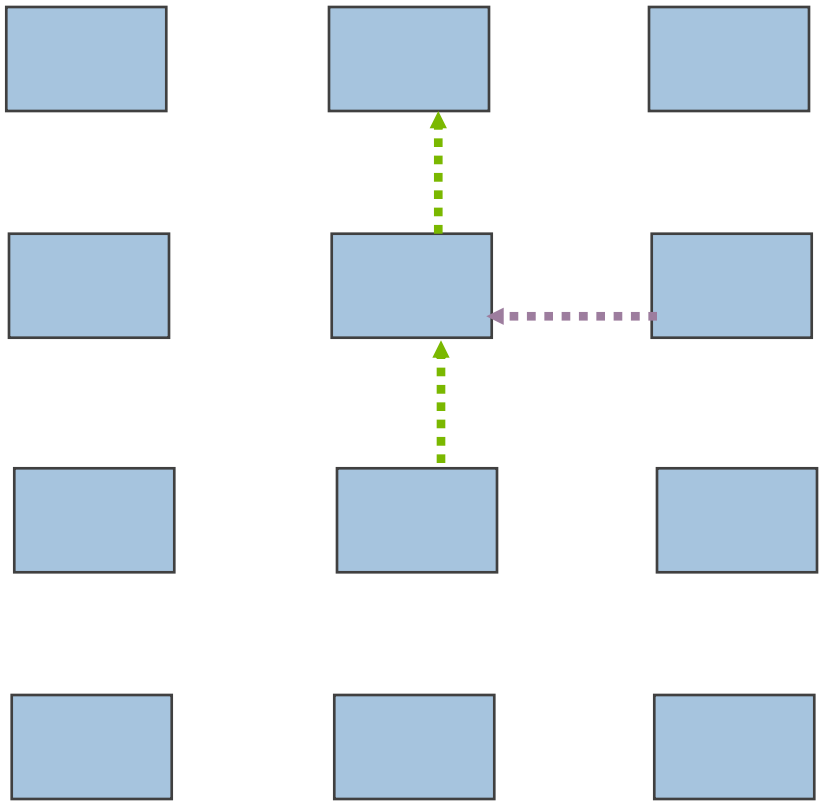
Mesh Exchange - Step 4

- Exchange data on a mesh



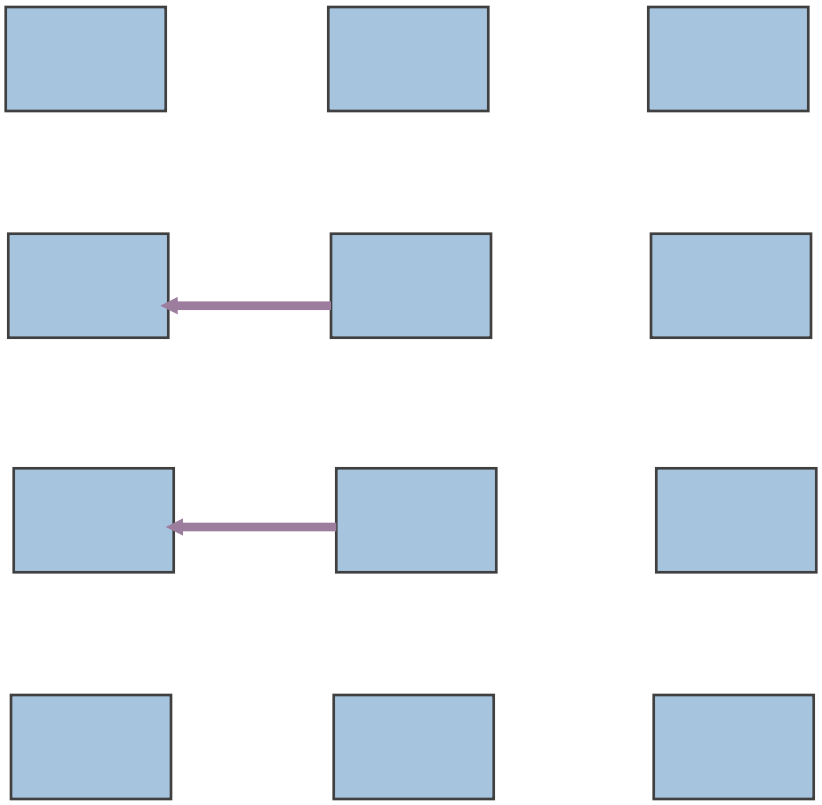
Mesh Exchange - Step 5

- Exchange data on a mesh

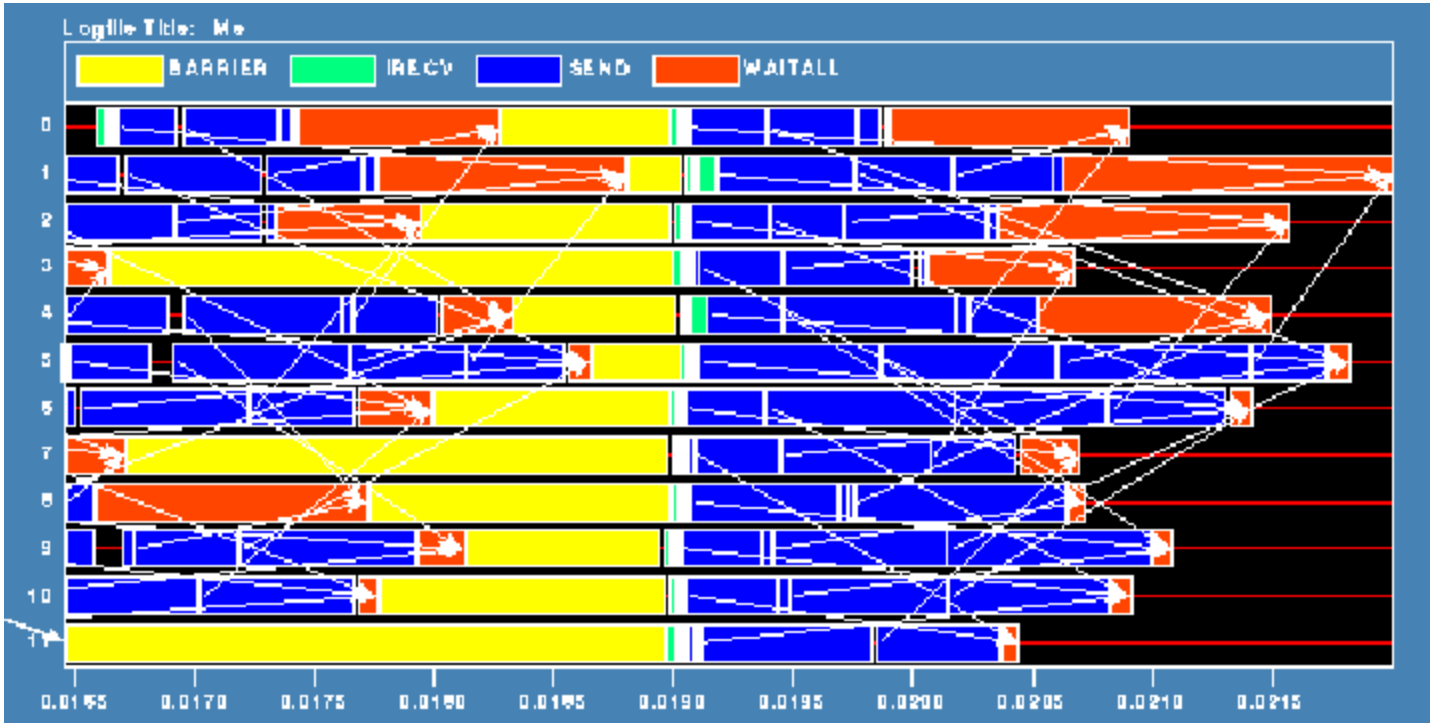


Mesh Exchange - Step 6

- Exchange data on a mesh

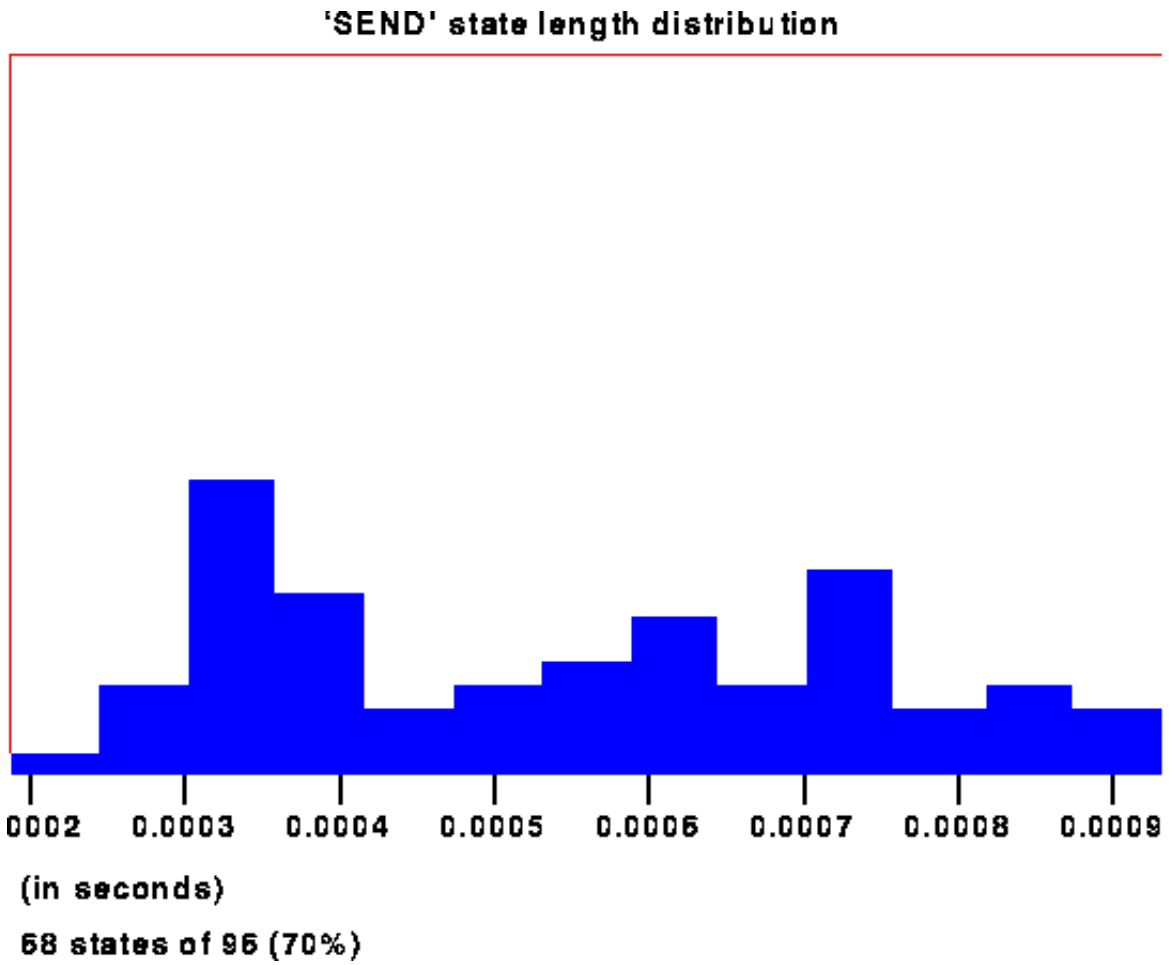


Timeline from IBM SP



- Note that process 1 finishes last, as predicted

Distribution of Sends



Why Six Steps?

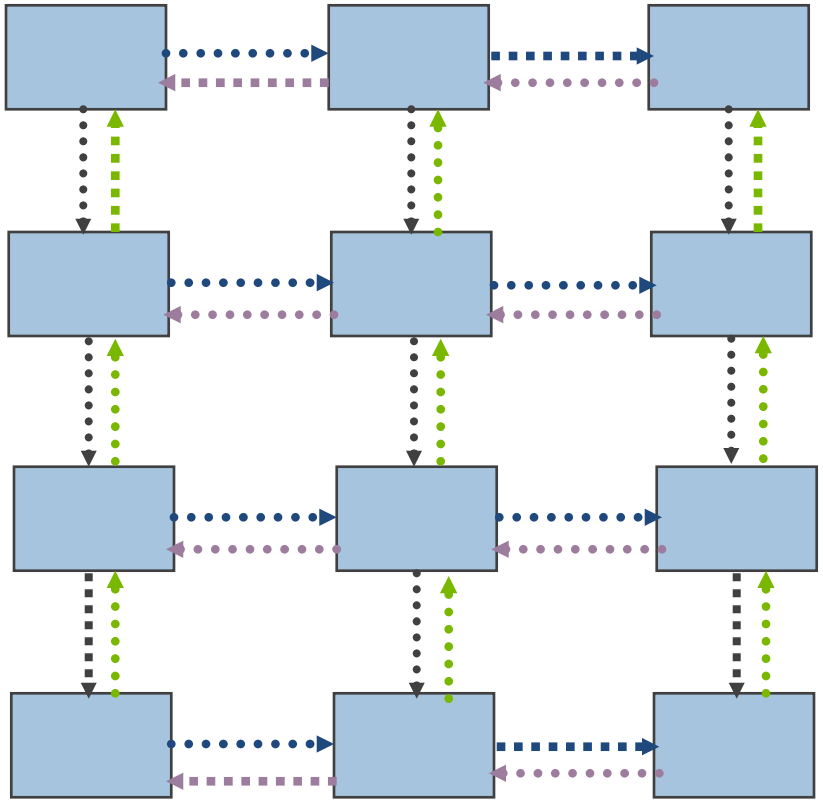
- Ordering of Sends introduces delays when there is contention at the receiver
- Takes roughly twice as long as it should
- Bandwidth is being wasted
- Same thing would happen if using memcpy and shared memory

Fix 2: Use Isend and Irecv

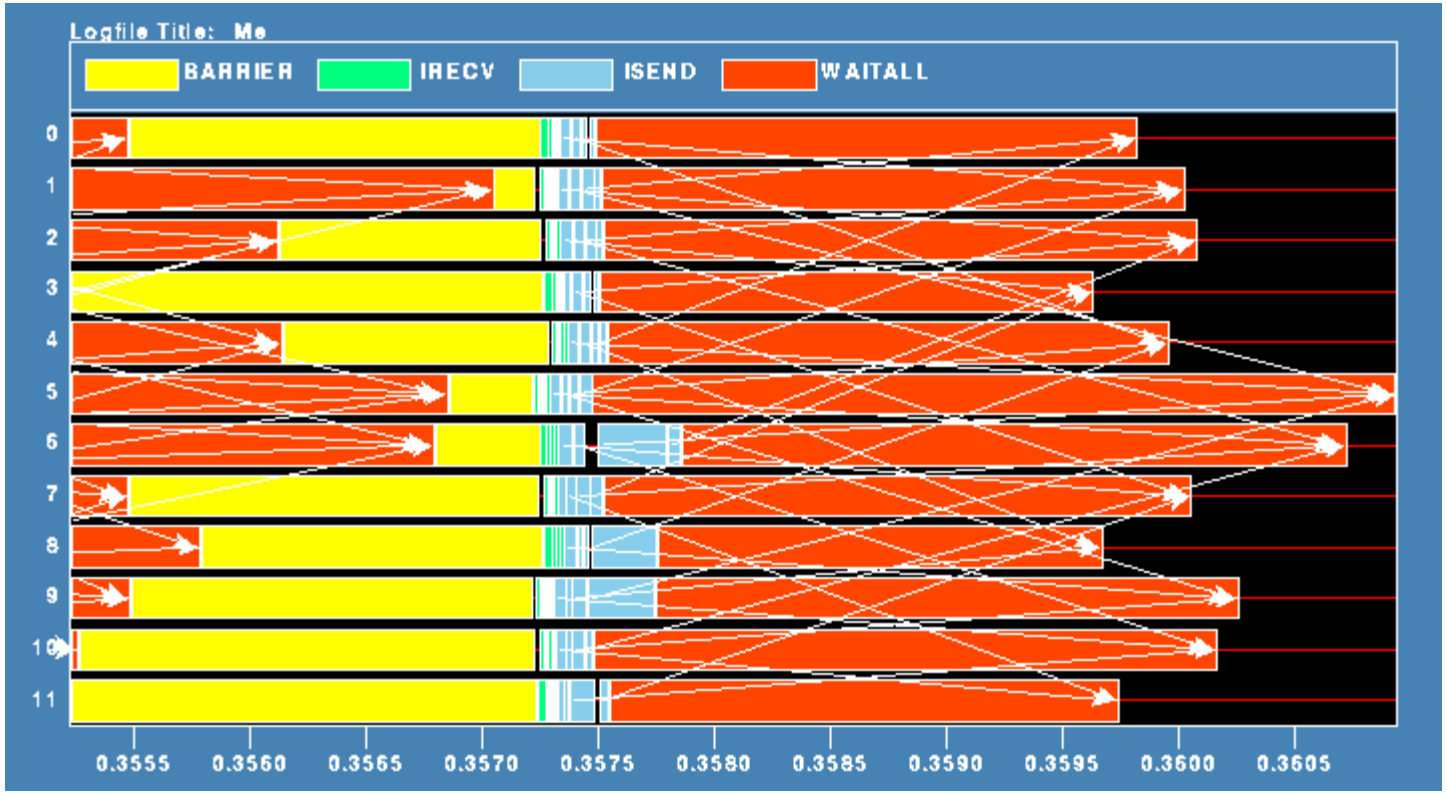
- Do i=1,n_neighbors
  Call MPI_Irecv(edge,len,MPI_REAL,nbr(i),tag,
  comm,request(i),ierr)
  Enddo
  Do i=1,n_neighbors
  Call MPI_Isend(edge, len, MPI_REAL, nbr(i), tag,
  comm, request(n_neighbors+i), ierr)
  Enddo
  Call MPI_Waitall(2*n_neighbors, request, statuses,
  ierr)

Mesh Exchange - Steps 1-4

- Four interleaved steps



Timeline from IBM SP



Note processes 5 and 6 are the only interior processors; these perform more communication than the other processors

Lesson: Defer Synchronization

- Send-receive accomplishes two things:
  - Data transfer
  - Synchronization
- In many cases, there is more synchronization than required
- Use nonblocking operations and MPI_Waitall to defer synchronization

MPI Message Ordering

- Multiple messages from one process to another will be ***matched*** in order, not necessarily *completed* in order

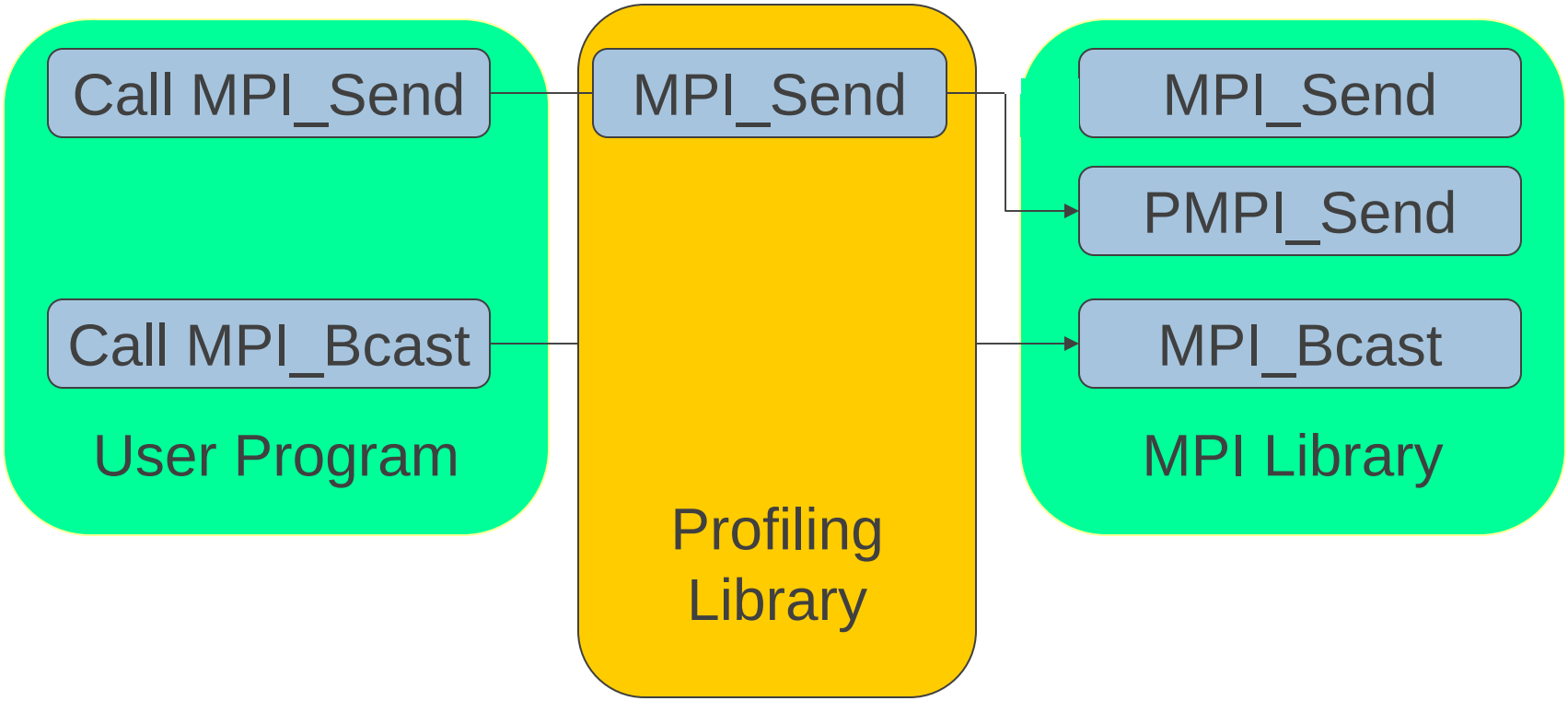
Rank 0	Rank 1	Rank 2
MPI_Isend(dest=1)	MPI_Irecv(any_src, any_tag)	MPI_Isend(dest=1)
MPI_Isend(dest=1)	MPI_Irecv(any_src, any_tag)	MPI_Isend(dest=1)
	MPI_Irecv(any_src, any_tag)	
	MPI_Irecv(any_src, any_tag)	

MPI Profiling Interface

Tools Enabled by the MPI Profiling Interface

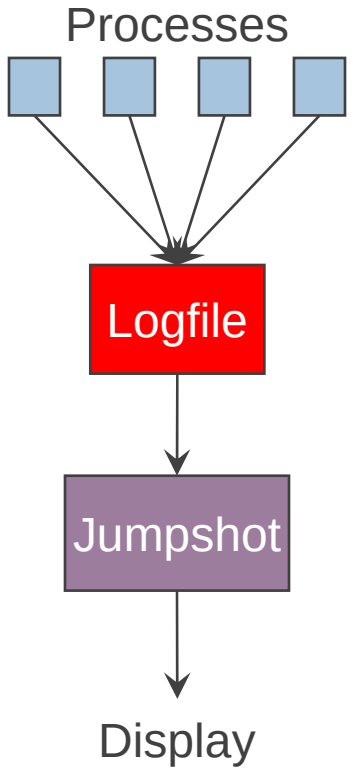
- The MPI profiling interface: how it works
- Some freely available tools
  - Those to be presented in other talks
  - A few that come with MPICH2
    - SLOG/Jumpshot: visualization of detailed timelines
    - FPMPI: summary statistics
    - Collcheck: runtime checking of consistency in use of collective operations

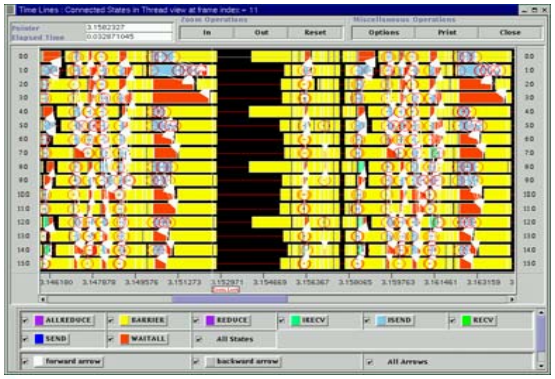
The MPI Profiling Interface

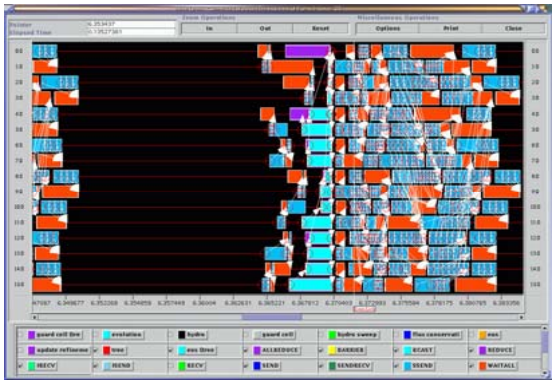


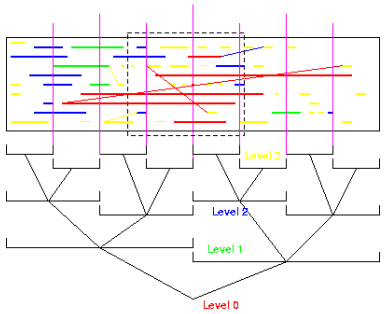
Performance Visualization with Jumpshot

- For detailed analysis of parallel program behavior, timestamped events are collected into a log file during the run.
- A separate display program (Jumpshot) aids the user in conducting a post mortem analysis of program behavior.
- We use an indexed file format (SLOG-2) that uses a preview to select a time of interest and quickly display an interval, without ever needing to read much of the whole file.

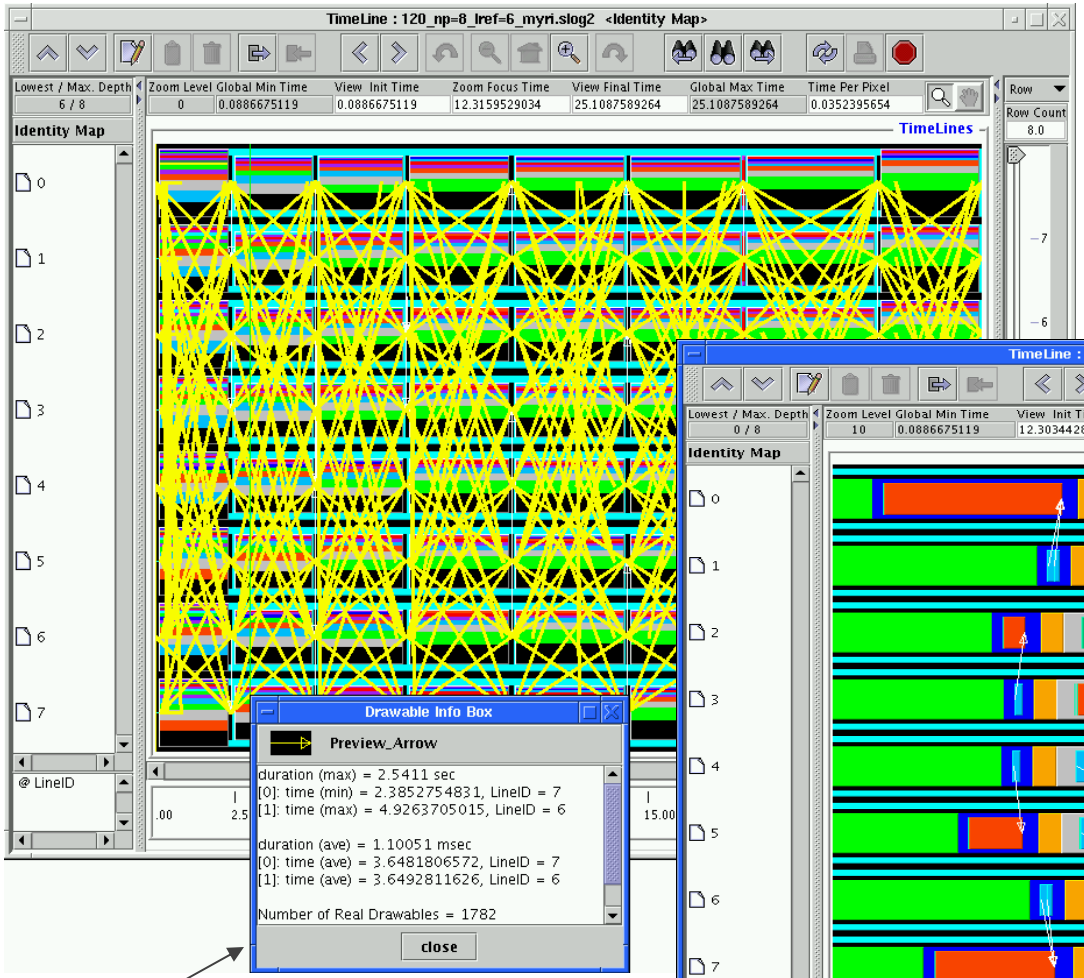






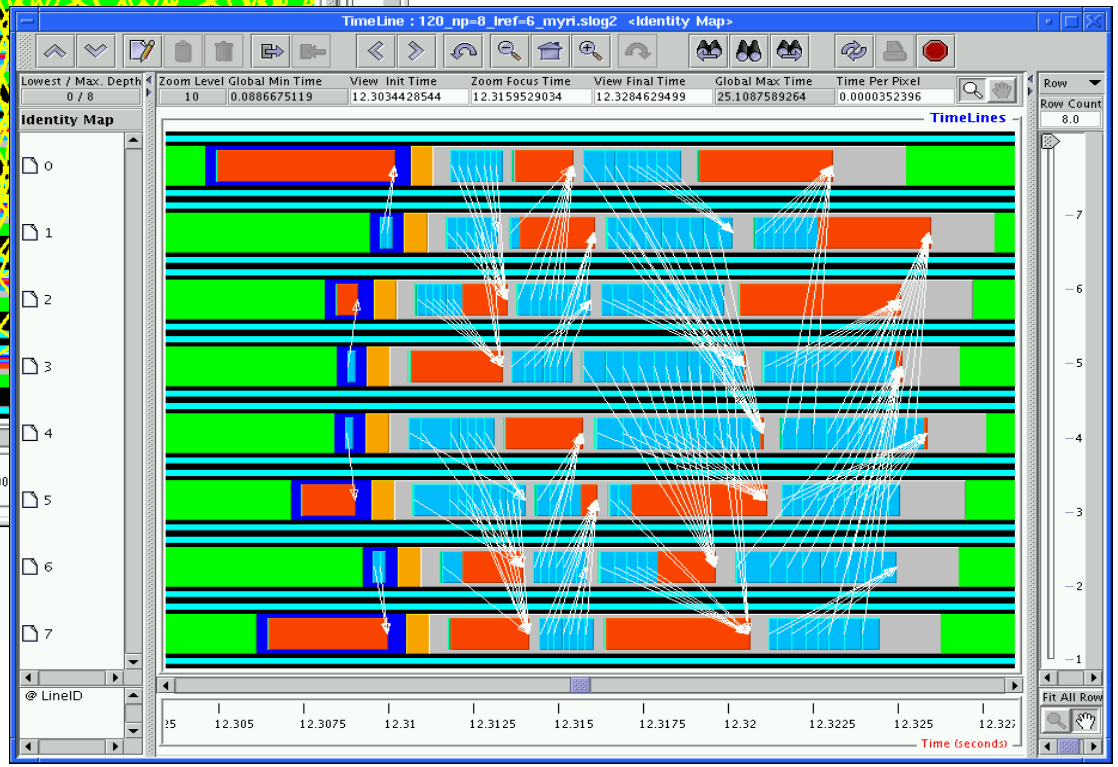


Viewing Multiple Scales



Each line represents 1000's of messages

Detailed view shows opportunities for optimization



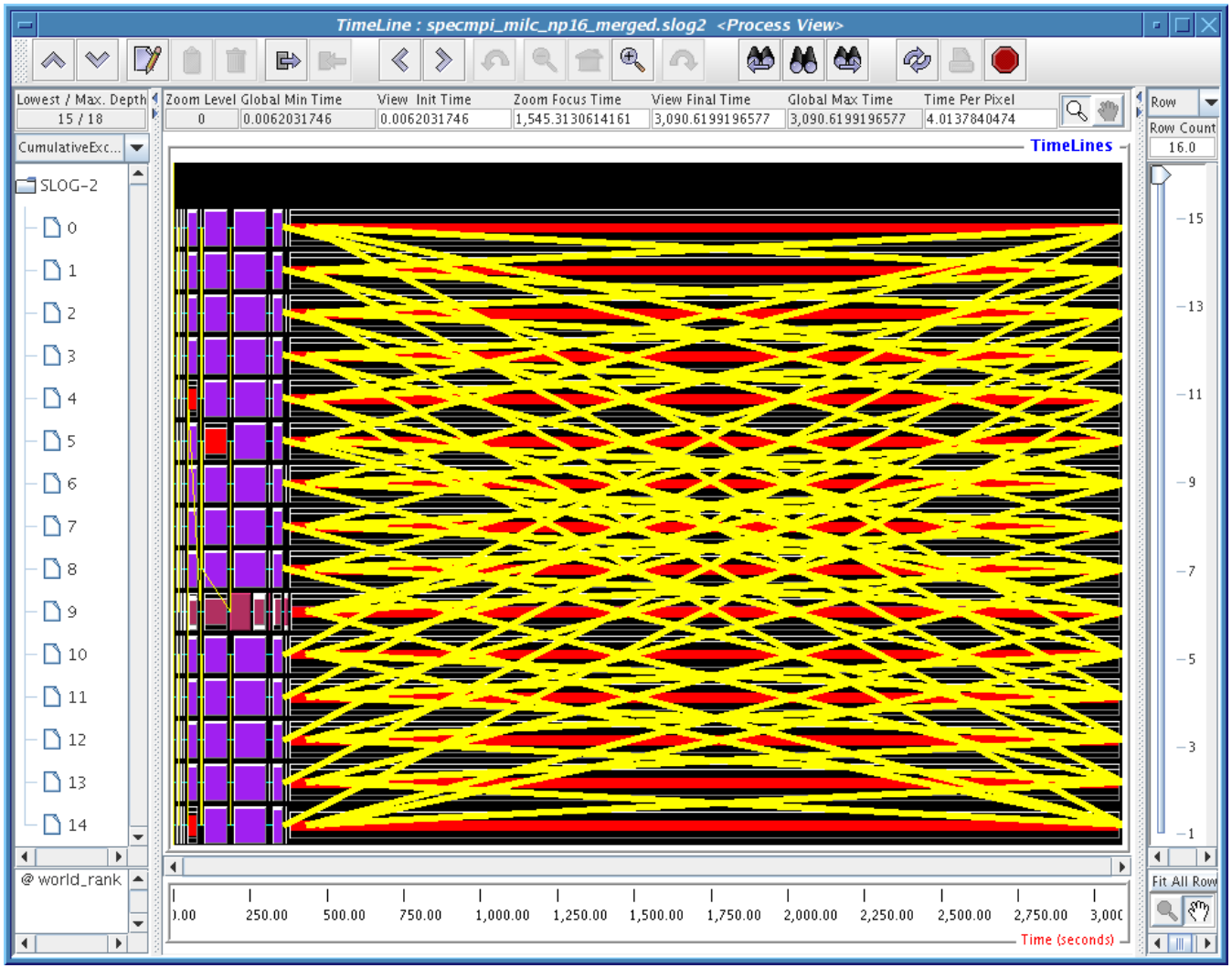
1000x zoom

Pros and Cons of this Approach

- Cons:
  - Scalability limits
    - Screen resolution
    - Big log files, although
      - Jumpshot can read SLOG files fast
      - SLOG can be instructed to log few types of events
  - Use for debugging only indirect
- Pros:
  - Portable, since based on MPI profiling interface
  - Works with threads
  - Aids understanding of program behavior
    - Almost always see something unexpected

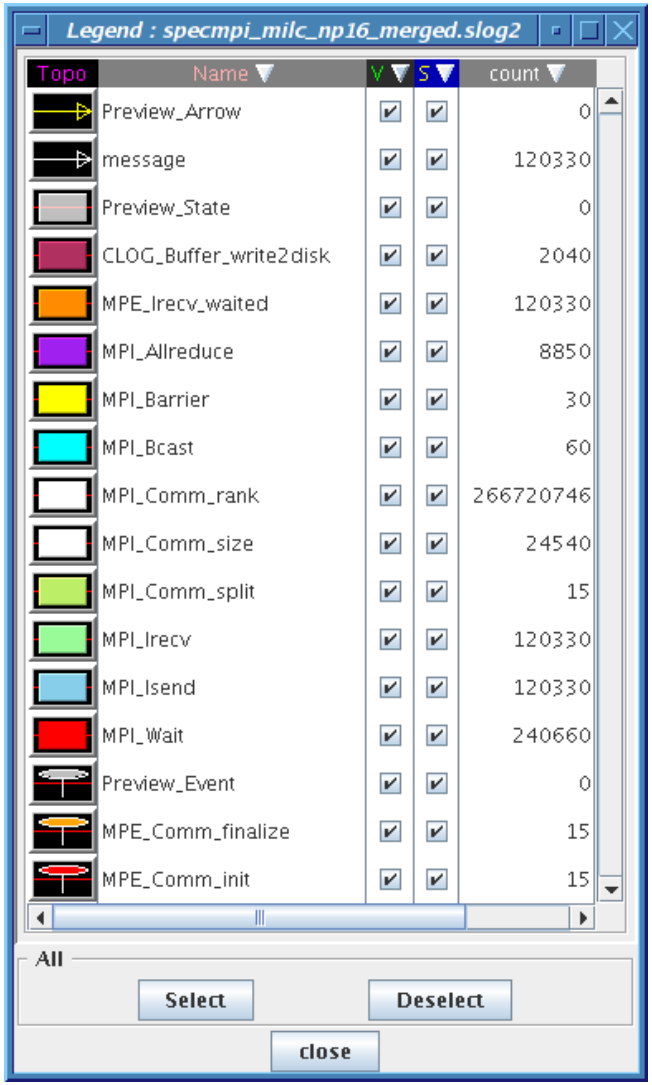
Looking at MILC in SPEC2007

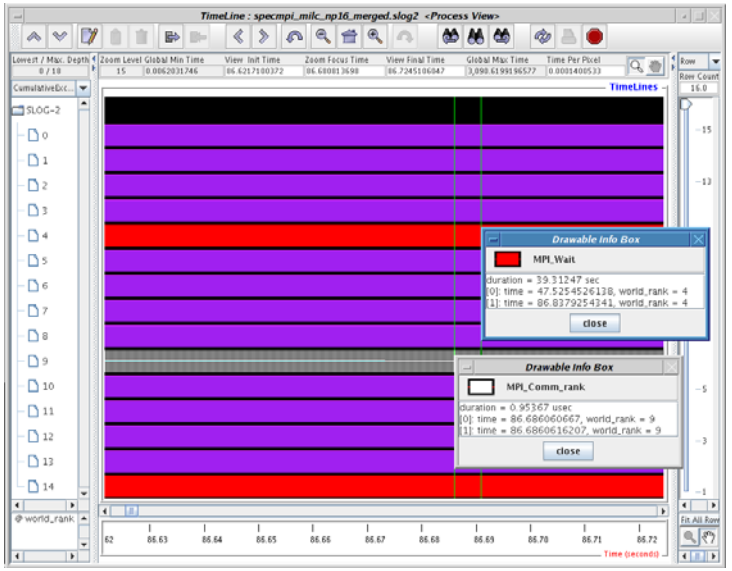
- Curious amount of All_reduce in initialization - why?

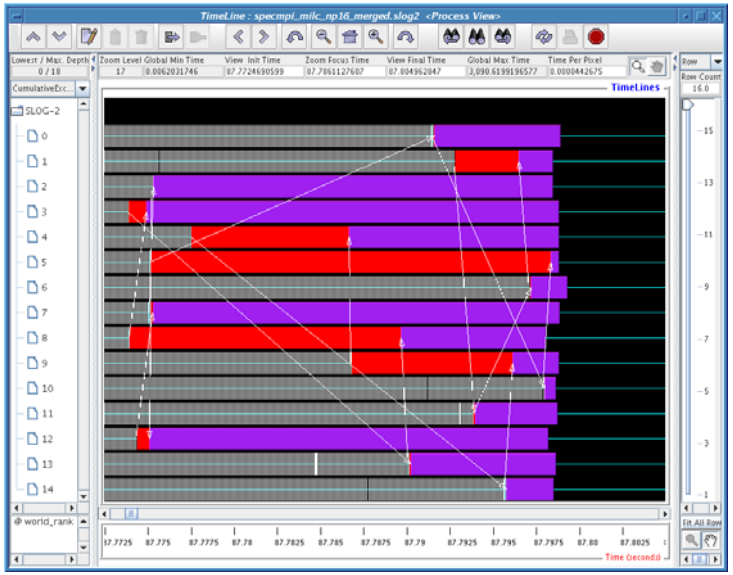


MILC

- The answer, and how







MILC

- The answer - why
  - Deep in innermost of quadruply nested loop, an innocent-looking line of code:

```
If ( i > myrank() ) …
```

And myrank is a function that calls MPI_Comm_rank

- It actually doesn't cost that much here, but
- It illustrates that you might not know what your code is doing what you think it is
  - Not a scalability issue (found on small # of processes)

Detecting Consistency Errors in MPI Collective Operations

- The Problem: the specification of MPI_Bcast:

  `MPI_Bcast( buf, count, datatype, root, comm )`

  requires that
  - **`root`** is an integer between 0 and the maximum rank.
  - **`root`** is the same on all processes.
  - The message specified by **`buf, count, datatype`** has the same signature on all processes.
- The first of these is easy to check on each process at the entry to the MPI_Bcast routine.
- The second two are impossible to check locally; they are consistency requirements requiring communication to check.
- There are many varieties of consistency requirements in the MPI collective operations.

Datatype Signatures

- Consistency requirements for messages in MPI (buf, count, datatype) are not on the MPI datatypes themselves, but on the signature of the message:
  - {$type_1$, $type_2$, …} where $type_i$ is a basic MPI datatype
- So a message described by **`(buf1, 4, MPI_INT)`** matches a message described by **`(buf2, 1, vectype)`**, where **`vectype`** was created to be a strided vector of 4 integers.
- For point-to-point operations, datatype signatures don't have to match exactly (it is OK to receive a short message into a long buffer), but for collective operations, matches must be exact.

Approach

- Use the MPI profiling interface to intercept the collective calls, “borrow” the communicator passed in, and use it to check argument consistency among its processes.
- For example, process 0 can broadcast its value of **`root`**, and each other process can compare with the value it was passed for **`root`**.
- For datatype consistency checks, we will communicate hash values of datatype signatures.
- Reference: Falzone, Chan, Lusk, Gropp, “Collective Error Detection for MPI Collective Operations”, Proceedings of EuroPVM/MPI 2005.

Types of Consistency Checks

- **Call** – checks that all processes have made the same collective call (not MPI_Allreduce on some processes and MPI_Reduce on others).
  - Used in all collective functions
- **Root** – checks that the same value of root was passed on all processes
  - Used in Bcast, Reduce, Gather(v), Scatter(v), Spawn, Spawn_multiple, Connect
- **Datatype** – checks consistency of data arguments
  - Used in all collective routines with data buffer arguments
- **Op** – checks consistency of operations
  - Used in Reduce, Allreduce, Reduce_scatter, Scan, Exscan

More Types of Consistency Checks

- **MPI_IN_PLACE** – checks whether all process or none of the processes specified MPI_IN_PLACE instead of a buffer.
  - Used in Allgather(v), Allreduce, and Reduce_scatter
- **Local leader and tag** – checks consistency of these arguments
  - Used only in MPI_Intercomm_create
- **High/low** – checks consistency of these arguments
  - Used only in MPI_Intercomm_merge
- **Dims** – checks consistency of these arguments
  - Used in Cart_create and Cart_map

Still More Types of Consistency Checks

- Graph – checks graph consistency
  - Used in Graph_create and Graph_map
- Amode – checks file mode argument consistency
  - Used in File_open
- Size, datarep, flag – checks consistency of these I/O arguments
  - Used in File_set_size, File_set_automicity, File_preallocate
- Etype – checks consistency of this argument
  - Used in File_set_view
- Order – checks that split-collective calls are properly ordered
  - Used in Read_all_begin, Read_all_end, other split collective I/O

Example Output

- We try to make error output instance specific:

- **`Validate Bcast error (Rank 4) – root parameter (4) is inconsistent with rank 0's (0)`**

- **`Validate Bcast error (Rank 4) – datatype signature is inconsistent with Rank 0's`**

- **`Validate Barrier (rank 4) – collective call (Barrier) is inconsistent with Rank 0's (Bcast)`**

Experiences

- Finding errors
  - Found error in MPICH2 test suite, in which a message with one MPI_INT was allowed to match sizeof(int) MPI_BYTEs.
  - MPICH2 allowed the match, but shouldn't have.
  - Ran large astrophysics application (FLASH) containing many collective operations
    - Collective calls all in third-party AMR library (Paramesh), but could still be examined through MPI profiling library approach.
    - Found no errors ☺(☹)
- Portability, Performance
  - Linux cluster (MPICH2)
  - Blue Gene (IBM's BG/L MPI)
  - Relative overhead decreases as size of message increases
    - The extra checking messages are much shorter than the real messages
  - Overhead can be relatively large for small messages
    - Opportunities for optimization remain
  - Profiling library can be removed after finding errors

MPI and Threads

MPI and Threads

- MPI describes parallelism between *processes* (with separate address spaces)
- *Thread* parallelism provides a shared-memory model within a process
- OpenMP and Pthreads are common models
  - OpenMP provides convenient features for loop-level parallelism. Threads are created and managed by the compiler, based on user directives.
  - Pthreads provide more complex and dynamic approaches. Threads are created and managed explicitly by the user.

Programming for Multicore

- Almost all chips are multicore these days
- Today's clusters often comprise multiple CPUs per node sharing memory, and the nodes themselves are connected by a network
- Common options for programming such clusters
  - All MPI
    - Use MPI to communicate between processes both within a node and across nodes
    - MPI implementation internally uses shared memory to communicate within a node
  - MPI + OpenMP
    - Use OpenMP within a node and MPI across nodes
  - MPI + Pthreads
    - Use Pthreads within a node and MPI across nodes
- The latter two approaches are known as "hybrid programming"

MPI's Four Levels of Thread Safety

- MPI defines four levels of thread safety. These are in the form of commitments the application makes to the MPI implementation.
  - MPI_THREAD_SINGLE: only one thread exists in the application
  - MPI_THREAD_FUNNELED: multithreaded, but only the main thread makes MPI calls (the one that called MPI_Init or MPI_Init_thread)
  - MPI_THREAD_SERIALIZED: multithreaded, but only one thread *at a time* makes MPI calls
  - MPI_THREAD_MULTIPLE: multithreaded and any thread can make MPI calls at any time (with some restrictions to avoid races – see next slide)
- MPI defines an alternative to MPI_Init
  - MPI_Init_thread(requested, provided)
    - *Application indicates what level it needs; MPI implementation returns the level it supports*

Specification of 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
  - e.g., accessing an info object from one thread and freeing it from another thread
- User must ensure that collective operations on the same communicator, window, or file handle are correctly ordered among threads
  - e.g., cannot call a broadcast on one thread and a reduce on another thread on the same communicator

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-safe implementation will support MPI_THREAD_MULTIPLE
- A program that calls MPI_Init (instead of MPI_Init_thread) should assume that only MPI_THREAD_SINGLE is supported
- *A threaded MPI program that does not call MPI_Init_thread is an incorrect program (common user error we see)*

An Incorrect Program

	Process 0	*Process 1*
Thread 1	MPI_Bcast(comm)	MPI_Bcast(comm)
Thread 2	MPI_Barrier(comm)	MPI_Barrier(comm)

- Here the user must use some kind of synchronization to ensure that either thread 1 or thread 2 gets scheduled first on both processes
- Otherwise a broadcast may get matched with a barrier on the same communicator, which is not allowed in MPI

A Correct Example

	Process 0	*Process 1*
Thread 1	MPI_Recv(src=1)	MPI_Recv(src=0)
Thread 2	MPI_Send(dst=1)	MPI_Send(dst=0)

- An implementation must ensure that the above example never deadlocks for any ordering of thread execution
- That means the implementation cannot simply acquire a thread lock and block within an MPI function. It must release the lock to allow other threads to make progress.

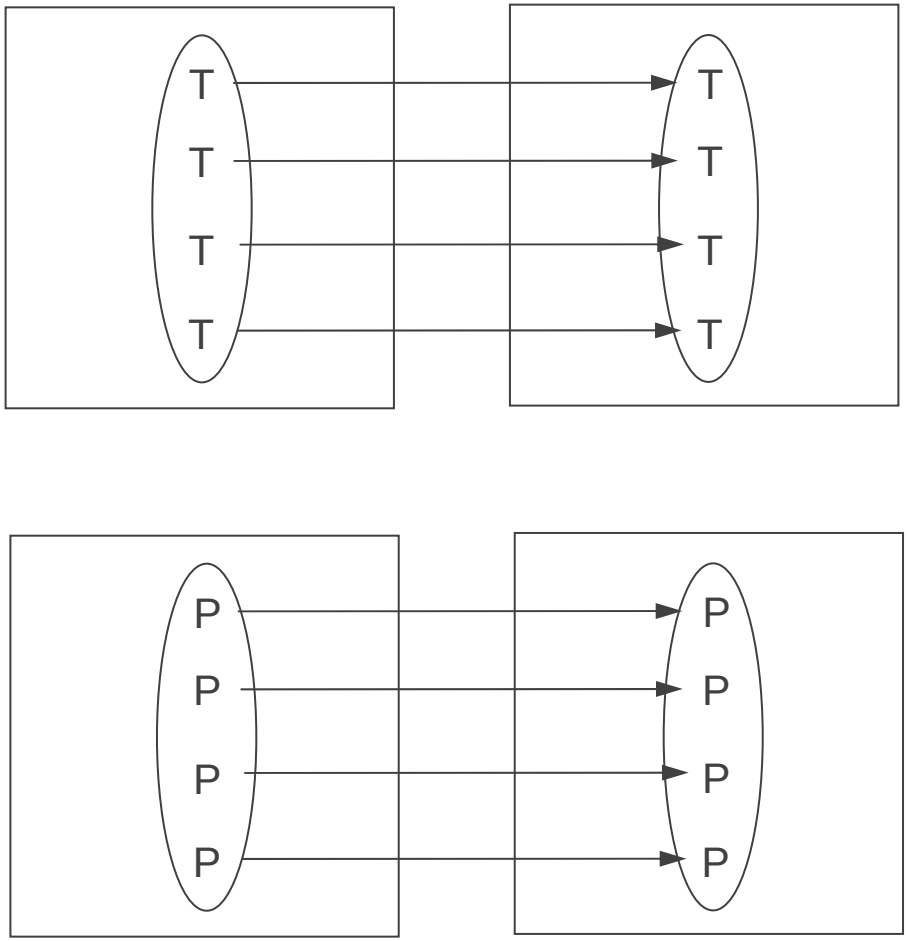
The Current Situation

- All MPI implementations support MPI_THREAD_SINGLE (duh).
- They probably support MPI_THREAD_FUNNELED even if they don't admit it.
  - Does require thread-safe malloc
  - Probably OK in OpenMP programs
- Many (but not all) implementations support THREAD_MULTIPLE
  - Hard to implement efficiently though (lock granularity issue)
- "Easy" OpenMP programs (loops parallelized with OpenMP, communication in between loops) only need FUNNELED
  - So don't need "thread-safe" MPI for many hybrid programs
  - But watch out for Amdahl's Law!

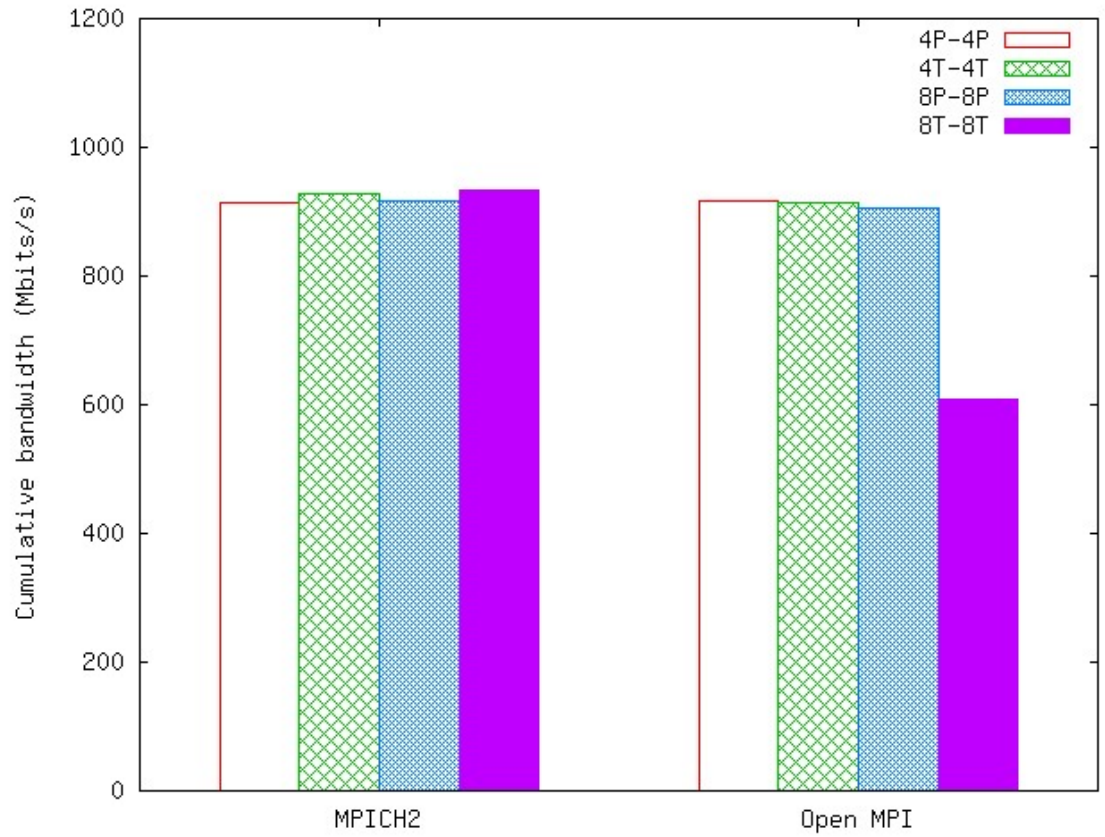
Performance with MPI_THREAD_MULTIPLE

- Thread safety does not come for free
- The 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 our *Parallel Computing* (journal) paper (2009)

Tests with Multiple Threads versus Processes

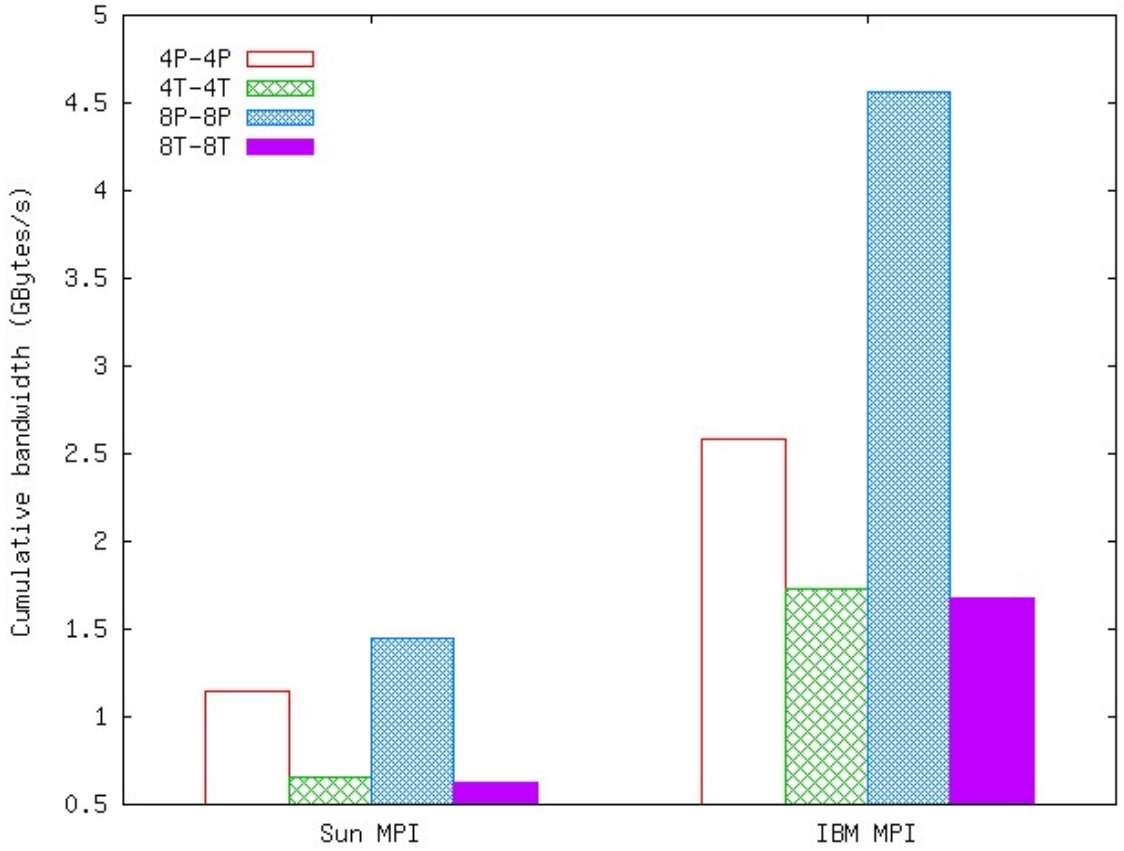


Concurrent Bandwidth Test on Linux Cluster

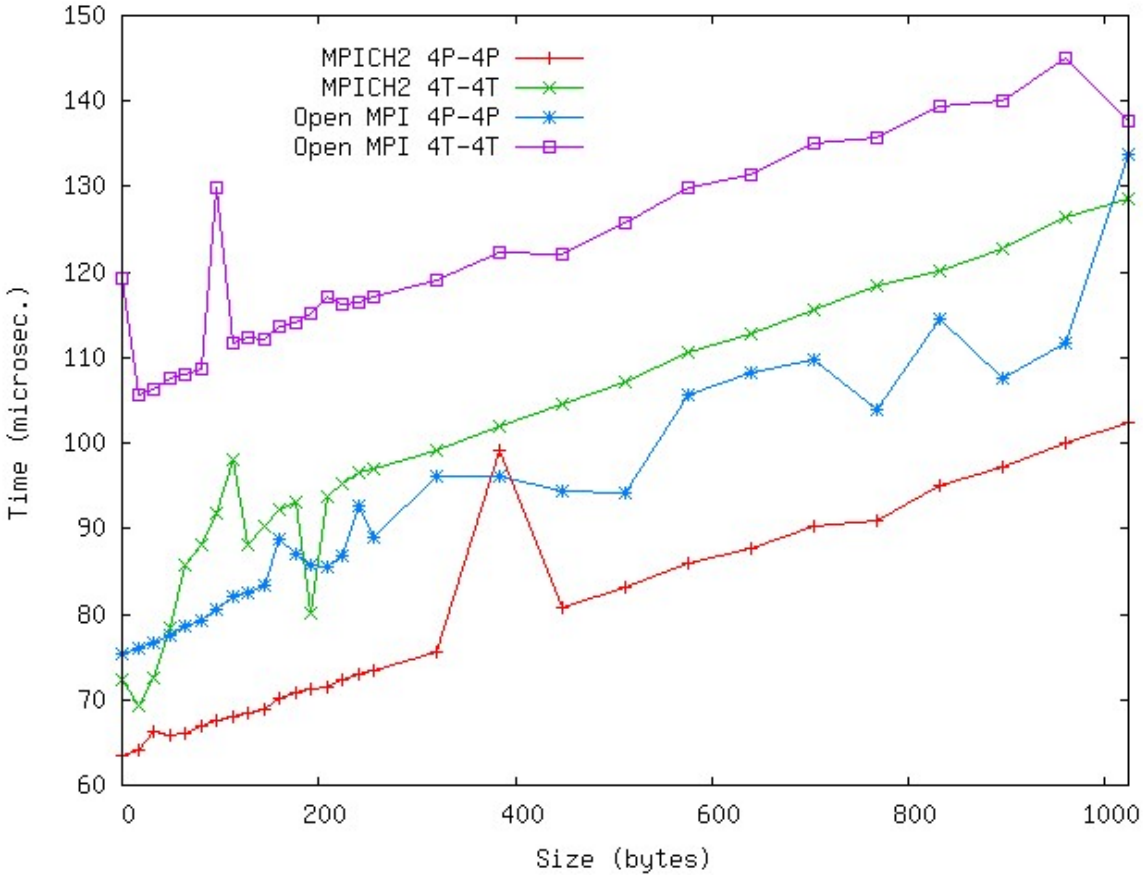


MPICH2 version 1.0.5
Open MPI version 1.2.1

Concurrent Bandwidth Test on a single SMP (Sun and IBM)

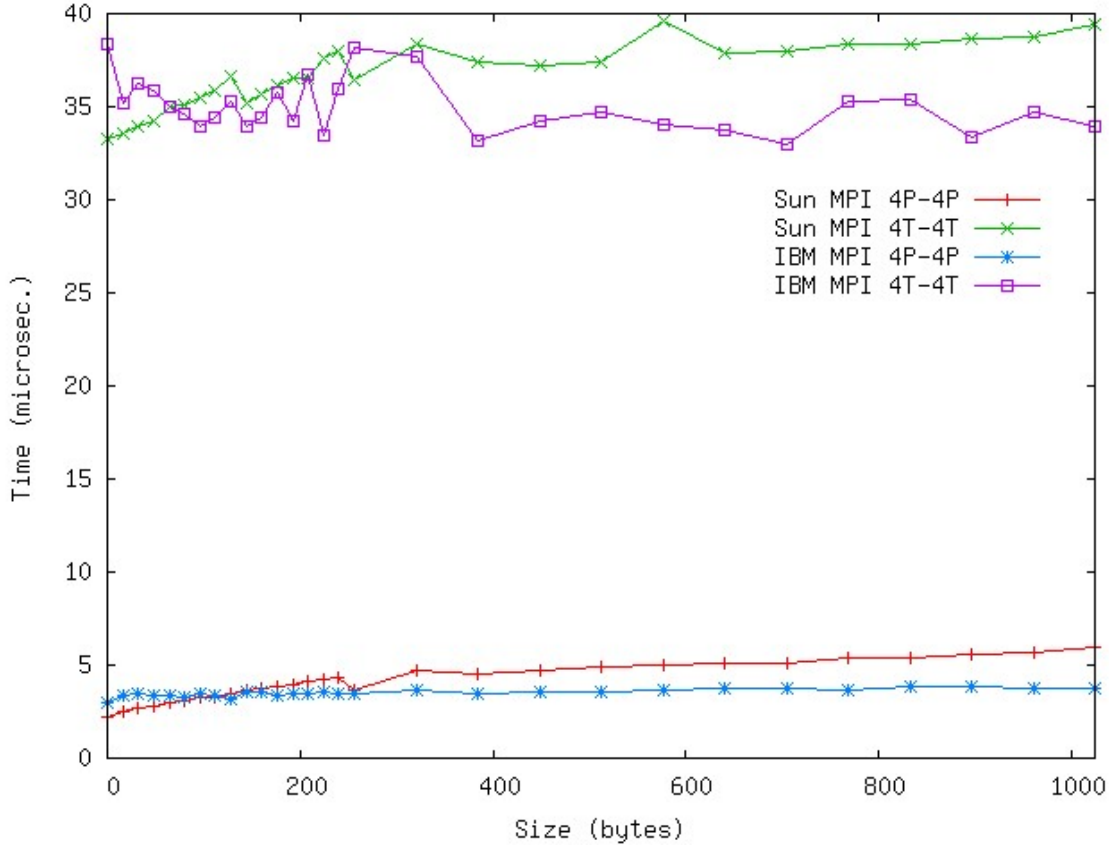


Concurrent Latency Test on Linux Cluster



MPICH2 version 1.0.5
Open MPI version 1.2.1

Concurrent Latency Test on a single SMP (Sun and IBM)



What MPI's Thread Safety Means in the Hybrid MPI+OpenMP Context

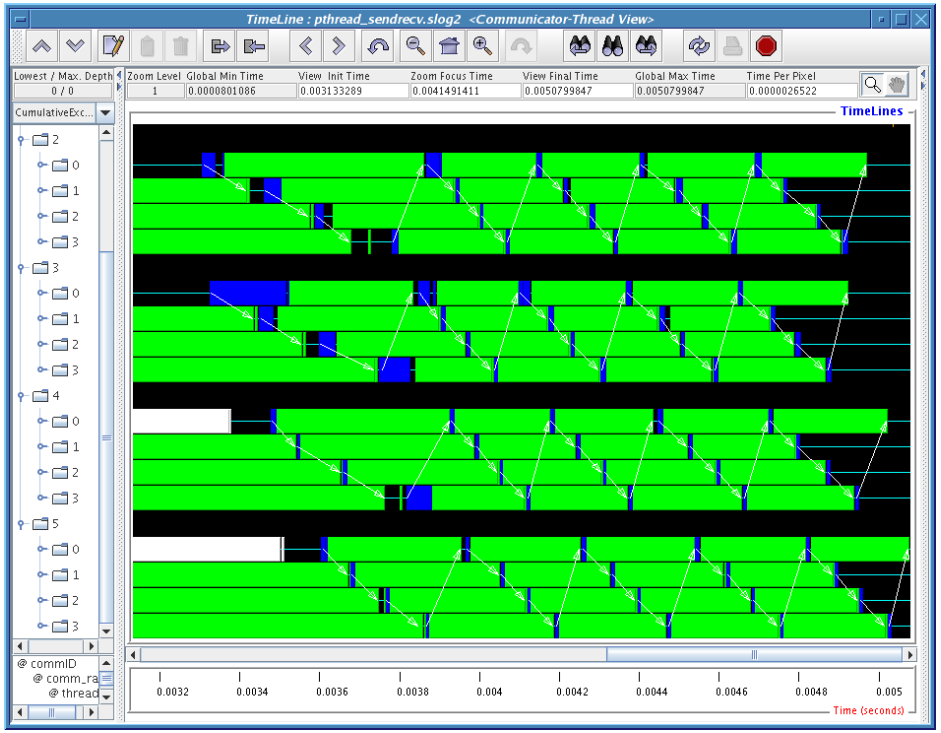
- MPI_THREAD_SINGLE
  - There is no OpenMP multithreading in the program.
- MPI_THREAD_FUNNELED
  - All of the MPI calls are made by the master thread. i.e. all MPI calls are
    - *Outside OpenMP parallel regions, or*
    - *Inside OpenMP master regions, or*
    - *Guarded by call to MPI_Is_thread_main MPI call.*
      - (same thread that called MPI_Init_thread)
- MPI_THREAD_SERIALIZED

```
#pragma omp parallel
…
#pragma omp atomic
{
  …MPI calls allowed here…
}
```

- MPI_THREAD_MULTIPLE
  - Any thread may make an MPI call at any time

Visualizing Hybrid Programs with Jumpshot

- Recent additions to Jumpshot for multithreaded and hybrid programs that use Pthreads
  - Separate timelines for each thread id
  - Support for grouping threads by communicator as well as by process



Using Jumpshot with Hybrid MPI+OpenMP Programs

- SLOG2/Jumpshot needs two properties of the OpenMP implementation that are not guaranteed by the OpenMP standard
  - OpenMP threads must be Pthreads
    - *Otherwise, the locking in the logging library (which uses Pthread locks) necessary to preserve exclusive access to the logging buffers would need to be modified*
  - These Pthread ids must be reused (threads are "parked" when not in use)
    - *Otherwise Jumpshot would need zillions of time lines*

Three Platforms for Hybrid Programming Experiments

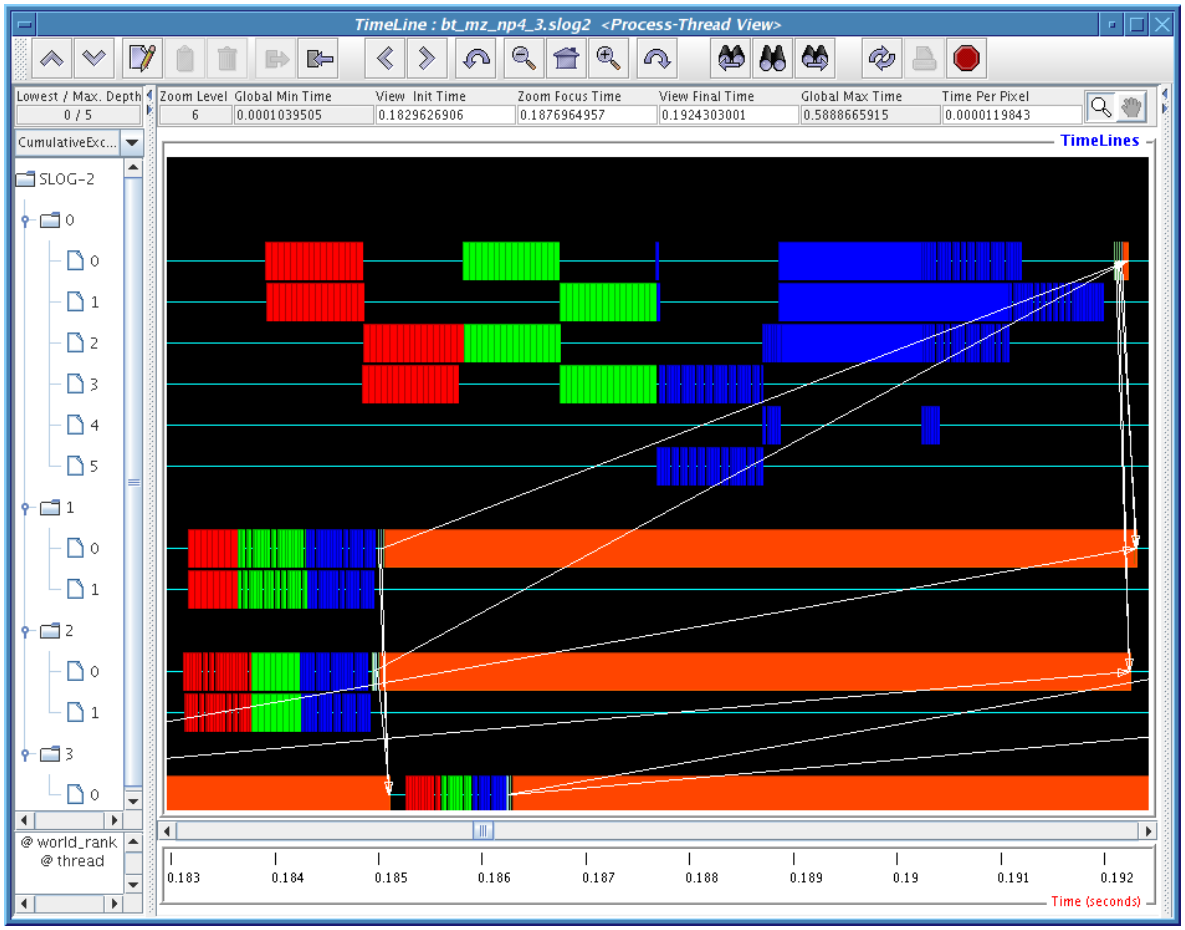
- Linux cluster
  - 24 nodes, each with two Opteron dual-core processors, 2.8 Ghz each
  - Intel 9.1 Fortran compiler
  - MPICH2-1.0.6, which has MPI_THREAD_MULTIPLE
  - Multiple networks; we used GigE
- IBM Blue Gene/P
  - 40,960 nodes, each consisting of four PowerPC 850 MHz cores
  - XLF 11.1 Fortran cross-compiler
  - IBM's MPI V1R1M2 (based on MPICH2), has MPI_THREAD_MULTIPLE
  - 3D Torus and tree networks
- SiCortex SC5832
  - 972 nodes, each consisting of six MIPS 500 MHz cores
  - Pathscale 3.0.99 Fortran cross-compiler
  - SiCortex MPI implementation based on MPICH2, has MPI_THREAD_FUNNELED
  - Kautz graph network

Experiments

- Basic
  - Proved that necessary assumptions for our tools hold
    - *OpenMP threads are Pthreads*
    - *Thread id's are reused*
- NAS Parallel Benchmarks
  - NPB-MZ-MPI, version 3.1
  - Both BT and SP
  - Two different sizes (W and B)
  - Two different modes ("MPI everywhere" and OpenMP/MPI)
    - *With four nodes on each machine*
- Demonstrated satisfying level of portability of programs and tools across three quite different hardware/software environments

It Might Not Be Doing What You Think

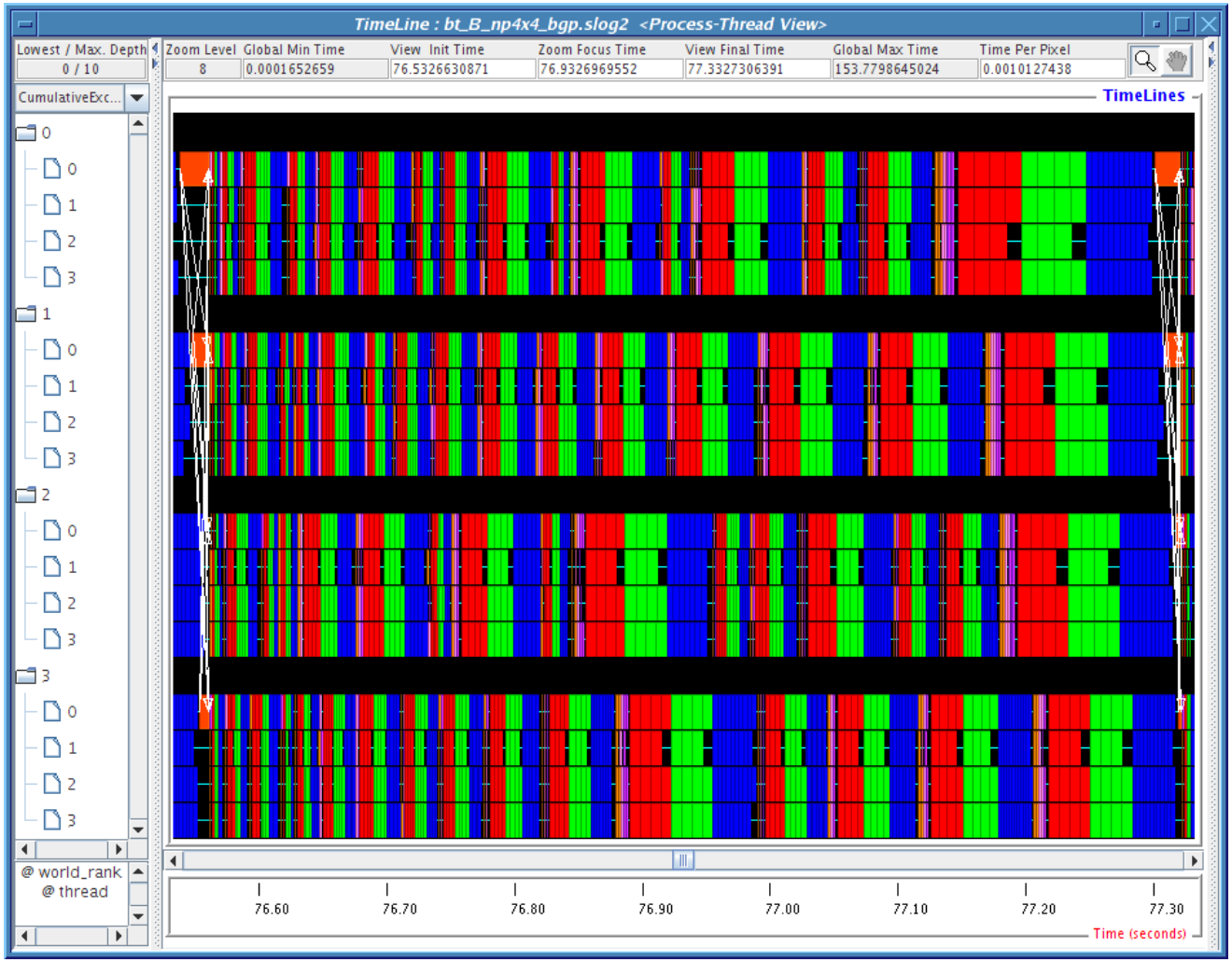
- An early run:



- Nasty interaction between the environment variables OMP_NUM_THREADS and NPB_MAX_THREADS

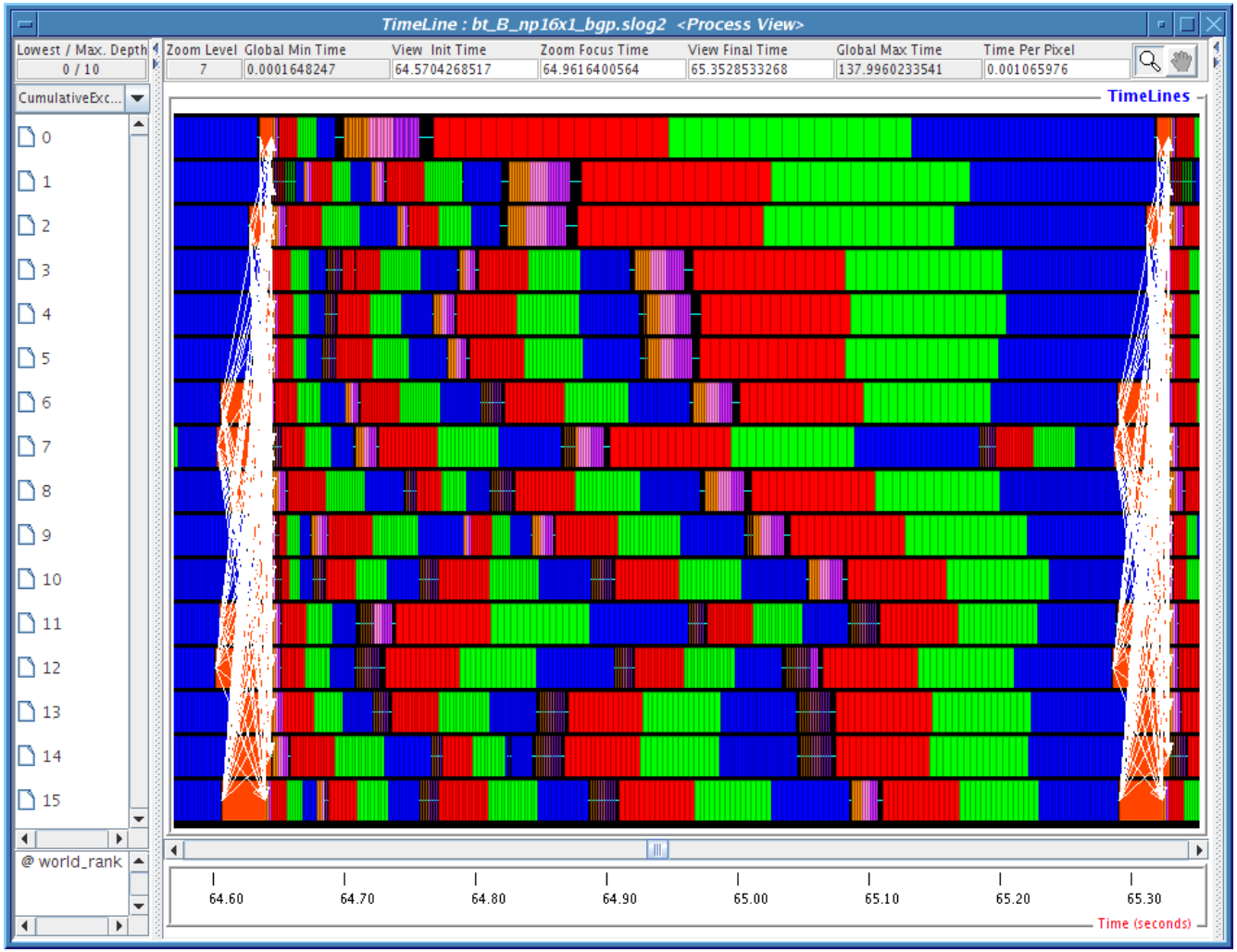
More Like What You Expect

- BT class B on 4 BG/P nodes, using OpenMP on each node



MPI Everywhere

- BT class B on 4 BG/P nodes, using 16 MPI processes



Observations on Experiments

Experiment	Cluster	BG/P	SiCortex
Bt-mz.W.16x1	1.84	9.46	20.60
Bt-mz-W.4x4	0.82	3.74	11.26
Sp-mz.W.16x1	0.42	1.79	3.72
Sp-mz.W.4x4	0.78	3.00	7.98
Bt-mz.B.16.1	24.87	113.31	257.67
Bt-mz.B.4x4	27.96	124.60	399.23
Sp-mz.B.16x1	21.19	70.69	165.82
Sp-mz.B.4x4	24.03	81.47	246.76
Bt-mz.B.24x1			241.85
Bt-mz.B.4x6			337.86
Sp-mz.B.24x1			127.28
Sp-mz.B.4x6			211.78

- Time in seconds
- On the small version of BT (W), hybrid was better
- For SP and size B problems, MPI everywhere is better
- On SiCortex, more processes or threads are better than fewer

Observations

- This particular benchmark has been studied much more deeply elsewhere
  - Rolf Rabenseifner, "Hybrid parallel programming on HPC platforms," *Proceedings of EWOMP'03.*
  - Barbara Chapman, Gabriele Jost, and Ruud van der Pas, *Using OpenMP: Portable Shared Memory Parallel Programming, MIT Press, 2008.*
- Adding "hybridness" to a well-tuned MPI application is not going to speed it up. So this NPB study doesn't tell us much.
- More work is needed to understand the behavior of hybrid programs and what is needed for future application development.

A Few Words of Warning about Threads

Thread Programming is Hard

- *"The Problem with Threads,"* IEEE Computer
  - Prof. Ed Lee, UC Berkeley
  - http://ptolemy.eecs.berkeley.edu/publications/papers/06/problemwithThreads/
- *"Why Threads are a Bad Idea (for most purposes)"*
  - John Ousterhout
  - http://home.pacbell.net/ouster/threads.pdf
- *"Night of the Living Threads"*
  http://weblogs.mozillazine.org/roc/archives/2005/12/night_of_the_living_threads.html
- Too hard to know whether code is correct
- Too hard to debug
  - I would rather debug an MPI program than a threads program

Ptolemy and Threads

- Ptolemy is a framework for modeling, simulation, and design of concurrent, real-time, embedded systems
- Developed at UC Berkeley (PI: Ed Lee)
- It is a rigorously tested, widely used piece of software
- Ptolemy II was first released in 2000
- Yet, on April 26, 2004, four years after it was first released, the code deadlocked!
- The bug was lurking for 4 years of widespread use and testing!
- A faster machine or something that changed the timing caught the bug

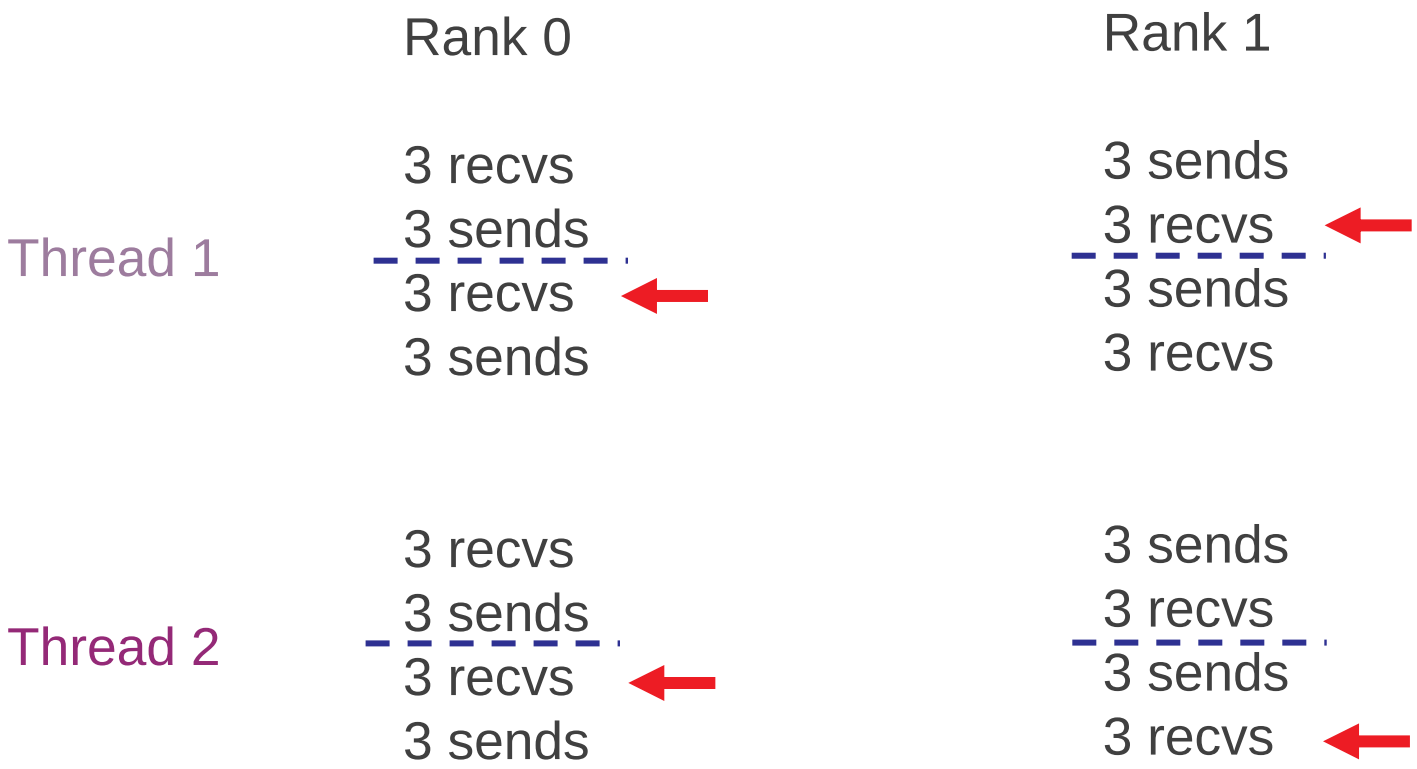
An Example I encountered recently

- We received a bug report about a very simple multithreaded MPI program that hangs
- Run with 2 processes
- Each process has 2 threads
- Both threads communicate with threads on the other process as shown in the next slide
- I spent several hours trying to debug MPICH2 before discovering that the bug is actually in the user's program ☹

2 Proceses, 2 Threads, Each Thread Executes this Code

```
for (j = 0; j < 2; j++) {
    if (rank == 1) {
        for (i = 0; i < 3; i++)
            MPI_Send(NULL, 0, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
        for (i = 0; i < 3; i++)
            MPI_Recv(NULL, 0, MPI_CHAR, 0, 0, MPI_COMM_WORLD, &stat);
    }
   else {  /* rank == 0 */
        for (i = 0; i < 3; i++)
            MPI_Recv(NULL, 0, MPI_CHAR, 1, 0, MPI_COMM_WORLD, &stat);
        for (i = 0; i < 3; i++)
            MPI_Send(NULL, 0, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
    }
}
```

What Happened



- All 4 threads stuck in receives because the sends from one iteration got matched with receives from the next iteration
- Solution: Use iteration number as tag in the messages

Shared-Memory Programming is also Hard

- We (Thakur, Ross, Latham) developed a distributed byte-range locking algorithm using MPI one-sided communication
- Published at Euro PVM/MPI 2005
- Six months later, a group at Univ. of Utah model checked the algorithm and discovered a race condition that results in deadlock
- I could reproduce the bug after being told about it ☹
- Our own testing had not caught it, nor did the reviewers of our paper
- The bug is timing related, caused by a specific interleaving of events
- It is very hard for the human mind to figure out all possible interleavings

Formal Verification as a Tool

- In collaboration with Univs. of Utah and Delaware, we have had success in using formal methods to verify and debug MPI and threaded programs
- Two approaches
  1. Build a model of the program (by hand) using a modeling language, such as Promela. Verify it using a model checker such as SPIN
     -- Tedious, error prone
  2. Verify the program without creating a model (in-situ model checking)
- We are exploring both options with our collaborators at Utah and Delaware (Profs. Ganesh Gopalakrishnan and Stephen Siegel)

Commercial Uses of Formal Verification for Software

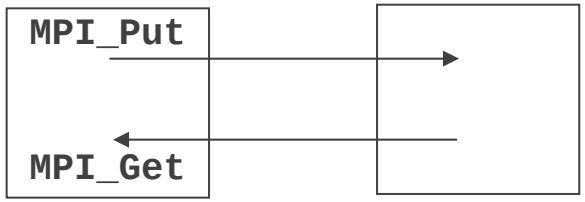
- Coverity is a company founded by Stanford professor, Dawson Engler
- Their software detects bugs in Linux and FreeBSD kernel code and reports to the respective mailing lists
- Also funded by Dept of Homeland Security to detect bugs and vulnerabilities in commonly used open-source software such as Perl, Python, TCL,…

- Microsoft uses FV to verify device-driver code

One-Sided Communication

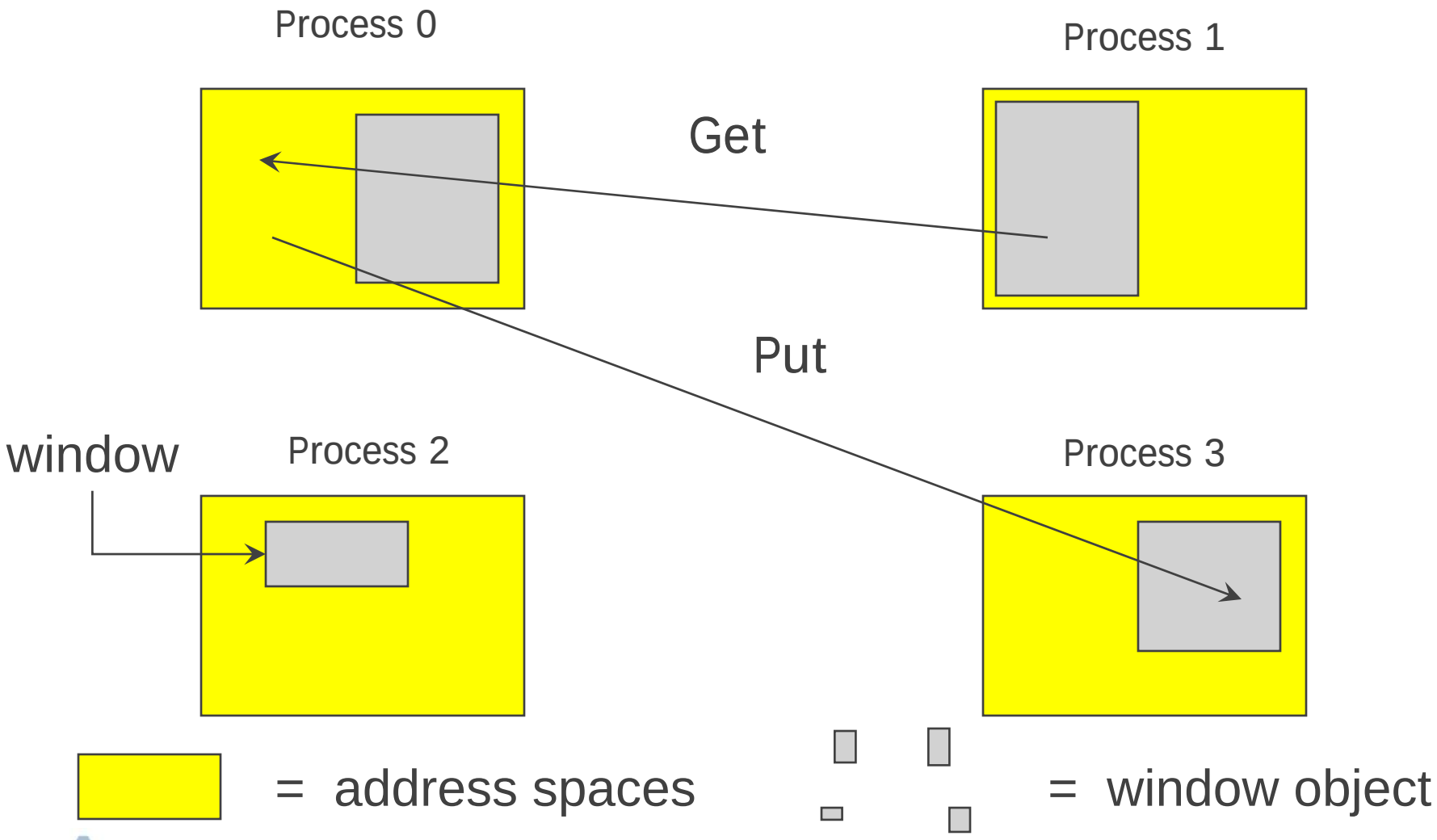
One-Sided Communication

- A process can directly access another process's memory (with a function call)
- Three data transfer functions
  - **`MPI_Put, MPI_Get, MPI_Accumulate`**



- Three synchronization methods
  - **`MPI_Win_fence`**
  - **`MPI_Win_post/start/complete/wait`**
  - **`MPI_Win_lock/unlock`**

Remote Memory Access Windows and Window Objects



Window Creation

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

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

- **`MPI_Win_free`** deallocates window object

Fence Synchronization

Process 0

```
MPI_Win_fence(win)

MPI_Put(1)
MPI_Get(1)

MPI_Win_fence(win)
```

Process 1

```
MPI_Win_fence(win)

MPI_Put(0)
MPI_Get(0)

MPI_Win_fence(win)
```

- **`MPI_Win_fence`** is collective over the communicator associated with the window object
- (The numbers in parentheses refer to the target ranks)

Post-Start-Complete-Wait Synchronization

Process 0	Process 1	Process 2
	`MPI_Win_post(0,2)`	
`MPI_Win_start(1)`		`MPI_Win_start(1)`
`MPI_Put(1)`		`MPI_Put(1)`
`MPI_Get(1)`		`MPI_Get(1)`
`MPI_Win_complete(1)`		`MPI_Win_complete(1)`
	`MPI_Win_wait(0,2)`	

- Scalable: Only the communicating processes need to synchronize
- (The numbers in parentheses refer to the target ranks)

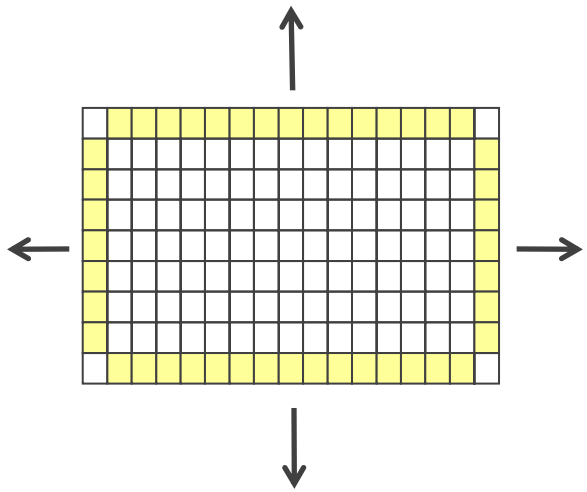
Lock-Unlock Synchronization

Process 0	Process 1	Process 2
`MPI_Win_create`	`MPI_Win_create`	`MPI_Win_create`
`MPI_Win_lock(shared,1)`		`MPI_Win_lock(shared,1)`
`MPI_Put(1)`		`MPI_Put(1)`
`MPI_Get(1)`		`MPI_Get(1)`
`MPI_Win_unlock(1)`		`MPI_Win_unlock(1)`
`MPI_Win_free`	`MPI_Win_free`	`MPI_Win_free`

- "Passive" target: The target process does not make any synchronization call
- (The numbers in parentheses refer to the target ranks)

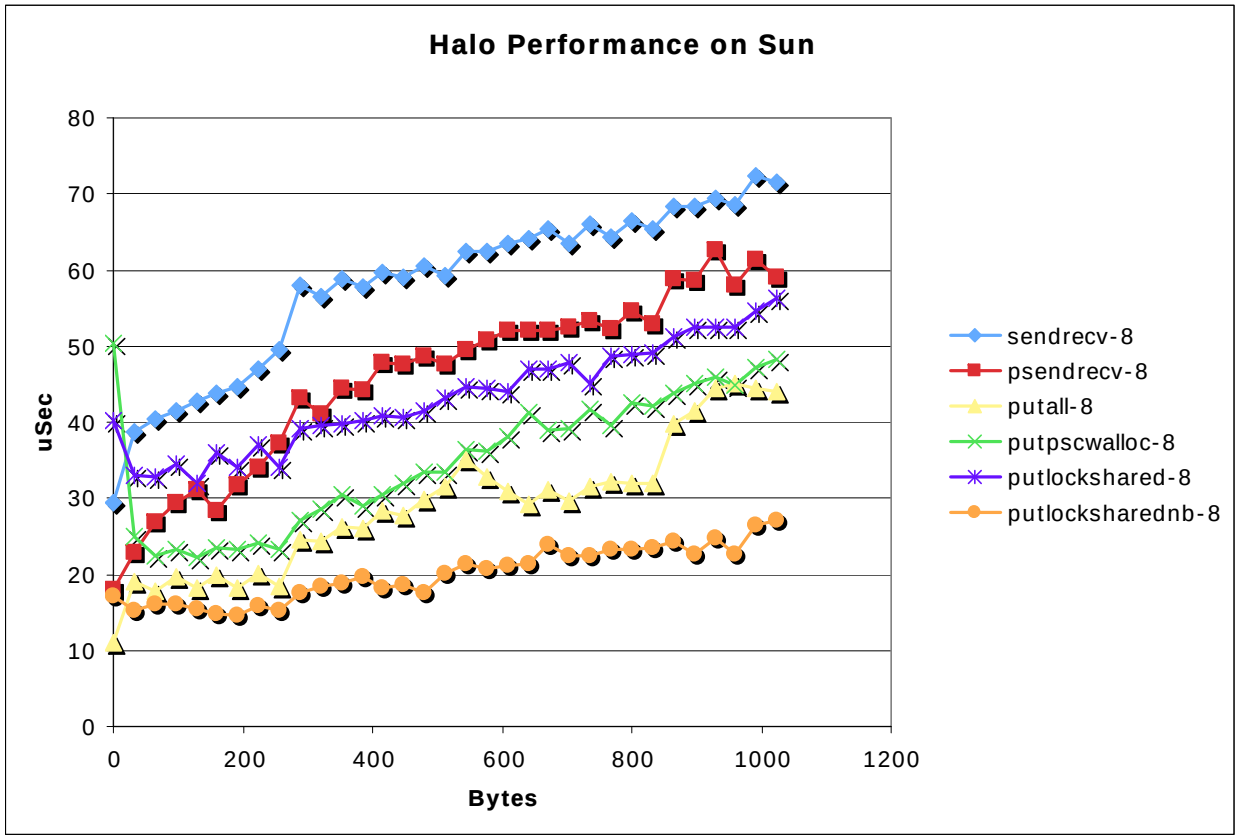
Performance Tests

- "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

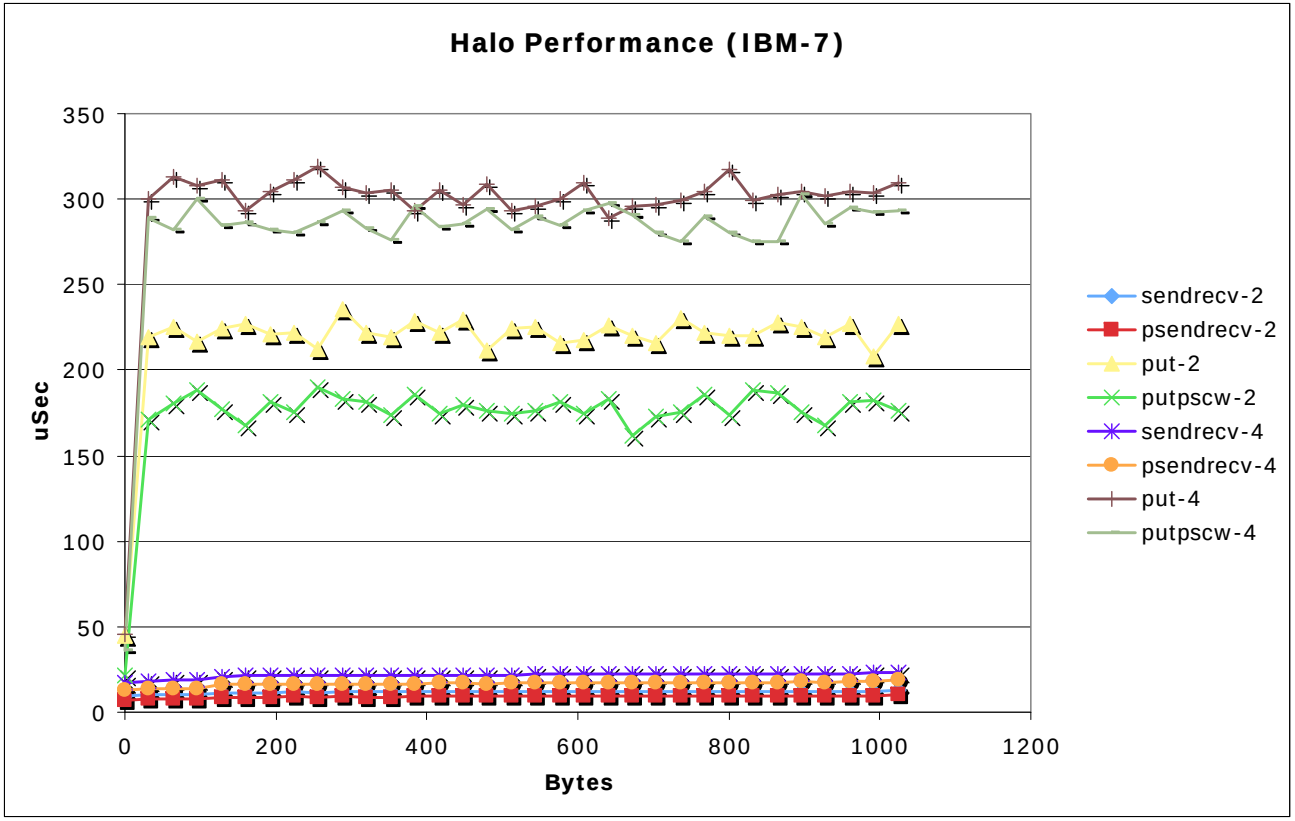


- Ran on
  - Sun Fire SMP at Univ. of Aachen, Germany
  - IBM p655+ SMP at San Diego Supercomputer Center

One-Sided Communication on Sun SMP with Sun MPI



One-Sided Communication on IBM SMP with IBM MPI

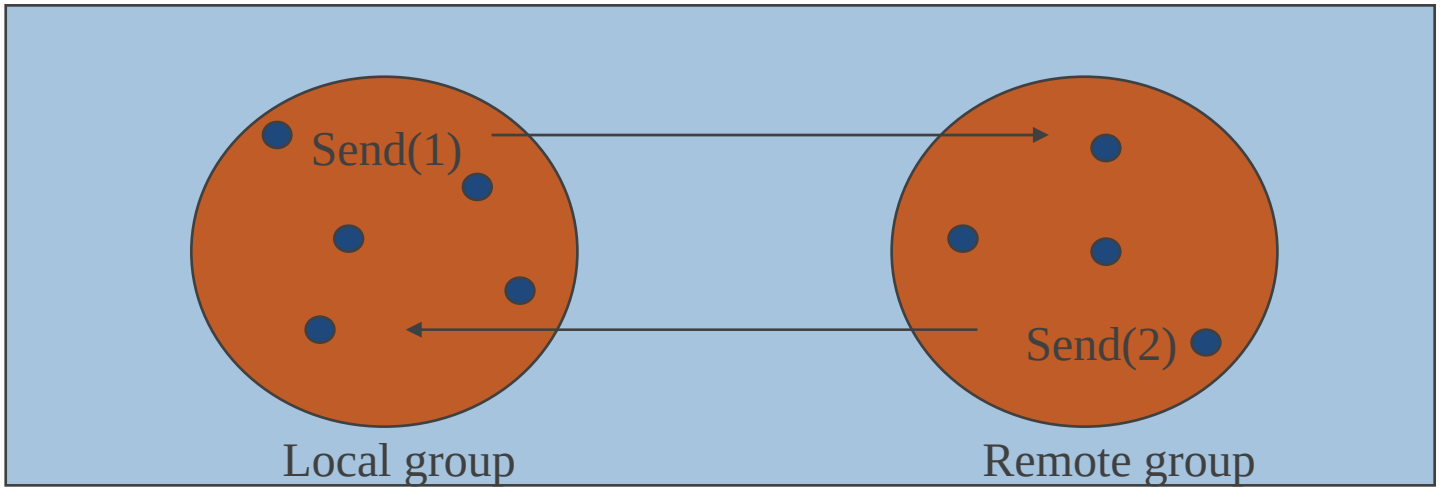


Dynamic Process Management

Intercommunicators

- MPI has two kinds of communicators: intra and inter communicators
- Intracommunicators
  - Contain a single group of processes
  - MPI_COMM_WORLD is an intracommunicator
- Intercommunicators
  - Contain a *local* group and a *remote* group
  - Point-to-point communication is between a process in one group and a process in the other.
  - Can be merged into a normal (intra) communicator
  - Created by **`MPI_Intercomm_create`** in MPI-1
  - Play a more important role in MPI-2, created in multiple ways

Intercommunicators



- In MPI-1, created out of separate intracommunicators
- In MPI-2, created by partitioning an existing intracommunicator
- In MPI-2, the intracommunicators may come from different MPI_COMM_WORLDs
- We'll cover the creation of intercommunicators with the MPI-1 MPI_Intercomm_create in our discussion of fault tolerance, then create intercommunicators with the MPI-2 dynamic process routines

Dynamic Process Management

- Issues
  - maintaining simplicity, flexibility, and correctness
  - interaction with operating system, resource manager, and process manager
  - connecting independently started processes
- Spawning new processes is *collective,* returning an intercommunicator.
  - Local group is group of spawning processes.
  - Remote group is group of new processes.
  - New processes have own **`MPI_COMM_WORLD`**.
  - **`MPI_Comm_get_parent`** lets new processes find parent communicator.

Spawning Processes

```
MPI_Comm_spawn(command, argv, numprocs, info, root,
  comm, intercomm, errcodes)
```

- Tries to start **numprocs** process running **command**, passing them command-line arguments **argv**
- The operation is collective over **comm**
- Spawnees are in remote group of **intercomm**
- Errors are reported on a per-process basis in **errcodes**
- **Info** used to optionally specify hostname, archname, wdir, path, file, softness.

Spawning Multiple Executables

- **`MPI_Comm_spawn_multiple( ... )`**

- Arguments **`command, argv, numprocs, info`** all become arrays
- Still collective

In the Children

- **`MPI_Init`** (only MPI programs can be spawned)
- **`MPI_COMM_WORLD`** is processes spawned with one call to **`MPI_Comm_spawn`**
- **`MPI_Comm_get_parent`** obtains parent intercommunicator
  - Same as intercommunicator returned by **`MPI_Comm_spawn`** in parents
  - Remote group is spawners
  - Local group is those spawned

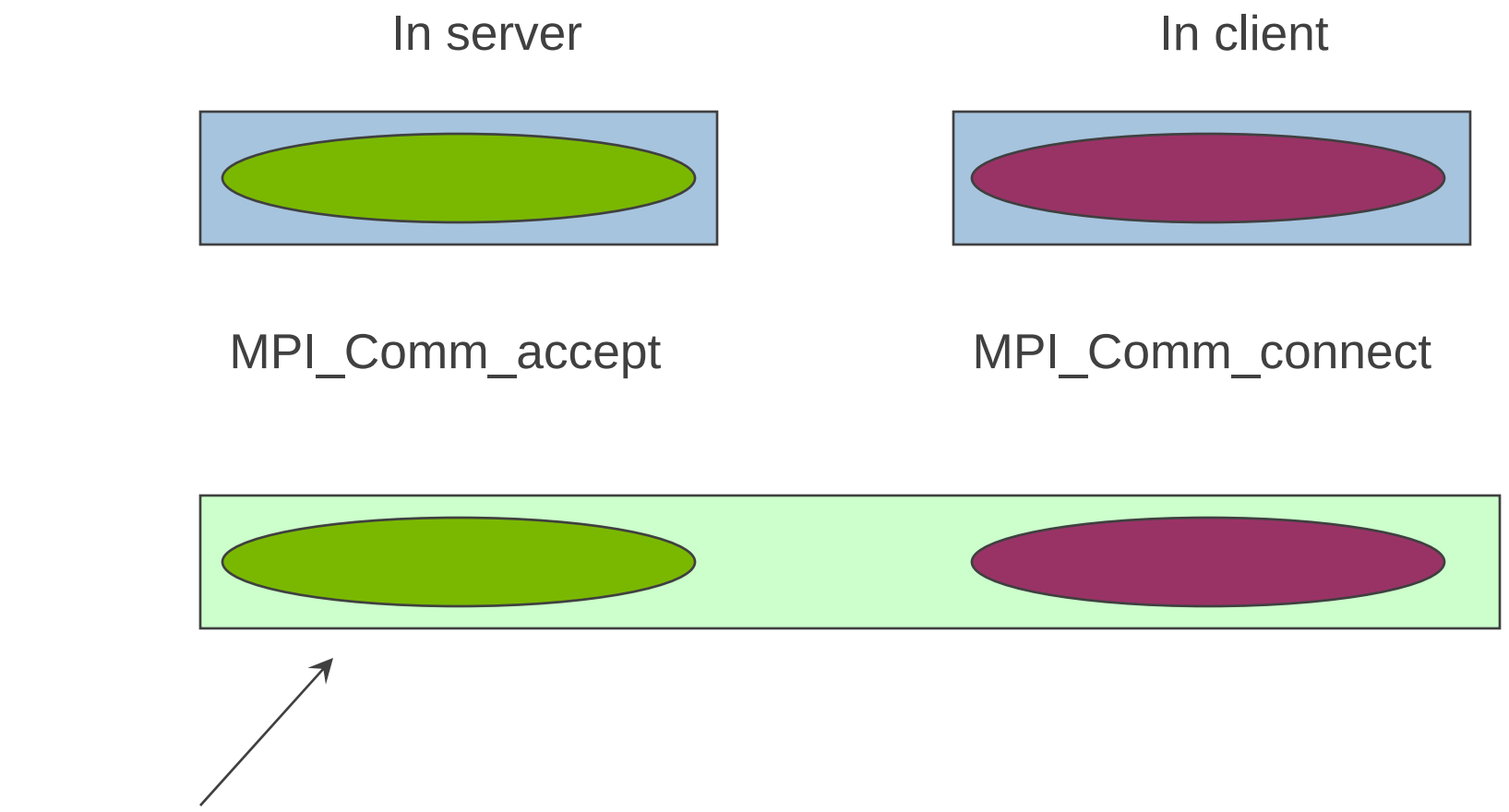
Manager-Worker Example

- Single manager process decides how many workers to create and which executable they should run
- Manager spawns n workers, and addresses them as 0, 1, 2, ..., n-1 in new intercomm
- Workers address each other as 0, 1, ... n-1 in **`MPI_COMM_WORLD`**, address manager as 0 in parent intercomm
- One can find out how many processes can usefully be spawned

Establishing Connections

- Two sets of MPI processes may wish to establish connections, e.g.,
  - Two parts of an application started separately
  - A visualization tool wishes to attach to an application
  - A server wishes to accept connections from multiple clients Both server and client may be parallel programs
- Establishing connections is collective but asymmetric (“Client”/“Server”)
- Connection results in an intercommunicator

Establishing Connections Between Parallel Programs



Connecting Processes

- Server:
  - **`MPI_Open_port( info, port_name )`**
    - system supplies port_name
    - might be host:num; might be low-level switch #
  - **`MPI_Comm_accept( port_name, info, root, comm, intercomm )`**
    - collective over comm
    - returns intercomm; remote group is clients
- Client:
  - **`MPI_Comm_connect( port_name, info, root, comm, intercomm )`**
    - remote group is server

Optional Name Service

```
MPI_Publish_name( service_name, info, port_name )

MPI_Lookup_name( service_name, info, port_name )
```

- allow connection between **`service_name`** known to users and system-supplied **`port_name`**
- MPI implementations are allowed to ignore this service

Bootstrapping

- **`MPI_Join( fd, intercomm )`**
- collective over two processes connected by a socket.
- **`fd`** is a file descriptor for an open, quiescent socket.
- **`intercomm`** is a new intercommunicator.
- Can be used to build up full MPI communication.
- **`fd`** is *not* used for MPI communication.

Argonne NATIONAL LABORATORY

MPI at Exascale

Rajeev Thakur

Mathematics and Computer Science Division

Argonne National Laboratory

U.S. DEPARTMENT OF ENERGY

MPI on the Largest Machines Today

- Systems with the largest core counts in June 2010 Top500 list

Juelich BG/P	294,912 cores
Oak Ridge Cray XT5	224,162 cores
LLNL BG/L	212,992 cores
Argonne BG/P	163,840 cores
LLNL BG/P (Dawn)	147,456 cores

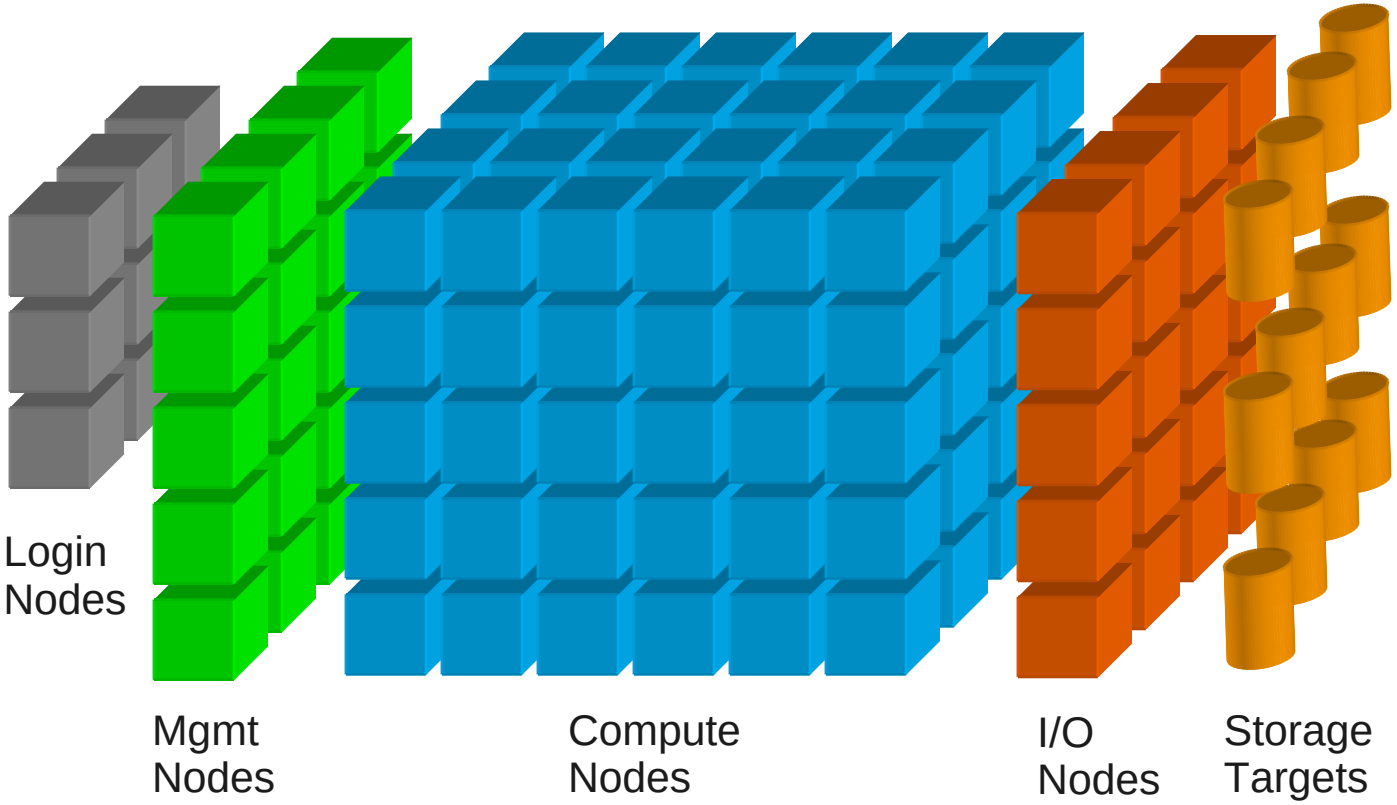
  (All these systems run MPICH2-based MPI implementations)

- In a couple of years, we will have systems with more than a million cores

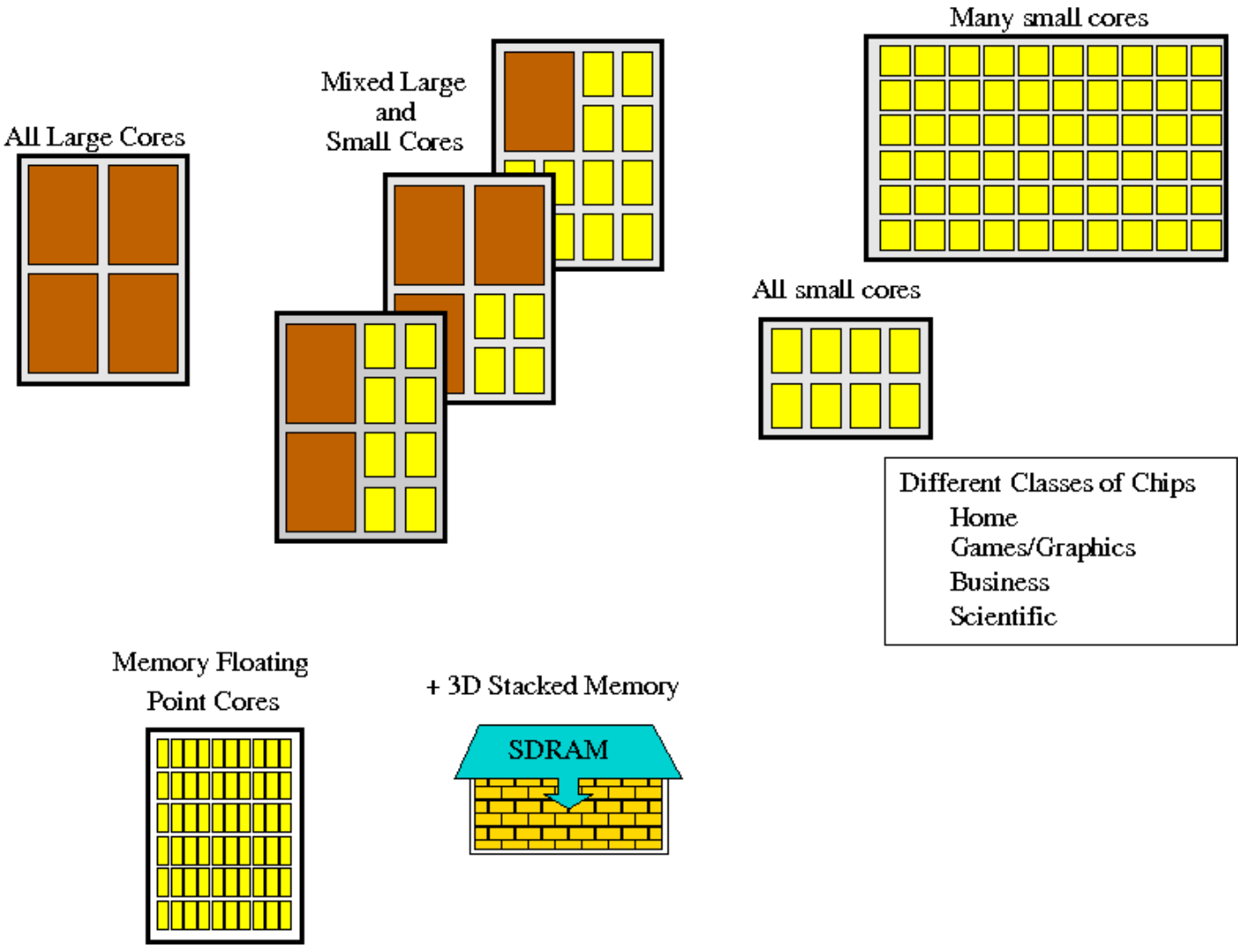
- For example, in 2012, the Sequoia machine at Livermore will be an IBM Blue Gene/Q with 1,572,864 cores (~1.6 million cores)

Future Extreme Scale Platforms

- Hundreds of thousands of “nodes”
- Each node has large numbers of cores, including
  - Regular CPUs and accelerators (e.g., GPUs)



Multiple Cores Per Node



Scaling MPI to Exascale

- MPI already runs on the largest systems today at ~300,000 cores
- What would it take to scale MPI to exascale systems with millions of cores?
- On exascale, MPI is likely to be used as part of a "hybrid programming" model (MPI+X), much more so than it is today
  - MPI being used to communicate between "address spaces"
  - With some other "shared-memory" programming model (OpenMP, UPC, CUDA, OpenCL) for programming within an address space
- How can MPI support efficient "hybrid" programming on exascale systems?

Scaling MPI to Exascale

- Although the original designers of MPI were not thinking of exascale, MPI was always intended and designed with scalability in mind. For example:
  - A design goal was to enable implementations that maintain very little global state per process
  - Another design goal was to require very little memory management within MPI (all memory for communication can be in user space)
  - MPI defines many operations as *collective* (called by a group of processes), which enables them to be implemented scalably and efficiently

- Nonetheless, some parts of the MPI specification may need to be fixed for exascale
  - Being addressed by the MPI Forum in MPI-3

Factors Affecting MPI Scalability

- Performance and memory consumption
- A nonscalable MPI function is one whose time or memory consumption per process increase linearly (or worse) with the total number of processes (all else being equal)
- For example
  - If memory consumption of MPI_Comm_dup increases linearly with the no. of processes, it is not scalable
  - If time taken by MPI_Comm_spawn increases linearly or more with the no. of processes being spawned, it indicates a nonscalable implementation of the function
- Such examples need to be identified and fixed (in the specification and in implementations)
- The goal should be to use constructs that require only constant space per process

Requirements of a message-passing library at extreme scale

- No O(*nprocs*) consumption of resources (memory, network connections) per process
- Resilient and fault tolerant
- Efficient support for hybrid programming (multithreaded communication)
- Good performance over the entire range of message sizes and all functions, not just latency and bandwidth benchmarks
- Fewer performance surprises (in implementations)

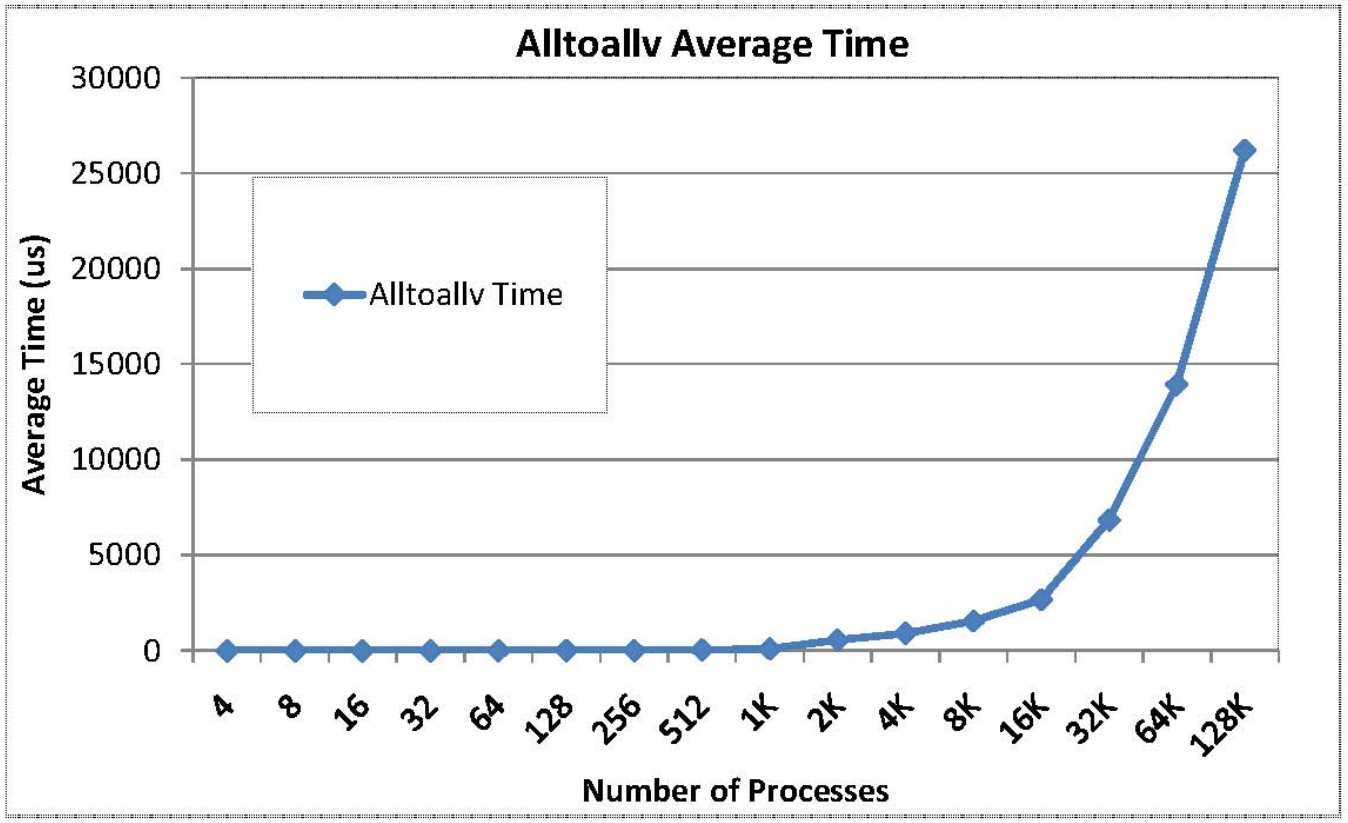
- These issues are being addressed by the MPI Forum for MPI-3 and by MPI implementations

Scalability Issues in the MPI Specification

- Some functions take parameters that grow linearly with number of processes
- E.g., irregular (or "v") version of collectives such as MPI_Gatherv
- Extreme case: MPI_Alltoallw takes six such arrays
  - On a million processes, that requires 24 MB on each process
- On low-frequency cores, even scanning through large arrays takes time (see next slide)

- Solution: The MPI Forum is considering a proposal to define sparse, neighborhood collectives that could be used instead of irregular collectives

Zero-byte MPI_Alltoallv time on BG/P



- This is just the time to scan the parameter array to determine it is all 0 bytes. No communication performed.

Scalability Issues in the MPI Specification

- Graph Topology
  - In MPI 2.1 and earlier, requires the entire graph to be specified on each process
  - Already fixed in MPI 2.2 – new distributed graph topology functions
- One-sided communication
  - Synchronization functions turn out to be expensive
  - Being addressed by RMA working group of MPI-3
- Representation of process ranks
  - Explicit representation of process ranks in some functions, such as MPI_Group_incl and MPI_Group_excl
  - Concise representations should be considered

Scalability Issues in the MPI Specification

- All-to-all communication
  - Not a scalable communication pattern
  - Applications may need to consider newer algorithms that do not require all-to-all
- Fault tolerance
  - Large component counts will result in frequent failures
  - Greater resilience needed from all components of the software stack
  - MPI can return error codes, but need more support than that
  - Being addressed in the fault tolerance group of MPI-3

MPI Implementation Scalability

- MPI implementations must pay attention to two aspects as the number of processes is increased:
  - memory consumption of any function, and
  - performance of *all* collective functions
    - Not just collective communication functions that are commonly optimized
    - Also functions such as MPI_Init and MPI_Comm_split

Process Mappings

- MPI communicators maintain mapping from ranks to processor ids
- This mapping is often a table of O(nprocs) size in the communicator
- Need to explore more memory-efficient mappings, at least for common cases
- More systematic approaches to compact representations of permutations (research problem)
  - See recent paper at HPDC 2010 by Alan Wagner et al. from the University of British Columbia

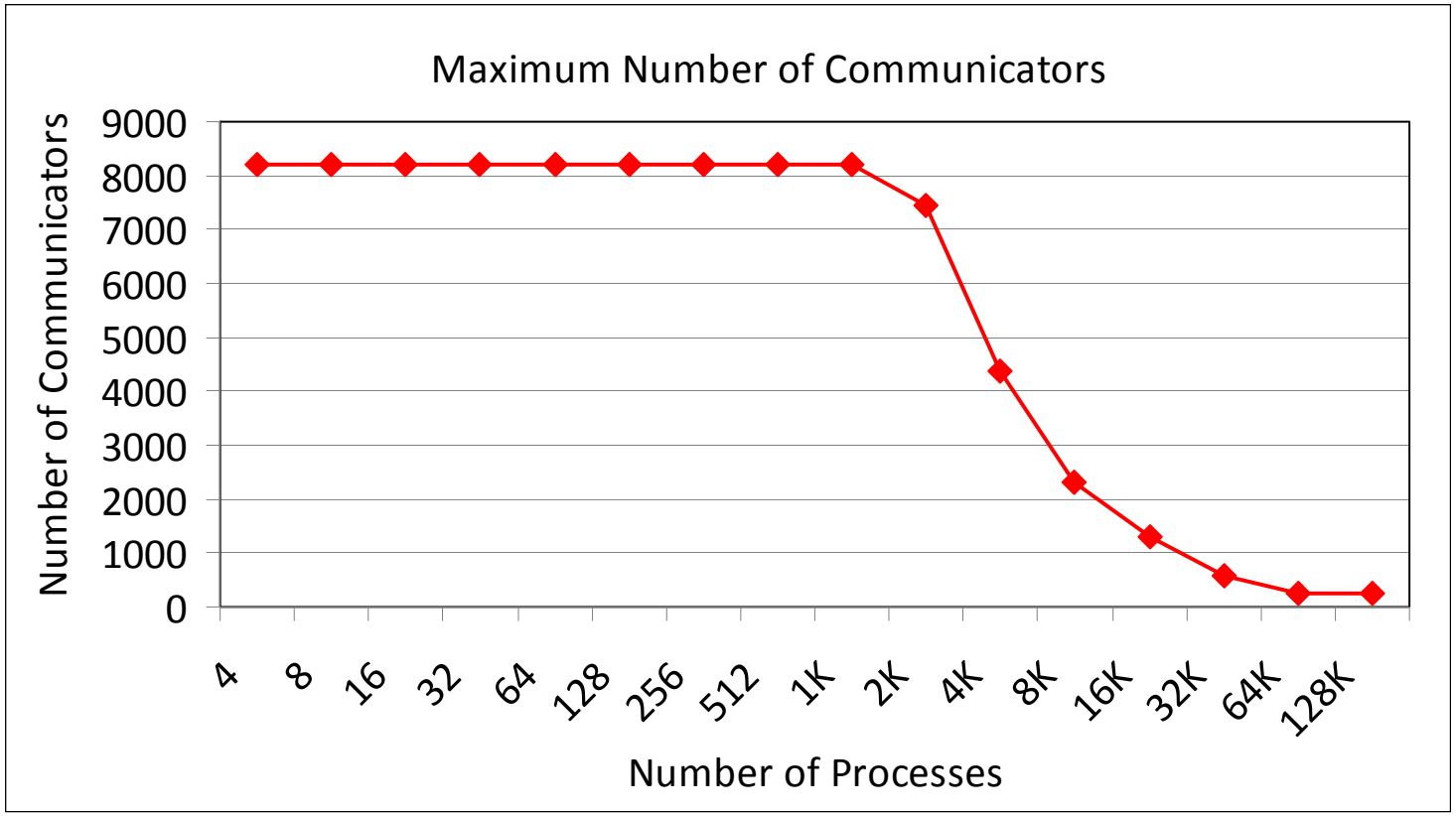
Communicator Memory Consumption

- NEK5000 is a well-known fluid dynamics code developed by Paul Fischer and colleagues at Argonne
- When they first tried to scale this code on the BG/P, it failed on as little as 8K processes because the MPI library ran out of communicator memory
- NEK5000 calls MPI_Comm_dup about 64 times (because it makes calls to libraries)
- 64 is not a large number, and, in any case, MPI_Comm_dup should not consume O(nprocs) memory (it doesn't in MPICH2)

- We ran an experiment to see what was going on…

Communicator Memory Consumption with original MPI on BG/P

- Run MPI_Comm_dup in a loop until it fails. Vary the no. of processes



What was going on --- and the fix

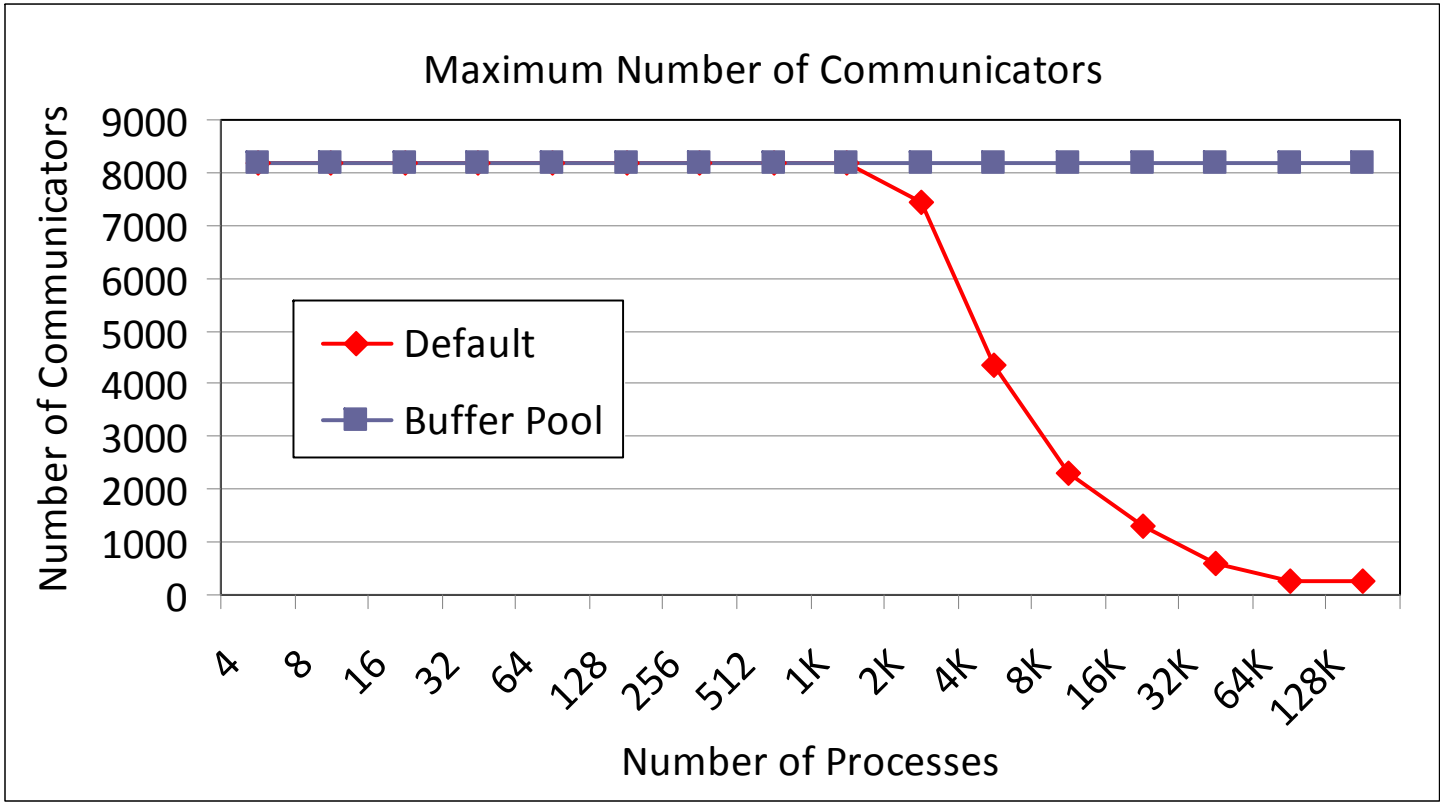
- The default MPI_Comm_dup in IBM's MPI was allocating memory to store process mapping info for optimizing future calls to collective communication (Alltoall)
- Allocated memory was growing linearly with system size
- One could disable the memory allocation with an environment variable, but that would also disable the collective optimizations

- On further investigation we found that they really only needed one buffer per thread instead of one buffer per new communicator

- Since there are only four threads on the BG/P, we fixed the problem by allocating a fixed buffer pool within MPI

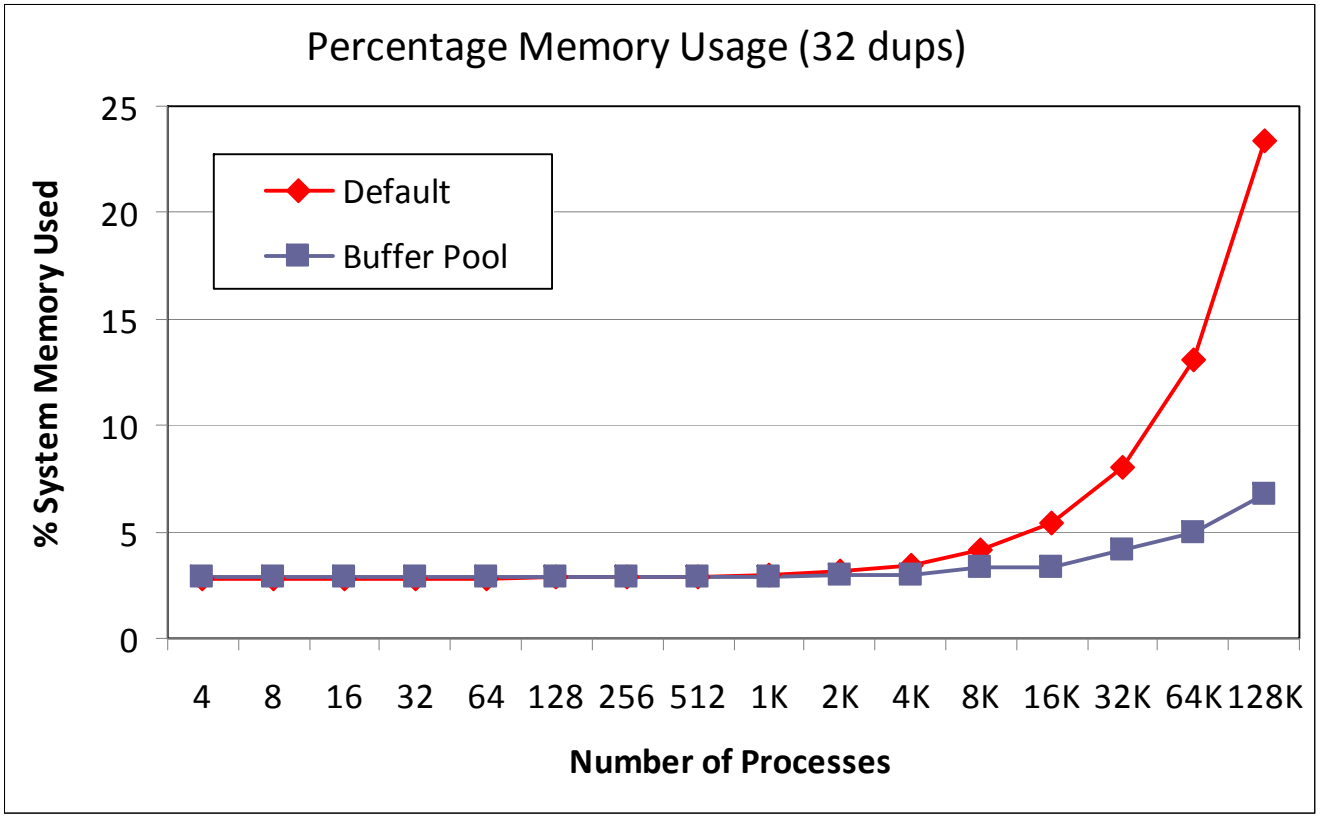
- We provided IBM with a patch that fixed the problem

Communicator Memory Consumption Fixed



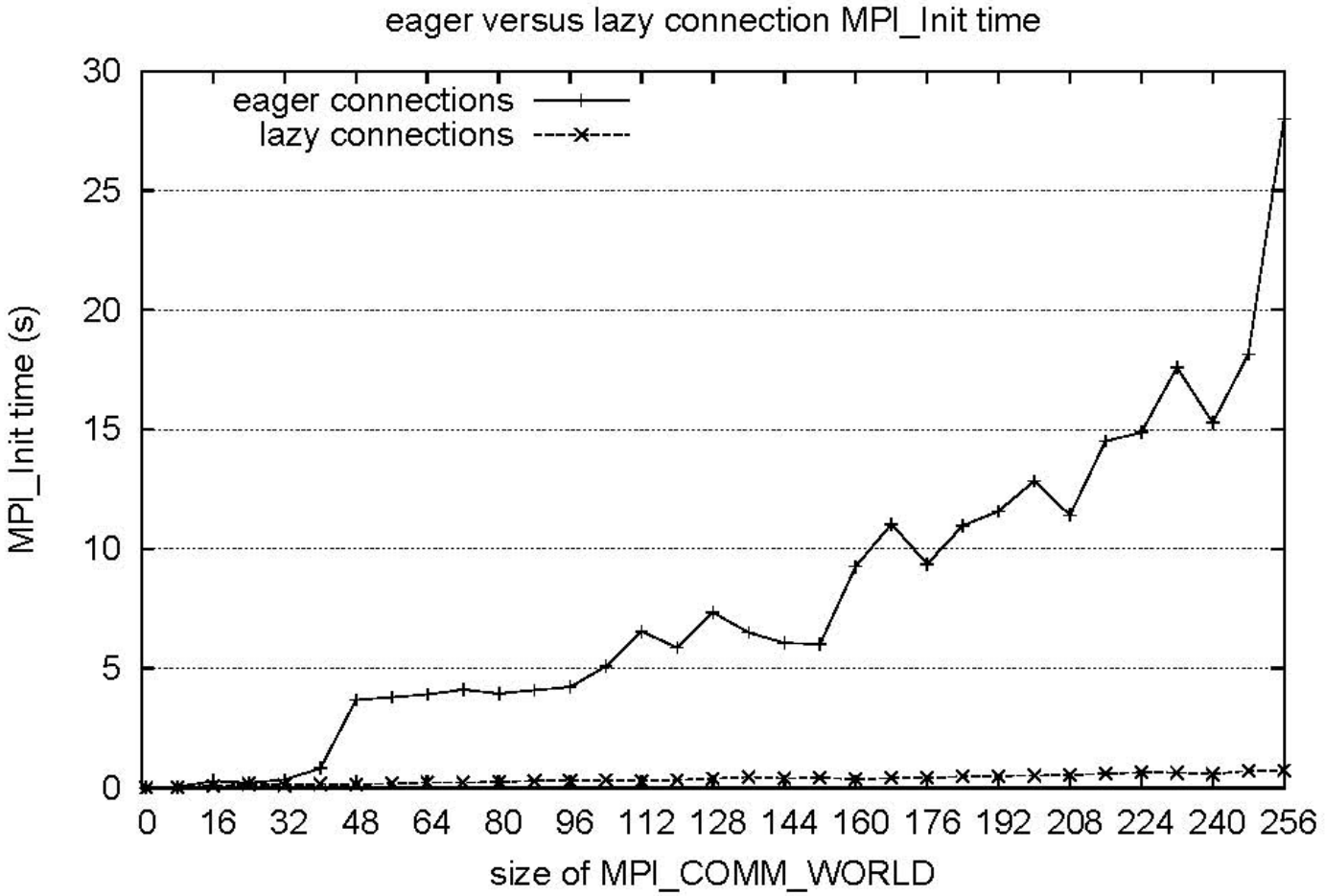
- NEK5000 code failed on BG/P at large scale because MPI ran out of communicator memory. We fixed the problem by using a fixed buffer pool within MPI and provided a patch to IBM.

MPI Memory Usage on BG/P after 32 calls to MPI_Comm_dup



- Using a buffer pool enables all collective optimizations and takes up only a small amount of memory

Scalability of MPI_Init



- Cluster with 8 cores per node. TCP/IP across nodes
- Setting up all connections at Init time is too expensive at large scale; must be done on demand as needed

Scalable Algorithms for Collective Communication

- MPI implementations typically use
  - O(lg p) algorithms for short messages (binomial tree)
  - O(m) algorithms, where m=message size, for large messages
    - E.g., bcast implemented as scatter + allgather
- O(lg p) algorithms can still be used on a million processors for short messages
- However, O(m) algorithms for large messages may not scale, as the message size in the allgather phase can get very small
  - E.g., for a 1 MB bcast on a million processes, the allgather phase involves 1 byte messages
- Hybrid algorithms that do logarithmic bcast to a subset of nodes, followed by scatter/allgather may be needed
- Topology-aware pipelined algorithms may be needed
- Use network hardware for broadcast/combine

Enabling Hybrid Programming

- MPI is good at moving data between address spaces
- Within an address space, MPI can interoperate with other “shared memory” programming models
- Useful on future machines that will have limited memory per core
- (MPI + X) Model: MPI across address spaces, X within an address space
- Examples:
  - MPI + OpenMP
  - MPI + UPC/CAF (here UPC/CAF address space could span multiple nodes)
  - MPI + CUDA/OpenCL on GPU-accelerated systems
- Precise thread-safety semantics of MPI enable such hybrid models

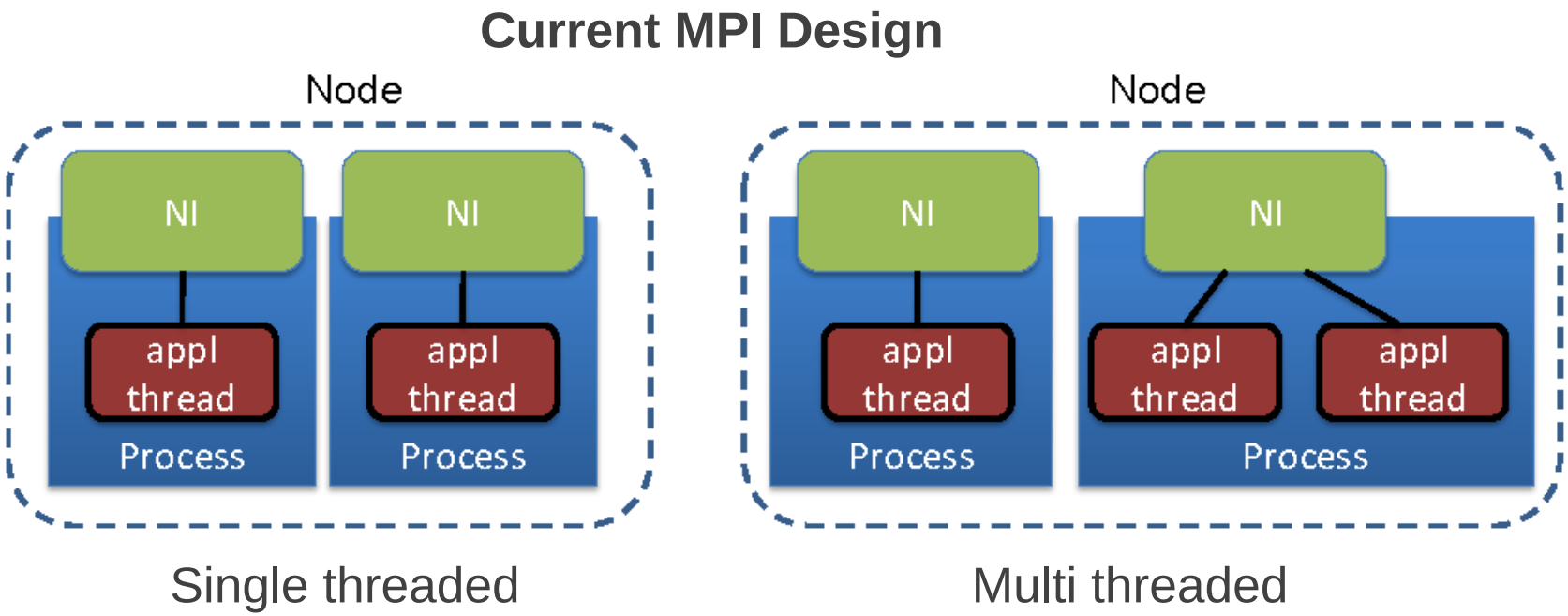
- MPI Forum is exploring further enhancements to MPI to support efficient hybrid programming

MPI-3 Hybrid Proposal on Endpoints

- In MPI today, each process has one communication endpoint (rank in MPI_COMM_WORLD)
- Multiple threads communicate through that one endpoint, requiring the implementation to do use locks etc., which are expensive
- This proposal (originally by Marc Snir) allows a process to have multiple endpoints
- Threads within a process attach to different endpoints and communicate through those endpoints as if they are separate ranks
- The MPI implementation can avoid using locks if each thread communicates on a separate endpoint

MPI-3 Hybrid Proposal on Endpoints

- Today, each MPI process has one communication endpoint (rank in MPI_COMM_WORLD)
- Multiple threads communicate through that one endpoint, requiring the implementation to do use locks etc. (expensive)

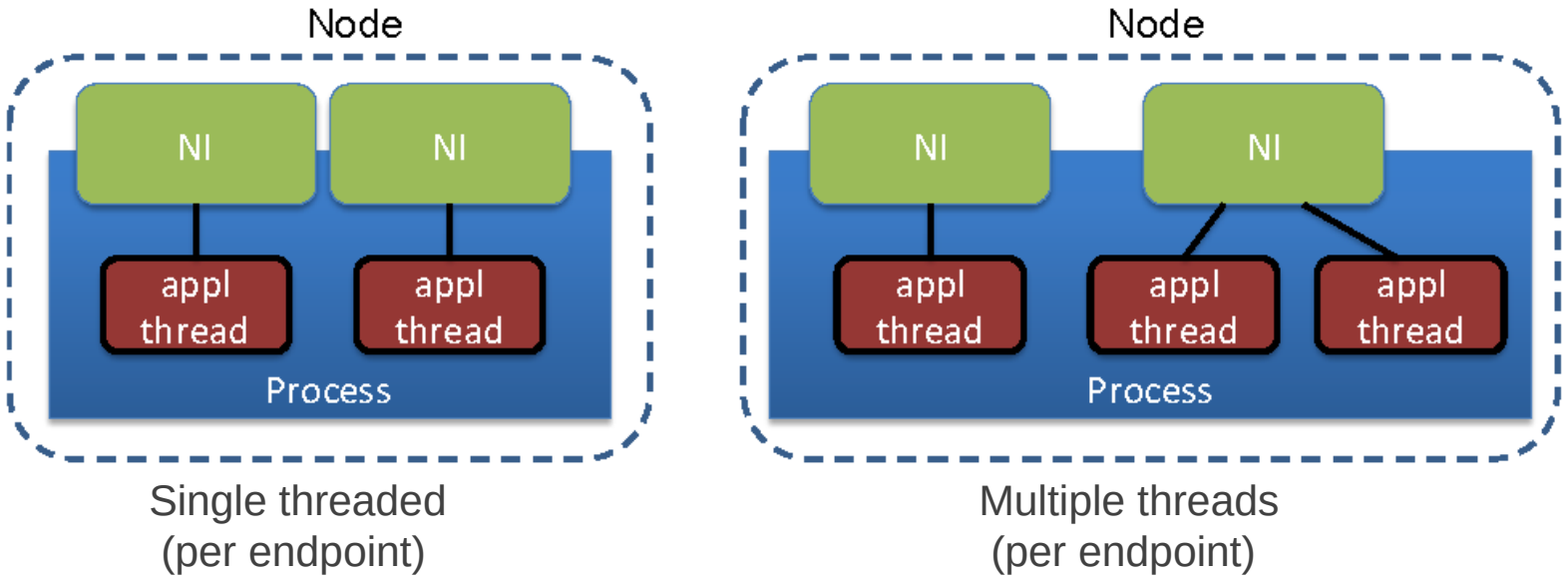


Separate address spaces for each endpoint

MPI-3 Hybrid Proposal on Endpoints

- The proposal is to allow a process to have multiple endpoints
- Threads within a process attach to different endpoints and communicate through those endpoints as if they are separate ranks
- The MPI implementation can avoid using locks if each thread communicates on a separate endpoint

Proposed MPI Design



Single threaded (per endpoint)

Multiple threads (per endpoint)

Multiple endpoints are mapped in the same address space

Fewer Performance Surprises

- Sometimes we hear…

"I replaced

MPI_Allreduce

by

MPI_Reduce + MPI_Bcast

And got better results…"

Should not happen…

Or…

"I replaced

MPI_Send(n)

by

MPI_Send(n/k) + MPI_Send(n/k) + … + MPI_Send(n/k)

And got better results…"

Well, should probably not happen…

Or…

"I replaced

MPI_Bcast(n)

by

<this homemade algorithm with MPI_Send(n) and MPI_Recv(n)>

And got better results…"

Should not happen…

Self-Consistent MPI Performance Guidelines

- Although MPI is portable, there is a lot of performance variability among MPI implementations
  - Lots of performance surprises
- We (Traff, Gropp, Thakur) have defined some common-sense performance guidelines for MPI
  - *"Self-Consistent MPI Performance Guidelines", IEEE TPDS, 2010*
- Tools could be written to check for these requirements

General Principles

If there is an obvious way – intended by the MPI standard – of improving communication time,

a sound MPI implementation should do so!

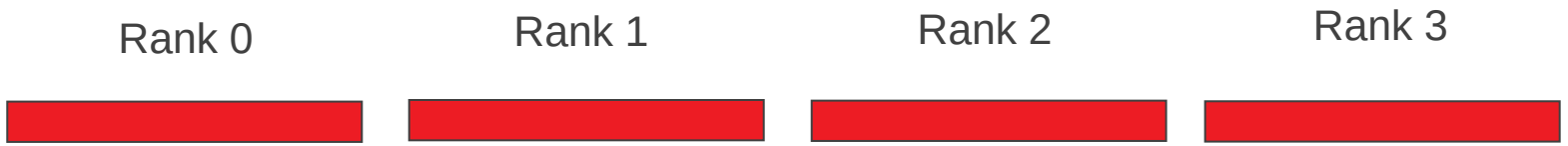
- And not the user!

Sample Requirements

- Subdividing messages into multiple messages should not reduce the communication time
  - MPI_Send(1500 bytes) <= MPI_Send(750 bytes) + MPI_Send(750 bytes)
- Replacing an MPI function with a similar function that provides additional semantic guarantees should not reduce the communication time
  - MPI_Send <= MPI_Ssend
- Replacing a specific MPI operation by a more general operation by which the same functionality can be expressed should not reduce communication time
  - MPI_Scatter <= MPI_Bcast

Example: Broadcast vs Scatter

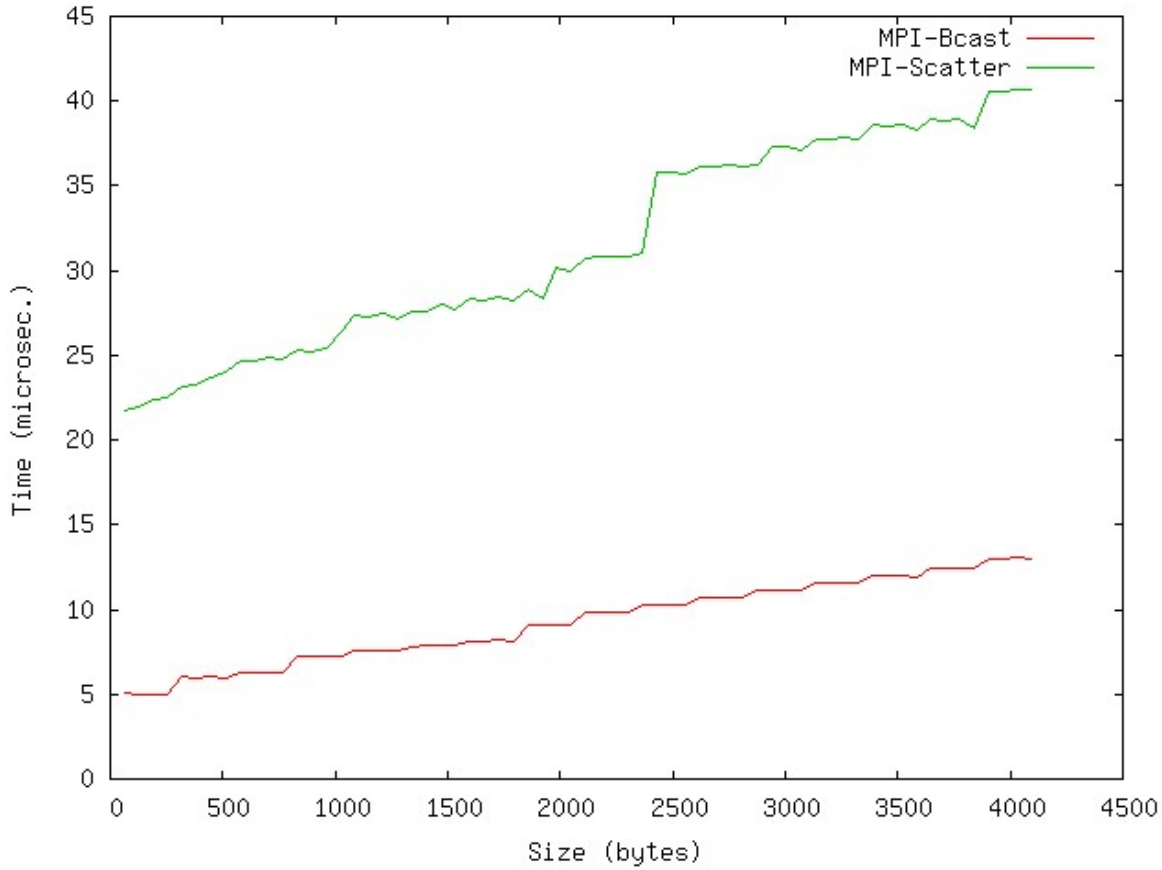
Broadcast



Scatter

- Scatter should be faster (or at least no slower) than broadcast

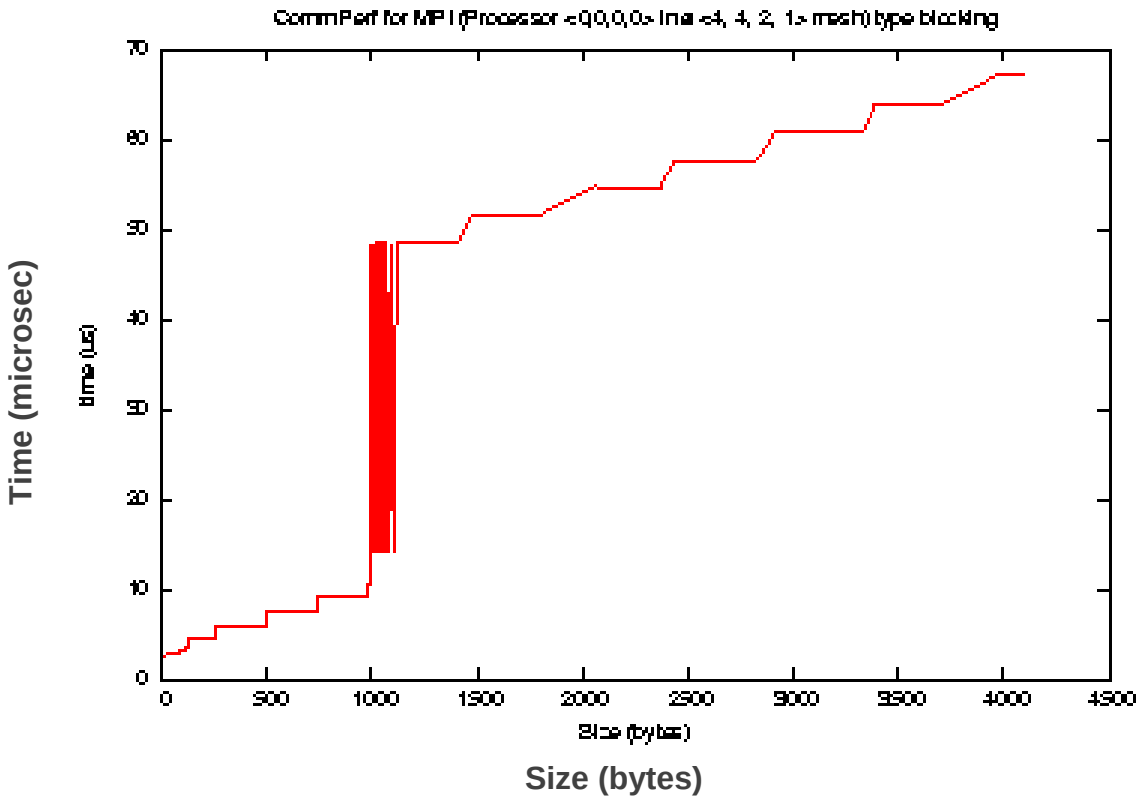
MPI_Bcast vs MPI_Scatter



64 processes

- On BG/P, scatter is 3-4 times slower than broadcast
- Broadcast has been optimized using hardware, scatter hasn't

Eager vs Rendezvous Messages



- Large jump in time when message delivery switches from eager to rendezvous
- Sending 2 750-byte messages is faster than 1 1500-byte message

Recent Efforts of the MPI Forum

MPI Standard Timeline

- MPI-1 (1994)
  - Basic point-to-point communication, collectives, datatypes, etc
- MPI-2 (1997)
  - Added parallel I/O, RMA, dynamic processes, C++ bindings, etc

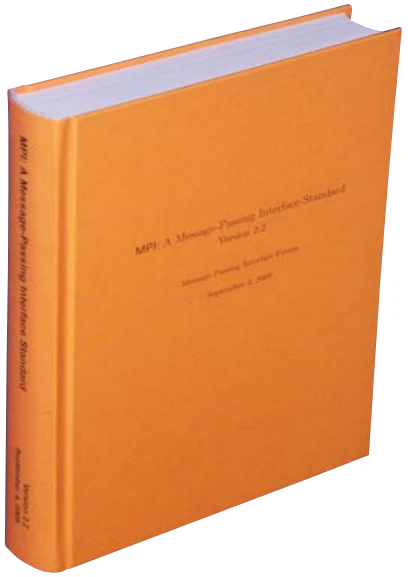
- ---- Stable for 10 years ----

- MPI-2.1 (2008)
  - Minor clarifications and bug fixes to MPI-2
- MPI-2.2 (2009)
  - Today's official standard
  - Small updates and additions to MPI 2.1. Backward compatible
- MPI-3 (in progress, expected late 2011)
  - Major new features and additions to extend MPI to exascale
  - Organized into several working groups

MPI 2.2 (Today's Official MPI Standard)

- Led by Bill Gropp
- Officially approved by the MPI Forum at the Sept 2009 meeting
- Small updates to the standard
  - Does not break backward compatibility
- Spec can be downloaded from the MPI Forum web site

  www.mpi-forum.org
- Also available for purchase as a book from
  https://fs.hlrs.de/projects/par/mpi/mpi22/
- Supported by MPICH2 1.2



New Features in MPI 2.2

- Scalable graph topology interface
  - Existing interface requires the entire graph to be specified on all processes, which requires too much memory on large numbers of processes
  - New functions allow the graph to be specified in a distributed fashion (MPI_Dist_graph_create, MPI_Dist_graph_create_adjacent)
- A local reduction function
  - MPI_Reduce_local(inbuf, inoutbuf, count, datatype, op)
  - Needed for libraries to implement user-defined reductions
- MPI_Comm_create extended to enable creation of multiple disjoint communicators
- Regular (non-vector) version of MPI_Reduce_scatter called MPI_Reduce_scatter_block

New Features in MPI 2.2

- MPI_IN_PLACE option added to MPI_Alltoall, Alltoallv, Alltoallw, and Exscan
- The restriction on the user not being allowed to access the contents of the buffer passed to MPI_Isend before the send is completed by a test or wait has been lifted
- New C99 datatypes (MPI_INT32_T, MPI_C_DOUBLE_COMPLEX, etc) and MPI_AINT/ MPI_OFFSET

New Features being considered in MPI-3

- **Note: All these are still under discussion in the Forum and not final**

- Support for hybrid programming (Lead: Pavan Balaji, Argonne)
  - Extend MPI to allow multiple communication endpoints per process
  - Helper threads: application sharing threads with the implementation

- Improved RMA (Leads: Bill Gropp, UIUC, and Rajeev Thakur, Argonne)
  - Fix the limitations of MPI-2 RMA
  - New compare-and-swap, fetch-and-add functions
  - Collective window memory allocation
  - Test for completion of individual operations
  - Others…

New Features being considered in MPI-3

- New collectives (Lead: Torsten Hoefler, UIUC)
  - Nonblocking collectives already voted in (MPI_Ibcast, MPI_Ireduce, etc)
  - Sparse, neighborhood collectives being considered as alternatives to irregular collectives that take vector arguments
- Fault tolerance (Lead: Rich Graham, Oak Ridge)
  - Detecting when a process has failed; agreeing that a process has failed
  - Rebuilding communicator when a process fails or allowing it to continue in a degraded state
  - Timeouts for dynamic processes (connect-accept)
  - Piggybacking messages to enable application-level fault tolerance

New Features being considered in MPI-3

- Fortran 2008 bindings (Lead: Craig Rasmussen, LANL)
  - Full and better quality argument checking with individual handles
  - Support for choice arguments, similar to (void *) in C
  - Passing array subsections to nonblocking functions
  - Many other issues

- Better support for Tools (Lead: Martin Schulz, LLNL)
  - MPIT performance interface to query performance information internal to an implementation
  - Standardizing an interface for parallel debuggers

MPI Forum Mailing Lists and Archives

- Web site: http://lists.mpi-forum.org/

- Lists
  - mpi-forum
  - mpi-22, mpi-3
  - mpi3-coll
  - mpi3-rma
  - mpi3-ft
  - mpi3-fortran
  - mpi3-tools
  - mpi3-hybridpm

- Further info: http://meetings.mpi-forum.org/
- Wiki: https://svn.mpi-forum.org/trac/mpi-forum-web/wiki

What are we doing in MPICH2

Goals of the MPICH2 project

- Be the MPI implementation of choice for the highest-end parallel machines
  - 7 of the top 10 machines in the June 2010 Top500 list use MPICH2-based implementations
- Carry out the research and development needed to scale MPI to exascale
  - Optimizations to reduce memory consumption
  - Fault tolerance
  - Efficient multithreaded support for hybrid programming
  - Performance scalability
- Work with the MPI Forum on standardization and early prototyping of new features

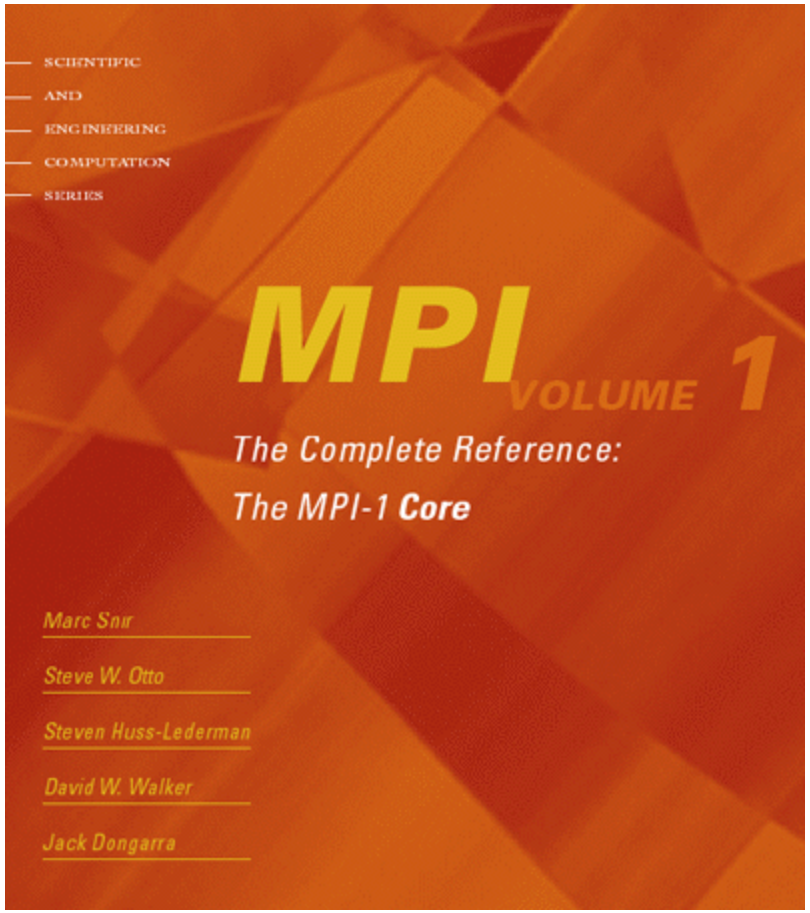
MPICH2 collaboration with vendors

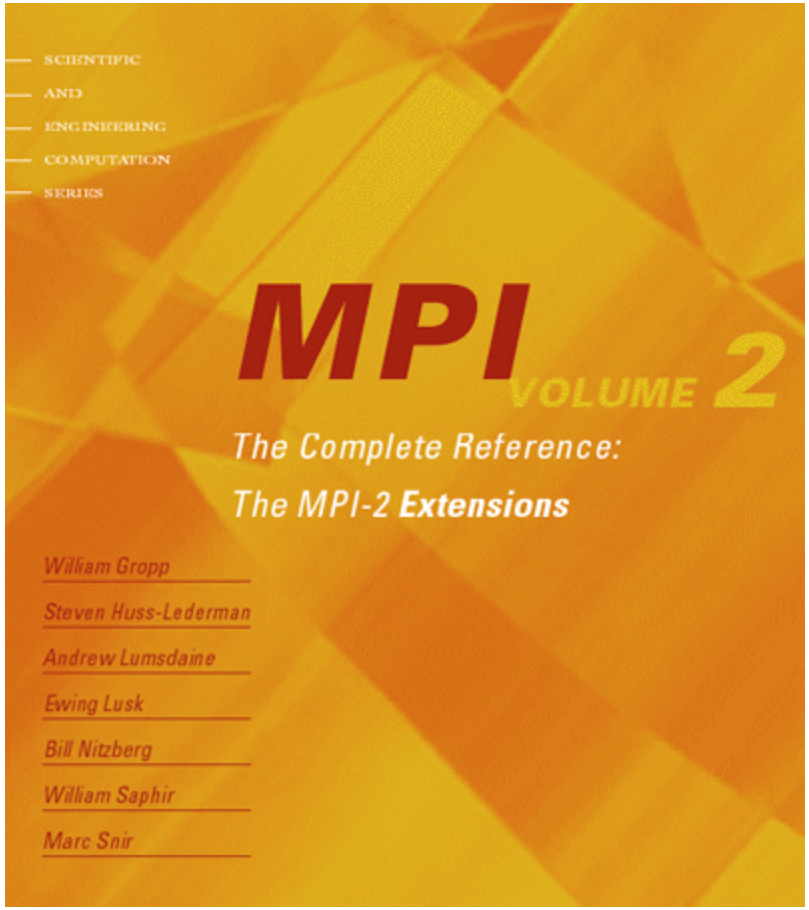
- Enable vendors to provide high-performance MPI implementations on the leading machines of the future
- Collaboration with IBM on MPI for the Blue Gene/Q
  - Aggressive multithreaded optimizations for high concurrent message rates
  - Recent publications in Cluster 2010 and EuroMPI 2010
- Collaboration with Cray for MPI on their next-generation interconnect (Gemini)
- Collaboration with UIUC on MPICH2 over LAPI for Blue Waters
- Continued collaboration with Intel, Microsoft, and Ohio State (MVAPICH)

Conclusions

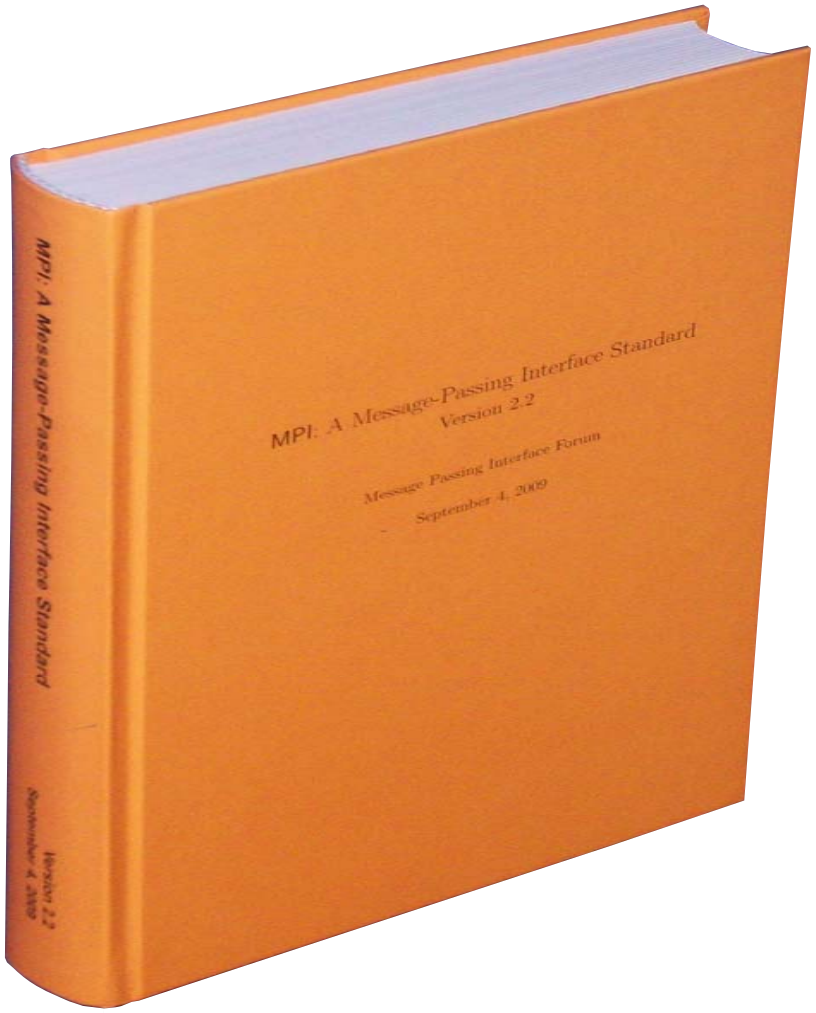
- MPI has succeeded because
  - features are orthogonal (complexity is the product of the number of *features*, not routines)
  - complex programs are no harder than easy ones
  - open process for defining MPI led to a solid design
  - programmer can control memory motion and program for locality (critical in high-performance computing)
  - precise thread-safety specification has enabled hybrid programming
- MPI is ready for scaling to extreme scale systems with millions of cores barring a few issues that can be (and are being) fixed by the MPI Forum and by MPI implementations

The MPI Standard (1 & 2)

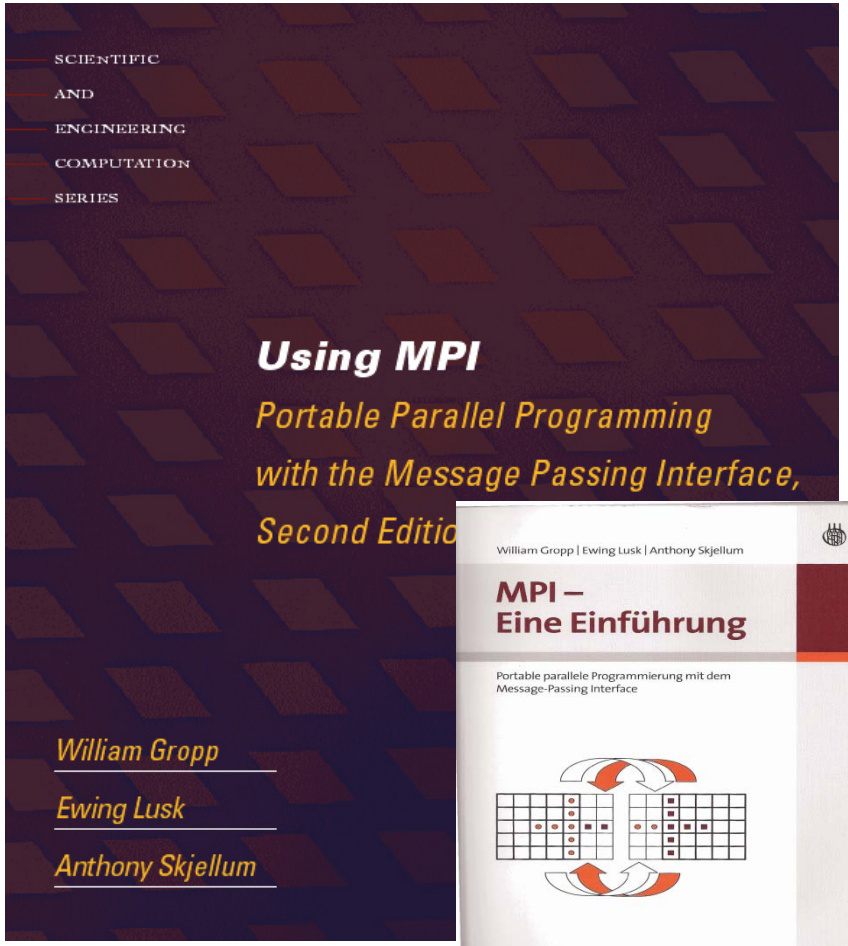


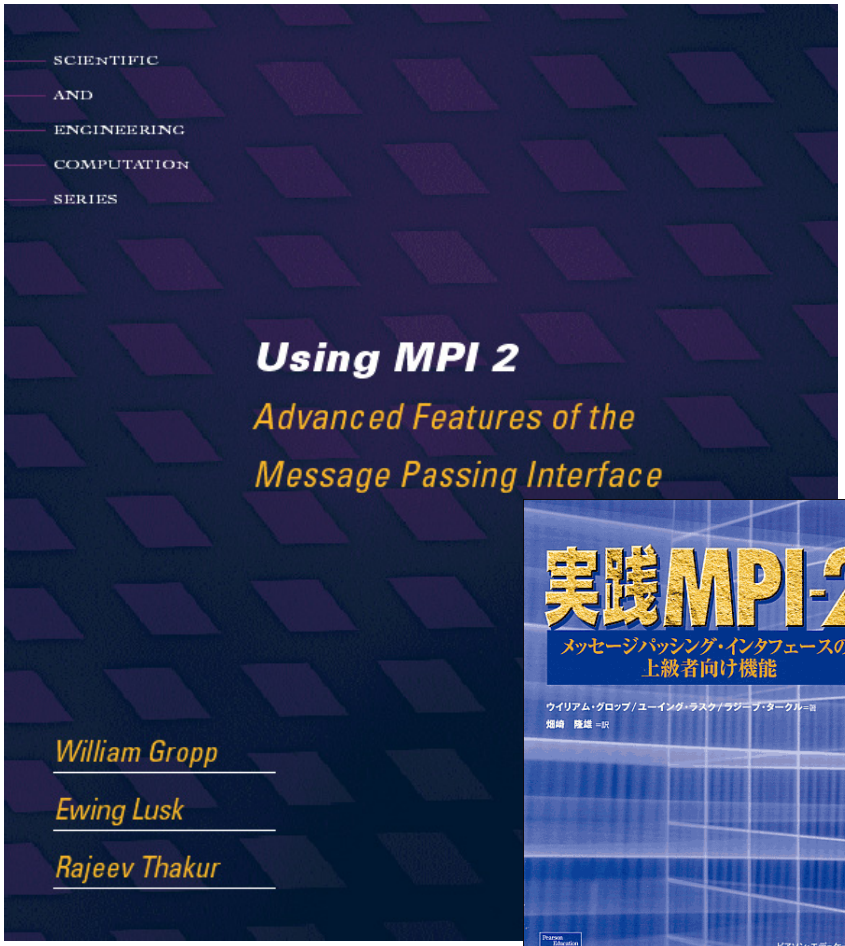


MPI 2.2 Standard



Tutorial Material on MPI, MPI-2





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