Argonne NATIONAL LABORATORY

... for a brighter future

Programming in MPI for Performance

William Gropp and Rusty Lusk

U.S. Department of Energy

UChicago Argonne LLC

Office of Science
U.S. DEPARTMENT OF ENERGY

A U.S. Department of Energy laboratory managed by UChicago Argonne, LLC

Outline

- Background: us, status of machines, near-term schedules
- Software environment on Leadership Class machines in which we seek performance
  - MPI and MPI-2
  - Threads
- Some experimental data on LC machines and others
  - Halo exchanges
  - Topologies
  - One-sided operations
  - MPI and threads
- Selected tools enabled by MPI profiling interface
  - SLOG/Jumpshot: visualizing parallel performance
  - FPMPI: gathering summary statistics
  - Collchk: runtime checking of correct use of collective operations

Our Status

- Full disclosure: we are getting started ourselves on investigating the environments on the Leadership Class (LC) machines
  - At Argonne we currently have a small (2048 cores) BG/L
    - *Many applications ported*
    - *Occasional access to larger (20 rack) BG/L at IBM Watson*
  - We have remote access to a BG/P at IBM
    - *BG/P is a lot like BG/L, but better*
  - We have access to ORNL's XT3 and XT4
  - We have tried some things related to MPI performance (which we will talk about here); we haven't tried everything
  - Experimentation is ongoing
- We are in the process of transferring considerable experience on other environments to the LC machines.

The Application Programming Environment on the Leadership Class Machines

- Underlying common abstraction: multiple cores per node, many nodes, fast network
- MPI-1
  - Can run one MPI process per CPU (virtual node mode)
    - *Many MPI processes available*
    - *Challenge: algorithm scaling*
  - Can run one MPI process per node
    - *More memory per MPI process*
    - *More flops per process if can use threads to access multiple cpus per process*
    - *MPI implementations are "thread-safe" (MPI_THREAD_MULTIPLE) except on BG/L*

Programming Environment (cont.)

- MPI-2
  - MPI one-sided operations (MPI_Put, MPI_Get, and friends) are available on BG/P, XT4, not BG/L
    - *Performance is still a question*
  - MPI-IO available on all machines, implemented on different parallel file systems
    - *Lustre on XT3 and XT4 at Oak Ridge, GPFS on XT4 at NERSC*
    - *PVFS on BG/P*
  - MPI dynamic process management (MPI_Comm_spawn and friends) not available on any of the LC machines

Programming Environment (cont.)

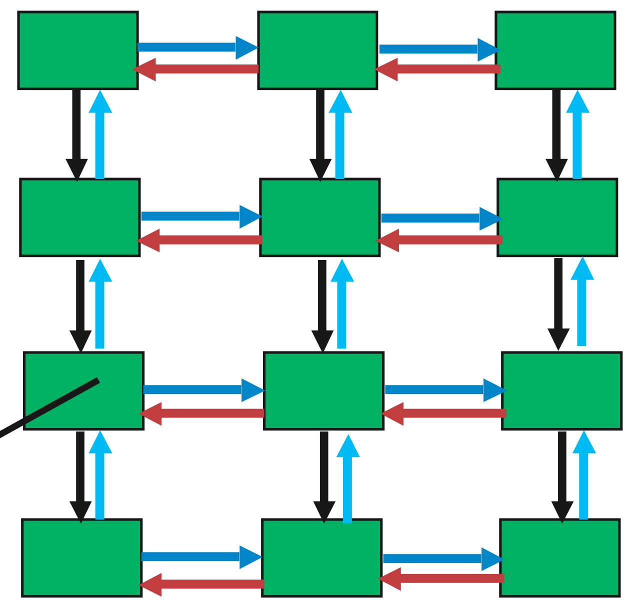
- Threads
  - All machines have multicore nodes, allowing local shared memory model
  - On BG/L, MPI implementation is not thread-safe; nodes are not cache-coherent
  - On BG/P, exactly 4 threads per node in virtual node mode
  - Similarly on XT4 (2 per node now, eventually 4)
- Programming with threads
  - OpenMP compilers
  - Pthreads library
- Languages
  - C, C++, Fortran-90
  - Co-Array Fortran and UPC
  - OpenMP (both C and Fortran versions)
    - *Forthcoming book by Barbara Chapman et al.*

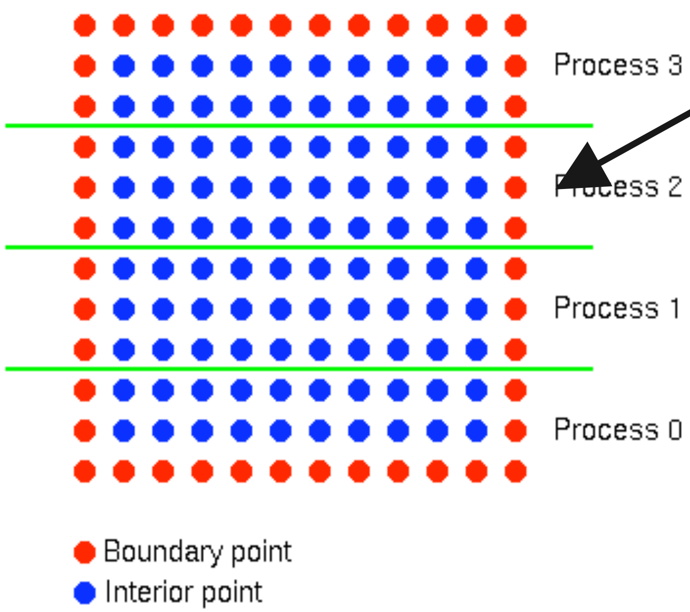
Basic MPI: Looking Closely at a Simple Communication Pattern

- 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



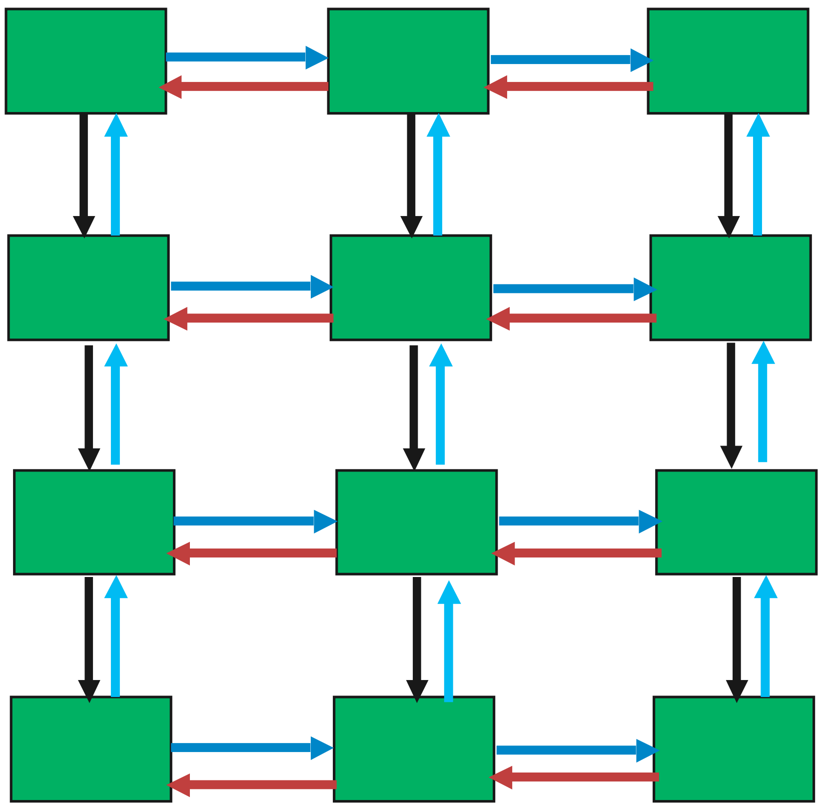


Scalability of Mesh Exchange

- How does the computational effort and communication change as the task size changes?
  - Classic example is mesh exchange
- Data exchanged is the “surface” of the mesh patch; computation is on the “volume”
  - Important term is the surface to volume ratio
  - Cost of surface exchanges (3-d domain, faces only):
    - *1-d = 2 (s + r n2)*
    - *2-d = 4 (s + r n2/√p)*
    - *3-d = 6 (s + r n/p1/3)*
  - Best approach is to make these relative to floating-point work (this is the dimensionless quantity):
    - *1-d = 2(s + r n2) / n3f*
- These assume that communications are non-interfering. Simple mistakes can violate that assumption…

Mesh Exchange

- Exchange data on a mesh



Sample Code

- Do i=1,n_neighbors
```
  Call MPI_Send(edge(1,i), len, MPI_REAL,&
                    nbr(i), tag,comm, ierr)
Enddo
Do i=1,n_neighbors
  Call MPI_Recv(edge(1,i), len, MPI_REAL,&
               nbr(i), tag, comm, status, ierr)
Enddo
```

Deadlocks!

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

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

  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

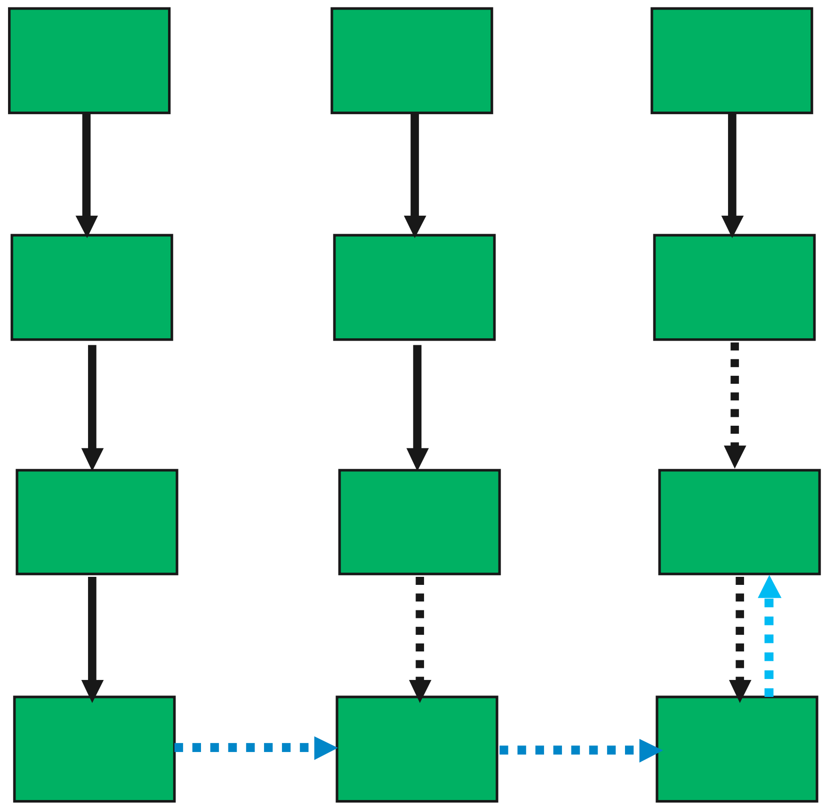
- Do i=1,n_neighbors
  Call MPI_Irecv(inedge(1,i), len, MPI_REAL, nbr(i), tag,&
  comm, requests(i), ierr)
  Enddo
  Do i=1,n_neighbors
  Call MPI_Send(edge(1,i), len, MPI_REAL, nbr(i), tag,&
  comm, ierr)
  Enddo
  Call MPI_Waitall(n_neighbors, requests, statuses, ierr)
- Does not perform well in practice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

Understanding the Behavior: 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)

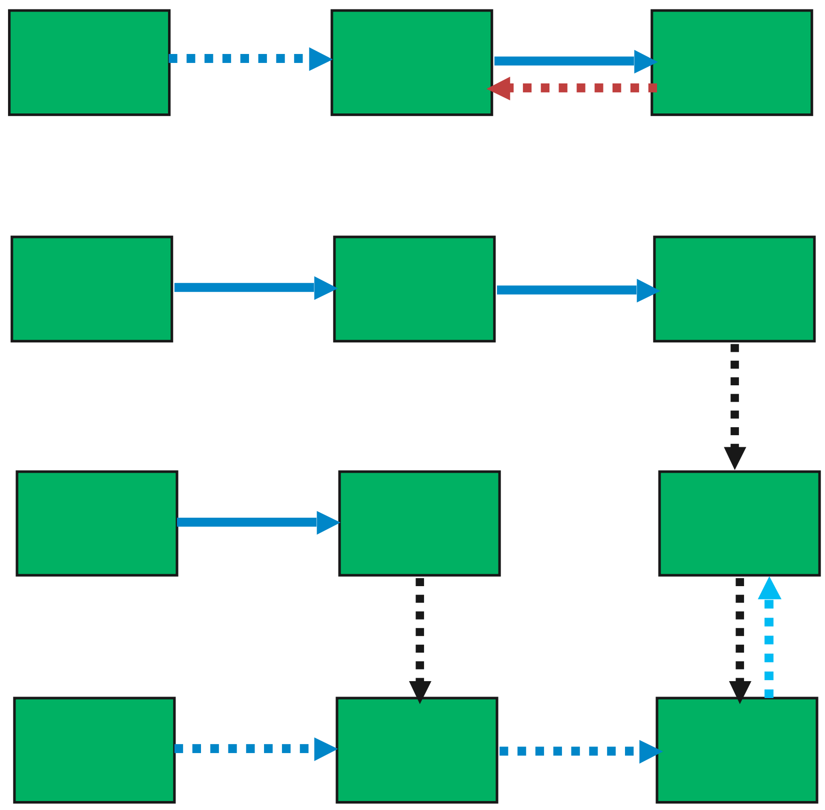
Mesh Exchange - Step 1

- Exchange data on a mesh



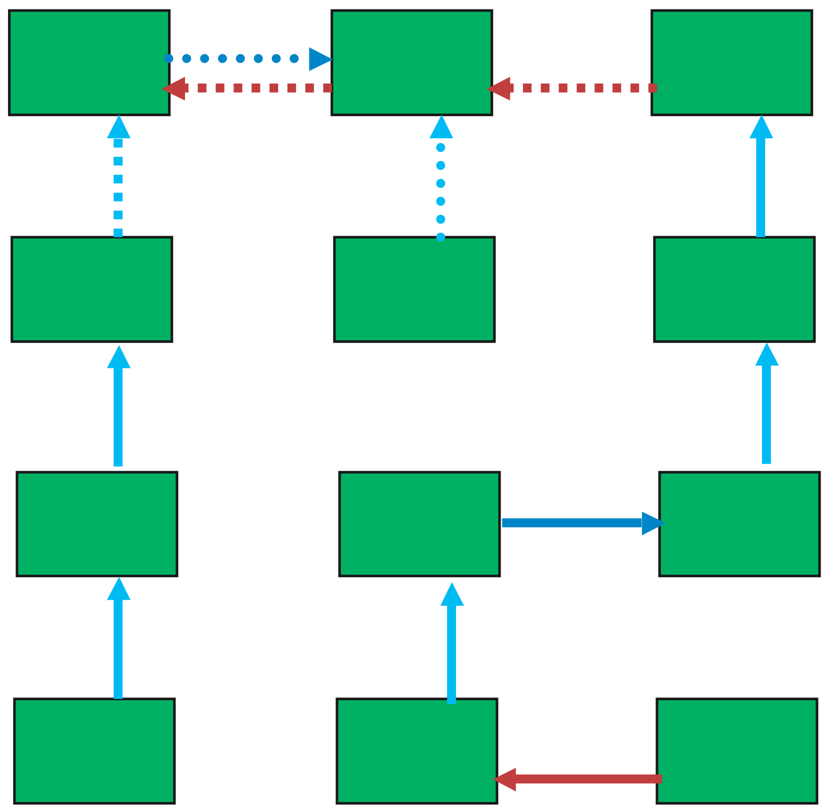
Mesh Exchange - Step 2

- Exchange data on a mesh



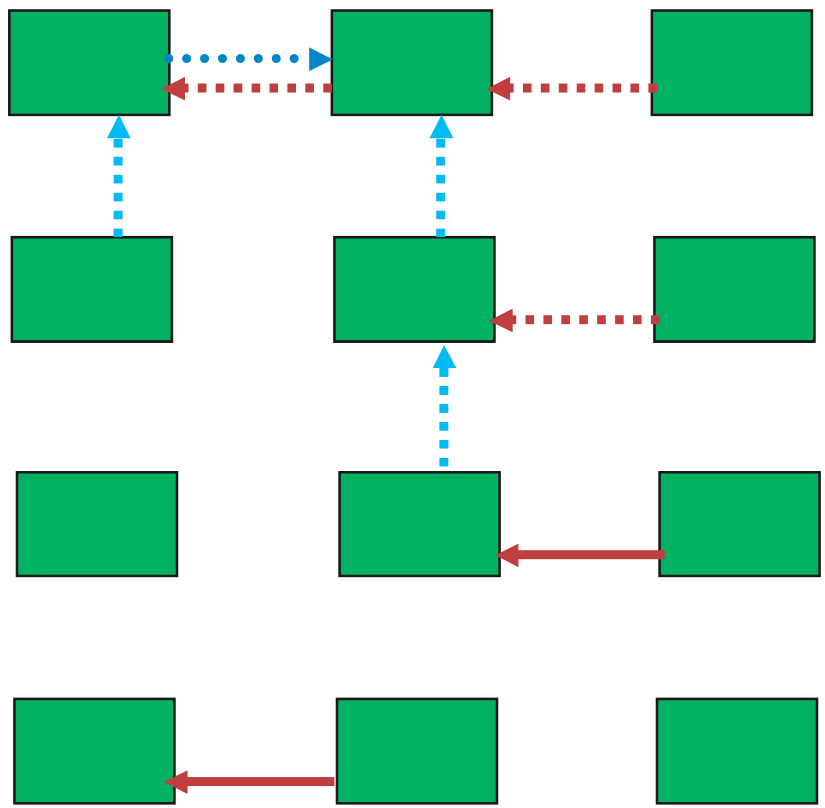
Mesh Exchange - Step 3

- Exchange data on a mesh



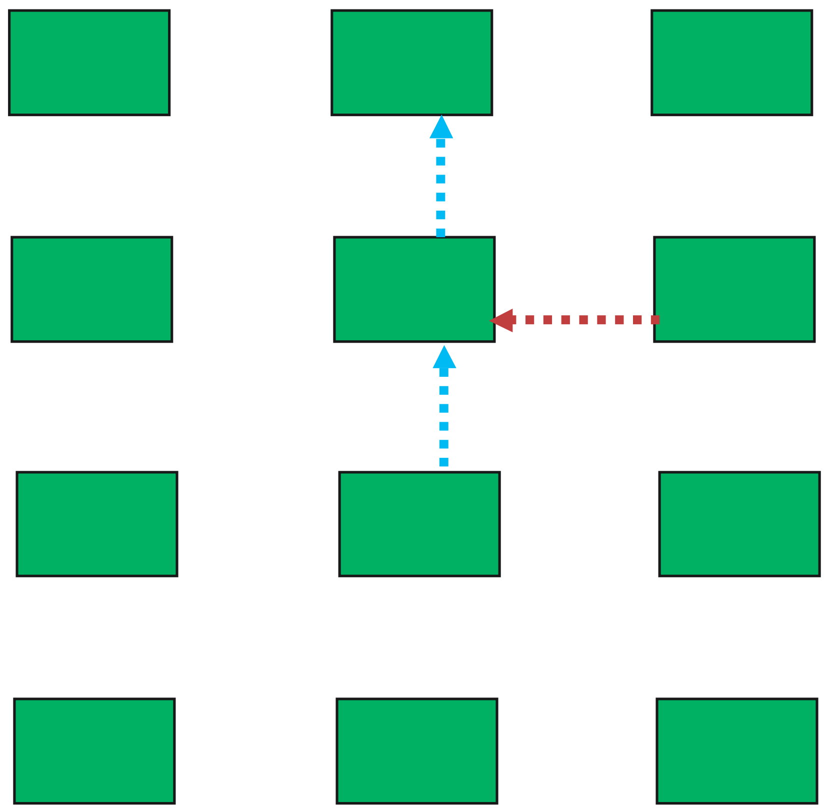
Mesh Exchange - Step 4

- Exchange data on a mesh



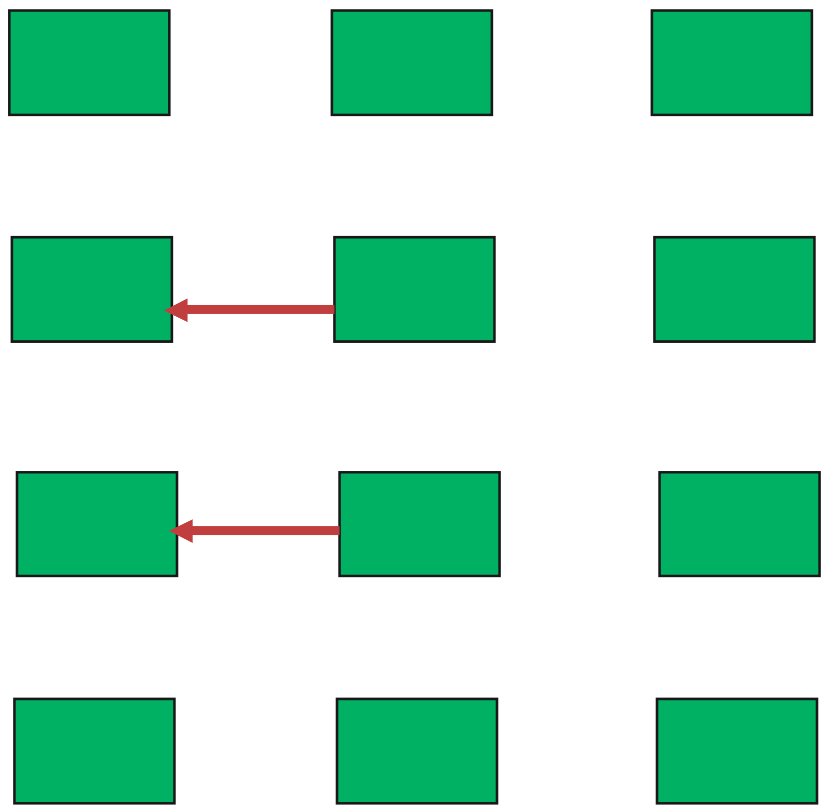
Mesh Exchange - Step 5

- Exchange data on a mesh

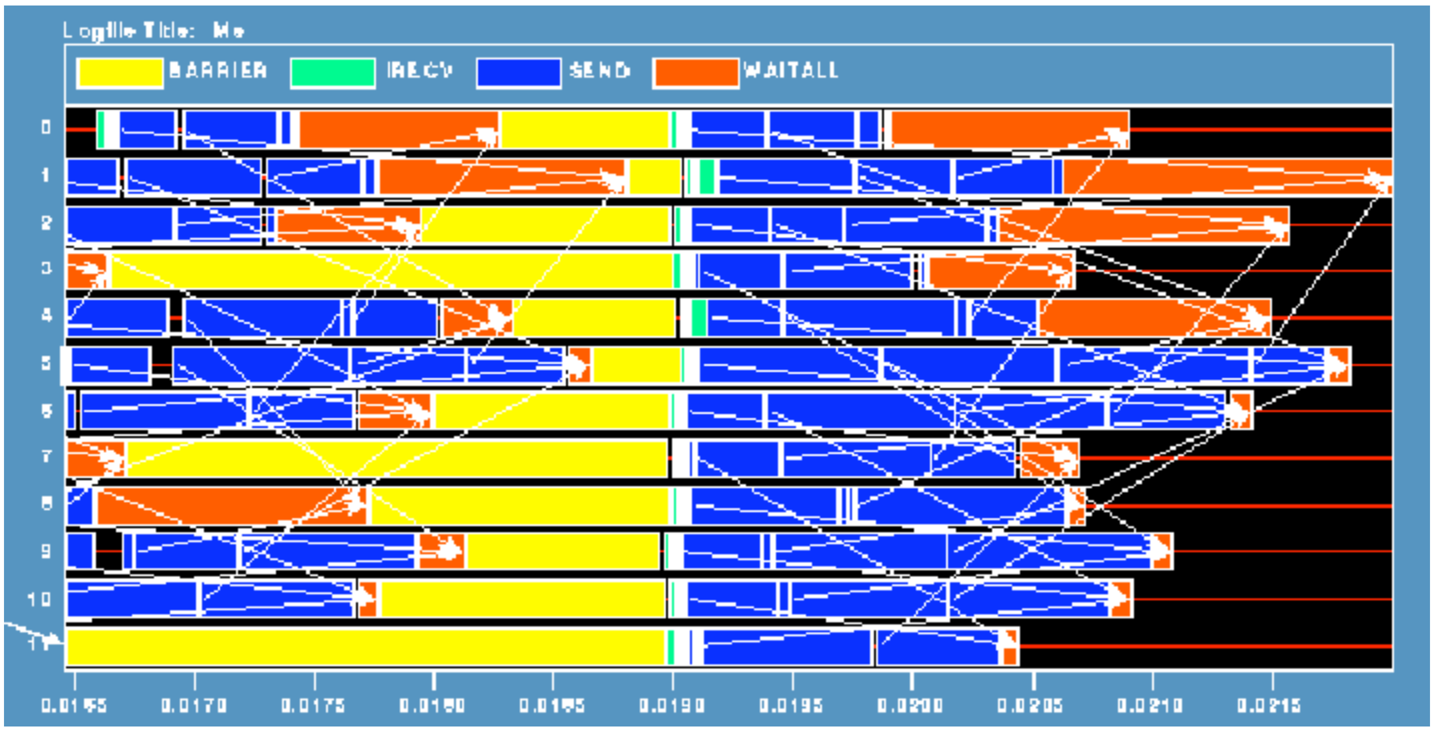


Mesh Exchange - Step 6

- Exchange data on a mesh

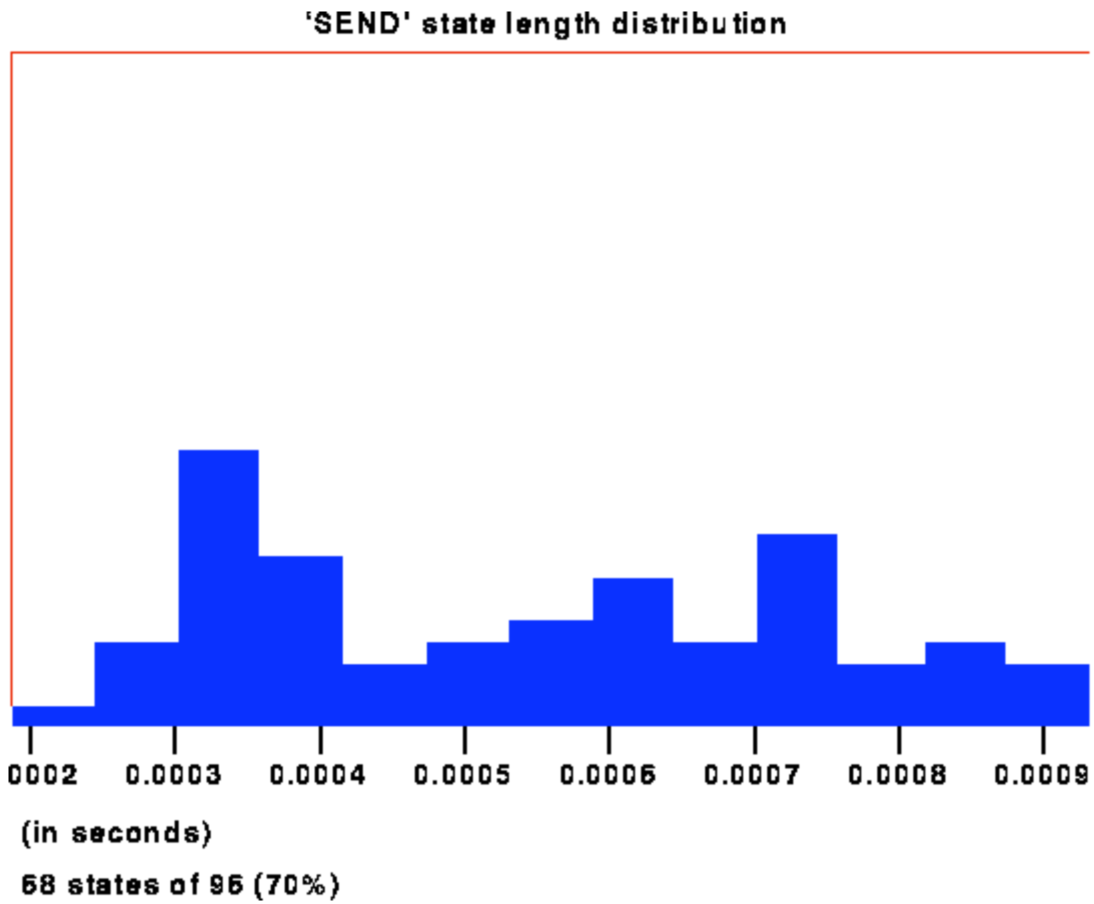


Timeline from IBM SP



- Note that process 1 finishes last, as predicted

Distribution of Sends



Why Six Steps?

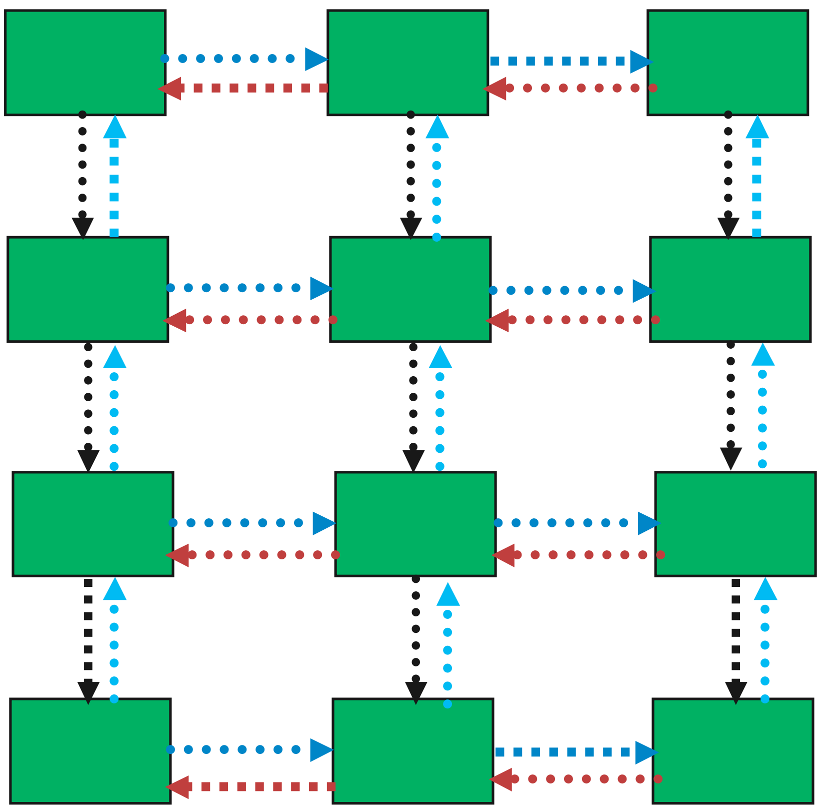
- Ordering of Sends introduces delays when there is contention at the receiver
- Takes roughly twice as long as it should
- Bandwidth is being wasted
- Same thing would happen if using memcpy and shared memory
- The interference of communication is why adding an MPI_Barrier (normally an unnecessary operation that reduces performance) can occasionally *increase* performance. But don't add MPI_Barrier to your code, please :)

Fix 2: Use Isend and Irecv

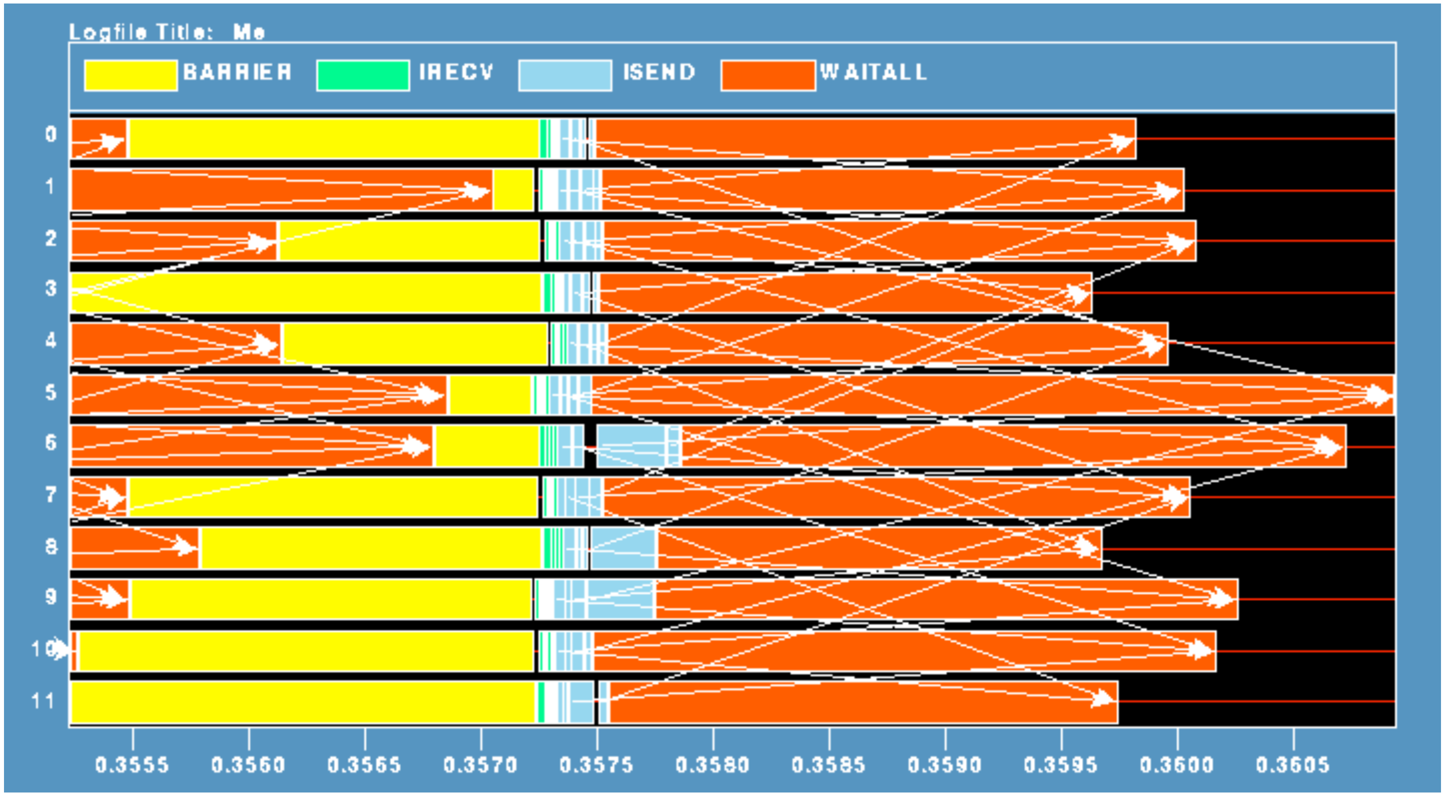
```
Do i=1,n_neighbors
   Call MPI_Irecv(inedge(1,i),len,MPI_REAL,nbr(i),tag,&
                  comm, requests(i),ierr)
Enddo
Do i=1,n_neighbors
   Call MPI_Isend(edge(1,i), len, MPI_REAL, nbr(i), tag,&
                  comm, requests(n_neighbors+i), ierr)
Enddo
Call MPI_Waitall(2*n_neighbors, requests,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

Mesh Exchange - Steps 1-4

- Four interleaved steps (at least, in principle)



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
- However, this relies on the MPI implementation taking advantage of the opportunities provided by MPI_Waitall (more on this later)

Using MPI For Process Placement

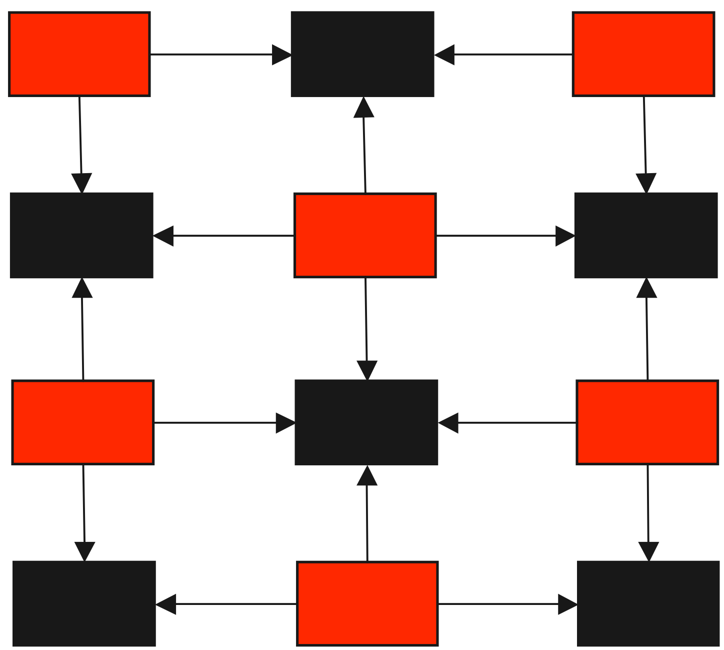
- MPI provides “process topology” routines to create a new communicator with a “better” layout
- When using a regular grid, consider using these routines:
  int dims[2], periodic[2];
  for (i=0; i<2; i++) { dims[i] = 0; periodic[i] = 0; }
  MPI_Dims_create(size, 2, dims);
  MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periodic, 1, &newComm);
- The “1” tells MPI_Cart_create to reorder the mapping of processes to create a “better” communicator for neighbor communication.
- Use newComm instead of MPI_COMM_WORLD in neighbor communication
- There’s also an MPI_Graph_create, but it isn’t very useful (too general). You can use MPI_Comm_split to create your very own reordering.

Experiments with Topology and Halo Communication on LC Machines

- The following slides show some results for a simple halo exchange program (halocompare) that tries several MPI-1 approaches and several different communicators:
  - MPI_COMM_WORLD
  - Dup of MPI_COMM_WORLD
    - *Is MPI_COMM_WORLD special in terms of performance?*
  - Reordered communicator - all even ranks in MPI_COMM_WORLD first, then the odd ranks
    - *Is ordering of processes important?*
  - Communicator from MPI_Dims_create/MPI_Cart_create
    - *Does MPI Implementation support these, and do they help*
- Communication choices are
  - Send/Irecv
  - Isend/Irecv
  - “Phased”

Phased Communication

- It may be easier for the MPI implementation to either send or receive
- Color the nodes so that all senders are of one color and all receivers of the other. Then use two phases
  - Just a “Red-Black” partitioning of nodes
  - For more complex patterns, more colors may be necessary



This is an example of manual scheduling a communication step. Only consider this if there is evidence of inefficient communication.

Halo Exchange on BG/L

- 64 processes, co-processor mode, 2048 doubles to each neighbor
- Rate is MB/Sec (for all tables)

	4 Neighbors		8 Neighbors	
	Irecv/Send	Irecv/Isend	Irecv/Send	Irecv/Isend
World	112	199	94	133
Even/Odd	81	114	71	93
Cart_create	107	218	104	194

Halo Exchange on BG/L

- 128 processes, virtual-node mode, 2048 doubles to each neighbor
- Same number of *nodes* as previous table

	4 Neighbors		8 Neighbors	
	Irecv/Send	Irecv/Isend	Irecv/Send	Irecv/Isend
World	64	120	63	72
Even/Odd	48	64	41	47
Cart_create	103	201	103	132

BG/P Comments

- Like BG/L, except a little faster/core; 2x cores per node
- Halo exchange results show similar properties to BG/L results
  - Default layout of MPI_COMM_WORLD is better for nearest neighbor exchanges compared to BG/L, at least when these tests were run
  - Topology still matters (poor layout results in significantly reduced effective bandwidth)
  - Still running pre-ship software, so no results yet

Halo Exchange on Cray XT3

- 1024 processes, 2000 doubles to each neighbor

	4 Neighbors			8 Neighbors	
	Irecv/Send	Irecv/Isend	Phased	Irecv/Send	Irecv/Isend
World	134	128	148	116	113
Even/Odd	118	114	125	102	97
Cart_create	114	117	129	99	99

(Periodic)	4 Neighbors			8 Neighbors	
	Irecv/Send	Irecv/Isend	Phased	Irecv/Send	Irecv/Isend
World	109	110	121	97	96
Even/Odd	100	104	108	91	90
Cart_create	125	123	139	111	109

Halo Exchange on Cray XT4

- 1024 processes, 2000 doubles to each neighbor

	4 Neighbors			8 Neighbors	
	Irecv/Send	Irecv/Isend	Phased	Irecv/Send	Irecv/Isend
World	153	153	165	133	136
Even/Odd	128	126	137	114	111
Cart_create	133	137	143	117	117

(Periodic)	4 Neighbors			8 Neighbors	
	Irecv/Send	Irecv/Isend	Phased	Irecv/Send	Irecv/Isend
World	131	131	139	115	114
Even/Odd	113	116	119	104	104
Cart_create	151	151	164	129	128

Halo Exchange on Cray XT4

- 1024 processes, SN mode, 2000 doubles to each neighbor

	4 Neighbors			8 Neighbors	
	Irecv/Send	Irecv/Isend	Phased	Irecv/Send	Irecv/Isend
World	311	306	331	262	269
Even/Odd	257	247	279	212	206
Cart_create	265	275	266	236	232

(Periodic)	4 Neighbors			8 Neighbors	
	Irecv/Send	Irecv/Isend	Phased	Irecv/Send	Irecv/Isend
World	264	268	262	230	233
Even/Odd	217	217	220	192	197
Cart_create	300	306	319	256	254

Observations on Halo Exchange

- Topology is important (again)
- For these tests, MPI_Cart_create always a good idea for BG/L; often a good idea for periodic meshes on Cray XT3/4
- Cray performance is significantly under what the “ping-pong” performance test would predict
  - The success of the “phased” approach on the Cray suggests that some communication contention may be contributing to the slow-down
  - To see this, consider the performance of a single process sending to four neighbors

Discovering Performance Opportunities

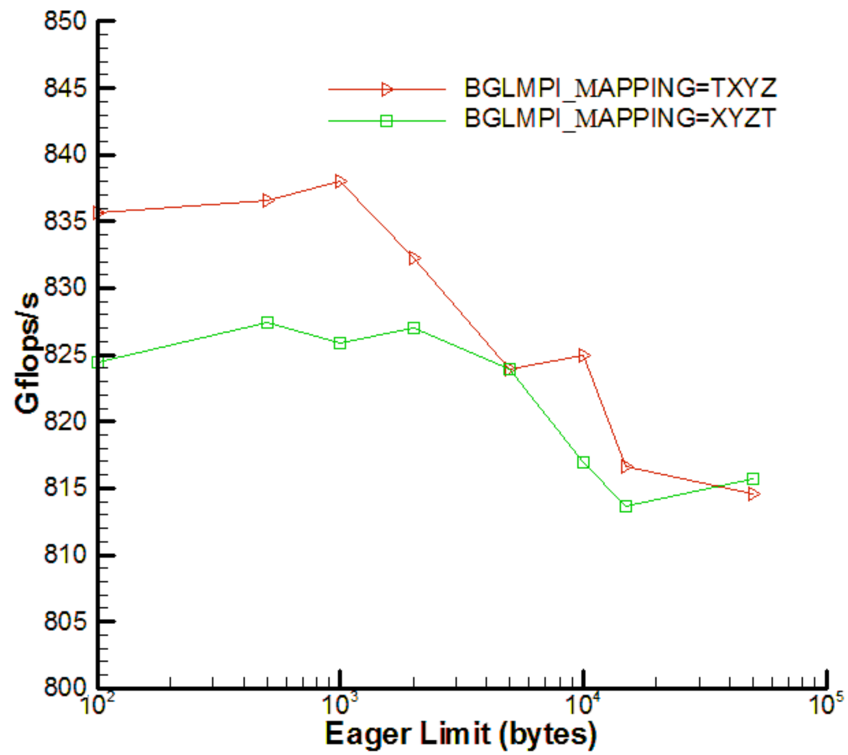
- Lets look at a single process sending to its neighbors. We *expect* the rate to be roughly twice that for the halo (since this test is only sending, not sending and receiving)

System	4 neighbors		8 Neighbors	
		Periodic		Periodic
BG/L	488	490	389	389
BG/L, VN	294	294	239	239
XT3	1005	1007	1053	1045
XT4	1634	1620	1773	1770
XT4 SN	1701	1701	1811	1808

- BG gives roughly double the halo rate. XTn is much higher
  - It should be possible to improve the halo exchange on the XT by scheduling the communication
  - Or improving the MPI implementation

Tuning MPI with Environment variables

- The plot shows the effect of BGLMPI_EAGER and BGLMPI_MAPPING on the performance of PETSc-FUN3D (http://www.mcs.anl.gov/~kaushik/perf.htm) on 2048 processors of BGL.
- BGLMPI_ALLREDUCE=TORUS, BGLMPI_ALLREDUCE=TREE select which network is used for MPI_Allreduce
- Cray XT also uses environment variables
  - MPICH_RANK_REORDER_METHOD
  - MPI_COLL_OPT_ON
- Mapping controls can help applications that use MPI_COMM_WORLD (most apps should use a comm to allow the setup code to form a “good” communicator



Why Environment Variables are Bad

- On BG/P, the environment variable to control process mapping is BGML_MAPPING
- If you use BGLMPI_MAPPING as needed on BG/L, you will not get the expected mapping, and no warning message
- It is better to do this (portably) in your program than to count on the vendors to remember the names of their own environment variables.

MPI-2: Revisiting Mesh Communication

- 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*
- MPI-2 provides these "one-sided" or "remote memory access" routines
  - BG/L does not support these
  - BG/P and Cray XTn do, but performance is still an open question
  - It is possible to implement these well and get an advantage over point-to-point communications
- First, we'll cover some of the RMA basics. Then we'll see some examples of a good implementation

Remote Memory Access

- A key feature is that it separates data transfer from indication of completion (synchronization)
- In message-passing, they are combined:

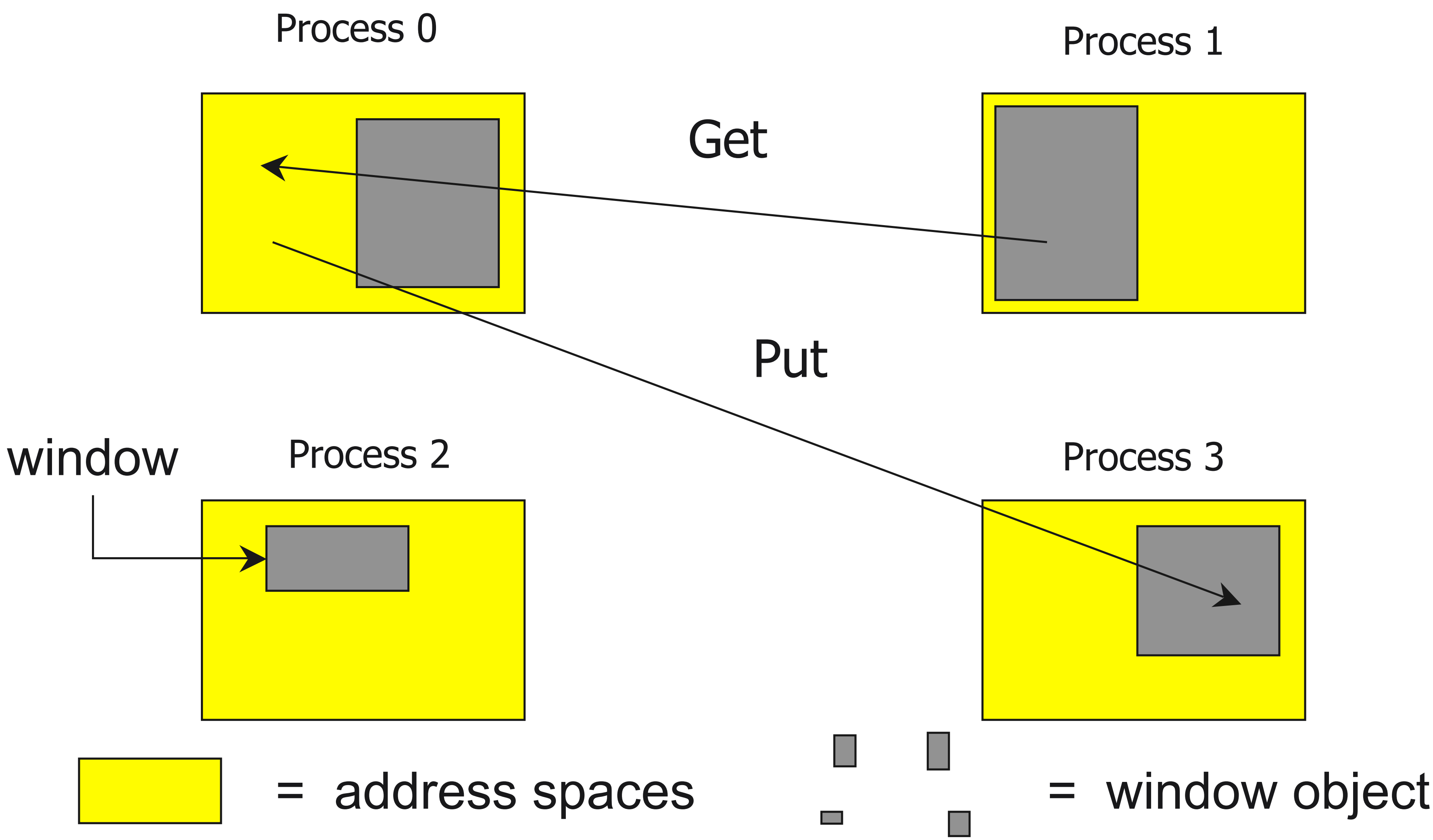
Proc 0	Proc 1
store	
send	receive
	load

Proc 0	Proc 1
fence	fence
put	
fence	fence
	load
or	
store	
fence	fence
	get

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

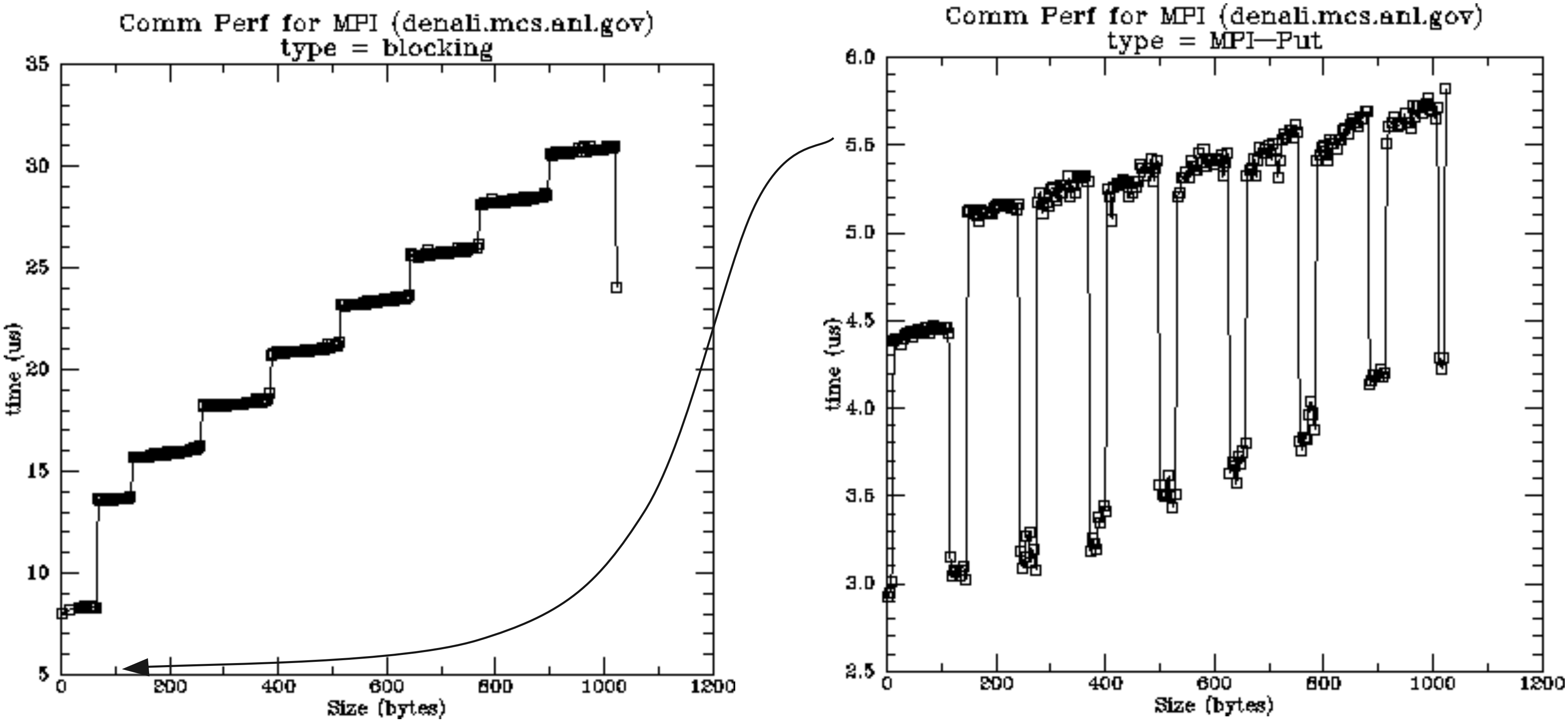


Basic RMA Functions for Communication

- **`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**

Performance of RMA (early results)



Caveats: On SGI, MPI_Put uses specially allocated memory

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 a priori, the send-recv model requires an extra step to determine how many sends-recvs 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

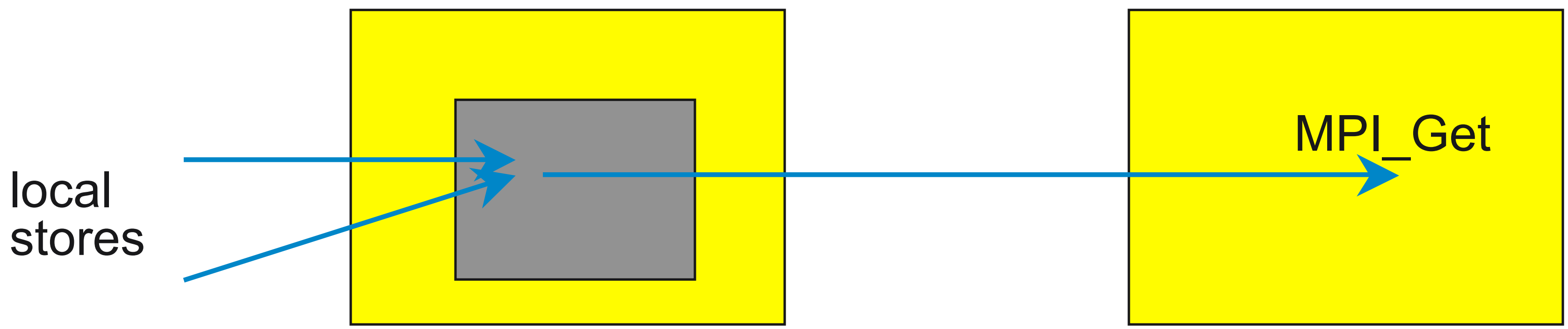
Put, Get, and Accumulate

- `MPI_Put(origin_addr, origin_count, origin_datatype, target_rank, target_offset, target_count, target_datatype, window)`

```
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

Process 0	Process 1
MPI_Win_fence(win)	MPI_Win_fence(win)
MPI_Put	
MPI_Put	
MPI_Win_fence(win)	MPI_Win_fence(win)

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

Process 0	Process 1
MPI_Win_start(target_grp)	MPI_Win_post(origin_grp)
MPI_Put	
MPI_Put	
MPI_Win_complete(target_grp)	MPI_Win_wait(origin_grp)

Lock-Unlock Synchronization

- "Passive" target: The target process does not make any synchronization call
- When MPI_Win_unlock returns, the preceding RMA operations are complete at both source and target

```
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
```

Fence vs Lock/Unlock Synchronization

- Fence synchronization method requires all processes in the communicator (that created the window) to call the fence function. It is almost like a barrier.
- Lock/unlock synchronization is called only by the process that needs to do the Put or Get. The target process does not call anything.
  - But this is more challenging for the MPI implementation to make fast, especially if the underlying hardware doesn't support direct RMA operations

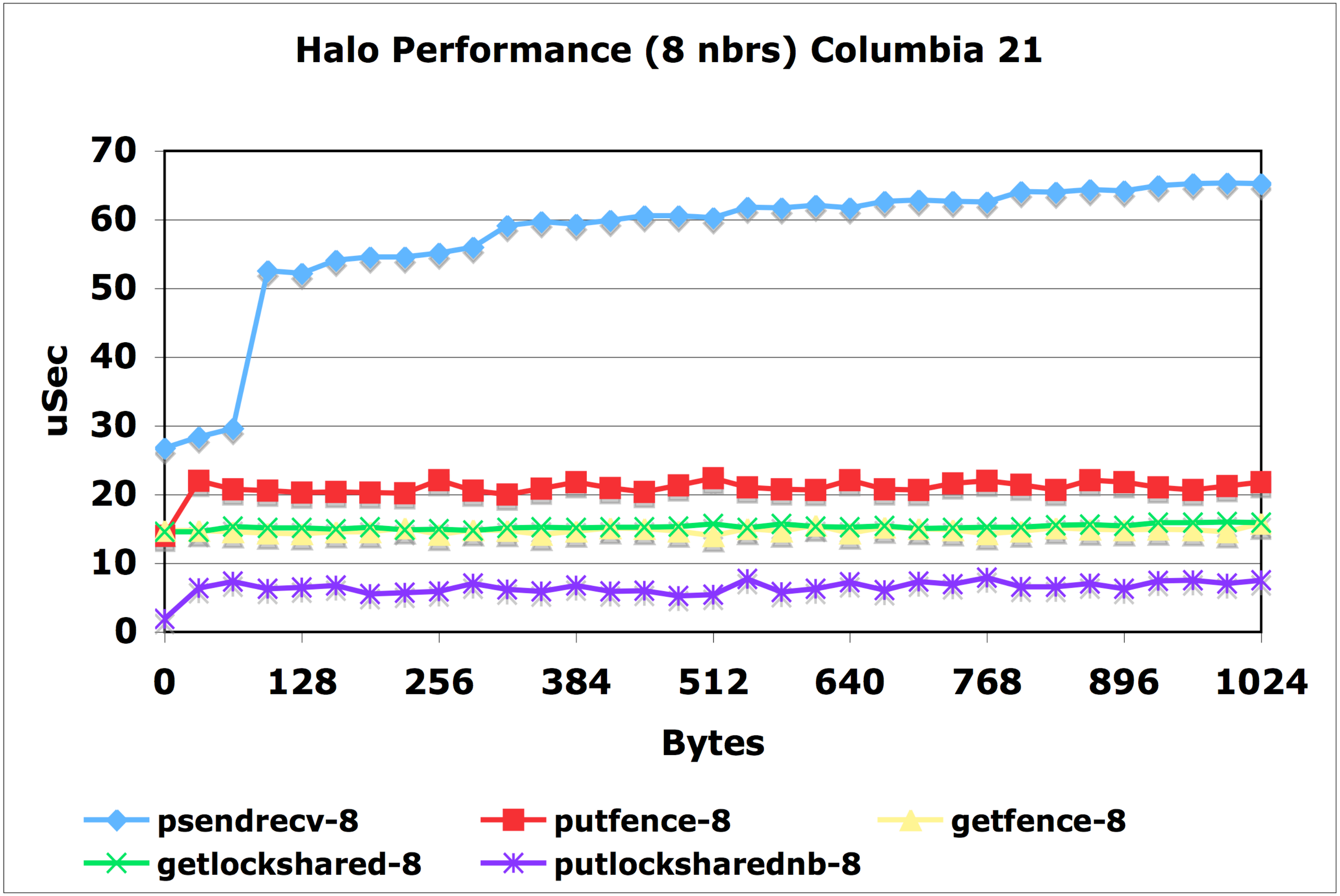
Halo Exchange Benchmark

- Part of the mpptest benchmark; works with any MPI implementation
  - Even handles implementations that only provide a subset of MPI-2 RMA functionality
  - Similar code to that in halocompare, but doesn't use process topologies (yet)
- Available from
- http://www.mcs.anl.gov/mpi/mpptest
- Mimics a halo, or ghost-cell, exchange that is a common component of parallel codes that solve partial differential equations

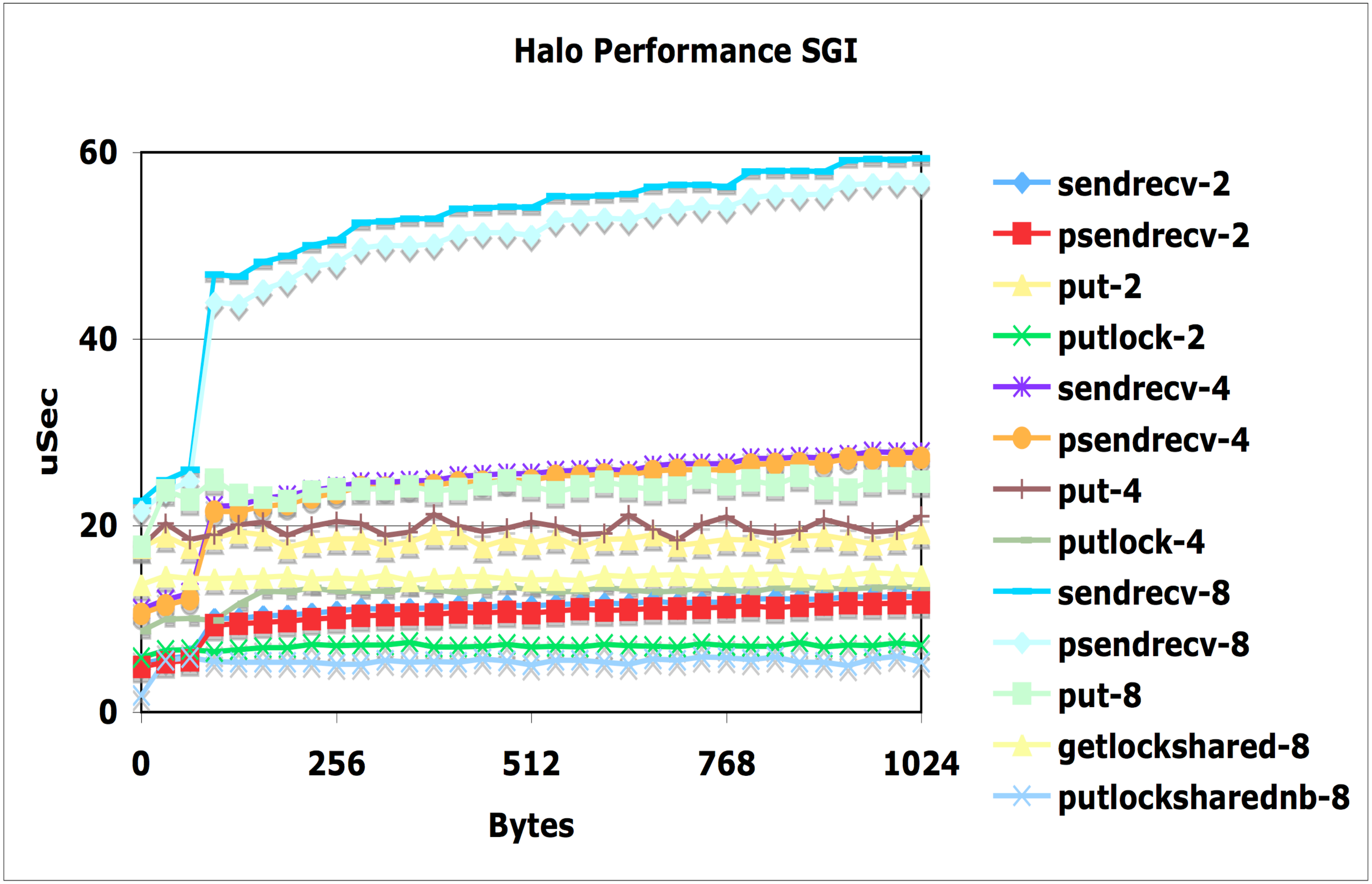
Persistent Send/recv

- Persistent Send/recv:
  - This version uses nonblocking operations for both sending and receiving; primarily, this is to handle the buffering issues. In order to increase the efficiency, MPI persistent operations are used
- This is very similar to the simple nonblocking example.
  - The halo experiments with the LC systems did not show an advantage to using persistent operations in the halocompare tests.

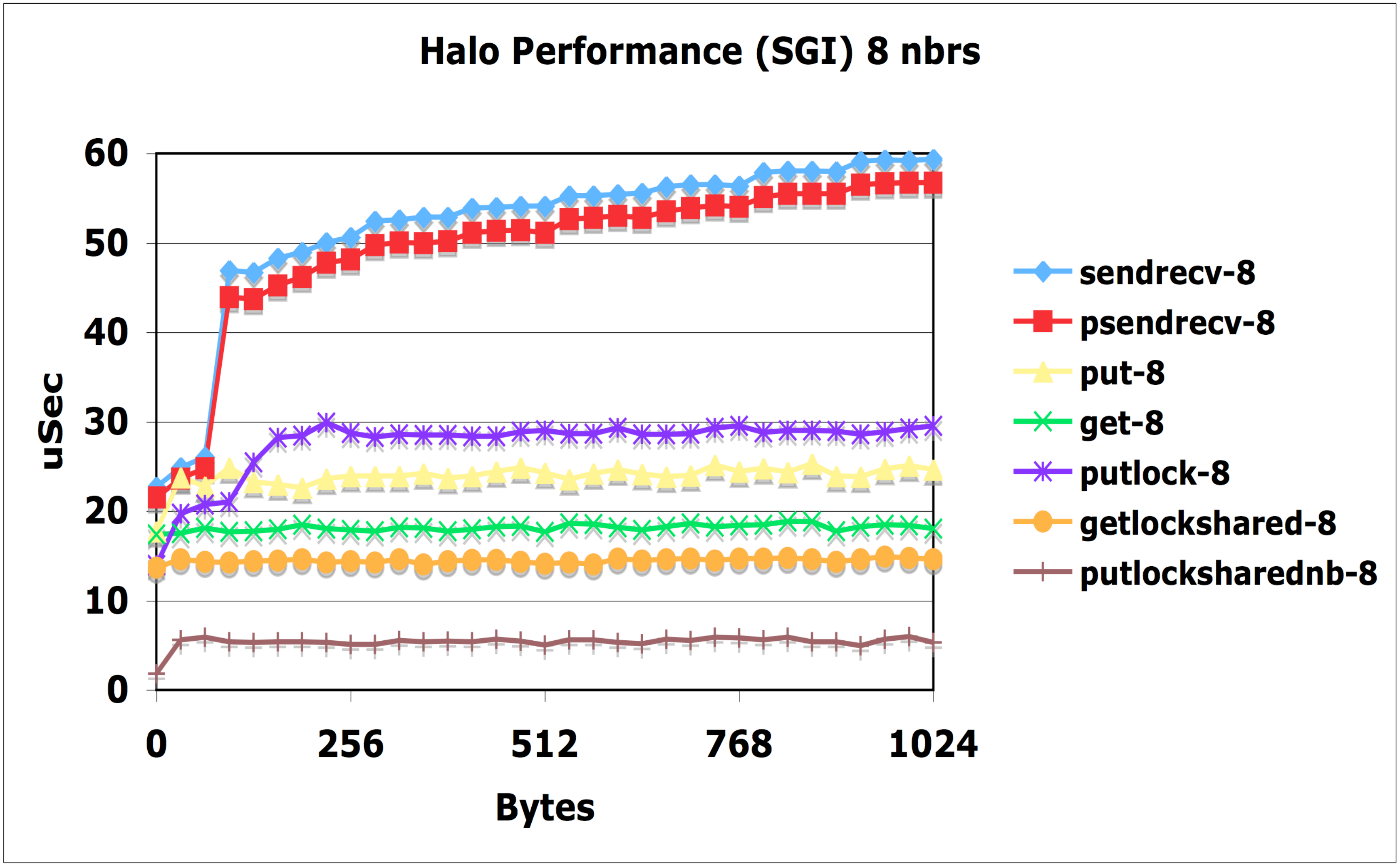
Halo Performance (8 nbrs) Columbia 21



Columbia 20


Halo Performance SGI
60
40
20
0
uSec
0
256
512
768
1024
Bytes
sendrecv-2
psendrecv-2
put-2
putlock-2
sendrecv-4
psendrecv-4
put-4
putlock-4
sendrecv-8
psendrecv-8
put-8
getlockshared-8
putlocksharednb-8

Columbia 20



Halo Performance (SGI) 8 nbrs
60
50
40
30
20
10
0
uSec
0
256
512
768
1024
Bytes
sendrecv-8
psendrecv-8
put-8
get-8
putlock-8
getlockshared-8
putlocksharednb-8

MPI RMA on SGI Altix

- Performance of Columbia 21 > Columbia 20 > Columbia 8
- Performance of "GET" > "PUT"
- Performance of "PUT" and "GET" is much better than "SEND" and "RECV"
- Performance MPI RMA is much better than the POINT-TO-POINT communication on Columbia
- RMA performance on Columbia is excellent
- On Columbia "lock-put-unlock" is 10 times better than "send-receive"
- On Columbia "fence" method is 2 times better than "send-receive"

Acknowledgement

- A special thanks to Subhash Saini of NASA Advanced Supercomputing for providing the Altix runs
- Thanks to Dale Talcott of NASA Ames Research Center for running earlier version of the benchmarks on Columbia 21.
- Thanks to Dinesh Kaushik for the XT experiments and to ORNL for access to their machines.

MPI and Threads

- MPI describes parallelism between *processes*
- *Thread* parallelism provides a shared-memory model within a process
- OpenMP and Pthreads are common models
  - 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

Threads on LC Machines

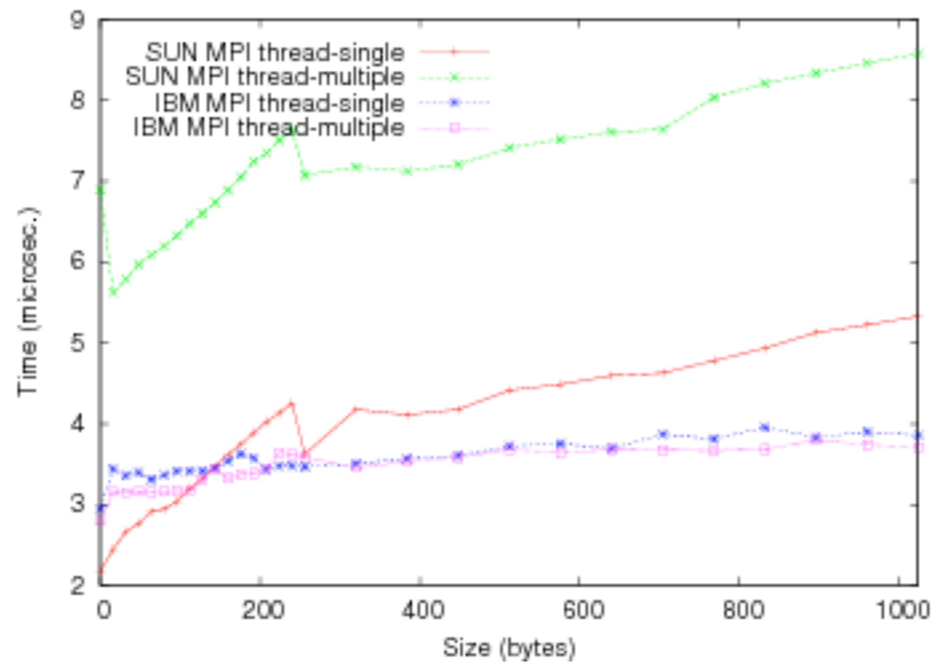
- MPI and Threads
  - MPI_Init_thread(&argc, &argv, requested, &provided)
  - The four levels of thread safety
    - *MPI_THREAD_SINGLE*
    - *MPI_THREAD_FUNNELED*
    - *MPI_THREAD_SERIAL*
    - *MPI_THREAD_MULTIPLE*
- Using threads
  - OpenMP
    - *Compiler handles most operations*
  - Pthreads
    - *Like MPI, you get to do everything yourself :)*
  - Limitations imposed by OS
    - *With current compute-node kernels, threads bound to cores*
    - *Linux will enable real thread programming*

Thread Performance

- Thread safety is not free
  - Managing atomic access to shared data structures adds overhead (you never know when a thread might update the same item)
  - Scheduling access to shared resources (e.g., interconnect) can introduce additional contention

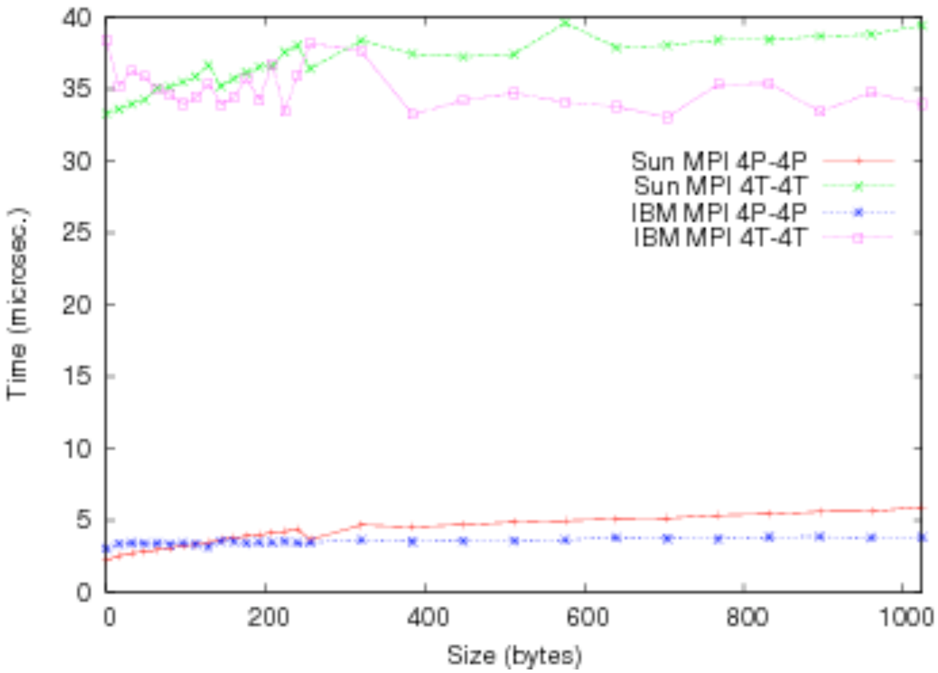
Overhead of Providing Thread Safety

- This test uses a single-threaded MPI process, but uses the “requested” argument to MPI_Init_thread to select either MPI_THREAD_SINGLE or MPI_THREAD_MULTIPLE
- The IBM SP implementation has very low overhead
- The Sun implementation has about a 3.5 usec overhead
  - Shows cost of providing thread safety
  - This cost can be lowered, but requires great care



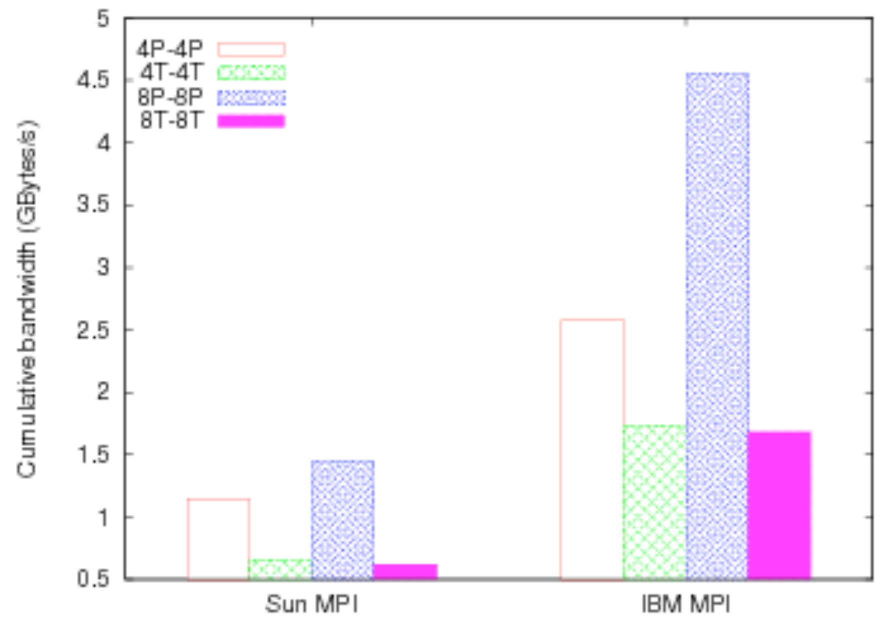
Thread Overhead

- These tests compare the performance of short message sends when using single-threaded MPI processes and multiple threaded processes, with the same total number of threads
- For these systems, thread overhead is high
  - Achieving low-overhead thread-safe code is difficult



Threads vs. Processes

- This test compares using processes or threads to communicate between nodes on an SMP; the machines are a Sun and an IBM SP
- Processes achieve a much higher bandwidth
  - Likely that processes share interconnect more effectively than threads on these systems



Some Recommendations on the Use of Threads

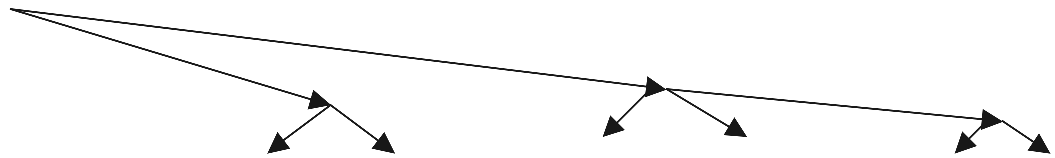
- Best used when threads can help balance compute load or distribute communication
- Always estimate performance and measure.
- Provide realistic (but simple) test cases to help implementations identify and solve real performance issues
- The impact of the multithreaded programming model on scalable scientific applications is a new issue for vendors, middleware developers, and applications alike.

MPI Collectives

- Can provide access to tuned algorithms for the particular physical hardware
  - Depends on the MPI implementation
  - BG/L and BG/P have special networks that are used for some collective operations when applied to all processes in MPI_COMM_WORLD
- However, the optimized collectives may not always be the best choice in an application

Broadcast Algorithms

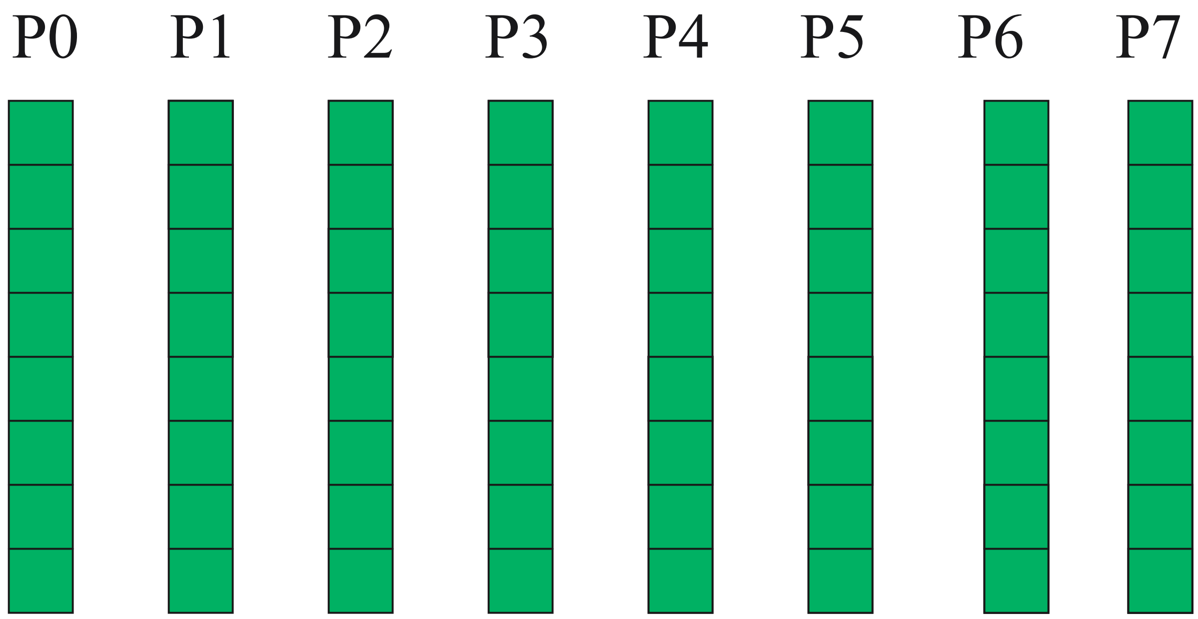
- MPI_Bcast(buf, 100000, MPI_DOUBLE, …);
- Use a tree-based distribution:



- Use a *pipeline*: send the message in b byte pieces. This allows each subtree to begin communication after b bytes sent
- Improves total performance:
  - Root process takes same time (asymptotically)
  - Other processes wait less
    - *Time to reach leaf is* b log p + (n-b)*, rather than* n log p

Bcast with Scatter/Gather

- Implement MPI_Bcast(buf,n,…) as
  MPI_Scatter(buf, n/p,…, buf+rank*n/p,…)
  MPI_Allgather(buf+rank*n/p, n/p,…,buf,…)



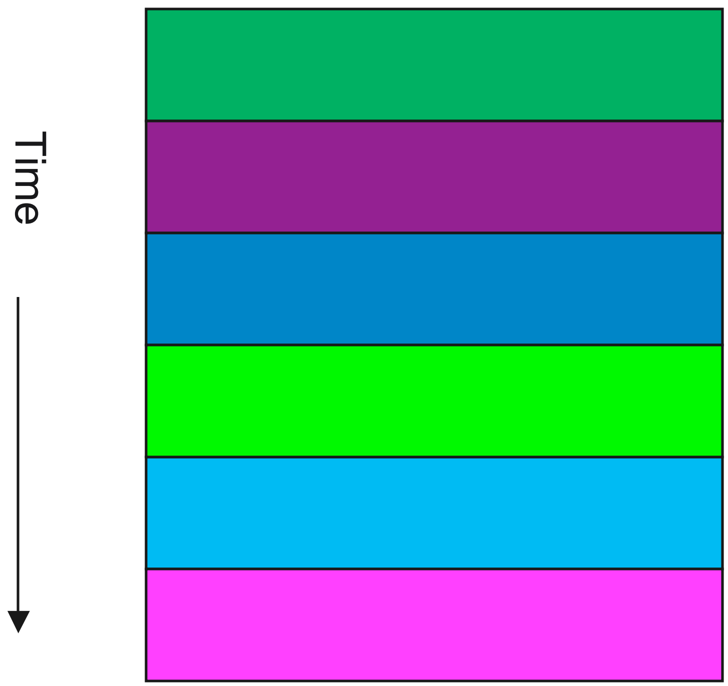
Time is O(n) + O(log p) instead of O(nlogp)

When not to use Collective Operations

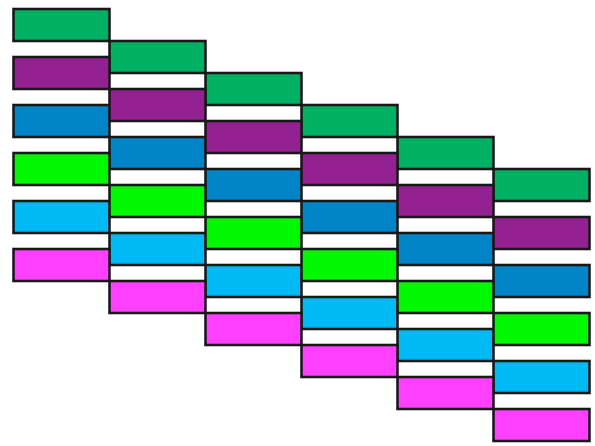
- Sequences of collective communication can be pipelined for better efficiency
- Example: Processor 0 reads data from a file and broadcasts it to all other processes.
  - ```
    Do i=1,m
      if (rank .eq. 0) read *, a
      call mpi_bcast( a, n, MPI_INTEGER, 0, comm, ierr )
    EndDo
    ```
  - Takes m n log p time.
- It can be done in (m+p) n time!
    ```

Pipeline the Messages

- Processor 0 reads data from a file and sends it to the next process. Other forward the data.
  - ```
    Do i=1,m
      if (rank .eq. 0) then
          read *, a
          call mpi_send(a, n, type, 1, 0, comm,ierr)
      else
          call mpi_recv(a,n,type,rank-1, 0,comm,status, ierr)
          call mpi_send(a,n,type,next, 0, comm,ierr)
      endif
    EndDo
    ```
    ```

Concurrency between Steps

- Broadcast:



- Pipeline



Each broadcast takes less time then "optimized" version, but total time is longer

Total time ≠ Σtime each

Another example of deferring synchronization.

Always evaluate your strategy in the context of the big picture

Be careful of "peephole optimization"

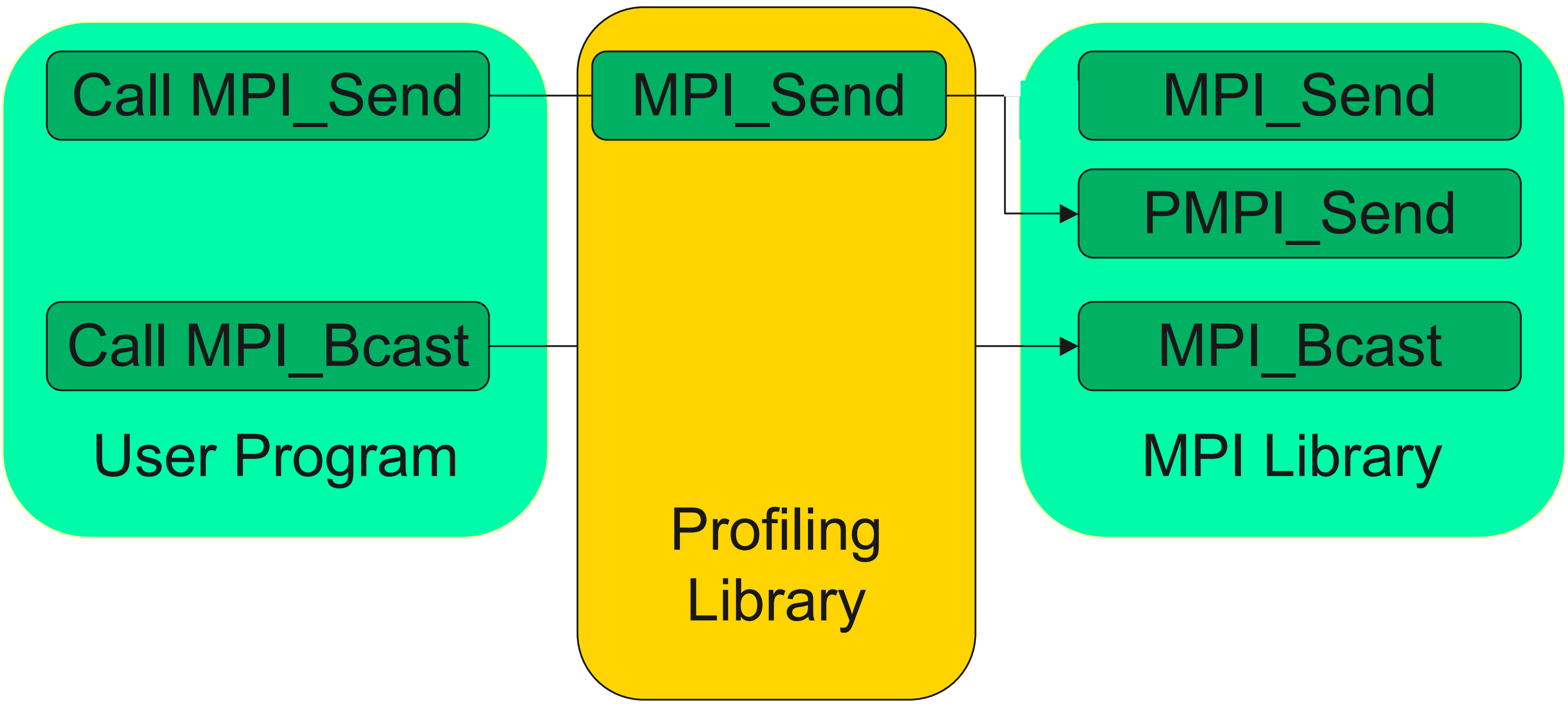
Solving Performance Problems

- Solving *your* performance problem requires that
  - You understand how fast your code should go
  - How fast it actually goes
  - Possible interactions that may help explain the behavoir
- MPI provided a powerful hook on which tools can and are built - the profiling interface
  - In addition to general-purpose tools, this interface is available to all
    - *You can build custom tools to explore application-specific hypotheses*

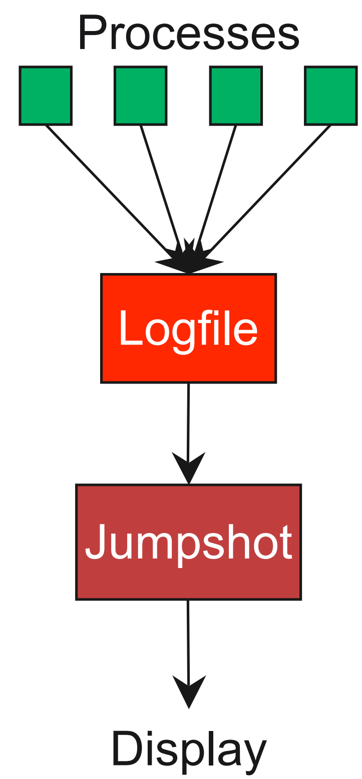
Tools Enabled by the MPI Profiling Interface

- 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*

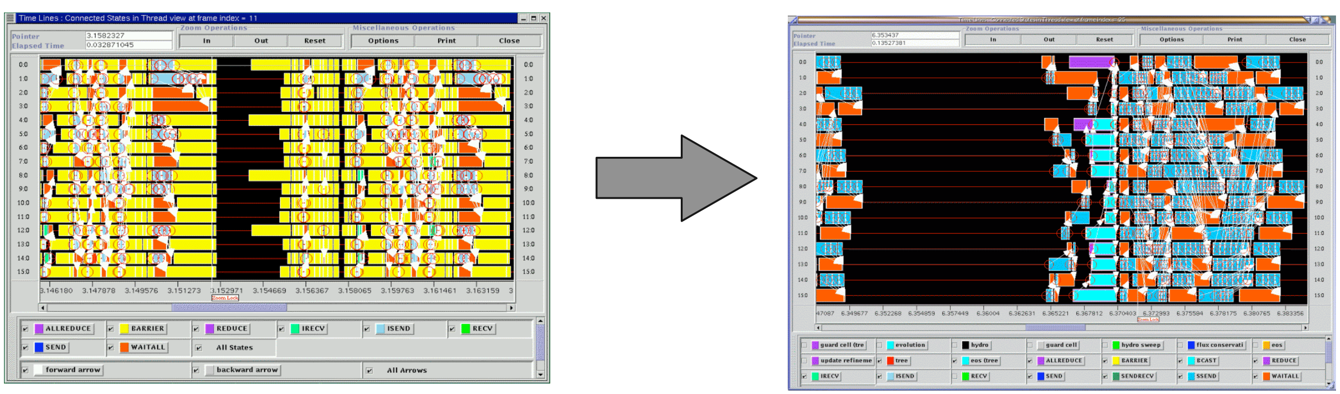
The MPI Profiling Interface

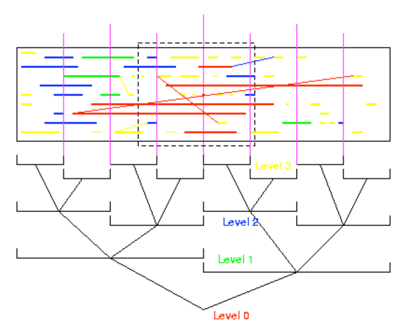


Performance Visualization with Jumpshot

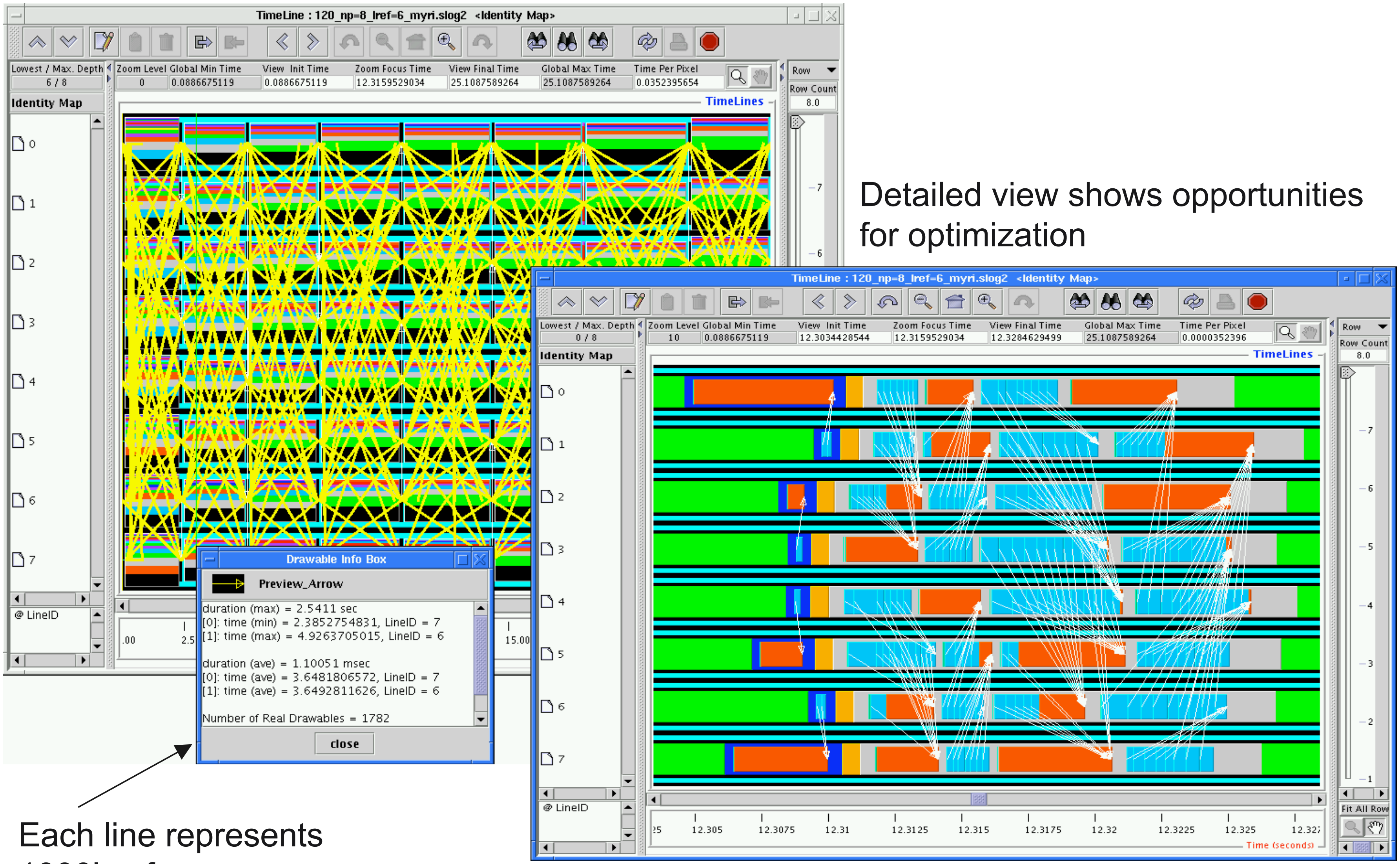


- For detailed analysis of parallel program behavior, timestamped events are collected into a log file during the run.
- A separate display program (Jumpshot) aids the user in conducting a post mortem analysis of program behavior.
- We use an indexed file format (SLOG-2) that uses a preview to select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





Viewing Multiple Scales



Detailed view shows opportunities for optimization

Each line represents 1000's of messages

1000x zoom

Pros and Cons of this Approach

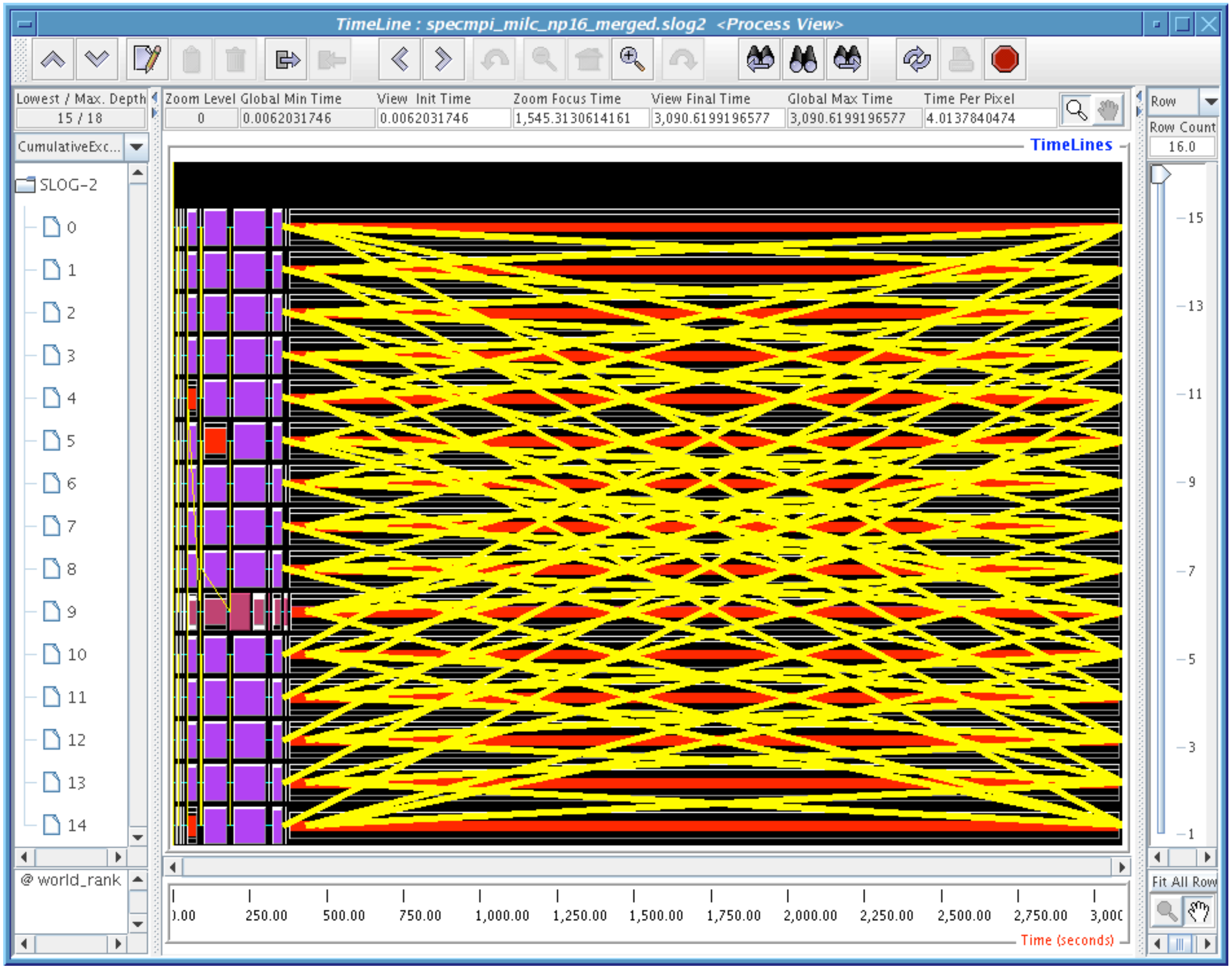
- Cons:
  - Scalability limits
    - *Screen resolution*
    - *Big log files, although*
      - Jumpshot can read SLOG files fast
      - SLOG can be instructed to log few types of events
  - Use for debugging only indirect
- Pros:
  - Portable, since based on MPI profiling interface
  - Works with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

Some Examples of Jumpshot in Use

- Original FLASH Sedov, after first round of tuning
- Observing MPI interacting with threads
- GFMC
- ADLB

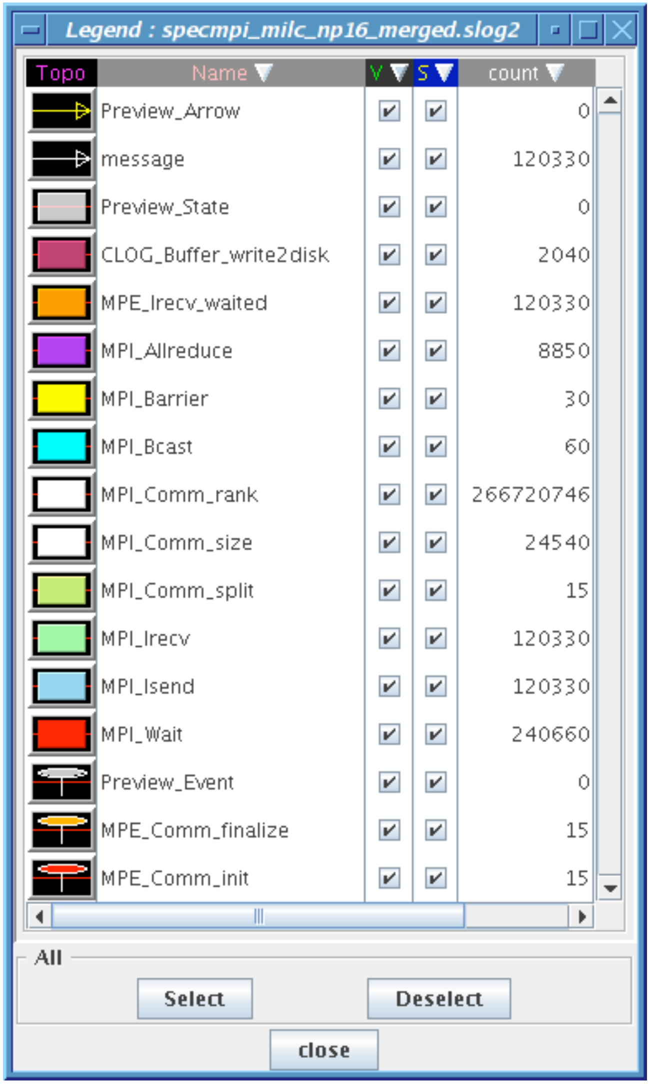
Looking at MILC in SPEC2007

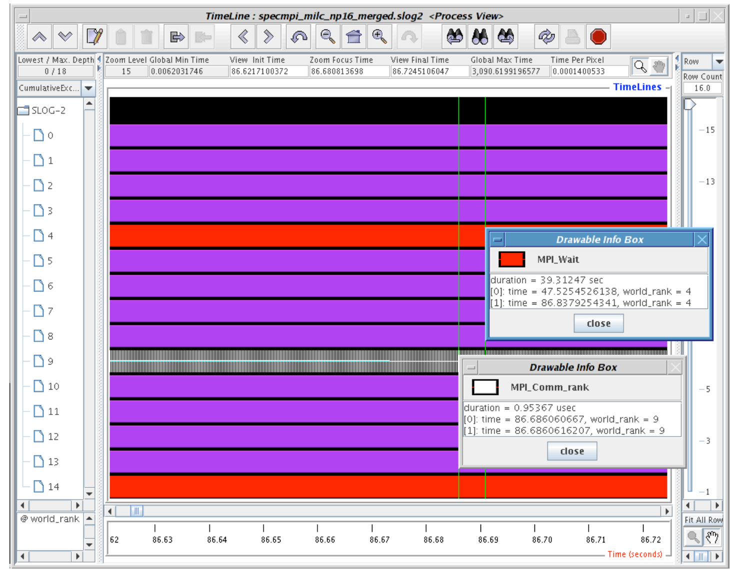
- Curious amount of All_reduce in initialization - why?

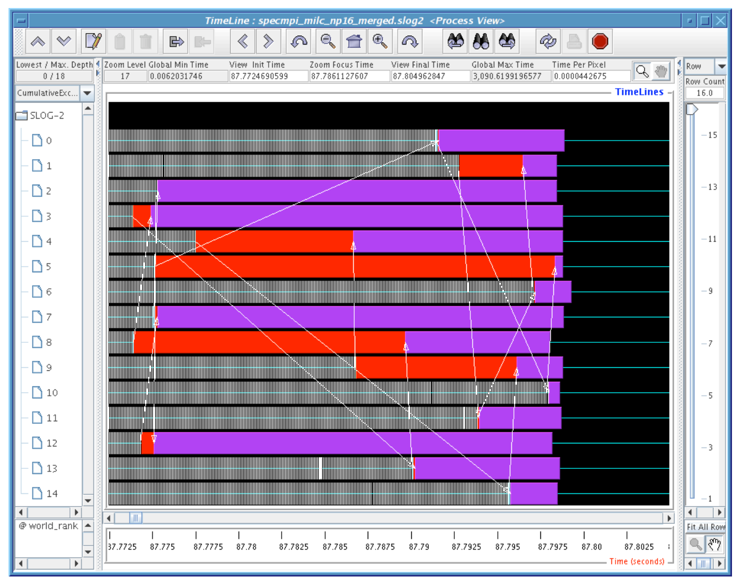


MILC

- 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)

FPMPI2

- Creates a text summary of the use of each MPI call
- Special Capability
  - Distinguishes between messages of different sizes within 32 message bins (essentially powers of two)
- Optionally identifies *synchronization* time - the time that an MPI call is forced to wait
  - On collective calls
    - *Separates* the time that a collective call waits for the other processes to enter the call from the time to perform the collective operation
  - On blocking sends
    - *Determine the time until the matching receive is posted*
  - On blocking receives
    - *Determine the time that the receive waits until the message arrives*
  - All implemented with MPI calls
    - *Pro: Completely portably*
    - *Con: Adds overhead (e.g., MPI_Send -> MPI_Issend/Test)*
- Available from www.mcs.anl.gov/fpmpi .

Example FPMPI Output (1)

```
Date:                   Fri Sep  8 15:20:03 2006
Processes:              4
Execute time:           0.07528
Timing Stats: [seconds]        [min/max]           [min rank/max rank]
   wall-clock: 0.07528 sec     0.074663 / 0.076100 1 / 2
         user: 0.05685 sec     0.055616 / 0.059816 1 / 0
          sys: 0.03737 sec     0.036252 / 0.038266 0 / 2
  Memory Usage Stats (RSS) [min/max KB]:           6068/6116
                    Average of sums over all processes
Routine                     Calls       Time Msg Length    %Time by message length
                                                        0.........1........1........
                                                                  K        M
MPI_Bcast               :       2   1.81e-05        8.5 0040600000000000000000000000
MPI_Reduce              :       1   0.000124          8 00*0000000000000000000000000
MPI_Isend               :      40    0.00054   3.96e+03 0000080200000000000000000000
MPI_Irecv               :      40   0.000221   3.96e+03 0000070300000000000000000000
MPI_Waitall             :      20   0.000382
```

Example FPMPI Output (2)

```
Details for each MPI routine
                     Average of sums over all processes
                                     % by message length
                           (max over          0.........1........1........
                            processes [rank])           K        M
MPI_Bcast:
Calls      :          2             2 [    0] 0050500000000000000000000000
Time       :   1.81e-05       2.1e-05 [    2] 0040600000000000000000000000
Data Sent  :        8.5            34 [    0]
MPI_Reduce:
Calls      :          1             1 [    0] 00*0000000000000000000000000
Time       :   0.000124      0.000163 [    0] 00*0000000000000000000000000
Data Sent  :          8             8 [    0]
MPI_Isend:
Calls      :         40            40 [    0] 0000050500000000000000000000
Time       :    0.00054      0.000637 [    1] 0000080200000000000000000000
Data Sent  :   3.96e+03          4000 [    2]
Partners   :          2 max 2(at 0) min 2(at 0)
MPI_Irecv:
Calls      :         40            40 [    0] 0000050500000000000000000000
Time       :   0.000221      0.000269 [    2] 0000070300000000000000000000
Data Sent  :   3.96e+03          4000 [    2]
```

The newest version also estimates synchronization time, allowing identification of load imbalance or misplaced sends/receives

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 on not on the MPI datatypes themselves, but on the <u>signature</u> 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.

Datatype Signature Hashing

- Gropp – EuroPVM/MPI 2000
- Matching is done on pairs (a, n), where a is a hash value and n is the number of basic datatypes in the message.
- Elementary datatypes assigned (a, 1) for chosen values of a.
- Concatenate types with
  - (a,n) # (b,n) = (a xor (b << n), n+m), where << is circular left shift
  - Note non-commutative to prevent (int, float) from colliding with (float, int)
- The pairs (a,n) are easy to communicate to other processes, unlike the signatures themselves
  - (No MPI datatype for MPI_Datatype)
  - We will use PMPI_Bcast, PMPI_Scatter, PMPI_Allgather, PMPI_Alltoall as needed to communicate the (vector of) hash pairs to the other processes.

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 <u>instance specific:</u>

- **`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

Some Thoughts on "Hierarchical Parallelism," Master-Slave Algorithms, and Load Balancing

- Old way:
  - Master code manages work pool, hands out work to slaves, collects results, "automatic" load balancing
  - Intrinsically not scalable
- A possible new way: "Symmetric task farming"
  - All processes repeat:

    Get work from pool

    Do work

    Send results to whoever wants them

    Put newly created work in pool
- The pool of work is managed by an opaque library
  - Might use threads
  - Might use some processes

Conclusions

- MPI provides effective ways to access communication performance
  - You may need to help the implementation out
  - See vendor's documentation; e.g., for BG/L and BG/P, see the IBM RedBooks
  - However, avoid the non-standard extensions unless you can get a significant benefit from them (e.g., use MPI_Cart_create instead of non-standard routines)
  - MPI RMA merits consideration
    - *But perform timing tests before committing to it*
    - *Best to form a communication abstraction with RMA one available implementation*
  - MPI Profiling interface gives *you* access to ways to diagnose performance problems

Discussion

- Connecting these ideas to applications at this workshop
  - Use of tools
  - Improving performance
- Preparation of application kernels
- Is this workshop on the right track?
  - Do you want to meet again next year?