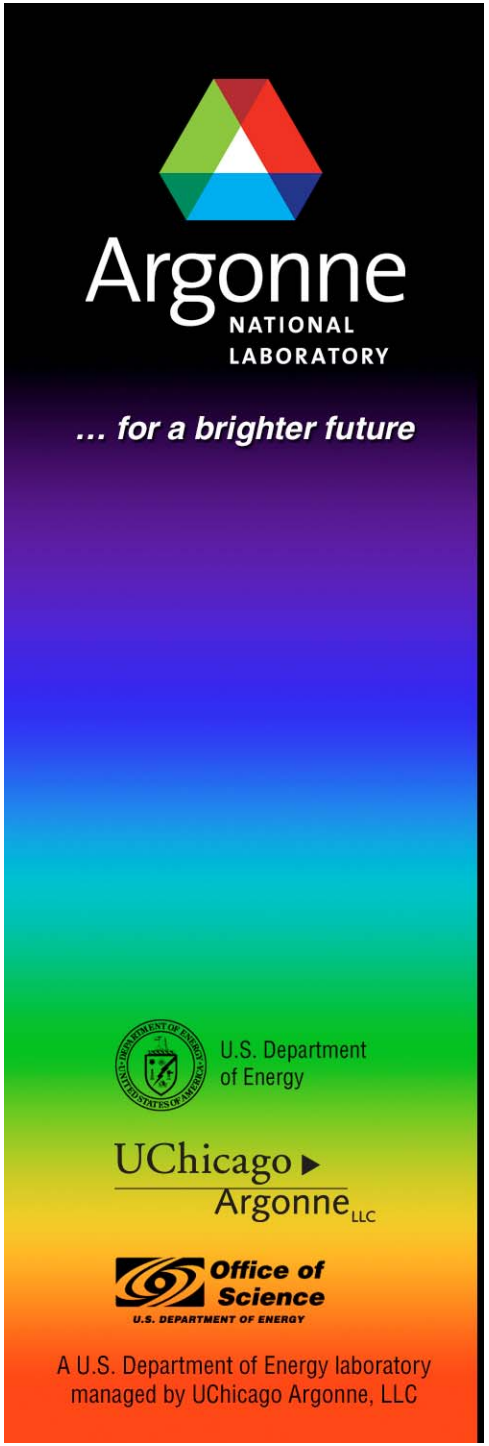


Programming in MPI for Performance

and

Programming Models for HPC

Rusty Lusk

Mathematics and Computer Science Division

Argonne National Laboratory

Outline

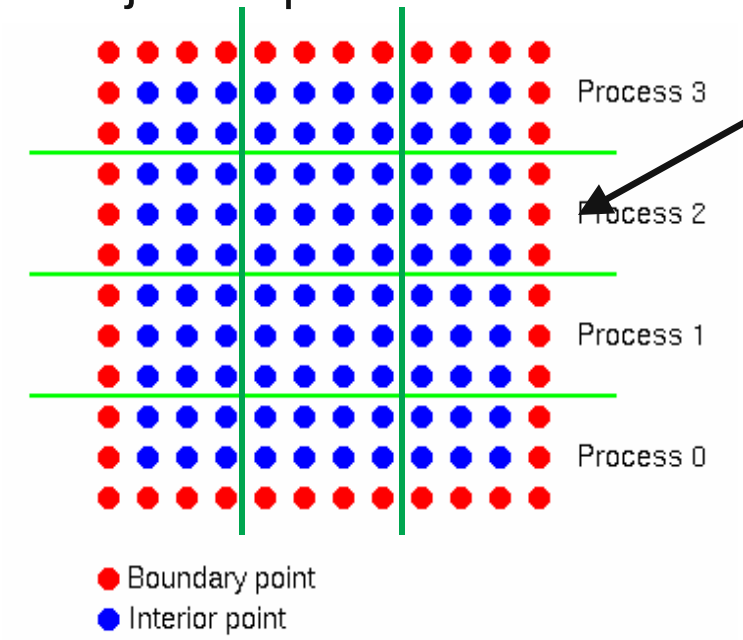
- Topics in MPI programming
- Selected tools enabled by MPI profiling interface
  - SLOG/Jumpshot: visualizing parallel performance
  - FPMPI: gathering summary statistics
  - Collchk: runtime checking of correct use of collective operations
- Programming models for HPC
  - MPI
  - PGAS
  - HPCS
- Hybrid programming (MPI and OpenMP)
- Avoiding MPI with special libraries (ADLB)

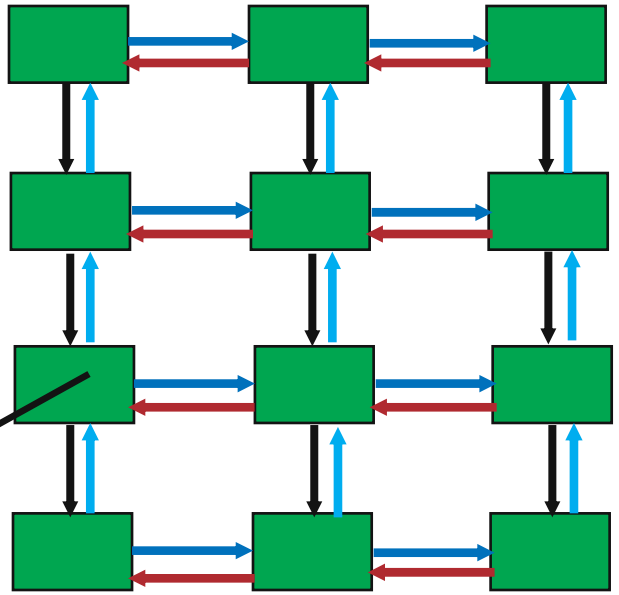
Basic MPI: Looking Closely at a Simple Communication Pattern

- 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



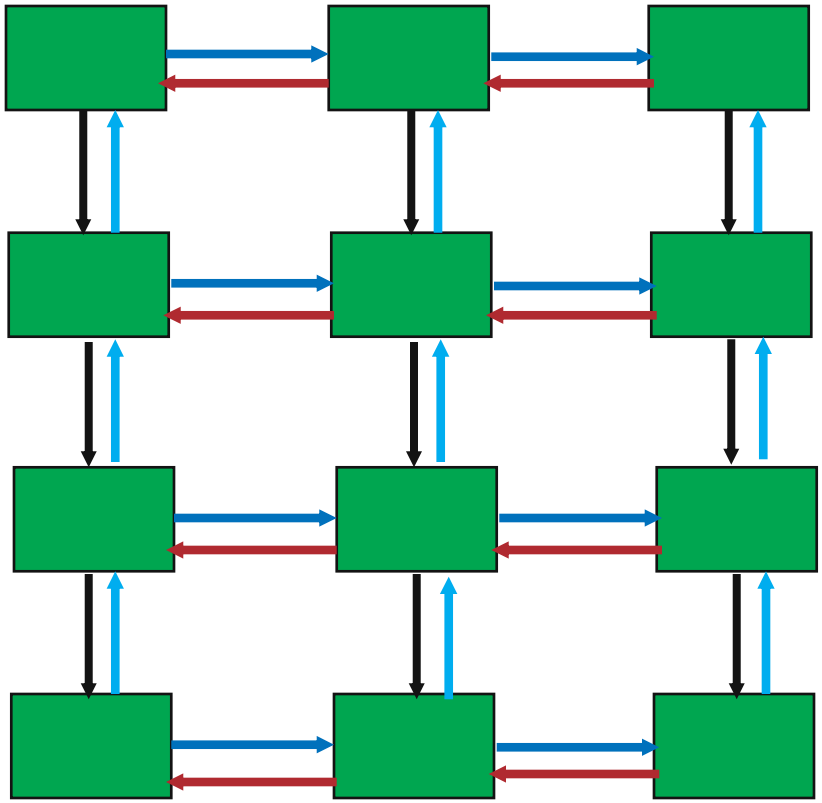


Scalability of Mesh Exchange

- 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

Fix 1: Use Irecv

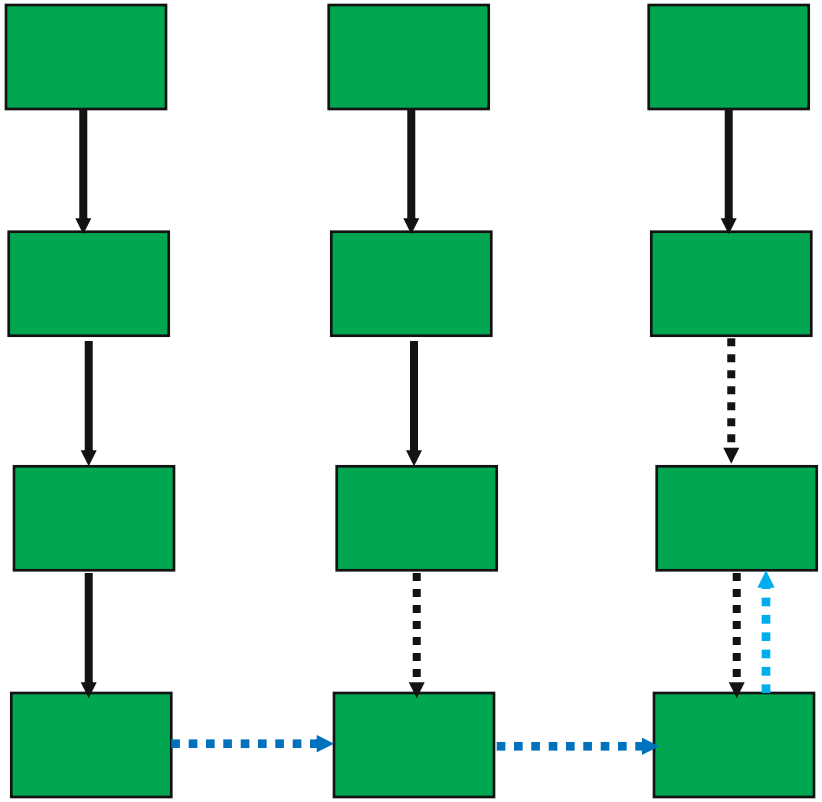
- Do i=1,n_neighbors
  Call MPI_Irecv(inedge(1,i), len, MPI_REAL, nbr(i), tag,&
  comm, requests(i), ierr)
  Enddo
  Do i=1,n_neighbors
  Call MPI_Send(edge(1,i), len, MPI_REAL, nbr(i), tag,&
  comm, ierr)
  Enddo
  Call MPI_Waitall(n_neighbors, requests, statuses, ierr)
- Does not perform well in practice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

Understanding the Behavior: A 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)

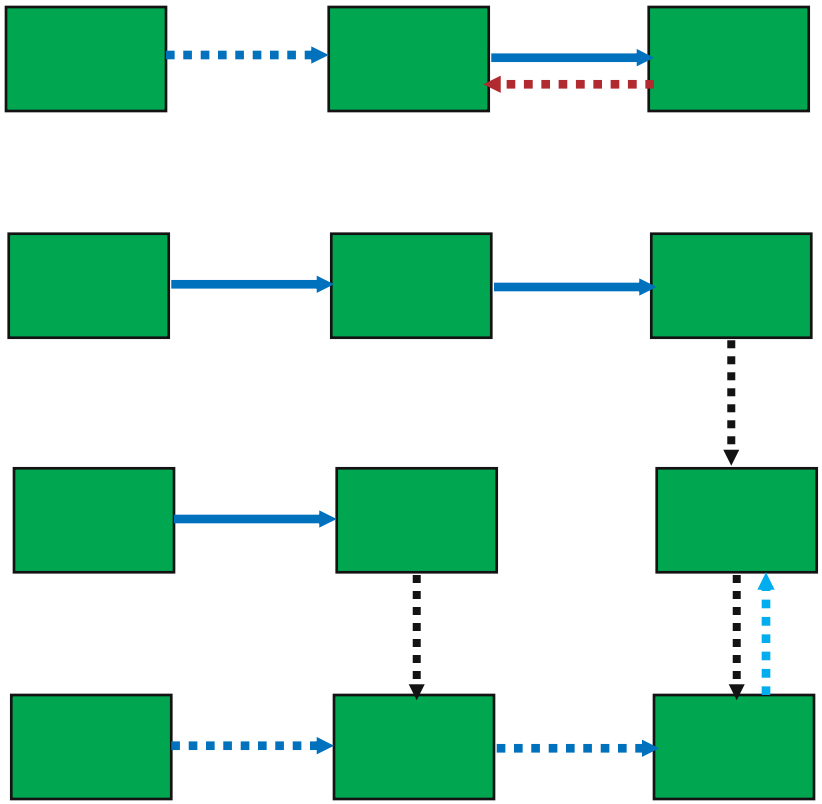
Mesh Exchange - Step 1

- Exchange data on a mesh



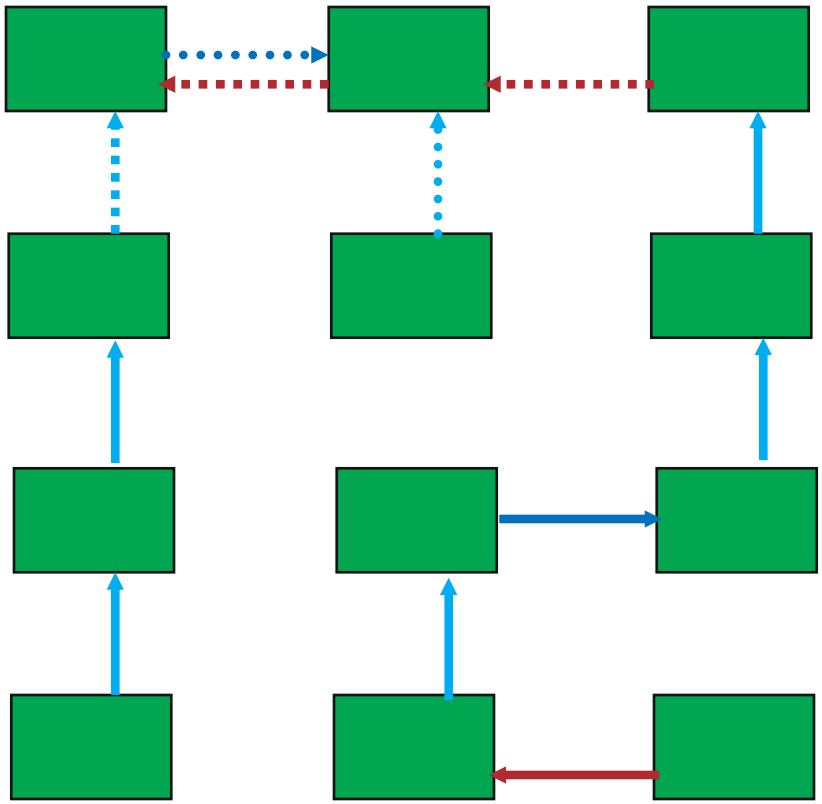
Mesh Exchange - Step 2

- Exchange data on a mesh



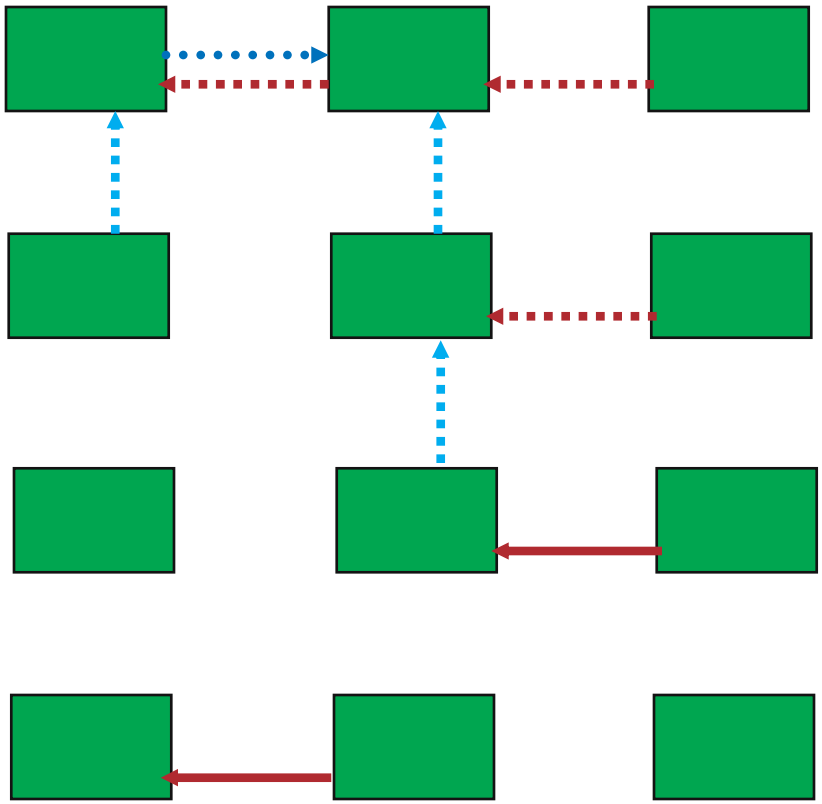
Mesh Exchange - Step 3

- Exchange data on a mesh



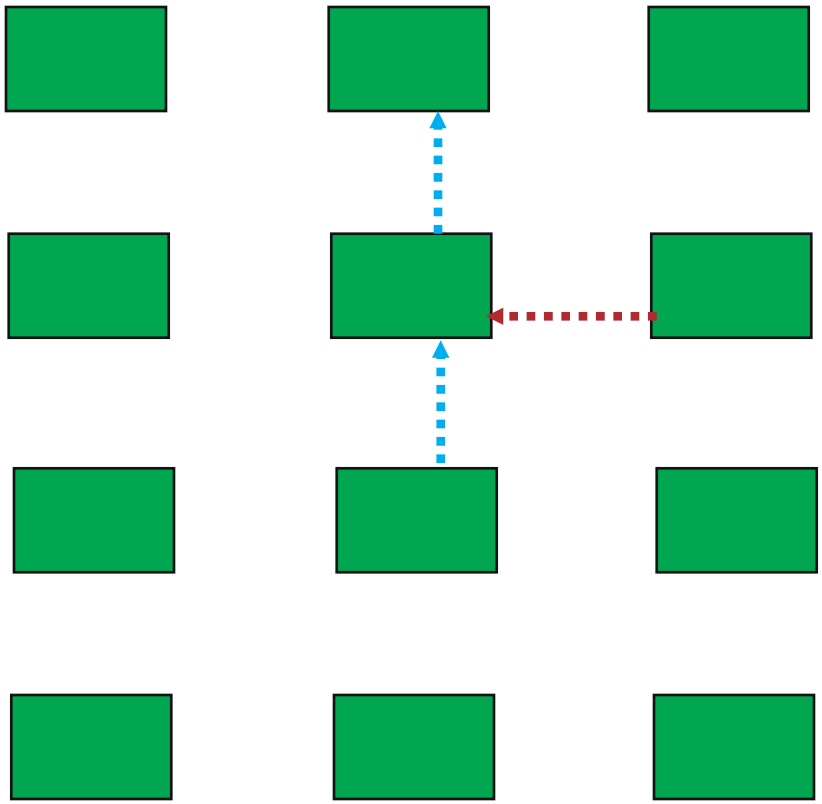
Mesh Exchange - Step 4

- Exchange data on a mesh



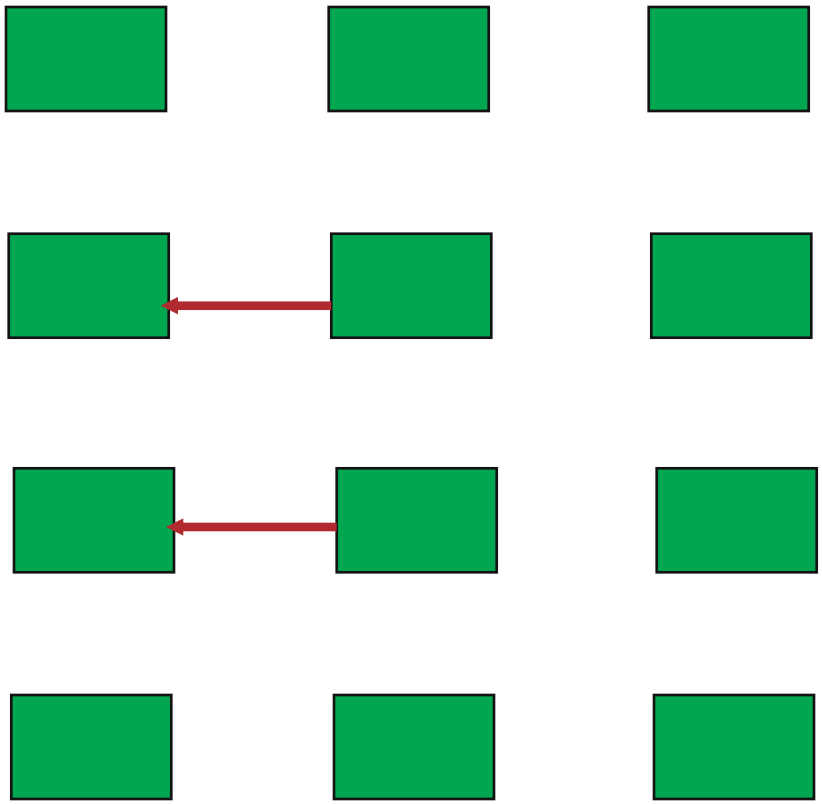
Mesh Exchange - Step 5

- Exchange data on a mesh

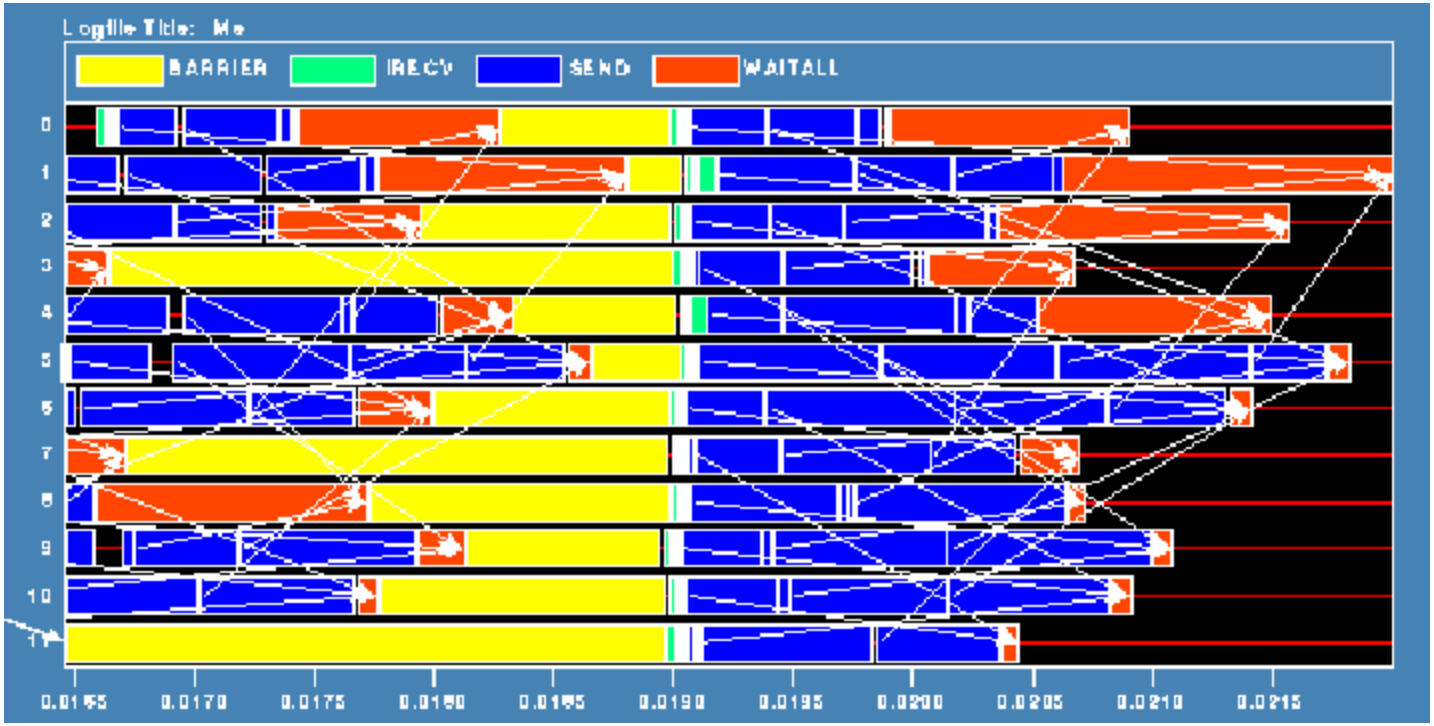


Mesh Exchange - Step 6

- Exchange data on a mesh

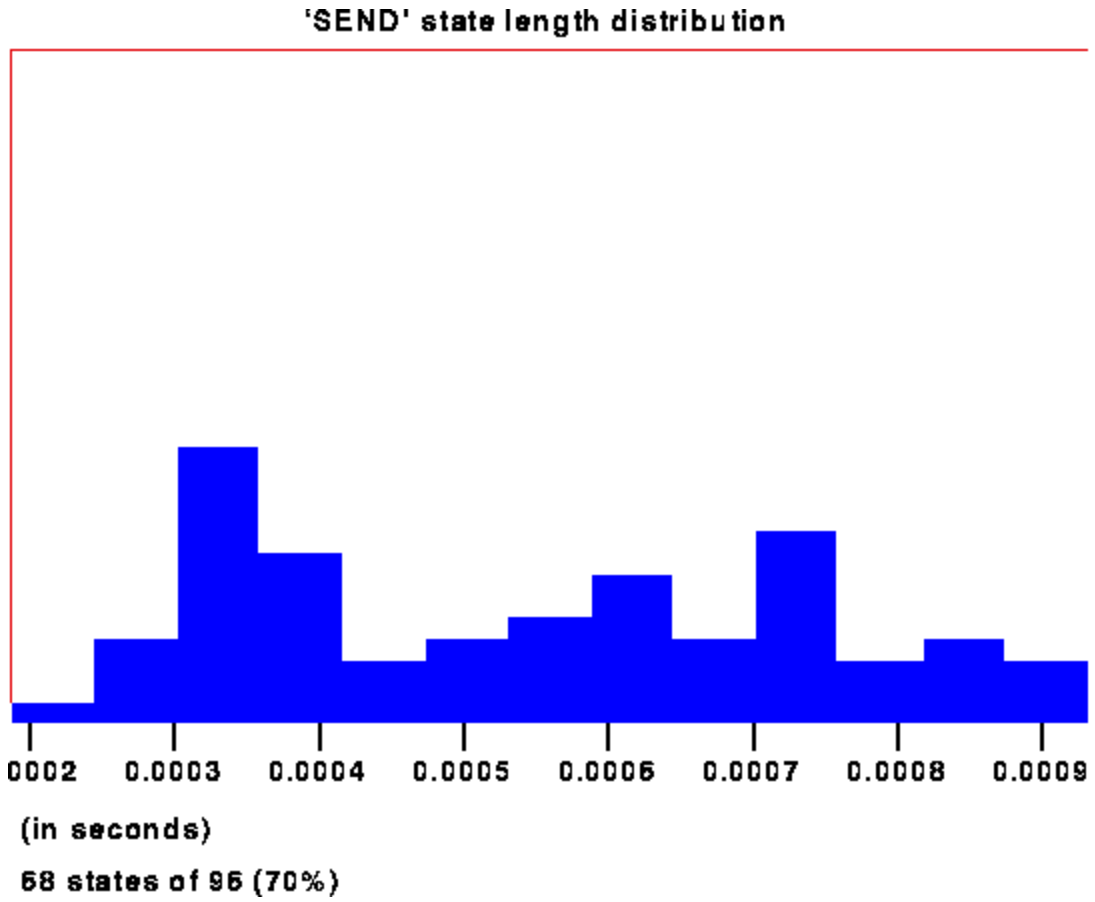


Timeline from IBM SP



- Note that process 1 finishes last, as predicted

Distribution of Sends



Why Six Steps?

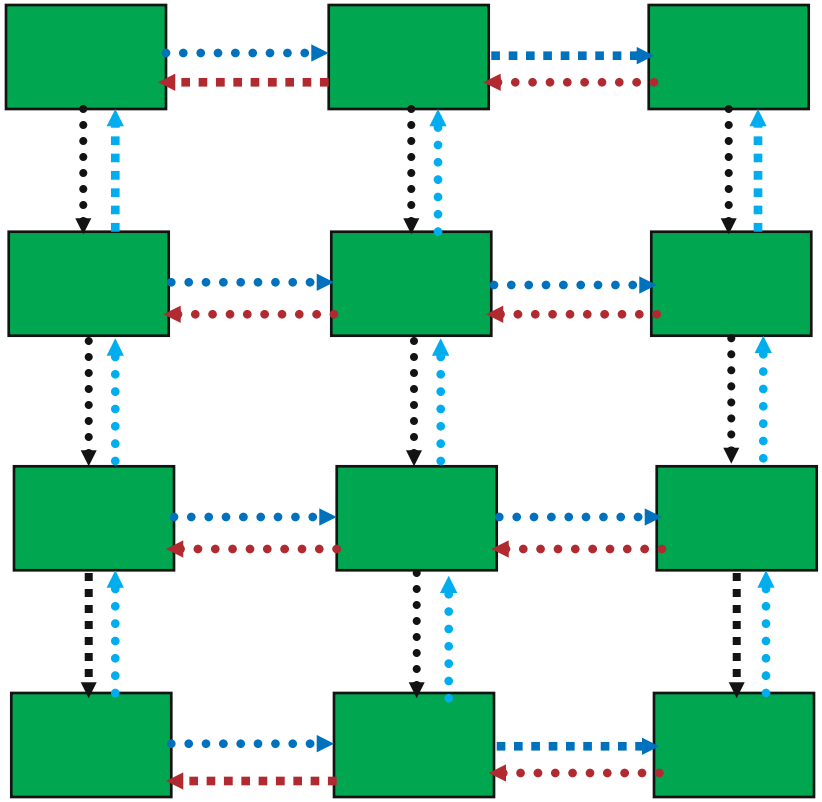
- Ordering of Sends introduces delays when there is contention at the receiver
- Takes roughly twice as long as it should
- Bandwidth is being wasted
- Same thing would happen if using memcpy and shared memory
- The interference of communication is why adding an MPI_Barrier (normally an unnecessary operation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

Fix 2: Use Isend and Irecv

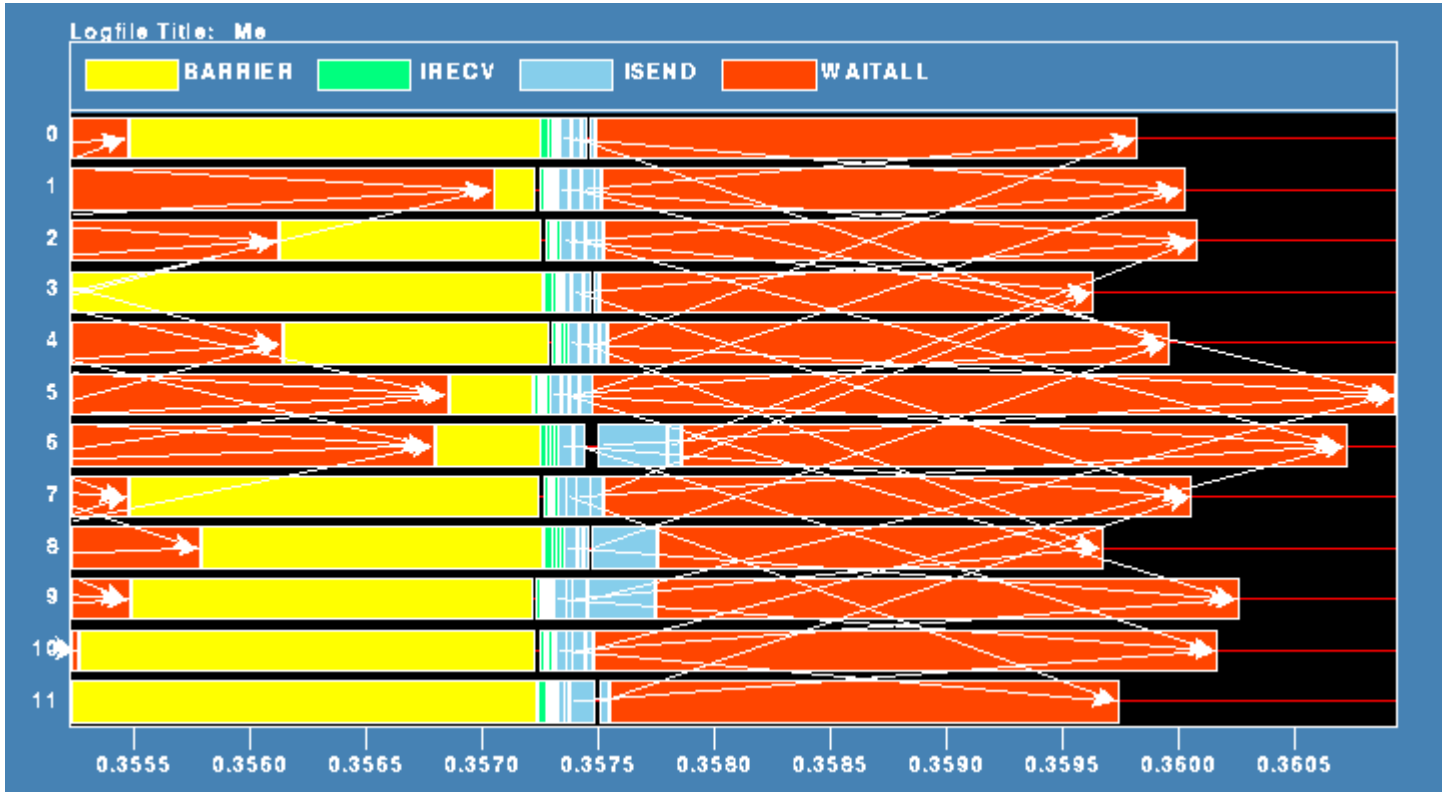
- Do i=1,n_neighbors
  Call MPI_Irecv(inedge(1,i),len,MPI_REAL,nbr(i),tag,&
  comm, requests(i),ierr)
  Enddo
  Do i=1,n_neighbors
  Call MPI_Isend(edge(1,i), len, MPI_REAL, nbr(i), tag,&
  comm, requests(n_neighbors+i), ierr)
  Enddo
  Call MPI_Waitall(2*n_neighbors, requests, statuses, ierr)

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)

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 process's 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:

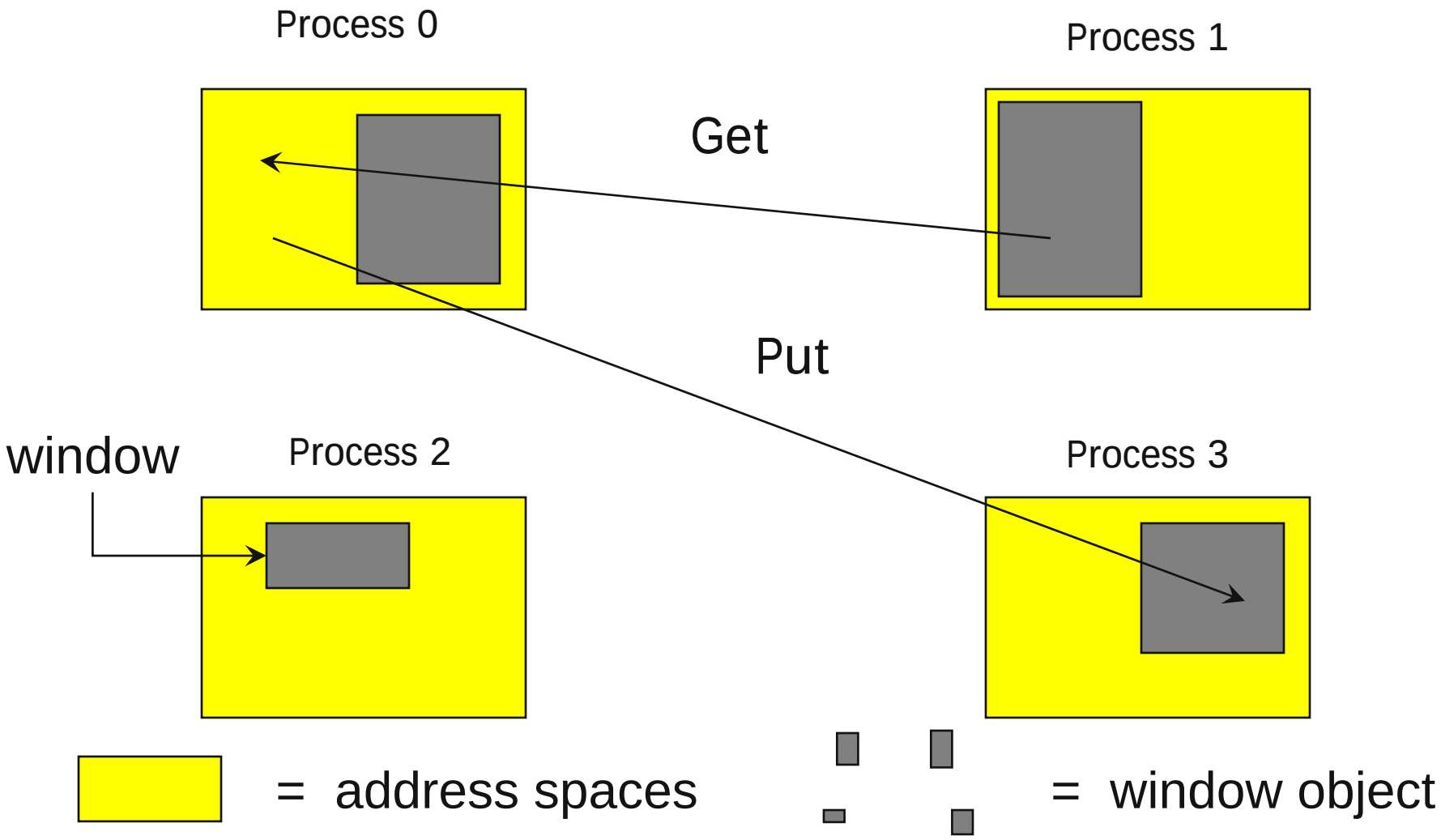
Proc 0	Proc 1
store	
send	receive
	load

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

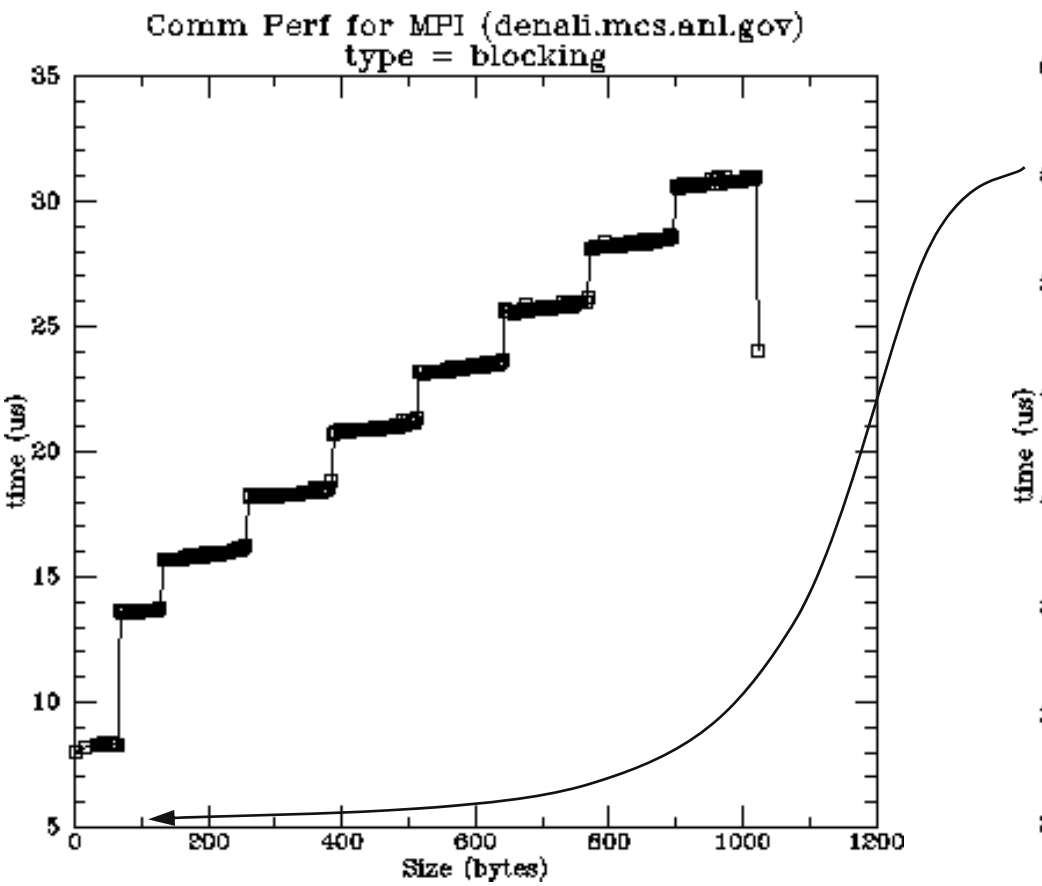


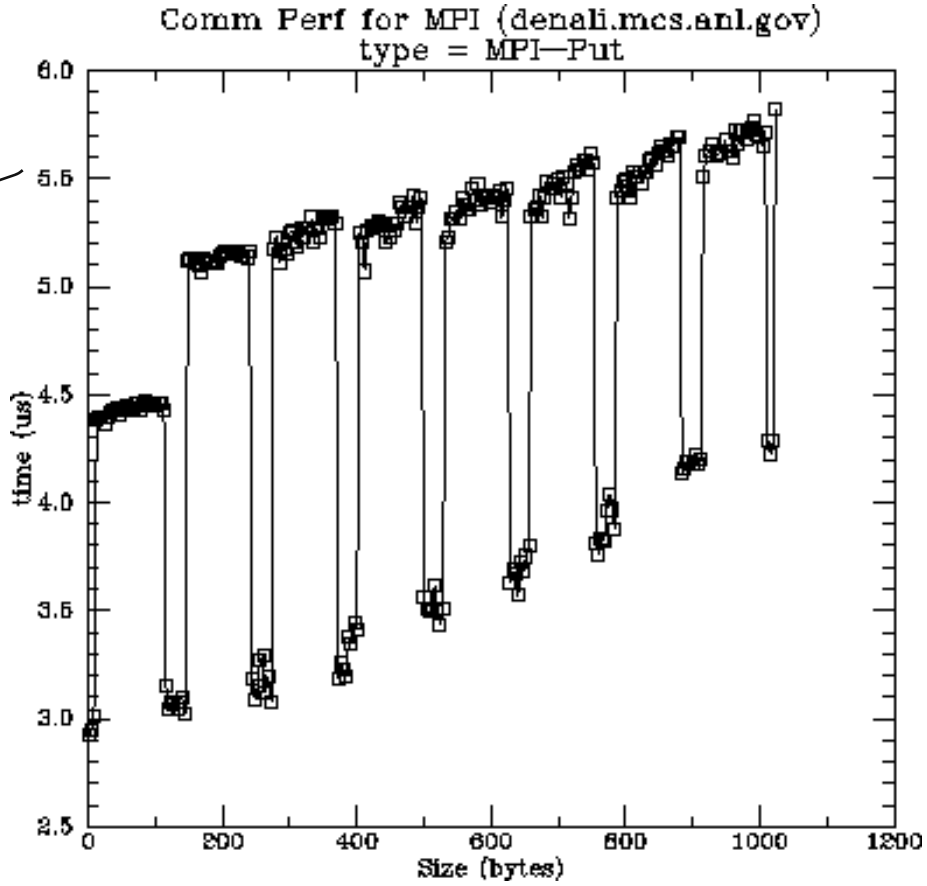
Basic RMA Functions for Communication

- **`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_Ge`**t 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

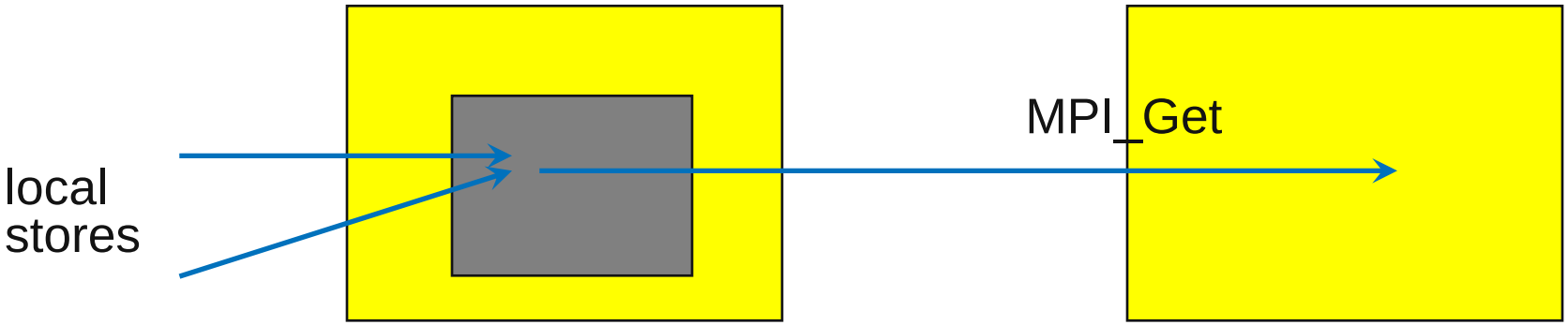
```
MPI_Win_create(base, size, disp_unit, info, comm, win)
```

- 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_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
```

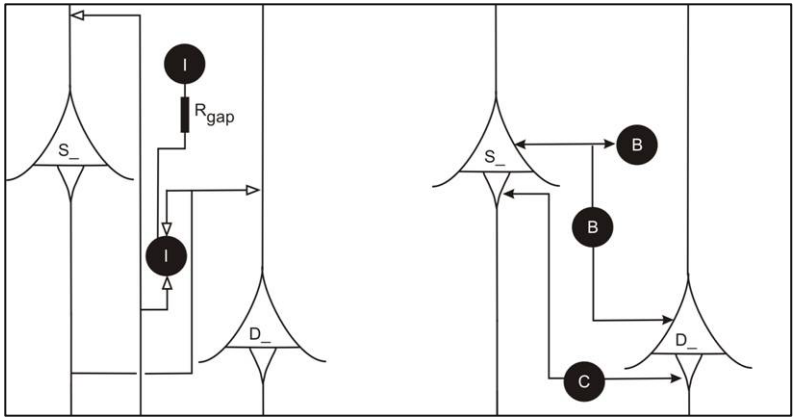
Fence vs Lock/Unlock Synchronization

- 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

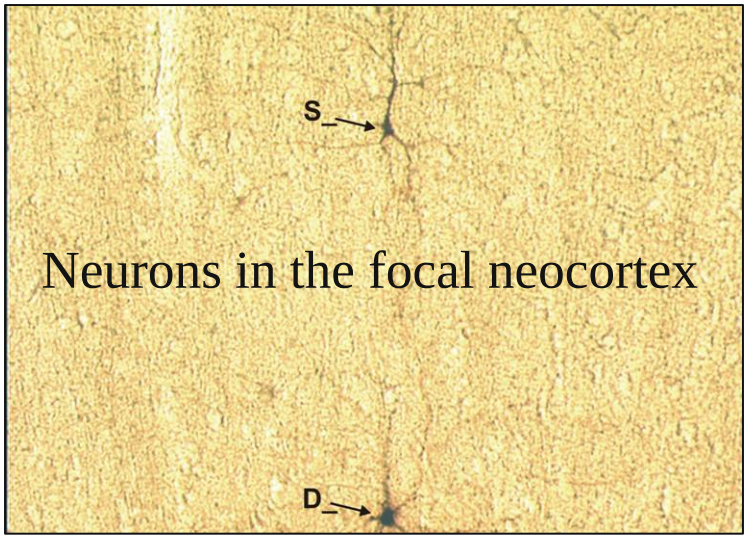
An Application: Modeling the Human Brain

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

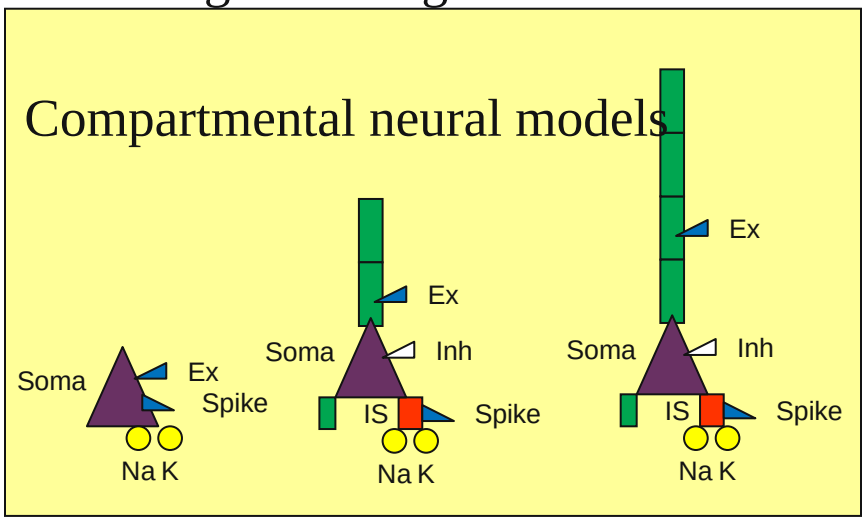
Model Neurons



Excitatory and inhibitory signal wiring between neurons

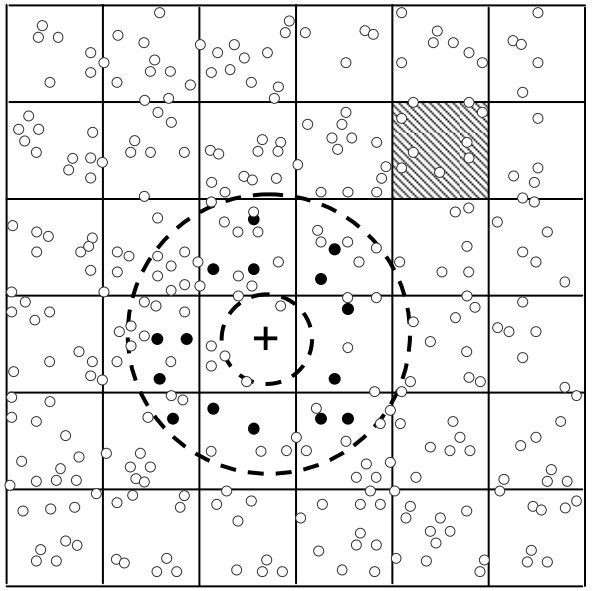


Neurons in the focal neocortex



Modeling Approach

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



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

Abstract pNeo for Tutorial Example

- "Simulate the simulation" of the evolution of neuron state instead of solving the differential equations
- Focus on how to code the interactions between cells in MPI
- Assume one cell per process for simplicity
  - Real code multiplexes many individual neurons onto one MPI process

What Happens In Real Life

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

What Happens In Existing pNeo Code

- In pNeo, each cell is connected to about 1000 cells
  - Large runs have 73,000 cells
  - Brain has ~100 billion cells
- Connections are derived from neuro-anatomical data
- There is a global clock marking time steps
- The state evolves according to a set of differential equations
- About 10 or more time steps between spikes
  - I.e., communication is unpredictable and sparse
- Possible MPI-1 solutions
  - Redundant communication of communication pattern before communication itself, to tell each process how many receives to do
  - Redundant “no spikes this time step” messages
- MPI-2 solution: straightforward use of Put, Fence

What Happens in Tutorial Example

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

Two Examples Using RMA in Pneo

- Global synchronization
  - Global synchronization of all processes at each step
  - Illustrates Put, Get, Fence
- Local synchronization
  - Synchronization across connected cells, for improved scalability (synchronization is local)
  - Illustrates Start, Complete, Post, Wait

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

Columbia 20

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

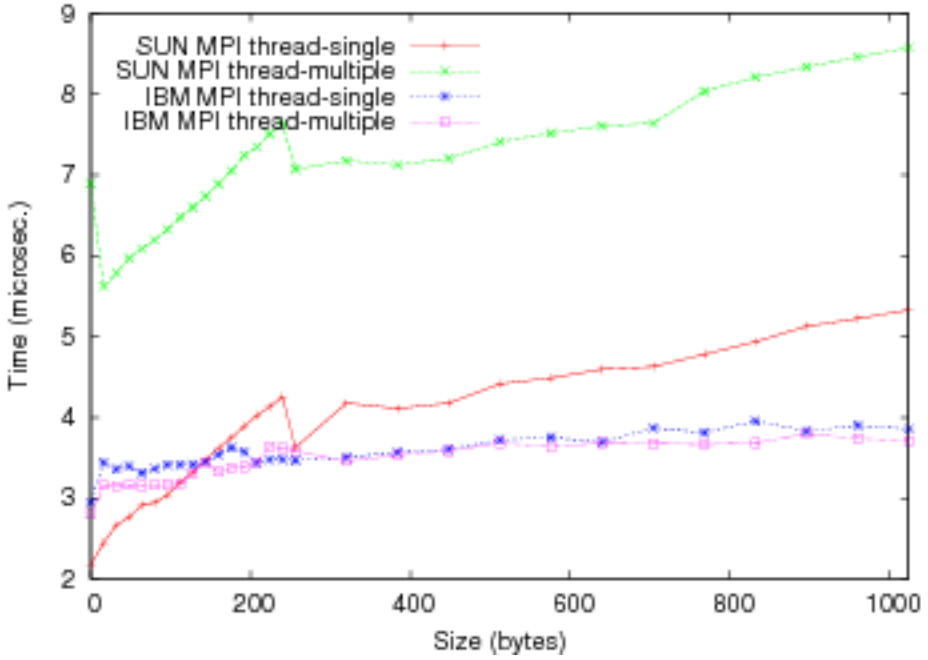
- MPI and Threads
  - MPI_Init_thread(&argc, &argv, requested, &provided)
  - The four levels of thread safety
    - *MPI_THREAD_SINGLE*
    - *MPI_THREAD_FUNNELED*
    - *MPI_THREAD_SERIAL*
    - *MPI_THREAD_MULTIPLE*
- Using threads
  - OpenMP
    - *Compiler handles most operations*
  - Pthreads
    - *Like MPI, you get to do everything yourself :)*
  - Limitations imposed by OS
    - *On BG/P, threads bound to cores (so four threads)*
    - *Linux will enable real thread programming (now true on Jaguar)*

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

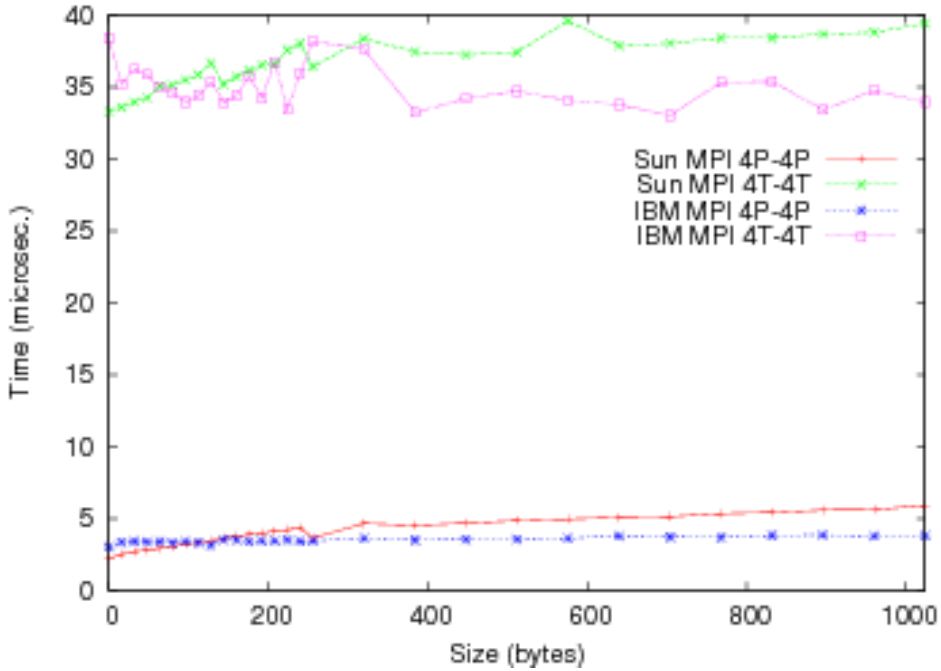
Overhead of Providing Thread Safety

- 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



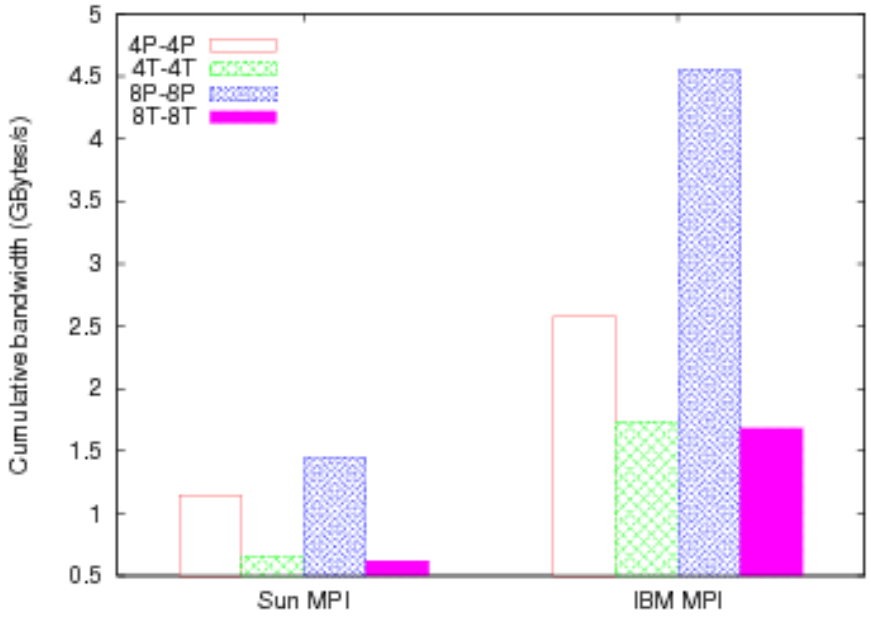
Thread Overhead

- 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.

Standards Issues

- Hybrid programming (two programming models) requires that the standards make commitments to each other on semantics.
- OpenMP's commitment: if a thread is blocked by an operating system call (e.g. file or network I/O), the other threads remain runnable.
  - This is a major commitment; it involves the thread scheduler in the OpenMP compiler's runtime system and interaction with the OS.
  - What this means in the MPI context: An MPI call like MPI_Recv or MPI_Wait only blocks the calling thread.
- MPI's commitments are more complex.

MPI's Four Levels of Thread Safety

- Note that these are not specific to OpenMP
- The are in the form of commitments that the multithreaded application makes to the MPI implementation
  - MPI_THREAD_SINGLE: there is only one thread in the application
  - MPI_THREAD_FUNNELED: there is only one thread that makes MPI calls
  - MPI_THREAD_SERIALIZED: Multiple threads make MPI calls, but only one at a time
  - MPI_THREAD_MULTIPLE: Any thread may make MPI calls at any time
- MPI-2 defines an alternative to MPI_Init
  - MPI_Init_thread(requested, provided)
    - Allows applications to say what level it needs, and the MPI implementation to say what it provides

What This Means in the 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…
}
```

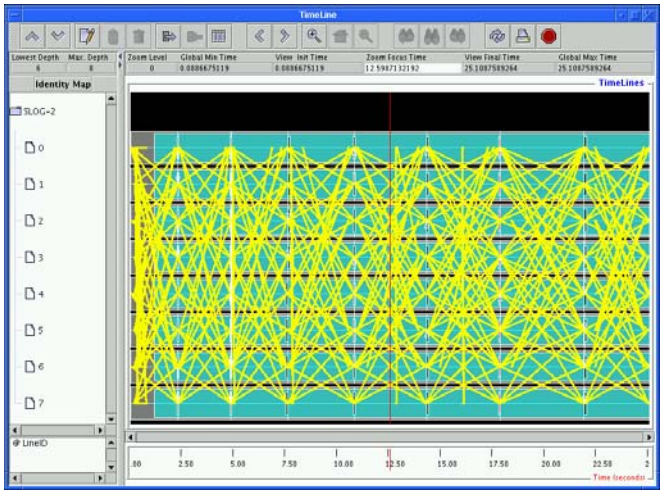
- MPI_THREAD_MULTIPLE
  - Anything goes

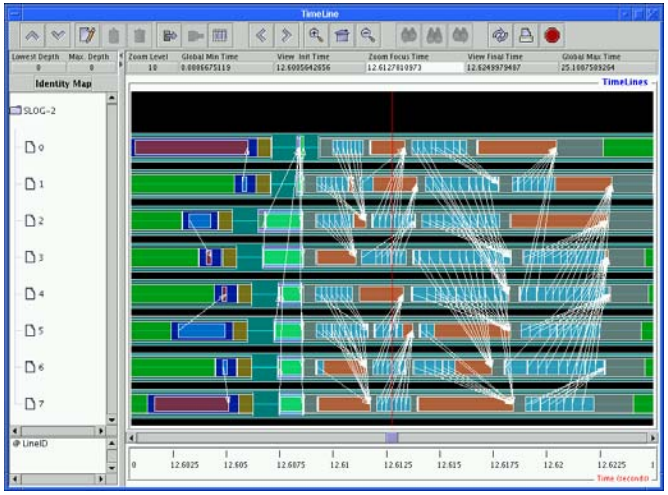
The Current Situation

- 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
- "Thread-safe" usually means MPI_THREAD_MULTIPLE.
- This is hard for MPI implementations that are sensitive to performance, like MPICH2.
  - 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!

Visualizing the Behavior of Hybrid Programs

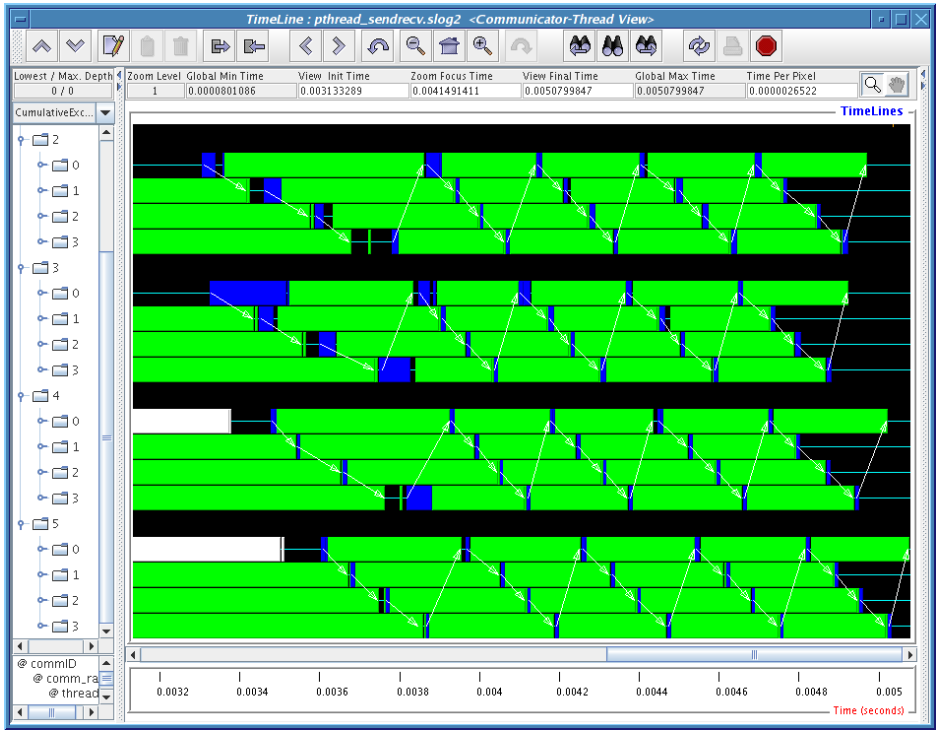
- Jumpshot is a logfile-based parallel program visualizer of the "standard" type. Uses MPI profiling interface.
- Recently it has been augmented in two ways to improve scalability.
  - Summary states and messages are shown as well as individual states and messages.
    - Provides a high-level view of a long run.
    - SLOG2 logfile structure allows fast interactive access (jumping, scrolling, and zooming) for large logfiles.





Jumpshot and Multithreading

- Newest additions are 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 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 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

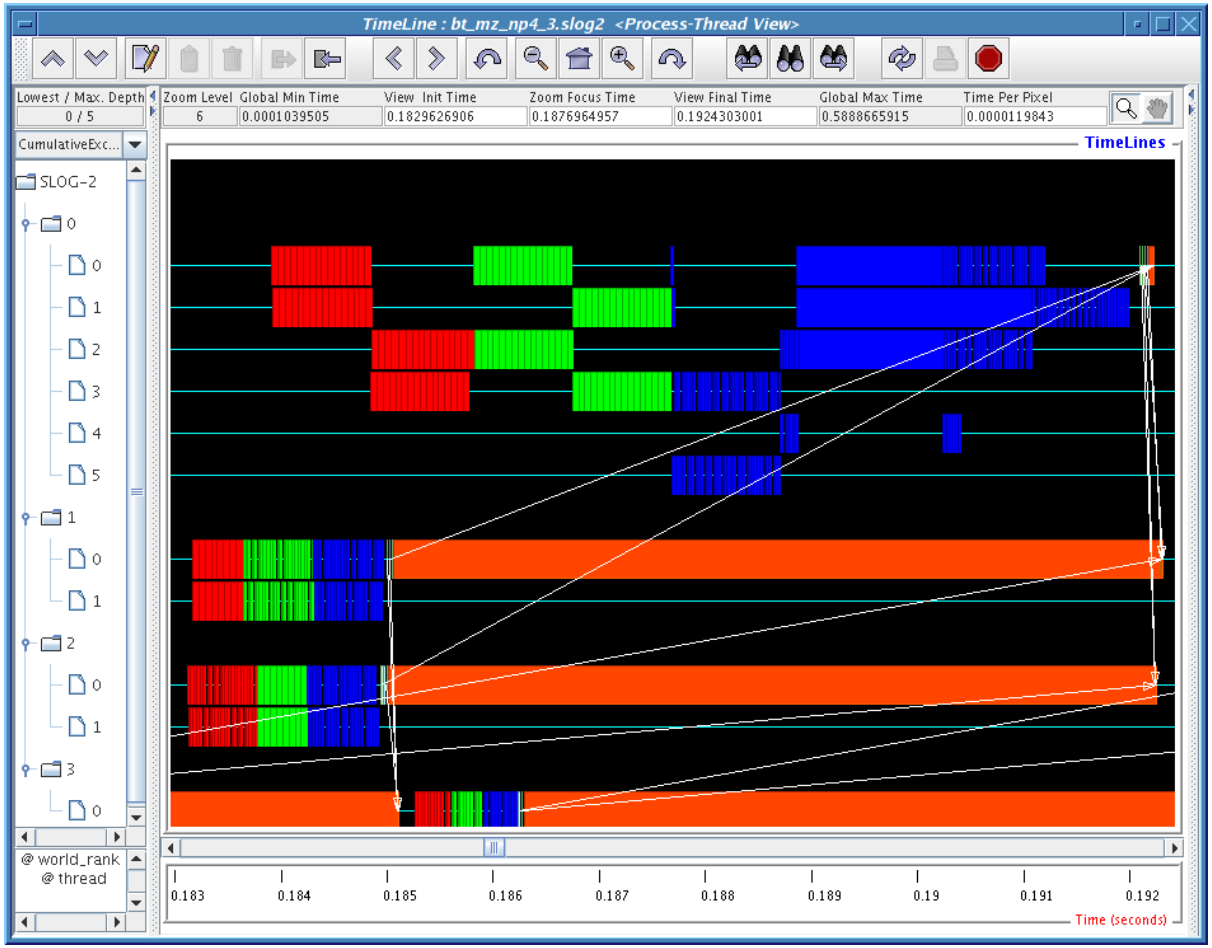
- Linux cluster
  - 24 nodes, each consisting of two Opteron dual-core processors, 2.8 Ghz each
  - Intel 9.1 fortran compiler
  - MPICH2-1.0.6, which has MPI_THREAD_MULTIPLE
  - Multiple networks; we used GigE
- IBM Blue Gene/P
  - 40,960 nodes, each consisting of four PowerPC 850 MHz cores
  - XLF 11.1 Fortran cross-compiler
  - IBM's MPI V1R1M2 (based on MPICH2), which 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
  - Funky 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
- But we didn't get it right the first time…

It Might Not Be Doing What You Think

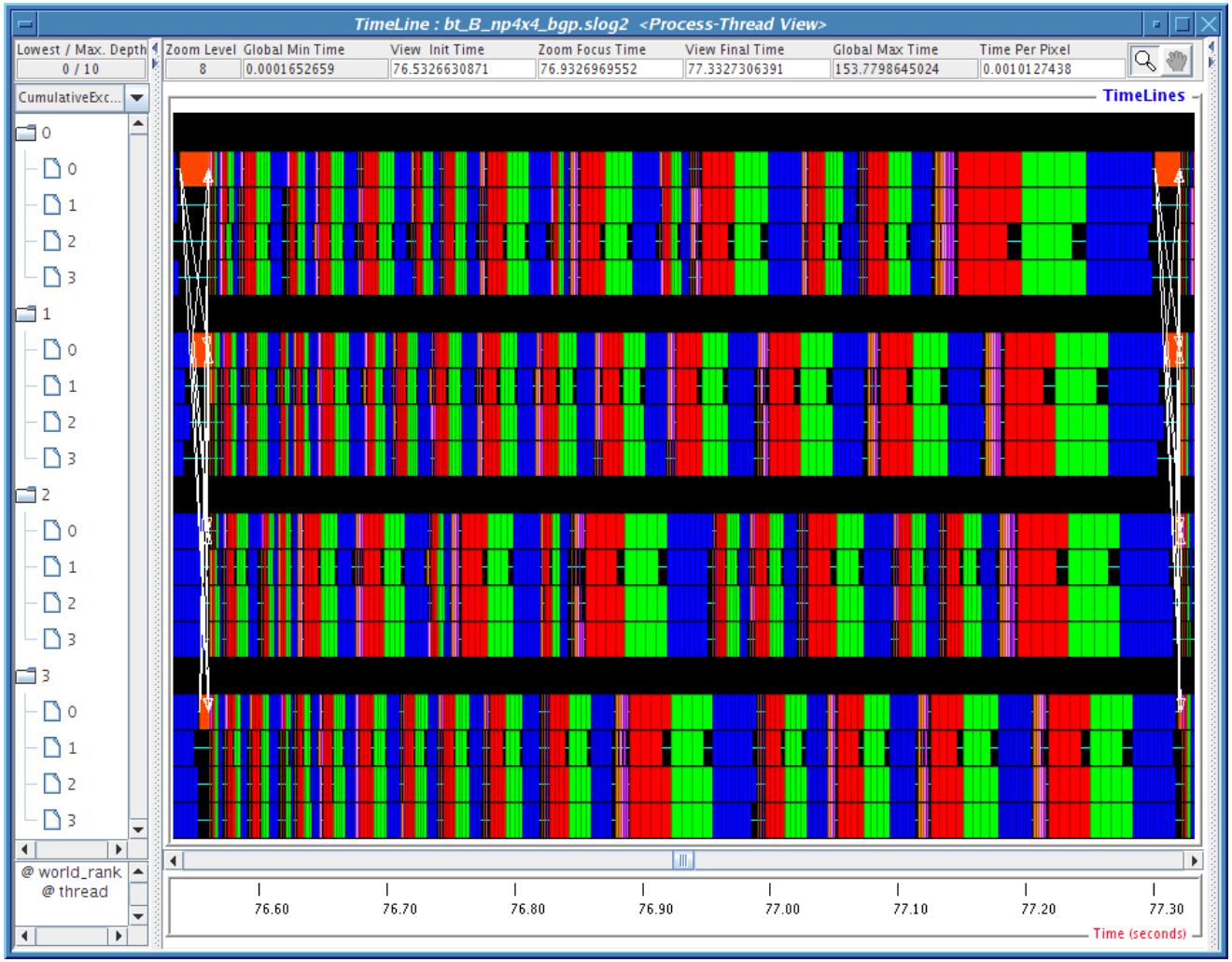
- An early run:



- Nasty interaction between the environment variables OMP_NUM_THREADS and NPB_MAX_THREADS

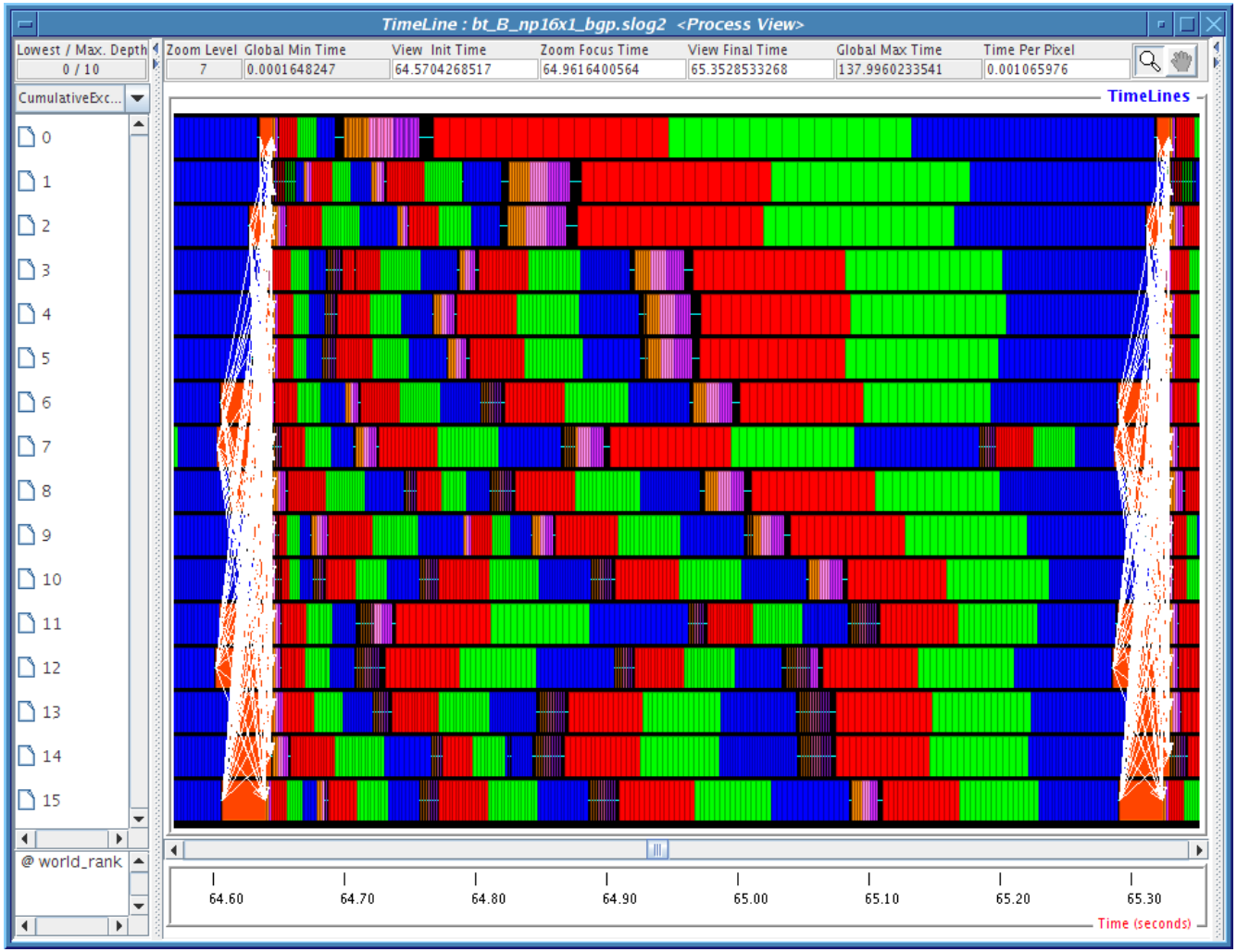
More Like What You Expect

- 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 Small 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

- 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

Conclusions

- 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 (Hybriditude?) 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.
- (This work is reported on in the Proceedings of EWOMP ‘08.)

The OpenMP Books

- Both old book and new book (and OpenMP tutorials) have two parts:
- The Front:
  - OpenMP is magically convenient
  - A few comments added to serial code and *voila!*
- The Back:
  - Scalability is not so easy
  - Performance issues are subtle
  - Need more than comments – function calls
  - SPMD structure
  - It starts to look like MPI…
- (But the new book is really good; both front and back parts.)

What is OpenMP For?

- Word on laptop? Probably not.
- Matlab back end on desktop? Probably so.
- Small scientific applications on desktops? Yes.
- HPC on biggish SMPs? Maybe, but hard.
- Collaborating with MPI on big machines? Almost certainly.

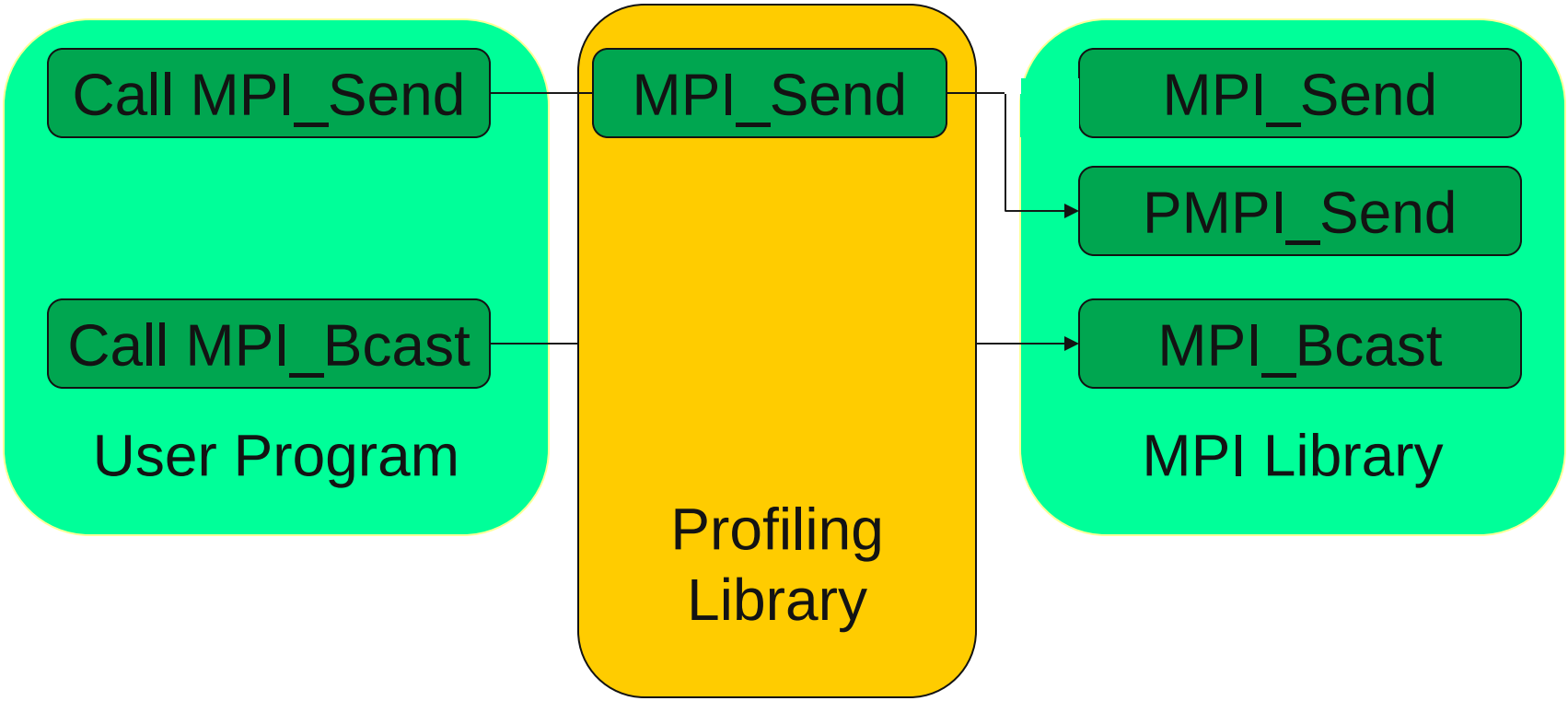
Solving Performance Problems

- 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

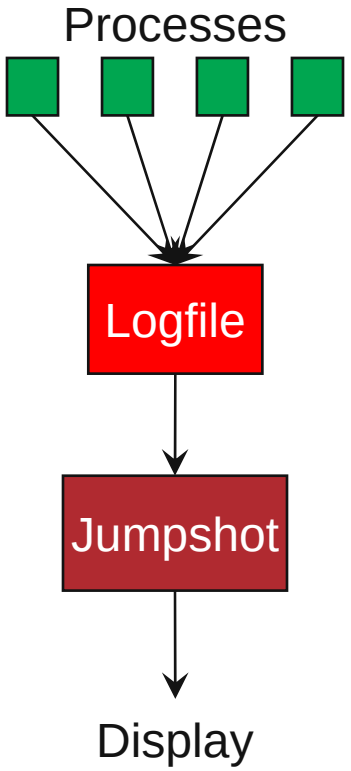
- The MPI profiling interface: how it works
- Some freely available tools
  - Those to be presented in other talks
  - A few that come with MPICH2
    - *SLOG/Jumpshot: visualization of detailed timelines*
    - *FPMPI: summary statistics*
    - *Collcheck: runtime checking of consistency in use of collective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

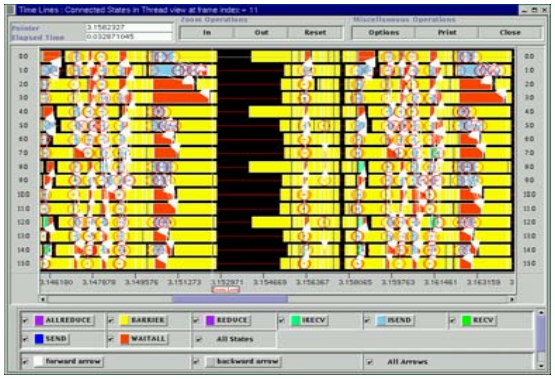
The MPI Profiling Interface

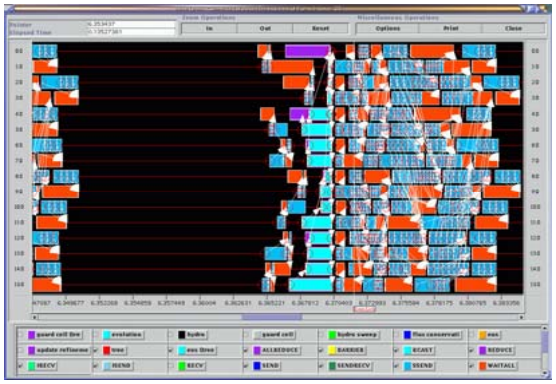


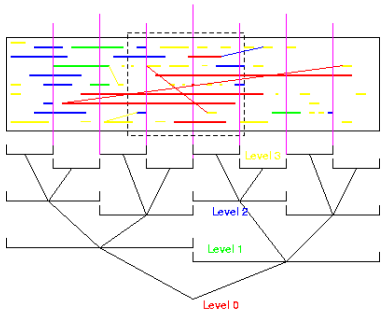
Performance Visualization with Jumpshot

- 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.

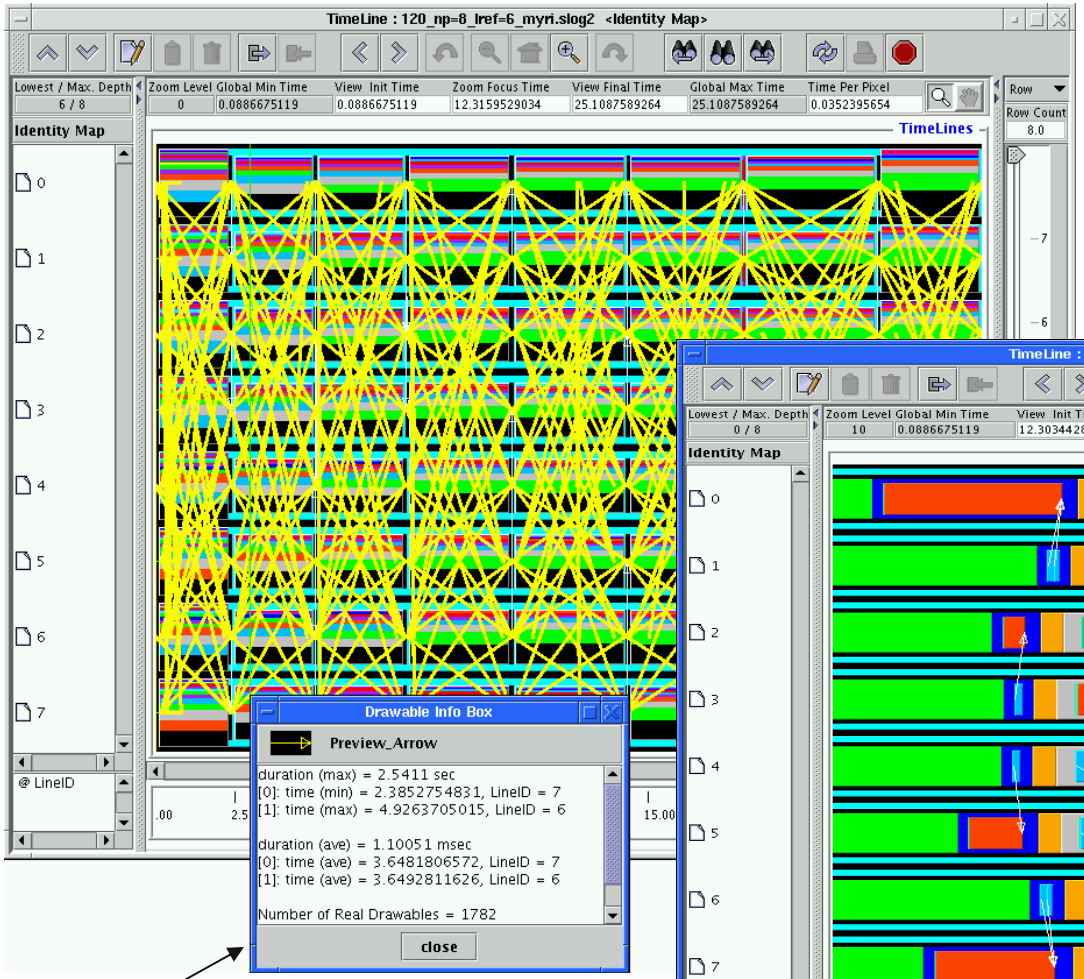






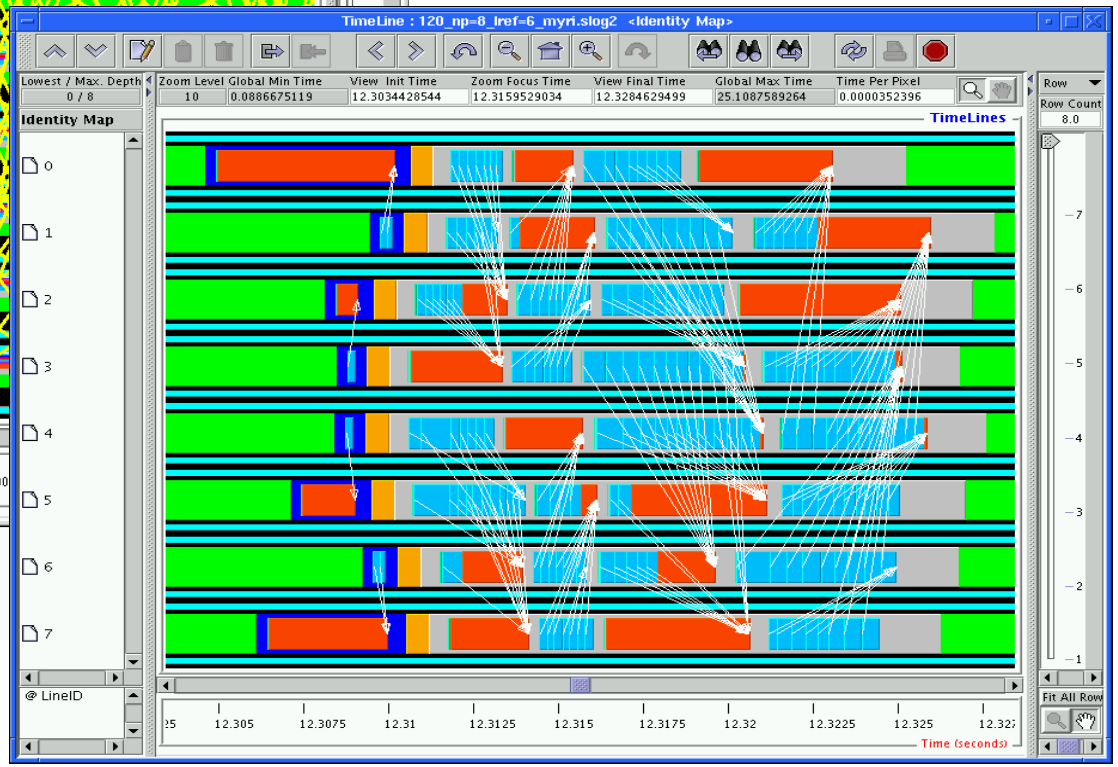


Viewing Multiple Scales



Each line represents 1000's of messages

Detailed view shows opportunities for optimization



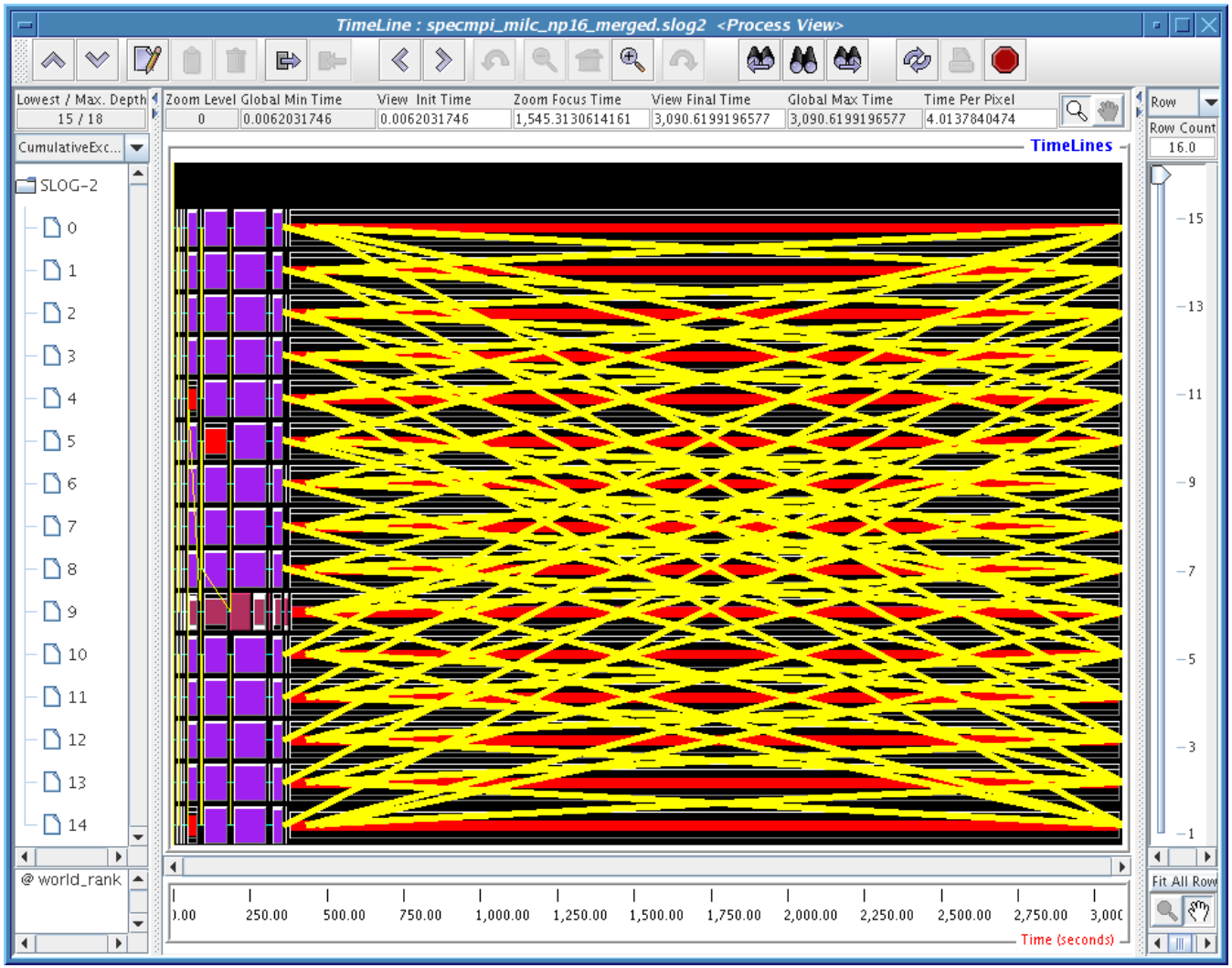
1000x zoom

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*

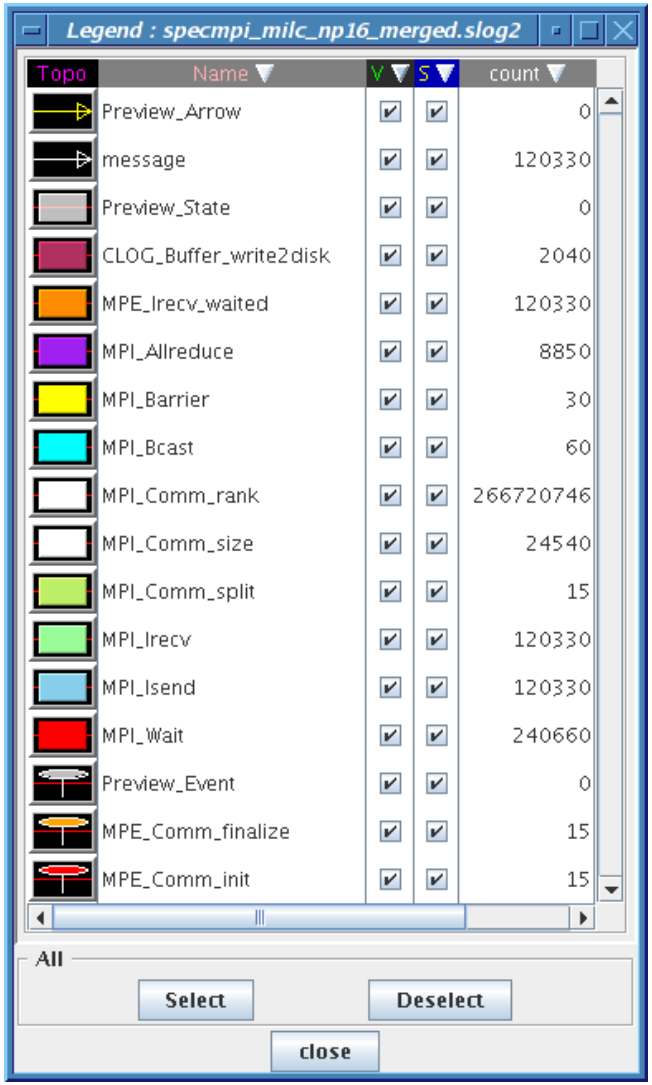
Looking at MILC in SPEC2007

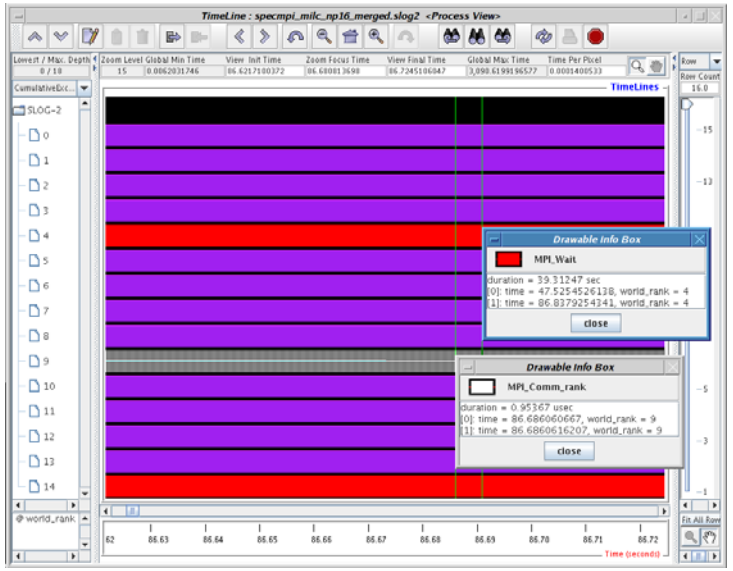
- Curious amount of All_reduce in initialization - why?

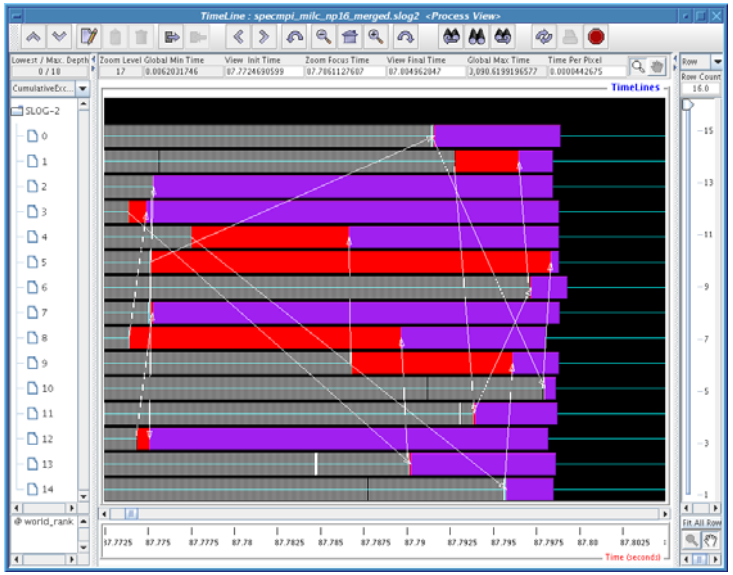


MILC

- 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 on 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.

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 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`**

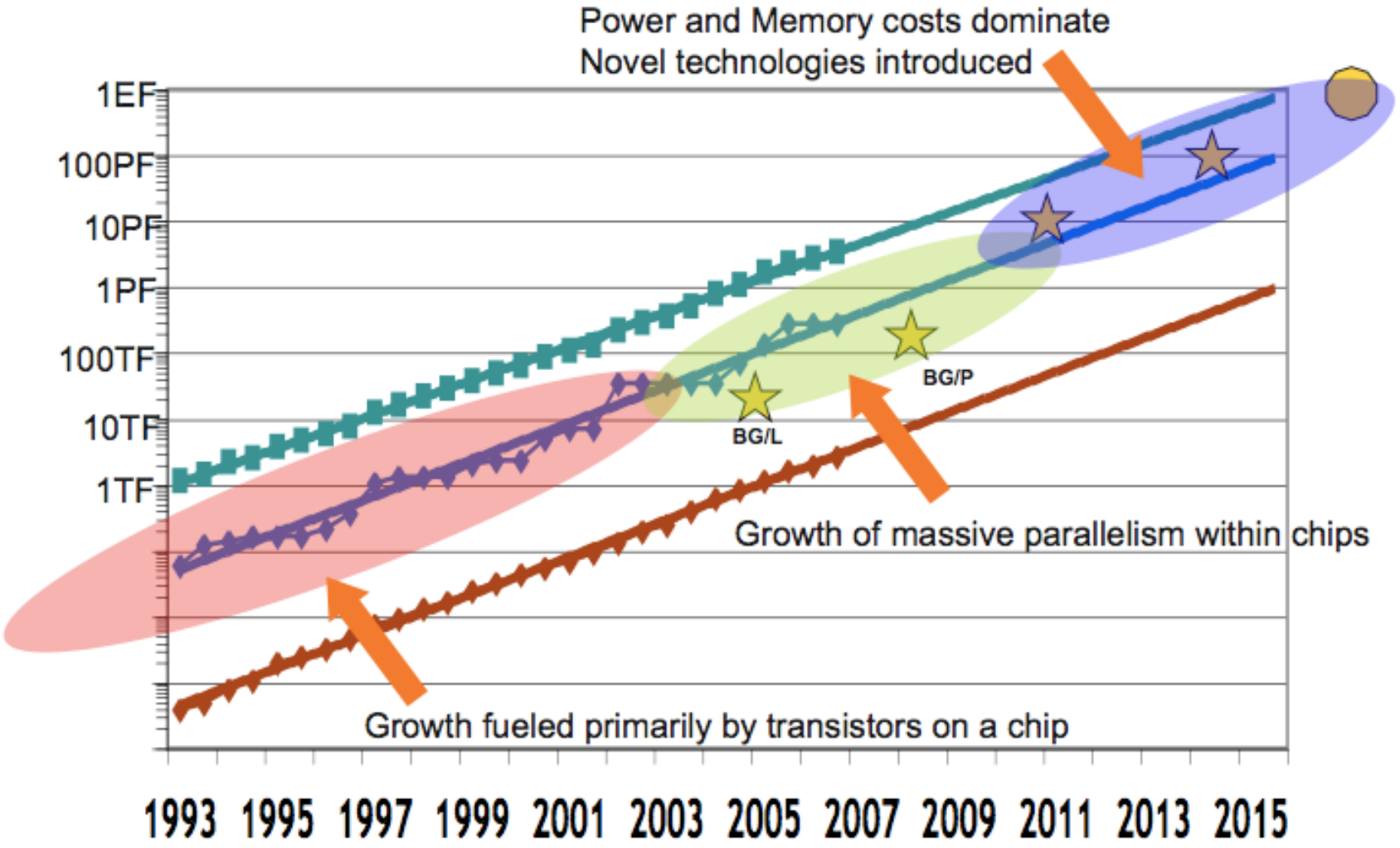
- **`Validate Barrier (rank 4) – collective call (Barrier) is inconsistent with Rank 0's (Bcast)`**

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

The (Foggy) Future of Programming Models

Looking out to Exascale…
Concurrency will be Doubling every 18 months



Outline of the Situation

- Million core systems and beyond are on the horizon
- Today labs and universities have general purpose systems with 10k-200K cores (BGL@ LLNL 200K, BGP@Argonne 160K, XT5@ORNL 150K cores)
- By 2012 there will be more systems deployed in the 200K-1M core range
- By 2020 there will be systems with perhaps 100M cores
- Personal systems with > 1000 cores within 5
- Personal systems with requirement for 1M threads is not too far fetched (GPUs for example)

How Will We Program Them?

- Still an unsolved problem
- Some believe a totally new programming model and language (e.g. X10, Chapel, Fortress).
- Some mechanism for dealing with shared memory will be necessary
  - This (whatever it is) plus MPI is the conservative view
- Whatever it is, it will need to interact properly with MPI
- May also need to deal with on-node heterogeneity
- The situation is somewhat like message-passing before MPI
  - And it is too early to standardize

MPI is Current HPC Programming Model

- MPI represents a very complete definition of a well-defined programming model
- MPI programs are *portable*
- There are many implementations
  - Vendors
  - Open source
- Enables high performance for wide class of architectures
  - Scalable algorithms are key
- Small subset easy to learn and use
- Expert MPI programmers needed most for libraries, which are encouraged by the MPI design.

The MPI Forum Continues to Refresh MPI

- New signatures for old functions
  - E.g. MPI_Send(…,MPI_Count,…)
- Details
  - Fortran binding issues..
- New features
  - MPI_Process_Group and related functions for fault tolerance
  - New topology routines aware of more hierarchy levels
  - Non-blocking collective operations
  - A simpler one-sided communication interface
    - *Or perhaps standardized semantics for interacting with shared-memory programming systems in general*
  - More scalable versions of the “v” collectives
  - …
- See http://www.mpi-forum.org for details of working groups

Why Won't "MPI Everywhere" suffice?

- Core count on a node is increasing faster than memory size.
- Thus memory available per MPI process is going down.
- Thus we need parallelism *within* an address space, while continuing to use MPI for parallelism among separate address spaces.
- We don't have a good way to do this yet.
- Whatever we use, it must cooperate with parallelism across address spaces, so its API must interact in a well-defined way with MPI.
- Some applications are expressing the need for large address spaces that span multiple multi-core nodes, yet still are each a small part of the memory of the entire machine.

Traditional Sources of Performance Improvement are Flat-Lining (2004)

- New Constraints
  - 15 years of *exponential* clock rate growth has ended
- Moore's Law reinterpreted:
  - How do we use all of those transistors to keep performance increasing at historical rates?
  - Industry Response: #cores per chip doubles every 18 months *instead* of clock frequency!

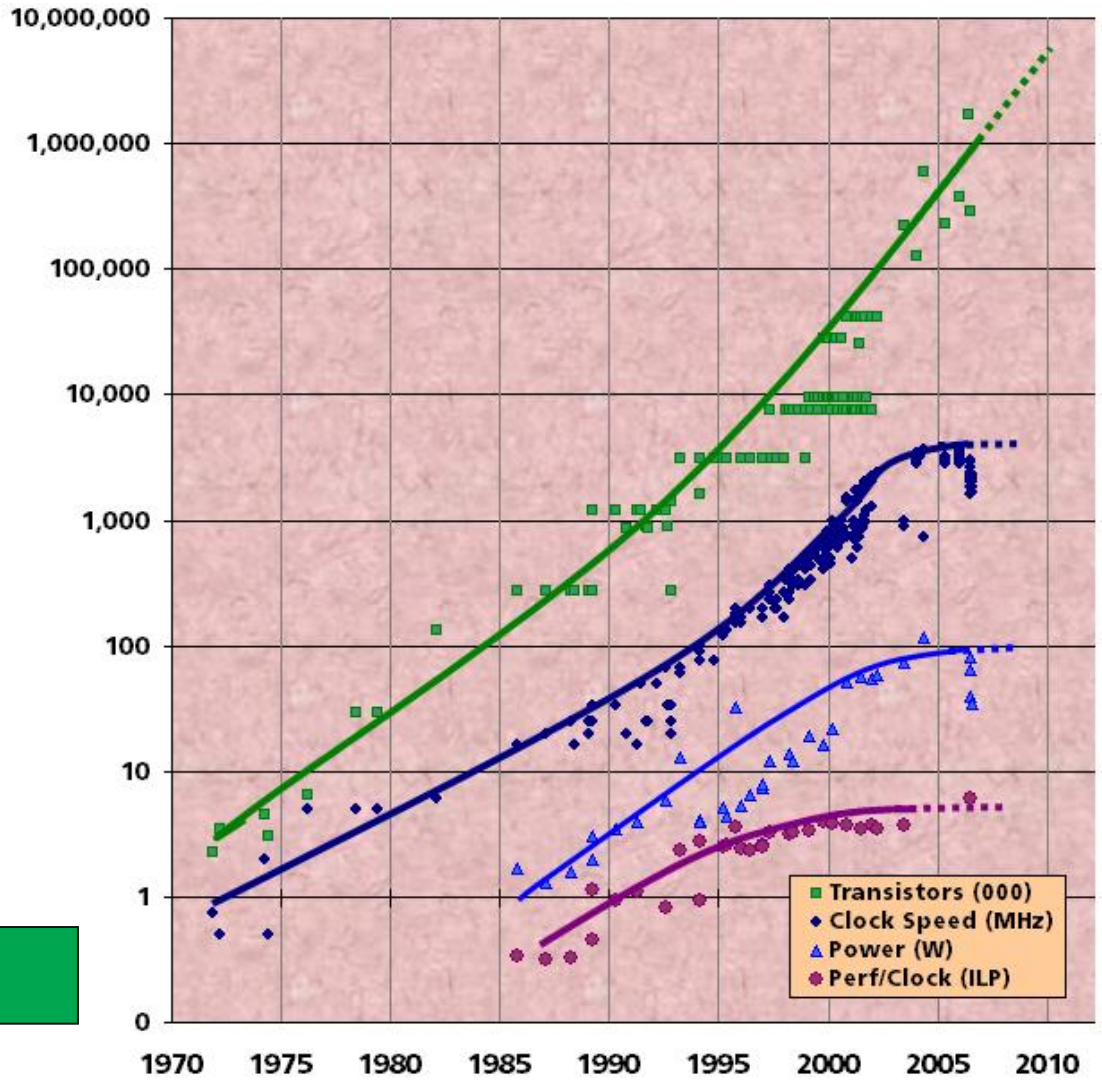
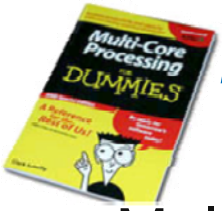
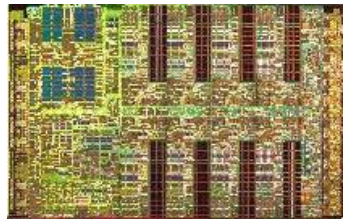


Figure courtesy of Kunle Olukotun, Lance Hammond, Herb Sutter, and Burton Smith

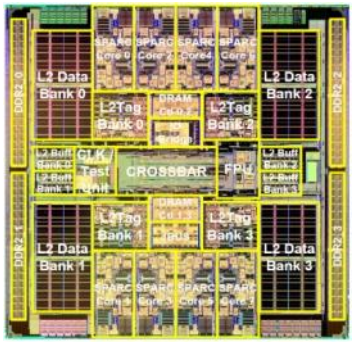


Multicore comes in a wide variety

- Multiple parallel general-purpose processors (GPPs)
- Multiple application-specific processors (ASPs)

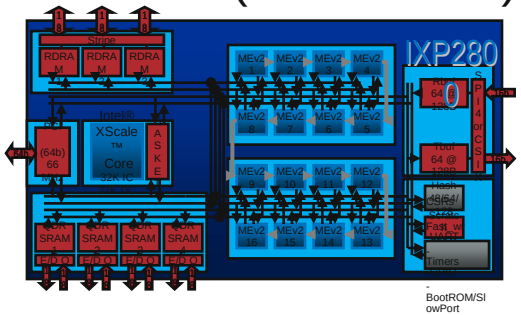


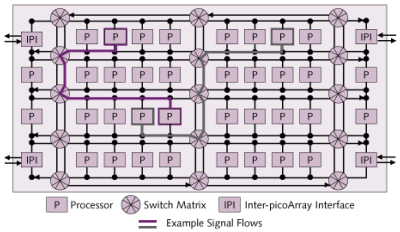
IBM Cell
1 GPP (2 threads)
8 ASPs



Sun Niagara
8 GPP cores (32 threads)

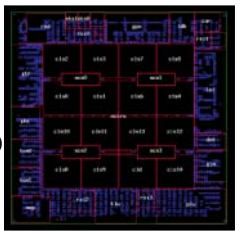
Intel Network Processor
1 GPP Core
16 ASPs (128 threads)

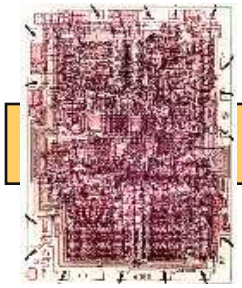




Picochip DSP
1 GPP core
248 ASPs

Cisco CRS-1
188 Tensilica GPP





Intel 4004 (1971):
4-bit processor,
2312 transistors,
~~~100 KIPS,~~
10 micron PMOS,
11 mm^2 chip

"The Processor is the new Transistor" [Rowen]
~~~

Moving Beyond MPI

- Any alternative to MPI (at its own level) will have to have some of the good properties of MPI
  - Portability
  - Scalability
  - Performance
- Perhaps alternatives exist at different levels.
- But they will still have to interact with MPI, in order to provide a path from where we are now to more abstract models
  - Clear interoperability semantics
  - Can be used either above or below C/Fortran/MPI code

Some Families of Programming Models and Associated Languages

- Shared-memory and annotation languages
  - Especially OpenMP
  - Likely to coexist with MPI
- Partitioned Global Address Space Languages
  - UPC, Co-Array Fortran, and Titanium
  - One step removed from MPI
- The HPCS languages
  - X10, Chapel, Fortress
  - Two steps removed from MPI

OpenMP

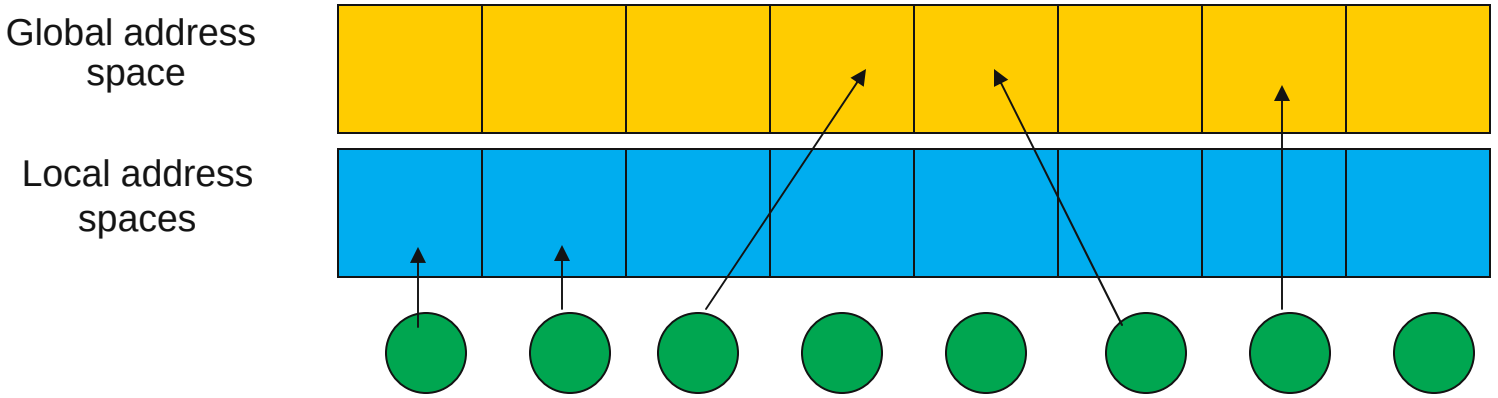
- OpenMP is a set of compiler directives (in comments, like HPF) plus some library calls
- The comments direct the execution of loops in parallel in a convenient way.
- Data placement is not controlled, so performance is hard to get except on machines with real shared memory.
- Likely to be more successful on multicore chips than on previous SMP's (multicore = really, *really* shared memory).
- Can co-exist with MPI
  - MPI's levels of thread safety correspond to programming constructs in OpenMP
    - *Formal methods can be applied to hybrid programs*
- New book by Barbara Chapman, et al.

Other Annotation-based approaches

- The idea is to retain the sequential programming model
- Annotations guide source-to-source transformations or compilation into a parallel program
- HPF and OpenMP (part 1) are examples
- Others in research mode

The PGAS Languages

- PGAS (Partitioned Global Address Space) languages attempt to combine the convenience of the global view of data with awareness of data locality, for performance
  - Co-Array Fortran, an extension to Fortran-90)
  - UPC (Unified Parallel C), an extension to C
  - Titanium, a parallel version of Java



- Fixed number of processes, like MPI-1

Status

- Compilers exist
  - In some cases more than one
- Applications are being tried
- Substantial support, at least for UPC
- Early experiments are encouraging with respect to performance
  - Some reports are misleading.

The DARPA HPCS Language Project

- The DARPA High Productivity Computer Systems (HPCS) Project is a 10-year, three-phase, hardware/software effort to transform the productivity aspect of the HPC enterprise.
- In Phase II,three vendors were funded to develop high productivity language systems, and each assigned a small group to language development
  - IBM: X10
  - Cray: Chapel
  - Sun: Fortress
- In Phase III, Sun was dropped from DARPA support. Both IBM and Cray efforts are continuing. Actually, Sun's effort is too, internally supported.

The Transition is Starting

- In large-scale scientific computing today essentially all codes are message-passing based. Additionally many are starting to use some form of multithreading on SMP or multicore nodes.
- Multicore is challenging programming models but there has not yet emerged a dominate model to augment message passing
- There is a need to identify new hierarchical programming models that will be stable over long term and can support the concurrency doubling pressure
- Current approaches to programming GPU's are for library developers, not application developers

Hybrid Programming Models

- Some shared-memory API's that can be used with MPI
  - POSIX threads -- explicit thread creation, locks, condition vars
  - OpenMP
    - Sequential programming model with annotations, parallel execution model
  - Yet to be invented…
- The current situation: OpenMP + MPI
  - Works because of well-thought-out explicit contracts between the models.
    - MPI standard defines levels of thread safety
    - OpenMP defines types of code regions
    - These work together in ways defined by the respective standards
  - Hard to get performance with OpenMP because of lack of locality management, excessive synchronization.

One Possible Near Future: PGAS+MPI

- Locality management within an address space via local, remote memory
- An address space could be bigger than one node
  - Might need more hierarchy in PGAS definitions
- Just starting to work with PGAS folks on UPC+MPI and CAF+MPI
  - Center for Programming Models base program project with ANL, LBNL, Rice, Houston, PNNL, OSU
- Until recently PGAS has focused either on competing with MPI or with OpenMP on single node
  - Need to make interoperability with MPI a priority to attract current HPC applications

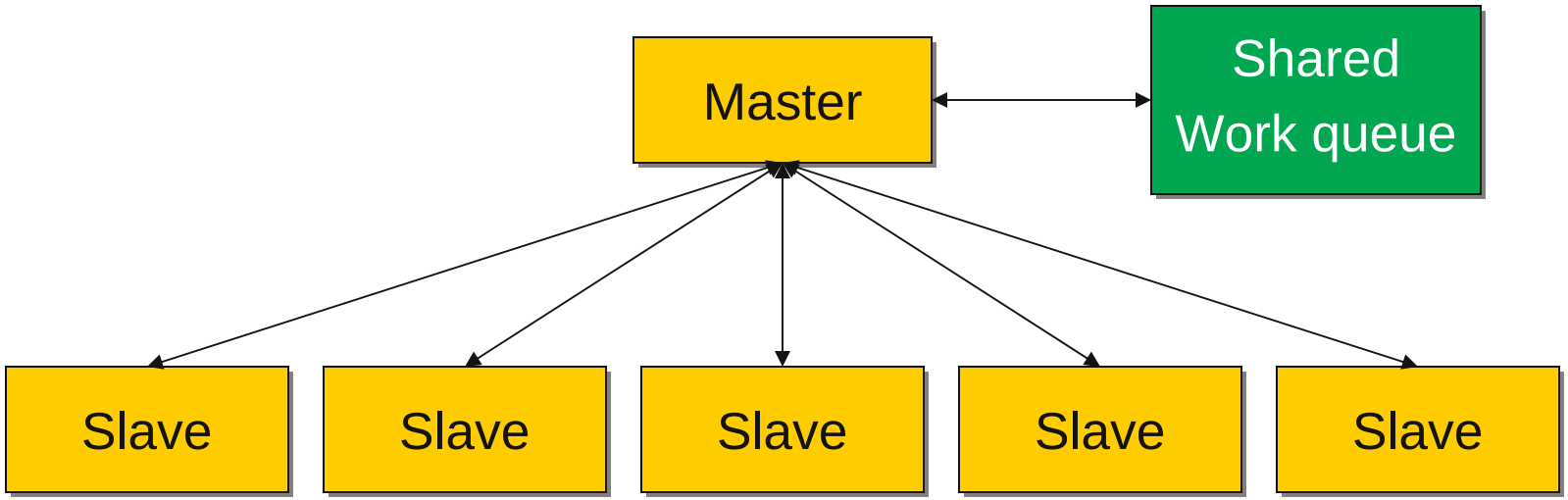
A More Distant Future

- HPCS-type languages have many interesting ideas for exploiting less obvious parallelism
- Need coordination and freedom from vendor ownership
- A convergence plan
  - (DARPA briefly funded a convergence project, which was promising until cancelled)
- A migration plan for current applications
  - Interaction with MPI
  - Use in libraries

Avoiding MPI: The Asynchronous Dynamic Load-Balancing Library

- Overview of ADLB
- The API in a nutshell
- How it works
- Tutorial example

Master/Slave Algorithms and Load Balancing

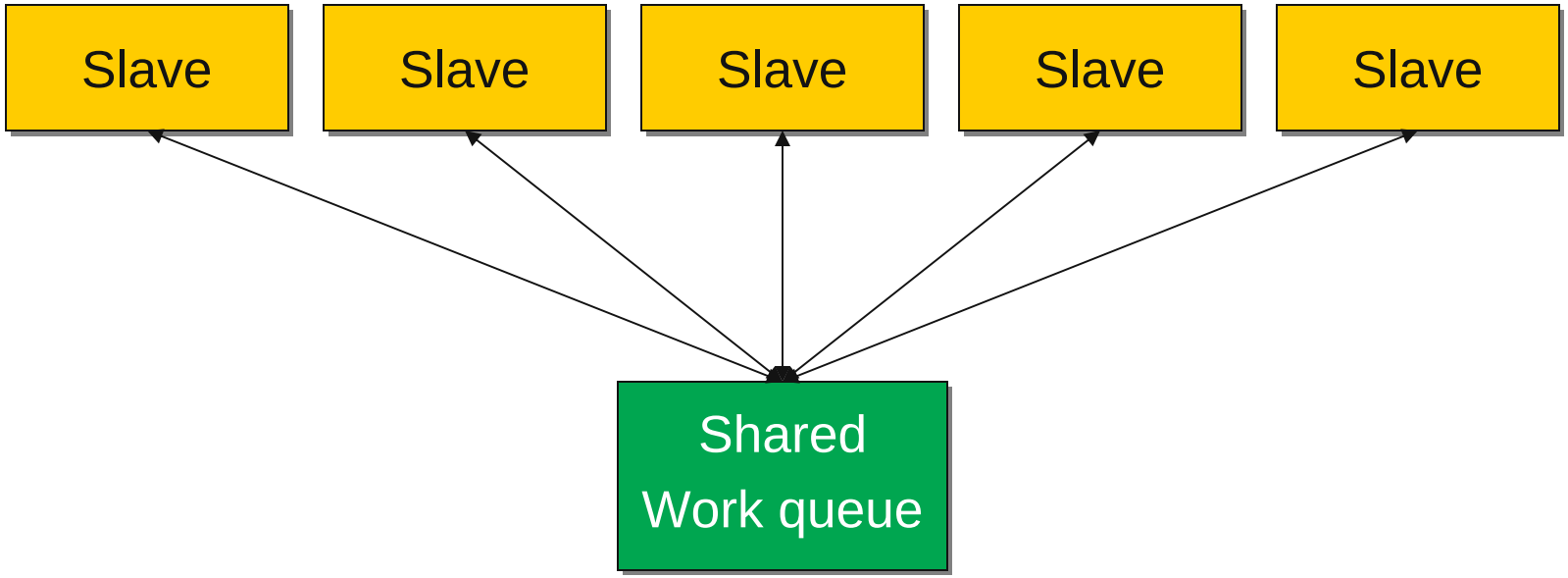


- Advantages
  - Automatic load balancing
- Disadvantages
  - Scalability - master can become bottleneck
- Wrinkles
  - Slaves may create new work
  - Multiple work types and priorities that impose work flow

The ADLB Vision

- No explicit master for load balancing; slaves make calls to ADLB library; those subroutines access local and remote data structures (remote ones via MPI).
- Simple Put/Get interface from application code to distributed work queue hides most MPI calls
  - Advantage: multiple applications may benefit
  - Wrinkle: variable-size work units, in Fortran, introduce some complexity in memory management
- Proactive load balancing in background
  - Advantage: application never delayed by search for work from other slaves
  - Wrinkle: scalable work-stealing algorithms not obvious

The ADLB Model (no master)



- Doesn't really change algorithms in slaves
- Not a new idea (e.g. Linda)
- But need scalable, portable, distributed implementation of shared work queue
  - MPI complexity hidden here.

API for a Simple Programming Model

- Basic calls
  - ADLB_Init(num_servers, am_server, app_comm)
  - ADLB_Server()
  - ADLB_Put(type, priority, len, buf, answer_dest)
  - ADLB_Reserve(req_types, handle, len, type, prio, answer_dest)
  - ADLB_Ireserve(…)
  - ADLB_Get_Reserved(handle, buffer)
  - ADLB_Set_Done()
  - ADLB_Finalize()
- A few others, for tuning and debugging
  - ADLB_{Begin,End}_Batch_Put()
  - Getting performance statistics with ADLB_Get_info(key)

Parallel Sudoku Solver with ADLB

1	2				9			7
		3				6	1	
				7		8		
					5	3		
7		9	1		8	2		6
		5	6					
		1		9				
	6	7				1		
2			5				3	8

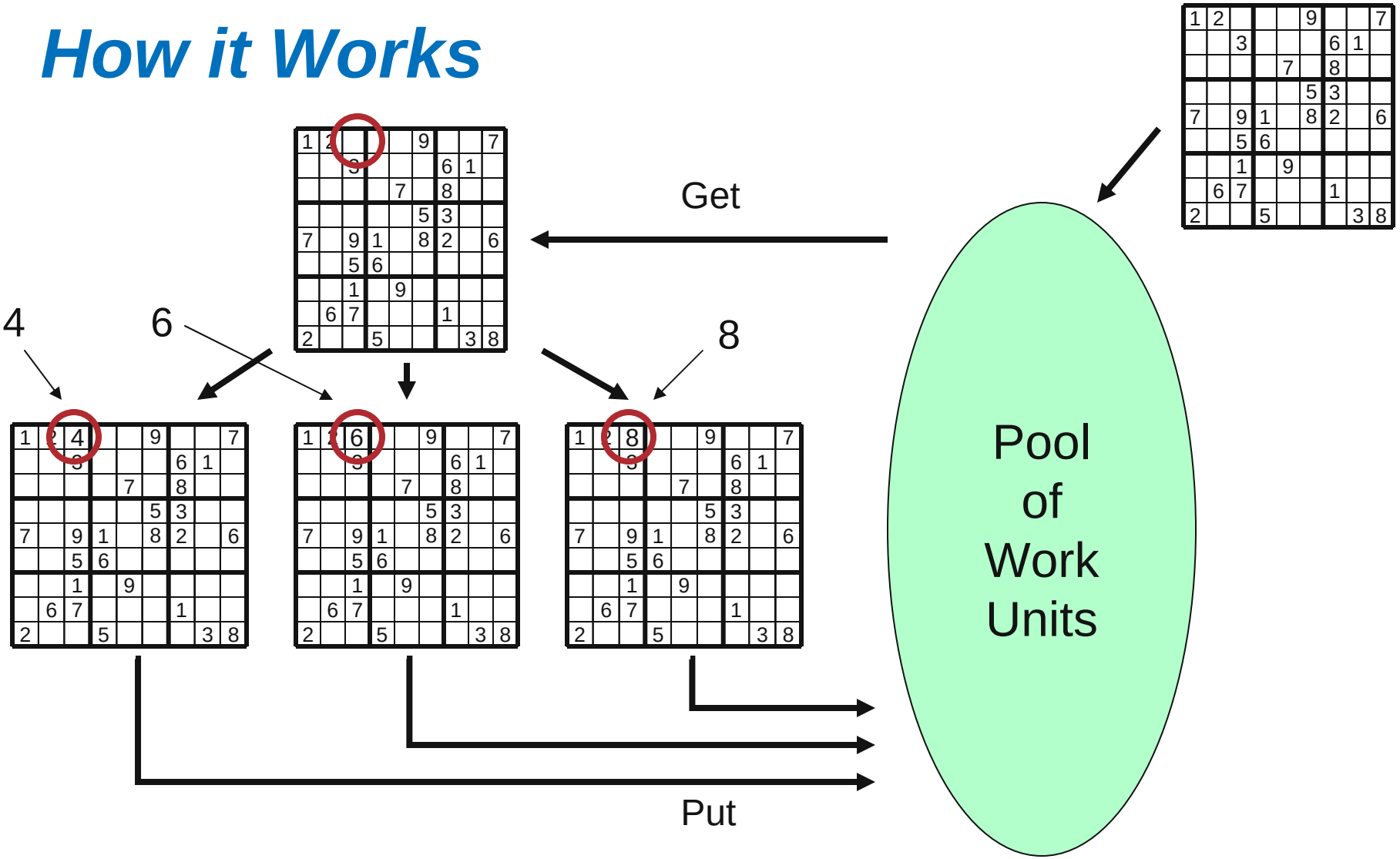
Work unit =

partially completed “board”

Program:

```
if (rank = 0)
    ADLB_Put initial board
ADLB_Get board (Reserve+Get)
while success  (else done)
    ooh
    find first blank square
    if failure  (problem solved!)
        print solution
        ADLB_Set_Done
    else
        for each valid value
            set blank square to value
            ADLB_Put new board
        ADLB_Get board
end while
```

How it Works



- After initial Put, all processes execute same loop (no master)

Optimizing Within the ADLB Framework

- Can embed smarter strategies in this algorithm
  - ooh = “optional optimization here”, to fill in more squares
  - Even so,potentially a *lot* of work units for ADLB to manage
- Can use priorities to address this problem
  - On ADLB_Put, set priority to the number of filled squares
  - This will guide depth-first search while ensuring that there is enough work to go around
    - *How one would do it sequentially*
- Exhaustion automatically detected by ADLB (e.g., proof that there is only one solution, or the case of an invalid input board)

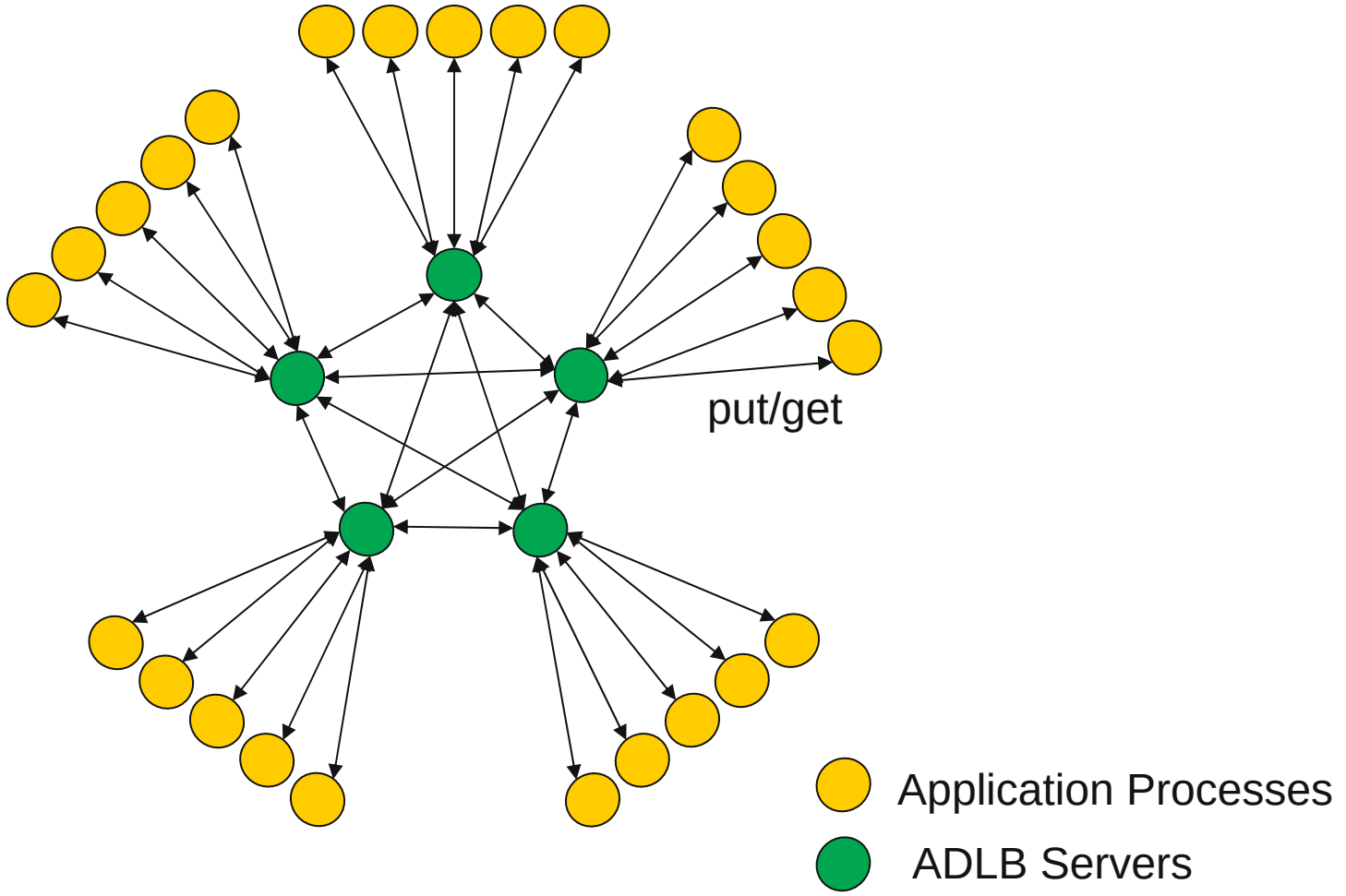
Experiments with GFMC/ADLB on BG/P

- Using GFMC to compute the binding energy of 14 neutrons in an artificial well (“neutron drop” = teeny-weeny neutron star)
- A weak scaling experiment

BG/P cores	ADLB Servers	Configs	Time (min.)	Efficiency (incl. serv.)
4K	130	20	38.1	93.8%
8K	230	40	38.2	93.7%
16K	455	80	39.6	89.8%
32K	905	160	44.2	80.4%

- Recent work: “micro-parallelization” needed for ^{12}C, OpenMP in GFMC.

How It Works



- Real numbers: 1000 servers out of 32,000 processors on BG/P
  - And recently introduced other communication paths

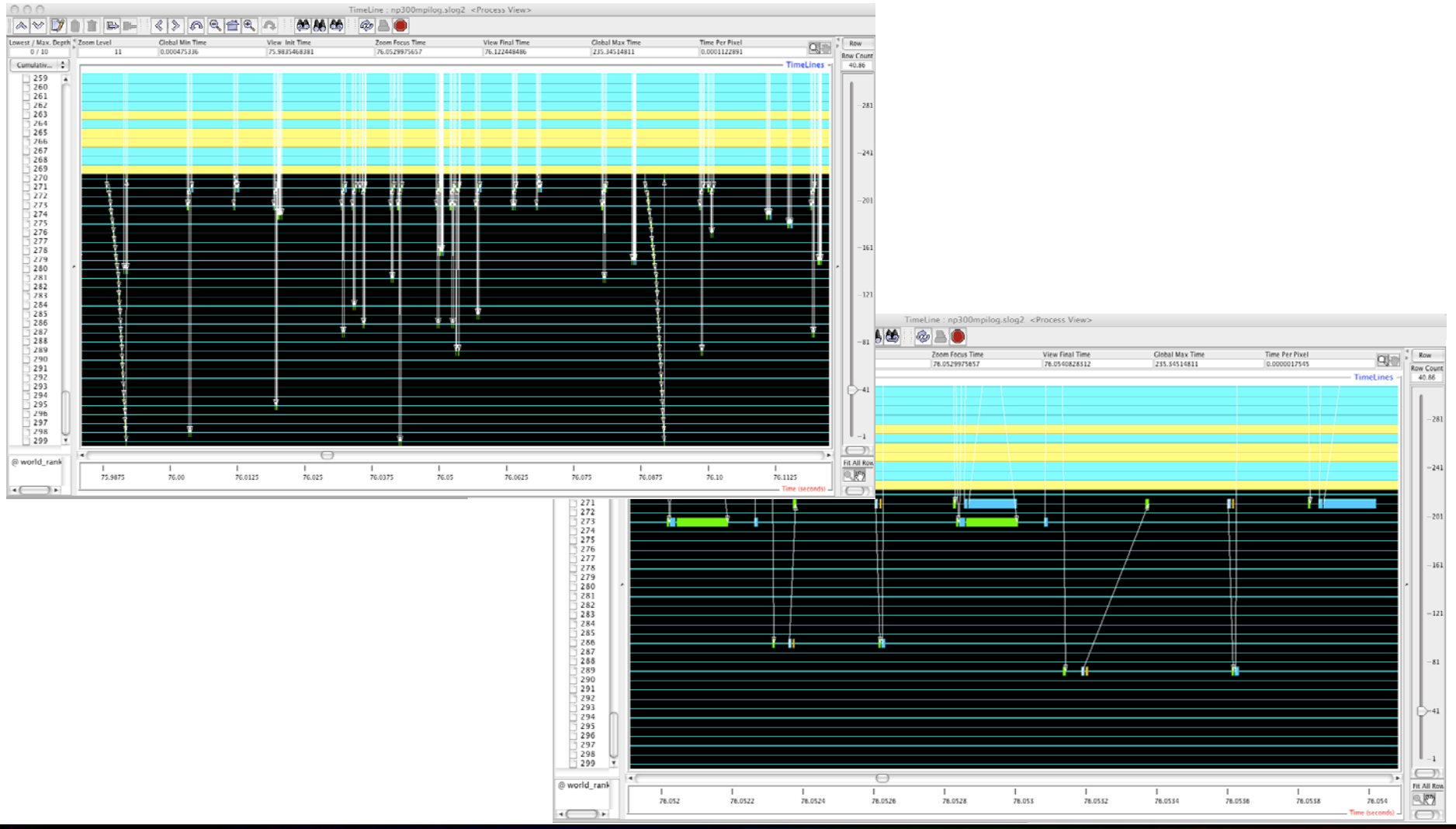
The ADLB Server Logic

- Main loop:
  - MPI_Iprobe for message in busy loop (emit diagnostics)
  - MPI_Recv message
  - Process according to type (20 types)
    - *Update status vector of work stored on remote servers*
    - *Manage work queue and request queue*
    - *(may involve posting MPI_Isends to isend queue)*
  - MPI_Test all requests in isend queue
  - Return to top of loop
- The status vector replaces single master or shared memory
  - Circulates every .1 second at high priority

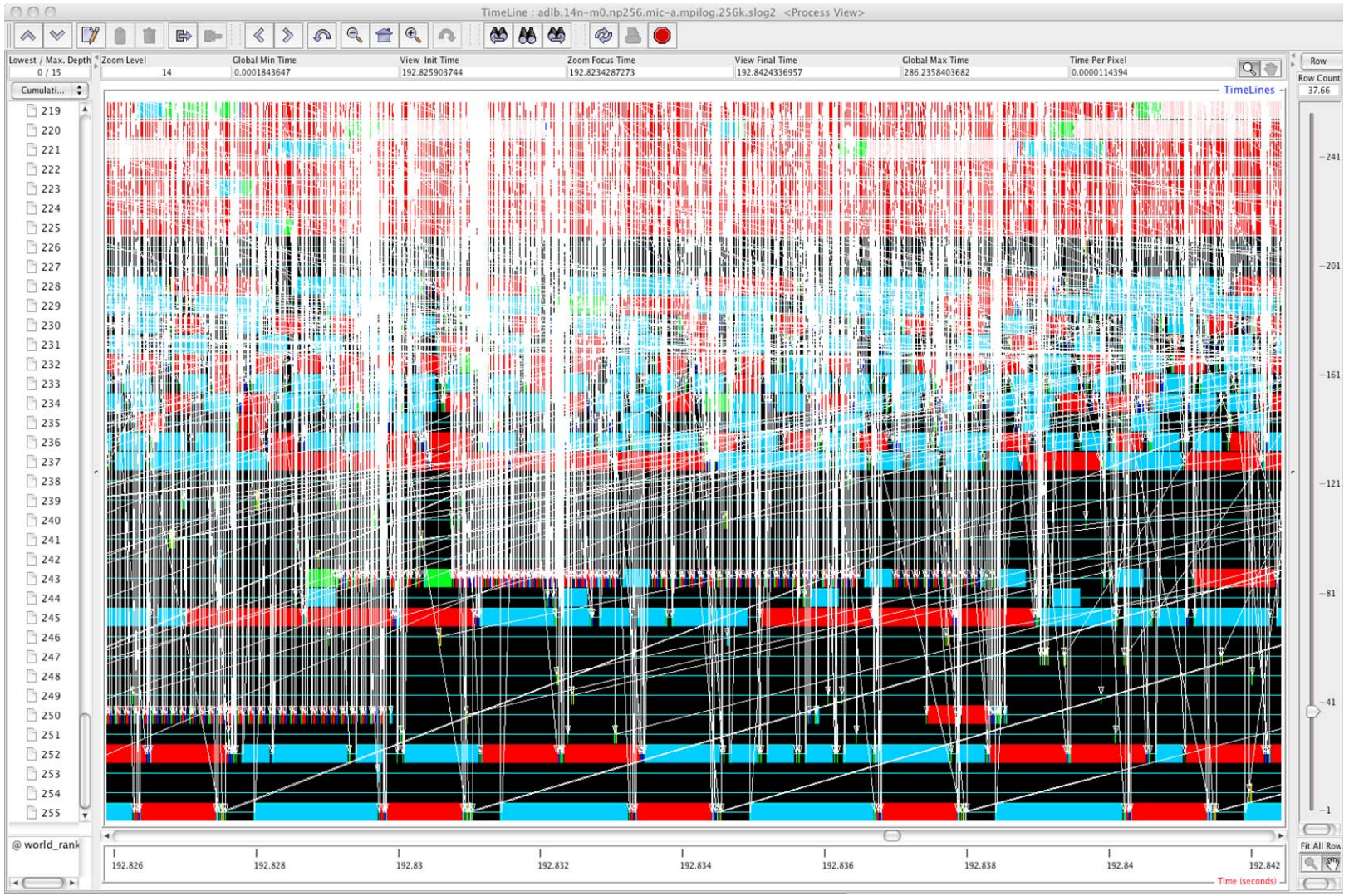
ADLB Uses Multiple MPI Features

- ADLB_Init returns separate application communicator, so application can use MPI for its own purposes if it needs to.
- Servers are in MPI_Iprobe loop for responsiveness.
- MPI_Datatypes for some complex, structured messages (status)
- Servers use nonblocking sends and receives, maintain queue of active MPI_Request objects.
- Queue is traversed and each request kicked with MPI_Test each time through loop; could use MPI_Testany.
- Client side uses MPI_Ssend to implement ADLB_Put in order to conserve memory on servers, MPI_Send for other actions.
- Servers respond to requests with MPI_Rsend since MPI_Irecvs are known to be posted by clients before requests.
- MPI provides portability: laptop, Linux cluster, SiCortex, BG/P
- MPI profiling library is used to understand application/ADLB behavior.

Looking at GFMC/ADLB with Jumpshot (in the good old days)



Things Can Get Worse at Larger Scale



Multiple Load-Balancing Regimes

- The original objective was to do balancing of processing load
- Much of the last year has been spent on balancing of the memory load
  - Work units may to be moved from server to server
  - Even proactively
- We may now be having problems that can only be solved by balancing of the message-passing load.

Summary

- MPI provides effective ways to access communication performance
  - You may need to help the implementation out
  - 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
- Programming models for exascale are still in experimental stages
- Hiding MPI calls in higher-level libraries can be a useful approach to programmer productivity

The End