

Future Developments in MPI

William Gropp

Mathematics and Computer Science

www.mcs.anl.gov/~gropp



Outline

- MPI is a success!
 - ◆ What's next?
- MPI and Programming Models
 - ◆ How can we build a better programming environment?
- MPI Implementations
 - ◆ What's left to do?
- MPI Scalability
 - ◆ Is MPI scalable?

Programming Models

- The MPI programming model
 - ◆ (messaging as powerful model – Fox talk)
 - ◆ MPI as pt2pt/collective(MPI-1) and RMA(MPI-2)
- Support for libraries!
 - ◆ Contexts, communicators, process subsets

Examples: I/O – PnetCDF

- Simple programming language (write/print). Looks simple. But real data files have headers, special data representations, ...
- Solution (real world): I/O libraries – netCDF, HDF4
 - ◆ Abstract away details of file layout
 - ◆ Provide standard, portable file formats
 - ◆ Include metadata describing contents
- Solution (real world parallel): PnetCDF, HDF5
 - ◆ PnetCDF exploits MPI concepts: communicators, collective to get better performance (developed as part of the DOE Scientific Data Management project)

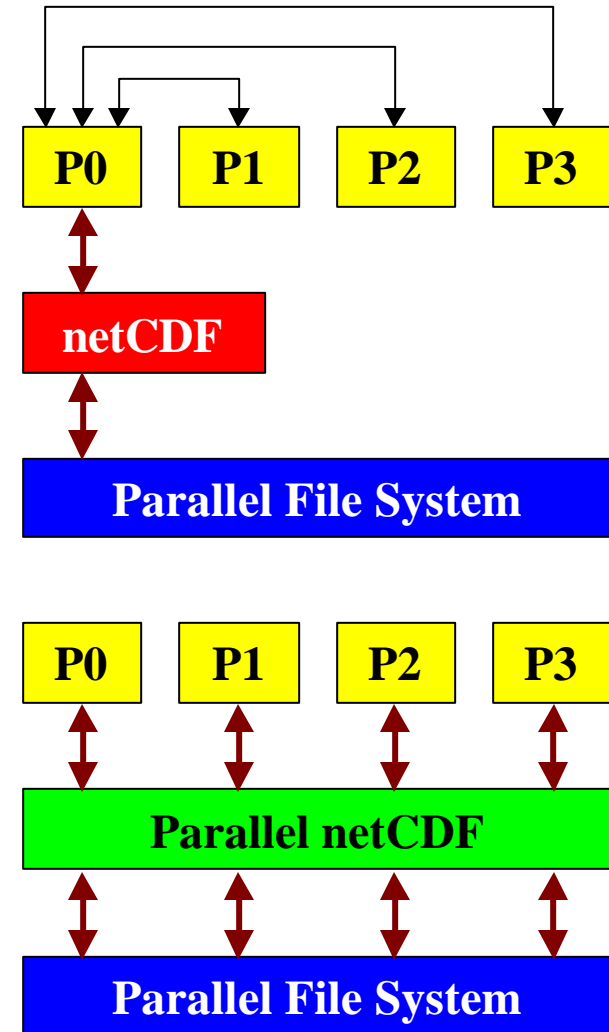
Parallel netCDF (PnetCDF)

- Collaboration between NWU and ANL as part of the Scientific Data Management SciDAC
- netCDF
 - ◆ API for accessing multi-dimensional data sets
 - ◆ Portable file format
- Popular in both fusion and climate communities
- Parallel netCDF is an effort to
 - ◆ Very similar API to netCDF
 - ◆ Tuned for better performance in today's computing environments
 - ◆ Retains the file format so netCDF and PnetCDF applications can share files

I/O in netCDF

- Original netCDF
 - ◆ Parallel read
 - All processes read the file independently
 - No possibility of collective optimizations
 - ◆ Sequential write
 - Parallel writes are carried out by shipping data to a single process

- PnetCDF
 - ◆ Parallel read/write to shared netCDF file
 - ◆ Built on top of MPI-IO which utilizes optimal I/O facilities of the parallel file system and MPI-IO implementation
 - ◆ Allows for MPI-IO hints and datatypes for further optimization
 - (e.g., noncontig IO opt (Worringer et al) and fast MPI datatypes (Ross))

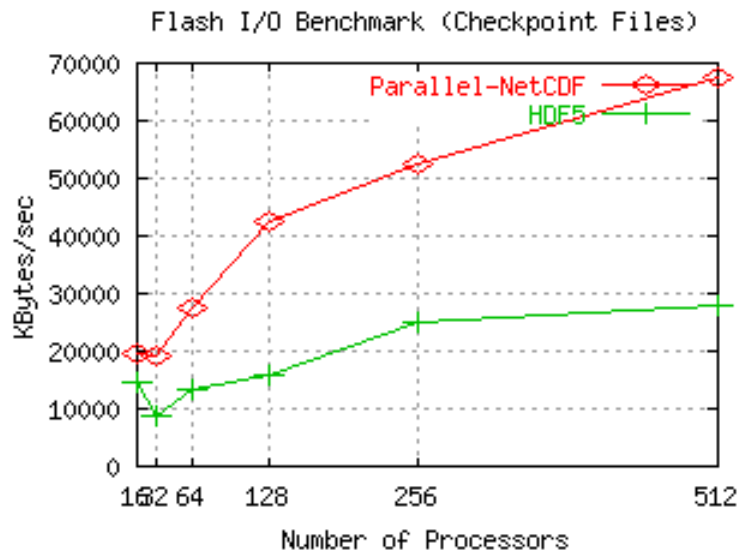


Example: I/O in FLASH

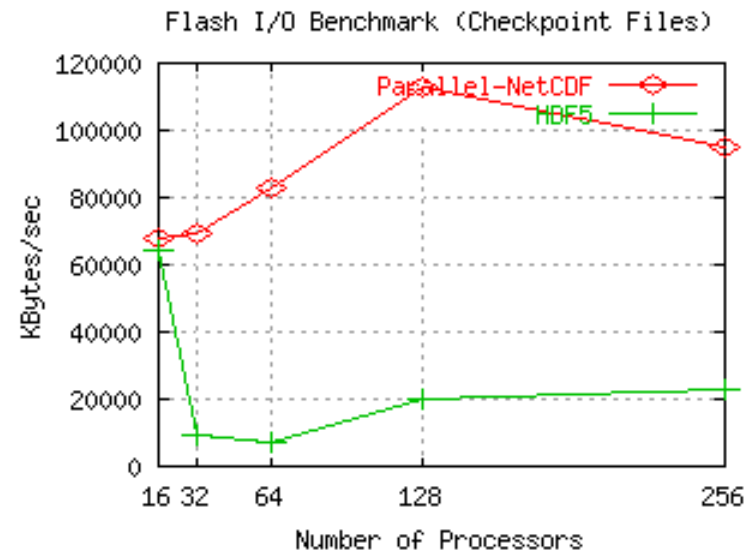
- FLASH is an astrophysics simulation code from the University of Chicago ASCI Center
- Fluid dynamics code using adaptive mesh refinement (AMR)
- Runs on systems with thousands of nodes
- Has been using HDF5 for checkpointing and other output

Performance Comparison vs. HDF5

- Results using FLASH I/O benchmark on ASCI Frost at LLNL
 - ◆ IBM's MPI
 - ◆ GPFS parallel file system



8x8x8 Blocks



16x16x16 Blocks

FLASH and PnetCDF in Practice

- The amount of time FLASH spends in I/O varies widely between platforms
- For some platforms, the change in I/O performance has resulted in 2-3 times speedup for the application as a whole!
- No general “language level” I/O will provide this level of application support

Examples: Distributed data structures

- Simple programming language (HPF, CAF, UPC) supports “natural” objects (e.g., 2-d array)
- Real world: sparse matrices, periodic grids, “C” grids, unstructured grids, staggered grids, All must be built from other structures.
- PETSc, HYPRE provide more general distributed data structures
 - ◆ (For more info, see, e.g., DOE TOPS SciDAC)

Example: Earthquake ground motion simulation

- Thanks to Omar Ghattas CMU; see his paper at SC03 for more information
- PETSc provides framework for all simulations; the “programming model” is PETSc, enabled by MPI.
- Typical run time ~4 hrs (exclusive of input, setup, and output) on 3000 PEs to simulate 60s of ground motion for LA Basin 1 Hz model
- Actual run time depends on attenuation model particulars
- Simulation runs at ~80% parallel efficiency on 3K PEs



PEs	model	grids pts	pts/PE	Gfbps	Mfbps/PE	efficiency
1	LA10S	134,500	134,500	0.505	505	1.00
16	LA5S	618,672	38,667	7.85	491	0.972
128	LA2S	14,792,064	115,563	60.0	469	0.929
512	LA1HA	47,556,096	92,883	231	451	0.893
1024	LA1HB	101,940,152	99,551	460	450	0.891
2048	LA1HB	101,940,152	49,775	907	443	0.874
3000	LA1HB	101,940,152	33,980	1,210	403	0.800

Lessons from MPI

- Make hard things possible; making easy things easy isn't relevant
- Implication for future programming models: Rather than new languages,
 - ◆ Consider better support for current languages (telescoping languages; source-to-source transformation including a clean interface to correctness and performance debugging, etc.)
 - ◆ Consider better application environments. E.g., "domain-specific" languages. Matlab, R, Mathematica, Nastran, ...
 - ◆ Consider new languages, but ones that better support building the necessary tools

MPI Implementations

- Performance of MPI
 - ◆ Already very good
 - Few microsecond latency close to memory reference times (read/write to main memory)
 - Bandwidths asymptotic to underlying communication layer
 - ◆ Great progress already described at this meeting
 - Non-contiguous data and I/O (Worringer et al)
 - Collective (Thakur et al)
 - Fault tolerant (Graham et al)
 - Datatype (Ross et al)
 - Many others today and tomorrow
 - ◆ Opportunities Include (where things could be better)
 - Mixed transports (e.g., even better performance when using shared memory on SMP nodes, IB between nodes)
 - RMA, IO
 - Key idea “relaxed memory consistency”; separate the initiation of a operation from the guarantee of completion
 - ◆ Example: MPICH2

MPICH2

- Features preserved from MPICH1
 - ◆ Portable to everything
 - ◆ High performance (MPICH1 had high performance except for the old ch_p4 device; MPICH2 has high performance for sockets, etc.)
 - ◆ Multiple levels of modules
 - Easy, for example, to replace a single collective algorithm, on particular choices of communicator (e.g., only on MPI_COMM_WORLD)
 - Easy to introduce new device implementations on multiple levels
 - Simple “channel” interface for quick ports with good performance
 - Full-featured ADI3 for full performance; still leverages common modules such as datatype handling

New Features in MPICH2

- Scalable process management interface
 - ◆ MPICH1 typically used (but did not depend on; e.g., bproc (Scyld) or secure server) a remote shell program
 - ◆ MPICH2 simply asks for the scalable startup of the application processes
 - ◆ Multiple startup mechanisms can be used with the same MPI executable. MPICH2 includes a demon-based system (see Rusty's talk tomorrow), a single-node (based on fork), and (by SC03) a "classic" rsh/ssh interface
 - ◆ Example: BG/L implementation (Gupta's talk yesterday)

New Features in MPICH2 II

- Separate modules for datatype, topologies, attributes, timing, collectives, groups, RMA, name publishing, ...
 - ◆ Lazy loading of modules reduces footprint, speeds startup, improves scalability
 - ◆ Improved general interface to collectives (also BG/L)
 - ◆ Run and compile time control over high-quality, instance specific error handling
 - ◆ Designed to efficiently support multiple threading levels (MPI_THREAD_MULTIPLE to MPI_THREAD_FUNNELLED), exploiting processor atomic instructions (not locks)

New Features in MPICH2 III

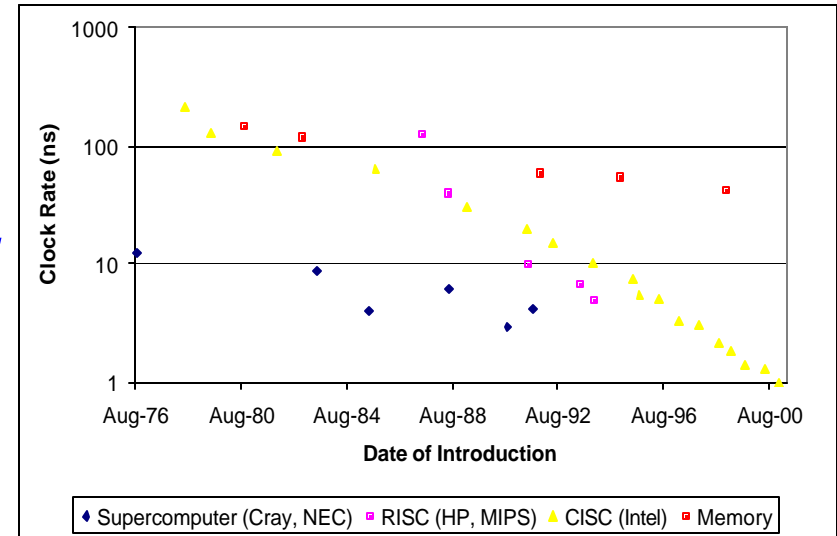
- Extensive tests
 - ◆ MPICH1, MPICH2, Intel, C++ version of IBM, John May's IO tests (about 2000 programs)
 - ◆ Coverage analysis ensures that tests (nearly) complete
- Goal is to support all of MPI1 and MPI2
 - ◆ By SC03!
- Scalable to 100K+ processes
 - ◆ E.g., no explicit description of MPI groups required
- Exploits opportunities for RMA optimizations...
(but first some discussion of *achievable* performance)

What is “Relaxed Memory Consistency”

- Sequential consistency (Lamport):
 - ◆ A parallel program executes as if the statements are executed sequentially, in an arbitrary interleaving, but retaining the partial order from each process
- Relaxed:
 - ◆ Do not require sequential consistency
 - ◆ Do have some mechanism for enforcing a consistent view of memory at specific points (times and places) in the computation
 - ◆ A generic term for memory consistency models that are weaker than sequential consistency

Why Relaxed Memory Consistency?

- Engineering
 - ◆ DRAM memory much slower than processors, and the gap is growing
- Physics
 - ◆ Signals travel no faster than the speed of light (about 3×10^8 m/sec)
 - ◆ Clock speeds are now very high (2.4×10^9 cycles/sec)
 - ◆ Signals travel no more than 12.5 cm per cycle!



The Dimensions of a Typical Cluster

- 6.1 m x 2.1 m x 1m
- 1-norm size (airline baggage norm) = 9.2m
- At 2.4Ghz, =
74 cycles
 (49 x 17 x 8)
- Real distance is greater
 - ◆ Routes longer
 - ◆ Signals slower than light in a vacuum



Computers and Special Relativity

- Simultaneity is already broken for parallel computers (we can only talk about “at the same time” in the rest frame of the computer)
- Consequences
 - ◆ A “load” operation from anywhere in memory may require 148 cycles, just to move the data at the speed of light
 - Does not include the time to access the data
 - ◆ Must separate out initiation from completion
 - ◆ Must be careful about requiring every processor’s view of memory to be the same
 - As in, this does not have any meaning any more
 - ◆ Many HPC programming models are moving in this direction
 - A natural fit for MPI (nonblocking operations in MPI-1 + MPI-2)

RMA implementation

- Gabriel et al talk on performance of RMA operations
 - ◆ Many implementations suboptimal (slower than send-receive for large messages)
 - ◆ Latency of ping-pong high
 - Note mpptest contains an RMA “halo exchange” test (and many other features) that may be more relevant to RMA (and application) performance
 - ◆ Note asymptotic performance must mirror underlying hardware – anything lower indicates a flaw in the MPI implementation (not in the MPI specification)
- A key to low latency and high bandwidth in RMA is proper handling of
 - ◆ Deferred synchronization
 - ◆ Lazy messaging and message aggregation
 - ◆ Attention to details and limitations of RDMA hardware and lower-level software
 - (e.g., D.K. Panda’s IB talk for collective RDMA)
- Let’s look at RMA and point-to-point messaging (joint work with Rajeev Thakur)...

Remote Memory Access Compared with Message-Passing

- **Separates** data transfer from synchronization
- In message-passing, they are combined

Proc 0	Proc 1	Proc 0	Proc 1
store		fence	fence
send	receive	put	
	load	fence	fence
			load
			<i>or</i>
		store	
		fence	fence
			get
			fence

MPI RMA Synchronization

- No assumption of cache/memory coherent shared memory
 - ◆ A feature of the fastest machines (almost by definition)
- Three models
 - ◆ All processes take part
 - ◆ “Neighboring” processes (both origin and target) take part
 - ◆ Only the origin process takes part
- First two are called “active target”, last is called “passive target”

Fence Implementations

- Obvious implementation
 - ◆ First MPI_Win_fence becomes MPI_Barrier
 - ◆ Operations (i.e., put, get, accumulate)
 - ◆ Last MPI_Win_fence becomes Barrier
- Problem: Barriers can be expensive
- Alternative
 - ◆ Aggregate
 - Collect all operations between two MPI_Win_fence calls and keep track of their targets
 - ◆ Allreduce with MPI_SUM of target processes as array of ints, indexed by rank of the *target* process
 - Some hardware can perform integer allreduce
 - ◆ Perform RMA operations; let the last operation flag that it is the last operation (op + memory fence)
 - This is similar to a memory fence in uniprocessors
 - ◆ No second barrier required!
 - ◆ Also suggests a good primitive for implementing MPI_Put

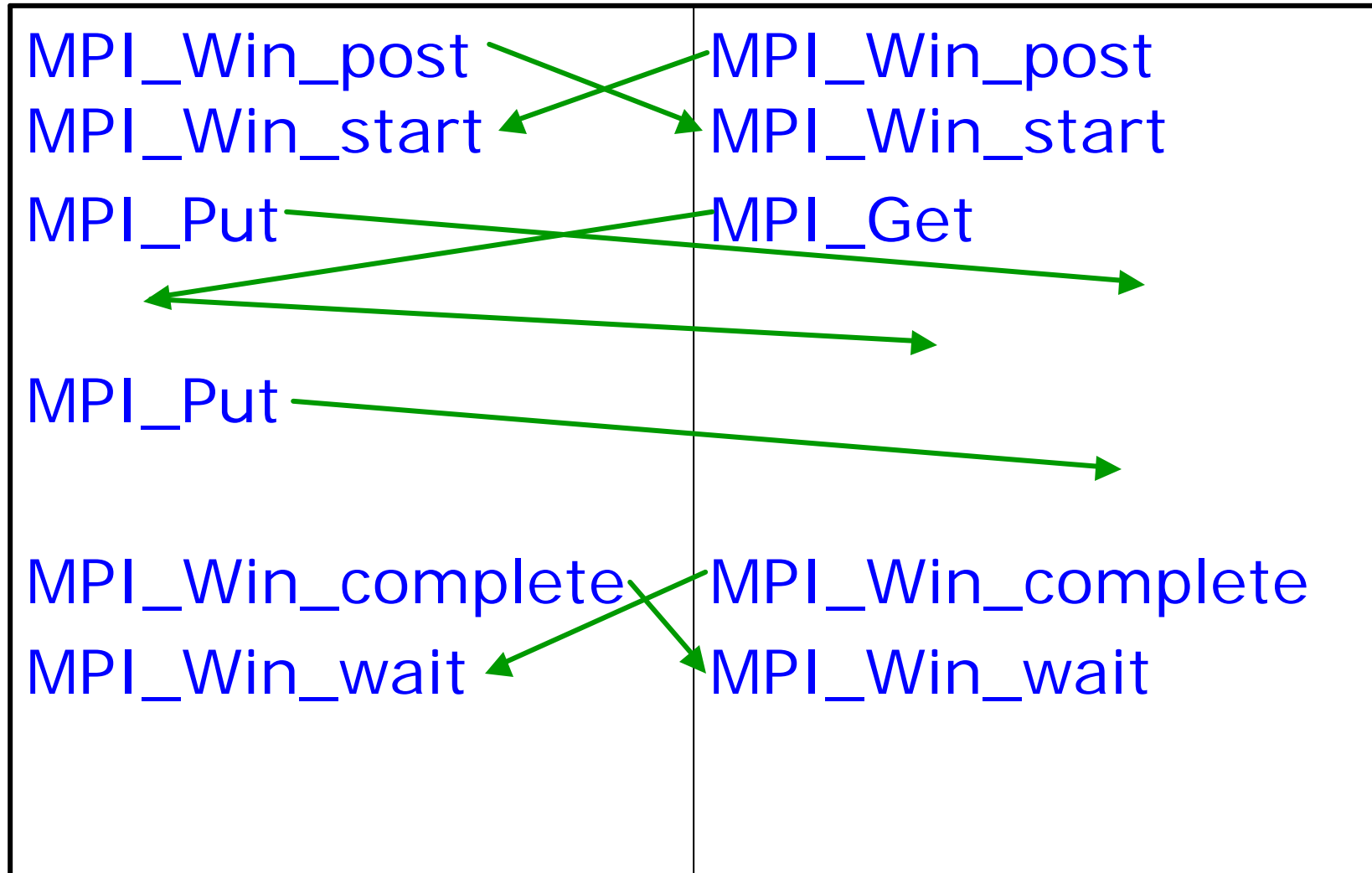
Scalable Synchronization

- This second model defines two subsets:
 - ◆ Exposure of the process's memory window (called an *exposure epoch*)
 - Defined in terms of the group (an MPI_Group) of processes that may access the window
 - MPI_Win_post ... MPI_Win_wait
 - ◆ Intent to access another process's memory window (called an *access epoch*)
 - Also defined in terms of a group of processes that will be accessed
 - MPI_Win_start ... MPI_Win_complete
- This model is scalable; there is no barrier (in some sense, the information that the barrier provides is included in the MPI_Groups passed to the routines)

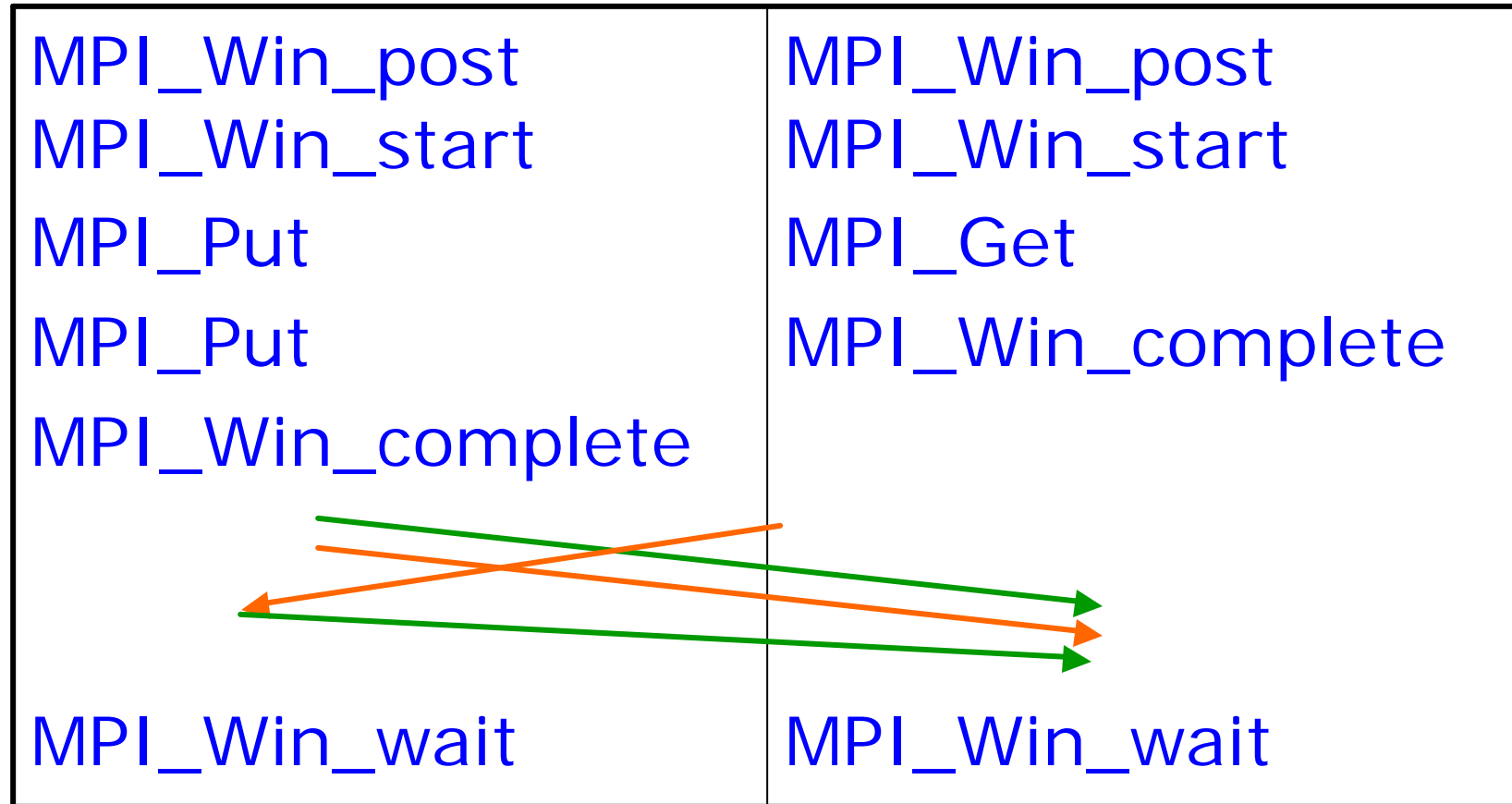
Implementing Scalable Synchronization

- Obvious implementation
 - ◆ Send messages to partners (setup)
 - E.g., MPI_Win_post sends to all in group, as does MPI_Win_start
 - ◆ Perform RMA operations
 - ◆ Send done messages to partners
 - Based on group arguments
- Alternative
 - ◆ Aggregate and use information on partners provided by the group arguments
 - (debug mode uses collective alltoall of a bit vector to check correct definition of groups)
 - ◆ First RMA operation held at target until Win_post
 - ◆ Last RMA operation includes a marker
 - ◆ No other communication required!

Scalable Synchronization in Pictures



Scalable Synchronization in Pictures



→ Combined RMA op and last op flag

Independent RMA Operations

- Some applications need to update a remote memory location without the explicit participation of the remote process:

```

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

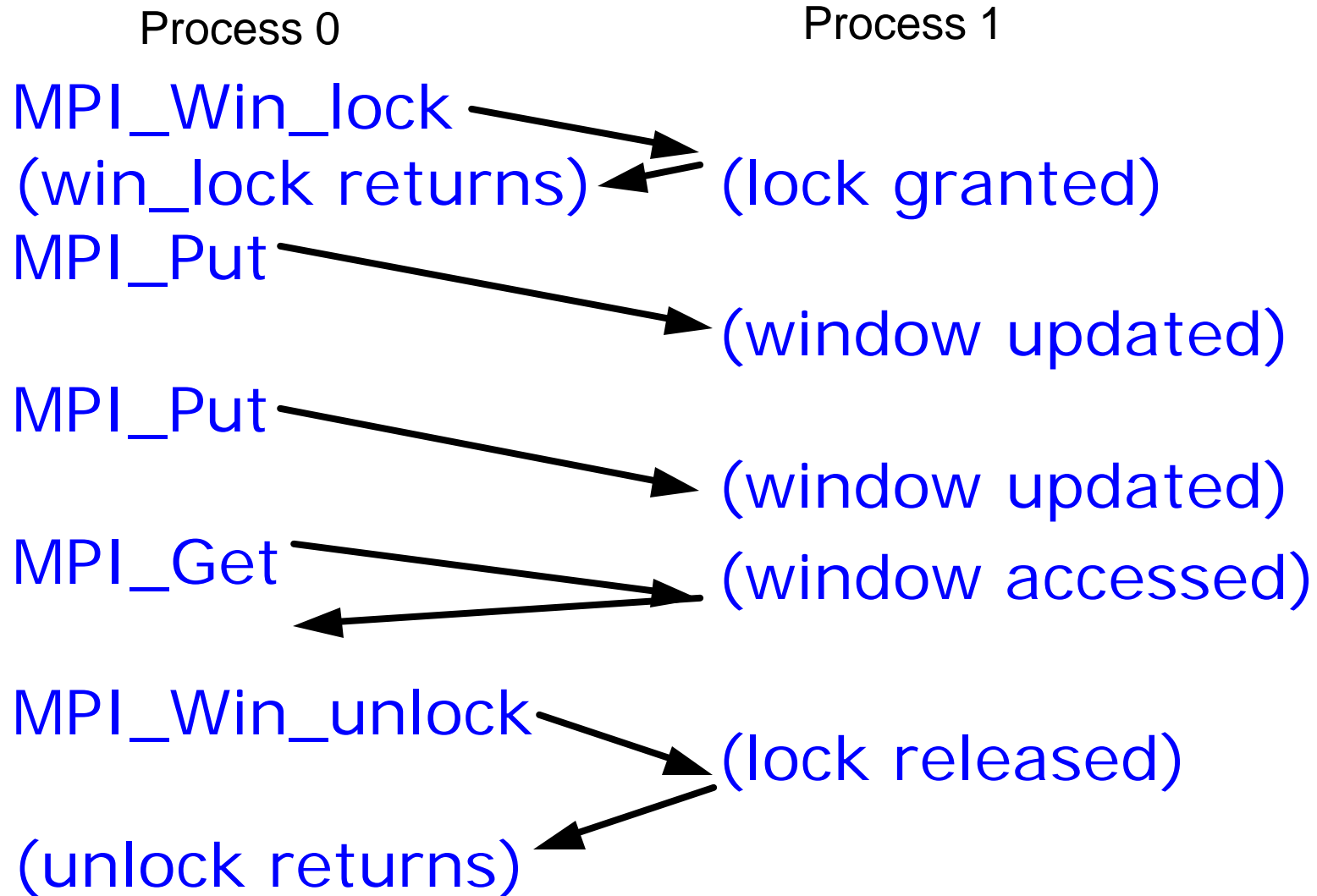
```

- Only process performing MPI_Put makes MPI RMA calls
 - ◆ Process with memory need not make any MPI calls; it is "passive"

Implementing Passive Target RMA

- Lock is not lock (except at self)
- Instead, marks begin and end of sequence of RMA operations
- Obvious Implementation
 - ◆ Lock: send message to target; wait for lock grant
 - ◆ Ops as issued
 - ◆ Unlock: send message to target releasing lock

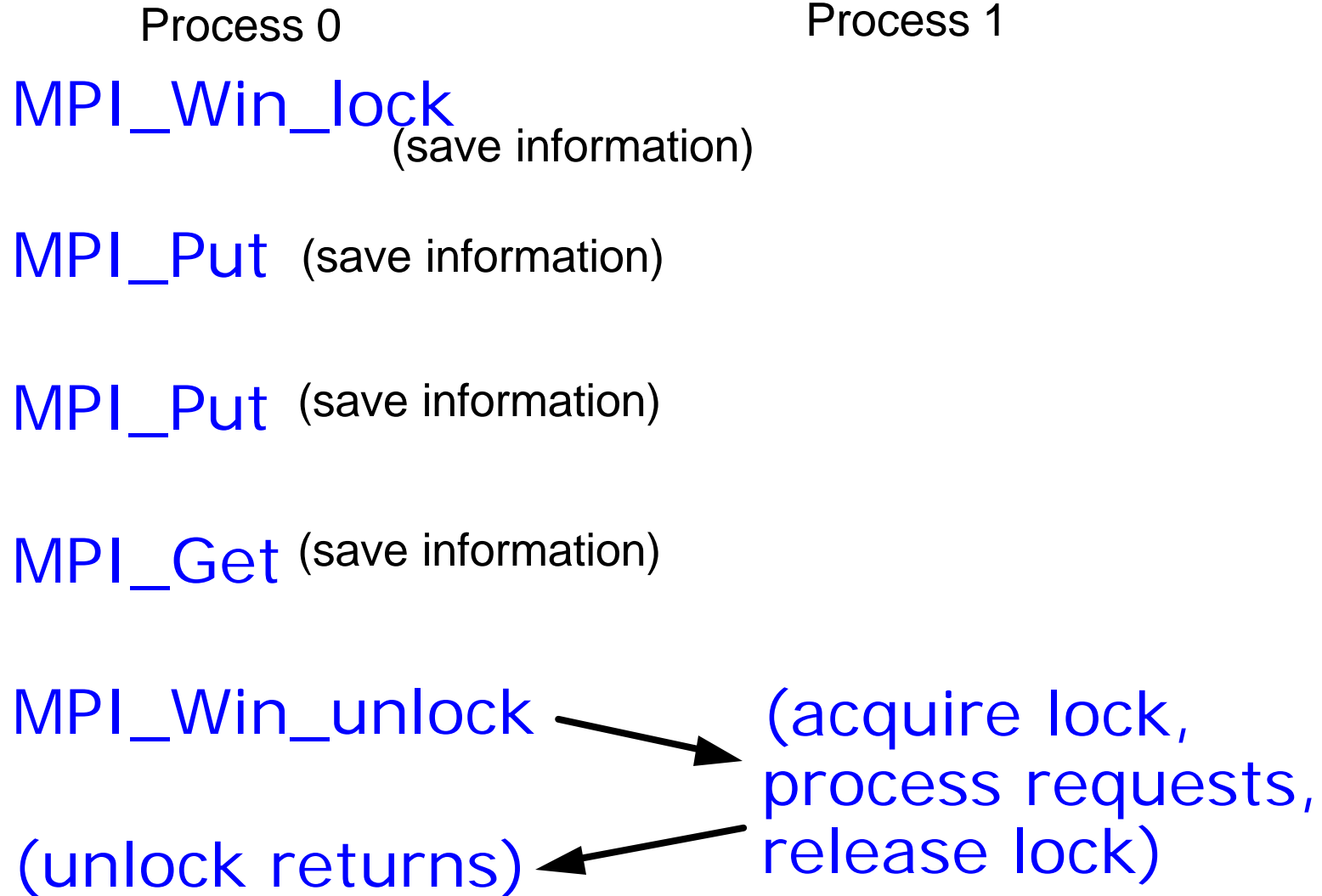
Early Handling of Win_lock



Implementing Passive Target Revisited

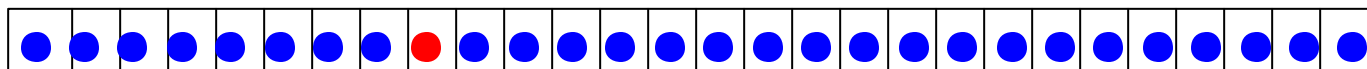
- Alternative
 - ◆ Aggregate.
 - If lock/rma op/unlock, send a single “perform this op atomically”
 - If lock/rma op/rma op ..., send an “acquire lock and perform op, followed by “continue op”, until the last rma op/unlock, which is “perform and release lock”
 - Uses operations similar to those for the active target synchronization (op and set flag)
- Further enhancements
 - ◆ Concurrent updates are allowed if “locks” have byte ranges (common optimization in I/O operations)
 - ◆ An alternative is to use MPI_REPLACE with MPI_Accumulate instead of MPI_Put.
 - Careful choice of consistency semantics

Late Handling of Win_lock



Example of the Strength and Weakness of MPI RMA

- Atomic fetch and increment on a remote node
 - ◆ Recall, MPI has no RMW
- Idea:
 - ◆ non-overlapping get/accumulates
 - ◆ We can always remember our contribution
- Simple implementation
 - ◆ Array of contributions from all processes
 - ◆ Get all but my contribution
 - ◆ Accumulate to my location (for everyone else)
 - ◆ Add up the values from the other processes after the get returns

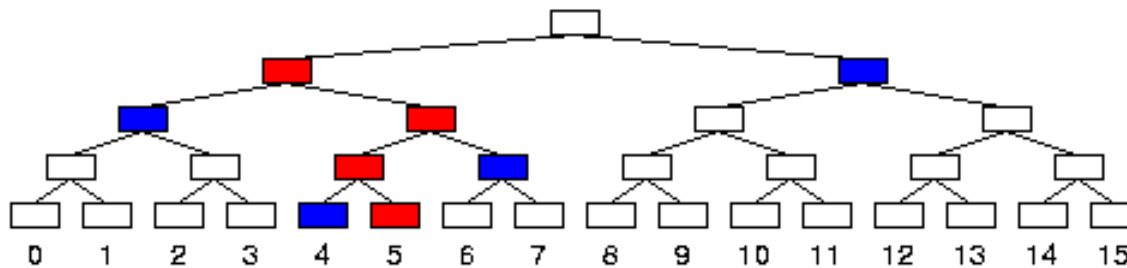
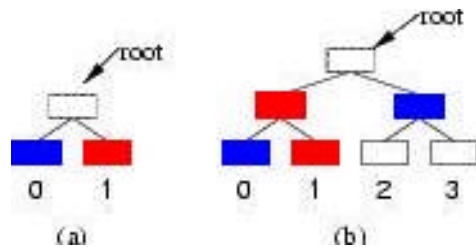


Scalable Remote Fetch and Increment

- Problem with simple array
 - ◆ Requires $O(p)$ time and space
- Idea
 - ◆ Use a tree of contributions
 - ◆ Update the subtree that of the calling process
 - ◆ Get the subtree that the calling process is not in, and the sub-subtrees of the subtrees that the calling process is in
- $\log(p)$ accumulates and $\log(p)$ gets.
- Still requires $O(p)$ space

RMA Accesses for Scalable Fetch and Increment

- ◆ Red — Nodes for accumulate
- ◆ Blue — Nodes for get



Evaluation Of Fetch and Increment in MPI RMA

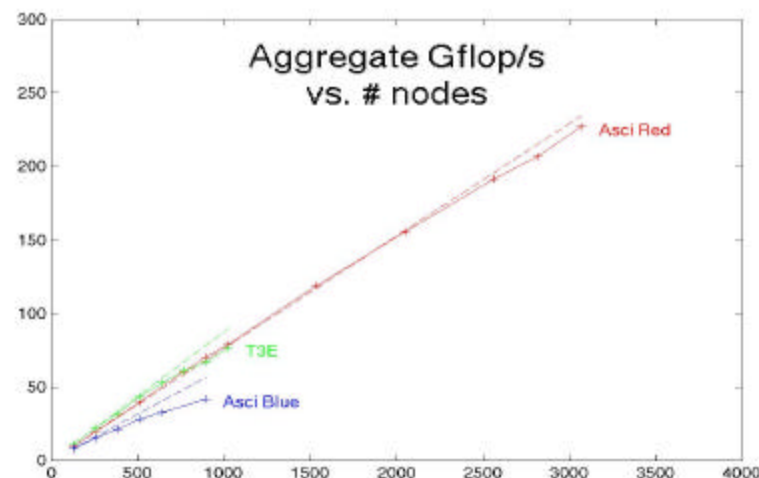
- General datatype operations allow complex remote atomic operations
- More general than a list of available Read-Modify-Write routines
- But
 - ◆ Complex (initial version had several bugs)
 - ◆ Other remote atomic operations, such as double word compare and swap, are much harder.
- Further experience will guide any additions to MPI

I/O

- I/O is all of the previous, with far larger latencies. Even more opportunities for deferred synchronization
 - ◆ E.g., the split collective I/O routines
 - ◆ I/O transfer overlap with communication, computation, particularly on system with multiple networks
- From the applications perspective, PnetCDF and HDF5 are good starts, but more domain-specific libraries are needed
 - ◆ But must follow the approach of PnetCDF – define the semantics so that high-performance is possible
 - ◆ E.g., high performance and POSIX are mutually exclusive*

Scalability

- How Scalable is MPI (the specification)?
 - ◆ We know that it is scalable to thousands of nodes on real applications:



- How scalable do we need it to be?
 - ◆ 64K at least (BG/L), 10+K (Cray RedStorm)

Easy part: Scalability of MPI routines

- Point to point: all are scalable; rank as destination/source allows implicit specification of target/source, allows sparse data structures (do not need a data structure of size $O(p)$ at each process
 - ◆ Though may not be so bad – BG/L accepts this space overhead in exchange for time
- Buffering: dynamic assignment of buffer spaces means that scalable algorithms* use scalable space
- Collective: designed (successfully) to be scalable.
- Communicator management (dup/split)
 - ◆ MPI-2 dynamic process (spawn/spawn_multiple) maintains scalability in the creation and connection to new processes (spawn single process is intrinsically not scalable)
- RMA without fence (post/complete, start/wait)

Scalability (the harder part)

- Less (or not) Scalable
 - ◆ Group routines (enumerate members)
 - ◆ Topology routines (describe complete graph)
 - ◆ But not really important
 - Group routines only real use is for RMA scalable sync, and is used only for specifying a handful of processes
 - Graph topology routine unused
- Don't forget startup/rundown
 - ◆ Scalable startup
 - Do not need separate OS processes (an MPI process need not be an OS process)
 - Do need to provide a way for processes to find each other
 - ▶ Implicit information
 - ▶ Shared and/or distributed data
- Scalability and Faults
 - ◆ Large scale -> faults more likely

Final Comments

- The Future is Bright!
 - ◆ Just look at the many excellent papers at this conference
 - More improvements in implementation
 - Implementations work at greater and greater scale
 - ◆ Increasing numbers of libraries and applications (see the many papers at www.mcs.anl.gov/mpi/papers) use MPI to provide a rich parallel computing environment

Will there be an MPI-3?

- Yes, but not yet; we don't know (quite) enough
 - ◆ Implementations of MPI-2 are available but are not yet performance mature
 - ◆ Applications are not yet stressing the MPI-2 functionality
- But we are very suspicious that
 - ◆ RMA needs some RMW operation (but what?)
 - ◆ Fault tolerance (in an environment where faults are likely) may require a container other than an MPI communicator.

Conclusion

- MPI is successful
 - ◆ Matches the (sometimes painful) realities about computing
 - ◆ Provides good support for “programming in the large”
- Much remains to be done
 - ◆ Implementations can still improve significantly
 - Further enhancements to collectives (e.g., machine topology), datatype (e.g., cache-sensitive), fault-tolerance, RMA, ...
 - ◆ More libraries can exploit MPI
 - Many good libraries for PDEs, linear and nonlinear algebra
 - Need more software components (e.g., CCA), simulation environments, domain-specific libraries