



*... for a brighter future*

# *Programming in MPI for Performance*

*William Gropp*

[www.mcs.anl.gov/~gropp](http://www.mcs.anl.gov/~gropp)



U.S. Department  
of Energy

UChicago ►  
Argonne<sub>LLC</sub>



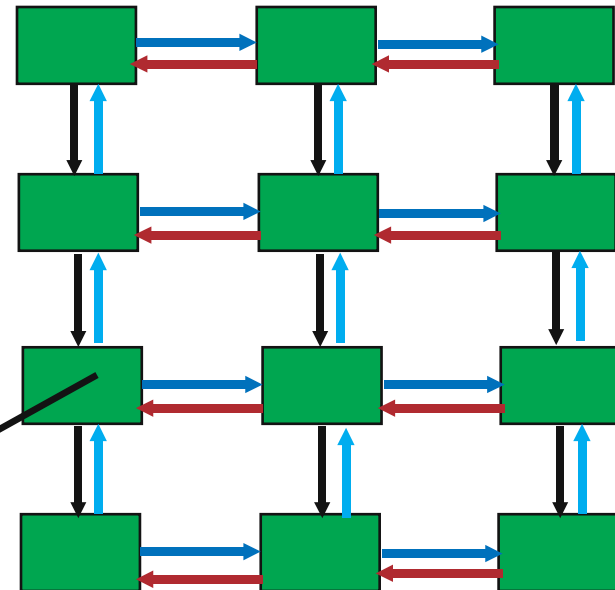
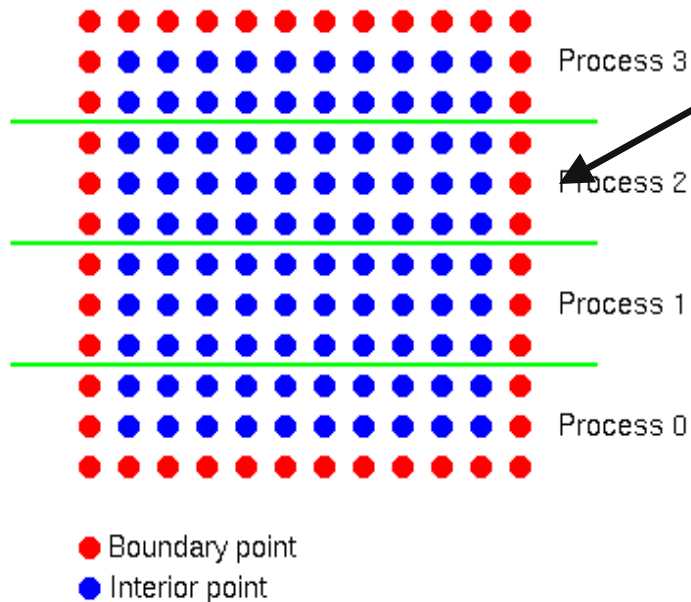
A U.S. Department of Energy laboratory  
managed by UChicago Argonne, LLC

# *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  $a(i,j) = f(a(i-1,j), a(i+1,j), a(i,j+1), a(i,j-1), \dots)$
  - Requires access to “neighbors” in adjacent patches

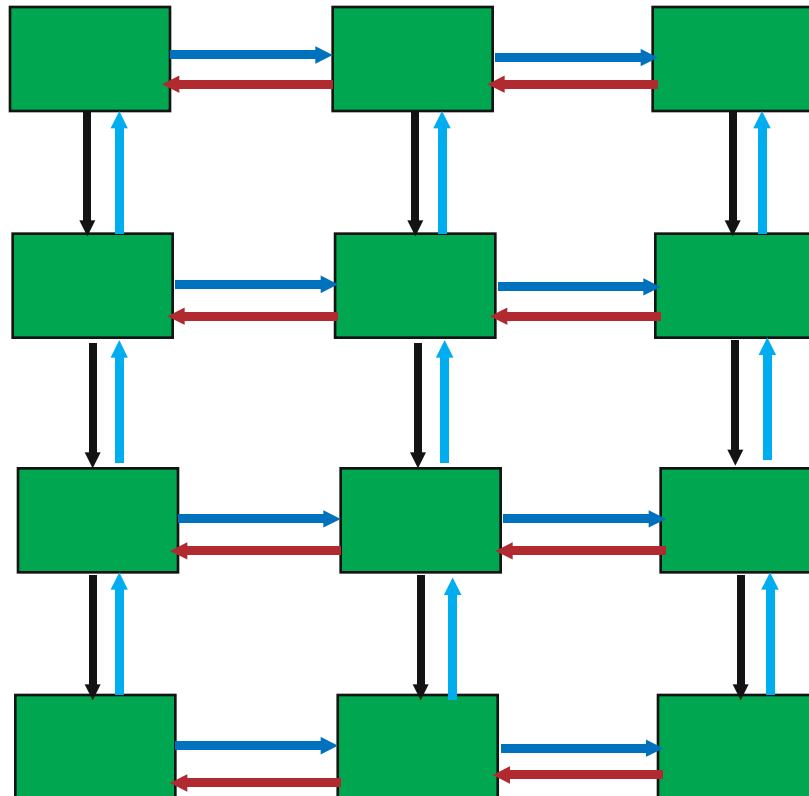


# Scalability of Mesh Exchange

- 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):
    - $1-d = 2 ( s + r n^2 )$
    - $2-d = 4 ( s + r n^2 / \sqrt{p} )$
    - $3-d = 6 ( s + r n / p^{1/3} )$
  - Best approach is to make these relative to floating-point work (this is the dimensionless quantity):
    - $1-d = 2(s + r n^2) / n^3 f$
    - *(Note that the important values are the ratios  $s/f$  and  $r/f$ )*
- 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 Send	Start Send	Start Send	Start Send	Start Send	Start Send Send	Send Recv	Recv
				Send	Recv		
		Send	Recv				
Send	Send Recv	Recv					

# Fix 1: Use Irecv

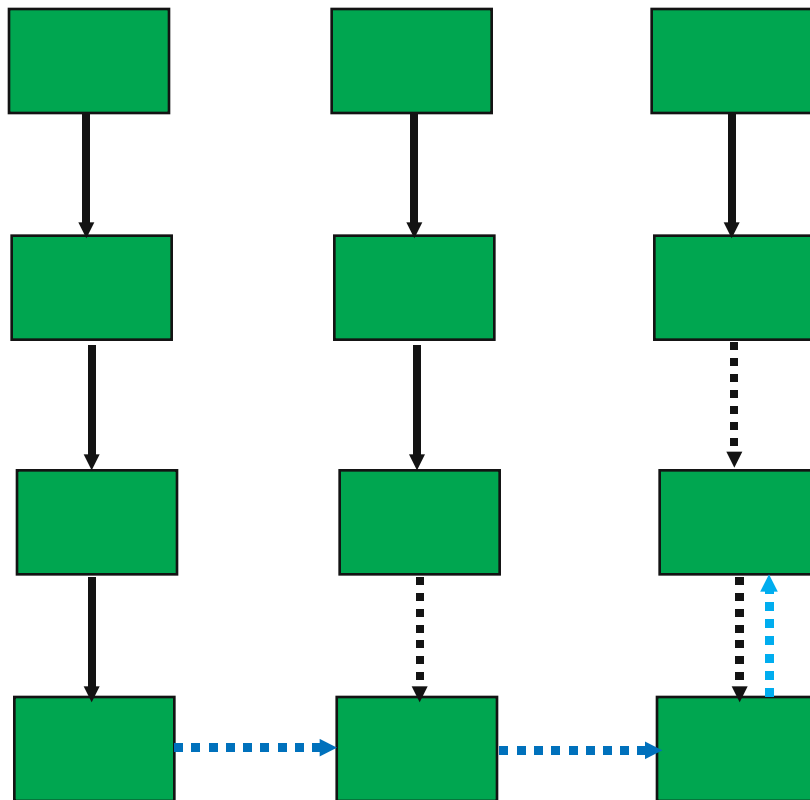
- Do i=1,n\_neighbors  
    Call MPI\_Irecv(inedge(1,i), len, MPI\_REAL, nbr(i), tag,&  
                  comm, requests(i), ierr)  
Enddo  
Do i=1,n\_neighbors  
    Call MPI\_Send(edge(1,i), len, MPI\_REAL, nbr(i), tag,&  
                  comm, ierr)  
Enddo  
Call MPI\_Waitall(n\_neighbors, requests, 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)

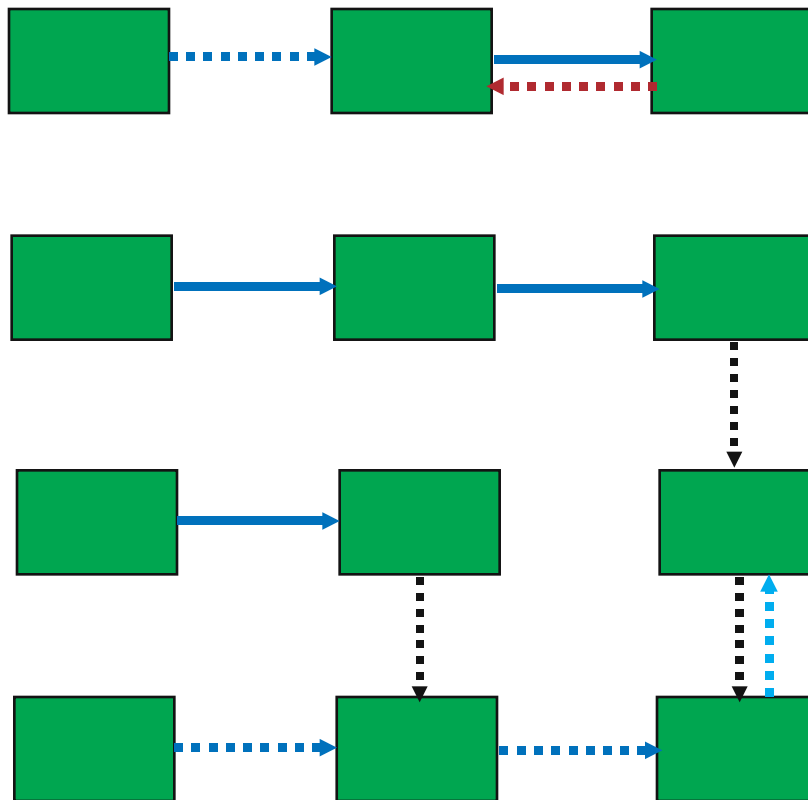
# Mesh Exchange - Step 1

- Exchange data on a mesh



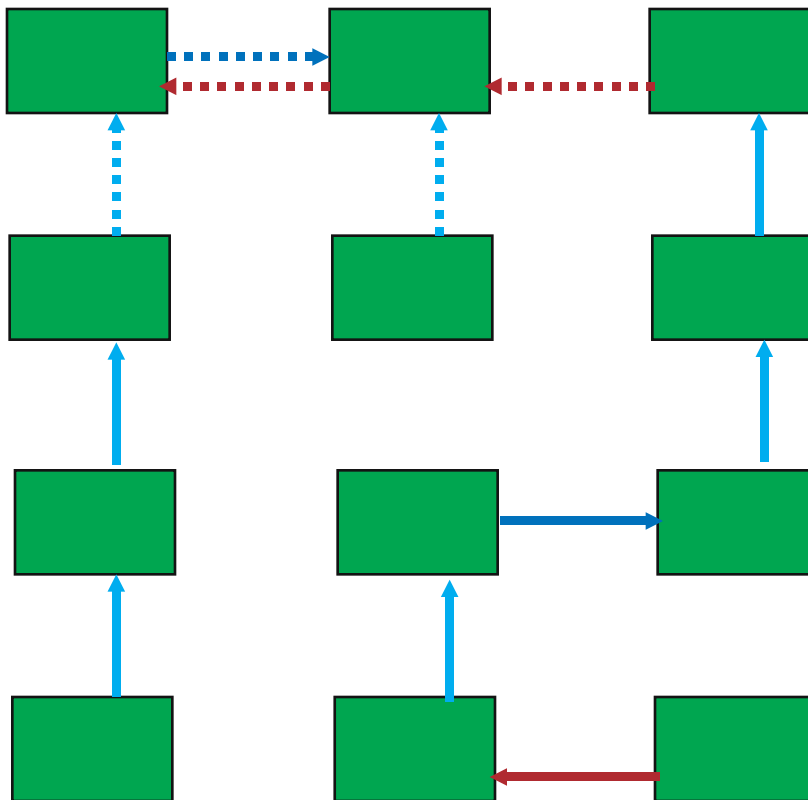
# Mesh Exchange - Step 2

- Exchange data on a mesh



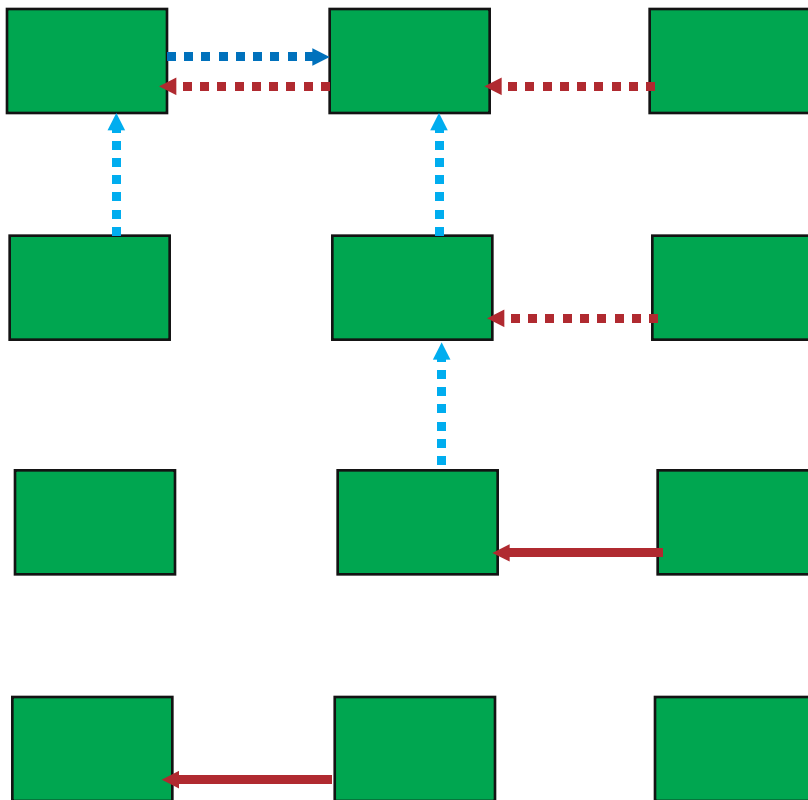
# Mesh Exchange - Step 3

- Exchange data on a mesh



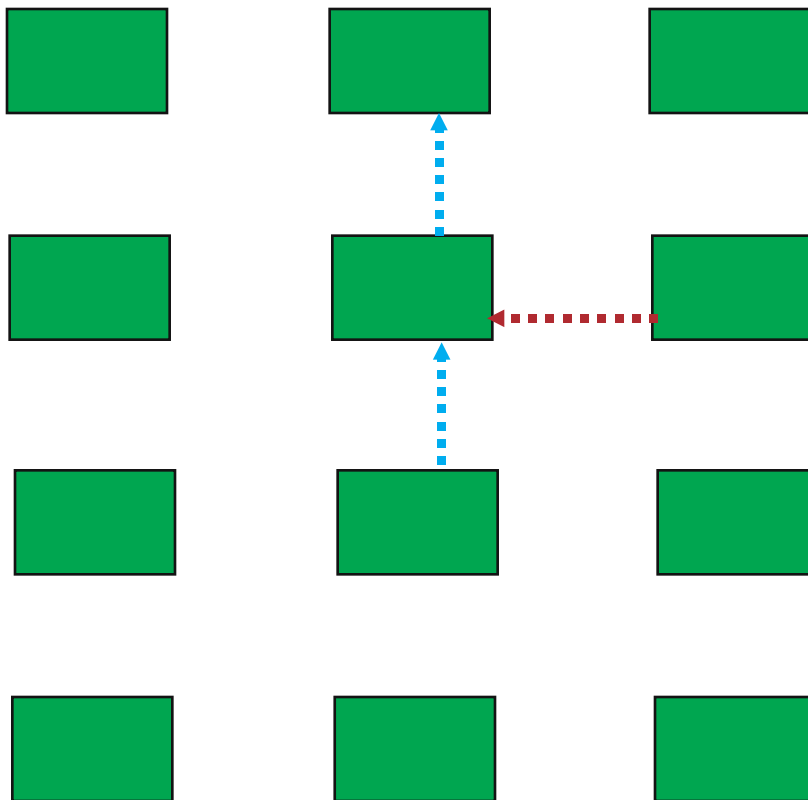
# Mesh Exchange - Step 4

- Exchange data on a mesh



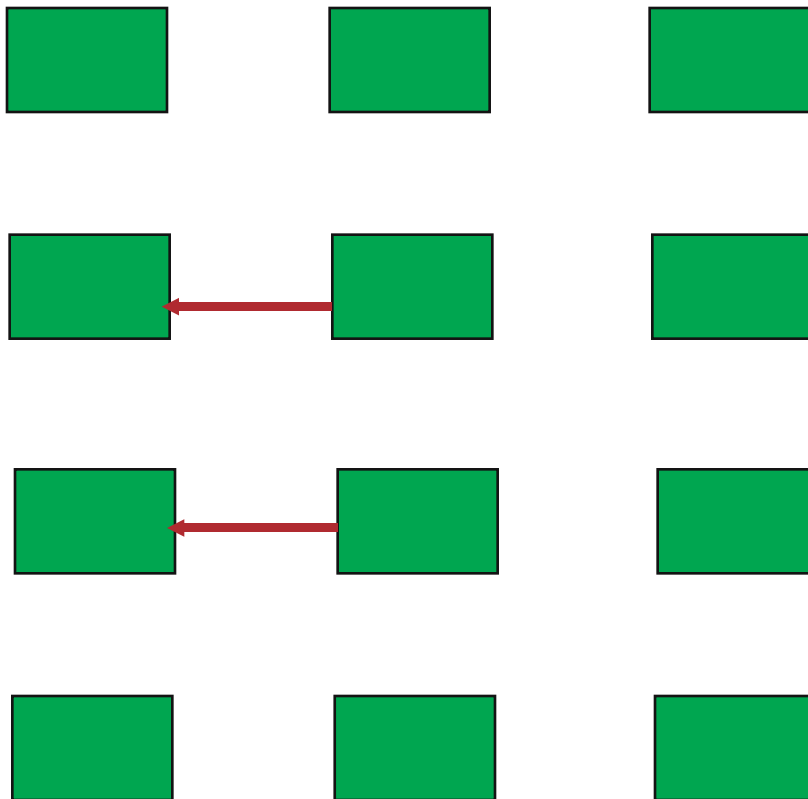
# Mesh Exchange - Step 5

- Exchange data on a mesh

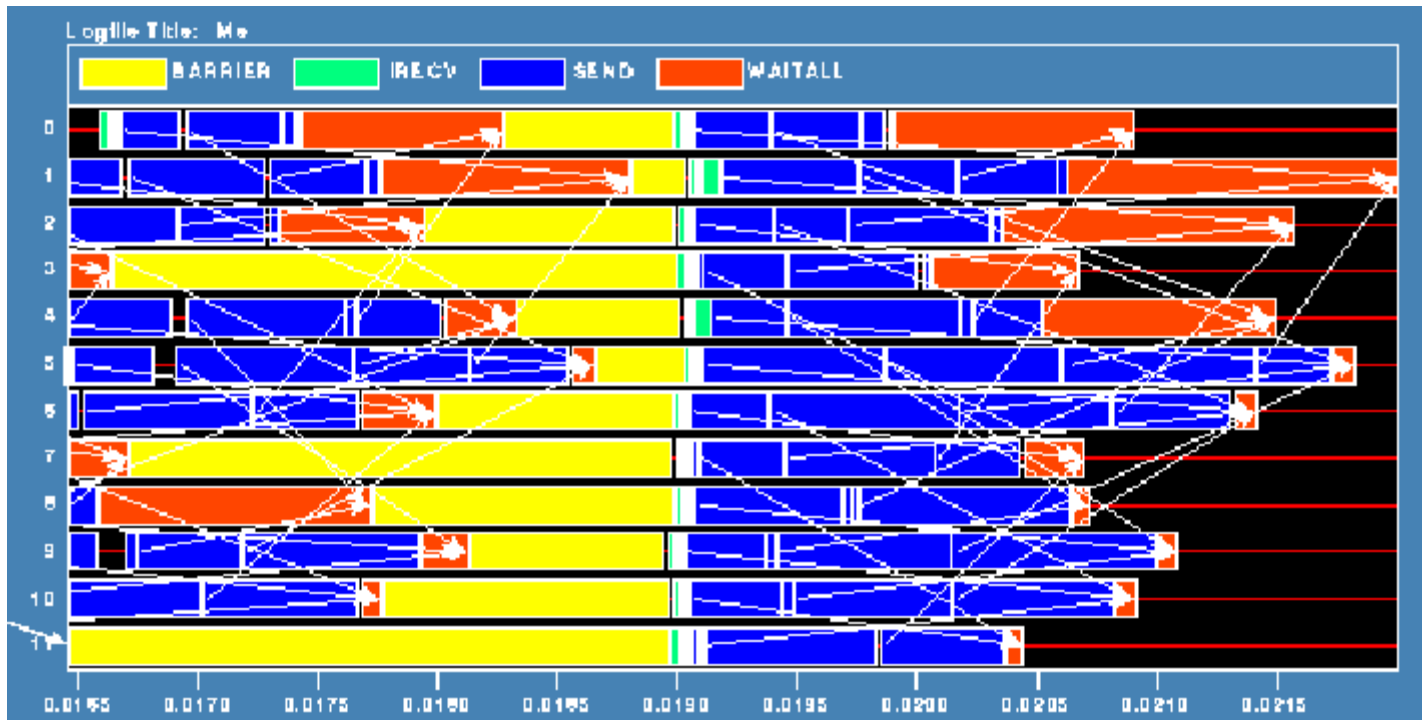


# Mesh Exchange - Step 6

- Exchange data on a mesh

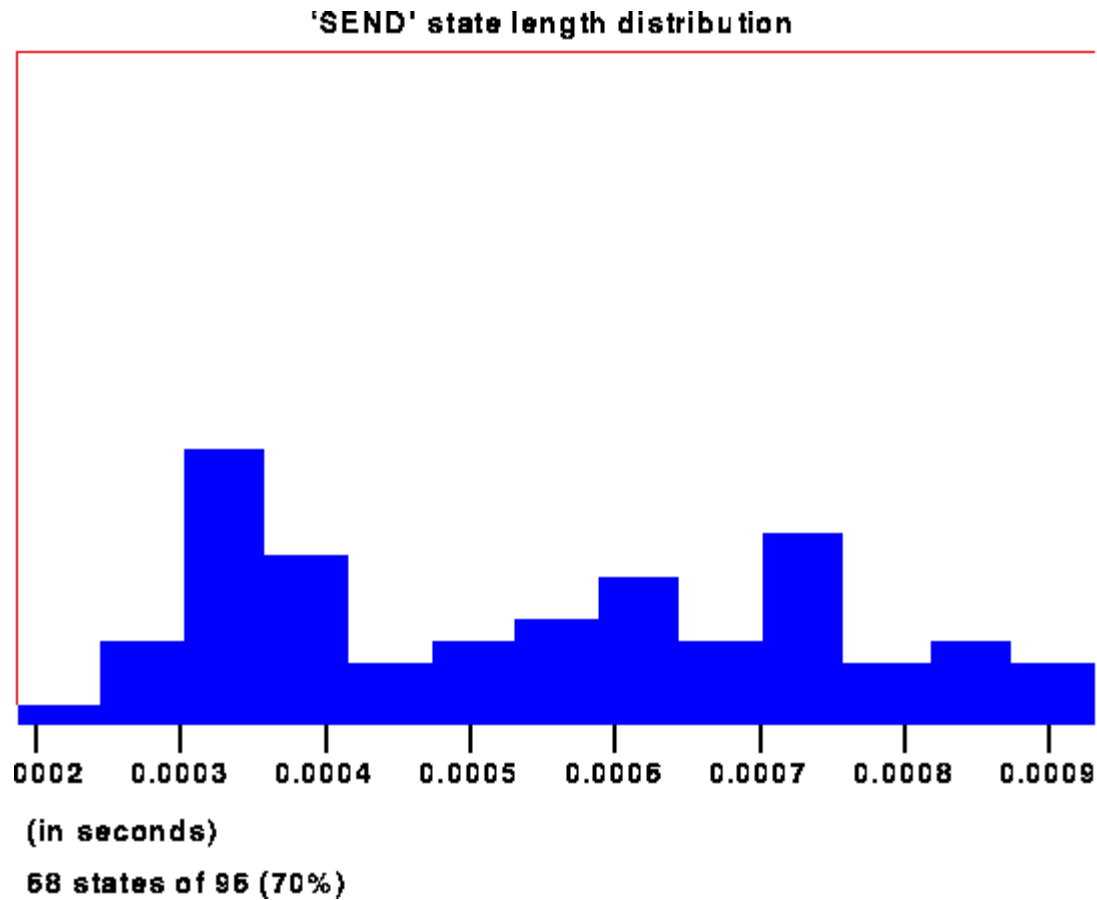


# Timeline from IBM SP



- Note that process 1 finishes last, as predicted

# Distribution of Sends



# Why Six Steps?

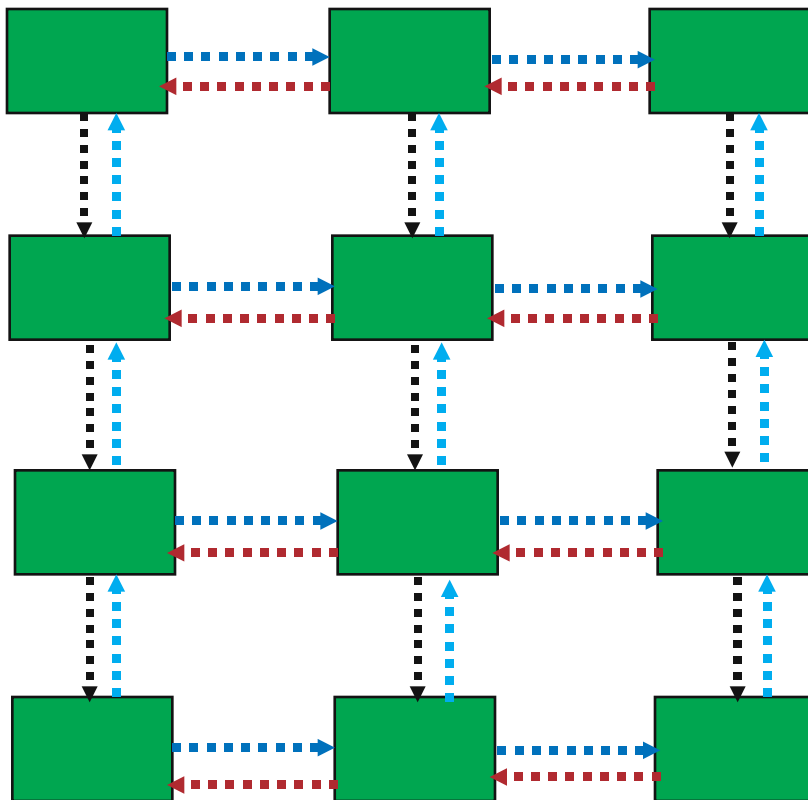
- Ordering of Sends introduces delays when there is contention at the receiver
- Takes roughly twice as long as it should
- Bandwidth is being wasted
- Same thing would happen if using memcpy and shared memory
- The interference of communication is why adding an MPI\_Barrier (normally an unnecessary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

## Fix 2: Use Irecv and Irecv

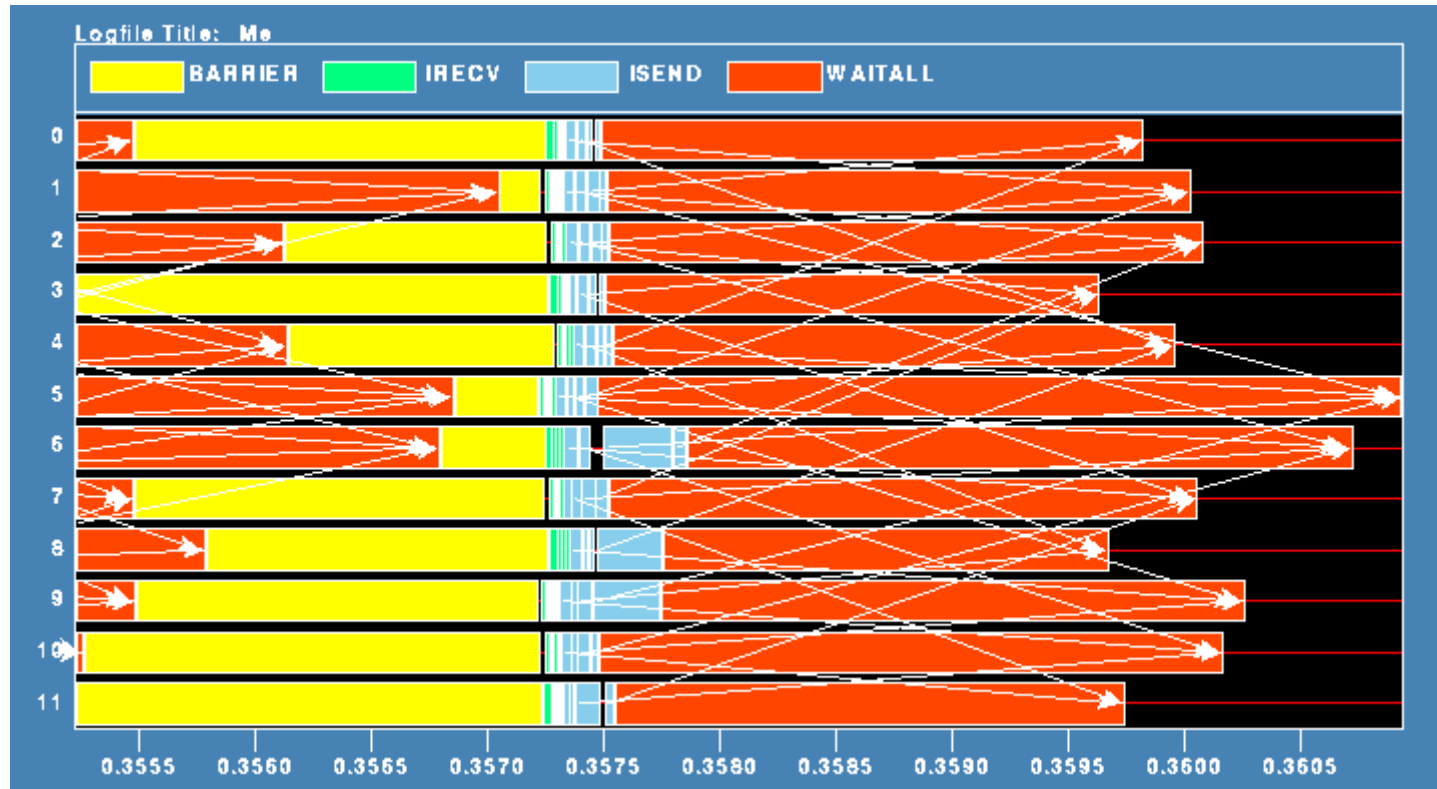
- Do i=1,n\_neighbors  
    Call MPI\_Irecv(inedge(1,i),len,MPI\_REAL,nbr(i),tag,&  
                  comm, requests(i),ierr)  
  
Enddo  
Do i=1,n\_neighbors  
    Call MPI\_Isend(edge(1,i), len, MPI\_REAL, nbr(i), tag,&  
                  comm, requests(n\_neighbors+i), ierr)  
  
Enddo  
Call MPI\_Waitall(2\*n\_neighbors, 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-recv 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:  

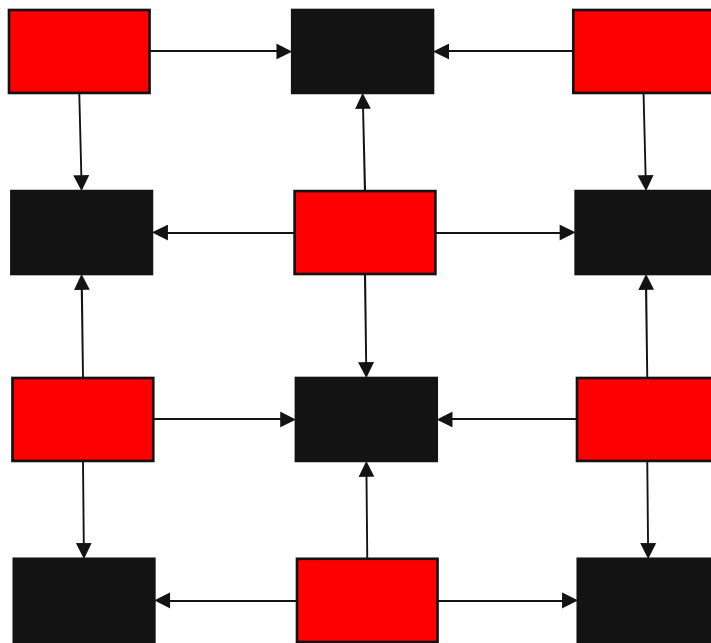
```
int dims[2], periodic[2];  
for (i=0; i<2; i++) { dims[i] = 0; periodic[i] = 0; }  
MPI_Dims_create( size, 2, dims );  
MPI_Cart_create( MPI_COMM_WORLD, 2, dims, periodic, 1,  
&newComm );
```
- The “1” tells `MPI_Cart_create` to reorder the mapping of processes to create a “better” communicator 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

# Discovering Performance Opportunities

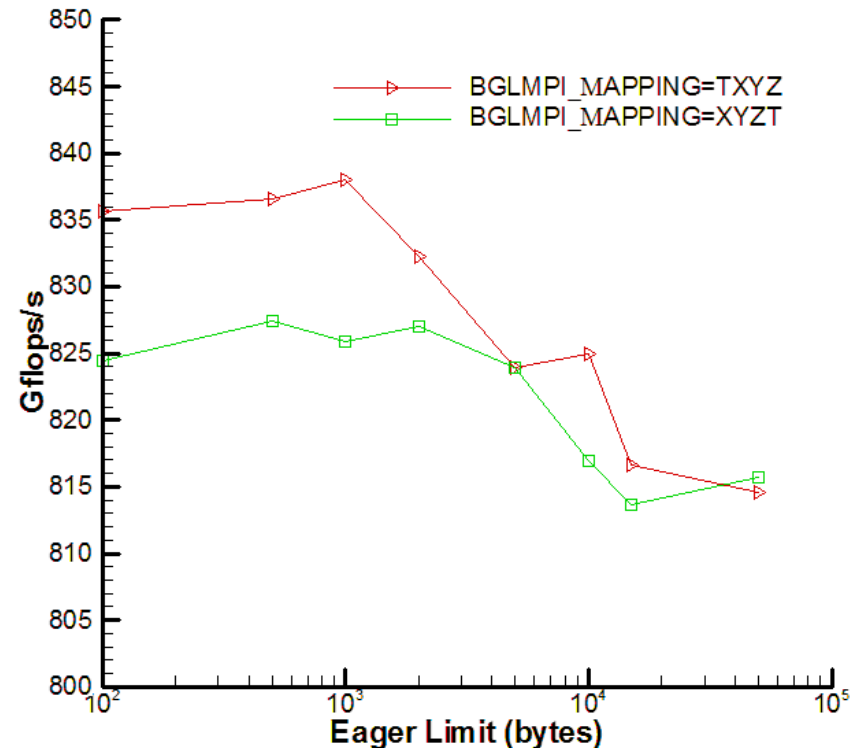
- Ratios of a single sender to all processes sending (in rate)
- *Expect* a factor of roughly 2 (since processes must also receive)

System	4 neighbors		8 Neighbors	
		Periodic		Periodic
BG/L	2.24		2.01	
BG/L, VN	1.46		1.81	
XT3	7.5	8.1	9.08	9.41
XT4	10.7	10.7	13.0	13.7
XT4 SN	5.47	5.56	6.73	7.06

- BG gives roughly double the halo rate. XT<sub>n</sub> 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*

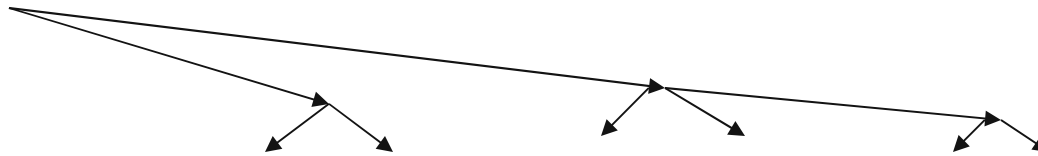
- On BG/P, the environment variable to control process mapping is BGML\_MAPPING
- If you use BGLMPI\_MAPPING as needed on BG/L, you will not get the expected mapping, and no warning message
- It is better to do this (portably) in your program than to count on the vendors to remember the names of their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

# MPI 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

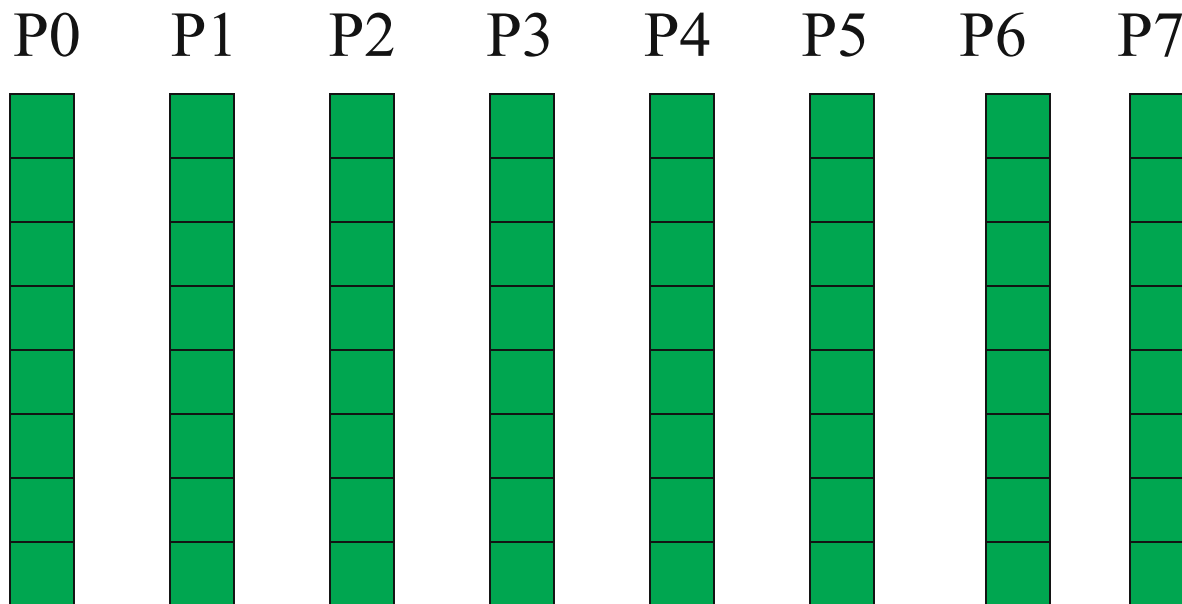
- `MPI_Bcast( buf, 100000, MPI_DOUBLE, ... );`
- Use a tree-based distribution:



- 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  
 $O(n \log p)$

# When not to use Collective Operations

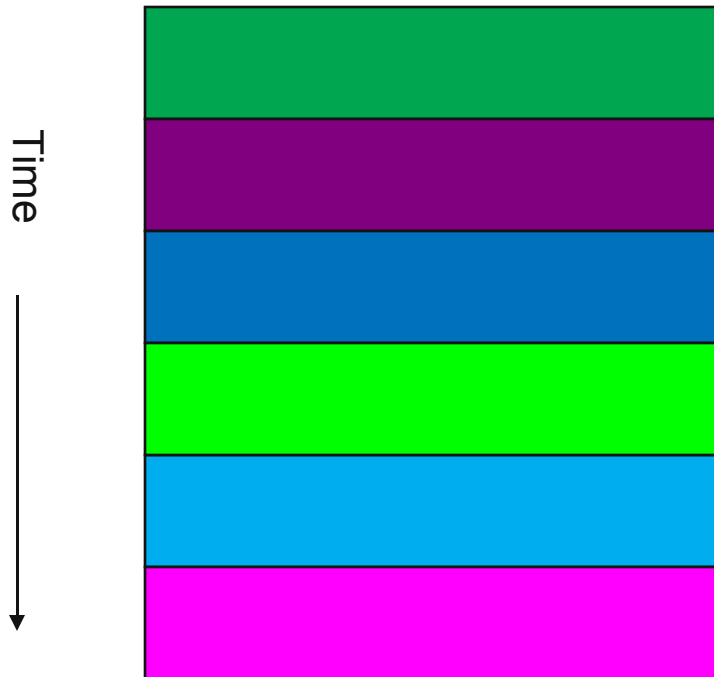
- Sequences of collective communication can be pipelined for better efficiency
- Example: Processor 0 reads data from a file and broadcasts it to all other processes.
  - Do  $i=1,m$ 
    - if (rank .eq. 0) read \*, a
    - call mpi\_bcast( a, n, MPI\_INTEGER, 0, comm, ierr )
  - EndDo
  - Takes  $m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

# Pipeline the Messages

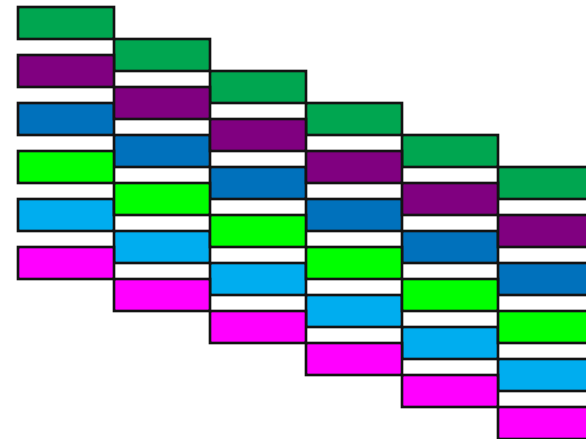
- Processor 0 reads data from a file and sends it to the next process. Other forward the data.
  - Do i=1,m
    - if (rank .eq. 0) then
      - read \*, a
      - call mpi\_send(a, n, type, 1, 0, comm,ierr)
    - else
      - call mpi\_recv(a,n,type,rank-1, 0,comm,status, ierr)
      - call mpi\_send(a,n,type,next,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

# Concurrency between Steps

■ Broadcast:



■ Pipeline



Each broadcast takes less time than “optimized” version, but total time is longer

Total time  $\neq \Sigma$ time each

Another example of deferring synchronization.

Always evaluate your strategy in the context of the big picture

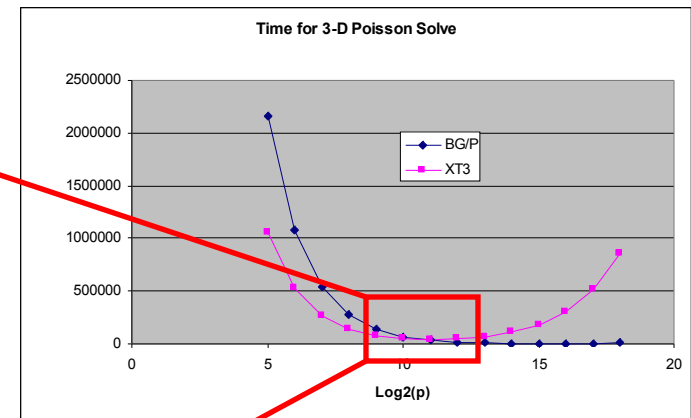
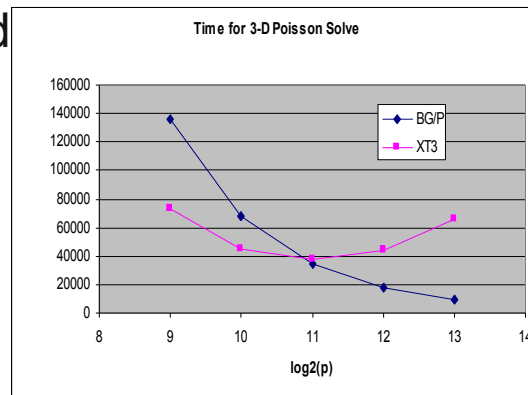
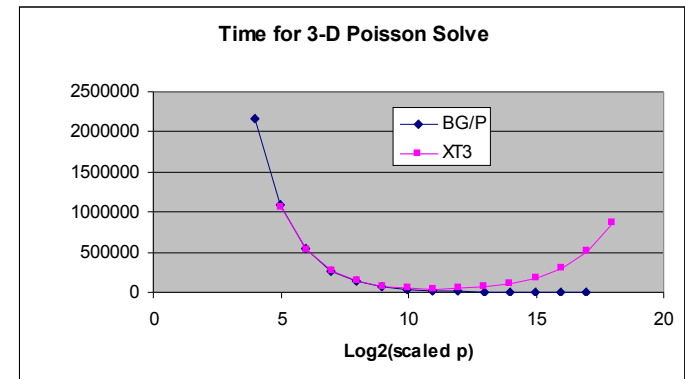
Be careful of “peephole optimization”

# Importance of Fast Collectives: A Simple Example

- Implicit linear methods (also used in solution of nonlinear equations) requires global communication
  - Either iterations proportional to  $p^{1/3}$  or preconditioned methods including dot products that scale as  $\log p$
- $T = N^3/p + 6(s+r(N/p^{1/3})^2) + 2(s+r)\log p$
- Log  $p$  term can dominate for large  $p$ ;  $T$  has a minimum at finite  $p$
- Tree network in BG/P addresses the  $\log p$  term:
  - Cost is much lower than  $(s+r) \log p$ ; grows very slowly.
  - Measured BG/L times: 10us for MPI\_Allreduce on 1024 processor; using the Torus instead of the Tree gives  $>120$  us.
- See next slide for an example of the benefit of the tree network...

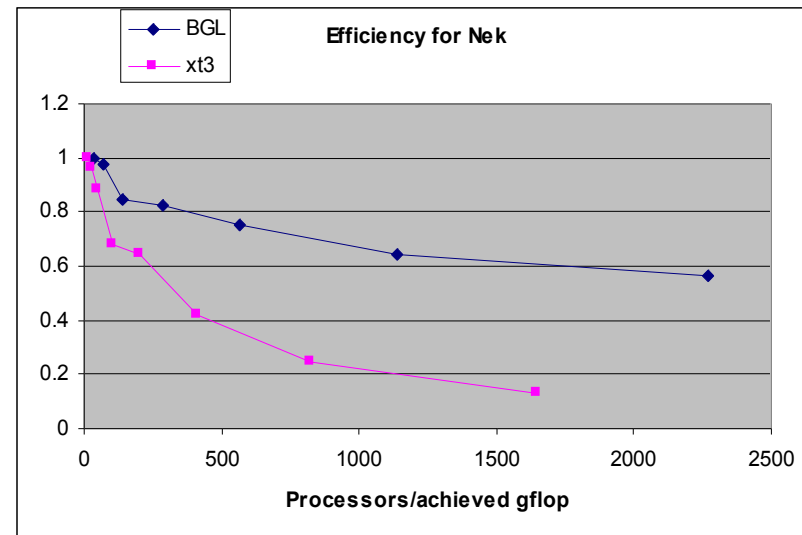
# Example Analysis: 3-D Poisson solve with Multigrid-preconditioned CG

- Strong scaling analysis for 1000 cube
- Thanks to Paul Fischer and Dinesh Kaushik
- Model contains:
  - Neighbor exchanges
  - Fast coarse-grid solve
  - BG/P uses fast tree network
- Results show key role of fast tree in maintaining scalability
- Qualitatively matches measured Nek performance (next slide)



# Measured Performance of Nek5000

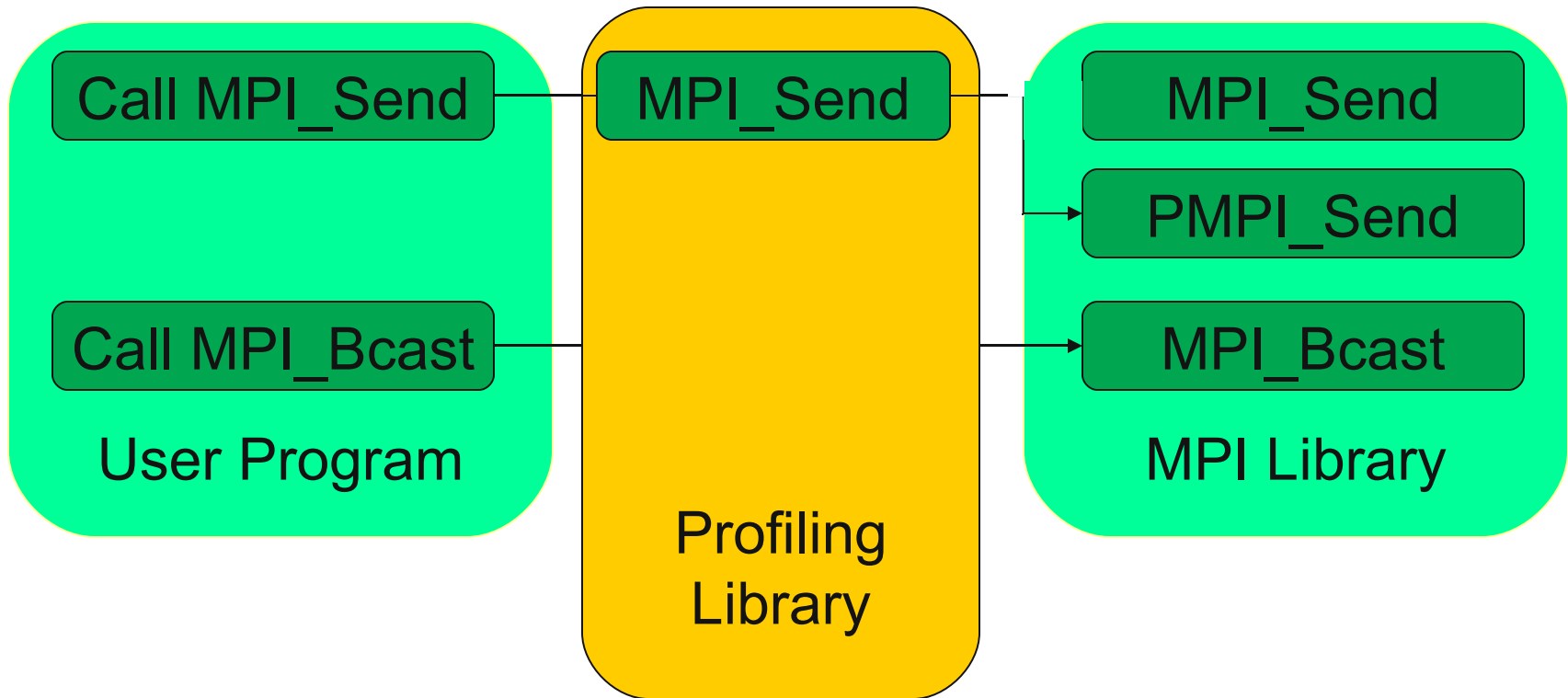
- Simple MHD problem; compares BG/L to XT3
- 10.7M point incompressible MHD (80 M dof)
- To normalize for the different processor performance, scaled so that 1 “processor” has 1 GF of achieved performance at 16 real processors (1/.45 BG/L processors and 1/1.24 XT3 processors)
- BG scaling much better per achieved performance, not just per processor



# *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 behavior
- 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*

# The MPI Profiling Interface



# Using the Profiling Interface

```
static int nsend = 0;

int MPI_Send( void *start, int count, MPI_Datatype datatype, int
             dest,
             int tag, MPI_Comm comm )
{
    nsend++;
    return PMPI_send(start, count, datatype,
                    dest, tag, comm);
}
```

# Using the Profiling Interface (2)

```
static int nAllreduce = 0;
static double syncAllreduce = 0.0;
static double tAllreduce = 0.0;

int MPI_Allreduce( void *inBuf, void *outBuf, int count, MPI_Datatype type,
                  MPI_Op op, MPI_Comm comm )
{
    double ts,te,ta;
    int    err;

    nAllreduce++;
    ts = MPI_Wtime();
    MPI_Barrier( comm );
    te = MPI_Wtime();
    err = PMPI_Allreduce( inBuf, outBuf, count, type, op, comm );
    ta = MPI_Wtime();
    syncAllreduce += te - ts;
    tAllreduce += ta - te;
    return err;
}
```

# Conclusions and Recommendations

- 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)
- Steps in Tuning Your Code
  - Estimate Performance
    - *Simple “back of the envelope” estimators are often sufficient*
    - *Understand where scalability problems may arise*
  - Measure achieved performance
    - *Look for under-achieving parts of your code*
  - Exploit available tools to understand possible sources of problems
  - MPI Profiling interface gives you access to ways to diagnose performance problems

# Hands On Exercises

- Explore different approaches for neighbor communication
  - Using halo code provided, try
    - *Different process mappings*
    - *Different MPI send/receive modes. For example, try `MPI_Irsend`*
    - *Try changing the eager buffer limit*
    - *Try to get multiple communication channels working*
      - Measure performance to test your code
- Use the MPI Profiling interface in your code to measure performance
  - Implement the MPI routines of interest using PMPI
  - For example, replace `MPI_Allreduce` with `PMPI_Barrier/PMPI_Allreduce` to measure load-imbalance separate from `Allreduce`