

# Final Comments on Cache Oblivious Algorithms; Parallel Performance

William Gropp



## Limitations of the Cache Oblivious Approach

---

- Why hasn't CO taken over the world?
  - ◆ Constant terms are important
    - For example, the number of cache misses in the transpose algorithm could be double because of misalignments of cache lines with respect to the subdivisions formed by the CO approach
  - ◆ The performance of a CO algorithm depends critically on the “kernel” – the part that actually performs the computation



2

## Limitations of the Complexity Model

---

- The complexity model weights computation and cache misses equally
  - ◆ A cache miss to main memory is typically 100's of cycles
- Only one level of cache is modeled
  - ◆ Relies on recursive subdivision to make good use of higher levels of cache
    - L1 miss is  $O(10)$  cycles; L2 is  $O(100)$ .
- No overlap of work and cache misses
  - ◆ Probably the critical flaw in the model for computation-bound operations
  - ◆ The opposite of the memory bandwidth based bounds – we assumed a full overlap



3

## Prefetch

---

- A major cost of memory access is latency in load request
- Standard technique is a split operation—change a load into initiate load and complete load
- Most processors do this now with all load operations, permitting several loads to be outstanding while other operations continue
  - ◆ Processor waits when the result of the load is needed
  - ◆ May reorder operations to permit operations independent of this load to start
- Problem is to issue the load early enough to hide latency and to issue enough loads
  - ◆ The number of loads needed is given by Little's law:
    - #pending = latency / time to complete each request
    - For BG/L, this is  $60 / 8$  or 8; for a typical commodity processor, this is 32 or more
  - ◆ Few systems are designed to support this number of outstanding loads



4

# Prefetch in Practice

- $d = a(i+cachewords,j)$ 
  - ◆ Assumes that the compiler doesn't move or even delete the reference.
  - ◆ Use  $d$  to force compiler to retain reference (but the compiler may still move it)
  - ◆ Uses up a register (load must have a target)
  - ◆ Be careful to avoid access past the end of the array
  - ◆ Costly if data is already in cache
- Directives or pragmas
  - ◆ Vendor-specific (#pragmas may generate warnings on other systems)
- Language Question
  - ◆ What is the abstraction or language concept that would allow a compiler to accurately issue prefetch without forcing the programmer to program for details of the memory architecture?



5

# Parallel Architecture Issues

- An easy way to build a parallel computer is to connect complete computers together
- Each system communications by sending *messages*
- This is the *distributed memory* model because memory is distributed amongst the processors (nodes) and is only directly accessible from the system in which the memory is installed.
- Extending a performance model to this system is relatively easy, though there are pitfalls.



6

# Latency and Bandwidth

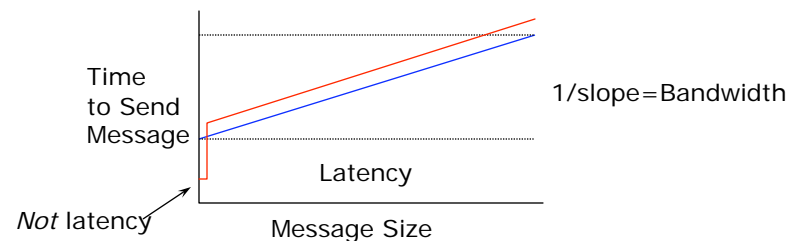
- Simplest model  $s + r n$
- $s$  includes both hardware (gate delays) and software (context switch, setup)
- $r$  includes both hardware (raw bandwidth of interconnection and memory system) and software (packetization, copies between user and system)
- head-to-head and pingpong values may differ



7

# Interpreting Latency and Bandwidth

- Bandwidth is the inverse of the slope of the line  
 $\text{time} = \text{latency} + (1/\text{rate}) \text{ size\_of\_message}$
- Latency is sometimes described as "time to send a message of zero bytes". This is true *only* for the simple model. The number quoted is sometimes misleading.



8

# Including Contention

- Lack of contention greatest limitation of latency/bandwidth model
- Hyperbolic model of Stoica, Sultan, and Keyes provides a way to estimate effects of contention for different communication patterns; see <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.11.6377>



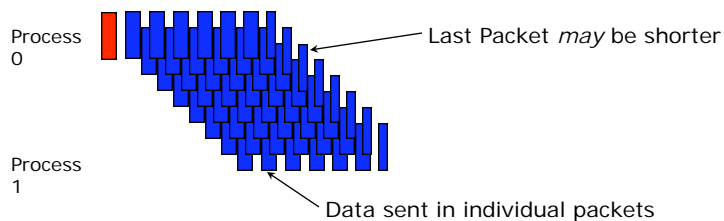
# Other Impacts on Performance

- Contention
  - ◆ In the network
  - ◆ At the processors
- Memory Copies
- Packet sizes and stepping



# Packetization

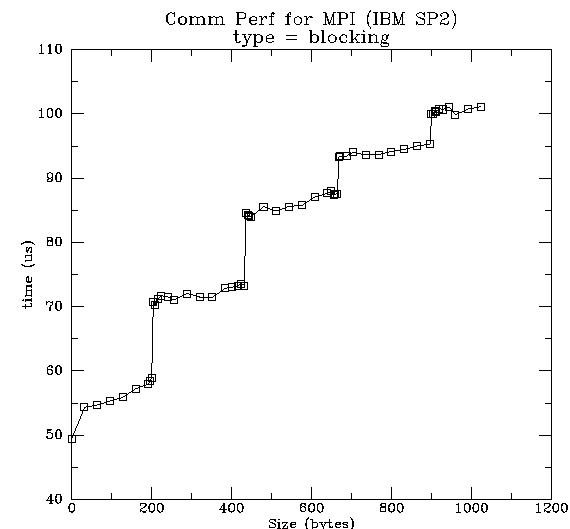
- Some networks send data in discrete chunks called *packets*



Introduces a  $\text{ceil}(n/\text{packet\_size})$  term  
Staircase appearance of performance graph



# Example of Packetization



Packets contain 232 bytes of data. (first is 200 bytes, so MPI header is probably 32 bytes).

Data from mptest, available at <ftp://ftp.mcs.anl.gov/pub/mpi/misc/perftest.tar.gz>



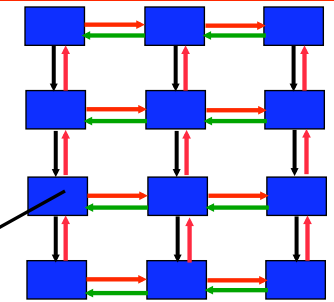
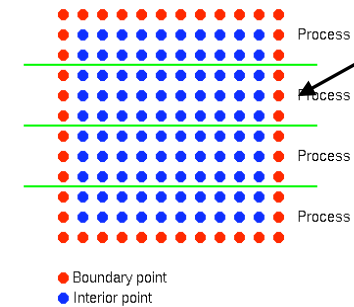
# Basic MPI: Looking Closely at a Simple Communication Pattern

- 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



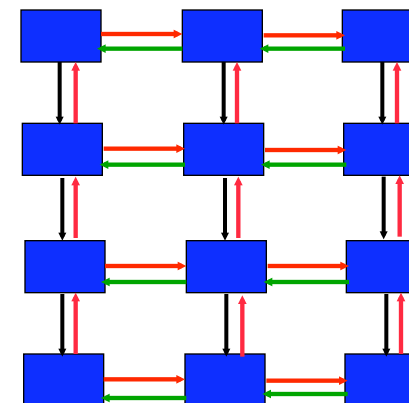
# Scalability of Mesh Exchange

- 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 =  $2(s + r n^2)$  (both sides are  $n$ )
    - 2-d =  $4(s + r n^2/\sqrt{p})$  (1 side is  $n/\sqrt{p}$ , the other is  $n$ )
    - 3-d =  $6(s + r (n/p^{1/3})^2)$  (both sides are  $n/p^{1/3}$ )
  - ◆ Best approach is to make these relative to floating-point work (this is the dimensionless quantity):
    - 1-d =  $2(s + r n^2) / n^3 f$
- 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



17

17

## 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)



18

18

## 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					



19

19

## 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?



20

20

# 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)

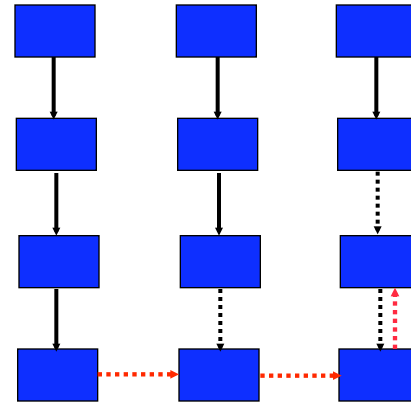


21

21

# Mesh Exchange - Step 1

- Exchange data on a mesh

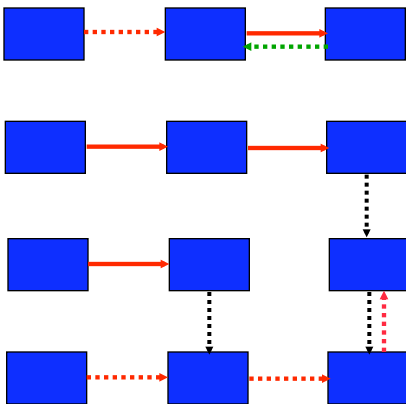


22

22

# Mesh Exchange - Step 2

- Exchange data on a mesh

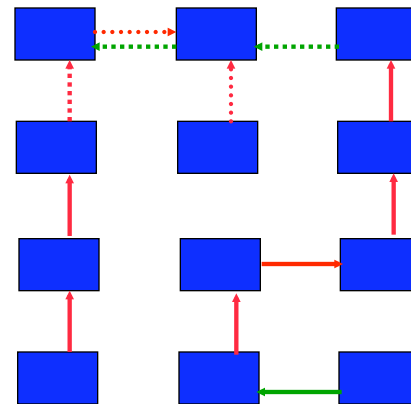


23

23

# Mesh Exchange - Step 3

- Exchange data on a mesh

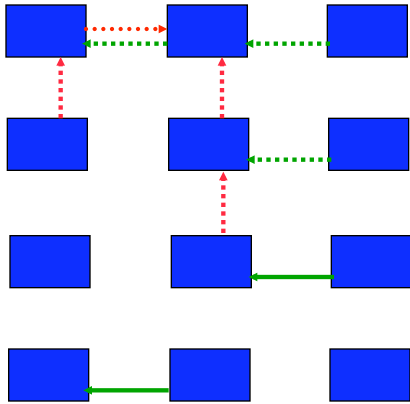


24

24

## Mesh Exchange - Step 4

- Exchange data on a mesh

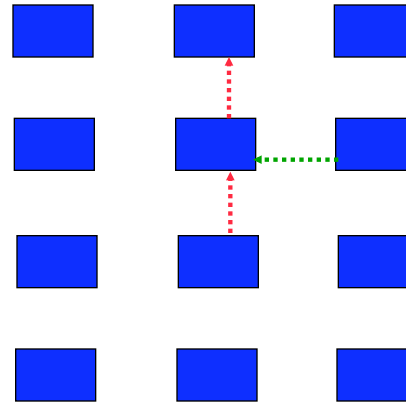


25

25

## Mesh Exchange - Step 5

- Exchange data on a mesh

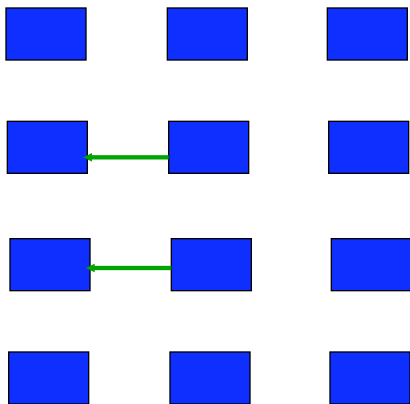


26

26

## Mesh Exchange - Step 6

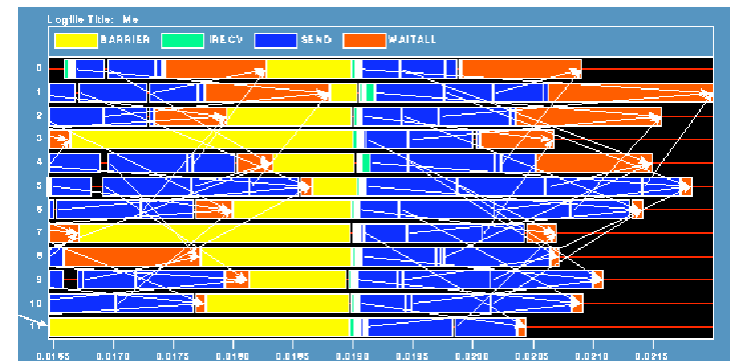
- Exchange data on a mesh



27

27

## Timeline from IBM SP



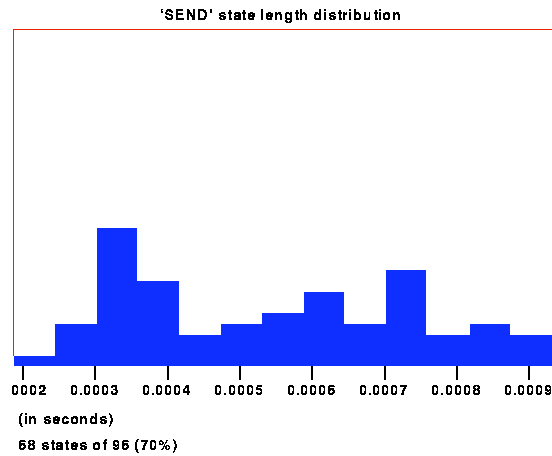
- Note that process 1 finishes last, as predicted



28

28

# 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 :)



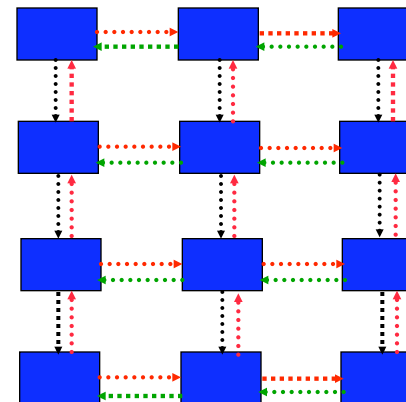
# Fix 2: Use Isend and Irecv

- 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)`

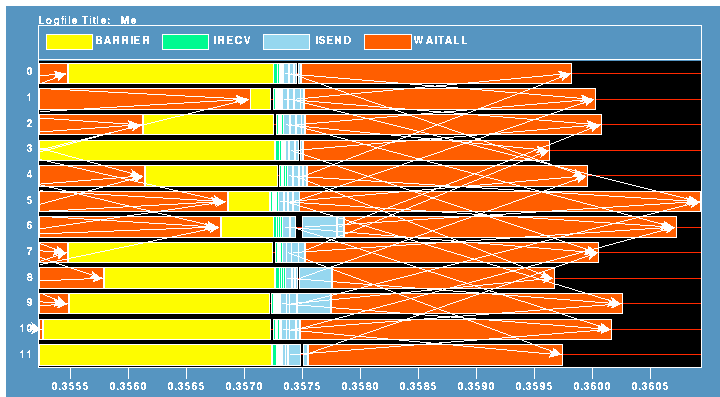


# Mesh Exchange - Steps 1-4

- 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-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 "Leadership Class" 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"



## Method 1: Use Irecv and Send

- 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).
  - Quiz for the audience: Why?



37

## Method 2: Use Isend and Irecv

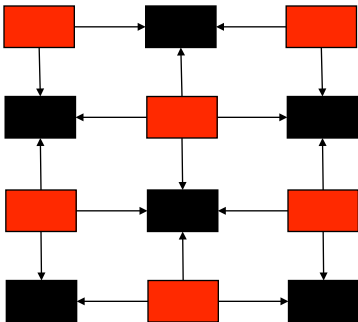
- Do  $i=1, n\_neighbors$   
Call `MPI_Irecv(inedge(1,i), len, MPI_REAL, nbr(i), tag, & comm, requests(i), ierr)`
- Enddo
- Do  $i=1, n\_neighbors$   
Call `MPI_Isend(edge(1,i), len, MPI_REAL, nbr(i), tag, & comm, requests(n_neighbors+i), ierr)`
- Enddo
- Call `MPI_Waitall(2*n_neighbors, requests, statuses,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



38

## 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.



39

39

## Halo Exchange on BG/L

	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

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



40

## Halo Exchange on BG/L

	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

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



41

## Halo Exchange on BG/P

	4 Neighbors		8 Neighbors	
	Irecv/Send	Irecv/Isend	Irecv/Send	Irecv/Isend
World	208	328	184	237
Even/Odd	219	327	172	243
Cart_create	301	581	242	410

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



42

## Halo Exchange on BG/P

	4 Neighbors		8 Neighbors	
	Irecv/Send	Irecv/Isend	Irecv/Send	Irecv/Isend
World	112	132	81	89
Even/Odd	74	84	50	50
Cart_create	109	132	84	89
World(txyz)	132	177	92	108
Even/Odd(txyz)	78	84	55	55
Cart_create(txyz)	132	177	84	108

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



43

## Halo Exchange on Cray XT4

	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

- 1024 processes, 2000 doubles to each neighbor



44

# Halo Exchange on Cray XT4

	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



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

45

# 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
  - ♦ Not clear if MPI\_Cart\_create creates optimal map on Cray
  - ♦ On BG/P, there is an environment variable controlling mapping. MPI\_Cart\_create has no effect in vn mode :(
- 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
    - Either contention along links (which should not happen when MPI\_Cart\_create is used) or contention at the destination node.
  - ♦ To see this, consider the performance of a single process sending to four neighbors



46

## 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
BG/P	1139	1136	892	892
BG/P, VN	468	468	600	601
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



47

## Discovering Performance Opportunities

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

System	4 neighbors		8 Neighbors	
		Periodic		Periodic
BG/L	2.2		2.0	
BG/L, VN	1.5		1.8	
BG/P	3.8		2.2	
BG/P, VN	2.6		5.5	
XT3	7.5	8.1	9.1	9.4
XT4	11	11	13	14
XT4 SN	5.5	5.6	6.7	7.1

- BG/L 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 (But is it topology routines or point-to-point communication? How would you test each hypothesis?)



48

# What Does All of This Mean?

- The simple  $s+rn$  model is a good starting point
  - ◆ But it can *overestimate* performance when there is contention on the links between processes
    - One “fix” is to reduce  $1/r$  (rate)
  - ◆ And it can *underestimate* performance when concurrency is possible on the links between processes
  - ◆ It also ignores the possibility of concurrency between computation and communication
    - Much as the Cache-Oblivious model ignores the possibility of overlap with cache misses and work



49

# Programming Model Considerations

- Process(or) topology influences performance
  - ◆ What role should the programming model take with respect to processor topology?
    - None: Assume complete interconnect
    - Total: Reflect underlying physical interconnect
    - Abstract: Assume some generic model
      - MPI sort of does this for the special case of regular grid (Cartesian) communication
- Other performance models include PRAM (assume infinite communication speeds) and Bulk Synchronous Programming (communication in phases, with overlap of computation)
  - ◆ BSP designed in concert with a program model (library) to make performance more predictable



50

# A Parallel Performance Model With Overlap

- The logP model
  - ◆ L – Latency (or delay) for a single short message (typically a word)
  - ◆ o – Overhead (time processor must devote to sending message)
  - ◆ g – “Gap” or time between consecutive message transmissions (basically controls bandwidth through  $L/g$  messages in flight)
  - ◆ P – number of processors
- Better separates time that it takes a message to move from one processor to another (L) from the time it takes a processor to manage that communication (o)



51