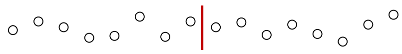
Relaxed Memory-Consistency Models

[Review](). Why are relaxed memory-consistency models needed?

How do relaxed MC models require programs to be changed?

The "safety net" between operations whose order needs to be guaranteed is often a *fence* instruction.



- The fence ensures that memory operations that are "younger" are not issued until the older mem ops have globally performed. The newer instruction must
  - wait until all older writes have been posted on the bus (or received InvAck);
  - wait until all older reads have completed;
  - flush the pipeline to avoid issuing younger mem ops early

- Programmers must insert fences.

What if amateur programmers perform their own synchronization, and forget fences?

A continuum of consistency models

Sequential consistency is one view of what a programming model should guarantee.

Let us introduce a way of diagramming consistency models. Suppose that—

- The value of a particular memory word in processor 2's local memory is 0.
- Then processor 1 writes the value 1 to that word of memory. Note that this is a remote write.
- Processor 2 then reads the word. But, being local, the read occurs quickly, and the value 0 is returned.

What's wrong with this?

This situation can be diagrammed like this (the horizontal axis represents time):

P_1:	$W(x)1$	
P_2:		$R(x)0$

Depending upon how the program is written, it may or may not be able to tolerate a situation like this.

But, in any case, the programmer must understand what can happen when memory is accessed in a DSM system.

Sequential consistency

Sequential consistency: *The result of any execution is the same as if*

- *the memory operations of all processors were executed in some sequential order, and*
- *the operations of each individual processor appear in this sequence in the order specified by its program.*

Sequential consistency does *not* mean that writes are instantly visible throughout the system (it would be impossible to implement that anyway).

The example below illustrates two sequentially consistent executions.

Note that a read from P_2 is allowed to return an out-of-date value (because it has not yet "seen" the previous write).

P_1:	$W(x)1$		
P_2:		$R(x)0$	$R(x)1$

P_1:	$W(x)1$		
P_2:		$R(x)1$	$R(x)1$

From this we can see that running the same program twice in a row in a system with sequential consistency may not give the same results.

Causal consistency

The first step in weakening the consistency constraints is to distinguish between events that are potentially *causally* connected and those that are not.

Two events are causally related if one can influence the other.

P_1:	$W(x)1$		
P_2:		$R(x)1$	$W(y)2$

Here, the write to x could influence the write to y, because

On the other hand, without the intervening read, the two writes would not have been causally connected:

P_1:	$W(x)1$	
P_2:		$W(y)2$

The following pairs of operations are potentially causally related:

- A read followed by a later write by the same processor.
- A write followed by a later read to the same location.
- The transitive closure of the above two types of pairs of operations.

Operations that are not causally related are said to be *concurrent*.

Causal consistency: *Writes that are potentially causally related must be seen in the same order by all processors.*

Concurrent writes may be seen in a different order by different processors.

Here is a sequence of events that is allowed with a causally consistent memory, but disallowed by a sequentially consistent memory:

P_1:	$W(x)1$		$W(x)3$		
P_2:		$R(x)1$	$W(x)2$		
P_3:		$R(x)1$		$R(x)3$	$R(x)2$
P_4:		$R(x)1$		$R(x)2$	$R(x)3$

[Why is this not]() allowed by sequential consistency?

Why is this allowed by causal consistency?

What is the violation of causal consistency in the sequence below?

P_1:	$W(x)1$				
P_2:		$R(x)1$	$W(x)2$		
P_3:				$R(x)2$	$R(x)1$
P_4:				$R(x)1$	$R(x)2$

Without the $R(x)1$ by P_2, this sequence would've been causally consistent.

Implementing causal consistency requires the construction of a dependency graph, showing which operations depend on which other operations.

Processor consistency

Causal consistency requires that all processes see causally related writes from *all* processors in the same order.

The next step is to relax this requirement, to require only that writes from the *same* processor be seen in order. This gives processor consistency.

Processor consistency: *Writes performed by a single processor are received by all other processors in the order in which they were issued.*

Writes from different processors may be seen in a different order by different processors.

Processor consistency would permit this sequence that we saw violated causal consistency:

P_1:	$W(x)1$		
P_2:		$R(x)1$ $W(x)2$	
P_3:			$R(x)2$ $R(x)1$
P_4:			$R(x)1$ $R(x)2$

Another way of looking at this model is that all writes generated by different processors are considered to be concurrent.

Note: Some definitions of processor consistency require cache coherence too. Processor consistency *without* cache coherence is called PRAM consistency.

Exercise: What is the strongest consistency model that each of the following satisfy?

P_1:	$W(x)1$		
P_2:		$R(x)1$ $W(x)2$	
P_3:			$R(x)1$ $R(x)2$
P_4:			$R(x)2$ $R(x)1$

P_1:	$W(y)1$		
P_2:		$R(x)1$ $W(y)2$	
P_3:			$R(y)1$ $R(y)2$
P_4:			$R(y)2$ $R(y)1$

P_1:	$W(x)1$		
P_2:		$R(x)1$ $W(y)2$	
P_3:			$R(x)1$ $R(y)2$
P_4:			$R(y)2$ $R(x)1$

Sometimes processor consistency can lead to counterintuitive results.

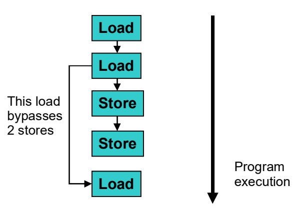
P_1:	P_2:
$a = 0$;	$b = 0$;
:	:
$a = 1$;	$b = 1$;
if (b == 0)	**if** (a == 0)
kill(p_2);	*kill*(p_1);

At first glance, it seems that no more than one process should be killed.

With processor consistency, however, it is possible for both to be killed. Explain how.

What processor consistency guarantees

- SC ensures ordering of
  - LD → LD
  - LD → ST
  - ST → LD
  - ST → ST
- PC removes the ST→LD constraint, with significant implications for ILP:
  - Values can be loaded into other caches, even if there's a store to the same location in some write buffer.
  - Loads do not wait for stores to complete ("perform"), they access the cache right away (without being speculative!).
  - A load dependent on an older store (in the same processor) can "bypass" (directly obtain the store value before it is stored).
- PC also removes write atomicity.



Implications

P1:	P2:	P1:	P2:
data = 2000;	**while** (*flag* == 0) {};	*flag1* = 1;	*flag2* = 1;
flag = 1;	print *data*;	**if** (*flag2* == 0)	**if** (*flag1* == 0)
		…	…

PC produces SC results, because ordering between 2 stores is preserved.

PC fails to produce SC results, because PC does not guarantee ordering betw. store & younger load

- How close is PC to programmers' expectation?
  - Most of the time, very close (e.g., post-wait synchronization works correctly)
  - Major OSes are ported to PC with relative ease
- Cases that cause errors in PC usually are due to races that also happen in SC.
  - However, debugging races in PC is more difficult.

Weak ordering

Processor consistency is still stronger than necessary for many programs, because it requires that writes originating in a single processor be seen in order everywhere.

But it is not always necessary for other processors to see writes in order—or even to see all writes, for that matter.

Suppose a processor is in a tight loop in a critical section, reading and writing variables.

Other processes aren't supposed to touch these variables until the process exits its critical section.

Under processor consistency, the memory has no way of knowing that other processes don't care about these writes, so it has to propagate all writes to all other processors in the normal way.

To relax our consistency model further, we have to divide memory operations into two classes and treat them differently.

- Accesses to *synchronization variables* are sequentially consistent.
- Accesses to other memory locations can be treated as concurrent.

This strategy is known as *weak ordering.*

With weak ordering, we don't need to propagate accesses that occur during a critical section.

We can just wait until the process exits its critical section, and then—

- make sure that the results are propagated throughout the system, and
- stop other actions from taking place until this has happened.

Similarly, when we want to enter a critical section, we need to make sure that all previous writes have finished.

These constraints yield the following definition:

Weak ordering: *A memory system exhibits weak ordering iff—*

1. *Accesses to synchronization variables are sequentially consistent.*
2. *No access to a synchronization variable can be performed until all previous writes have completed everywhere.*
3. *No data access (read or write) can be performed until all previous accesses to synchronization variables have been performed.*

Thus, by doing a synchronization before reading shared data, a process can be assured of getting the most recent values written by other processes before their immediately preceding *S*s.

Note that this model does not allow more than one critical section to execute at a time, even if the critical sections involve disjoint sets of variables.

This model puts a greater burden on the programmer, who must decide which variables are synchronization variables.

Weak ordering says that memory does not have to be kept up to date between synchronization operations.

This is similar to how a compiler can put variables in registers for efficiency's sake. Memory is only up to date when these variables are written back.

If there were any possibility that another process would want to read these variables, they couldn't be kept in registers.

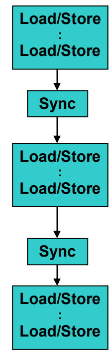
This shows that processes can live with out-of-date values, provided that they know when to access them and when not to.

The following is a legal sequence under weak ordering. Can you explain why?

P_1:	$W(x)1$	$W(x)2$	S		
P_2:			$R(x)2$	$R(x)1$	S
P_3:			$R(x)1$	$R(x)2$	S

Here's a sequence that's illegal under weak ordering. Why?

P_1:	$W(x)1$	$W(x)2$	S	
P_2:			S	$R(x)1$



Synch may be implemented as a lock acquire/release

Before a synch, all previous ops must finish.
Before any ld/st, all previous synch must finish.

Why safe? Typically within a critical section, we have made sure that only one process is inside, thus safe to reorder anything in the critical section.

Outside a critical section, we usually do not care about the order of mem ops (we would have used synchronization if we had cared).

How to know whether a particular ld/st serves as a synchronization point?

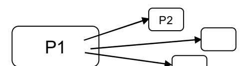
- Assume all atomic instructions are synchronization points
  - fetch-and-op, test-and-set
- Assume all load linked (LL) and store conditional (SC) are synchronization points

Release consistency

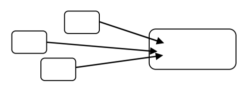
Weak ordering does not distinguish between entry to critical section and exit from it.

Thus, on both occasions, it has to take the actions appropriate to both:

- making sure that all locally initiated writes have been propagated to all other memories, and



- making sure that the local processor has seen all previous writes anywhere in the system.



If the memory could tell the difference between entry and exit of a critical section, it would only need to satisfy one of these conditions.

Release consistency provides two operations:

- *acquire* operations tell the memory system that a critical section is about to be entered.
- *release* operations say a c. s. has just been exited.

It is possible to acquire or release a single synchronization variable, so more than one critical section can be in progress at a time.

When an acquire occurs, the memory will make sure that all the local copies of shared variables are brought up to date.

When a release is done, the shared variables that have been changed are propagated out to the other processors.

But—

- doing an acquire does not guarantee that locally made changes will be propagated out immediately.
- doing a release does not necessarily import changes from other processors.

Here is an example of a valid event sequence for release consistency (A stands for "acquire," and Q for "release" or "quit"):

P_1:	$A(L)$	$W(x)1$	$W(x)2$	$Q(L)$			
P_2:					$A(L)R(x)2$	$Q(L)$	
P_3:							$R(x)1$

Note that since P_3 has not done a synchronize, it does not necessarily get the new value of x.

Release consistency: A system is release consistent if it obeys these rules:

1. *Before an ordinary access to a shared variable is performed, all previous acquires done by the process must have completed.*

2. *Before a release is allowed to be performed, all previous reads and writes done by the process must have completed.*
3. *The acquire and release accesses must be processor consistent.*

If these conditions are met, and processes use *acquire* and *release* properly, the results of an execution will be the same as on a sequentially consistent memory.

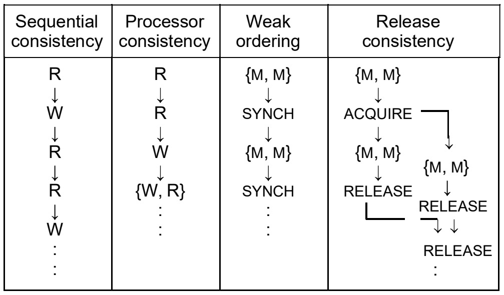
Summary: Sequential consistency is possible, but costly. The model can be relaxed in various ways.
Consistency models not using synchronization operations:

Type of consistency	Description
Sequential	All processes see all shared accesses in same order.
Causal	All processes see all causally related shared accesses in the same order.
Processor	All processes see writes from each processor in the order they were initiated. Writes from different processors may not be seen in the same order, except that writes to the same location will be seen in the same order everywhere.

Consistency models using synchronization operations:

Type of consistency	Description
Weak	Shared data can only be counted on to be consistent after a synchronization is done.
Release	Shared data are made consistent when a critical region is exited.

The following diagram contrasts various forms of consistency.



Scalable Multiprocessors

[§10.1] A *scalable* system is one in which resources can be added to the system without reaching a hard limit.

What does scalability mean?

- Avoids inherent design limits on resources.
- Bandwidth increases with # of processors *p*.
- Latency does not.
- Cost increases slowly with *p*.

Why doesn't a bus-based design scale?

- Physical constraints
- Protocol constraints
- Contention everywhere: bus, snooper, memory

Scalability and coherence

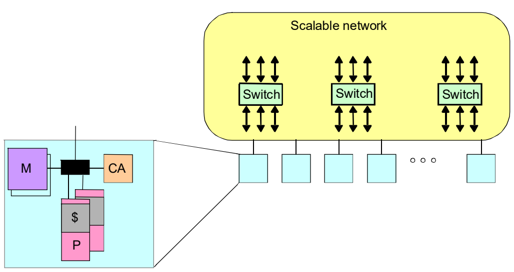
All of the cache-coherent systems we have talked about until now have had a bus.

Not only does the bus guarantee serialization of transactions; it also serves as a convenient *broadcast* mechanism to assure that each transaction is propagated to all other processors' caches.

How can cache coherence can be provided on a machine with physically distributed memory and no globally snoopable interconnect?

- To support a shared address space?
- To be able to satisfy a cache miss transparently from local or remote memory?

This means data is replicated widely. How can it be kept coherent?



Scalable distributed memory machines consist of P-C-M nodes connected by a network.

The *communication assist* interprets network transactions and forms the interface between the processor and the network.

A coherent system must do these things.

- Provide a set of states, a state-transition diagram, and actions.
- Manage the coherence protocol.
  - (0) Determine when to invoke the coherence protocol
  - (a) Find a source of information about the state of this block in other caches.
  - (b) Find out where the other copies are
  - (c) Communicate with those copies (invalidate/update)

(0) is done the same way on all systems

- The state of the line is maintained in the cache
- The protocol is invoked if an "access fault" occurs on the line.

The different approaches to scalable cache coherence are distinguished by their approach to (a), (b), and (c).

Bus-based coherence

In a bus-based coherence scheme, all of (a), (b), and (c) are done through broadcast on bus.

- The faulting processor sends out a "search."
- Other processors respond to the search probe and take necessary action.

We *could* do this in a scalable network too—broadcast to all processors, and let them respond. Why don't we?

Why not? On a scalable network, every fault leads to at least *p* network transactions.

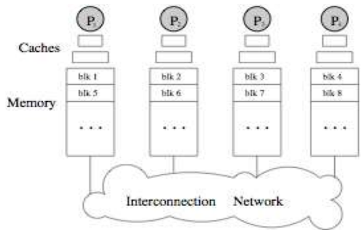
		Interconnection	
		Bus	*Point-to-point*
Protocol	*Snoopy*	Least scalable	More scalable
	Directory	–	Most scalable

Directory-based protocol

- Instead of broadcasting to find out who has the block, *keep track* of copies in the directory.
- Invalidation requests must be sent (individually) to all sharers; can you explain why this doesn't render the protocol too slow?
- Used with distributed shared memory (DSM) multiprocessors
- Can scale to tens or hundreds of processors.

How to map memory on a DSM?

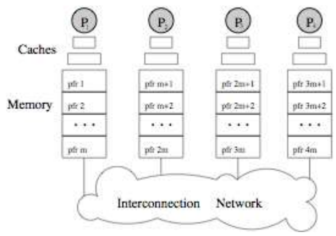
- Block interleaving?
  - distributes data around
  - hard to exploit spatial locality



- No interleaving?

  [pfr = page frame]

  Of course, the OS is responsible for placing pages in page frames.



- The OS must be involved in deciding where to allocate a page. Answer [these questions](#) …
- How are pages typically replaced on a uniprocessor?
- Why is the decision different on a multiprocessor?
- Why is "first touch" a sensible policy for many situations?
- Why is "first touch" grossly suboptimal for many parallel algorithms?

- What is an alternative allocation policy that often works well?

Handling misses in directory-based coherence

The basic idea of a directory-based approach is this.

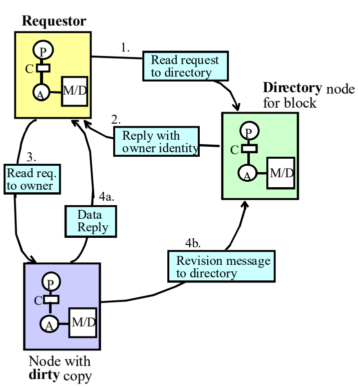
- Every memory block has associated directory information; it keeps track of copies of cached blocks and their states.
- On a miss, it finds the directory entry, looks it up, and communicates only with the nodes that have copies (if necessary).

In scalable networks, communication with the directory and with copies occurs through *network transactions*.

Let us assume that the directory is distributed, with each node holding directory information for the blocks it contains.

This node is called the *home node* for these blocks.

What happens on a read miss?

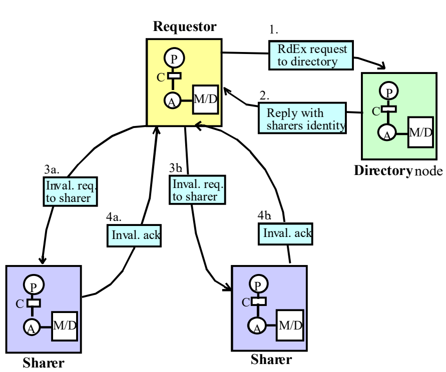


The requesting node sends a request transaction over the network to the home node.

The home node responds with the identity of the *owner*—the node that currently holds a valid copy of the block.

The requesting node then gets the data from the owner, and revises the directory entry accordingly.

On a write miss, the directory identifies copies of the block, and invalidation or update messages may be sent to the copies.



Now, see if you can tell [how many directory messages are needed](#) in each of several cases.

One major difference from bus-based schemes is that we can't assume that a write has

What information will be held in the directory?

- There will be a dirty bit telling if the block is dirty in some cache.
- Not all state information (MESI, etc.) needs to be kept in the directory, only enough to determine what actions to take.

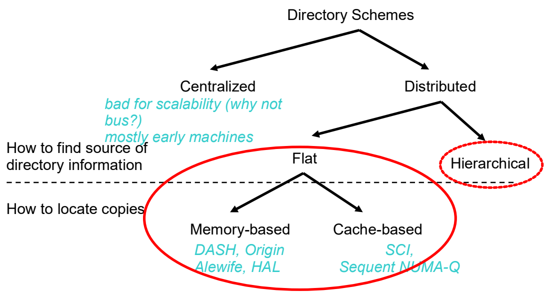
  Sometimes the state information in the directory will be out of date. Why?

  So, sometimes a directory will send a message to the cache that is no longer correct when it is received.

Flat vs. hierarchical directories

When a miss occurs, how do we find the directory information? There are two main alternatives.

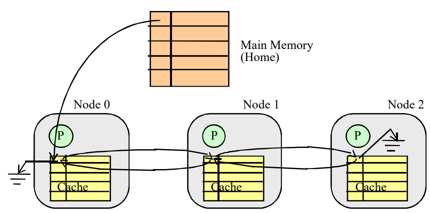
- A *flat* directory scheme. Directory information is in a fixed place, usually at the *home* (where the memory is located).
  - On a miss, a transaction is sent to the home node.
- A *hierarchical* directory scheme. Directory information is organized as a tree, with the processing nodes at the leaves.
  - Each node keeps track of which, if any, of its (immediate) children have a copy of the block.
  - When a miss occurs, the directory information is found by traversing up the hierarchy level until the block is found (in the "appropriate state").
  - The state indicates, e.g., whether copies of the block exist outside the subtree of this directory.



How do flat schemes store information about copies?

- *Memory-based schemes* store the information about all cached copies at the home node of the block. E.g., Dash, Alewife, SGI Origin.
- *Cache-based schemes* distribute information about copies among the copies themselves. E.g., IEEE SCI, Sequent NUMA-Q.
    - The home contains a pointer to one cached copy of the block.
    - Each copy contains the identity of the next node that has a copy of the block.

This means that the copies are located through network transactions.



When do hierarchical schemes outperform flat schemes?

Why might hierarchical schemes be slower than flat schemes?

Summary

Flat Schemes:

- Issue (a): finding source of directory data
    - go to home, based on address
- Issue (b): finding out where the copies are
    - memory-based: all info is in directory at home
    - cache-based: home has pointer to first element of distributed linked list
- Issue (c): communicating with those copies
    - memory-based: point-to-point messages (perhaps coarser on overflow)
        - can be multicast or overlapped
    - cache-based: part of point-to-point linked list traversal to find them
        - serialized

Hierarchical Schemes:

- all three issues through sending messages up and down tree
- no single explict list of sharers
- only direct communication is between parents and children

Distributing the directory

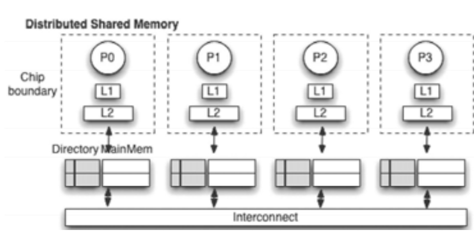
The directory needs to be distributed, but how many "pieces" should there be, and where should they be located?

Classical DSM

P-C-M nodes (p. 2, above) are connected to form a distributed shared memory system.

LL cache miss → request to directory determined by PFA of block

Directory is located at the same node as the block. Why?



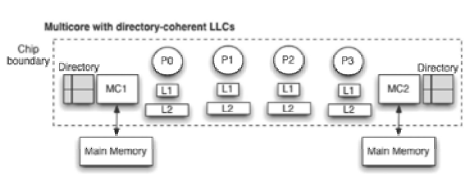
Multicore with coherent LLCs

Directory entries point to cache blocks, not main memory!

If the LLC misses, block can be fetched from another cache.

If it's not cached, then it needs to go through a memory controller (MC) to fetch it from main memory.

The number of memory controllers is limited by pin count, which may cause bottlenecks.



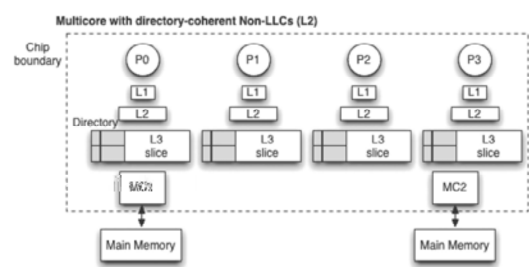
Multicore with coherent non-LLCs

In the diagram below,

- the L3 cache is "physically distributed but logically shared," and

- the L2 caches are kept coherent.

L2 miss → L3 directory searched, block retrieved from L3 or memory



In this case, the directory can be merged with the L3 tag array!

Not only does the L3 tag tell which block the L3 line holds, but also

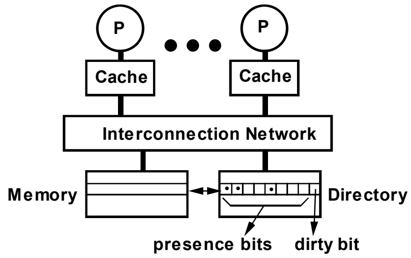
Benefit: Lower miss latency for L2 and L3.

Drawback: Directory can hold only as many entries as there are lines in the L3.

So the L3 cache has to include all blocks cached in the L2. Why?

Basic DSM Cache Coherence

[§10.3] Let us start off by considering a full bit-vector approach.



For each block of memory, assuming there are *k* processors, it maintains at the home node of the block …

- *k* presence bits *p*[..]
- 1 dirty bit *D*

Cache state is represented the same way as in bus-based designs (MSI, MESI, etc.).

- On a read by processor *i*, the home node reacts this way:
  - **If** (*D* == 0) { supply data; *p*[*i*] = 1; }
  - **else** { send intervention to owner; update home; *D* = 0; *p*[*i*] = 1; supply data to *i*;}
- On a write by processor *i*; tell how the home reacts:
- On a write by processor *i*; tell how the home reacts:
  - **if** (*D* == 0) { ________________________ ; *D* = 1; *p*[*i*]=1; supply data to node *i*; }
  - **else** { ____________________________; *p*[*owner*] = 0; *p*[*i*] = 1 ; supply data to *i*;}

On the replacement of a dirty block by node *i*, the data is written back to memory and the directory is updated to turn off the dirty bit and *p*[*i*].

On the replacement of a shared block, the directory may or may not be updated.

How does a directory help? It keeps track of which nodes have copies of a block, eliminating the need for ____________ .

Would directories be valuable if most data were shared by most of the nodes in the system?

Fortunately, the number of valid copies of data on most writes is small.

The attached animation uses the MESI protocol, with 3 block states in main memory:

- EM (exclusive or modified)
- S (shared)
- U (unowned)

✓ Network transactions for coherence
  - Read: read request
  - ReadX: read exclusive (or write) request
  - Upgr: upgrade request
  - ReplyD: home replies with data to requestor
  - Reply: home replies to requestor with IDs of sharers
  - Inv: home asks sharer to invalidate
  - WB+Inv: home asks owner to flush and invalidate
  - WB+Int: home asks owner to flush and change to S
  - Flush: owner flushes data to home + requestor
  - InvAck: sharer/owner acks an invalidation msg
  - Flush+InvAck: Flush, piggybacking an InvAck message

✓ Notation
  - Transaction (Source → Destination)
  - H = Home node

The following example is used in the animation:

Proc action	P1 state	P2 state	P3 state	Dir state @home	Network messages	# of hops
R1	E	–	–	EM, 100	Read (P1 → H), ReplyD (H → P1)	2
W1	M	–	–	EM, 100	—	0
R3	S	–	S	S, 101	Read (P3 → H), WB+Int (H → P1), Flush (P1 → H, P3)	3
W3	I	–	M	EM, 001	Upgr (P3 → H), Reply (H → P3) // Inv (H → P1), InvAck(P1 → P3)	3
R1	S	–	S	S, 101	Read (P1 → H), WB+Int (H → P3), Flush (P3 → H, P1)	3
R3	S	–	S	S, 101	—	0
R2	S	S	S	S, 111	Read (P2 → H), ReplyD (H → P2)	2

Scaling with number of processors

In order for directory schemes to be practical, they must scale gracefully.

- Scaling of memory and directory bandwidth
  - Centralized directory is bandwidth bottleneck, just like centralized memory.

  - How can we maintain directory information in a distributed way?
- Scaling of performance characteristics
  - traffic: # of network transactions each time protocol is invoked
  - latency: # of network transactions in critical path each time
- Scaling of directory storage requirements
  - Number of presence bits needed grows as the number of processors.

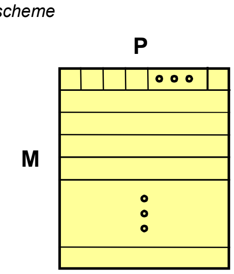
    E.g., 64-byte block size and 1024 processors. How many bits in block, vs. # of bits in directory?

Directory organization affects all of these issues.

Organizing a memory-based directory scheme

All info about copies is colocated with the block itself at the home

This works just like a centralized scheme, except that it is distributed.



Scaling of performance characteristics

- Traffic on a write is proportional to number of sharers.
- Latency? Can issue invalidations in parallel.

Scaling of storage overhead? Assume representation is a full bit-vector.

Optimizations for full bit-vector schemes

- Increase ____(1)____ size (reduces storage overhead proportionally).
- Use multicore nodes (one bit per multicore node, not per processor)

- still scales as *pm*, but only a problem for very large machines
  - 256 procs, 4 per chip, 128B line: ___(2)___% o'head

► Reducing "width": addressing the *p* term

- Observation: most blocks are cached by only few nodes
- Instead of keeping a bit per node, make entry contain a few ___(3)___.
  If *p* = 1024, 10-bit ______ ⇒ can use 100 _____ and still save space.
- Sharing patterns indicate a few pointers should suffice (five or so).
- We also need an overflow strategy for when there are more sharers than pointers.

► Reducing "height": addressing the *m* term.

- Observation: number of memory blocks >> number of cache lines.
- Thus, most blocks will not be cached at any particular time; therefore, most directory entries are useless at any given time
  - organize directory as a cache, rather than having one entry per memory block (key is ___(4)___, value is ___(5)___)

Organizing a cache-based directory scheme.

In a cache-based scheme, the home node only holds a pointer to the rest of the directory information.

The copies are linked together via a distributed list that weaves through caches.

Each cache tag has a pointer that points to the next cache with a copy.

- On a read, a processor adds itself to the head of the list (communication needed).
- On a write, it makes itself the head node on the list, then propagates a chain of invalidations down the list.

Each invalidation must be acknowledged.

- On a write-back, the node must delete itself from the list (and therefore communicate with the nodes before and after it).

Disadvantages: All operations require communicating with at least three nodes (node that is being operated on, previous node, and next node).

Write latency is proportional to number of sharers.

Synchronization is needed

Advantages: Directory overhead is small.

Work of performing invalidations can be distributed among sharers.

The IEEE Scalable Coherent Interface has formalized protocols for handling cache-based directory schemes.

The SSCI protocol

- SCI (Scalable Coherence Interface) protocol
  - IEEE standard, ratified in 1993
  - 7 state bits, 29 stable states + many pending states
- For illustration we will use Simple SCI (SSCI)
  - Retains similarity with full-bit vector protocol:
    - MESI states in the cache
    - U, S, EM states in the memory directory
    - Replaces the presence bits with a pointer
  - Similar features to SCI
    - Overall protocol operation
    - Doubly linked list
  - Many possible race conditions, which are mostly ignored in the illustration

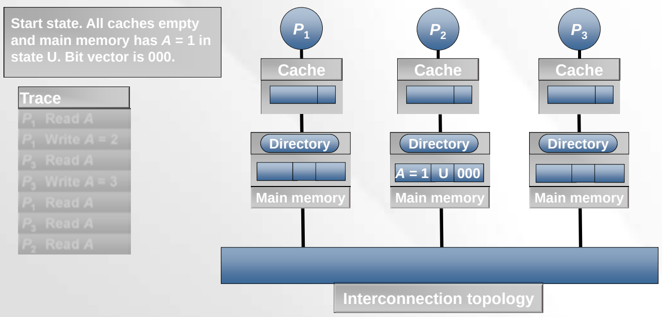
- Additional coherence network transactions (in addition to those used in full bit-vector approach):
  - WB+Int+UpdPtr
  - UpdPtr: update next/prev/head pointers

Here is the example used in the animation.

Proc action	P1 state	P2 state	P3 state	Dir state @home	Network message	# of hops
R1	E,0,0	–	–	EM, 1	Read (P1→ H), ReplyD (H→P1)	2
W1	M,0,0	–	–	EM, 1	—	0
R3	S,3,0	–	S,0,1	S, 3	Read (P3 → H), Reply (H → P3), WB+Int+UpdPtr (P3→P1), Flush (P1 → H, P3)	4
W3	I,3,0	–	M,0,0	EM, 3	Upgr (P3 → H) // Inv (P3 → P1) InvAck(P1 → P3)	2
R1	S,0,3	–	S,1,0	S, 1	Read (P1 → H), Reply (H → P1), WB+Int+UpdPtr (P1→P3), Flush (P3 → H, P1)	4
R3	S,0,3	–	S,1,0	S, 1	—	0
R2	S,2,3	S,0,1	S,1,0	S, 2	Read (P2 → H), ReplyD/ID (H → P2), UpdPtr (P2 → P1)	3

Full Bit-Vector Visualization – Start State

Start state. All caches empty and main memory has *A* = 1 in state U. Bit vector is 000.

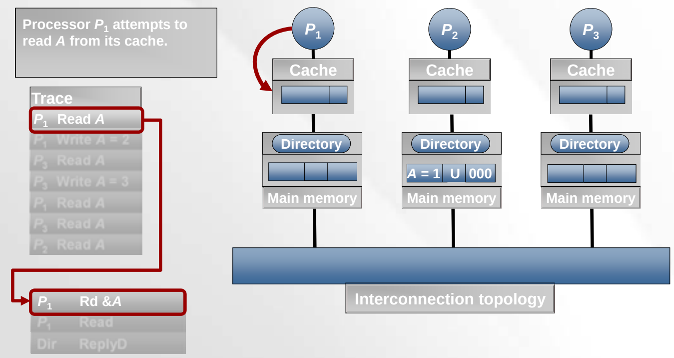


Edited by Samuel Christie and Amey Deshpande

1

Full Bit-Vector: Processor P_1 Reads *A*

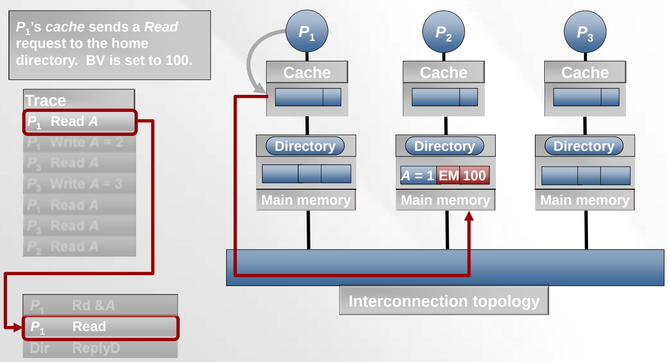
Processor P_1 attempts to read *A* from its cache.



2

Full Bit-Vector: Processor P_1 Reads *A*

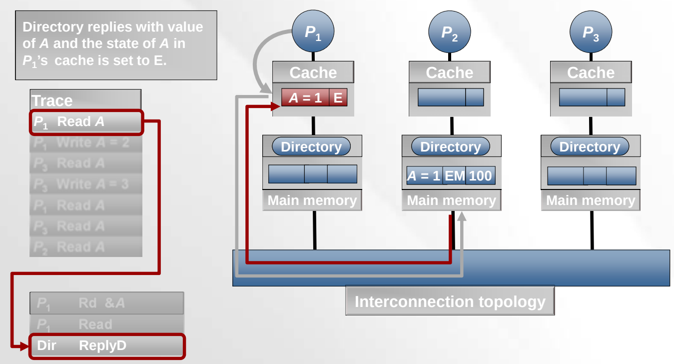
P_1's *cache* sends a *Read* request to the home directory. BV is set to 100.



3

Full Bit-Vector: Processor P_1 Reads *A*

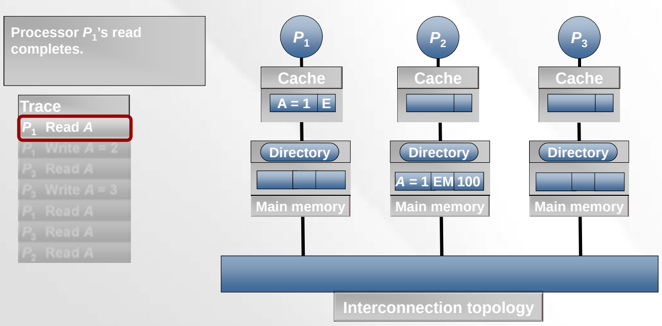
Directory replies with value of *A* and the state of *A* in P_1's cache is set to E.



4

Full Bit-Vector: Processor P_1 Reads *A*

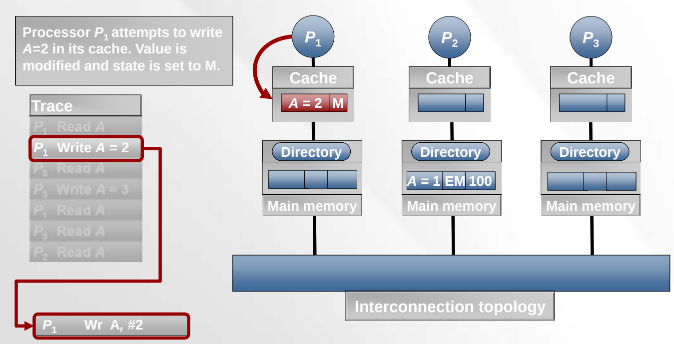
Processor P_1's read completes.



5

Full Bit-Vector: Processor P_1 Writes *A*

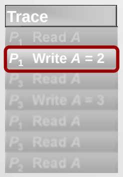
Processor P_1 attempts to write *A*=2 in its cache. Value is modified and state is set to M.

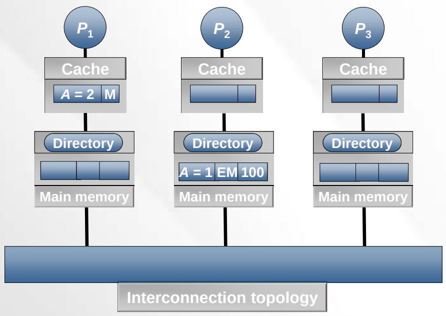


6

Full Bit-Vector: Processor P_1 Writes A

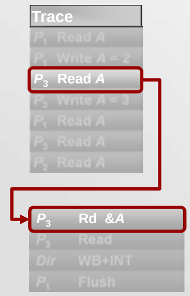
Processor P_1 completes writing A=2 to its cache.

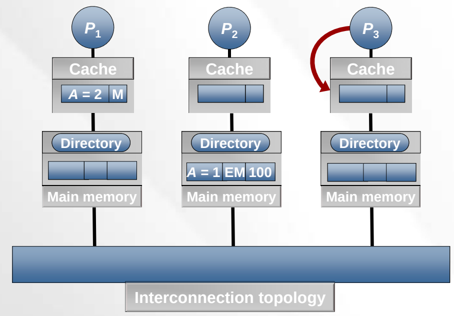




Full Bit-Vector: Processor P_3 Reads A

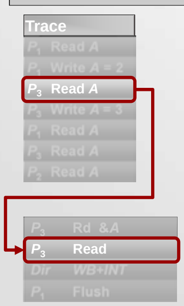
Processor P_3 attempts to read A from its cache, misses.

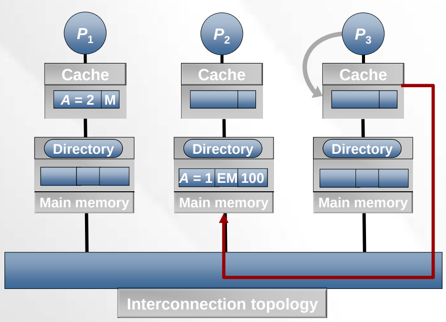




Full Bit-Vector: Processor P_3 Reads A

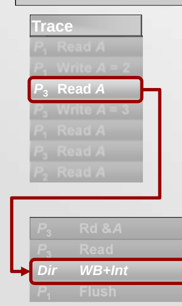
P_3's cache sends *Read* request to home directory.

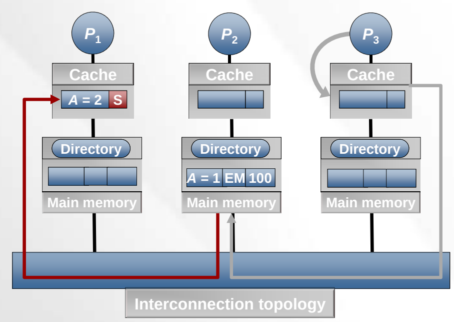




Full Bit-Vector: Processor P_3 Reads A

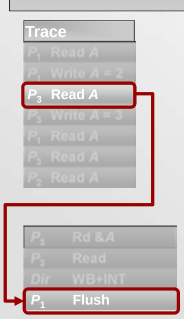
Directory sends owning cache a *WB+Int*. Owner P_1 changes state to S.

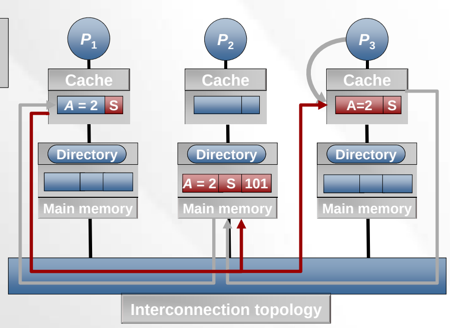




Full Bit-Vector: Processor P_3 Reads A

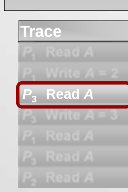
P_1 flushes the block. Directory and P_3 update the block and set state = *S*. BV is set to 101.

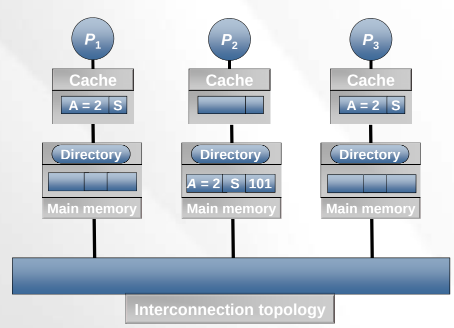




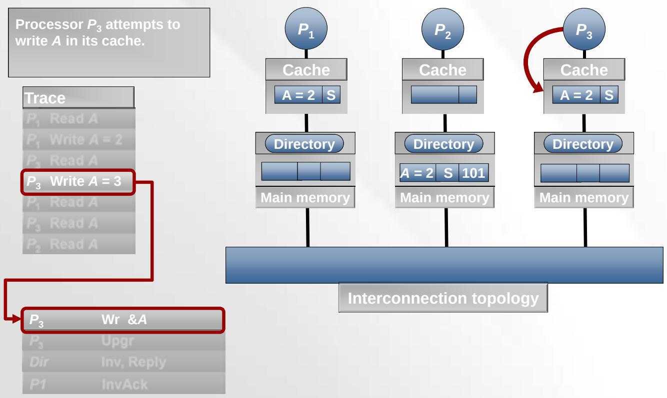
Full Bit-Vector: Processor P_3 Reads A

Processor P_3 completes read operation from its cache.

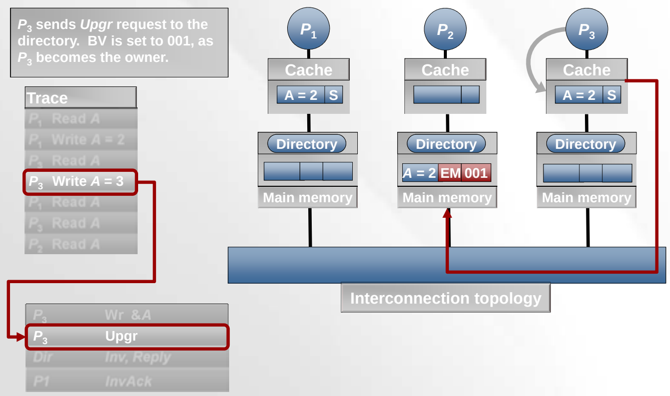




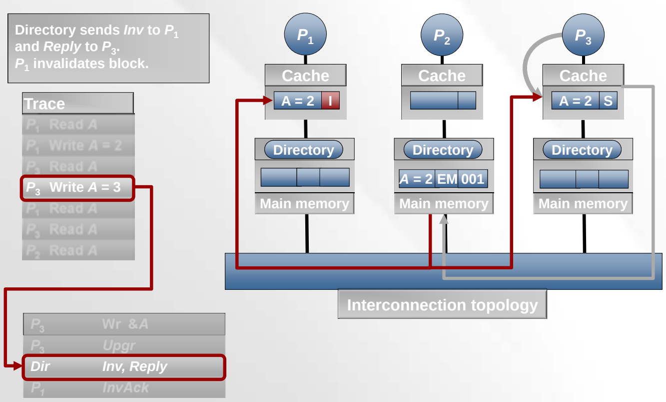
Full Bit-Vector: Processor P_3 Writes A



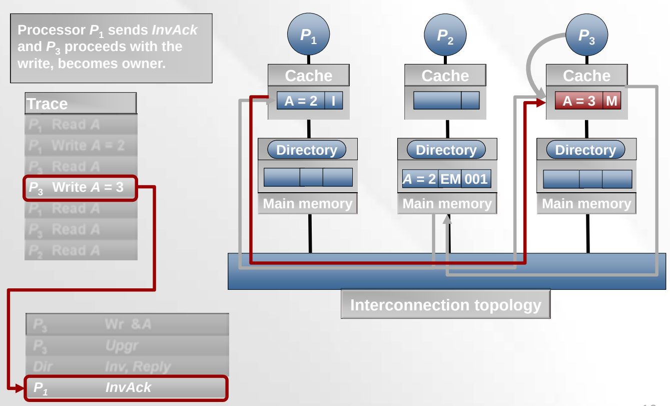
Full Bit-Vector: Processor P_3 Writes A



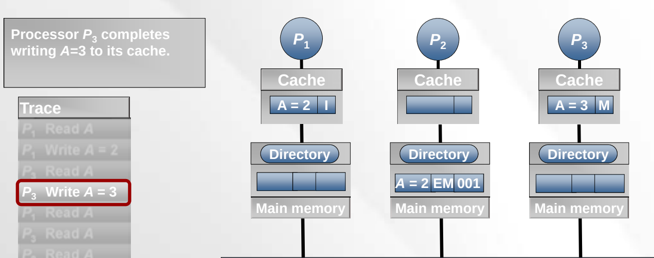
Full Bit-Vector: Processor P_3 Writes A



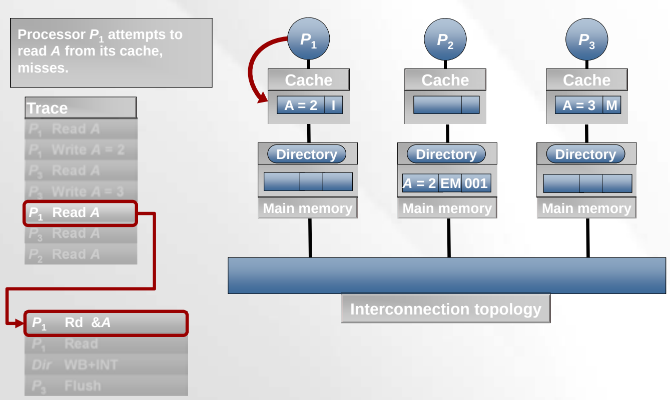
Full Bit-Vector: Processor P_3 Writes A



Full Bit-Vector: Processor P_3 Writes A

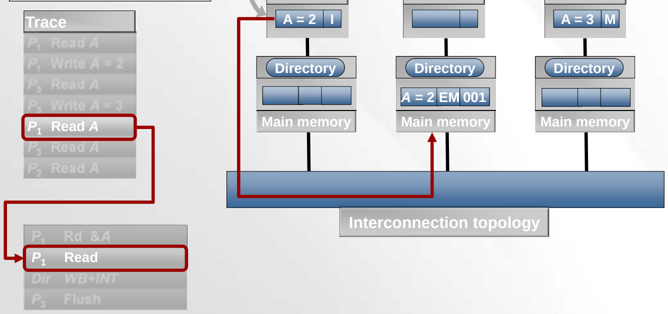


Full Bit-Vector: Processor P_1 Reads A



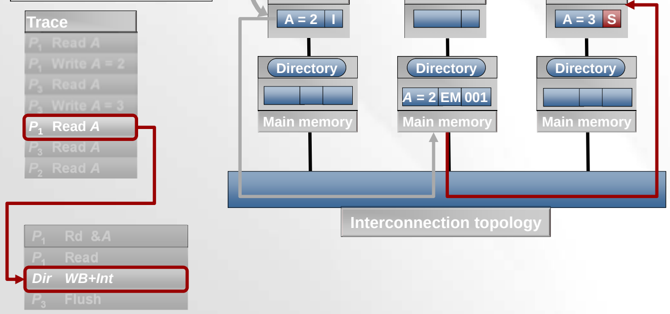
Full Bit-Vector: Processor P_1 Reads A

P_1's cache sends *Read* request to home directory.



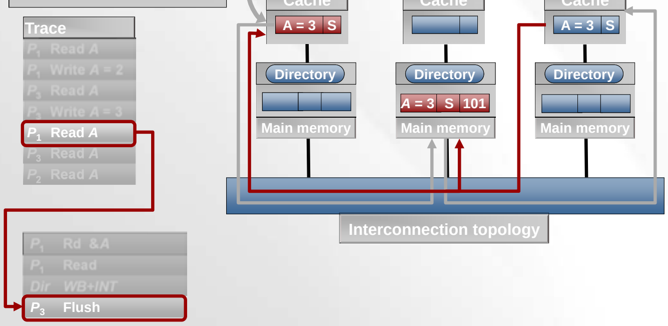
Full Bit-Vector: Processor P_1 Reads A

Directory sends *WB+Int* to the owner cache. The owner is downgraded to state S.



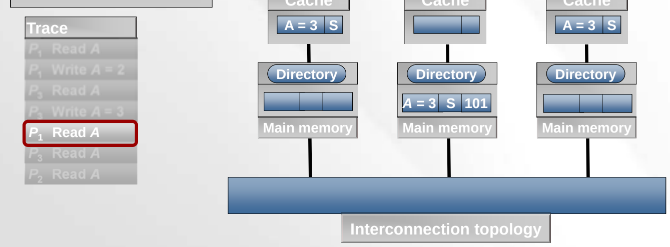
Full Bit-Vector: Processor P_1 Reads A

P_3 *flushes* the block to the directory and P_1's cache. State is set to S. BV is set to 101.



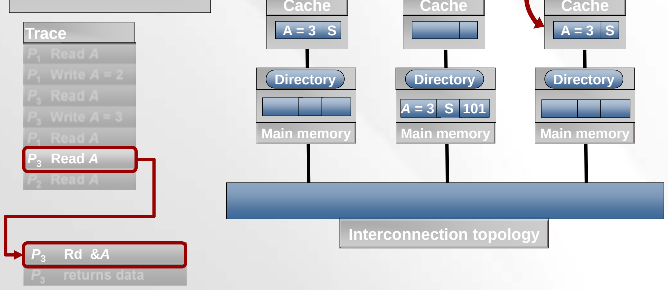
Full Bit-Vector: Processor P_1 Reads A

Processor P_1 completes reading *A* from its cache.



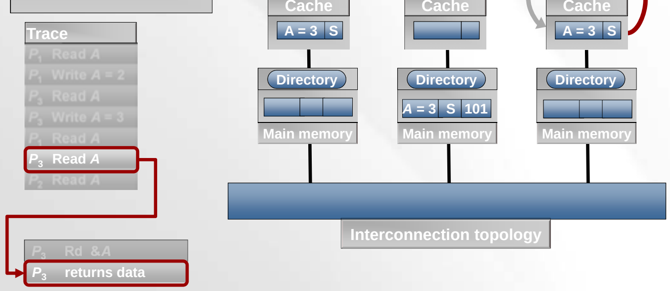
Full Bit-Vector: Processor P_3 Reads A

Processor P_3 attempts to read *A* from its cache, hits.



Full Bit-Vector: Processor P_3 Reads A

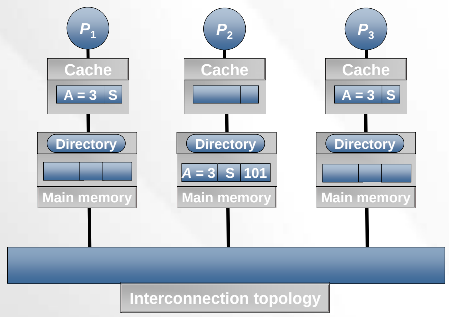
P_3's cache returns the value of *A* immediately.



Full Bit-Vector: Processor P_3 Reads A

Processor P_1 completes reading A from its cache.

Trace
- P_1 Read A
- P_1 Write A = 2
- P_3 Read A
- P_3 Write A = 3
- P_1 Read A
- **P_3 Read A**
- P_2 Read A

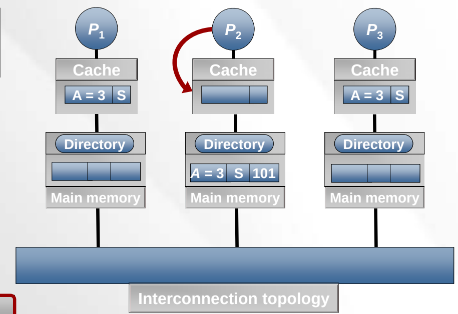


Full Bit-Vector: Processor P_2 Reads A

Processor P_2 attempts to read A from its cache.

Trace
- P_1 Read A
- P_1 Write A = 2
- P_3 Read A
- P_3 Write A = 3
- P_1 Read A
- P_3 Read A
- **P_2 Read A**

- **P_2 Rd &A**
- P_2 Read
- Dir ReplyD

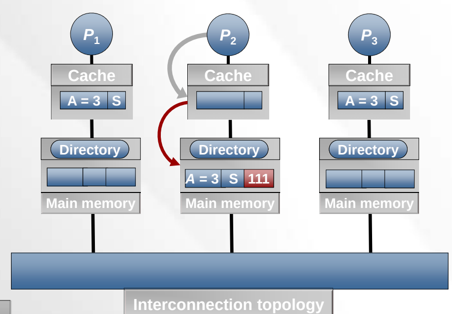


Full Bit-Vector: Processor P_2 Reads A

P_2's cache sends a *Read* request to its directory. BV is set to 111.

Trace
- P_1 Read A
- P_1 Write A = 2
- P_3 Read A
- P_3 Write A = 3
- P_1 Read A
- P_3 Read A
- **P_2 Read A**

- P_2 Rd &A
- **P_2 Read**
- Dir ReplyD

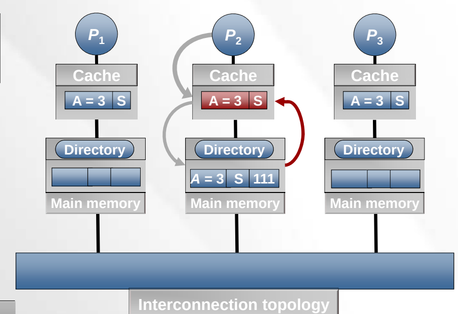


Full Bit-Vector: Processor P_2 Reads A

Directory returns the block and state with *ReplyD.*

Trace
- P_1 Read A
- P_1 Write A = 2
- P_3 Read A
- P_3 Write A = 3
- P_1 Read A
- P_3 Read A
- **P_2 Read A**

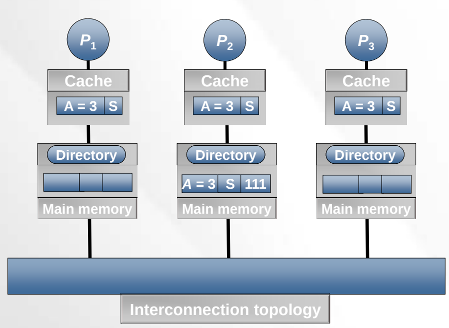
- P_2 Rd &A
- P_2 *Read*
- **Dir ReplyD**



Full Bit-Vector: Processor P_2 Reads A

Processor P_2 finishes *Read* operation.

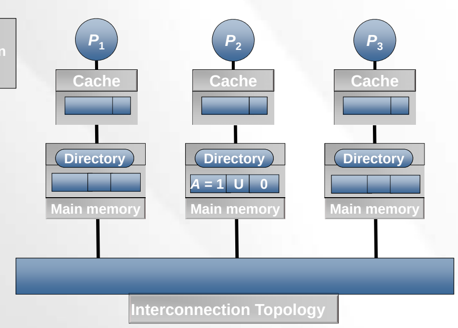
Trace
- P_1 Read A
- P_1 Write A = 2
- P_3 Read A
- P_3 Write A = 3
- P_1 Read A
- P_3 Read A
- **P_2 Read A**



SSCI Visualization – Start State

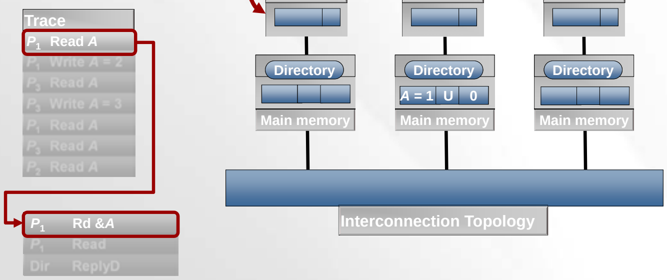
Start state. All caches empty and main memory has A = 1 in state U.

Trace
- P_1 Read A
- P_1 Write A = 2
- P_3 Read A
- P_3 Write A = 3
- P_1 Read A
- P_3 Read A
- P_2 Read A



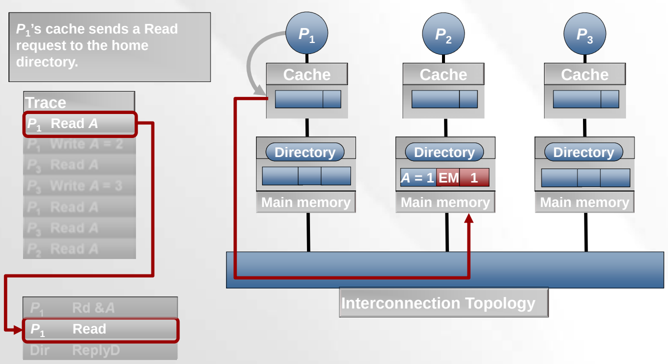
SSCI: Processor P_1 Reads A

Processor P_1 attempts to read A from its cache.



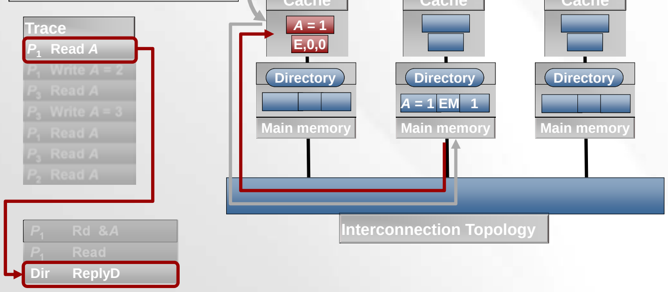
SSCI: Processor P_1 Reads A

P_1's cache sends a Read request to the home directory.



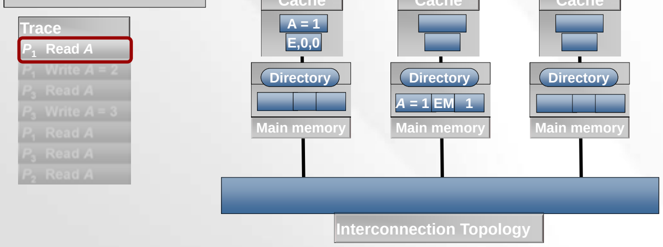
SSCI: Processor P_1 Reads A

Directory replies with value of A and the state of A in P_1's cache is set to E. The cache line contains fields *state*, *prev*, and *next*



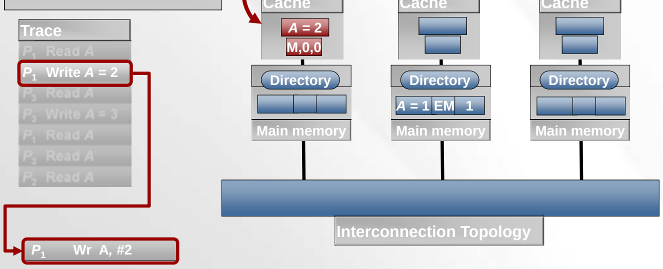
SSCI: Processor P_1 Reads A

Processor P_1's read completes.



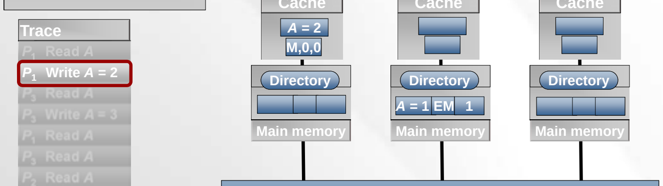
SSCI: Processor P_1 Writes A

Processor P_1 attempts to write A=2 in its cache. Block is modified and state is set to M.



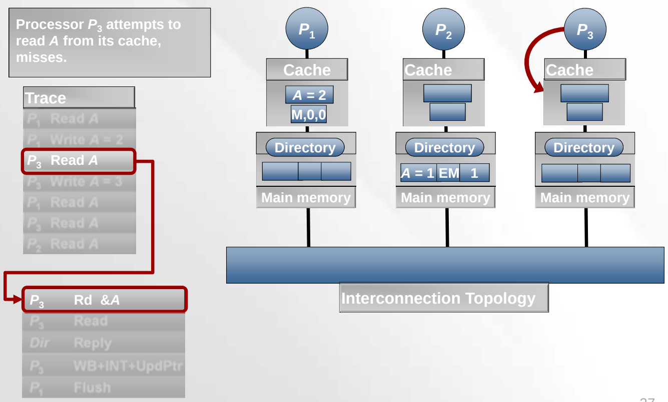
SSCI: Processor P_1 Writes A

Processor P_1 completes writing A=2 to its cache.



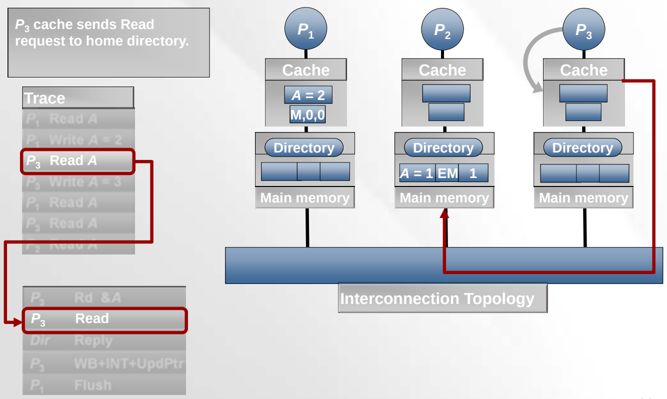
SSCI: Processor P_3 Reads A

Processor P_3 attempts to read A from its cache, misses.



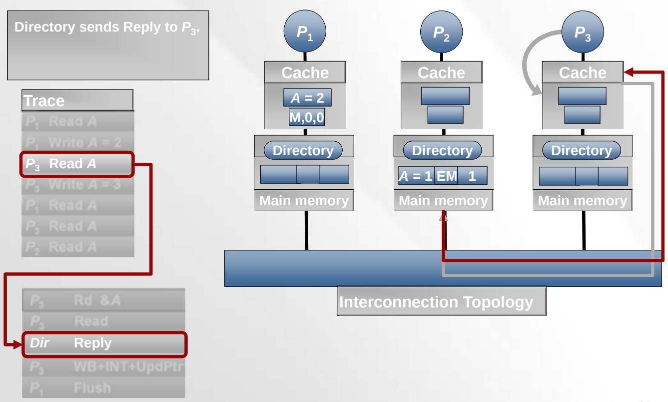
SSCI: Processor P_3 Reads A

P_3 cache sends Read request to home directory.



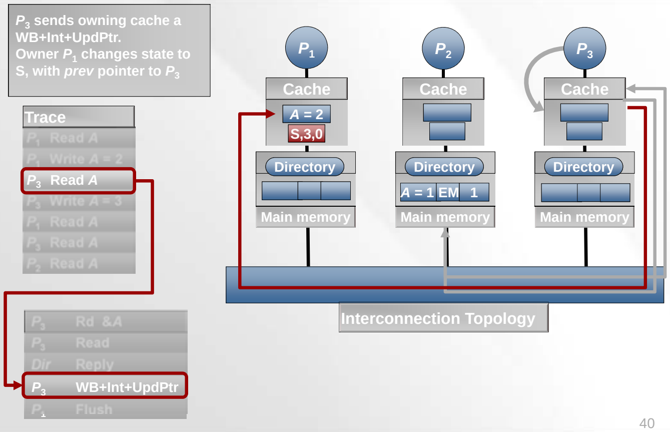
SSCI: Processor P_3 Reads A

Directory sends Reply to P_3.



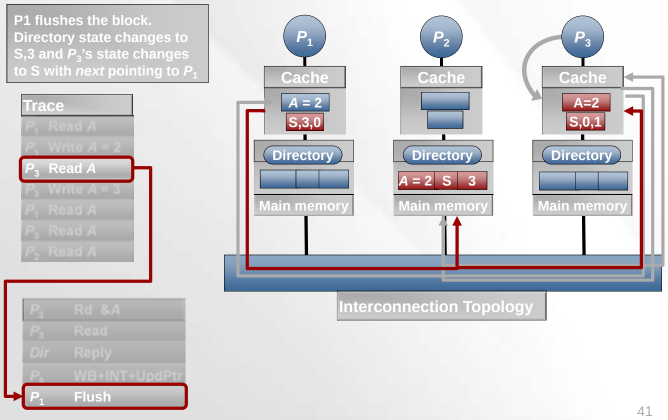
SSCI: Processor P_3 Reads A

P_3 sends owning cache a WB+Int+UpdPtr.
Owner P_1 changes state to S, with *prev* pointer to P_3



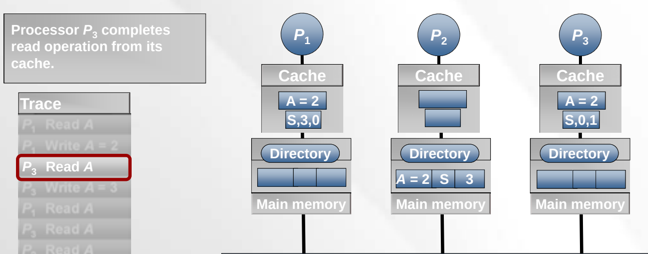
SSCI: Processor P_3 Reads A

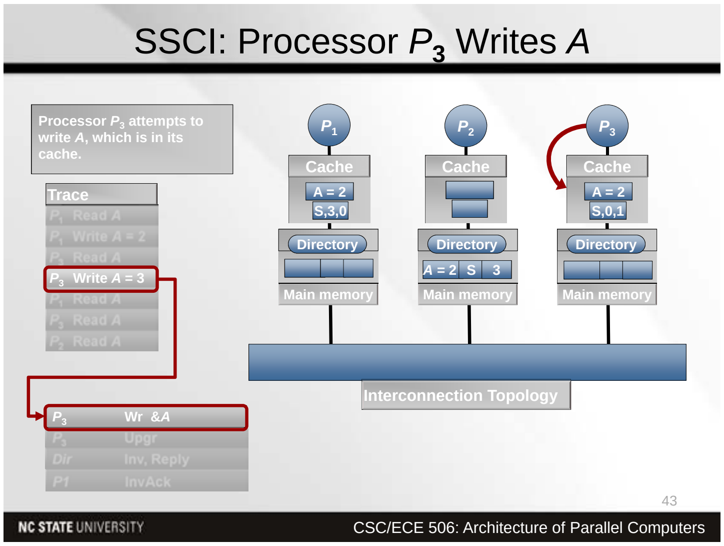
P1 flushes the block.
Directory state changes to S,3 and P_3's state changes to S with *next* pointing to P_1

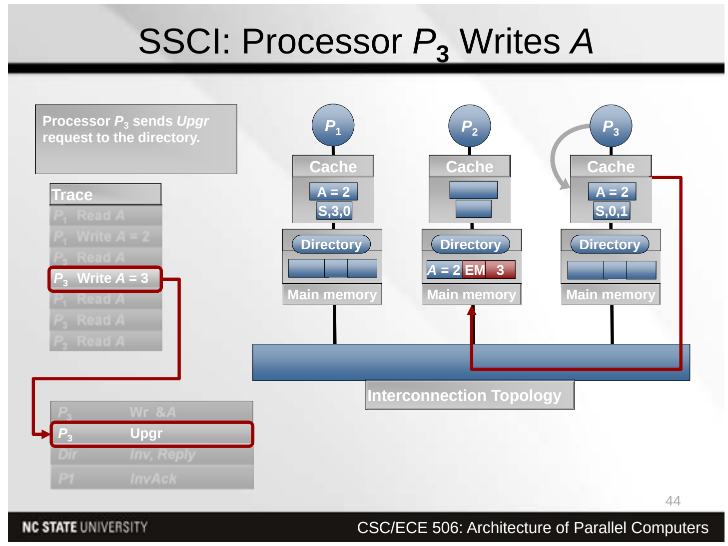


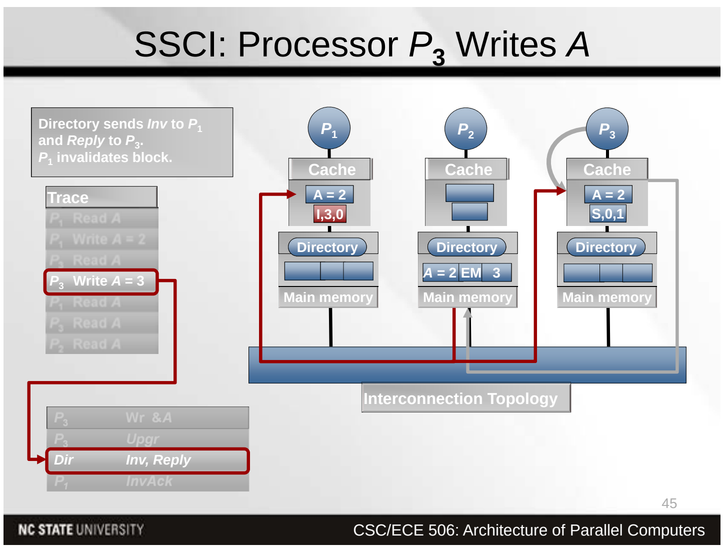
SSCI: Processor P_3 Reads A

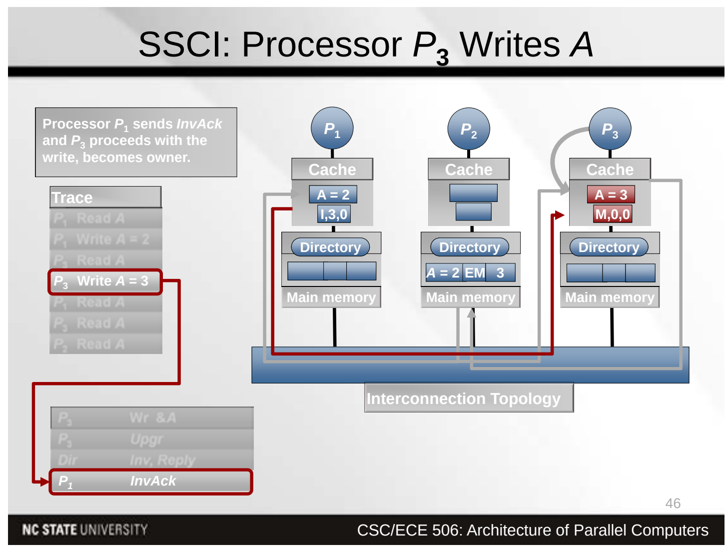
Processor P_3 completes read operation from its cache.

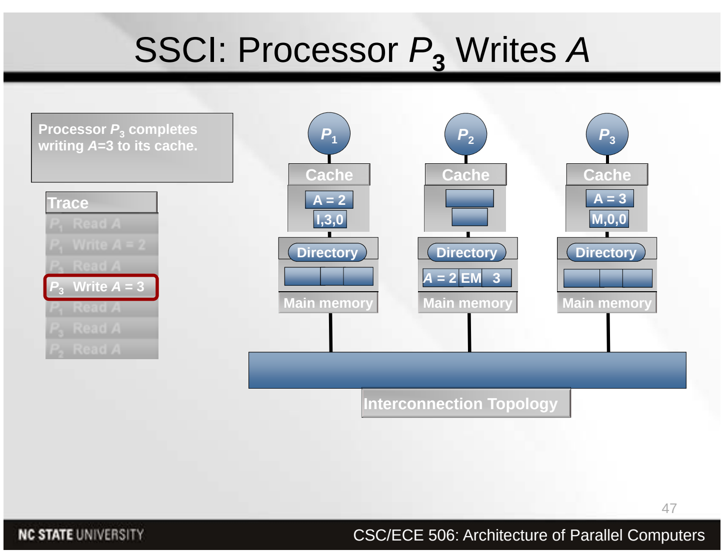


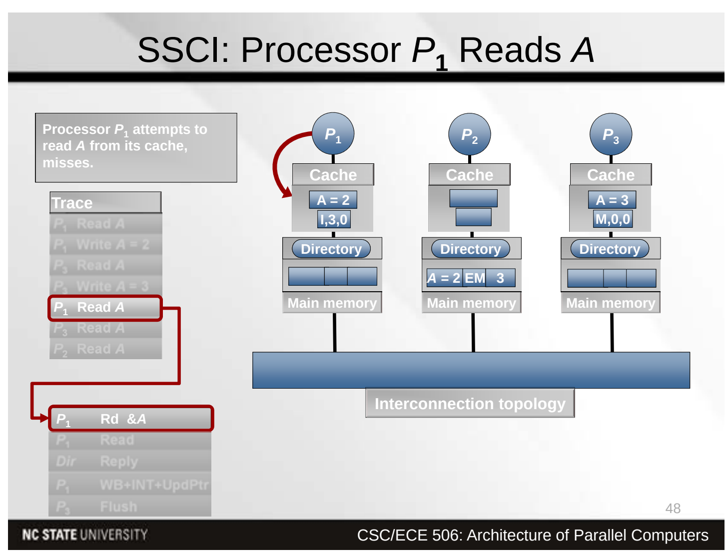

SSCI: Processor P3 Writes A
Processor P3 attempts to write A, which is in its cache.
Trace
P1 Read A
P1 Write A = 2
P3 Read A
P3 Write A = 3
P1 Read A
P3 Read A
P2 Read A
P3 Wr &A
P3 Upgr
Dir Inv, Reply
P1 InvAck
P1 P2 P3
Cache
A = 2
S,3,0
A = 2
S,0,1
Directory
A = 2 S 3
Main memory
Interconnection Topology
43
NC STATE UNIVERSITY
CSC/ECE 506: Architecture of Parallel Computers


SSCI: Processor P3 Writes A
Processor P3 sends Upgr request to the directory.
Trace
P1 Read A
P1 Write A = 2
P3 Read A
P3 Write A = 3
P1 Read A
P3 Read A
P2 Read A
P3 Wr &A
P3 Upgr
Dir Inv, Reply
P1 InvAck
P1 P2 P3
Cache
A = 2
S,3,0
A = 2
S,0,1
Directory
A = 2 EM 3
Main memory
Interconnection Topology
44
NC STATE UNIVERSITY
CSC/ECE 506: Architecture of Parallel Computers


SSCI: Processor P3 Writes A
Directory sends Inv to P1 and Reply to P3.
P1 invalidates block.
Trace
P1 Read A
P1 Write A = 2
P3 Read A
P3 Write A = 3
P1 Read A
P3 Read A
P2 Read A
P3 Wr &A
P3 Upgr
Dir Inv, Reply
P1 InvAck
P1 P2 P3
Cache
A = 2
I,3,0
A = 2
S,0,1
Directory
A = 2 EM 3
Main memory
Interconnection Topology
45
NC STATE UNIVERSITY
CSC/ECE 506: Architecture of Parallel Computers

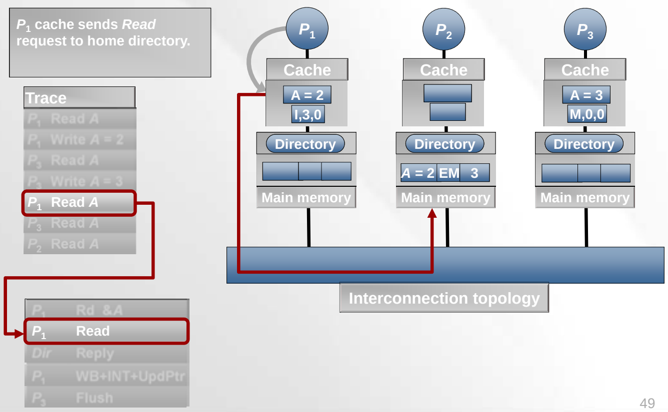

SSCI: Processor P3 Writes A
Processor P1 sends InvAck and P3 proceeds with the write, becomes owner.
Trace
P1 Read A
P1 Write A = 2
P3 Read A
P3 Write A = 3
P1 Read A
P3 Read A
P2 Read A
P3 Wr &A
P3 Upgr
Dir Inv, Reply
P1 InvAck
P1 P2 P3
Cache
A = 2
I,3,0
A = 3
M,0,0
Directory
A = 2 EM 3
Main memory
Interconnection Topology
46
NC STATE UNIVERSITY
CSC/ECE 506: Architecture of Parallel Computers


SSCI: Processor P3 Writes A
Processor P3 completes writing A=3 to its cache.
Trace
P1 Read A
P1 Write A = 2
P3 Read A
P3 Write A = 3
P1 Read A
P3 Read A
P2 Read A
P1 P2 P3
Cache
A = 2
I,3,0
A = 3
M,0,0
Directory
A = 2 EM 3
Main memory
Interconnection Topology
47
NC STATE UNIVERSITY
CSC/ECE 506: Architecture of Parallel Computers


SSCI: Processor P1 Reads A
Processor P1 attempts to read A from its cache, misses.
Trace
P1 Read A
P1 Write A = 2
P3 Read A
P3 Write A = 3
P1 Read A
P3 Read A
P2 Read A
P1 Rd &A
P3 Read
Dir Reply
P3 WB+INT+UpdPtr
P3 Flush
P1 P2 P3
Cache
A = 2
I,3,0
A = 3
M,0,0
Directory
A = 2 EM 3
Main memory
Interconnection topology
48
NC STATE UNIVERSITY
CSC/ECE 506: Architecture of Parallel Computers

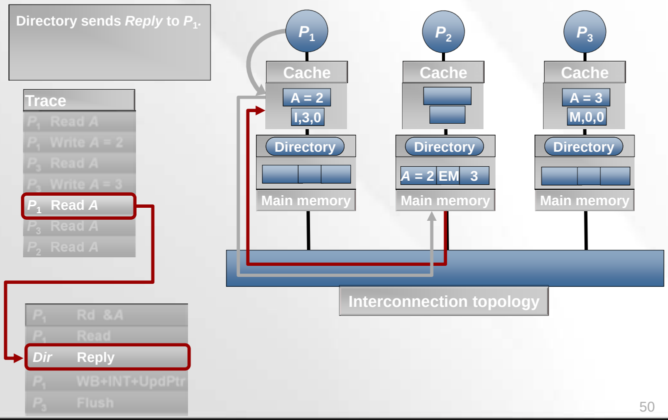
SSCI: Processor P_1 Reads A

P_1 cache sends *Read* request to home directory.



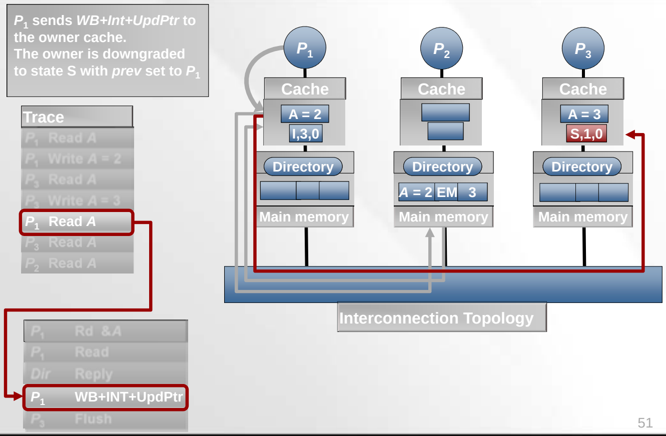
SSCI: Processor P_1 Reads A

Directory sends *Reply* to P_1.



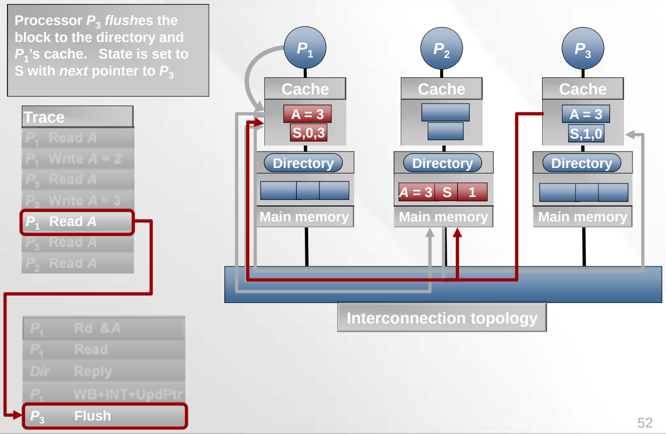
SSCI: Processor P_1 Reads A

P_1 sends *WB+Int+UpdPtr* to the owner cache.
The owner is downgraded to state S with *prev* set to P_1



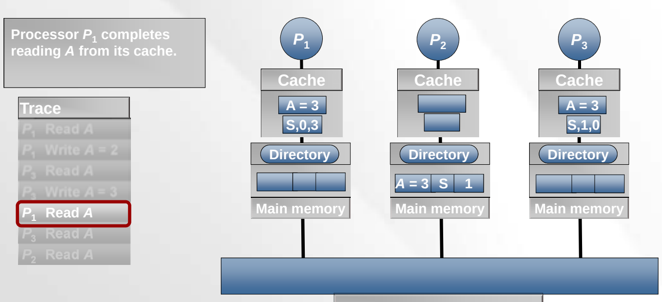
SSCI: Processor P_1 Reads A

Processor P_3 *flushes* the block to the directory and P_1's cache. State is set to S with *next* pointer to P_3



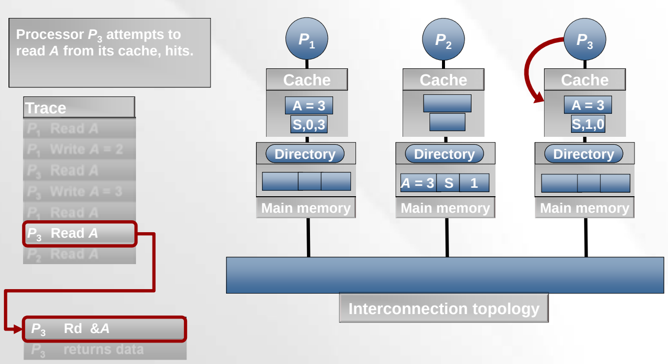
SSCI: Processor P_1 Reads A

Processor P_1 completes reading A from its cache.



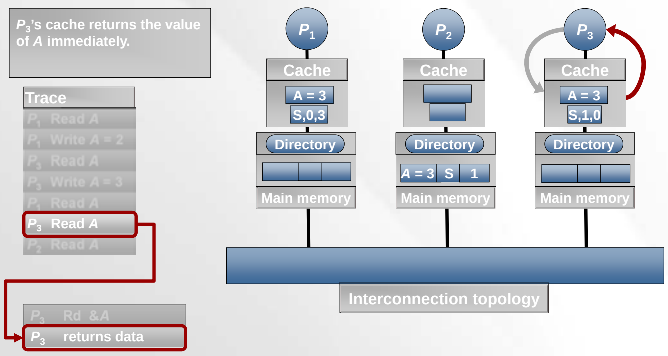
SSCI: Processor P_3 Reads A

Processor P_3 attempts to read A from its cache, hits.



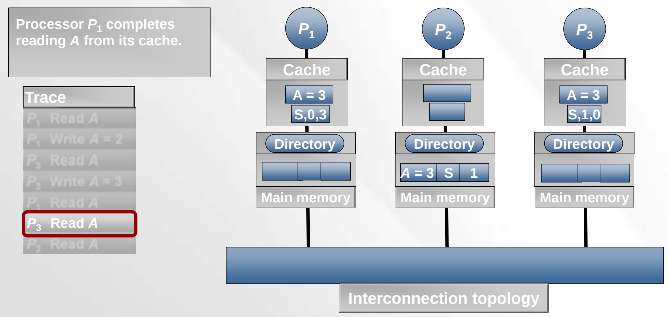
SSCI: Processor P_3 Reads A

P_3's cache returns the value of A immediately.



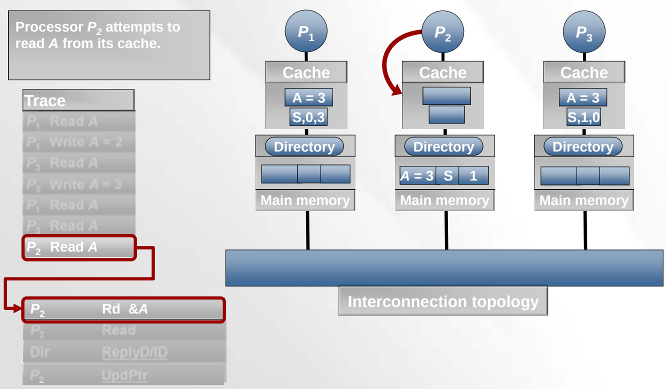
SSCI: Processor P_3 Reads A

Processor P_1 completes reading A from its cache.



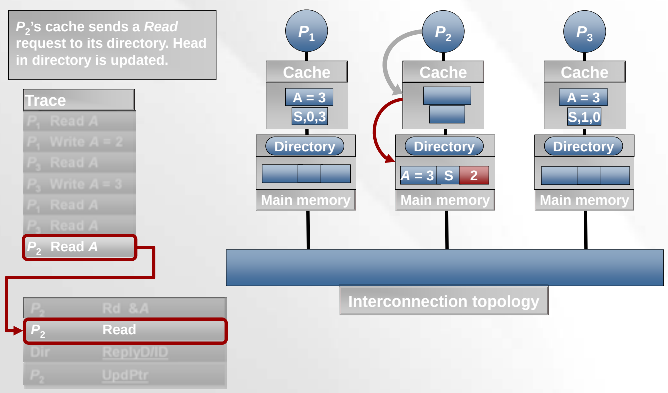
SSCI: Processor P_2 Reads A

Processor P_2 attempts to read A from its cache.



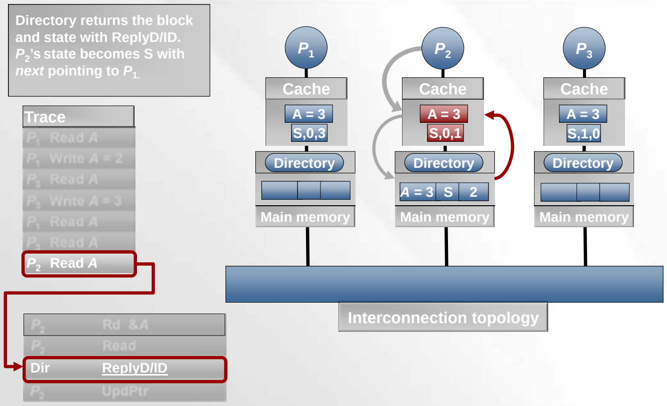
SSCI: Processor P_2 Reads A

P_2's cache sends a *Read* request to its directory. Head in directory is updated.



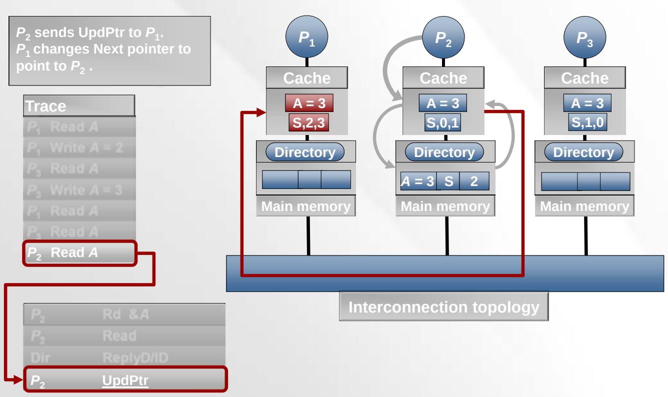
SSCI: Processor P_2 Reads A

Directory returns the block and state with ReplyD/ID. P_2's state becomes S with *next* pointing to P_1.



SSCI: Processor P_2 Reads A

P_2 sends UpdPtr to P_1. P_1 changes Next pointer to point to P_2 .

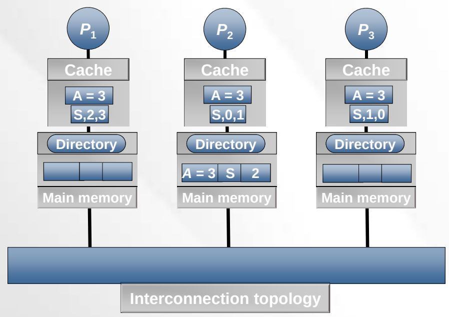


SSCI: Processor P_2 Reads *A*

Processor P_2 finishes *Read* operation.

Trace

- P_1 Read *A*
- P_1 Write *A* = 2
- P_3 Read *A*
- P_3 Write *A* = 3
- P_1 Read *A*
- P_1 Read *A*
- P_2 Read *A*



Scalable shared-memory multiprocessing and the Silicon Graphics S2MP architecture[1] (Dan Lenoski): [20a] Today I'd like to discuss scalable shared-memory multiprocessing, and the S2MP architecture, which is at the heart of SGI's latest multiprocessor.

Shared-memory multiprocessors, or SMPs, are the most popular form of multiprocessing today, because they can handle both parallel and throughput workloads.

They also offer powerful central resources, such as large memories and fast secondary storage, that are sharable by a number of processors.

To date, the drawback of these systems has been their limited scalability and high entry cost.

This talk introduces a new class of computer, the scalable shared-memory multiprocessor, which removes the drawbacks of traditional SMP systems.

Here is an outline of the talk.

I. Today's MP architectures. This introduces the scalable SMP, or SSMP.

II. Scaling the SMP model. This focuses on a particular SSMP, the Silicon Graphics Origin architecture and its S2MP memory architecture.

III. SGI's Origin.

IV. Design issues in Origin. … and then discuss some of the important design tradeoffs.

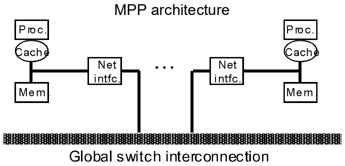
V. Conclusion.

Today's MP architectures: Let's begin by reviewing the four classes of parallel processors available on the market today.

- *Message-passing* (MPP), or massively parallel architectures.
- *Cluster* of workstations.
- *Shared memory* (SMP).
- Parallel vector (PVP).

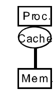
[1]Video © 1996, University Video Communications. This video is available from University Video Communications, http://www.uvc.com.

MPP architecture



Here is the structure of a message-passing, or SMP design, also referred to as a distributed memory system. It consists of a collection of CPU/memory nodes that are connected by a high-speed interconnection network.

The structure of the individual nodes is similar to a standalone computer, except that the individual nodes are usually somewhat smaller, and most are not connected directly to I/O devices. Generally, the packaging is geared to a large processor count.



Examples of this kind of machine include the Intel Paragon, Thinking Machines' CM-5, and the Cray T3D and T3E.

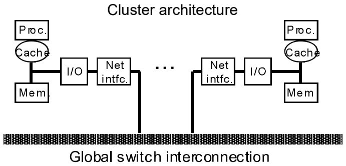
The strength of MPP systems lies in their scalability. The fact that the nodes are small, and are connected by a high-speed interconnection network allows these systems to grow to hundreds or thousands of processors.

The drawback is that programming these systems involves restructuring applications into a message-passing style, so programmers have to rewrite their application to explicitly manage all communication.

In addition, performance often suffers, since the overhead of passing a message is tens to hundreds of µsec., which is tens to thousands of instructions on a modern microprocessor.

The performance and programming overheads have limited the use of these machines to a small user base that can justify the effort of recoding their applications in return for the high aggregate computing power of a large MPP.

Cluster architecture

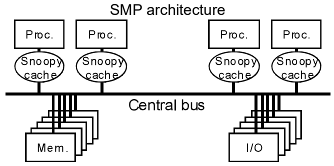


Clusters address the volume issues of MPPs by replacing the integrated MPP node with standard workstation or SMP nodes. Some cluster systems are the IBM SP series, the DEC True Cluster systems, and SGI's Power Challenge arrays. These systems are popular because they can leverage the volume of the individual nodes to hit better price/performance points.

The structure of these machines differs from MPPs in the sense that the interconnection network connects to the I/O subsystem instead of being integrated into the memory bus.

Physically these machines are generally not as tightly packaged as an MPP machine, since the nodes have I/O controllers, disks, etc. Unfortunately the fact that they are less integrated implies that the overhead of communicating between the nodes is higher than in an MPP system. Of course, they suffer from the same programming and message-passing overheads as MPP systems.

SMP architecture



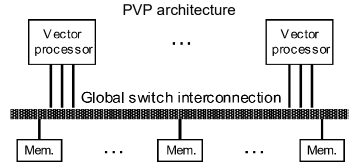
The next class of system, the shared-memory or *symmetric* multiprocessor, is quite different from the first two. Generally, SMPs combine a number of processors and a high-performance bus that provides both high bandwidth and low latency to central memory and I/O devices.

These systems employ snoopy cache coherence to keep processor utilization high and reduce bus loading. There are numerous examples of this class of system, ranging from the high-end SGI Power Challenge, Sun Ultraserver and DEC Alpha Server to the low-end two- and four-processor PC systems.

The SMPs primary advantage is the shared-memory programming model, which is a more natural extension to the uniprocessor model than the message-passing model. Shared memory also permits low-latency interprocessor communication.

Finally, the large central memory and I/O resources in an SMP are directly accessible to all processes running on the system, unlike the distributed resources of an MPP or cluster.

PVP architecture



The last class of MP system is the parallel vector processor, or PVP. PVPs differ from the other classes in that they are based on specialized vector processors instead of high-volume microprocessors.

They are also different in that the vector processors operate directly out of a high-speed memory without intermediate caches. They can achieve high throughput by hiding the latency to memory by operating on vectors instead of individual memory words.

The primary example of this kind of system is Cray's line of vector machines J90, C90, and T90.

The high-end vector machines are based on bipolar technology and utilize a very high performance interconnection to an SRAM main memory. This makes PVPs unique in that their performance can remain high on codes that cannot use caches effectively. Unfortunately, they also suffer from high cost and low volume due to their special-purpose nature.

PVPs do serve an important niche of scientific applications such as computational fluid dynamics codes that need very high performance, but cannot utilize caches well.

[20b] The ideal multiprocessor would combine the best of all of these. It would provide the scalability of MPP systems, the cost economics of cluster-based systems, the programming model and tight coupling of an SMP, and the floating-point performance and high memory bandwidth of PVPs.

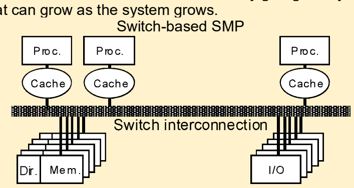
A scalable SMP restructures the SMP class to incorporate the advantages of the other architectures while retaining the programming model and low-latency communication of the SMP.

In this talk, I will focus on how the SSMP incorporates the functions of the MPP, cluster, and SMP. Integrating PVP into the SSMP is primarily a question of per-processor floating-point performance and memory-latency tolerance, together with the amount of memory spent on the memory system to achieve high bandwidth.

To understand how an SSMP is built, let's begin by looking at the structure of the SMP. Its bus structure is key to both its tight integration and cache coherence, but is also the inherent limitation on the scalability of the system.

- The cost of the bus itself limits how small a system can effectively be configured.
- The fixed bandwidth of the bus limits how far the SMP can scale to support a large number of processors.

The first step in the evolution of an SMP is to remove the bus and replace it with a switch. The switch removes the bus bottleneck by giving the system scalable bandwidth that can grow as the system grows.

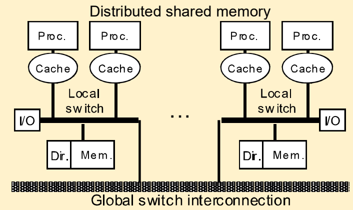


Further, the switch can be small when the system is small, and grow as the system grows.

An additional change is to the means of cache coherence, since the snoopy schemes used on bus-based SMPs rely on broadcasting every memory reference to every cache. This is done by adding directories to memory so that the memory knows which processors hold a copy of a memory block; the removes the need for broadcasts. We'll return to directory-based coherence in a moment.

The overall effect of replacing the bus with a switch is that the bus bottleneck is removed and the system is much more scalable. We also have increased modularity, in the sense that the switch structure can grow as the number of processors grows, in order to provide higher performance.

But we still have not attained the ideal structure, because the switch adds latency and uses shared bandwidth for memory locations that are accessed only by a single processor. The next step is to push portions of the memory through the switch, and distribute the shared memory and I/O with the set of processors.



With the distributed shared memory, or DSM, structure, memory and I/O that has an affinity to a set of processors can be accessed with lower latency and does not use the shared bandwidth of the global interconnection.

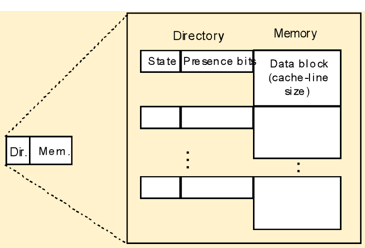
Memory bandwidth increases naturally as processors are added. Moreover, modularity is greatly increased, because each node is a complete functioning unit, and an entry system need not have a global switch at all.

DSM systems are also referred to as NUMA, or non-uniform memory access, machines. This is in contrast to traditional bus-based SMP or switch-based PVP systems, where all memory is equidistant, and there is a uniform memory access, or UMA.

NUMA systems that support caching of local and remote memory are referred to as CC-NUMA, for cache-coherent NUMA. The DSM, or CC-NUMA, system has the same basic structure, and thus scalability, as the MPP or workstation cluster. The primary difference is that the memory is accessible to all processors directly.

Furthermore, I/O can be accessed directly by each processor, and I/O devices can DMA directly into any portion of memory, as in an SMP. All that is changed from an SMP is that the memory and I/O resources have been distributed along with the processors.

Let's look at the structure of a simple directory scheme. A directory is organized as an array of state information that supplements each bank of data memory.



Each memory block, which is a cache-line sized block of memory, typically 32 to 128 bytes, has an associated directory entry.

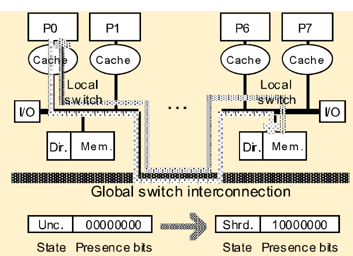
This added state information contains—

- *state bits*, that indicate whether the particular block is cached, and, if cached, whether in a shared read-only state, or an exclusive read-only state, and
- *pointer information*, which indicates which processors have this block cached. In this example, the pointer information is stored as a bit-vector with each bit representing one processor.

Let's look at how the directory maintains coherence in a system with eight processors.

Load to cached/unshared block: Assume processor 0 starts by doing a load from memory on another node.

- The processor finds that it does not have the data already in its cache, and issues a request for a shared copy of the memory block.
- This request travels to the appropriate memory, based on its address, accesses the memory location and directory, and determines that the line is either uncached, or cached only for reading by other processors.
- The memory returns a copy of the memory block, and updates the directory to indicate that the line is now shared, and that processor 0 has a shared copy.
- Other processors can also read this block, updating their sharing or presence bit in the directory as well.



Store to shared block: Now assume that processor 7 does a store to the memory location.

- This generates a read-exclusive command that is sent to the memory.
- The memory receives this command, and uses the information in the directory to determine that the line is shared, and which processors are sharing the line.
- The memory then sends invalidation requests to those processors and only those processors, and returns the line to the writing processor.
- The directory transitions to the dirty state, indicating that processor 7 has the only up-to-date copy of the memory block.

Excl.	00000001
State	Presence bits

- The invalidate messages also generate acknowledgments to the writing processor, so that it can determine when all stale copies have been eliminated.
- Now that the writing processor has exclusive ownership, it can read and write the block in its cache without further memory transactions.

Load to dirty block: Upon a subsequent read by another processor, however, the reading processor sends its request to memory, and the directory indicates that an exclusive copy is held by the writing processor, and that memory is not up to date.

This read request is then sent on to the writing processor's cache. The dirty cache returns the data to the reading processor, and sends a copy of the data to update memory and return the directory to the sharing state.

We've now returned to the original shared state before

Shrd.	10000000
State	Presence bits

processor 7's write.

Write-back and removal from cache: The other possibility is that the writing processor replaces the dirty line in its cache by issuing a write-back request to memory.

This message indicates that the dirty cache is removing its exclusive copy and updating the data memory, leaving the directory in the uncached state.

Importance of directories:

- Only processors that access a memory block are involved with coherence for that block.

  Thus, the overhead of cache coherence is never more than a fraction of the traffic required to access the given memory block if it were never cached at all.
- Memories only communicate with processors, never with one another.

  Thus, bandwidth to global cache-coherent memory can be scaled with directories by simply adding additional memory banks, or, in DSM systems, by adding additional nodes to the system.

[20c] **Scaling the SMP model.** The directory structure was originally proposed by L. Censier and P. Feautrier in 1978.

1980s: Commercial cache-coherence schemes were based on snoopy cache coherence because snoopy schemes were simpler and placed the burden of coherence on the caches themselves.

Late 1980s: Directory-based cache coherence attracted renewed interest in academia when the inherent bottlenecks of bus-based SMP systems began to be felt.

Many universities began programs to investigate scalable systems.

Another early effort at a directory-protocol implementation was taken on by the IEEE Scalable Coherence Interface Working Group. This group defined an interface standard for modules that includes a directory-based cache-coherence scheme that could be used to build up SSMP systems out of nodes conforming to the SCI standard.

1991: IEEE Scalable Coherent Interface standard.

The earliest commercial DSM systems were the Kendall Square Research KSR-1, introduced in 1991, and the Convex Exemplar SPP-1000, introduced in 1993.

These early machines were not very successful, with KSR folding, and Convex struggling financially and eventually being acquired by Hewlett-Packard.

The limited acceptance of these early DSM machines was due to improved bus technology that yielded bus-based SMP machines with more than 1 GB/s. of memory bandwidth, and to the fact that high-performance switches could only be built from expensive bipolar or gallium-arsenide technology at this time.

Today, the need for higher performance and greater scalability has driven much interest in DSM systems. Technology improvements and commodity CMOS have also made such systems much more cost effective.

Some of the announced second-generation DSM systems include—

- SGIs Origin servers
- Sequent's NUMA-cube systems, and
- Data General's NUMA-Q line.
- Convex, in conjunction with HP, has announced their second major generation of Exemplar DSM systems, the X class.

Other products are rumored to be in the works from other major computer vendors.

The performance characteristics of a DSM system can greatly affect its usability.

DSM ?= scalable SMP?

- DSM structure similar to that of distributed memory. Only difference is means of accessing interconnection network and remote memory.
- Difference is support for SMP programming model. In SMP, accesses to remote nodes are supported by hardware.
- Effectiveness depends on latency and bandwidth to remote memory.

If a system can achieve high bandwidth and low latency to all memory, then it can function as an SMP.

If latency is very high, or bandwidth is very low, then use of remote memory needs to be carefully controlled by the user. The system functions more as a distributed-memory system with a shared-memory communication system than a scalable SMP.

Let's take a closer look at the importance of remote-memory bandwidth and latency.

If one assumes that the processor is stalled on cache misses and waiting for memory 25% of the time, then one can calculate its relative efficiency when accessing remote memory, which has greater latency than local memory.

Plotting relative efficiency, based on the ratio of local references, one sees that if remote-memory access times are kept within 1 1/2 to 3 times local, then even when all references are remote, efficiency is still 2/3 of when all references are local.

If locality can be increased to 50% or better, then efficiency is 80% or better.

By contrast, if remote accesses cost 5–10 times more than local, then locality must be kept very high, or performance falls off dramatically. (It is about 30% if 100% remote references and remote references take 10 times as long as local. It is about 40% under those assumptions if 50% of references are local.)

With a large ratio, the programmer must take great care to keep locality high and manage all references to remote memory.

Thus, this kind of system cannot be programmed as an SMP, where memory placement is irrelevant and only cache reuse is important.

Likewise, looking at remote bandwidth, if there is only a fraction of local bandwidth available to remote memory, then queueing delays can increase latency and hold down efficiency when remote memory is accessed.

For example, if one assumes local memory is kept 40% occupied if all references were local, then if remote bandwidth is a fraction of local, then utilization of memory will be higher than if all references were local.

Ideally, remote bandwidth equals local, and memory utilization is unchanged by locality. But, in many systems, remote bandwidth is less than half of local, and possibly even lower than 1/8. The effect can be to drive memory utilization very high, or even into saturation. If saturation is reached, then scalability will obviously be limited. Even near-saturation conditions will raise memory latency considerably.

A DSM system can't claim to scale the SMP model unless such effects are minimized and remote-memory bandwidth is kept near local.

I/O bandwidth: As with memory, scaling the SMP model requires that remote-I/O bandwidth is kept high. Furthermore, large central-bus SMP systems provide an attachment point for very high-performance I/O devices that are often not very well supported in workstation-class systems. Examples include high-performance networking, such as HIPPI; disk connectivity, such as Fibre Channel; and high-end graphics, such as SGI's Infinite Reality.

To function as an SMP replacement, DSM systems must include such high-performance attachment points, and must have sufficient system bandwidth to support these devices. Simply having a larger number of lower-performance interconnections cannot always replace the large I/O pipes found on today's SMPs.

[20d] **SGI's Origin:** Now I'd like to turn to SGI's Origin supercomputers. Design work on the Origin began in late 1993. Systems first shipped in September 1996.

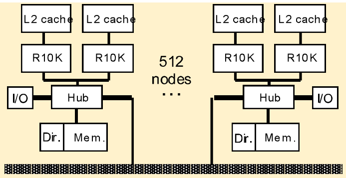
The Origin line ranges from inexpensive uniprocessor desk-sized servers to multirack supercomputers with 128 or more processors, all based on the same chip set and S2MP architecture.

The roots of the design lie in both SGI's previous SMP system, the Power Challenge, and the DASH project from Stanford. The Power Challenge set the bar of performance against which Origin was measured, while the experience on DASH provided the basis for many of the initial design directions of S2MP.

Among the design goals were—

- Follow on to Power Challenge SMP. There had to be a smooth transition that would not force customers to recode existing applications to S2MP's CC-NUMA architecture. This implied that latencies and bandwidth to remote memory had to be very aggressive.
- Scalability to many CPUs. Power Challenge could have up to 36 processors.
- Cost effectively scale up and down. Also needed to scale down more effectively than Power Challenge's 256-bit-wide bus.
- Continued I/O, graphics leadership.
- "Pay as you grow" modularity.

Here is a block diagram of the Origin system, which can scale from 1 to as many as 1024 MIPS R10000 processors.



Distributed switch interconnection

Each node within Origin is based on a highly integrated hub chip. It supports interfaces to two R10000 processors, up to 4 GB of synchronous DRAM, a pair of high-speed XIO links to the I/O subsystem, and a pair of links to the high-speed global interconnection network, or "Cray link."

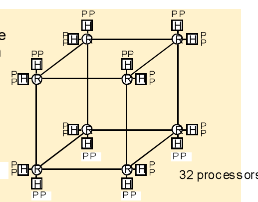
These interfaces are connected by an internal 64-bit crossbar, which can support up to 3.1 GB/s. of memory and I/O traffic.

Processor and I/O interfaces, along with the memory directory controllers within the hub of the system, communicate with via messages to implement the CC-NUMA protocol. The network interface adds the required information to route the hub internal messages across the global interconnection.

The Cray link interconnect is implemented on a pair of unidirectional 20-bit links that run at 390 MHz, supplying a peak data-transfer rate of 780 MB/s. in each direction. These links run both within a single module and between modules over cables up to 5 m. in length.

The routing network is based on a 6-ported Spider routing chip developed at SGI. Systems up to 64 processors are built by interconnecting hubs and routers in a hypercube topology.

With four-processor systems, hubs are directly connected. Beyond this, two hubs are connected to each router. The hubs are then configured into hypercubes of increasing dimension. Attaching two nodes to each router is referred to as "double bristling." It is a tradeoff between reducing cost and decreasing latency, vs. per-node system bandwidth.

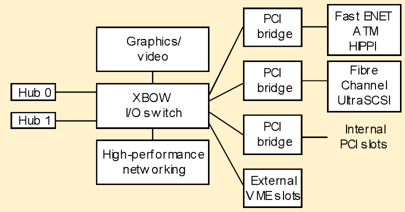


At the 64-processor level, all ports of the router are used, and going beyond this

requires another level of interconnection.

This extension is the second level of hypercubes, placed in parallel to form a "fat hypercube" topology that can grow to a 5D hypercube of 32-processor local cubes and interconnect up to 1024 processors.

XIO crossbar I/O subsystem: The XIO system uses the same high-speed physical interconnection as Cray link, but in a much more limited application, in the sense that there is a single level of crossbar, connecting up to six XIO cards to a pair of nodes. An XIO card can be native to XIO, for the maximum bandwidth of 1.6 GB/s., but more commonly, device interfaces connect to a PCI bus, which is bridged to an XIO link. Generally, the PCI bus is embedded on the XIO card, which implements a multiported unit of I/O expandibility.



The effect is that I/O is added to the system not one PCI card at a time, but an entire PCI bus at a time. For low-volume and legacy devices, there is also support for standard internal PCI cards and bridges to external VME card cages.

Modular system packaging: A single Origin module includes—

- 1 to 4 nodes (8 CPUs).
- 12 XIO slots.
- 2 XBOW switches.
- 2 router switches.
- 5 UltraSCSI disk drives.

The packaging allows the system to scale down to a cost-effective uniprocessor desk-size system, as well as to scale up with multiple 8-processor modules to a supercomputer-size system.

[20e] **Design issues:** The design was driven by our overriding goal to provide a truly scalable shared-memory design. This meant the ability to support small systems, as well as very large systems, and to grow incrementally.

Support of large systems also required us to address system reliability.

Design issues include—

- Processor and system interface.
- Node structure and size.
- Interconnection topology.
- Locality optimizations—to increase locality of reference.
- Directory structure.
- Coherence-protocol optimizations.
- System-availability features.

One requirement of any parallel system is that the processor be both high performance and highly integrated.

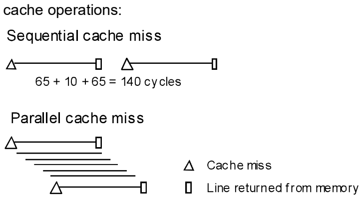
Processor design considerations:

- *High-performance processors needed.* Due to sections of limited parallelism and Amdahl's law, few parallel systems will outperform a uniprocessor if the processors in the parallel system are significantly less powerful.
- *Large shared address space.* In Origin, up to 1 TB (2^{40}) of physical memory is addressable from each processor. This requires a large virtual address space, larger than 2^{32}.
- *Multiple outstanding memory operations.* The dynamic pipeline allows a high degree of parallelism in the memory subsystem. The caches of the R10000 are non-blocking, and generate up to four outstanding reads to the memory system. Further, the hub can also process up to eight concurrent write operations, for a total of up to 12 transactions per processor.

  These multiple transactions both increase processor efficiency, by reducing the impact of memory latency, and increase throughput of algorithms that have limited cache reuse.

  In Origin, these are especially important, since these references are how processor-to-processor communication takes place.

Non-blocking cache operations:

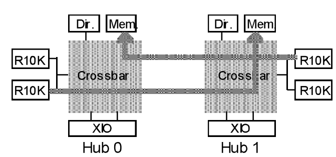


Integrated node structure: The next tradeoff was the structure and size of DSM nodes. Each node is tied together by a single hub chip that provides multiple interfaces, each capable of moving data at 780 MB/s.

The node design is primarily a tradeoff in the number of processors supported per node. The smaller nodes used in Origin provide a very tight coupling between the processors and the local and remote memory. With small nodes, local memory latency is comparable to the most integrated uniprocessor designs, since both only have a single chip between the processor and the memory itself. The small node size also allows a lower-cost entry point.

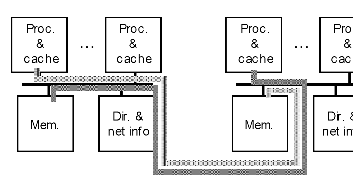
Larger nodes permit a tighter coupling of processors within a node, which can decrease communication costs between the processors within the node. Also, it can potentially reduce the overhead of the node interface to the global interconnection.

In Origin, we push for the single-chip design to reduce access time for both local and remote memory. The single chip also permits a cost-effective crossbar within the hub chip. This is important, because it permits local processors to access remote memory without interfering with remote processors accessing the local memory.



An alternative scheme, used on a number of systems, including DASH, is to add a DSM-interface card to an existing small-scale bus-based system.

The problem with this type of design is illustrated by the following graphic.



- In order to access remote memory, the bus must be traversed three times. This adds considerable latency.
- Assuming all processors are accessing remote memory, there are conflicts on the bus passing local data to remote processors while also passing remote data to local processors.

  Since remote accesses require multiple bus transactions, remote bandwidth will be reduced by a factor of 2 to 3. This can lead to a large disparity between local and remote memory-access times.

Cray link interconnection design: Goal of interconnection is to provide low latency, high bandwidth, and scalable performance and cost.

One important metric is bisection bandwidth, or the bandwidth across the center of the interconnection. Generally, for uniform data accesses, bisection bandwidth is akin to an SMP bus.

Interconnections vary from bus structures, to unidimensional ring structures, through 2D and 3D mesh structures, to hypercubes.

Early large parallel machines predominantly employed hypercube architectures; however, work done by Bill Dally and Chuck Seitz created a thrust toward lower-dimensional networks. One of the key findings from Dally's 1990 *IEEETC* paper is that for an equal number of wires, the lower-dimensional networks permitted larger, wider links. This reduces the time to receive a message, and makes up for the larger number of switches that must be traversed.

Looking in more detail at the parameters used in the study, the time to receive a message once it reaches its destination usually dominates the latency.

However, in Origin, we started with the most aggressive link and router design we could implement, and then studied what topologies would give the best performance. Given that the links are 16 bits wide, and that the latency of a router is 20 times the period of a word, the effective message size is very small, 0.8 words, falling out of the latency equation (instead of 150 words estimated by Dally).

With this new parameter, the latency curve vs. message dimension shows that latency is always reduced by increasing the dimension of the network.

Implementing a hypercube with 16-bit links for up to 512 nodes was not feasible, since it would imply support for 10 links/router, which was too many pins for Origin's technology. This is why Origin employs a hierarchical fat hypercube structure. In this structure, the bisection characteristics of the hypercube are maintained, with the only penalty being the two additional switch latencies to traverse the added hierarchy.

In larger systems, beyond 128 nodes, the simple star structure at the top level becomes a 3-, 4-, or up to 5D hypercube. This supports up to 1024 processors. The fat hypercube has latency that is proportional to the log of the number of nodes in the system, and its bisection bandwidth grows linearly with the number of nodes.

For large systems, the latency is slightly higher than a pure hypercube, but much lower than for a 2D-mesh design that could be built with the same router chip, especially above 256 processors (larger than 8 × 8 mesh).

Also note that unloaded local latencies are 320 ns. The closest remote memory is 500 ns. away, and latency grows ≈ 100 ns. every time the system size is doubled. Thus, Origin keeps the ratio of remote to local latency at 2 or 3 to 1, and is below 4 to 1 even for the largest 1024-processor system.

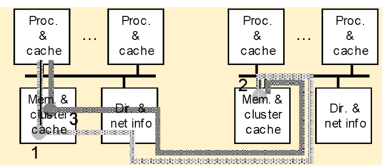
Also, the hypercube network also has bisection bandwidth that grows linearly, not by the square root or cube root, as would be the case in a 2D or 3D mesh.

[20f] While minimizing latency is important, achieving higher performance on a DSM system than an SMP system relies on having a good fraction of the references satisfied by local memory. This can be aided if the OS allocates memory to processes on the same processor that they are running on. For single-threaded jobs, this is fairly easy. For parallel jobs, it is not clear which processor will reference the given memory location the most.

Block-transfer engine instead of cluster cache: Many DSM systems implement a 3rd-level node or cluster cache to help improve locality automatically in hardware. Such a cache can reduce the number of capacity misses that must be satisfied by remote memory; however, they do not help communication misses, and are subject to conflict misses themselves. In this case, the cluster cache has a negative effect on remote bandwidth and latency.

Since the cluster cache must be large, it is made of DRAM. Using a cluster cache implies that misses result in three DRAM accesses:

1. to determine that the block is not in the local cluster cache,
2. to fetch the block from its home memory, and
3. to allocate the data into the local cluster cache.



This will obviously impact latency.

Origin does not use a cluster cache, and instead relies on page migration to improve locality.

- Page migration is assisted by hardware that keeps 64 reference counts on each 4K page of memory.
- On every access to memory, the count of the accessing node is incremented and compared with the home node.
- If the count is higher than a given programmable threshold, the hardware interrupts one of the local processors.

  This counting function does not affect the bandwidth of the data memory; it is implemented in the directory memory.

Within the hub, there's also support for a block-transfer engine, or BTE, per processor, which can copy the page at near the memory-bandwidth limit.

The BTE allows migration without polluting the cache of either the local or remote processor.

Furthermore, the read operations of the BTE actually "poison" the source page, so that subsequent accesses by other processors receive a bus error.

This error is recoverable, and is used to implement a "lazy TLB shootdown" algorithm, which reduces the overall cost of migrating memory and changing the virtual-to-physical address mappings.

Similarly to a cluster cache, the BTE scheme used in Origin can help optimize locality. It has the added advantage that it does not increase latency or decrease bandwidth to remote memory.

The only downside is that it does not react as quickly as the cluster cache to changes in locality. But this effect is reduced by the filtering of references by the processor caches.

Directory organization: The structure of the directory can become a scalability limit in systems using a simple bit-vector scheme. This is because length of the bit-vector grows by the square of the processor count. (The amount of memory grows linearly with the number of processors, and the width of the bit-vector also grows with the number of processors.)

In order to minimize this overhead, the directory entries have two formats.

- The smaller 16-bit width of directory memory supports systems up to 16 nodes, or 32 processors.

- The other, extended directory adds 32 bits to the base directory to create 48-bit-wide directory entries. In addition, the directory is implemented in two sequential memory locations, so the effective width of the directory status information, bit-vector pointers, and ECC, is either 32 or 96 bits (compared with 1152 bits for the data block plus ECC).

In either format, the directory pointers are either a binary pointer to the exact dirty processor or I/O cache, or a bit-vector specifying which nodes have the block cached in the shared state.

Directory formats for ≤ 128 processors (≤ 64 nodes):

State	Binary pointer

Memory block exclusive
3-bit state + 6-bit binary pointer — standard
+ 11-bit binary pointer — extended

State	Bit-vector

Memory block shared
3-bit state + 16-bit vector — standard
+ 64-bit vector — extended

Coarse directory format (for > 128 processors):

For systems with larger than 64 nodes, an additional coarse directory format is used. The coarse format is only needed when more than one-eighth, or octant, is caching a line.

When the line is only cached within an octant, the binary octant field, together with the 64-bit bit-vector, fully specifies which processors are caching a block.

If a memory location is cached in more than one octant, the bit-vector is interpreted as a coarse bit-vector, where each bit represents eight nodes.

State	Binary pointer

Memory block exclusive
3-bit state + 11-bit binary pointer

State	Octant	Bit-vector

Memory block shared in 1 octant (≤ 128 processors)

3-bit state + 3-bit octant + 64-bit vector

State	Coarse bit-vector

Memory block shared in > 1 octant
3-bit state + 64-bit coarse vector

Thus, with the coarse bit-vector format, we can cover the sharing case where all 1024 processors are caching a given memory block. We only need to resort to the inefficiencies of the coarse format when we have a > 128-processor system, and a memory block is shared by processors that are not in the same octant.

Overall, while the directory overhead in Origin is high, it is robust—meaning that it does not have access patterns that result in severe performance degradation—and because the directory-memory overheads, including the migration counts, are still reasonable, being less than 6% in small systems and less than 17% in large systems.

[20g] *Coherence-protocol optimizations:* The DASH coherence protocol was used, but it has been optimized in several ways, to reduce latency and maximize bandwidth for uniprocessor and parallel workloads.

The first enhancement is support for the clean exclusive (CEX) cache state, in addition to the normal invalid, shared, and dirty states.

The CEX state is used when data is returned from memory for a read request, but is currently uncached by any other processor (as would be the case for normal uniprocessor data). The data is returned exclusively to the processor, which can store directly to that location, without having to reference memory again to obtain exclusive ownership.

In contrast, without CEX support, a processor would first obtain a shared copy for the read request, and then have to re-access memory to obtain exclusive ownership.

The second enhancement is support within the coherence protocol for processors dropping CEX or shared data from their cache without updating the directory. This enhancement maximizes memory bandwidth, especially in the uniprocessor case, because no memory transactions are required simply to update the directory. All accesses are simple reads and write-backs used to obtain data.

If directory updates are required, then every cache replacement requires two directory accesses, and memory bandwidth could be reduced by up to a factor of two.

There are other enhancements to enhance multiprocessor communication, similar to those used in DASH.

In particular, there is support for request forwarding. For reads of data held dirty in another processor's cache, this implies that the memory forwards the read request to the dirty cache.

This cache responds by sending the dirty data to the requesting processor in parallel with sending the sharing write-back to memory.

Likewise, upon a read-exclusive request to satisfy a store by the processor, there is a requirement to eliminate the other cached copies. Forwarding in this case implies that memory sends invalidations to the sharing processors, and they return invalidate-acknowledgments directly to the running processor.

In both the read and read-exclusive case, forwarding reduces serialization by one system message, reducing latency by 25%.

Availability features: Another important aspect of the design of a large system is support for high reliability.

- Modularity and redundancy are basis for high availability.
- All SRAM and SDRAM covered by SEC/DED ECC.
- Highly integrated VLSI with controlled operating temperature.
- All high-speed links covered by CRC and include link-level error detection and HW retransmission.
- Cray link interconnection for multiple paths between modules and hot plug capability.

Control of sharing.

- Large-scale machines require protection from OS panics.
- Internal registers and I/O devices protected by 64-bit access-control registers.
- Each 4KB page protected by similar vectors.

The thrust of the work on IRIX, SGI's Unix clone, is to improve its scalability and make its virtual-memory manager and scheduler NUMA-aware.

Benchmark results: From Stream benchmarks, which carry out stride-one vector operations over memory, and the benchmark allows each processor to access its local memory. Origin outperforms the competition.

On this benchmark, though, even cluster-based systems scale, since the benchmark allows processors to reference their local memory. On benchmarks that require running out of remote memory, Origin shows only a 12% degradation when memory placement is uncontrolled.

Origin functions as a truly scalable SMP.

Conclusion: This work has shown that the SMP programming model can be made to scale to large processor counts with high performance. The two key techniques are directory-based cache coherence and scalable interconnection networks. These allow SMP model to stretch to design space previously only covered by parallel vector processors. It can also scale down to more common smaller configurations.

Protocol Races

[§10.4] We have assumed—

- Directory state reflects the most up-to-date state of caches.
- Messages due to a request are processed atomically.

In reality, one of or both conditions may be violated

- *Protocol races* can occur
- Some protocol races can be handled in a simple way; others are trickier.

We will discuss how protocol races can be handled.

- Purpose of discussion: illustrate approaches for dealing with protocol races.
- Discussing *all* possible races is not the goal.

Handling races: out-of-sync directory

[§10.4.1] Suppose the home sends an invalidation to a node that has replaced the block silently.

- The node can reply with

Suppose that the home receives a read request from a node that is already a sharer from the home point of view.

- The directory can reply with data

Suppose that the home receives a read/write request from a node that the home thinks is the owner.

- (In the directory, what state is this block in?)
- What might have happened to the block?
  - If the block was clean,
  - If the block was dirty,
- What should the home do? (Why will neither of these work?)
  - Wait?

- Reply with data?

- The directory alone cannot resolve this. Coherence controllers at other nodes must participate in the solution.

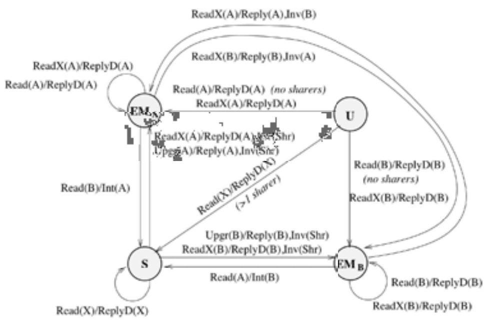
- What does the coherence controller at a node *n* need to do when a flush or writeback occurs?
  - Maintain an *outstanding transaction buffer* (OTB) for flush messages.
  - Require the home to acknowledge the receipt of a flush

- These two steps allow node *n* to delay a Read/ReadX request to a block that is still being written back.

- Hence, the home only receives Read/ReadX to a block that is not being written back.
  - When it does, it can send a

Protocol modification

Here is a modified state-transition diagram.



What is the meaning of "owner" in a directory protocol?

The meaning of "owner" is ambiguous here …

- because the directory may be out of sync with cache states,
- the directory may get a Read or ReadX from a node it *thinks* is the owner (but actually isn't).

(This isn't permitted by the protocol.)

What do we do about it?

- Split EM into two states (EM_A and EM_B) to reflect this situation.
- EM_A means the directory thinks the current owner is A.
- EM_B means the directory thinks the current owner is B.

Transitions from state U

Suppose the block is in state U in the directory.

- What happens on a ReadX request?
  - The system fetches the block from the local memory, sends a ReplyD to the Requester, and moves to state

- What happens on a Read request?
  - The system
  - What state does the requesting cache transition to?
  - What state does the directory transition to?

Transitions from state S

Suppose the directory state is S.

- What happens on a Read request?
  - The directory knows it has a valid block in the local memory.
  - It sends a ___________ to the Requester and updates the sharing vector.
  - Directory state

- What happens on a ReadX request?
  - Directory sends _______________ to the Requester.
  - Directory sends ___________ to all (other) sharers.
  - State changes to
  - But, if it's an upgrade, it just

Transitions from state EM

Suppose WLOG the directory state is EM_A.

Suppose a Read request (from a different node B) is received.

- The state is set to
- An ________________ is sent to the owner (A) to change its state to ___

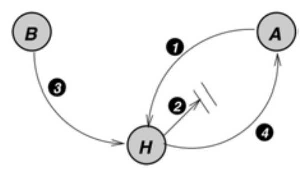
Suppose a ReadX request (from a different node B) is received.

- Directory sends an invalidation message to
- This message also says to send the data to
- Directory sends a reply message to B, saying that __ will supply the data.
- State transitions to
- (Note that it doesn't matter whether owner is in state E or M.)

Suppose the directory has an out-of-sync view of cache states, and is in state EM_A.

- Suppose it receives a Read or ReadX from A.
  - This means A's block must've been replaced due to a cache miss.
- The directory knows that A is really the owner.
- Thus, it can just respond with

Handling races: non-atomic messages



1. [§10.4.2] A sends a read request to home.
2. Home replies with data (but the message gets delayed).
3. B sends a write request to home.
4. Home sends invalidation to A, and it arrives before the ReplyD

Why is this a problem?

This is called an "early invalidation" race.

How should A respond to the invalidation?

Two incorrect ways to respond:

- A replies with InvAck.
  - B thinks that its write propagation is complete
  - A receives a ReplyD and places the block in its cache (the block that should have been invalidated).
- A ignores the invalidation message
  - The message is lost; write propagation has failed to occur

Solution:

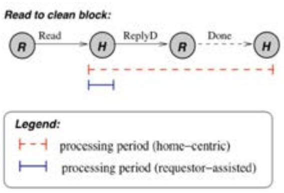
- Brute force (avoids overlapped handling of requests):
  - Home waits until it receives ack from all parties (home-centric)
- Allow overlapping but ask nodes to participate (requester-assisted)
  - Node keeps an OTB

- It does not entertain requests (to the same block) until the current transaction is completed

Exercise: Explain how each of these scenarios would play out using the four-step diagram above.

Processing a Read Request

Case 1: Read to clean block



Home-centric approach

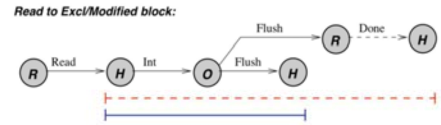
- Directory enters a transient state.
- Home replies with data
- Requester receives data, sends ack to home.
- Home closes transaction (transitions to a stable state, update sharing vector).

Cons: too much serialization at home, transaction closed late, and it requires ack

Requester-assisted approach

- Directory sends ReplyD, then closes transaction
- Requester buffers/nacks all new requests until ReplyD received (i.e., till the current Read transaction is completed)

Case 2: Read to block in EM state



Home-centric approach

- Requester sends Read to home
- Home enters a transient state, sends intervention to owner
- Owner flushes block to home and requester
- Requester sends ack back to home
- Home closes transaction (transitions to shared state, updates sharing vector)

Requester-assisted approach

- Requester sends Read to home
- Home enters a transient state, sends intervention to owner
- Home cannot close the transaction yet, because in the final state (Shared), it must have a clean copy of the block
- Owner flushes block to home and requester
- Upon receiving the block from owner, home closes transaction

Processing a write (ReadX) request

We will cover this in the next class.

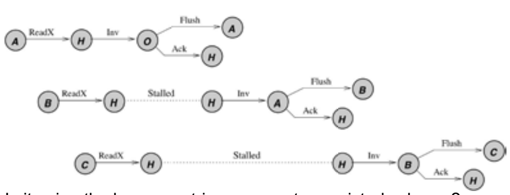
Write Propagation and Serialization

[§10.4.3] In a directory-based protocol,

- Write propagation is achieved through invalidation.
- Multiple writes to a block are serialized by the protocol.
  - Transaction closes after the ack from current owner is received by home.

  - A new ReadX request is not served until the previous ReadX request is closed.
  - This provides write serialization

Here is a diagram of serializing writes by A, B, and C.



Is it using the home-centric or requester-assisted scheme?

Memory consistency models

[§10.4.5] Implementing sequential consistency:

All memory accesses by a processor must be issued and completed in program order.

Which of the two (issuing or completion) is hardest to assure?

- Write completion detected when all InvAcks are collected
- When does read completion occur?

- Prefetching and load speculation can be used.

As the number of processors grows,

- Average latency of a cache miss increases
- Harder to hide it
- What does this do to the viability of SC?

Interconnection networks

When more than one processor needs to access a memory structure, interconnection networks are needed to route data—

- from processors to memories (concurrent access to a shared memory structure), or
- from one PE (processor + memory) to another (to provide a message-passing facility).

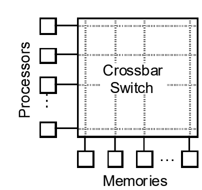
Inevitably, a large bandwidth is required to match the combined bandwidth of the processing elements.

» One extreme is a *shared bus*. How does the cost scale as the number of processors N increases?

How does the bandwidth scale?

» For concurrent access to shared memory, the ideal structure is a ________________, which can simultaneously connect any set of processors to any set of distinct memory modules.

All N processors can access all M memory units with an $N \times M$ crossbar switch.



Since there are usually about as many processors as memories, as processors are added, the complexity of a crossbar switch grows as N^2.

How does the bandwidth scale?

For reasonably large values of N, the crossbar switch may be more expensive than the processors and memories.

» For message passing, the most general is the *complete interconnection network*—a path from each processor to every other processor.

Unfortunately, this requires ________________ bidirectional links. Cost grows with the square of N.

Measures of interconnection performance

Several metrics are commonly used to describe the performance of interconnection networks:

- *Degree*, the number of links ("edges") that are attached to a node.
- *Diameter*, the maximum number of nodes through which a message must pass on its way from source to destination.

  Diameter measures the maximum delay in transmitting a message from one processor to another.
- *Average distance*, where the distance between two nodes is defined by the number of hops in the shortest path between those nodes. Average distance is given by

$$d_{avg} = \frac{\sum_{d=1}^{r} (d \cdot N_d)}{N-1}$$

  where N is the number of nodes, N_d is the number of nodes at distance d apart, and r is the diameter.
- *Bisection width*, the smallest number of wires you have to cut to disconnect the network into two equal halves (±1).

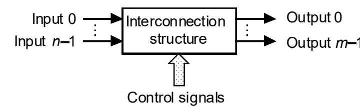
For a crossbar, give all of these metrics: Degree, diameter, average distance, bisection width.

Which of these metrics are measures of performance, and which are measures of cost?

Interconnection topologies

[§10.4] An idealized interconnection structure—

- takes a set of n input ports labeled 0, … , n–1 and
- sets up connections between them and a set of m output ports 0, … , m–1,
- with the connections determined by control signals.



Usually we will assume that $m = n$.

Here are some sample topologies.

1. Ring.

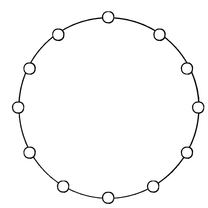
Processor i directly connected to processors i+1(mod N) and i–1 (mod N). Data can be moved from any processor to any other by a sequence of cyclic shifts.

Motivation: Many parallel algorithms include calculations of the form

$$X[i] := \frac{X[i-1] + X[i]}{2}$$

Usually every item of an array except the first and last is updated in this way.

The processor interconnections can be diagrammed as a bidirectional ring:



The diameter of a bidirectional ring is ____. Its bisection width is

What about average distance and degree?

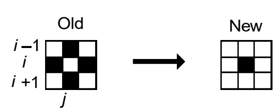
2. Mesh interconnection network

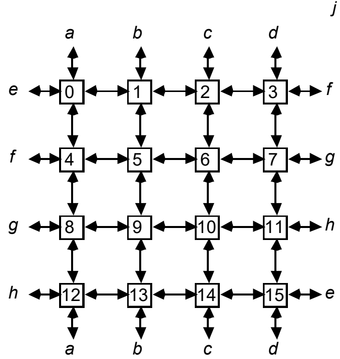
A mesh is like having "row & column" cyclic shifts.

One motivation: *Four-point iteration* is common in the solution of partial differential equations. Calculations of the form

$$X[i, j] := (X[i+1, j] + X[i-1, j] + X[i, j-1] + X[i, j+1]) \div 4)$$

are performed frequently.





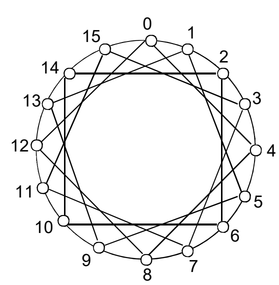
Here is an example of a 16-node mesh. Note that the last element in one row is connected to the first element in the next.

If the last element in each row were connected to the first element in the same row, we would have a *torus* instead.

In the Illiac IV, each processor i was connected to processors:

{i+1, i–1, i+8, and i–8} (mod 64).

The diameter of an Illiac IV mesh is $\sqrt{N} - 1$. For example, in a 16-node mesh structure, it takes a maximum of 3 steps. To see that, let us look at the mesh interconnection network shown in the form of a *chordal ring:*



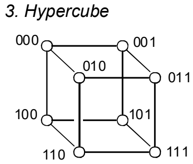
In a 64-element mesh, any node can be reached from any other in no more than 7 of these shifts.

Without the end-around connections (a "pure" 2D mesh), the diameter is $2(\sqrt{N} - 1)$.

It is also possible to have a multidimensional mesh. The diameter of a d-dimensional mesh is $d(N^{1/d})-1$ and its bisection width is $N^{(d-1)/d}$

The average distance is $d \times 2(N^{1/d})/3$ (without end-around connections).

3. Hypercube



A hypercube is a generalized cube. In a hypercube, there are 2^n nodes, for some n. Each node is connected to all other nodes whose numbers differ from it in only one bit position.

What is the degree of a hypercube?

What is the diameter of a hypercube?

What is the average distance?

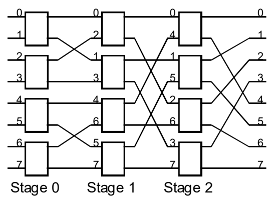
What is the bisection width?

An interconnection network can be either single stage or *multistage.*

- If it is single stage, then the individual control boxes must be set up to n times to get data from one node to another.

  Data may have to pass through several PEs to reach its destination.
- *Multistage* networks have several sets of switches in parallel, so data only needs to pass through several *switches*, not several nodes.

For a multistage cube network, we can diagram the paths from one cell to another like this:



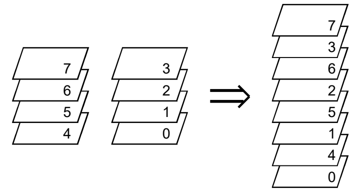
A multistage cube network is often called an *indirect binary n-cube.*

4. Perfect-shuffle interconnection

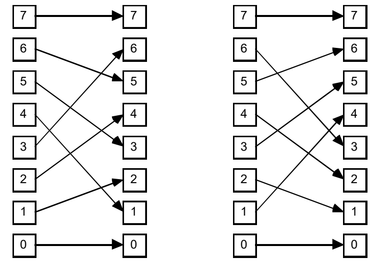
This interconnection network is defined by the routing function

$$S((a_{n-1} \ \dots \ a_1 a_0)_2) \equiv (a_{n-2} \ \dots \ a_1 a_0 \, a_{n-1})_2$$

It describes what happens when we divide a card deck of, e.g., 8 cards into two halves and shuffle them "perfectly."



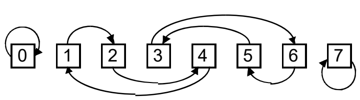
We can draw the processor interconnections required to obtain this transformation (at near right):



If the links are bidirectional, the *inverse perfect shuffle* is obtained (above, right).

5. Shuffle-exchange network

By itself, a shuffle network is not a complete interconnection network. This can be seen by looking at what happens as data is *recirculated* through the network:

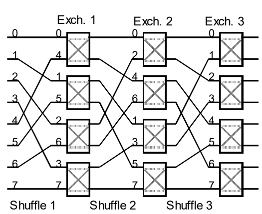


An *exchange* permutation can be added to a shuffle network to make it into a complete interconnection structure:

$$E(a_{n-1} \ \dots \ a_1 a_0)_2 + a_{n-1} \ \dots \ a_1 a_0$$

A shuffle-exchange network is isomorphic to a cube network, with a suitable renumbering of boxes.

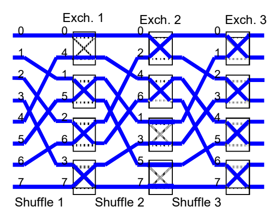
Here is a diagram of a multistage shuffle-exchange network for $N = 8$.



Sums (or other operations involving all the elements) can be performed in log N steps.

In addition, with a shuffle-exchange network, arbitrary cyclic shifts of an N-element array can be performed in log N steps.

This diagram shows how the switches in a shuffle-exchange network can be set to route input k to output $k + 3 \pmod 8$.



Switches are set to pass through or cross over depending on the exclusive-or of the input and output port numbers.

0 **xor** 3 = 000_2 **xor** 011_2 = 011 ➔ the first switch is set to pass through; the next two along the route are set to cross over.

1 **xor** 4 = 001_2 **xor** 100_2 = 101 ➔ the first switch is set to cross over, the next one to pass through, and the last one to cross over.

2 **xor** 5 = 010_2 **xor** 101_2 = 111 ➔ all three switches along the route are set to cross over.

The diameter of a shuffle-exchange network is

The bisection width is

6. Butterfly network

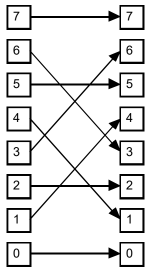
A butterfly network is closely related to shuffle-exchange networks.

The butterfly permutation is defined as—

$B(a_{n-1}\, a_{n-2} \ldots a_1\, a_0) \equiv a_0\, a_{n-2} \ldots a_1\, a_{n-1}$

i.e., the permutation formed by interchanging the most- and least-significant bits in the binary representation of the node number.

This permutation can be diagrammed as shown at the right:



Two variants of the butterfly permutation are the *k*th sub-butterfly, performed by interchanging bits 0 and *k*–1 in the binary representation—

$$B_k(a_{n-1}\, a_{n-2} \ldots a_1\, a_0) \equiv a_{n-1} a_{n-2}\, a_k\, a_0 \ldots a_1\, a_{k-1}$$

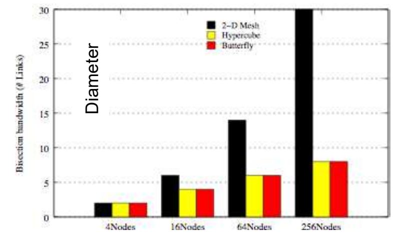
and the *k*th super-butterfly, peformed by interchanging bits *n*–1 and *k*–1:

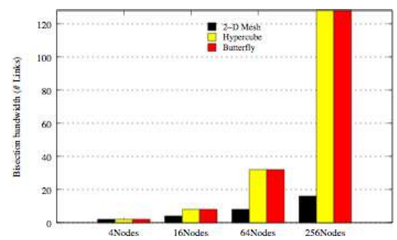
$$B^k(a_{n-1}\, a_{n-2} \ldots a_1\, a_0) \equiv a_{k-1}\, a_{n-2}\, a_k\, a_{n-1} a_{k-2} \ldots a_0$$

The textbook has an interesting diagram showing how metrics change with size for 2D meshes, hypercubes, and butterflies.

Explain what it says about increasing arity (*k*) vs. increasing dimension (*d*). Given the numbers here, which network would be more desirable for larger multiprocessors?

But why is this not the whole story?





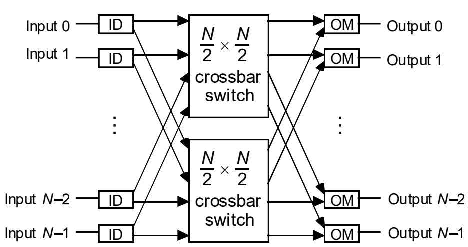
7. Beneš network

As we have seen, a crossbar switch is capable of connecting a set of inputs to *any* set of distinct outputs simultaneously.

A shuffle-exchange, or multistage cube, network is not capable of doing this. (It is easy to come up with an example.)

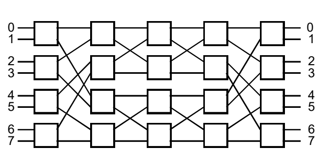
Is it possible to achieve an *arbitrary* permutation of input-output combinations with less than a full crossbar switch?

Yes. The Beneš network substitutes two $N/2 \times N/2$ crossbar switches, plus an *N*-input exchange switch for a full crossbar switch, as shown below.



The resulting $N/2 \times N/2$ crossbar switches can be similarly reduced.

Through this process, a full connection network can be produced from 2×2 switches at significantly lower cost than a full crossbar:



The stages of a Beneš network are connected by shuffle and inverse-shuffle permutations.

The network is called *rearrangeable*, since the switch settings can always be rearranged to accommodate any input-output mapping.

In some Beneš networks, the switches are capable of performing broadcasts, as well as pass-through or interchange.

Such Beneš networks can achieve all N^N possible input/output mappings.

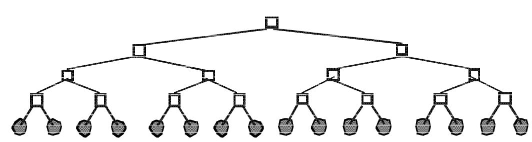
Trees

In meshes and hypercubes, the average distance increases with the *d*th root of *N*.

In a tree, the average distance grows only logarithmically.

A simple tree structure, however, suffers from two problems.

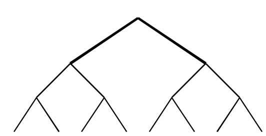
- Congestion
- Its fault tolerance is low.



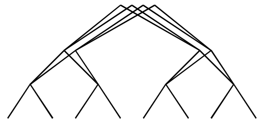
8. Fat trees

One approach to overcoming the limitations of the tree topology was devised by Leiserson and implemented in the Thinking Machines CM-5 data network.

The idea is that the edges at level *k* should have two or more times the capacity of the edges at level *k*+1 (the root is at level 0).



In reality, the links at higher levels are formed by replicating connections.



The algorithm for routing a message from processor *i* to processor *j* is as follows:

- Starting from processor *i*, a message moves up the tree along the path taking it to the first common ancestor of *i* and *j*.
- There are many possible paths, so at each level the routing processor chooses a path at random, in order to balance the load.
- Upon reaching the first common ancestor, the message is then routed down along the unique path connecting it to processor *j*.

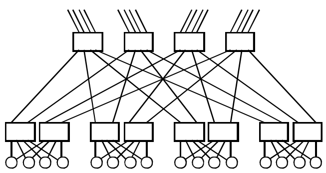
[What are some metrics](#) for a fat tree?

The diameter is

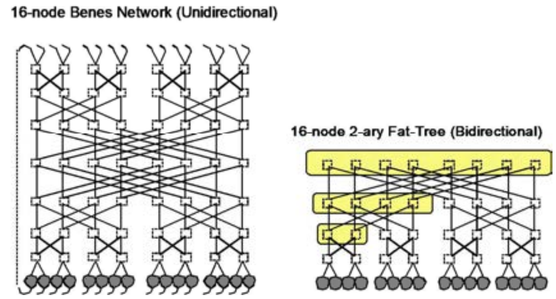
and its bisection width is

What is its degree?

We have shown a fat tree based on a binary tree. It may also be based on a *k*-ary tree. The CM-5 used fat trees based on 4-ary trees:



A *k*-ary fat tree can also be viewed as a *k*-ary Beneš network that is folded back on itself in the high-order dimension:



The collection of *N*/2 switches at level *i* is viewed as 2^{d-i} "fat nodes" of 2^{i-1} switches, where *d* is the *dimension* of the switch (where *d* is the number of levels in the tree—4 in the picture).

Routing algorithms

What path does a message travel from its source to its destination? This is determined by the routing algorithm.

Given a current node and a destination node, the routing algorithm chooses the next *port* and *channel* on which to send out the message.

Thus, a routing algorithm is a function $R: N \times N \rightarrow C$.

A switch usually uses one of three mechanisms to determine the output channel from info in the packet header.

- arithmetic,
- source-based port select, and
- a table lookup.

A switch needs to route a packet every few cycles, so it needs to be fast.

In regular topologies, simple arithmetic suffices.

Example: Δx, Δy routing in a grid. (What is Δx, Δy routing?)

[Finish](#) this example.

West ($-x$)	$\Delta x < 0$
East ($+x$)	
South ($-y$)	$\Delta x = 0$,
North ($+y$)	
Processor	$\Delta x = 0$, $\Delta y = 0$

To accomplish this routing, the switch needs to test the address in the header and increment or decrement one routing field.

Usually, routing is done in *dimension order*—first across the *x*-dimension, then the *y*-dimension, then the *z*-dimension (if any), etc.

So in a binary cube, the switch determines the most significant bit where the destination node number differs from the current node number.

Sometimes a packet header has a *relative address* embedded in it.

Example: If the source node is 001010 and the destination node is 100101, what would be the relative address embedded at the source node?

In this case, the switch just looks for the first non-zero bit and routes accordingly.

In general, in a mesh or cube, routing is done by moving from lowest (*x*) dimension to the highest.

This is called routing in *dimension order*. In a hypercube, it is called *e-cube* routing. It was used in nCube hypercubes and the Intel Paragon, among others.

More generally, the source builds a header consisting of the output port # for each switch along the route.

	P_3	P_2	P_1	P_0

This is called *source-based* routing.

Each switch just strips off the port number from the front of the message and sends it on.

Example: Let's assume we have a hypercube of dimension *n*, where *n* links are attached to each node. An output-port value of 0 means move across dimension 0 (i.e., flip the lsb); a value of *n*–1 means move across dimension *n*–1 (i.e., flip the msb).

Assume that *n* = 6, and that the header specifies these output ports:

	5	2	3	0

If the packet starts out at node 36 (= 100100_2), [where does it end up](#)?

All of the intelligence is at the source node. Arbitrary routing can be supported without much logic in the switches.

Disadvantage: Header is large, of variable size.

Table-driven routing associates a small routing table at each switch. It allows for a small fixed-size header.

In table-driven routing,

- The packet header contains a routing field *i*.
- The output port is *R*[*i*], where *R* is the routing table.
- Usually the table also contains the routing field for the next step in the route.

Disadvantage: The switch must contain quite a bit of routing state. Fairly large tables are needed even for simple routing algorithms.

This approach was used by ATM and HPPI switches, but isn't too practical for multiprocessors, because of the large number of routing patterns that they must support.

One important difference between network routers and multiprocessor switches: Time constraints.

Routing may be—

- *Deterministic*, where the route is completely determined by the (source, destination) pair, and not by the intermediate state, or
- *Adaptive*, where the route is influenced by traffic along the way.

A routing algorithm may be *minimal*, meaning that it only selects shortest paths toward the destination (no backtracking), or *nonminimal* (can allow longer paths).

Minimal algorithms need not be deterministic. Can you see why?

Similarly, adaptive algorithms may be minimal.

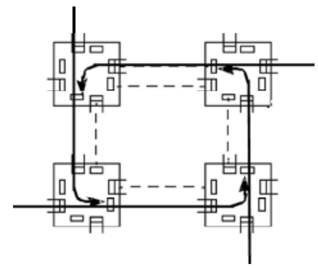
Let's compare the two types.

- Do nonminimal algorithms have any advantages over minimal algorithms?

- Do minimal algorithms have any advantages over nonminimal algorithms?

Deadlock-free routing

Deadlock occurs when two or more packets are "circularly" waiting for resources that are held by other packets in the group.



The diagram at the right illustrates how this can occur with 2-hop messages. Each packet is waiting for a link occupied by another packet.

What conditions are necessary for deadlock to occur?

- a *shared resource*
- that is *incrementally allocated* and
- non-preemptible.

A channel is a shared resource, and channels are acquired incrementally, as a route is built up.

How can deadlock be avoided? Basically, by constraining how channel resources are allocated. Routing in dimension order is one way. How can we see that in the diagram above?

How do we prove that an algorithm is deadlock free?

- Each channel *is* a resource (or each channel *contains* resources).

- Messages need to use resources in order; this introduces dependences between resources.
- We need to show that there are no cycles in the channel-dependence graph

We can do this by finding a numbering of channel resources such that every legal route follows a monotonic sequence

$\Rightarrow$ no traffic pattern can lead to deadlock

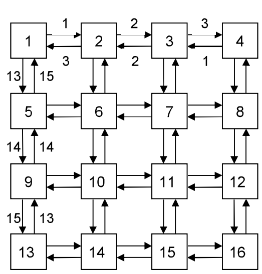
This is trivial for acyclic networks, such as shuffle-exchange and butterfly networks.

It is also trivial for trees and fat trees, as long as the upward and downward channels are independent.

Example: Show Δ*x*, Δ*y* routing on a *k*-ary 2D array is deadlock free.

We view each bidirectional channel as a pair of unidirectional channels numbered independently.

Channels in the *x* direction are given lower numbers than channels in the *y* direction.



Given this numbering, any routing sequence that starts out in the *x* direction, turns and then goes in the *y* direction is increasing.

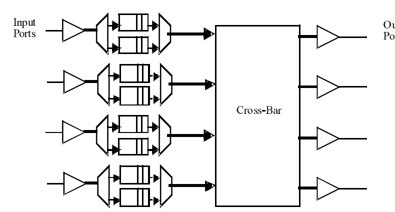
Virtual channels

This proof easily generalizes to hypercubes—but not to toruses because of the wraparound edges.

So, how can we do deadlock-free routing on toruses and other higher-order networks?

The basic approach is to provide *virtual channels.*

To do this we need more than one packet buffer per channel.



Example: Consider a torus with unidirectional links.

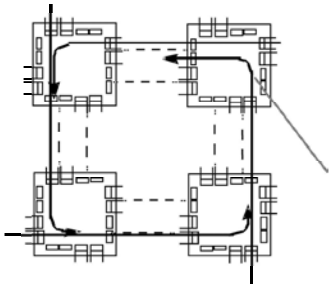
Reserve one packet buffer on each channel for messages destined for nodes with a higher number than their source, i.e., messages that don't use wraparound channels.

Now such messages will always be able to make progress.

Wraparound messages may be delayed, but the network will not deadlock.

Notice that this requires no more links or switches, just more buffers.

How can we break the four-message deadlock?



Let's provide two virtual channels per physical channel, as shown above.

- Messages at a node numbered higher than their destination are routed on the "high" channels.
- Messages at a node numbered lower than their destination are routed on the "low" channels.

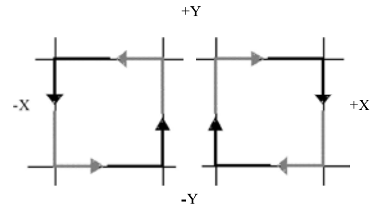
This breaks the dependence cycle.

Example: In the *k*-ary 2D array above, let's assume we have messages 1 → 4, 4 → 12, 12 → 10, and 10 → 1. Show that all messages can make progress.

This strategy can be generalized to *d* dimensions.

Turn-model routing

Note that *x*-*y* routing in dimension order allows only four of the eight possible turns a packet might make on its way to its destination.

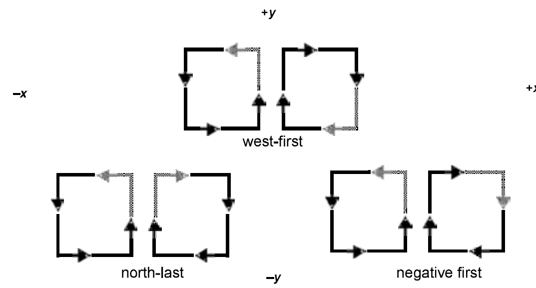


When a packet is traveling in the ±*x* direction, it is legal to turn into the ±*y* direction, but once it is traveling in the ±*y* direction, it can make no further turns.

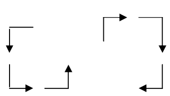
Intuitively, we could prevent deadlock by disallowing only one turn in each cycle.

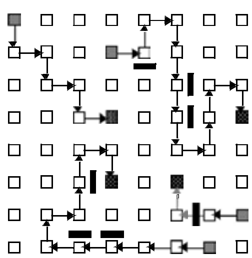
It turns out that of the 16 possible ways to prohibit two turns in a 2D array, 12 prevent deadlock.

These consist of the three algorithms below, and rotations of them.



Can you see that this "algorithm" does not prevent deadlock?





Each of these algorithms allows nonminimal routes to be followed. On the right are some legal west-first routes.

Exercise (on paper): Give one other example of a legal turn-model routing algorithm, and one that is not deadlock free.

Adaptive routing

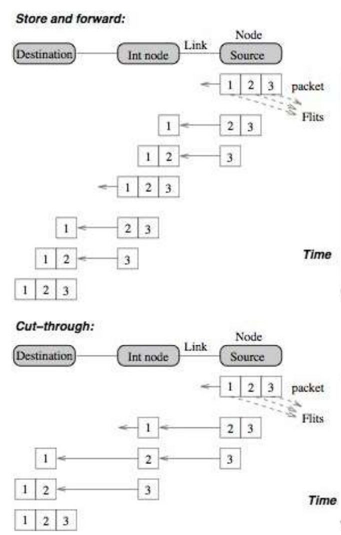
Adaptive routing has several advantages.

- If there is only one route from source to destination, failure of a single link can leave the network disconnected.
- It can allow messages to avoid regions where there are long queues and a lot of contention.

One interesting adaptive algorithm is "hot-potato" routing.

A switch never buffers packets. If > 1 packet is destined for the same output channel, the switch "misroutes" all but one.

Store and Forward vs. Cut-Through



Early networks used store-and-forward routing:

A packet was sent from one node to an intermediate node and buffered there.

But this took a lot of time:

$T = h \times (T_{switch} + T_{xmit})$

For the last 25 years or so, cut-through routing has taken over.

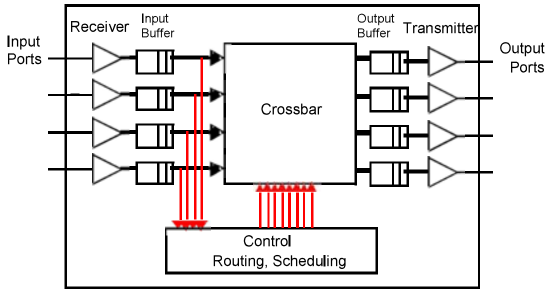
Effectively, a pipeline is set up, and flits are sent through, one after another, and are switched without being buffered.

$T = h \times$

Switch Design

[§12.4] In earlier lectures, we have seen that switches in an interconnection network connect inputs to outputs, usually with some kind of buffering.

Here is a basic diagram of a switch.



Usually, the number of input ports is the same as the number of output ports. This is called the *degree* of the switch.

Each input port includes a receiver. A received message goes into an input buffer.

Each output port includes a transmitter to drive the link. There may also be an output buffer.

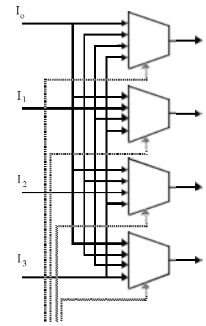
The control logic implements the routing algorithm. It must include a way to arbitrate among competing requests for the same output port.

A major constraint on switch size is the number of pins. How would this be determined?

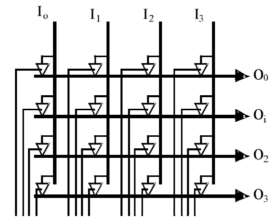
This tends to favor fast serial links. They use the least pins and eliminate problems with skew across bit lines in a channel.

With parallel links, one of the wires is essentially a clock for the data on the others.

Logically, the crossbar in an $n \times n$ switch is just an n-way multiplexer associated with each dimension (at right).



In VLSI, it is typically realized as a bus with a set of n tristate drivers (below).



An increasingly common technique is to use memory as a crossbar, by writing for each input port and reading for each output port. [Explain](#).