Shared Memory Without A Bus

In SMPs the bus is a centralized point where the writes can be serialized. When no such point exists, as in large parallel computers, the situation gets very much more complicated. We continue our examination of shared memory implementations

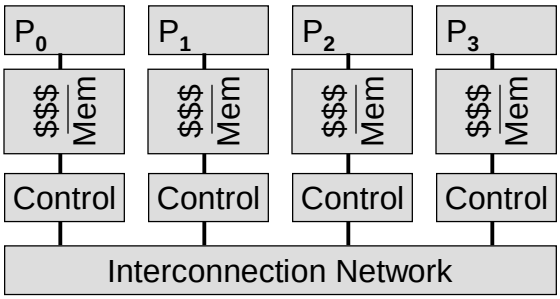
Source: Culler/Singh, Parallel Computer Architectures, MK '99

Preliminaries

- The computers implementing shared memory without a central bus are called "distributed shared memory" (DSM) machines
- The subclass is the CC-NUMA machines, for cache coherent non-uniform memory access
- On an access-fault by the processor
  - Find out information about the state of the cache block in other machines
  - Determine the exact location of copies, if necessary
  - Communicate with other machines to implement the shared memory protocol

"Distributed" Applies to Memory

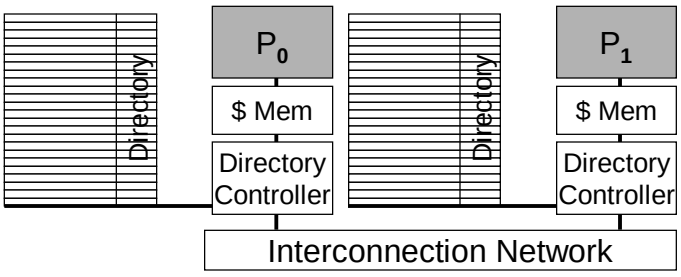
- DSM computers have a CTA architecture with additional hardware to maintain coherency
- Collectively, the controllers make the memory look shared



Directory Based Cache-coherence

Since broadcasting the memory references is impractical -- that's what buses do -- a directory-based scheme is an alternative

- A directory is a data structure giving the state of each cache block in the machine



How Does It Work?

- Using the directory it is possible to maintain cache coherency in a DSM, but its complex (and time consuming)
- To illustrate, we work through the protocols to maintain memory coherency
- Concepts
  - Events: A read or write access fault
  - Cache fields these for local data, controller fields these for remotely allocated data
  - Proc/Proc communication is by packets through the interconnection network

Terminology

- Node, a processor, cache and memory
- Home node, node whose main memory has the block allocated
- Dirty node, a node with a modified value
- Owner, node holding a valid copy, usually the home or dirty node
- Exclusive node, holds only valid cached copy
- Requesting node, (local) node asking for the block

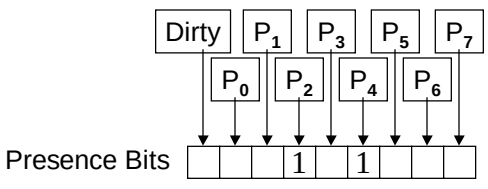
Sample Directory Scheme

- Local node has access fault
- Sends request to home node for directory information
  - Read -- directory tells which node has the valid data and the data is requested
  - Write -- directory tells nodes with copies ... Invalidation or update requests are sent
- Acknowledgments are returned
- Processor waits for all ACKs before completion

Notice that many transactions can be "in the air" at once, leading to possible races

A Directory Entry

- Directory entries don't usually keep cache state
- Use a P-length bit-vector to tell in which processors the block is present … presence bit
- Clean/dirty bit implies exactly 1 presence bit on
- Sufficient?
  - Determine who has valid copy for read-miss
  - Determine who has copies to be invalidated



A Closer Look (Read) I

- Postulate 1 processor per node, 1 level cache, local MSI protocol [from last week]
- On a read access fault at P_x, the local directory controller determines if block is locally/remotely allocated
  - If local, it delivers data
  - If remote it finds the home … by high order bits probably
- Controller sends request to home node for blk
- Home controller looks up directory entry for blk
  - Dirty bit OFF, controller finds blk in memory, sends reply, sets x^{th} presence bit ON

A Closer Look (Read) II

- Dirty bit ON -- controller sends reply to P_x of the processor ID of P_y, the owner
- P_x requests data from owner P_y
- Owner P_y controller, sets state to "shared," forwards data to P_x and sends data to home
- At home, data is updated, dirty bit is turned OFF and the x^{th} presence bit is set ON and y^{th} presence bit remains ON

This is basically the protocol for the LLNL S-1 multicomputer from the late '70s

A Closer Look (Write) I

On a write access fault at P_x, the local directory controller checks if the block is locally/remotely allocated; if remote it finds the home

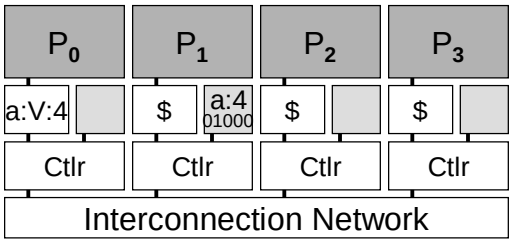
- Controller sends request to home node for blk
- Home controller looks up directory entry of blk
  - Dirty bit OFF, the home has a clean copy
    - Home node sends data to P_x w/presence vector
    - Home controller clears directory, sets x^{th} bit ON and sets dirty bit ON
    - P_x controller sends invalidation request to all nodes listed in the presence vector

A Closer Look (Write) II

    - P_x controller awaits ACKs from all those nodes
    - P_x controller delivers blk to cache in dirty state
  - Dirty bit is ON
    - Home notifies owner P_y of P_x's write request
    - P_y controller invalidates its blk, sends data to P_x
    - Home clears y^{th} presence bit, turns x^{th} bit ON and dirty bit stays ON
  - On writeback, home stores data, clears both presence and dirty bits

Detailed Example

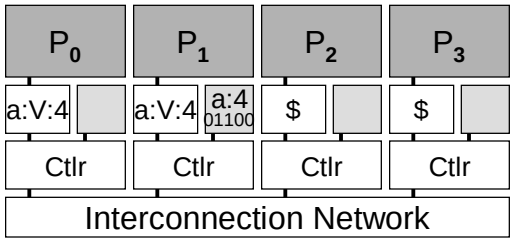
- Consider the example similar to last week
- The assumptions are …
  - a is globally allocated
  - a has it's home at P_1
  - P_0 previously read a



P_1 reads a into its cache
P_3 reads a into its cache
P_3 changes a to 5
{P_2 reads a into its cache
P_2 writes a in its cache}

P_1 Reads a Into Cache

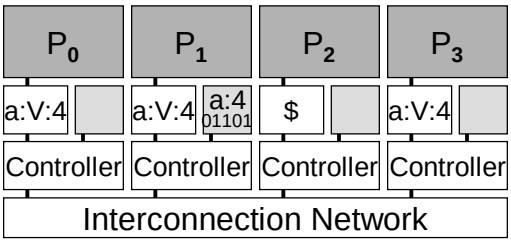
- The local directory controller determines if block is locally/remotely allocated
  - If remote it finds the home … by high order bits probably
- Controller asks home node for blk: No-op
- Home controller looks up directory entry for blk
  - Dirty bit OFF, controller finds blk in memory, sends reply, sets x^{th} presence bit ON



In the special case that a processor references it's own globally allocated data no communication is required, only manage the presence bits

P_3 Reads a Into Cache

- The local directory controller determines if block is locally/remotely allocated
  - If remote it finds the home … by high order bits probably
- Controller asks home node for blk: Message to P_1
- Home controller looks up directory entry for blk
  - Dirty bit OFF, controller finds blk in memory, sends message to P_3, sets x^{th} presence bit ON

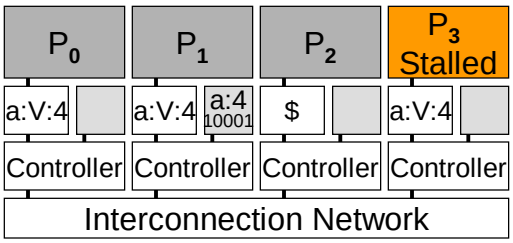


Msg: P_3 to P_1, Read a

Msg: P_1 to P_3, Here's a

P_3 Writes a Changing It To 5 Part I

- On a write access fault at P_x, local controller checks and finds it remote; finds the home
- Controller sends request to home node for blk
- Home controller looks up directory entry of blk
  - Dirty bit OFF, the home has a clean copy
    - Home node sends data to P_x w/presence vector
    - Home controller clears directory, sets x^{th} bit and dirty ON
    - P_x controller sends invalidation request to all nodes listed



Msg: P_3 to P_1, Write a

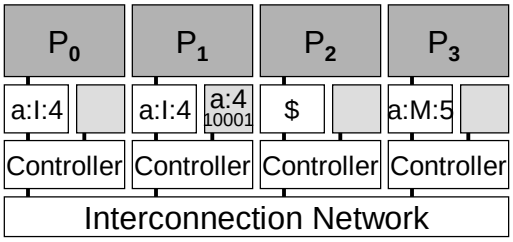
Msg: P_1 to P_3, a:01101

Msg: P_3 to P_0, Invalid a

Msg: P_3 to P_1, Invalid a

P_3 Writes a Changing It To 5 Part II

- Processor continues to be stalled
  - P_x controller awaits ACKs from all those nodes
  - P_x controller delivers blk to cache in dirty state
- Total messages when clean copy exists: ToHome, FromHome, (Invalidate, ACK)*S



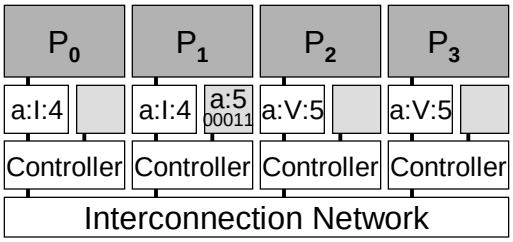
Msg: P_0 to P_3, ACK

Msg: P_1 to P_3, ACK

P_2 Reads a Into Cache

Dirty bit ON -- home controller sends reply to P_x of the processor ID of P_y, the owner; P_x asks P_y for data

- Owner P_y controller, sets state to "shared," forwards data to P_x and sends data to home
- At home, data is updated, dirty bit is turned OFF and the x^{th} presence bit is set ON and y^{th} presence bit remains ON



Msg: P_2 to P_1, Read a

Msg: P_1 to P_2, P_3 has it

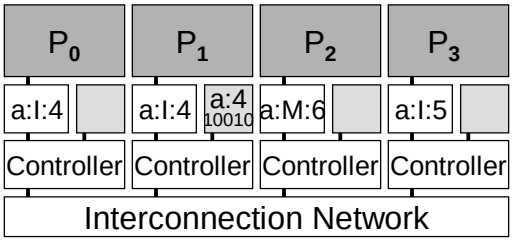
Msg: P_2 to P_3, Read a

Msg: P_3 to P_2, Here's a

Msg: P_3 to P_1, Here's a

Instead Let P_2's Request Be Write 6

- That is … this action replaces the previous slide
- Dirty bit is ON
  - Home notifies owner P_y of P_x's write request
  - P_y controller invalidates its block, sends data to P_x
  - Home clears y^{th} presence bit, turns x^{th} bit ON and dirty bit stays ON



Msg: P_2 to P_1, Write a

Msg: P_1 to P_3, P_2 asking

Msg: P_3 to P_2, Here's a

Summarizing The Example

- The controller sends out a series messages to keep the writes to the memory locations coherent
- The scheme differs from the bus solution in that all processors get the information at the same time using the bus, but at different times using the network
- The number of messages is potentially large if there are many sharers

Homework Assignment

Suppose a 100x100 array S is distributed by blocks across four processors, so that each contains a 50x50 subarray. At each position S[i,j] is updated by the sum of its 8 nearest neighbors:

S'[i,j]=(S[i-1,j-1]+S[i-1,j]+S[i-1,j+1]+S[i,j+1]+S[i+1,j+1]
+S[i+1,j]+S[i+1,j-1]+S[i,j-1])/8;

If each processor updates its own elements, how many messages must are produced to maintain concurrency for a directory based CC-NUMA?

HINT: Assume that an extra row and column, initialized to zero, surrounds A, A is allocated in rmo, and storage for S alternates with S'

Break

Alternative Directory Schemes

- The “bit vector directory” is storage-costly
- Consider improvements to Mblk*P cost
  - Increase block size, cluster processors
  - Just keep list of Processor IDs of sharers
    - Need overflow scheme
    - Five slots probably suffice
  - Link the shared items together
    - Home keeps the head of list
    - List is doubly-linked
    - New sharer adds self to head of list
    - Obvious protocol suffices, but watch for races

Assessment

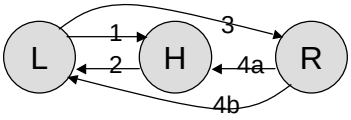
- An obvious difference between directory and bus solutions is that for directories, the invalidate request grows as the number of processors that are sharing
- Directories take memory
  - 1 bit per block per processor + c
  - If a block is B bytes, 8B processors imply 100% overhead to store the directory

Performance Data

- To see how much sharing takes place and how many invalidations must be sent, experiments were run
- Summarizing the data
  - Usually there are few shares
  - The mode is 1 other processor(s) sharing ~ 60
  - The “tail” of the distribution stretches out for some applications
- Remote activity increases as the number of processors
- Larger block sizes increase traffic, 32 is good

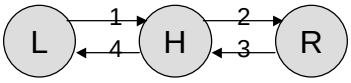
Protocol Optimizations I

- Read Request to Exclusively Held Block



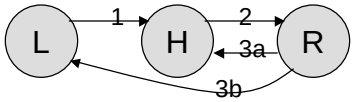
1: Request
2: Response
3: Intervention
4a: Revise
4b: Response

Strict Request/ Response



1: Request
2: Intervention
3: Revise
4: Response

Intervention Forwarding

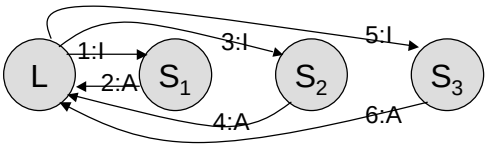


1: Request
2: Intervention
3a: Revise
3b: Response

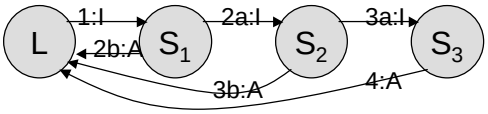
Reply Forwarding

Protocol Optimizations II (Lists)

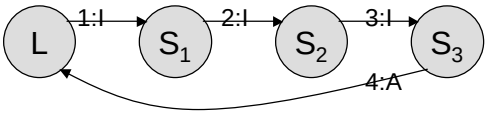
- Improved Invalidation



ACK includes next sharer on list



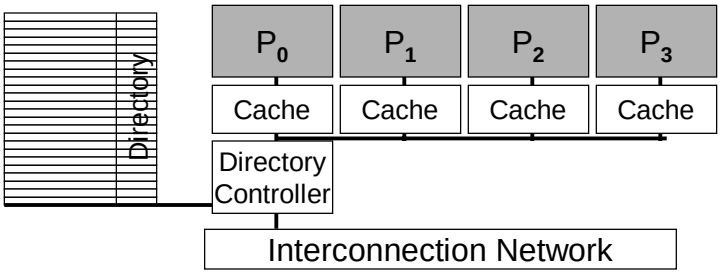
ACK and next invalidate in parallel



ACK comes from the last sharer

Higher Level Optimization

- Organizing nodes as SMPs with one coherent memory and one directory controller can improve performance since one processor might fetch data that the next processor wants … it is already present
- The main liability is that the controller resource and probably its channel into the network are shared



Serialization

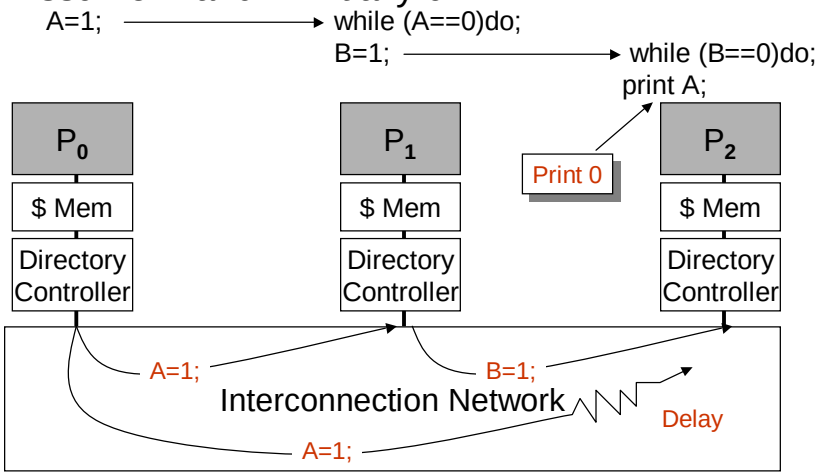
- The bus defines the ordering on writes in SMPs
- For directory systems, memory (home) does
- If home always has value, FIFO would work
  - Consider a block in modified state and two nodes requesting exclusive access in an invalidation protocol: The requests reach home in one order, but they could reach the owner in a different order; which order prevails?
- Fix: Add “busy state” indicating transaction in flight

Four Solutions To Ensure Serialization

- Buffer at home -- keep request at home, service in order … lower concurrency, overflow
- Buffer at requesters with linked list; follow P_y
- NACK and retry -- when directory is busy, just “return to sender”
- Forward to dirty node -- serialize at home for clean, serialize at owner otherwise

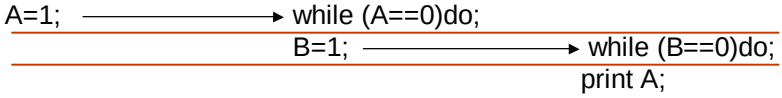
Coherency != Memory Consistency

Assume A and B initially 0



Sequential Consistency

- Sequential Consistency is a very strict form of memory consistency
- A MP is *sequentially consistent* if the result of any execution is the same as some sequential order and operations of each processor are in program order



Relaxed Consistency Models

- Since sequential consistency is so strict, alternative schemes allow reordering of reads and writes to improve performance
  - total store ordering (TSO)
  - partial store ordering (PSO)
  - relaxed memory ordering (RMO)
  - processor consistency (PC)
  - weak ordering (WO)
  - release consistency (RC)
- Many are difficult to use in practice

Relaxing Write-to-Read Program Order

- While a write miss is in the write buffer and not yet visible to other processors, the processor can issue and complete reads that hit in its cache or even a single read that misses in its cache. TSO and PSO allow this.
- This matches intuition often …

P_0	P_1
A=1;	while (Flag==0)do;
Flag=1;	print A;

P_0	P_1
A=1;	print B;
B=1;	print A;

- This code works as expected

Less Intuitive

- Some programs don't work as expected

P_0	P_1
A=1;	B=1;
print B;	print A;

We expect to get one of the following:

- A=0, B=1
- A=1, B=0
- A=1, B=1

- But not A=0, B=0 … but TSO would permit it
- Solution: Insert a memory barrier after write

Origin 2000

- Intellectual descendant of Stanford DASH
- Two processors per node
- Caches use MESI protocol
- Directory has 7 states:
  - Stable: unowned, shared, exclusive (cl/dirty in $)
  - Busy: Processor not ready to handle new requests to block, read, readex, uncached
- Generally O2000 follows protocols discussed
  - Proves basic ideas actually apply
  - Shows that simplifying assumptions must be revisited to get a system built and deployed

Summary

- Shared memory support is much more difficult when there is no bus
- A directory scheme achieves the same result, but the protocol requires a substantial number of messages, proportional to the amount of sharing
- Coherency applies to individual locations
- Consistent memory requires additional software or hardware to assure that updates or invalidations are complete