Lecture 11:
Multiprocessor 1: Reasons, Classifications, Performance Metrics, Applications

February 23, 2001
Prof. David A. Patterson
Computer Science 252
Spring 2001

Review: Networking

- Clusters +: fault isolation and repair, scaling, cost
- Clusters –: maintenance, network interface performance, memory efficiency
- Google as cluster example:
  - scaling (6000 PCs, 1 petabyte storage)
  - fault isolation (2 failures per day yet available)
  - repair (replace failures weekly/repair offline)
  - Maintenance: 8 people for 6000 PCs
- Cell phone as portable network device
  - # Handsets >> # PCs
  - Univerisal mobile interface?
- Is future services built on Google-like clusters delivered to gadgets like cell phone handset?

Parallel Computers

- Definition: "A parallel computer is a collection of processiong elements that cooperate and communicate to solve large problems fast."
    Almasi and Gottlieb, *Highly Parallel Computing* ,1989
- Questions about parallel computers:
  - How large a collection?
  - How powerful are processing elements?
  - How do they cooperate and communicate?
  - How are data transmitted?
  - What type of interconnection?
  - What are HW and SW primitives for programmer?
  - Does it translate into performance?

Parallel Processors "Religion"

- The dream of computer architects since 1950s: replicate processors to add performance vs. design a faster processor
- Led to innovative organization tied to particular programming models since "uniprocessors can't keep going"
  - e.g., uniprocessors must stop getting faster due to limit of speed of light: 1972, … , 1989
  - Borders religious fervor: you must believe!
  - Fervor damped some when 1990s companies went out of business: Thinking Machines, Kendall Square, ...
- Argument instead is the "pull" of opportunity of scalable performance, not the "push" of uniprocessor performance plateau?

What level Parallelism?

- Bit level parallelism: 1970 to ~1985
  - 4 bits, 8 bit, 16 bit, 32 bit microprocessors
- Instruction level parallelism (ILP): ~1985 through today
  - Pipelining
  - Superscalar
  - VLIW
  - Out-of-Order execution
  - Limits to benefits of ILP?
- Process Level or Thread level parallelism; mainstream for general purpose computing?
  - Servers are parallel
  - Highend Desktop dual processor PC soon?? (or just the sell the socket?)

Why Multiprocessors?

- Microprocessors as the fastest CPUs
  - Collecting several much easier than redesigning 1
- Complexity of current microprocessors
  - Do we have enough ideas to sustain 1.5X/yr?
  - Can we deliver such complexity on schedule?
- Slow (but steady) improvement in parallel software (scientific apps, databases, OS)
- Emergence of embedded and server markets driving microprocessors in addition to desktops
  - Embedded functional parallelism, producer/consumer model
  - Server figure of merit is tasks per hour vs. latency

Parallel Processing Intro

- Long term goal of the field: scale number processors to size of budget, desired performance
- Machines today: Sun Enterprise 10000 (8/00)
  - 64 400 MHz UltraSPARC® II CPUs,64 GB SDRAM memory, 868 18GB disk,tape
  - $4,720,800 total
  - 64 CPUs 15%,64 GB DRAM 11%, disks 55%, cabinet 16% ($10,800 per processor or ~0.2% per processor)
  - Minimal E10K – 1 CPU, 1 GB DRAM, 0 disks, tape ~$286,700
  - $10,800 (4%) per CPU, plus $39,600 board/4 CPUs (~8%/CPU)
- Machines today: Dell Workstation 220 (2/01)
  - 866 MHz Intel Pentium® III (in Minitower)
  - 0.125 GB RDRAM memory, 1 10GB disk, 12X CD, 17" monitor, nVIDIA GeForce 2 GTS,32MB DDR Graphics card, 1yr service
  - $1,600; for extra processor, add $350 (~20%)

Whither Supercomputing?

- Linpack (dense linear algebra) for Vector Supercomputers vs. Microprocessors
- "Attack of the Killer Micros"
  - (see Chapter 1, Figure 1–10, page 22 of [CSG99])
  - 100 x 100 vs. 1000 x 1000
- MPPs vs. Supercomputers when rewrite linpack to get peak performance
  - (see Chapter 1, Figure 1–11, page 24 of [CSG99])
- 1997, 500 fastest machines in the world: 319 MPPs, 73 bus–based shared memory (SMP), 106 parallel vector processors (PVP)
  - (see Chapter 1, Figure 1–12, page 24 of [CSG99])
- 2000, 381 of 500 fastest: 144 IBM SP (~cluster), 121 Sun (bus SMP), 62 SGI (NUMA SMP), 54 Cray (NUMA SMP)

  [CSG99] = Parallel computer architecture : a hardware/ software approach, David E. Culler, Jaswinder Pal Singh, with Anoop Gupta. San Francisco : Morgan Kaufmann, c1999.

Popular Flynn Categories
(e.g., ~RAID level for MPPs)

- SISD (Single Instruction Single Data)
  - Uniprocessors
- MISD (Multiple Instruction Single Data)
  - ???; multiple processors on a single data stream
- SIMD (Single Instruction Multiple Data)
  - Examples: Illiac-IV, CM-2
    » Simple programming model
    » Low overhead
    » Flexibility
    » All custom integrated circuits
  - (Phrase reused by Intel marketing for media instructions ~ vector)
- MIMD (Multiple Instruction Multiple Data)
  - Examples: Sun Enterprise 5000, Cray T3D, SGI Origin
    » Flexible
    » *Use off–the–shelf micros*
- MIMD current winner: Concentrate on major design emphasis <= 128 processor MIMD machines

Major MIMD Styles

- Centralized shared memory ("Uniform Memory Access" time or "Shared Memory Processor")
- Decentralized memory (memory module with CPU)
  - get more memory bandwidth, lower memory latency
  - Drawback: Longer communication latency
  - Drawback: Software model more complex

Decentralized Memory versions

- Shared Memory with "Non Uniform Memory Access" time (NUMA)
- Message passing "multicomputer" with separate address space per processor
  - Can invoke software with Remote Procedue Call (RPC)
  - Often via library, such as MPI: Message Passing Interface
  - Also called "Syncrohnous communication" since communication causes synchronization between 2 processes

Performance Metrics:
Latency and Bandwidth

- Bandwidth
  - Need high bandwidth in communication
  - Match limits in network, memory, and processor
  - Challenge is link speed of network interface vs. bisection bandwidth of network
- Latency
  - Affects performance, since processor may have to wait
  - Affects ease of programming, since requires more thought to overlap communication and computation
  - Overhead to communicate is a problem in many machines
- Latency Hiding
  - How can a mechanism help hide latency?
  - Increases programming system burden
  - Examples: overlap message send with computation, prefetch data, switch to other tasks

Page 2

CS 252 Administrivia

- Meetings this week
- Next Wednesday guest lecture
- Flash vs. Flash paper reading
- Quiz #1 Wed March 7 5:30-8:30 306 Soda
- La Val's afterward quiz: free food and drink

Parallel Architecture

- Parallel Architecture extends traditional computer architecture with a communication architecture
  - abstractions (HW/SW interface)
  - organizational structure to realize abstraction efficiently

Parallel Framework

- Layers:
  - (see Chapter 1, Figure 1-13, page 27 of [CSG99])
  - Programming Model:
    » Multiprogramming : lots of jobs, no communication
    » Shared address space: communicate via memory
    » Message passing: send and recieve messages
    » Data Parallel: several agents operate on several data sets simultaneously and then exchange information globally and simultaneously (shared or message passing)
  - Communication Abstraction:
    » Shared address space: e.g., load, store, atomic swap
    » Message passing: e.g., send, recieve library calls
    » Debate over this topic (ease of programming, scaling) => many hardware designs 1:1 programming model

Shared Address Model Summary

- Each processor can name every physical location in the machine
- Each process can name all data it shares with other processes
- Data transfer via load and store
- Data size: byte, word, ... or cache blocks
- Uses virtual memory to map virtual to local or remote physical
- Memory hierarchy model applies: now communication moves data to local processor cache (as load moves data from memory to cache)
  - Latency, BW, scalability when communicate?

Shared Address/Memory Multiprocessor Model

- Communicate via Load and Store
  - Oldest and most popular model
- Based on timesharing: processes on multiple processors vs. sharing single processor
- process: a virtual address space and ~ 1 thread of control
  - Multiple processes can overlap (share), but ALL threads share a process address space
- Writes to shared address space by one thread are visible to reads of other threads
  - Usual model: share code, private stack, some shared heap, some private heap

SMP Interconnect

- Processors to Memory AND to I/O
- Bus based: all memory locations equal access time so SMP = "Symmetric MP"
  - Sharing limited BW as add processors, I/O
  - (see Chapter 1, Figs 1-17, page 32-33 of [CSG99])

Page 3

Message Passing Model

- Whole computers (CPU, memory, I/O devices) communicate as explicit I/O operations
  - Essentially NUMA but integrated at I/O devices vs. memory system
- **Send** specifies local buffer + receiving process on remote computer
- **Receive** specifies sending process on remote computer + local buffer to place data
  - Usually send includes process tag and receive has rule on tag: match 1, match any
  - **Synch**: when send completes, when buffer free, when request accepted, receive wait for send
- Send+receive => memory-memory copy, where each each supplies local address, AND does pairwise sychronization!

Data Parallel Model

- Operations can be performed in parallel on each element of a large regular data structure, such as an array
- 1 Control Processsor broadcast to many PEs (see Ch. 1, Fig. 1-25, page 45 of [CSG99])
  - When computers were large, could amortize the control portion of many replicated PEs
- Condition flag per PE so that can skip
- **Data distributed in each memory**
- Early 1980s VLSI => SIMD rebirth: 32 1-bit PEs + memory on a chip was the PE
- Data parallel programming languages lay out data to processor

Data Parallel Model

- Vector processors have similar ISAs, but no data placement restriction
- SIMD led to Data Parallel Programming languages
- Advancing VLSI led to single chip FPUs and whole fast μProcs (SIMD less attractive)
- SIMD programming model led to Single Program Multiple Data (SPMD) model
  - All processors execute identical program
- Data parallel programming languages still useful, do communication all at once: "Bulk Synchronous" phases in which all communicate after a global barrier

Advantages shared-memory communication model

- **Compatibility with SMP hardware**
- **Ease of programming when communication patterns are complex or vary dynamically during execution**
- **Ability to develop apps using familiar SMP model, attention only on performance critical accesses**
- **Lower communication overhead, better use of BW for small items, due to implicit communication and memory mapping to implement protection in hardware, rather than through I/O system**
- **HW-controlled caching to reduce remote comm. by caching of all data, both shared and private.**

Advantages message-passing communication model

- **The hardware can be simpler (esp. vs. NUMA)**
- **Communication explicit => simpler to understand; in shared memory it can be hard to know when communicating and when not, and how costly it is**
- **Explicit communication focuses attention on costly aspect of parallel computation, sometimes leading to improved structure in multiprocessor program**
- **Synchronization is naturally associated with sending messages, reducing the possibility for errors introduced by incorrect synchronization**
- **Easier to use sender-initiated communication, which may have some advantages in performance**

Communication Models

- **Shared Memory**
  - Processors communicate with shared address space
  - Easy on small-scale machines
  - Advantages:
    - » Model of choice for uniprocessors, small-scale MPs
    - » Ease of programming
    - » Lower latency
    - » Easier to use hardware controlled caching
- **Message passing**
  - Processors have private memories, communicate via messages
  - Advantages:
    - » Less hardware, easier to design
    - » Focuses attention on costly non-local operations
- **Can support either SW model on either HW base**

3 Parallel Applications

- Commercial Workload
- Multiprogramming and OS Workload
- Scientific/Technical Applications

Parallel App: Commercial Workload

- **Online transaction processing workload (OLTP) (like TPC-B or –C)**
- **Decision support system (DSS) (like TPC-D)**
- **Web index search (Altavista)**

Benchmark	% Time User Mode	% Time Kernel	% Time I/O time (CPU Idle)
OLTP	71%	18%	11%
DSS (range)	82–94%	3–5%	4–13%
DSS (avg)	87%	4%	9%
Altavista	> 98%	< 1%	<1%

Parallel App: Multiprogramming and OS

- **2 independent copies of the compile phase of the Andrew benchmark (parallel make 8 CPUs)**
- **3 phases: compiling the benchmarks; installing the object files in a library; removing the object files (I/O little, lot, almost all)**

	User	Kernel	Synch. wait	I/O (CPU Idle)
% instructions	27%	3%	1%	69%
% time	27%	7%	2%	64%

Parallel App: Scientific/Technical

- **FFT Kernel: 1D complex number FFT**
  - 2 matrix transpose phases => all-to-all communication
  - Sequential time for n data points: $O(n \log n)$
  - Example is 1 million point data set
- **LU Kernel: dense matrix factorization**
  - Blocking helps cache miss rate, 16x16
  - Sequential time for nxn matrix: $O(n^3)$
  - Example is 512 x 512 matrix

Parallel App: Scientific/Technical

- **Barnes App: Barnes-Hut n-body algorithm solving a problem in galaxy evolution**
  - n-body algs rely on forces drop off with distance; if far enough away, can ignore (e.g., gravity is $1/d^2$)
  - Sequential time for n data points: $O(n \log n)$
  - Example is 16,384 bodies
- **Ocean App: Gauss-Seidel multigrid technique to solve a set of elliptical partial differential eq.s'**
  - red-black Gauss-Seidel colors points in grid to consistently update points based on previous values of adjacent neighbors
  - Multigrid solve finite diff. eq. by iteration using hierarch. Grid
  - Communication when boundary accessed by adjacent subgrid
  - Sequential time for nxn grid: $O(n^2)$
  - Input: 130 x 130 grid points, 5 iterations

Parallel Scientific App: Scaling

- p is # processors
- n is + data size
- Computation scales up with n by O( ), scales down linearly as p is increased
- Communication
  - FFT all-to-all so n
  - LU, Ocean at boundary, so $n^{1/2}$
  - Barnes complex: $n^{1/2}$ greater distance, x log n to maintain bodies relationships
  - All scale down $1/p^{1/2}$

App	Scaling computation	Scaling communication	Scaling comp-to-comm
FFT	$n \log n/p$	n/p	$\log n$
LU	n/p	$n^{1/2}/p^{1/2}$	$n^{1/2}/p^{1/2}$
Barnes	$n \log n/p$	$n^{1/2} \log n /p^{1/2}$	$n^{1/2}/p^{1/2}$
Ocean	n/p	$n^{1/2}/p^{1/2}$	$n^{1/2}/p^{1/2}$

- Keep n same, but inc. p?
- Inc. n to keep comm. same w. +p?

Amdahl's Law and Parallel Computers

- Amdahl's Law (FracX: original % to be speed up)
  Speedup = 1 / [(FracX/SpeedupX + (1-FracX)]
- A portion is sequential => limits parallel speedup
  - Speedup <= 1/ (1-FracX)
- Ex. What fraction sequetial to get 80X speedup from 100 processors? Assume either 1 processor or 100 fully used

80 = 1 / [(FracX/100 + (1-FracX)]

0.8*FracX + 80*(1-FracX) = 80 – 79.2*FracX = 1
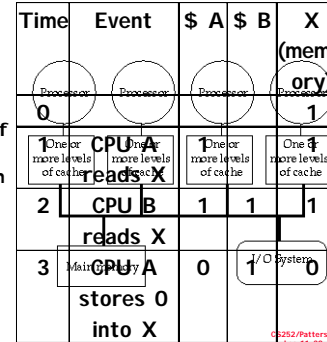
FracX = (80-1)/79.2 = 0.9975

- Only 0.25% sequential!

Small–Scale—Shared Memory

- Caches serve to:
  - Increase bandwidth versus bus/memory
  - Reduce latency of access
  - Valuable for both private data and shared data
- What about cache consistency?

Time	Event	$ A	$ B	X (memory)
0				1
1	CPU A reads X	1		1
2	CPU B reads X	1	1	1
3	CPU A stores 0 into X	0		0

What Does Coherency Mean?

- Informally:
  - "Any read must return the most recent write"
  - Too strict and too difficult to implement
- Better:
  - "Any write must eventually be seen by a read"
  - All writes are seen in proper order ("serialization")
- Two rules to ensure this:
  - "If P writes x and P1 reads it, P's write will be seen by P1 if the read and write are sufficiently far apart"
  - Writes to a single location are serialized: seen in one order
    » Latest write will be seen
    » Otherwise could see writes in illogical order (could see older value after a newer value)

Potential HW Cohernecy Solutions

- Snooping Solution (Snoopy Bus):
  - Send all requests for data to all processors
  - Processors snoop to see if they have a copy and respond accordingly
  - Requires broadcast, since caching information is at processors
  - Works well with bus (natural broadcast medium)
  - Dominates for small scale machines (most of the market)
- Directory-Based Schemes (discuss later)
  - Keep track of what is being shared in 1 centralized place (logically)
  - Distributed memory => distributed directory for scalability (avoids bottlenecks)
  - Send point-to-point requests to processors via network
  - Scales better than Snooping
  - Actually existed BEFORE Snooping-based schemes

Basic Snoopy Protocols

- Write Invalidate Protocol:
  - Multiple readers, single writer
  - Write to shared data: an invalidate is sent to all caches which snoop and invalidate any copies
  - Read Miss:
    » Write-through: memory is always up-to-date
    » Write-back: snoop in caches to find most recent copy
- Write Broadcast Protocol (typically write through):
  - Write to shared data: broadcast on bus, processors snoop, and update any copies
  - Read miss: memory is always up-to-date
- Write serialization: bus serializes requests!
  - Bus is single point of arbitration

Basic Snoopy Protocols

- Write Invalidate versus Broadcast:
  - Invalidate requires one transaction per write–run
  - Invalidate uses spatial locality: one transaction per block
  - Broadcast has lower latency between write and read

Snooping Cache Variations

Basic Protocol	Berkeley Protocol	Illinois Protocol	MESI Protocol
Exclusive Shared Invalid	Owned Exclusive Owned Shared Shared Invalid	Private Dirty Private Clean Shared Invalid	**M**odfied (private,!=Memory) e**X**clusive (private,=Memory) **S**hared (shared,=Memory) **I**nvalid

Owner can update via bus invalidate operation
Owner must write back when replaced in cache

If read sourced from memory, then Private Clean
if read sourced from other cache, then Shared
Can write in cache if held private clean or dirty

An Example Snoopy Protocol

- **Invalidation protocol, write-back cache**
- **Each block of memory is in one state:**
  - **Clean in all caches and up-to-date in memory (Shared)**
  - **OR Dirty in exactly one cache (Exclusive)**
  - **OR Not in any caches**
- **Each cache block is in one state (track these):**
  - **Shared** : block can be read
  - **OR Exclusive** : cache has only copy, its writeable, and dirty
  - **OR Invalid** : block contains no data
- **Read misses: cause all caches to snoop bus**
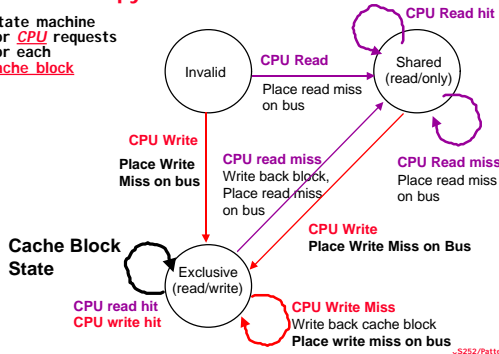- **Writes to clean line are treated as misses**

Snoopy-Cache State Machine-I

- State machine for _CPU_ requests for each cache_block

Snoopy-Cache State Machine-II

- State machine for _bus_ requests for each cache_block
- Appendix E? gives details of bus requests
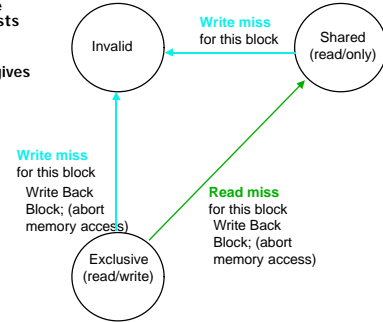
Snoopy-Cache State Machine-III

- State machine for _CPU_ requests for each cache_block and for _bus_ requests for each cache_block
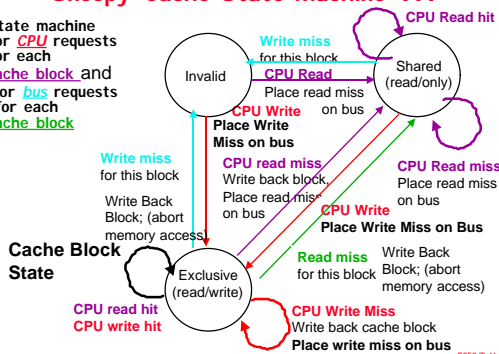
Example

step	P1			P2			Bus				Memory	
	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1 Write 10 to A1												
P1: Read A1												
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												

Assumes A1 and A2 map to same cache block,
initial cache state is invalid

Example

step	P1			P2			Bus				Memory	
	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1 Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1												
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												

Assumes A1 and A2 map to same cache block

Example

step	P1			P2			Bus				Memory	
	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1 Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												

Assumes A1 and A2 map to same cache block

Example

step	P1			P2			Bus				Memory	
	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1 Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1				Shar.	A1		RdMs	P2	A1			
	Shar.	A1	10				WrBk	P1	A1	10	A1	10
				Shar.	A1	10	RdDa	P2	A1	10	A1	10
P2: Write 20 to A1												
P2: Write 40 to A2												

Assumes A1 and A2 map to same cache block

Example

step	P1			P2			Bus				Memory	
	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1 Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1				Shar.	A1		RdMs	P2	A1			
	Shar.	A1	10				WrBk	P1	A1	10	A1	10
				Shar.	A1	10	RdDa	P2	A1	10	A1	10
P2: Write 20 to A1	Inv.			Excl.	A1	20	WrMs	P2	A1		A1	10
P2: Write 40 to A2												

Assumes A1 and A2 map to same cache block

Example

step	P1			P2			Bus				Memory	
	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1 Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1				Shar.	A1		RdMs	P2	A1			
	Shar.	A1	10				WrBk	P1	A1	10	A1	10
				Shar.	A1	10	RdDa	P2	A1	10	A1	10
P2: Write 20 to A1	Inv.			Excl.	A1	20	WrMs	P2	A1		A1	10
P2: Write 40 to A2							WrMs	P2	A2		A1	10
				Excl.	A2	40	WrBk	P2	A1	20	A1	20

Assumes A1 and A2 map to same cache block,
but A1 != A2

Implementation Complications

- **Write Races:**
  - **Cannot update cache until bus is obtained**
    - » Otherwise, another processor may get bus first, and then write the same cache block!
  - **Two step process:**
    - » Arbitrate for bus
    - » Place miss on bus and complete operation
  - **If miss occurs to block while waiting for bus, handle miss (invalidate may be needed) and then restart.**
  - **Split transaction bus:**
    - » Bus transaction is not atomic: can have multiple outstanding transactions for a block
    - » Multiple misses can interleave, allowing two caches to grab block in the Exclusive state
    - » Must track and prevent multiple misses for one block
- **Must support interventions and invalidations**

Implementing Snooping Caches

- Multiple processors must be on bus, access to both addresses and data
- Add a few new commands to perform coherency, in addition to read and write
- Processors continuously snoop on address bus
  - If address matches tag, either invalidate or update
- Since every bus transaction checks cache tags, could interfere with CPU just to check:
  - solution 1: duplicate set of tags for L1 caches just to allow checks in parallel with CPU
  - solution 2: L2 cache already duplicate, provided L2 obeys inclusion with L1 cache
    » block size, associativity of L2 affects L1

Implementing Snooping Caches

- Bus serializes writes, getting bus ensures no one else can perform memory operation
- On a miss in a write back cache, may have the desired copy and its dirty, so must reply
- Add extra state bit to cache to determine shared or not
- Add 4th state (MESI)

Fundamental Issues

- 3 Issues to characterize parallel machines
1) Naming
2) Synchronization
3) Performance: Latency and Bandwidth (covered earlier)

Fundamental Issue #1: Naming

- Naming: how to solve large problem fast
  - what data is shared
  - how it is addressed
  - what operations can access data
  - how processes refer to each other
- Choice of naming affects code produced by a compiler; via load where just remember address or keep track of processor number and local virtual address for msg. passing
- Choice of naming affects replication of data; via load in cache memory hierarchy or via SW replication and consistency

Fundamental Issue #1: Naming

- Global physical address space:
  any processor can generate, address and access it in a single operation
  - memory can be anywhere:
    virtual addr. translation handles it
- Global virtual address space: if the address space of each process can be configured to contain all shared data of the parallel program
- Segmented shared address space:
  locations are named
  <process number, address>
  uniformly for all processes of the parallel program

Fundamental Issue #2: Synchronization

- To cooperate, processes must coordinate
- Message passing is implicit coordination with transmission or arrival of data
- Shared address
  => additional operations to explicitly coordinate:
  e.g., write a flag, awaken a thread, interrupt a processor

Summary: Parallel Framework

�	**Programming Model**
▋greenbar	**Communication Abstraction**
▋orangebar	**Interconnection SW/OS**
▋redbar	**Interconnection HW**

- **Layers:**
  - Programming Model:
    - » **Multiprogramming** : lots of jobs, no communication
    - » **Shared address space**: communicate via memory
    - » **Message passing**: send and recieve messages
    - » **Data Parallel**: several agents operate on several data sets simultaneously and then exchange information globally and simultaneously (shared or message passing)
  - Communication Abstraction:
    - » **Shared address space**: e.g., load, store, atomic swap
    - » **Message passing**: e.g., send, recieve library calls
    - » Debate over this topic (ease of programming, scaling) => many hardware designs 1:1 programming model