
CS 267: Shared Memory Parallel Machines

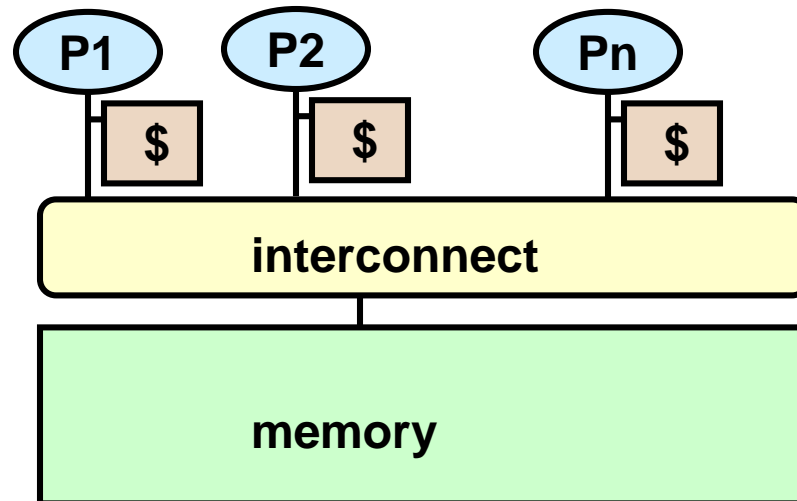
Katherine Yelick

yelick@cs.berkeley.edu

<http://www.cs.berkeley.edu/~yelick/cs267>

Basic Shared Memory Architecture

- Processors all connected to a large shared memory
- Local caches for each processor
- Cost: much cheaper to cache than main memory



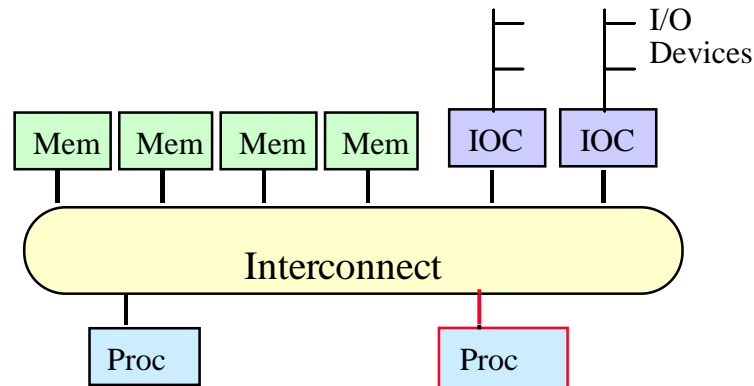
- Relatively easy to program, but hard to scale
- Now take a closer look at structure, costs, limits

Outline

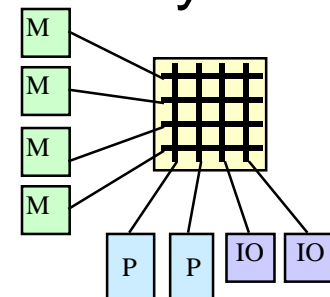
- Historical perspective
- Bus-based machines
 - Pentium SMP
 - IBM SP node
- Directory-based (CC-NUMA) machine
 - Origin 2000
- Global address space machines
 - Cray t3d and (sort of) t3e

60s Mainframe Multiprocessors

- Enhance memory capacity or I/O capabilities by adding memory modules or I/O devices

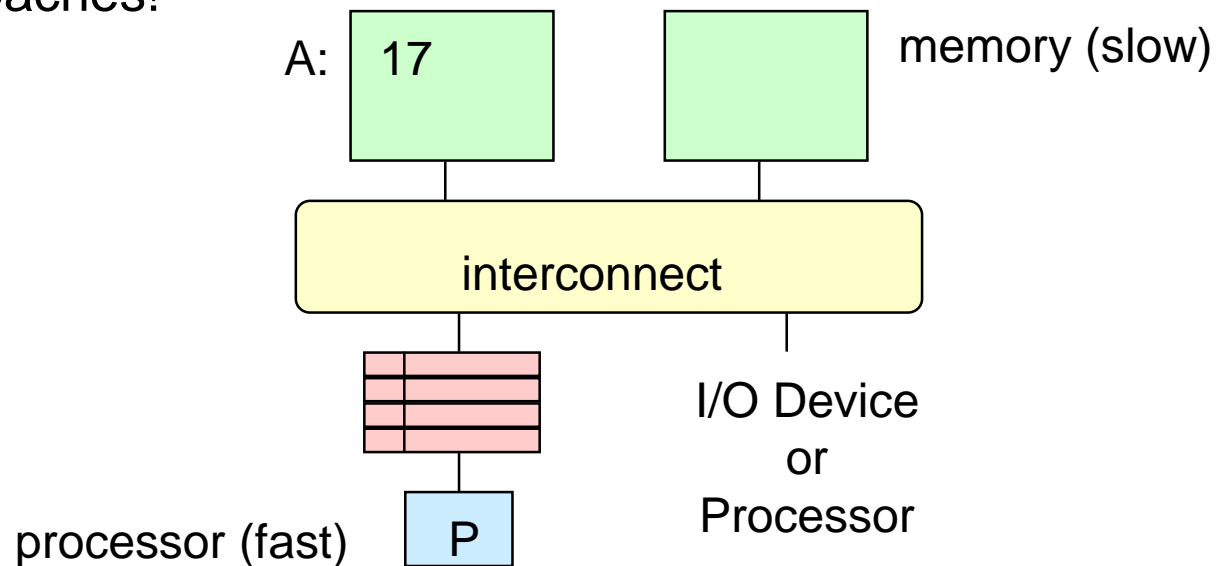


- How do you enhance processing capacity?
 - Add processors
- Already need an interconnect between slow memory banks and processor + I/O channels
 - cross-bar or multistage interconnection network



70s Breakthrough: Caches

- Memory system scaled by adding memory modules
 - Both bandwidth and capacity
- Memory was still a bottleneck
 - Enter... Caches!



- Cache does two things:
 - Reduces average access time (latency)
 - Reduces bandwidth requirements to memory

Technology Perspective

Capacity Speed

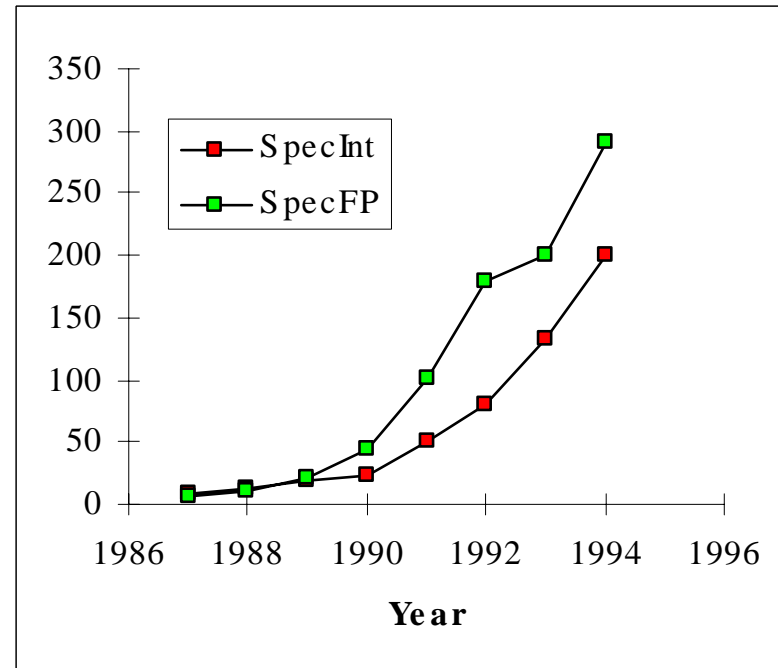
Logic: 2x in 3 years 2x in 3 years

DRAM: 4x in 3 years 1.4x in 10 years

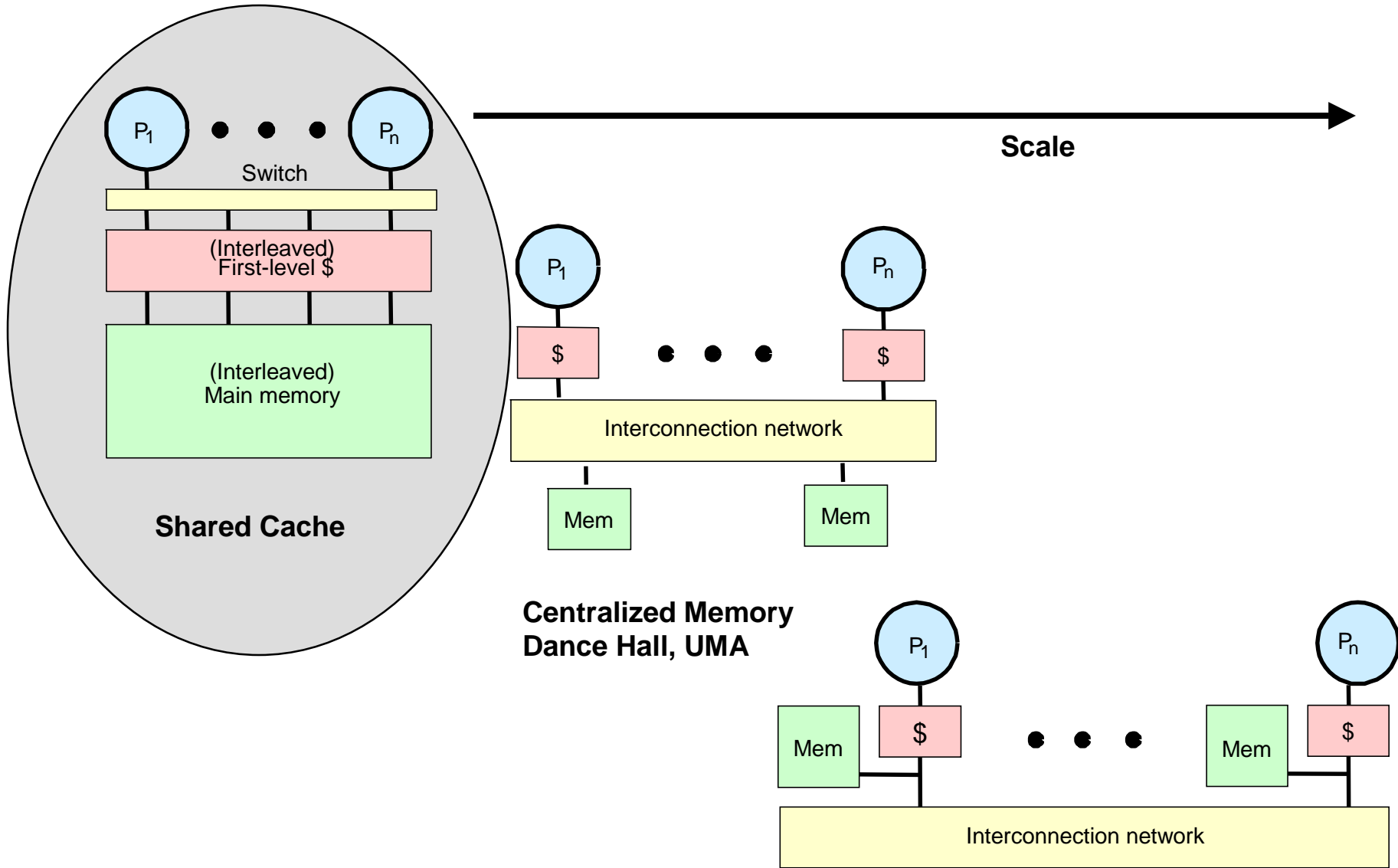
Disk: 2x in 3 years 1.4x in 10 years

Year	DRAM		Cycle Time
	Size	2:1!	
1980	64 Kb	2:1!	250 ns
1983	256 Kb		220 ns
1986	1 Mb		190 ns
1989	4 Mb		165 ns
1992	16 Mb		145 ns
1995	64 Mb		120 ns

1000:1! (indicated by a red arrow from 1980 to 1995)

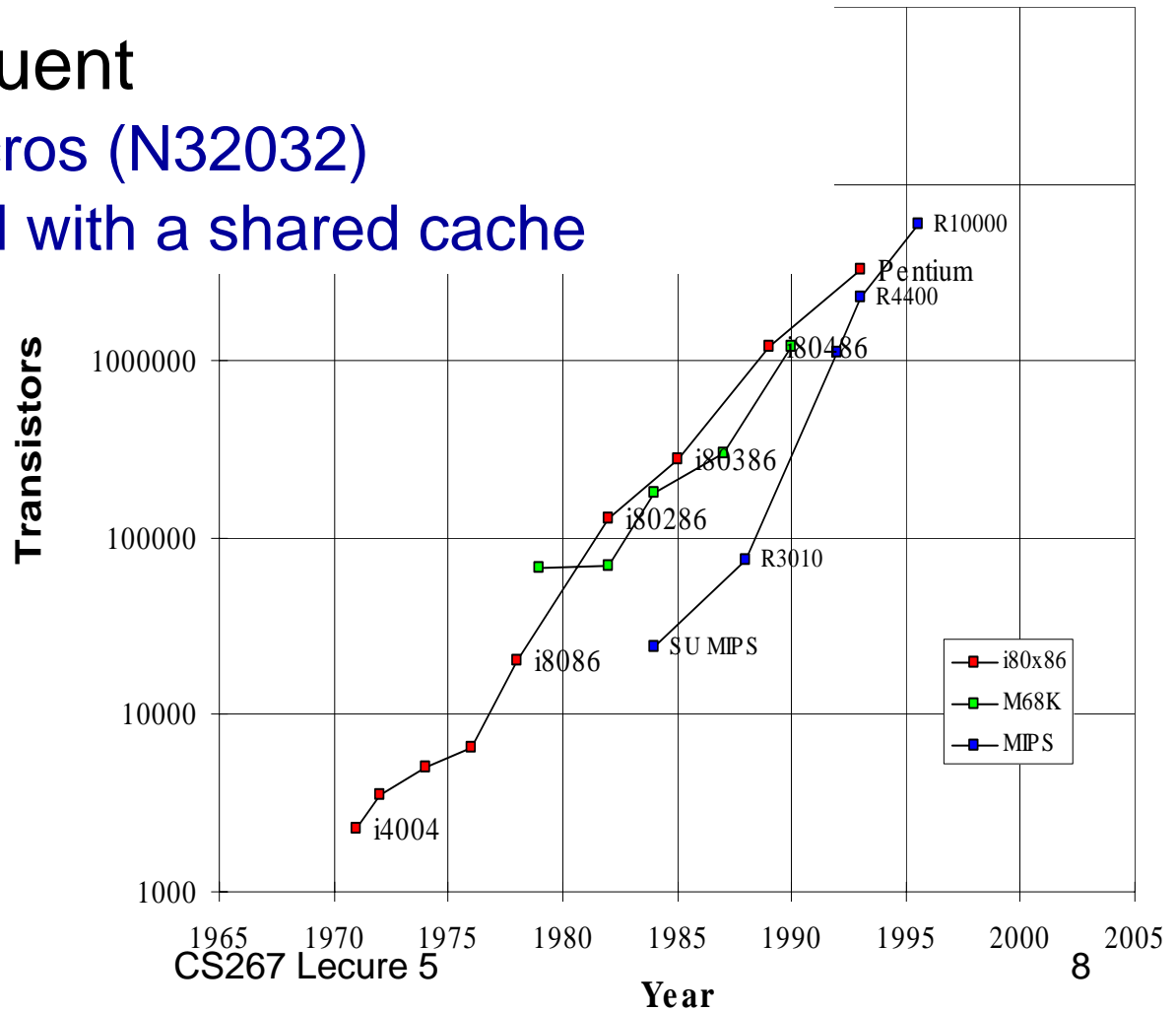
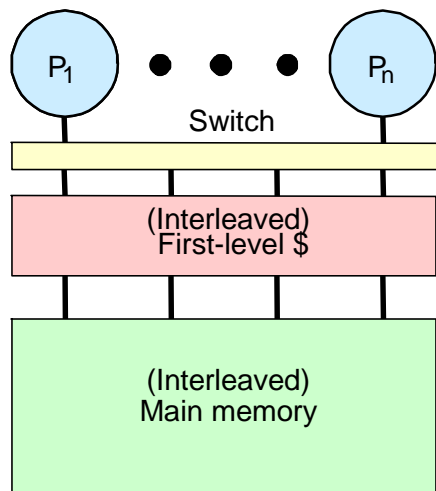


Approaches to Building Parallel Machines



80s Shared Memory: Shared Cache

- Alliant FX-8 (early 80s)
 - eight 68020s with x-bar to 512 KB interleaved cache
- Encore & Sequent
 - first 32-bit micros (N32032)
 - two to a board with a shared cache



Shared Cache: Advantages and Disadvantages

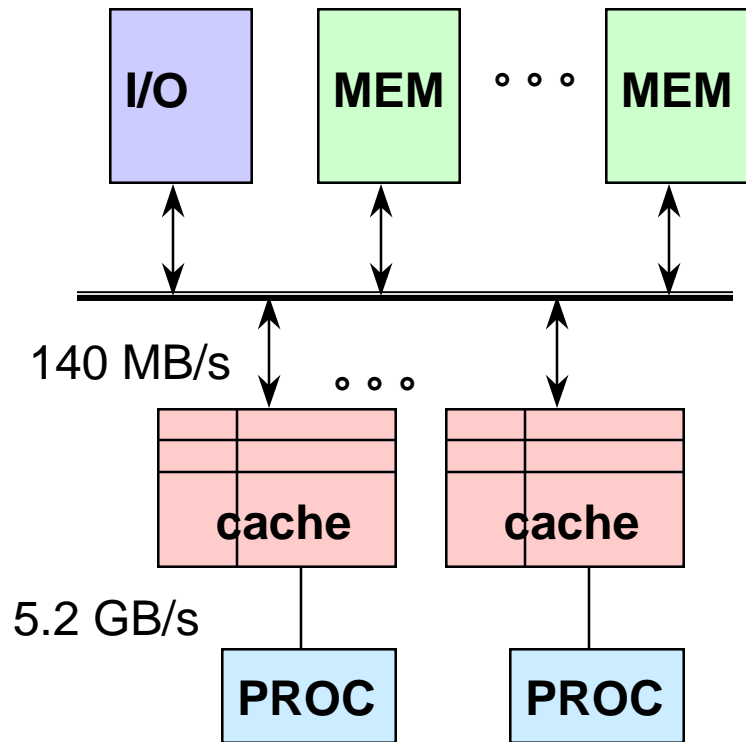
Advantages

- Cache placement identical to single cache
 - only one copy of any cached block
- Fine-grain sharing is possible
- Interference
 - One processor may prefetch data for another
 - Can share data within a line without moving line

Disadvantages

- Bandwidth limitation
- Interference
 - One processor may flush another processors data

Limits of Shared Cache Approach



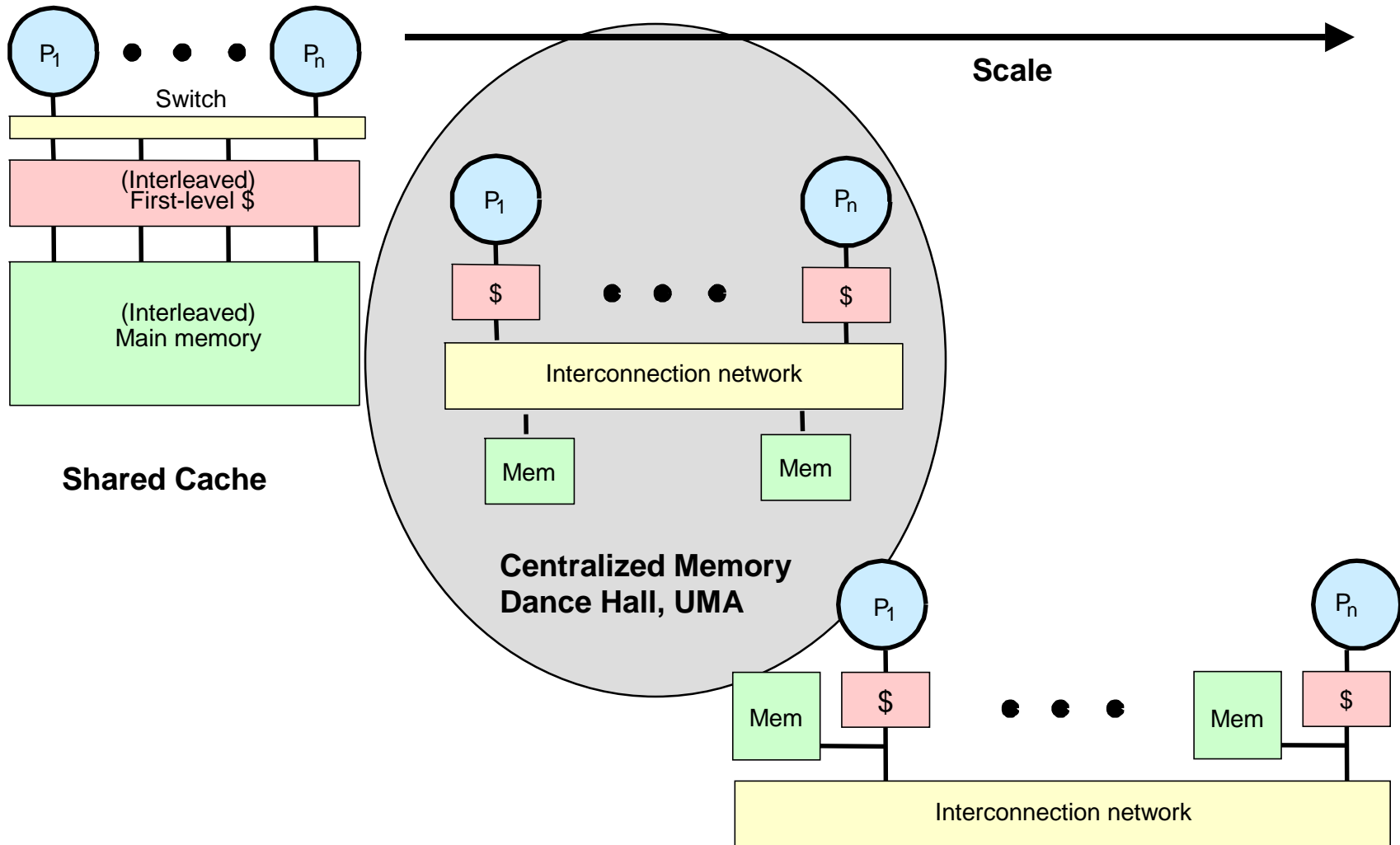
Assume:

- 1 GHz processor w/o cache
- => 4 GB/s inst BW per processor (32-bit)
- => 1.2 GB/s data BW at 30% load-store

Need 5.2 GB/s of bus bandwidth per processor!

- Typical bus bandwidth is closer to 1 GB/s

Approaches to Building Parallel Machines



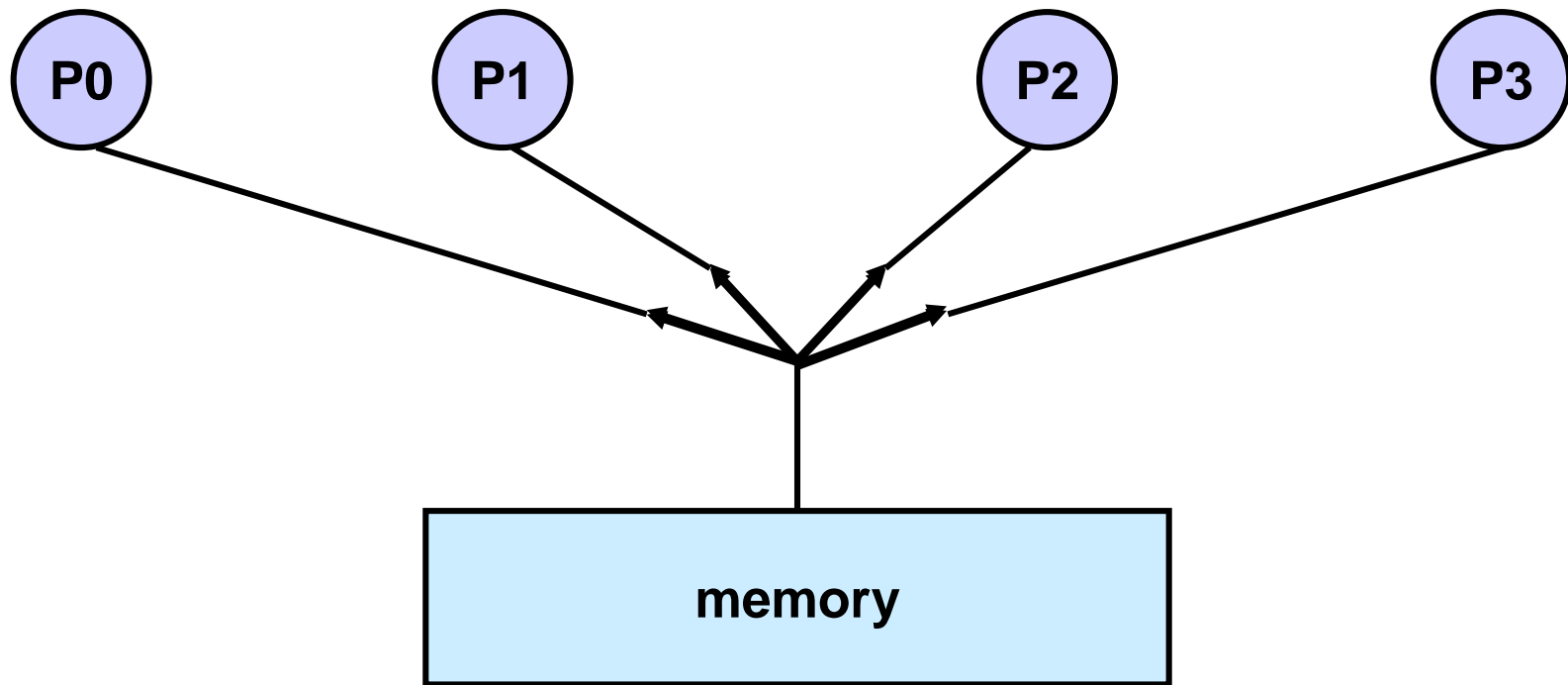
Intuitive Memory Model

- Reading an address should **return the last value written** to that address
- Easy in uniprocessors
 - **except for I/O**
- Cache coherence problem in MPs is more pervasive and more performance critical
- More formally, this is called **sequential consistency**:

“A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the 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.” [Lamport, 1979]

Sequential Consistency Intuition

- Sequential consistency says the machine *behaves as if* it does the following

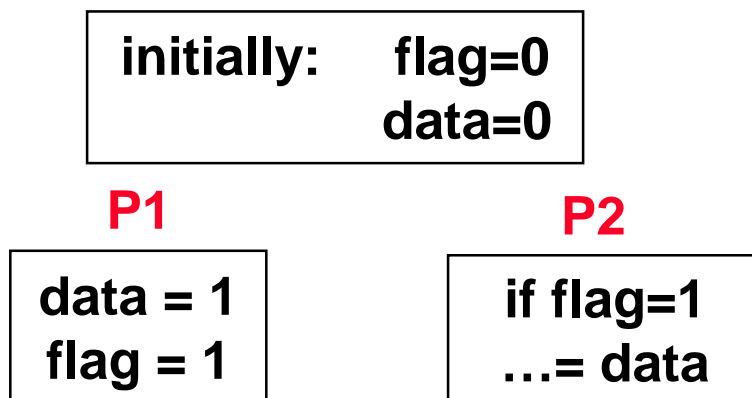


Memory Consistency Semantics

What does this imply about program behavior?

- No process ever sees “garbage” values, i.e., 1/2 of 2 values
- Processors always see values written by some processor
- The value seen is constrained by program order on all processors
 - Time always moves forward
- Example:
 - P1 writes data=1, then writes flag=1
 - P2 reads flag, then reads data

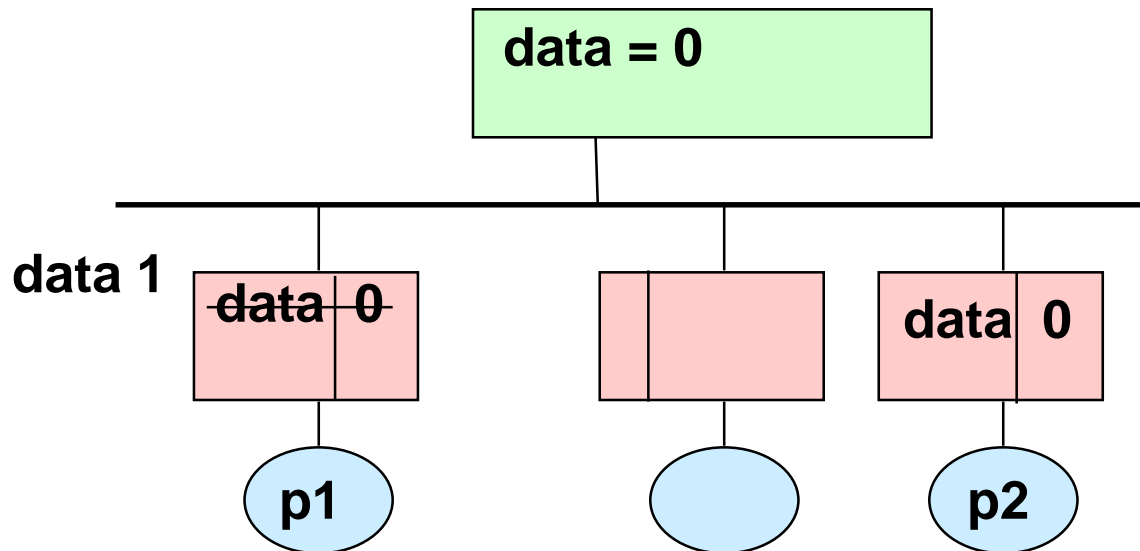
If P2 sees the new value of y, it must see the new value of x



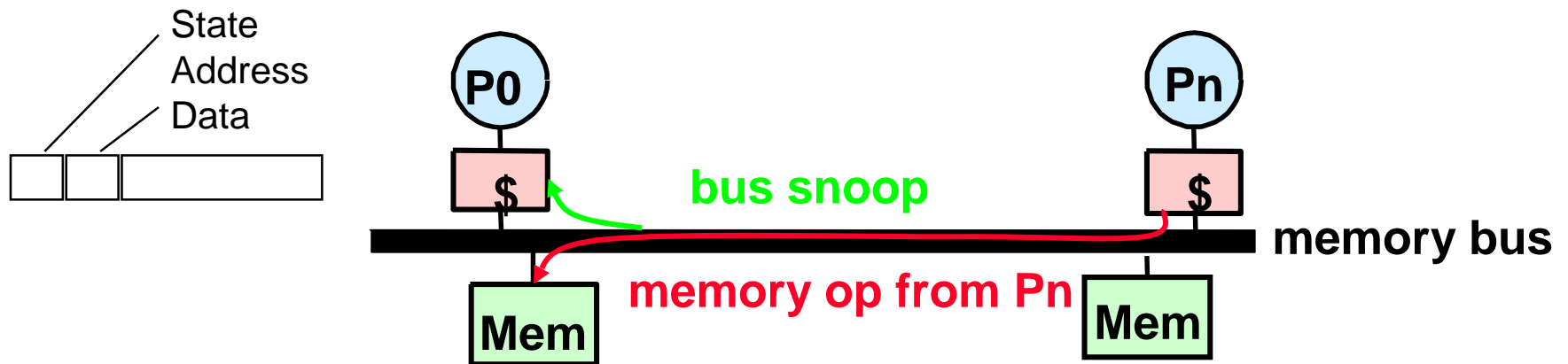
If P2 reads flag	Then P2 may read data
0	1
0	0
1	1

If Caches are Not Coherent

- p1 and p2 both have cached copies of data (as 0)
- p1 writes data=1
 - May “write through” to memory
- p2 reads data, but gets the “stale” cached copy
 - This may happen even if it read an updated value of another variable, flag, that came from memory



Snoopy Cache-Coherence Protocols

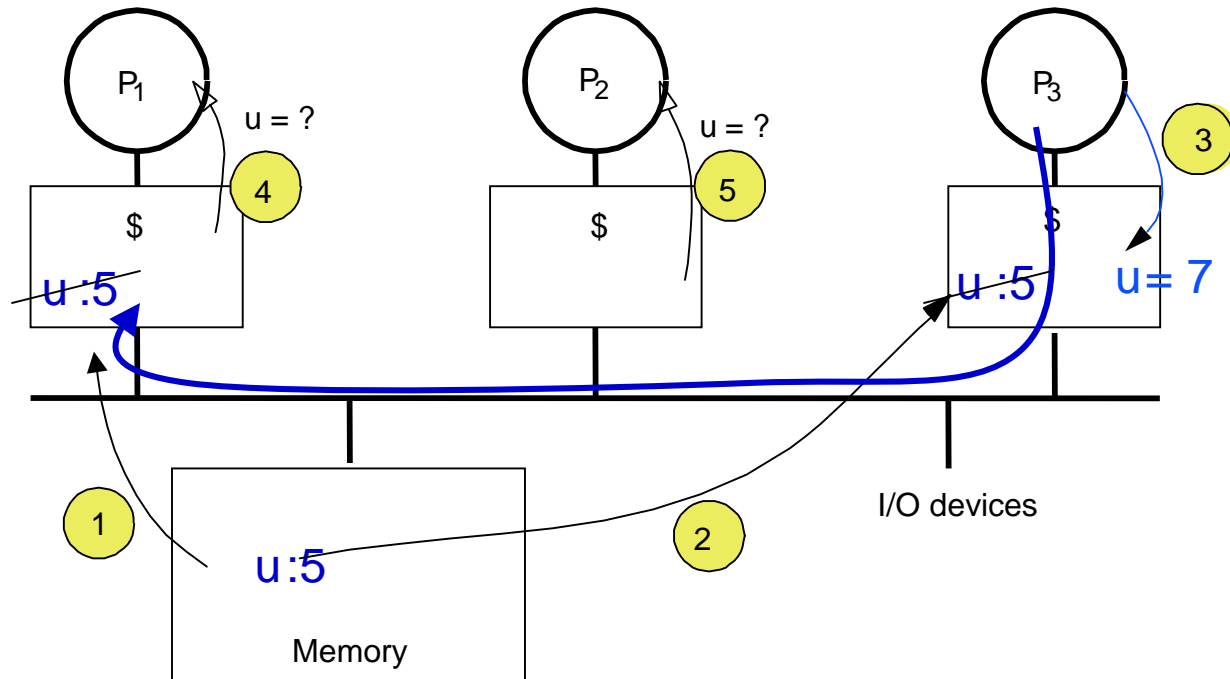


- Memory bus is a broadcast medium
- Caches contain information on which addresses they store
- Cache Controller “snoops” all transactions on the bus
 - A transaction is a relevant transaction if it involves a cache block currently contained in this cache
 - take action to ensure coherence
 - invalidate, update, or supply value
 - depends on state of the block and the protocol

Basic Choices in Cache Coherence

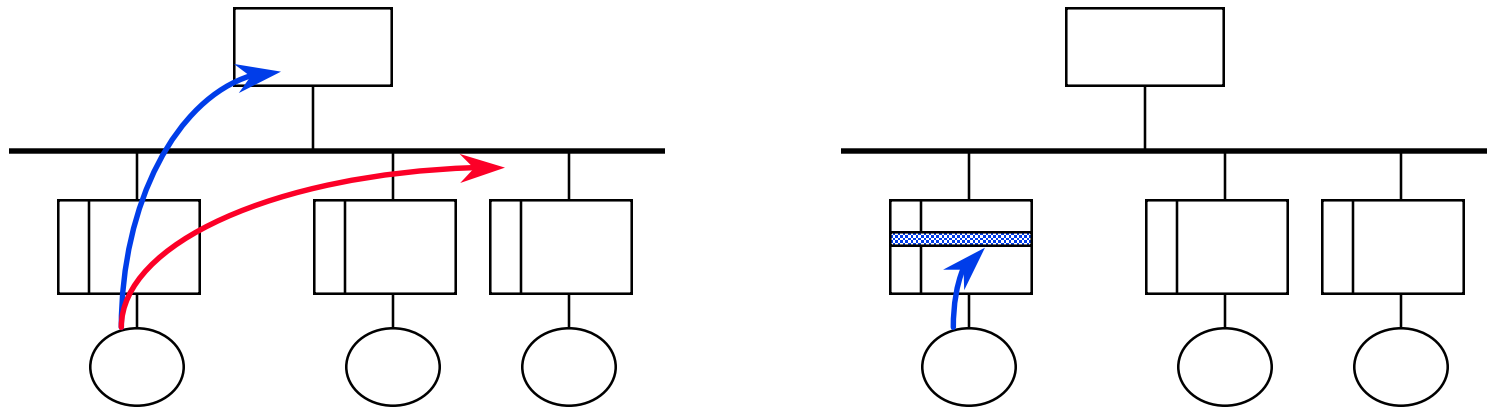
- Cache may keep information such as:
 - Valid/invalid
 - Dirty (inconsistent with memory)
 - Shared (in another caches)
- When a processor executes a write operation to shared data, basic design choices are:
 - **With respect to memory:**
 - Write through: do the write in memory as well as cache
 - Write back: wait and do the write later, when the item is flushed
 - **With respect to other cached copies**
 - Update: give all other processors the new value
 - Invalidate: all other processors remove from cache

Example: Write-thru Invalidate



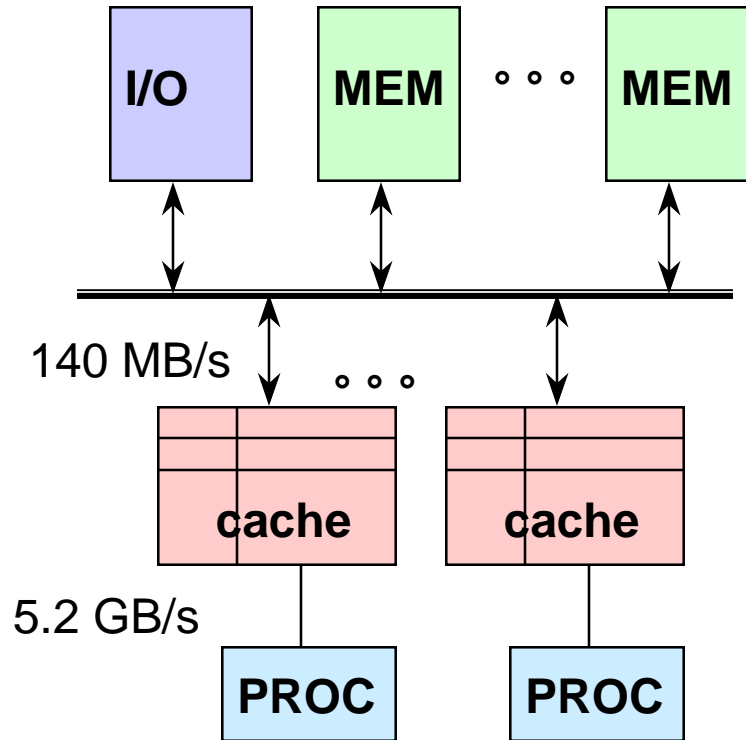
- Update and write-thru both use more memory bandwidth if there are writes to the same address
 - Update to the other caches
 - Write-thru to memory

Write-Back/Ownership Schemes



- When a single cache has ownership of a block, processor writes do not result in bus writes, thus conserving bandwidth.
 - reads by others cause it to return to “shared” state
- Most bus-based multiprocessors today use such schemes.
- Many variants of ownership-based protocols

Limits of Bus-Based Shared Memory



Assume:

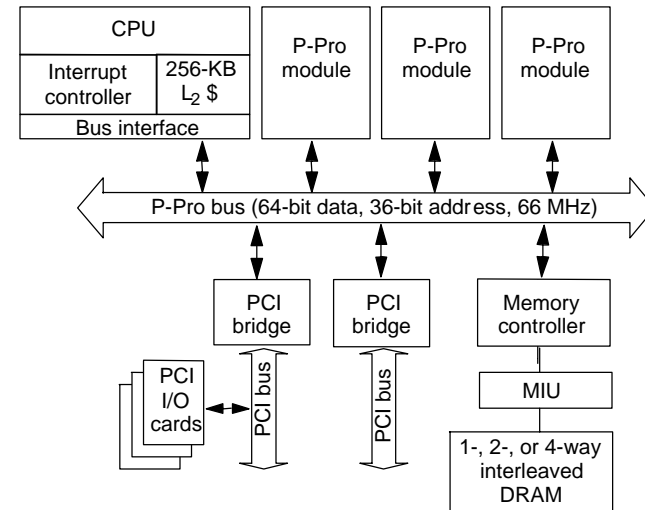
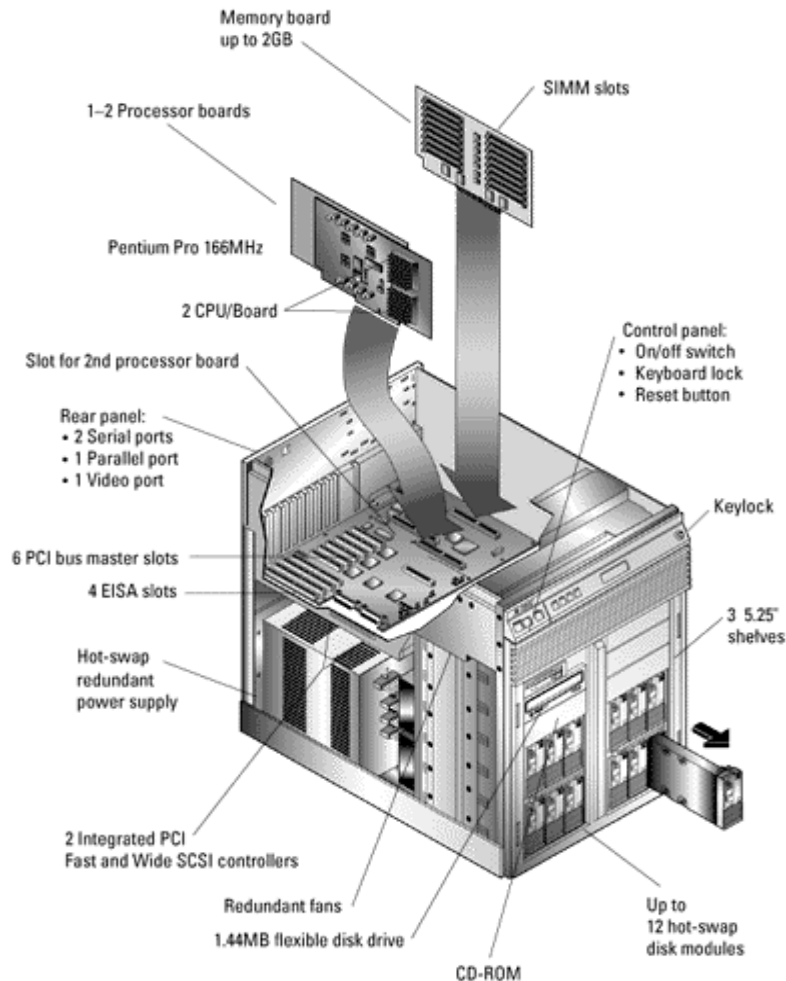
- 1 GHz processor w/o cache
- => 4 GB/s inst BW per processor (32-bit)
- => 1.2 GB/s data BW at 30% load-store

Suppose 98% inst hit rate and 95% data hit rate

- => 80 MB/s inst BW per processor
- => 60 MB/s data BW per processor
- => 140 MB/s combined BW

Assuming 1 GB/s bus bandwidth
∴ 8 processors will saturate bus

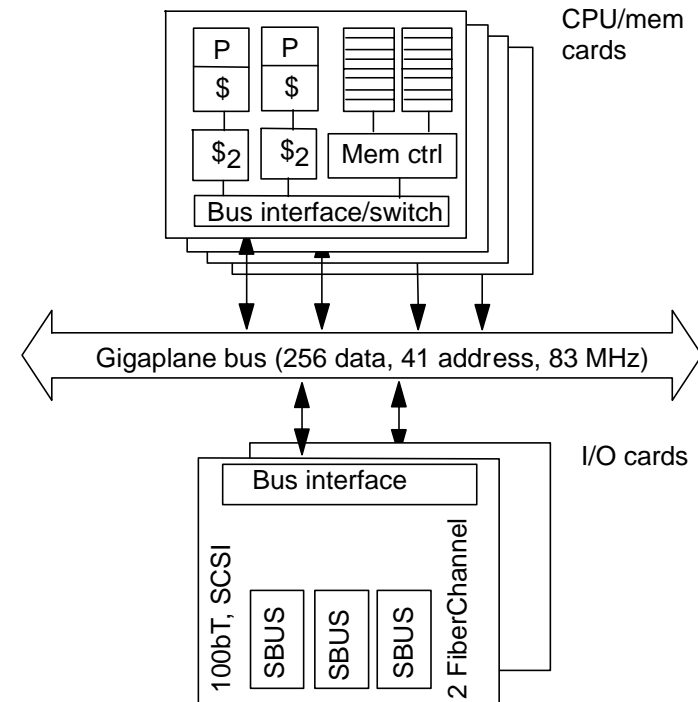
Engineering: Intel Pentium Pro Quad



SMP for the masses:

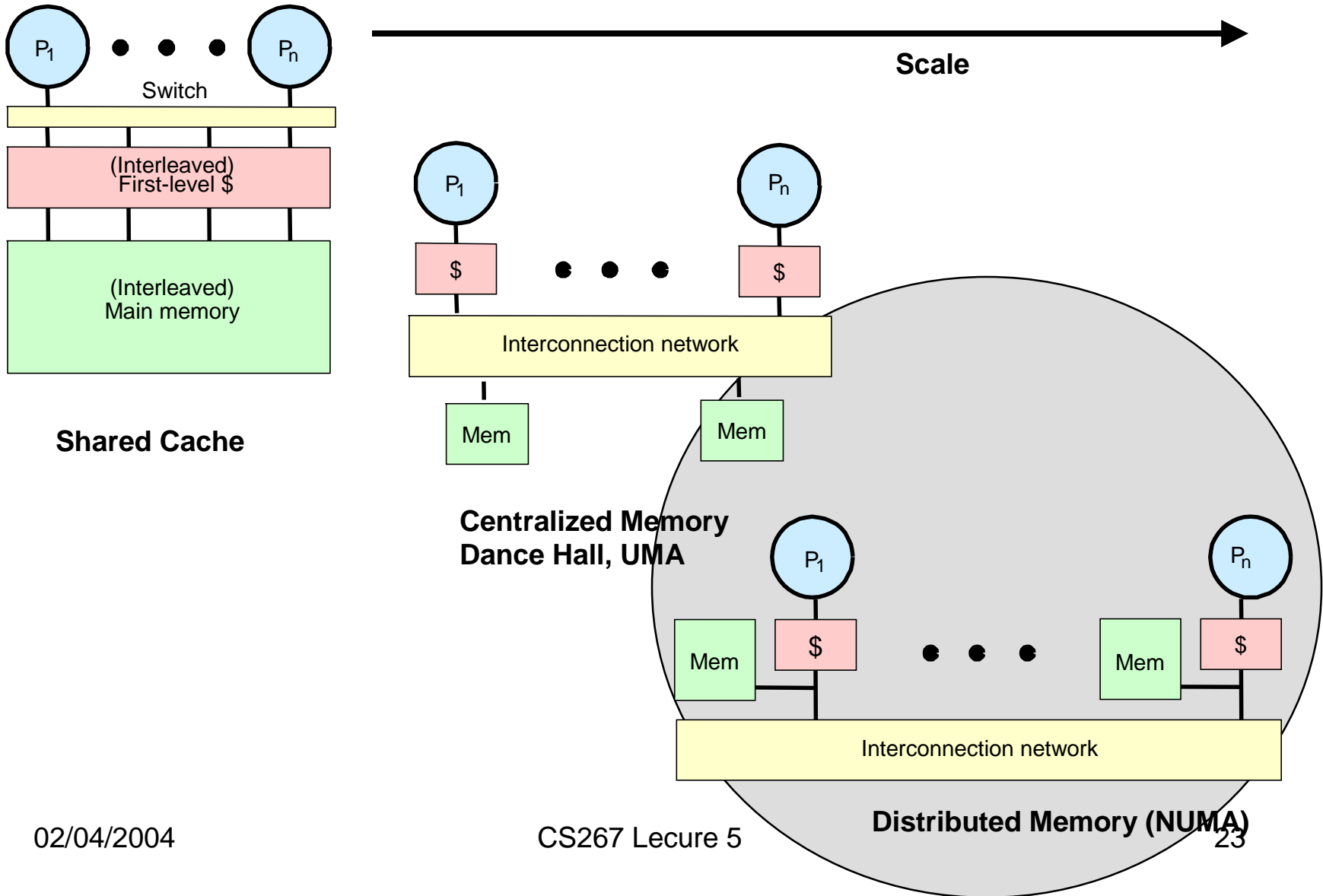
- All coherence and multiprocessing glue in processor module
- Highly integrated, targeted at high volume
- Low latency and bandwidth

Engineering: SUN Enterprise



- Proc + mem card - I/O card
 - 16 cards of either type
 - All memory accessed over bus, so symmetric
 - Higher bandwidth, higher latency bus

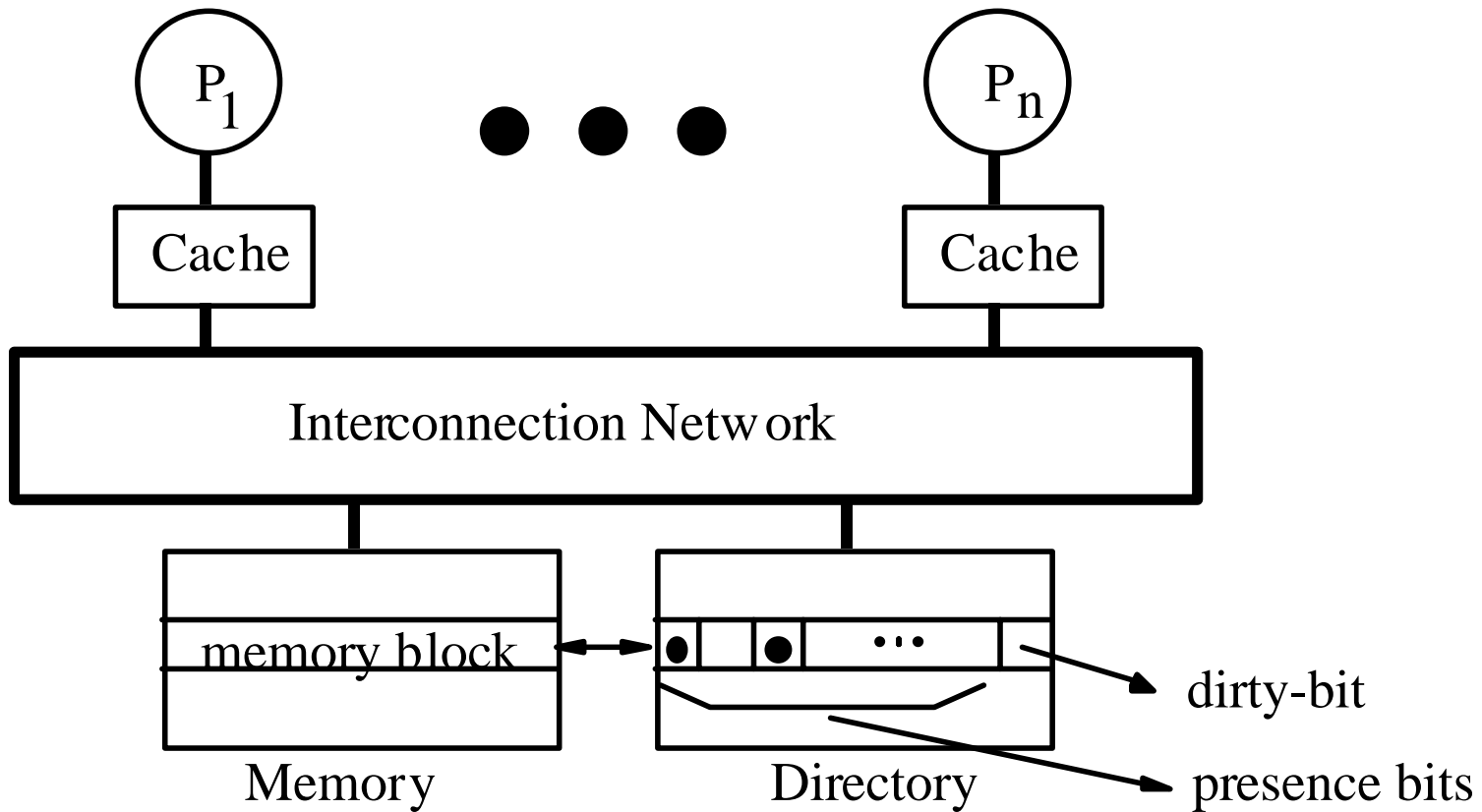
Approaches to Building Parallel Machines



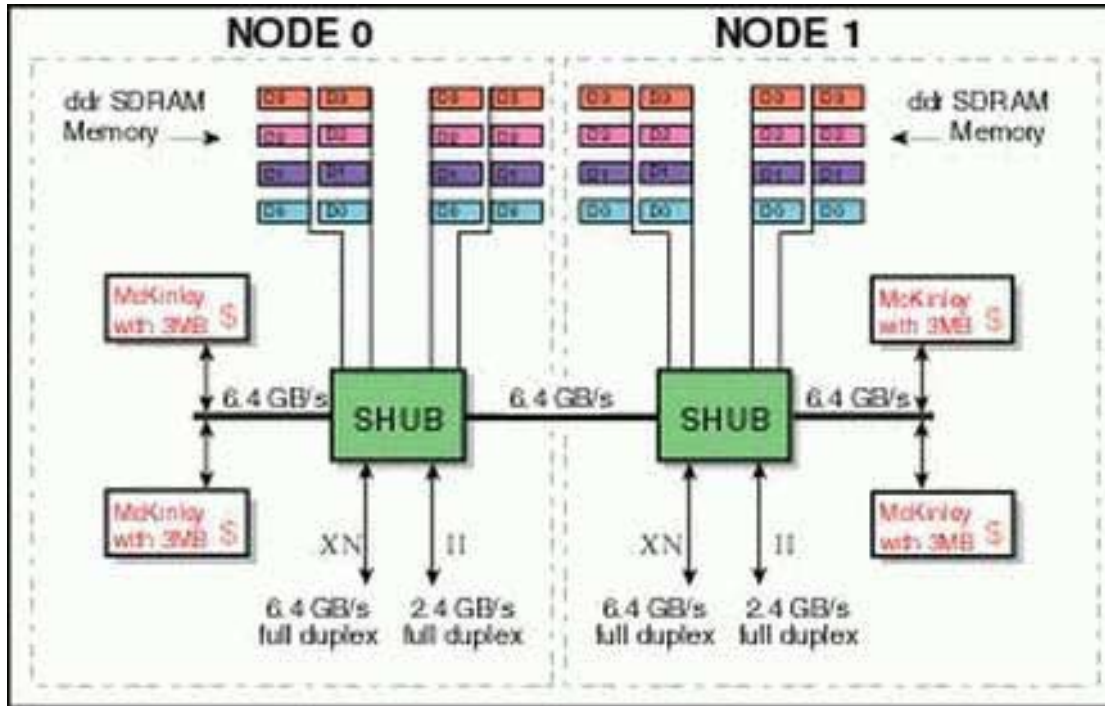
Directory-Based Cache-Coherence

02/04/2004

90 Scalable, Cache Coherent Multiprocessors



SGI Altix 3000



“Best of Show”
-LinuxWorld 2003

- A node contains up to 4 Itanium 2 processors and 32GB of memory
- Network is SGI’s NUMAlink, the NUMAflex interconnect technology.
- Uses a mixture of snoopy and directory-based coherence
- Up to 512 processors that are cache coherent (global address space is possible for larger machines)

Cache Coherence and Memory Consistency

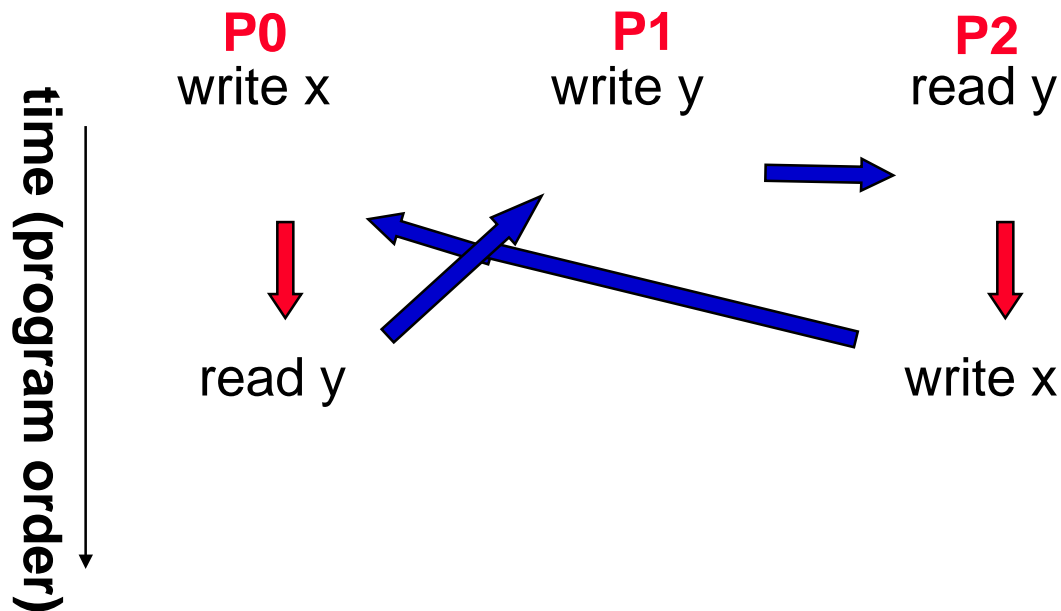
02/04/2004

Cache Coherence and Sequential Consistency

- There is a lot of hardware/work to ensure coherent caches
 - Never more than 1 version of data for a given address in caches
 - Data is always a value written by some processor
- But other hardware features may break sequential consistency (SC):
 - The compiler reorders/removes code (e.g., your spin lock)
 - The compiler allocates a register for flag on Processor 2 and spins on that register value without every completing
 - Write buffers (place to store writes while waiting to complete)
 - Processors may reorder writes to merge addresses (not FIFO)
 - Write X=1, Y=1, X=2 (second write to X may happen before Y's)
 - Prefetch instructions cause read reordering (read data before flag)
 - The network reorders the two write messages.
 - The write to flag is nearby, whereas data is far away.
 - Some of these can be prevented by declaring variables volatile

Violations of Sequential Consistency

- Flag/data program is one example that relies on SC
- Given coherent memory, all violations of SC based on reordering on independent operations are figure 8s
 - See paper by Shasha and Snir for more details
- Operations can be linearized (move forward time) if SC



Sufficient Conditions for Sequential Consistency

- Processors issues memory operations in program order
- Processor waits for store to complete before issuing any more memory operations
 - E.g., wait for write-through and invalidations
- Processor waits for load to complete before issuing any more memory operations
 - E.g., data in another cache may have to be marked as shared rather than exclusive
- A load must also wait for the store that produced the value to complete
 - E.g., if data is in cache and update event changes value, all other caches must also have processed that update
- There are much more aggressive ways of implementing SC, but most current commercial SMPs give up

Classification for Relaxed Models

- Optimizations can generally be categorized by
 - Program order relaxation:
 - Write \rightarrow Read
 - Write \rightarrow Write
 - Read \rightarrow Read, Write
 - Read others' write early
 - Read own write early
- All models provide safety net, e.g.,
 - A write fence instruction waits for writes to complete
 - A read fence prevents prefetches from moving before this point
 - Prefetches may be synchronized automatically on use
- All models maintain uniprocessor data and control dependences, write serialization
 - Memory models differ on orders to two different locations

Some Current System-Centric Models

Relaxation :	W →R Order	W →W Order	R →RW Order	Read Others' Write Early	Read Own Write Early	Safety Net
IBM 370	✓					serialization instructions
TSO	✓				✓	RMW
PC	✓			✓	✓	RMW
PSO	✓	✓			✓	RMW, STBAR
WO	✓	✓	✓		✓	synchronization
RCsc	✓	✓	✓		✓	release, acquire, nsync, RMW
RCpc	✓	✓	✓	✓	✓	release, acquire, nsync, RMW
Alpha	✓	✓	✓		✓	MB, WMB
RMO	✓	✓	✓		✓	various MEMBARs
PowerPC	✓	✓	✓	✓	✓	SYNC

Data-Race-Free-0: Some Definitions

- (Consider SC executions \Rightarrow global total order)
- Two conflicting operations race if
 - From different processors
 - Execute one after another (consecutively)

P1

Write, A, 23

Write, B, 37

Write, Flag, 1

P2

Read, Flag, 0

Read, Flag, 1

Read, B, ____

Read, A, ____

- Races usually labeled as **synchronization**, others **data**
- Can optimize operations that *never race*

Programming with Weak Memory Models

- Some rules for programming with these models
 - Avoid race conditions
 - Use system-provided synchronization primitives
 - If you have race conditions on variables, make them volatile
 - At the assembly level, may use fences (or analog) directly
- The high level language support for these differs
 - Built-in synchronization primitives normally include the necessary fence operations
 - lock : includes a read fence
 - unlock: includes a write fence
 - So all memory operations happen in the critical region

Cache-Coherent Shared Memory and Performance

02/04/2004

Caches and Scientific Computing

- Caches tend to perform worst on demanding applications that operate on large data sets
 - transaction processing
 - operating systems
 - sparse matrices
- Modern scientific codes use tiling/blocking to become cache friendly
 - easier for dense codes than for sparse
 - tiling and parallelism are similar transformations

Sharing: A Performance Problem

- True sharing
 - Frequent writes to a variable can create a bottleneck
 - OK for read-only or infrequently written data
 - Technique: make copies of the value, one per processor, if this is possible in the algorithm
 - Example problem: the data structure that stores the freelist/heap for malloc/free
- False sharing
 - Cache block may also introduce artifacts
 - Two distinct variables in the same cache block
 - Technique: allocate data used by each processor contiguously, or at least avoid interleaving
 - Example problem: an array of ints, one written frequently by each processor

What to Take Away?

- Programming shared memory machines
 - May allocate data in large shared region without too many worries about where
 - Memory hierarchy is critical to performance
 - Even more so than on uniproc, due to coherence traffic
 - For performance tuning, watch sharing (both true and false)
- Semantics
 - Need to lock access to shared variable for read-modify-write
 - Sequential consistency is the natural semantics
 - Architects worked hard to make this work
 - Caches are coherent with buses or directories
 - No caching of remote data on shared address space machines
 - But compiler and processor may still get in the way
 - Non-blocking writes, read prefetching, code motion...
 - Avoid races or use machine-specific fences carefully