Lecture 19/20: Shared Memory SMP and Cache Coherence

Adapted from UCB CS252 S01

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, ...
- "Pull" of opportunity of scalable performance, not the "push" of uniprocessor performance plateau?

What Level of 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
  - High-end desktop dual processor PC soon?

Why Multiprocessors?

1. Microprocessors as the fastest CPUs
   - Collecting several much easier than redesigning 1
2. Complexity of current microprocessors
   - Do we have enough ideas to sustain 1.5X/yr?
   - Can we deliver such complexity on schedule?
3. Slow (but steady) improvement in parallel software (scientific apps, databases, OS)
4. 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 Introduction

- 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)
- 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%)

Where is Supercomputing heading?

- 1997, 500 fastest machines in the world: 319 MPPs, 73 bus-based shared memory (SMP), 106 parallel vector processors (PVP)
- 2000, 381 of 500 fastest: 144 IBM SP (~cluster), 121 Sun (bus SMP), 62 SGI (NUMA SMP), 54 Cray (NUMA SMP)

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

http://www.top500.org/

Popular Flynn Categories for Parallel Computers

- SISD (Single Instruction Single Data)
  - Uniprocessors
- MISD (Multiple Instruction Single Data)
  - multiple processors on a single data stream
- SIMD (Single Instruction Multiple Data)
  - Early Examples: Illiac-IV, CM-2
  - 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

1. Centralized shared memory ("Uniform Memory Access" time or "Shared Memory Processor")
2. Decentralized memory (memory module with CPU)
   - Shared Memory with "Non Uniform Memory Access" time (NUMA)
   - Message passing "multicomputer" with separate address space per processor

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:

- Programming Model:
  - Multiprogramming : lots of jobs, no communication
  - Shared address space: communicate via memory
  - Message passing: send and recieve messages
  - Data Parallel: one operation, multiple data sets
- Communication Abstraction:
  - Shared address space: e.g., load, store, etc => multiprocessors
  - Message passing: e.g., send, recieve library calls
  - Debate over this topic (ease of programming, scaling)

May mix shared address space and message passing at different layers

Shared Address/Memory Processor Model

- 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-Memory Programming Examples

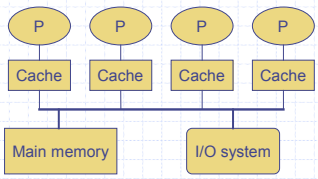
```
struct alloc_t { int first; int last } alloc[MAX_THR];
pthread_t tid[MAX_THR];
main()
{
  …
  for (int i=0; i<num_thr; i++) {
    alloc[i].first = i*N/num_thr;
    alloc[i].last = (i!=num_thr)?((i+1)*(N/num_thr)-1):N;
    pthread_create(&tid[i]/*thread id pointer*/,
      NLLL/*detach method*/, dmm_func/*thread function*/,
      (void *)&alloc[i]/*parameters*/);
  }
  for (i=0; i<num_thr; i++) {
    pthread_join(tid[i]/*thread id*/, NULL/*return value*/);
  }
}

dmm_func(struct alloc_t *alloc) {
   for (int i=alloc->first; i<alloc->last; i++)
     for (int k=0; k<N; k++)
       for (int j=0; j<N; j++)
         Z[i][j] += X[i][k]*Y[k][j];
}
```


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
  - ALL threads of a process share a process address space
  - Example: Pthread
- Writes to shared address space by one thread are visible to reads of other threads

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



Advantages of Shared-memory Model

- Ease of programming when communication patterns are complex or vary dynamically during execution
- Lower communication overhead, good utilization of communication bandwidth for small data items, no expensive I/O operations
- Hardware-controlled caching to reduce remote commutations when remote data is cached

Message Passing Model

- Whole computers (CPU, memory, I/O devices), explicit send/receive as explicit I/O operations
- Send specifies local buffer + receiving process on remote computer
- Receive specifies sending process on remote computer + local buffer to place data
- Send+receive => memory-memory copy, where each each supplies local address

Advantages of Message-Passing Communication

- The hardware can be much simpler and is usually standard
- Explicit communication => simpler to understand, help make effort to reduce communication cost
- Synchronization is naturally associated with sending/receiving messages
- Easier to use sender-initiated communication, which may have some advantages in performance

Important, but will not be discussed in details

Three Types of Parallel Applications

- Commercial Workload
  - TPC-C, TPC-D, Altavista (Web search engine)
- Multiprogramming and OS Workload
- Scientific/Technical Applications
  - FFT Kernel: 1D complex number FFT
  - LU Kernel: dense matrix factorization
  - Barnes App: Barnes-Hut n-body algorithm, galaxy evolution
  - Ocean App: Gauss-Seidel multigrid technique to solve a set of elliptical partial differential eq.s'

Amdahl's Law and Parallel Computers

- Amdahl's Law: speedup is limited by the fraction of the portions that can be parallelized
- Speedup ≤ 1 / (1-f), where f is the fraction of sequential computation
- How large can be *f* if we want 80X speedup from 100 processors?

  1 / (f+(1-f)/100) = 80

  f = 0.25% !

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)
- Cache coherency in multiprocessors: How does a processor know changes in the caches of other processors? How do other processors know changes in this cache?

Potential HW Coherency 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 with 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

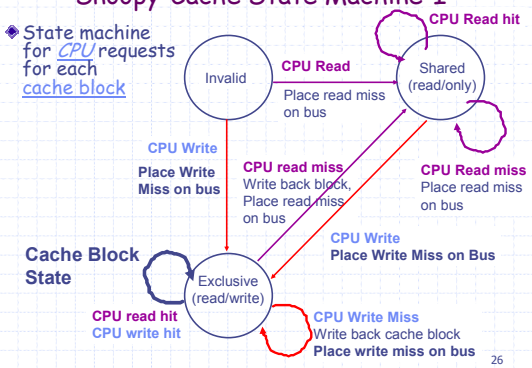
- Write Invalidate versus Broadcast:
  - Invalidate requires one transaction per write-run
  - Invalidate uses spatial locality: one transaction per block
  - Broadcast has lower latency between write and read

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

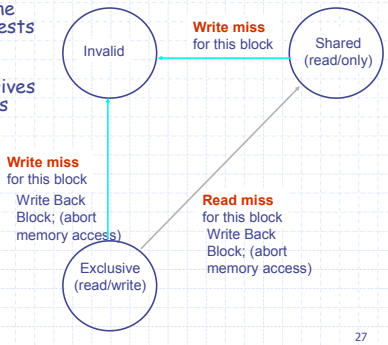
Snoopy-Cache State Machine-I

- State machine for *CPU* requests for each cache block



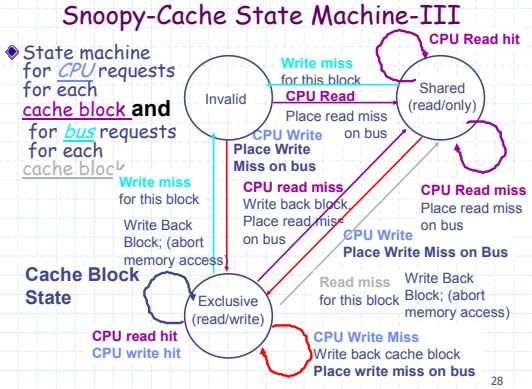
Snoopy-Cache State Machine-II

- State machine for *bus* requests for each cache block
- Appendix I 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 State	P1 Addr	P1 Value	P2 State	P2 Addr	P2 Value	Bus Action	Bus Proc.	Bus Addr	Bus Value	Memory Addr	Memory 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 State	P1 Addr	P1 Value	P2 State	P2 Addr	P2 Value	Bus Action	Bus Proc.	Bus Addr	Bus Value	Memory Addr	Memory 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 State	P1 Addr	P1 Value	P2 State	P2 Addr	P2 Value	Bus Action	Bus Proc.	Bus Addr	Bus Value	Memory Addr	Memory 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 State	P1 Addr	P1 Value	P2 State	P2 Addr	P2 Value	Bus Action	Bus Proc.	Bus Addr	Bus Value	Memory Addr	Memory 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 State	P1 Addr	P1 Value	P2 State	P2 Addr	P2 Value	Bus Action	Bus Proc.	Bus Addr	Bus Value	Memory Addr	Memory 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 State	P1 Addr	P1 Value	P2 State	P2 Addr	P2 Value	Bus Action	Bus Proc.	Bus Addr	Bus Value	Memory Addr	Memory 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

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)
  - **Modfied (private,!=Memory)**
  - **eXclusive (private,=Memory)**
  - **Shared (shared,=Memory)**
  - **Invalid**

37

MESI Highlights

From local processor *P's* viewpoint, for each cache block

- **M**odified: Only *P* has a copy and the copy has been modifed; must respond to any read/write request
- **E**xclusive-clean: Only *P* has a copy and the copy is clear; no need to inform others about my changes
- **S**hared: Some other machines else *may* have copy; have to inform others about *P*s changes
- **I**nvalid: The block has been invalidated (possibly on the request of someone else)

38

MESI Highlights

Actions:

- Have read misses on a block: send read request onto bus
- Have write misses on a block: send write request onto bus
- Receive bus read request: transit the block to shared state
- Receive bus write request: transit the block to invalid state
- Must write back data when transiting from modified state

39