Announcements

• NCSA forms: User agreement sheets

• Minor modifications to the course home page
  – lecture topics, order of tutorials

• Lectures 9 (Nov. 5) and 10 (Nov. 12) need to be rescheduled
  – Tuesdays, 7:30 - 9:00pm

• Any questions about Homework 2?

Outline

• Last lecture
  – models of parallel computation: PRAM, LogP
  – programming as successive refinement
    • architecture-independent partitioning:
      – load balance, reduce inherent communication, reduce extra work
    • architecture-dependent orchestration
      – artifactual communication costs
      – common issues: naming, synchronization, latency, bandwidth

• This lecture: Small-scale Shared Memory Machines
  – communication: coherence and consistency
    • snooping-based coherence protocols
  – synchronization
  – case studies: SGI Challenge, Sun Enterprise
  – tutorial: programming using MPI

[ Culler/Singh/Gupta: Chapter 4, Almasi/Gottlieb: Chapter 10 (Sections 10.3.1, 10.3.2) ]
Shared Memory Multiprocessors

• Symmetric multiprocessors (SMPs)
  – uniform access to all of main memory from any processor

• Dominates the server market
  – building blocks for larger systems
  – arriving to desktop

• Attractive for both parallel programs and throughput servers
  – fine-grain resource sharing
  – automatic data movement and coherent replication in caches

  ➔ Uniform access via loads and stores
  – private caches reduce access latency, bandwidth demands on bus
  – however, introduce the cache coherence problem
    • values in different caches need to be kept consistent

The Cache Coherence Problem

• Processors see stale values
  – with write-back caches, value written back to memory depends on which
    cache flushes or writes back value (and when)
  – clearly not a desirable situation!

So What Should Happen?

• Intuition for a coherent memory system
  reading a location should return latest value written (by any process)

• What does latest mean?
  – several alternatives even on uniprocessors
    • source program order, program issue order, order of completion, etc.
  – how to make sense of order among multiple processes?
    ➔ must define a meaningful semantics

• Note that cache coherence is also a problem in uniprocessors
  – interaction between caches and I/O devices
    • infrequent software solutions work well
      – uncachable memory, flush pages, route I/O through caches
    – however, the problem is performance-critical in multiprocessors
      • needs to be treated as a basic hardware design issue

Some Basic Definitions

• Uniprocessors:
  – memory operation: a single read, write or read-modify-write access
    • assumed to execute atomically with respect to each other
  – issue: a memory operation issues when it leaves the processor’s internal
    environment and is presented to the memory system (cache, buffer, etc.)
  – perform: operation appears to have taken place, as far as the
    processor can tell from other memory operations it issues
    • a write performs w.r.t. the processor when a subsequent read by the
      processor returns the value of that write or a later write
    • a read performs w.r.t the processor when subsequent writes issued by the
      processor cannot affect the value returned by the read

• Multiprocessors
  – all the above stay the same, but replace “the” by “a” processor
  – complete: perform with respect to all processors
  – still need to make sense of order in operations from different processes!
Order Among Multiple Processes: Intuition

- Assume a single shared memory, no caches
  - every read/write to a location accesses the same physical location
  - operation completes when it does so
  - so, memory imposes a serial or total order on operations to the location
  - operations to the location from a given processor are in program order
  - the order of operations to the location from different processors is some interleaving that preserves the individual program orders

- With caches
  - "latest" = most recent in a serial order that maintains these properties
  - for the serial order to be consistent, all processors must see writes to the location in the same order (if they bother to look)

- Note that we do not need to construct the total order
  - the program should just behave as if some serial order is enforced

Formal Definition of Coherence

A memory system is coherent if the results of any execution of a program are such that for each location, it is possible to construct a hypothetical serial order of all operations to the location that is consistent with the results of the execution and in which:

- operations issued by any particular process occur in the order issued by that process, and
- the value returned by a read is the value written by the last write to that location in the serial order

- Two necessary features:
  - write propagation: value written must become visible to others
  - write serialization: writes to a location seen in the same order by all

Cache Coherence Using a Bus

Two fundamentals of uniprocessor systems

- Bus transactions
  - three phases: arbitration, command/address, data transfer
  - all devices observe addresses, one is responsible for providing data

- Cache state transitions:
  - every block is a finite state machine
  - two states in write-through, write no-allocate caches: valid, invalid
  - write-back caches have one more state: modified ("dirty")

- Multiprocessors extend both these somewhat to implement coherence
  - "snoop" on bus events and take action
  - cache controller receives inputs from two sides: processor and bus
    - actions: update state, respond with data, generate new bus transactions
    - protocol implemented by cooperating state machines

Coherence with Write-through Caches

- Snoop on write transactions and invalidate/update cache
  - memory is always up-to-date (write-through)
  - invalidation causes next read to miss and fetch new value from memory (write propagation)
  - bus transactions impose serial order: writes are seen in the same order (write serialization)
Write-through State Transition Diagram

- 2 states per block (valid and invalid)
  - state of each memory block is a p-vector
- Hardware state bits associated with only blocks that are in the cache
  - other blocks can be seen as being in invalid (not-present) state in that cache
- Writes do not change block state locally
  - invalidate other caches
- Protocol allows multiple readers to be simultaneously active
  - until invalidated by writes

Problem with Write-Through

- High bandwidth requirements
  - every write from every processor goes to shared bus and memory
  - consider: 200MHz, 1CPI processor, and 15% instrs. are 8-byte stores
    - each processor generates 30M stores or 240MB data per second
  - 1GB/s bus can support only about 4 processors without saturating
- Write-back caches absorb most writes as cache hits
  - but need sophisticated protocols to ensure write propagation and serialization
- But, first let us understand other ordering issues ...

How to Order Reads/Writes by Different Processors?

Event synchronization

\[
\begin{array}{c|c}
\text{P}_1 & \text{P}_2 \\
\hline
A = 1; & \text{while (flag == 0);} /旋 idly */ \\
& \text{flag = 1;} \quad \text{print A;} \\
\end{array}
\]

More generally

\[
\begin{array}{c|c}
\text{P}_1 & \text{P}_2 \\
\hline
(1a) A = 1; & (2a) \text{print B;} \\
(1b) B = 2; & (2b) \text{print A;} \\
\end{array}
\]

Memory consistency model:
- specifies constraints on the order in which memory operations (from any process) can appear to execute with respect to one another
  - what orders are preserved?
  - given a load, constrains the possible values returned by it
  - contract between programmer and system

Sequential Consistency (Lamport’79)

[A multiprocessor system 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.

- Two aspects
  - program order: completion of previous memory operations
    - write completion is more crucial
  - write atomicity: serialization of writes to the same location

\[
\begin{array}{c|c}
\text{P}_1 & \text{P}_2 \\
\hline
A = 1; & \text{while (Flag == 0);} \\
\text{Flag = 1;} & \text{print A;} \\
\end{array}
\]

\[
\begin{array}{c|c|c}
\text{P}_1 & \text{P}_2 & \text{P}_3 \\
\hline
A = 1; & \text{if ( A==1 )} & \text{if ( B==1 )} \\
\text{Flag = 1;} & \text{B = 1} & \text{C = A} \\
\end{array}
\]
Sequential Consistency (contd.)

- Sufficient conditions
  - every process issues memory operations in program order
  - after a write operation is issued, the issuing process waits for the write to complete before issuing its next operation
  - after a read operation is issued, the issuing process waits for the read to complete, and for the write whose value is being returned by the read to complete, before issuing its next operation (provides write atomicity)

- Above conditions are not necessary
  - hardware needs only to appear to preserve sequential consistency

P 1 P 2
/* Assume initial values of A and B are 0 */

(1a) A = 1;
(1b) B = 2;
(2a) print B;
(2b) print A;

Memory Consistency and Cache Coherence

- Cache coherence is mechanism for implementing memory consistency
  - detect write completion (read completion is easy)
  - ensure write atomicity

- Centralized bus interconnect makes it easier
  - trivially true for write-through caches (earlier protocol)
    - if read obtains value of write \( W \), \( W \) is guaranteed to have completed since it caused a bus transaction
    - when write \( W \) is performed w.r.t. any processor, all previous writes in bus order have completed

- let us see some protocols for write-back caches
  - focus on invalidation-based protocols
  - see Culler/Singh/Gupta for examples of update-based protocols

Write-back Caches: Invalidation-based Protocols

- States
  - Valid, Invalid, Modified (Dirty)
  - Exclusive (for MESI protocol)
    - processor can modify without notifying anyone else (i.e. no bus transaction)
    - must first get block in exclusive state before writing into it
    - even if already in valid state, need transaction, so called a write miss

- Bus transactions
  - unprocessors
    - \( BusRd \): service a read miss
    - \( Flash \): to flush a cache block back to memory
  - multiprocessors
    - \( BusRdX \): tell others about impending write
      - makes the write visible, i.e., write is performed
      - only need this on first store to non-dirty data
    - coherence actions driven by \( BusRd \) and \( BusRdX \) transactions

3-State (MSI) Protocol

Snoop on \( BusRd \) and \( BusRdX \) transactions

- \( BusRd \)
  - if cache block is in Modified state, downgrade state to Shared, and flush data to memory

- \( BusRdX \)
  - if cache block is in Shared state, downgrade state to Invalid
  - if cache block is in Modified state, downgrade state to Invalid, and flush data to memory

- Lower-level choices
  - can also go to Invalid from Modified when \( BusRd \) is detected
    - decision depends on sharing pattern
Correctness of 3-State (MSI) Protocol

- Coherence conditions
  - write propagation because of BusRdX transactions
  - write serialization
    - all writes that appear on the bus (BusRdX) ordered by the bus
    - reads that appear on the bus ordered with respect to these
    - writes that don’t appear on the bus appear between two bus transactions
      - only issuing processor sees intermediate writes
      - other processors see writes serialized by the last bus transaction

- Sequential consistency conditions
  - write completion
    - can detect when write (the one that matters) appears on the bus
  - write atomicity
    - if a read returns the value of a write, that write has already become visible to all others already (can reason different cases)

4-state (MESI/Illinois) Protocol

- Problem with MSI protocol
  - reading and modifying data is 2 bus transactions, even with no sharing
    - 1->S followed by S->M
- Exclusive state
  - free to modify without transaction
  - main-memory is still kept up-to-date
  - I->E if no one else has a shared copy
    - needs “shared” line
- Who returns data when not in M state?
  - originally: cache-to-cache sharing
  - these days: memory
- Extension: MOESI protocol
  - owned state: exclusive and memory is not up-to-date

Cache Coherence: Performance Factors

- Impact of protocol optimizations
  - 3-state (MSI) versus 4-state (MESI) does not seem to matter much
    - workload-based evaluation (see Culler/Singh/Gupta for details)

- Impact of block size
  - affects compulsory and coherence misses
  - increasing block size has advantages and disadvantages
    - ✔ can reduce misses if spatial locality is good
    - ✗ can increase misses due to false sharing
    - ✗ can increase traffic due to fetching unnecessary data and false sharing
    - ✗ can increase miss penalty and hit cost
  - in practice (see Culler/Singh/Gupta for details)
    - impact of block size on miss rate varies with application
      - how well an application exploits spatial locality
      - bus traffic almost always increases

Synchronization Primitives in SMPs

*A parallel computer is a collection of processing elements that cooperate and communicate to solve large problems fast.*

- Types
  - mutual exclusion: “only one process is executing a portion of code”
  - event synchronization: “wait for another process”
    - point-to-point, group, global (barriers)

- History: Rich set of tradeoffs, no consensus
  - high-level language advocates want hardware locks/barriers
  - IBM 370: atomic compare&swap (for multiprogramming)
  - x86: any instruction can be prefixed with a lock modifier
  - SPARC: atomic register-memory ops (swap, compare&swap)
  - MIPS, IBM Power: no atomic operations but pair of instructions
    - load-locked, store-conditional (LL/SC)
    - later used by PowerPC and DEC Alpha too
Components of a Synchronization Event

• 3 steps
  – acquire method:
    • acquire right to the synch (enter critical section, go past event)
  – waiting algorithm
    • wait for synchronization to become available when it isn’t
  – release method
    • enable other processors to acquire right to the synch

• Waiting algorithm is independent of type of synchronization and difficult to support in hardware
  – blocking: waiting processes are descheduled
    • high overhead, but allows processor to do other things
  – busy-waiting: processes repeatedly test a location until it changes value
    • releasing process sets the location
    • lower overhead, but consumes resources (processor and network)
    • better when E(waiting time) < scheduling context switch time

Synchronization Primitives: Design Issues

• System versus user
  – user: wants high-level synchronization operations (locks, barriers)
  – system designer: how much hardware support?
    • speed versus cost and flexibility
    • current trend
      – system provides simple hardware primitives (atomic operations)
      – software libraries implement lock, barrier algorithms using these

• Challenges
  – same synchronization may have different needs at different times
    • lock accessed with low or high contention
    • different performance requirements: need for different primitives!
    • multiprogramming can also change synchronization behavior and needs
    – rich area of software-hardware interactions
      • which primitives available affects what algorithms can be used
      • which algorithms are effective affects what primitives to provide

Mutual Exclusion

• Hardware locks
  – separate lock lines on the bus
    • holder of a lock asserts the line (priority mechanism for multiple requesters)
    • inflexible: few locks can be in use at a time, hardwired waiting algorithm
  – lock registers (Cray XMP)
    • set of registers shared among processors

• Atomic read-modify-write (exchange) operations
  – test-and-set (T&S)
  – load-locked and store-conditional (LL-SC)
  – how software algorithms make use of these primitives

Need for Atomic Operations

• A simple software lock
  
  lock:   ld  register, location      /* copy location to register */
         cmp location, #0          /* compare with 0 */
         bnez lock               /* if not 0, try again */
         st  location, #1        /* store 1 to mark it locked */
         ret                     /* return control to caller */

  unlock:   st  location, #0      /* write 0 to location */
            ret                   /* return control to caller */

• Problem: lock needs atomicity in its own implementation
  – read (test) and write (set) of lock variable by a process not atomic

• Solution: atomic read-modify-write or exchange instructions
  – atomically test value of location and set it to another value, return success or failure somehow
Atomic Exchange Instruction

- General format
  - specifies a location and register
  - atomically
    - value in location read into a register
    - another value (function of value read or not) stored into location
  - many variants: varying degrees of flexibility in second part

- Simple example: test&set (T&S)
  - value in location read into a specified register, and (constant) 1 is stored
  - successful if value loaded into register is 0

```
lock: t&s register, location
  bx lock /* if not 0, try again */
ret /* return control to caller */
unlock: st location, #0 /* write 0 to location */
ret /* return control to caller */
```

Test&Set Locks: Performance Criteria

- Bus traffic
  - very low if repeatedly accessed by same processor
  - lots if many processors compete: poor scaling with \( p \)
    - each t&s generates invalidations, and all rush out again to t&s

- Poor fairness: processes can get starved

- Extensions to the basic test&set locks
  - test&set with backoff: wait a while before trying again
  - test and test&set: wait using read operations
    - reduces bus traffic, because no invalidations while waiting
    - lock variable will be invalidated on lock release

- Other read-modify-write primitives can be used too
  - swap, fetch&op (NYU Ultracomputer), compare&swap
  - can be cacheable or uncacheable (we assume cacheable)

Performance of Test&Set Locks

- Microbenchmark on SGI Challenge
  lock; delay(c)); unlock;
  - same total no. of lock calls as \( p \) increases; measure time per transfer

```
\begin{table}
\begin{tabular}{|c|c|c|c|c|c|}
\hline
\text{Number of processors} & \text{Time (\( \mu \)s)} \\
\hline
1 & 11 & 13 & 15 & 17 & 19 \\
2 & 9 & 11 & 13 & 15 & 17 \\
3 & 7 & 9 & 11 & 13 & 15 \\
4 & 5 & 7 & 9 & 11 & 13 \\
5 & 3 & 5 & 7 & 9 & 11 \\
\hline
\end{tabular}
\end{table}
```

Improved Hardware Primitives: LL-SC

- Goals
  - test lock availability with reads
  - failed read-modify-write attempts should not generate invalidations
  - use single primitive for a range of read-modify-write operations

- Load-Locked (or -linked), Store-Conditional
  - LL reads variable into register
  - follow with arbitrary instructions to manipulate its value
  - SC stores back to location if and only if no one else has written to the variable since this processor’s LL
    - if SC succeeds, means all three steps happened atomically
    - if it fails, does not write (or generate invalidations), but need to retry LL
  - success indicated by condition codes
    - see Culler/Singh/Gupta (Chapter 6) for implementation details
Simple Lock with LL-SC

lock:  ll   reg1, location /* LL location to reg1 */
bnz  reg1, lock /* retry if location is locked */
sc   location, reg2 /* SC reg2 into location*/
beqz reg2, lock /* if failed, start again */
ret
unlock: st location, #0 /* write 0 to location */
ret

• Operation
  – SC can fail (without putting transaction on bus) when the processor
    • detects intervening write even before trying to get bus
    • tries to get bus but another processor’s SC gets bus first
  – can do more fancy atomic ops by changing code between LL & SC
    • but keep it small (so SC succeeds), and no instructions that will need undoing
  – LL, SC are not lock, unlock: only allow non-atomicity to be detected!

• Problems
  – read misses on both successful SC and on release
    • no test-and-test&set analog, but can use backoff to reduce burstiness

10/1/98 33

More Efficient Software Locking Algorithms

• Ticket lock
  – fairness + only one process will try to get lock upon release
    • valuable when using test&set instructions; LL-SC does it already
  – works like the ticket system in a bank or deli
    • 2 counters per lock: next_ticket, now_serving
    • acquire: fetch-and-increment next_ticket, wait for (now_serving==next_ticket)
      – atomic operation on arrival, not when it is free
      • release: increment now_serving
    – still have read-miss problem on release

• Array-based queueing lock
  – above + only one process will incur read miss upon release
  – each process waits on a different locations
    • fetch-and-increment in acquire gives location address
    • release sets this address
  – requires space proportional to number of processes

10/1/98 34

Point-to-Point Event Synchronization

• Software methods
  – interrupts
  – busy-waiting: use ordinary variables as flags
  – blocking: use semaphores

• Full hardware support:
  – full-empty bit with each word in memory
    • set when word is “full” with newly produced data (i.e. when written)
    • unset when word is “empty” due to being consumed (i.e. when read)
  – natural for word-level producer-consumer synchronization
    • producer: write if empty, set to full; consumer: read if full, set to empty
  – hardware preserves atomicity of bit manipulation with read or write
  – problem: flexibility
    • multiple consumers, or multiple writes before consumer reads?
    • needs language support to specify when to use

10/1/98 35

Barriers

• Hardware barriers
  – wired-AND line separate from address/data bus
    • set input high when arrive, wait for output to be high to leave
    • in practice, multiple wires to allow reuse
  – useful when barriers are global and very frequent
    • difficult to support arbitrary subset of processors
      – even harder with multiple processes per processor
    • difficult to dynamically change number and identity of participants
      – e.g. latter due to process migration
    – not common today on bus-based machines

• Software algorithms
  – implemented using locks, flags, counters

10/1/98 36
A Simple Centralized Barrier

- A shared counter maintains number of processes that have arrived
  - increment when arrive (lock), check until it reaches numprocs

```c
struct bar_type { int counter; struct lock_type lock; int flag = 0; } bar_name;
BARRIER (bar_name, p) {
    LOCK(bar_name.lock);
    if (bar_name.counter == 0)
        bar_name.flag = 0; /* reset flag if first to reach*/
    mycount = bar_name.counter++; /* mycount is private */
    UNLOCK(bar_name.lock);
    if (mycount == p) {/* last to arrive */
        bar_name.counter = 0; /* reset for next barrier */
        bar_name.flag = 1; /* release waiters */
    } else while (bar_name.flag == 0) {}; /* busy wait for release */
}
```

- Problem?

A Working Centralized Barrier

- Problem with the simple centralized barrier
  - consecutively entering the same barrier doesn’t work
  - must prevent a process from entering until all have left previous instance

- Solution: Sense reversal
  - wait for flag to take different value consecutive times
  - toggle this value only when all processes reach

```c
BARRIER (bar_name, p) {
    local_sense = !(local_sense); /* toggle private sense variable */
    LOCK(bar_name.lock);
    mycount = bar_name.counter++; /* mycount is private */
    UNLOCK(bar_name.lock);
    if (mycount == p) /* last to arrive */
        bar_name.counter = 0; /* reset for next barrier */
        bar_name.flag = local_sense; /* release waiters */
    else {
        UNLOCK(bar_name.lock);
        while (bar_name.flag != local_sense) {}; /* busy wait for release */
    }
}
```

Barrier Algorithms: Performance Issues

- Latency: want short critical path in barrier
  - critical path length at least proportional to \( p \)
- Traffic: high contention likely, so want traffic to scale well
  - about \( 3p \) bus transactions

- Decentralized algorithm: Software combining tree

```
Contention
```

```
Little contention
```

```
Flat
```

- on a bus
  - same traffic, but higher latency
  - advantage is use of read/write operations instead of locks
- more important in scalable parallel machines (Lectures 5-7)

Implications for Parallel Software

- Load balance, inherent communication and extra work
  - issues same as before (unaffected by architecture)
  - assign so that only one processor writes a set of data
    - avoid write sharing
      - e.g. in RayTrace (read scene, write image): partition the image
  - Communication structure and mapping are not major issues
    - only require that processes do not migrate often (left to the OS)
  - Orchestration is the major issue
    - reduce cache misses and hence both latency and traffic
      - temporal locality: keep working sets tight enough to fit in cache
      - spatial locality: reduce fragmentation and false sharing
Bag of Tricks for Spatial Locality

- Reduce spatial interleaving of accesses
  - *(task assignment)* contiguous assignment of array elements
  - *(data structuring)* higher-dimensional arrays to keep partitions contiguous

(a) Two-dimensional array

(b) Four-dimensional array

Bag of Tricks for Spatial Locality (contd.)

- Beware of conflict misses
  - typically a problem with “power of 2” caches and arrays
  - leads to under-utilization of cache

Locations in subrows
Map to the same entries (indices) in the same cache.
The rest of the processor’s cache entries are not mapped to by locations in its partition (but would have been mapped to by subrows in other processor’s partitions) and are thus wasted.

Case Studies

- SGI Challenge with Powerpath bus

36 R4400 (2.7 GFLOPS)
1.2 GB/s bus (16 slots)
128-byte cache line
split-transaction with 8 outstanding requests
all transactions take 5 cycles

- Uses Illinois MESI protocol
  - with cache-to-cache sharing
- 250ns access time from address on bus to data on bus
  - but overall latency seen by processor is 1000ns!
    - 300ns for request to get from processor to bus
    - 300ns more for data to get to processor chip
Case Studies (contd.)

- SUN Enterprise, with Gigaplane bus

30 UltraSparcs (9 GFLOPS)
2.7 GB/s bus (16 slots)
64-byte cache line
split-transaction with
112 outstanding transactions
transactions take 11-18 cycles

- MOESI protocol
  - owned state for cache-to-cache sharing
- 300ns read miss latency
  - 11 cycle min bus protocol at 83.5 Mhz is 130ns of this time
  - rest is path through caches and the DRAM access

Lecture Summary

- Small-scale Shared Memory Machines
  - category with the largest volume
  - bus-based architectures
  - coherence and consistency
    - simple write-through invalidation-based protocol
    - MSI write-back invalidation-based protocol
    - MESI write-back invalidation-based protocol
  - synchronization
    - hardware/software techniques for mutual exclusion, event synchronization
  - performance issues
    - bus bandwidth: exploit spatial and temporal locality
  - case studies: SGI Challenge, Sun Enterprise

Next Lecture

- Scalable Distributed-Memory Machines
  - communication architectures
    - scalable networks, processor-network interfaces
  - support for put/get, remote memory access
  - case study: Cray T3E

- Tutorial
  - programming with threads

Readings
- Culler/Singh/Gupta: Chapter 7
- Almasi/Gottlieb: Chapter 10 (Section 10.2)