Announcements

- Homework 2 due date extended to Monday, October 15th
- 1 week to project proposal due date (October 17)
  - project groups (2-3 students per group)
    - Anybody looking for a project partner should sync up during the break today
    - Send me e-mail, set up an appointment to discuss project plans
- Homework 3 handed out today
  - Due back on Wednesday, October 24th
  - Start early!
- bm.scs.cs.nyu.edu (Linux Beowulf cluster)
  - run_seq and run_mpi scripts
    - problems with abnormal termination of program runs
  - let me know if things are still broken

Outline

- Last lecture
  - Architecture of symmetric multiprocessors
    - communication: coherence and consistency
      - snooping-based coherence protocols
- This lecture
  - Architecture of symmetric multiprocessors (cont’d)
  - synchronization
    - case study: Sun Enterprise
  - Architecture of distributed memory multiprocessors

(Review) Write-back Caches: Invalidation Protocols

- States
  - Shared (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
  - uniprocessors
    - BusRd: service a read miss
    - Flus: 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

[Culler/Singh/Gupta: Chapters 4, 7, web sources]
(Review) 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

(Review) 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
    - I→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
    - Other kinds: capacity, conflict
  - 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/Compaq/Intel 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 \( E(\text{waiting time}) < \text{scheduling context switch time} \)

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

• Early days: 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

• Current solution: 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
  
  ```c
  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

  ```c
  lock:   tss   register, location
          bnez  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
  - same total no. of lock calls as \( p \) increases; measure time per transfer

![Graph showing performance of test&set locks](chart.png)
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 */
bnez     reg1, lock  /* retry if location is locked */
sce      location, reg2 /* SC reg2 into location*/
bnez     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
  - unfair: see Culler/Singh/Gupta for ticket lock and array lock descriptions

Point-to-Point Event Synchronization

- **Software methods**
  - inter-processor 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

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
SMP Architectures: Implications for Parallel Software

- Load balance, inherent communication and extra work
  - issues same as before (unaffected by architecture)

- 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

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

- Copy data to increase locality
  - e.g., reuse of noncontiguous data
  - must trade off against cost of copying

- Pad and align arrays
  - can have false sharing versus fragmentation tradeoff

- Organize arrays of records for spatial locality
  - e.g. particles with fields: organize by particle or by field
    - in vector programs by field for unit-stride, in parallel often by particle
    - phases of program may have different access patterns and needs

- These issues can have greater impact than inherent communication
  - can cause us to revisit assignment decisions (e.g. row vs. block in grid)
Case Study: Sun Enterprise

- SUN Enterprise 3000-6000, with Gigaplane interconnect (circa 1997)

30 UltraSparcs (9 GFLOPS)
2.7 GB/s bus (16 slots)
64-byte cache line

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

GigaplaneTM bus (256 data, 41 address, 83 MHz)

I/O Cards

PMem ctrl

Bus Interface / Switch

CPU/Mem Cards

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

Case Study: Sun Enterprise (cont’d)

- Sun Enterprise 10000 (Starfire) (circa 1999)

High-level architecture remains the same
- Upto 16 system boards, each with
  - Up to 4 400-500 MHz UltraSPARC II processors (total: 64)
  - 4 GB memory (total: 64 GB)

Gigaplane XB interconnect
  - “bus” for addresses and control
    - 4 global interleaved address buses
    - 2 cycle address transfer rate (highly pipelined)
      - 167 million snoops per second (at 83.3 MHz clock)
    - 16x16 “crossbar” for data
      - Sustained data bandwidth of 12.8 GB/s (versus 2.7 GB/s)
    - Transactions take 26-38 cycles (versus 11-18): 500 ns
      - Latency has increased slightly, traded off versus bandwidth improvements

Small-scale SMPs: Summary

- Main points
  - 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 study: Sun Enterprise

Case Study: Sun Enterprise (cont’d)

- Sun Enterprise 15000 (Starcat) (September 2001)

Same high-level architecture as 10K, but expected improvements
- Upto 18 system boards
  - Each capable of holding up to four 900 MHz UltraSparc III processors
  - 72 – 106 processors (17 of the boards can be replaced with those capable of
    holding 6 processors apiece)
  - 500 GB of memory, expandable to 1 TB
  - Sustained 43 GB/s interconnect (versus 12.8 GB/s for 10K)
    - Architecture similar to 10K
      - Separate paths for address/control and data (18x18 crossbar)
Scalable Distributed-Memory Multiprocessors

Why Scalable Parallel Machines?

Limitations of a shared bus architecture

- **Hardware**
  - length considerations
    - typically, less than 2 ft
    - determined by capacitive loading (signal quality) and power considerations
  - fixed number of slots: each device loads the bus
  - fixed maximum bandwidth
  - limited number of outstanding bus transactions
    - bus imposes global order
  - Enterprise 15K (72-106 processors) vs. IBM SP (2000+ processors)

- **Software**
  - single operating system
    - if any processor fails, the system is rebooted
    - Sun Enterprise: Notion of “system domains” for fault isolation

What Does Scalable Mean?

*A scalable system attempts to avoid inherent design limits on the extent to which resources can be added to the system*

- Four dimensions
  - *bandwidth*: concurrent transactions on independent wires
  - *latency*: \( T(n) = \text{overhead} + \text{channel time} + \text{routing delay} \)
  - *cost of the system*: \( \text{fixed cost} + \text{incremental cost} (p, m) \)
  - *packaging of the system*: modules, clock distribution, wires

- In an ideal scalable system
  - latency stays constant
  - bandwidth and cost of the system grow linearly
  - for system packaging, balance between
    - dense packing: to reduce fixed cost of wires, connectors, etc.
    - loose coupling: to reduce engineering effort and permit technology scaling

Generic Distributed-Memory Multiprocessor

2 levels of switches
- *intranode* switch is typically a bus
- *internode* switches provide independent communication paths between nodes so that bandwidth increases as nodes are added
Key Distinction: Level of Integration

chip-level (processor bus)  
- J-machine,  
- Alewife, nCUBE/2

board-level (memory bus)  
- CM-5, Paragon,  
- T3E, Origin

system-level (I/O bus)  
- SP,  
- PCs/Myrinet  
- PCs/Fast Ethernet

Processor Bus  
Memory Bus  
I/O Bus

10 GB/s  
1-5 cycles

1 GB/s  
20-50 cycles

~100 MB/s  
500-1000 cycles

Network Transactions: Design Issues

- **Protection:** who performs the check?  
- **Format:** fixed vs. variable-sized packets, encapsulation  
- **Output buffering:** staging area on source  
- **Media arbitration:** global vs. per-link reservation  
- **Destination name and routing:** logical to physical translation  
- **Input buffering:** staging area on destination, N->1 merge  
- **Action:** what happens as a result of the transaction?  
- **Completion detection:** what is the source informed of?  
- **Transaction ordering:** what ordering guarantees?  
- **Deadlock avoidance:** affected by routing and end-point behavior  
- **Delivery guarantees:** what happens when the destination buffer is full?

Supporting a Shared Address Space

- **Issues**  
  - **input buffer overflow:** many nodes request from same address  
  - **fetch deadlock:** a node must be able to sink requests  
  - **delivery order:** important for implementing memory consistency models (more in Lectures 6 and 7)
Supporting Message Passing: Synchronous Protocol

- Alternately, can be receiver-initiated (but, restricts tag matching)
- Tag check on the sender: requires only two network transactions

Message Passing: Asynchronous Protocols

- Problems with the optimistic protocol
  - store-and-forward delay (if no posted receive buffer)
  - input buffer overflow (function of program behavior)

Active Messages

- A message-passing abstraction closer to the level of the underlying network transaction
  - message send associated with a handler at the other end
    - handler integrates message into ongoing computation
    - allocates a buffer to store message if necessary
    - very often, this is not required (e.g., for looking up an address)
  - no explicit receive
    - reduces cost of tag matching and buffering
  - request-response transactions constitute a restricted RPC
    - response handlers cannot send messages to prevent deadlock
  - interface permits very efficient implementations
    - Higher-level abstractions (MPI) typically built on top of active messages

For small messages, a credit-based scheme can reduce handshake costs
Next Lecture

- Scalable Distributed-Memory Machines (cont’d)
  - processor-network interfaces: HW design choices
  - support for put/get, remote memory access
  - case study: Cray T3E, IBM SP-2

- Scalable Shared-Memory Machines
  - directory-based coherence
  - relaxed memory consistency models

Reading
  - Culler/Singh/Gupta: Chapter 7