Outline

- Last lecture
  - parallel constructs in different programming models
    • data-parallel, shared memory, message passing
  - analytical models of parallel computation: PRAM, LogP

- This lecture
  - Programming for performance
    • Common issues: naming, synchronization, latency, bandwidth
  - Architecture of symmetric multiprocessors
    • communication: coherence and consistency
    • snooping-based coherence protocols
    • synchronization
    • case study: Sun Enterprise

Parallel Programming as Successive Refinement

- Not all issues dealt with up front

  - Partitioning often independent of architecture, and done first
    • view machine as a collection of communicating processors
      • PRAM + communication costs
      • balancing the workload
      • reducing required amount of inherent communication
  - Then, interactions with architecture (orchestration)
    • view machine as extended memory hierarchy
      • extra communication due to architectural interactions
      • cost of communication depends on how it is structured
    • may inspire changes in partitioning

/ Culler/Singh/Gupta: Chapter 4, web sources /
Algorithmic Issues: Partitioning

3 major focus areas
- **Balancing the workload** + reducing wait time at synchronization points
- Reducing inherent communication
- Reducing extra work

- Trade off even among these algorithmic issues
  - minimize communication → run on 1 processor → extreme load imbalance
  - maximize load balance → random assignment of tiny tasks → no control over communication
  - good partition may imply extra work to compute or manage it

- Goal is to compromise
  - fortunately, often not difficult in practice

Focus 1: Load Balance and Synchronization Time

- **Limits on speedup**
  \[ \text{speedup}_{\text{problem}}(p) \leq \frac{\text{sequential work}}{\max(\text{work on any processor})} \]
  - work includes data access and other costs
    - not just equal work, but processors must be busy at the same time
  - Four parts to the problem
    - identify enough concurrency
      - source: Loop-structure, fundamental dependencies, new algorithms
      - dimensions: Data vs. Task parallelism
    - decide how to manage it
      - static vs. dynamic tasking
    - determine the granularity at which to exploit it
      - fine vs. coarse granularity
    - reduce serialization and cost of synchronization
      - global (barriers) vs. point-to-point synchronization (locks)
      - reduce mutual exclusion regions: separate locks, smaller critical sections

Implications of Load Balance and Synchronization

- Extends speedup limit expression to
  \[ \text{speedup}_{\text{problem}}(p) \leq \frac{\text{sequential work}}{\max(\text{work on any processor} + \frac{\text{communication wait time}}{\text{communication costs}})} \]

- Generally, the responsibility of the programmer
  - algorithmic decisions, based on fairly simple machine model
    - PRAM + communication has non-zero cost

- How can architecture help?
  - **fine-grained communication** (low overhead, latency)
    - allows smaller tasks, better load balance (low-overhead access to queues)
  - **naming** logically shared data in the presence of task stealing
    - need to access data of stolen tasks
    - hardware shared address space advantageous

Focus 2: Reducing Inherent Communication

- Simple machine view: communication is expensive!
  \[ \text{speedup}_{\text{problem}}(p) \leq \frac{\text{sequential work}}{\max(\text{work on any processor} + \frac{\text{communication wait time}}{\text{communication costs}})} \]
  - metric: \textit{communication to computation ratio}
  - provides guidance on which communication aspect is important
    - if computation is execution time, ratio gives average BW need
    - if computation is operation count, gives extremes in impact of latency and BW
      - latency: assume no latency hiding
      - bandwidth: assume all latency is hidden
  - Solution: assign tasks that access same data to same process
    - solving communication and load balance is NP-hard (in general)
    - however, simple heuristic solutions work well
      - exploit application structure: e.g., domain decomposition
Focus 3: Reducing Extra Work

- Extends speedup limit expression
  \[ \text{speedup}_{\text{process}}(p) \leq \frac{\text{sequential work}}{\max\left( \text{work on any processor + synchronization wait time + communication costs + extra work} \right)} \]

- Common sources of extra work
  - computing a good partition (e.g., in a sparse matrix computation)
  - using redundant computation to avoid communication
  - task, data, and process management overhead
    - applications, languages, run-time systems, OS
    - imposing structure on communication
      - coalescing messages, allowing effective naming
- How can architecture help?
  - efficient support of communication and synchronization (orchestration)

Architecture Issues: Orchestration

- Architecture-independent partitioning insufficient for performance
  - communication/synchronization costs determined not only by amount
    - depends on structuring of communication (naming, synchronization)
    - cost of communication in system (latency, bandwidth)
- Memory-oriented view of a multiprocessor
- Communication Costs

  Accesses not satisfied in local hierarchy levels cause communication
  - Inherent
    - determined by program
    - assuming unlimited capacity, small transfers, perfect knowledge
  - Artifactual
    - determined by program implementation and architecture interactions
    - some reasons:
      - poor allocation of data across distributed memories
      - redundant communication of data
      - unnecessary data in a transfer or unnecessary transfers (system granularity)
      - finite replication capacity
        - four kinds of cache misses: compulsory, capacity, conflict, coherence
        - finite capacity affects capacity and conflict misses
    - tradeoff between reducing artifactual communication cost and improving spatial locality

Orchestration for Performance

  Two areas of focus
  - Reducing amount of communication
    - inherent: change logical data sharing patterns in algorithm
    - artifactual: exploit spatial, temporal locality in extended hierarchy
      - techniques often similar to those on uniprocessors
      - shared address space machines support this in hardware, distributed memory machines support the same techniques in software
  - Structuring communication to reduce cost
    - When does communication happen?
    - Is it overlapped with computation?
  - Lectures 4-7 focus on hardware approaches
  - Lectures 8-11 focus on software and hybrid approaches
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 coherency problem
  - values in different caches need to be kept consistent

The Cache Coherency 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

- Is cache coherence a problem on uniprocessors?
  - Yes!
    - interaction between caches and I/O devices
      - infrequent software solutions work well
      - uncacheable 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, 1 CPI 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 ...
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

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
    - Okay to do 1b -> 1a -> 2b -> 2a (indistinguishable from 1a -> 1b -> 2a -> 2b)

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)
    - Write and read misses to all locations serialized by bus into bus order
      - 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**
  - *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
    - *Flush*: 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

---

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

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 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

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

  ```
  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 /* if not 0, try again */
  bnez     lock /* return control to caller */
  ret
  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

![Graph showing performance of Test&Set Locks](image)

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!

Simple Lock with LL-SC

- 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)
  – 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

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.
Bag of Tricks for Spatial Locality (contd.)

- Use **per-processor heaps** for dynamic memory allocation
  - ensures data structures use different cache blocks
- 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 1998)
  - 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 study: 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, IBM SP-2

- Tutorial
  - programming with MPI

Reading
- Culler/Singh/Gupta: Chapter 7