Parallel Processing and Multiprocessors

why parallel processing?
types of parallel processors
cache coherence
synchronization
memory ordering

Why Parallel Processing

go past physical limits of uniprocessing (speed of light)
pros: performance
  • power
  • cost-effectiveness (commodity parts)
  • fault tolerance
cons: difficult to parallelize applications
  • automatic by compiler hard in general cases
  • parallel program development
  • IT IS THE SOFTWARE, stupid!

Amdahl’s Law

speedup = 1/(frac_{enhanced}/speedup_{enhanced} + 1 - frac_{enhanced})
speedup of 80 with 100 processors
=> frac_{parallel} = 0.9975
only 0.25% work can be serial
may help: problems where parallel parts scale faster than serial
  • O(n^2) parallel vs. O(n) serial
challenge: long latencies (often several microsecs)
  • achieve data locality in some fashion

Application Domains

Parallel Processing - true parallelism in one job
  • data may be tightly shared
OS - large parallel program that runs a lot of time
  • typically hand-crafted and fine-tuned
  • data more loosely shared
  • typically locked data structures at differing granularities
transaction processing - parallel among independent transactions
  • throughput oriented parallelism
**Types**

Flynn Taxonomy
- 1966
- not all encompassing but simple
- based on # instruction streams and data streams
- SISD - uniprocessor
- SIMD - like vector
- MISD - few practical examples
- MIMD - multiprocessors - most common, very flexible

**Single Instruction Single Data (SISD)**

Your basic uniprocessor

**Single Instruction Multiple Data (SIMD)**

Vectors are same as SIMD
- deeply pipelined FUs vs. multiple FUs in previous slide
- intrs and data usually separated
- leads to data parallel programming model
- works best for very regular, loop-oriented problems
- many important classes - eg graphics
- not for commercial databases, middleware (80% of server codes)
- automatic parallelization can work
Multiple Instruction Multiple Data (MIMD)

- most flexible and of most interest to us
- has become the general-purpose computer
- automatic parallelization more difficult

Perfection: PRAM

- parallel RAM - theoretical model
- fully shared memory - unit latency
- no contention, no need to exploit locality

Perfection not achievable

- latencies grow as the system size grows
- bandwidths restricted by memory organization and interconnect
- dealing with reality leads to division between
  - UMA and NUMA

UMA: uniform memory access

- Main Memory
  - contention in memory banks
- Interconnection Network
  - contention in network
- Processor
  - long latency

Processor
  - Processor
UMA: uniform memory access

- latencies are the same
  - but may be high
- data placement unimportant
- latency gets worse as system grows => scalability issues
- typically used in small MPs only
- contention restricts bandwidth
- caches are often allowed in UMA systems

Caches

- another way of tackling latency/bandwidth problems
- holds recently used data
- BUT cache coherence problems

NUMA: non-uniform memory access

- latency low to local memory
- latency much higher to remote memories
- performance very sensitive to data placement
- bandwidth to local may be higher
- contention in network and for memories
NUMA Multiprocessors

- shared memory
  - one logical address space
  - can be treated as shared memory
  - use synchronization (e.g., locks) to access shared data

- multicomputers (message passing)
  - each processor has its own memory address space
  - use message passing for communication

Clustered Systems

- small UMA nodes in large UMA systems
- hybrid of sorts
- note: ambiguity of the term “cluster”

Cluster types

- globally shared memory - Illinois Cedar

- No global memory - Stanford Dash, Wisconsin Typhoon
COMA: cache only memory architecture

- caches only causes data to migrate naturally

Writing Parallel Programs

Decomposition,
  - where is the parallelism
  - Break up work

Assignment
  - which thread does what (think of data)

Orchestration
  - synchronization, communication

Mapping
  - which thread runs where (usually thru OS)

Process communication

- parallel processes (tasks) need to communicate

communication method leads to another division
  - message passing
  - shared memory

Shared memory vs message passing

shared memory
  - programs use loads/stores
    + conceptually compatible with uniprocessors/small MPs
    + ease of programming if communication complex/dynamic
    + lower latency communicating small data items
    + hardware controlled sharing allows automatic data motion
Shared memory vs message passing

Message passing

- programs use sends/receives
  - simpler hardware
  - communication pattern explicit and precise
  - but they MUST be explicit and precise
  - least common denominator
- shared memory MP can emulate message passing easily
- biggest programming burden: managing communication artifacts

Shared memory

Thread1

....
compute (data)
store( A, B, C, D, ...)
synchronize
......
A B C D SAME in both threads - SINGLE shared memory

Thread2

....
synchronize
load (A, B, C, D, ....)
......

Mesg Passing

Thread1

....
compute (data)
store(A, B, C, D ..)
synchronize
......

Thread2

....
receive (msg)
scatter (msg to A B C D ..)
gather (A B C D into msg)
load (A, B, C, D, ....)
send (msg)
......
A B C D are DIFFERENT in each thread -- PRIVATE memory
Eg: Sequential Ocean

Eg: Shared Memory Ocean

Eg: Mesg Passing Ocean

Eg: Mesg Pass Ocean
Process vs. thread

heavy-weight process
- separate PC, regs, stack
- different address space (page table, heap)

light-weight processes aka threads
- different PC, regs, stack (“context”)
- same address space

sharing across heavy-weight processes possible via page table

Shared memory MPs: cache coherence

e.g.,
- proc 1 reads A
- proc 2 reads A
- proc 1 writes A
- now proc 2 has stale data regardless of write-thru/-back

informally - method to make memory coherent despite caches
- caches can be viewed as large buffers
- with potentially very large delays
- replication/buffering + writes = coherence

Shared memory MPs: cache coherence

cache coherence suggests absolute time scale
- not necessary
- what is required is appearance of coherence
  - not instantaneous coherence
- Bart Simpson’s famous words -
  - “nobody saw me do it so I didn’t do it!”
- e.g. temporary incoherence between
  - writeback cache and memory ok

Causes of Incoherence

sharing of writeable data
- cause most commonly considered

process migration
- can occur even if independent jobs are executing

I/O
- can be fixed by OS cache flushes
What is the software’s expectation?

P1                      P2                      P3
...                      ...                      ...
st A                     Id A                     Id A
..                      ..                      ..

to understand coherence it is KEY to really understand the software’s expectations

What is the software’s expectation?

do invalidations have to be instantaneous?
  • program expects threads to slip and slide
  • assumes NOTHING about relative speeds

can invalidations be delayed arbitrarily?
  • yes if there is no synchronization (not “defined” yet)
  • no if there is synchronization (complicated - later)

can invalidations to SINGLE location applied to each cache delayed arbitrarily?
  • write atomicity, write serialization

 Writes

Write atomicity
  • either a write happens for EVERYone or not at all to ANYone

Write serialization -- atomic achieves serialization
  • writes to the SAME location from DIFFERENT caches appear in the SAME order to EVERYone
  • ie not only does each write happen atomically for EVERYone but all those writes in SAME order for EVERYone
  • this order is the “bus grant order” for those writes

Above two requirements “intuitive” now, will be “formalized” later

 Cache coherence

the ONLY thing coherence provides is write atomicity
  • ie when a write happens, it happens for all so nobody can see the previous value
  • even this suggests instantaneous and global update of all copies but cannot be implemented that way
  • writes take a non-zero time window to complete (ie not instantaneous) and after this window nobody can see the previous value and during this window any access to the write location is blocked

the ONLY thing coherence provides is write atomicity
Solutions to Coherence Problems

Disallow private caches - put caches next to memory
make shared-data non-cacheable - simplest software solution
have software flush at strategic times - e.g., after critical sections
use global table - does not scale
use bus-based snoopy cache - small scale e.g., MULTICORES!
use distributed memory-based directories - large scale
  • like a phone book
  • e.g., SGI Origin

Bus-based Coherence

typically write-back caches are used to minimize traffic
typical states
  • invalid
  • shared
  • exclusive
consider state changes for
  • read hit, read miss
  • write hit, write miss
snooping: ALL caches monitor ALL traffic on the bus

Bus-based Coherence

writes?
  • invalidate copies in other caches
  • => invalidation protocols
  • OR
  • update copies in other caches
  • => update protocols

Snoopy Coherence

cache controller updates coherence states upon CPU access and bus requests
Simple Write Invalidate Protocol

State transitions for processor actions

Invalid

Place read miss on bus

CPU write

Write miss

CPU Read Hit

Shared

Cores read/write only

Place write miss on bus

CPU Hit

Exclusive

Cores read/write

Wide lines: 2 clock actions, narrow lines: 1

Simple Write Invalidate Protocol

State transitions for bus actions

Invalid

Write miss for this block

Exclusive

Read Miss for this block

Shared

Write back block

Wide lines: 2 clock actions, narrow lines: 1

So why is this hard?

I lied - in real snoopy protocols the transitions are not “atomic”

• bus requests/actions take arbitrary time (e.g., write miss ->
  bus acquire -> receive data -> complete) so transitions are
  split and many more “transition states”

• pipelined buses, so multiple requests at the same time
  (from one cache and multiple caches)

• so real protocols are non-atomic with need to nack others,
  let go of current requests, etc with plenty of chance for
  deadlocks, livelocks, starvation

Bus-based Protocols: Performance

Misses: Mark Hill’s 3C model

• capacity
• compulsory
• conflict
• coherence

coherence misses: additional misses due to coherence protocol
as processors are added

• capacity misses decrease (total cache size increases)
  • coherence misses increase (more communication)
Bus-based Protocols: Performance

- as cache size is increased
  - inherent coherence misses limit overall miss rate

- as block size is increased
  - less effective than a uniprocessor
    - less spatial locality
    - false sharing
  - more bus traffic
    - problem in bus-based MP

Directory-based Cache Coherence

- an alternative for large MPs
- sharing info in memory directory

- directory states of memory block
  - shared - >=1 processor has data and memory is upToDate
  - uncached - no processor has data
  - exclusive - only one processor has data/memory is old

- directory entry tracks which processors has data
  - e.g., via a bit vector

Directory-based Cache Coherence

- block in uncached state: memory current
  - read miss - send data to requester; mark block shared; sharers <- requester
  - write miss - send data to requester; mark block exclusive; sharers <- requester (owner)

- block in shared: memory current
  - read miss - send data to requester; sharers +<- requester (add to the list)
  - write miss - send data to requester; invalidate all sharers; mark block exclusive; sharers <- requester (owner)
Directory-based Cache Coherence

block in exclusive; memory stale
• read miss - fetch data from owner; update memory; mark data shared; sharers +<- requester (add to prev owner)
• data write-back - mark block uncached; sharers <- empty
• write miss - fetch data from owner; mark block exclusive; sharers <- requester (new owner)

Pretty much the SAME state machine as in snoopy

Consider 3 processors - P1 P2 P3
sequence of events
• data in no caches
• all 3 processors do read
• P3 does write
• C3 hits; but does not have ownership

• C3 makes write request
• directory sends invalidates to C1 and C2
• C1 and C2 invalidate and ack
• directory receives acks; sets line exclusive; sends write permission to C3
• C3 sets line dirty; writes to cache; P3 proceeds

How are writes made atomic?
How are writes serialized?
• in the order of arrival at directory

Performance

divide misses into local and remote
miss rate decreases as caches grow larger
but coherence dampens the decrease in miss rate

latency effects
• cache hit: 1 cycle
• local miss: 25+ cycles
• remote miss (home): 75+ cycles
• remote miss (3-hop): 100+ cycles

Research: How do we make all misses look like hits or local miss?
So why is this hard?

In reality, transitions are not atomic
  • like a real snoopy protocol but worse because no single “bus grant order” to make sense of things
  • correctness is guaranteed by serializing writes to one location at the directory and ordering replies at the requester

Need to reduce latency => reduce #hops => overlap some steps in protocol (eg forward from owner directly to requestor instead of going first to directory and then to requestor
  • could deadlock

in general, more possibilities of deadlocks than snoopy

Types of synchronization

mutual exclusion - allow only one thread in a critical section
  • enter critical section only if lock available else wait
  • after finishing - unlock to let next thread
  • eg ticket reservation, Ocean (order unimportant)

producer-consumer communicatn (producer BEFORE consumer)
  • use locks to do signal-wait (666 or other courses)

barrier ( global synchronization for all or subset of threads)
  • threads wait at barrier till all threads reach barrier - Ocean
  • use locks to count threads at barrier (666)

Communication and Synchronization

consider shared memory MPs

communication: exchange of data among processes
synchronization: special communication where data is control info
  • e.g., critical section

Synchronization

Synchronization can be done with ordinary loads and stores
  • <proc 1>
  • flag1 = 0
  • ---
  • flag1 =1
  • lab1: if (flag2 == 1) goto lab1
  • (critical section)
  • ---
  • flag1 = 0
Synchronization

• <proc 2>
• flag2 = 0
• --
• flag2 = 1
• lab2: if (flag1 == 1) goto lab2
• (critical section)
• --
• flag2 = 0

BUT difficult to implement/debug (this can deadlock!)

Software typically loops until lock=0 is returned (spin lock)

• while (test&set(a_lock) == 1) ; /* wait */
• a lock is a variable (memory location)
• program associates locks with data structures to protect data structures but lock address and data structure address are unrelated - the relationship is unknown to hardware and is in mind of the programmer

How might this be implemented in hardware?
• KEY - in test&set, first two instructions should be atomic
• ie we need atomic swap of temp and lock

Synchronization Primitives

Let's look at locks -- can build other sync using locks

• test&set (lock) {
  • temp := lock;
  • lock := 1;
  • return (temp);
• }
• reset(lock) {
  • lock := 0;
• }

Example Synchronization Primitives

locks need a read and a write to be atomic but hard to do in hardware TWO operations indivisibly (eg disallow other access in between => 2 threads locking at about same time will deadlock!)

Solution: Use two instructions but check if atomicity violated

load linked + store conditional - two instructions

• load linked reads value and sets global address register
• store conditional writes value if global address is “unchanged”
  • any change => write from another CPU => invalidate global address => global address register invalid => store conditional fails
Example Synchronization Primitives

- e.g., atomic exchange
- try: mov r3, r4  # move exchange value
- li r2, 0(r1)  # load linked
- sc r3, 0(r1)  # store conditional
- beqz r3, try  # if store fails
- mov r4, r2  # load value to r4

User-level Synchronization

spin locks
- li r2, #1  # r1 has lock address
- lockit: exchg r2, 0(r1)  # atomic exchange
- bnez r2, lockit  # if locked

with coherence, exchange => writes => lots of traffic
- key problem - checking and writing fused together
- alternative: separate checking and writing
- spin on check for unlock only; if unlocked, try to lock
- also called “test and test and set”

Memory ordering and Consistency

Coherence gives write atomicity & serialization
- for ONE location - across threads

What about accesses to MULTIPLE locations from ONE thread --
when should coherence action for each access complete?
- sequentially -- too slow (OoO CPU, lookup-free caches)
- overlapped - fast but there are DEEP correctness issues
- But why - accesses to DIFFERENT locations are independent (as per Ch. 3) so why are there issues?

Issues so deep that new topics: ordering and consistency
Memory Ordering

You may think coherence should complete at synchronization

- thread1
- st X
- ld Y
- st B
- unlock B_lock (unlock = st)

- thread2
- lock B_lock (test&set)
- ld B

synchronization is done to see “updated” values, so coherence should complete at synch to make new values visible

- complete st X, B invalidations at lock/unlock

Memory Ordering - Eg1

Previous slide assumes that synchronization is clearly marked to be hardware-visible (eg test&set uses ll and sc)

But that may not be true - look at this GENERIC example

- A, flag = 0
- thread1
- thread2
- A = 1
- flag = 1

- while (!flag); # wait
- print A

- thread1 has 2 stores which are not re-ordered as per Ch. 2 (even if different addresses - stores in program order) but such re-ordering can happen in interconnect

Memory Ordering - Eg2

memory updates may become re-ordered by the memory system e.g.,

- Proc 1
- A = 0
- --
- A = 1
- L1: print B

- Proc 2
- B = 0
- --
- B =1
- L2: print A

intuitively impossible for BOTH A and B to be 0

can happen if memory operations are REOrdered

Reasons for inconsistency
Memory Ordering - Eg3

Time in MP is relative
- there is no system-wide total time ordered scale
- time is only partial order

consider possible interleavings
- A, B, C = 0
- P1                       P2                     P3
- a: A = 1               c: B =1              e: C =1
- b: print BC           d: print AC         f: print AB

we can observe actual sequence indirectly via printed values
if each stream is executed in order; a set of interleavings possible
acbdef, ecdfab, etc

=> (10, 10, 11) (11,01,01)

if instructions execute out of order more interleavings possible
e.g., adbcfe

=> 00, 10, 11

Memory Ordering - Eg3

each interleaving suggests a specific value will be printed
- if memory is atomic

if memory is not atomic
- different processors may perceive different orderings
  eg 01, 10, 01=> P1: e->c     P2: a->e     P3: c-> a
  - a cycle which is impossible intuitively
- can cause correctness problems in MPs
- synchronization will break - if lock acquire is seen by some but not all threads then more than one thread may get lock!

Memory Ordering

Why is this happening? Because addresses are different (Eg1 - flag and A) so hardware has no way to know their semantic relationship - that flag “protects” A (which is in the programmer’s mind)

- KEY -- Problem occurs EVEN if writes atomic/serialized
- Ch. 2 prohibits reordering same addresses
- Ch. 4 prohibits reordering different addresses!

So are we doomed? NO!

- Answer 1: tell hardware when not to reorder
  - different choices of “when” - different mem models
- Answer 2: don’t tell hardware but reorder speculatively
Memory models

Formally defines what ordering is enforced by hardware/compiler

- allows programmer to know what to expect
- eg loads are re-ordered wrt stores but not other loads
- eg loads are re-ordered wrt other loads and stores

If programmer wants no reordering, use a special instruction

- in general called a “memory barrier” instruction (membar)
- all instrs before membar complete and nothing after started
  - sounds familiar?

Sequential Consistency

"system is sequentially consistent if the result of ANY execution is the same as if the operations of all processors were executed in SOME sequential order and the operations of each individual processor appears in this sequence in the order specified by the program"

Memory Models

Intel x86, Sun Sparc, Dec Alpha - all different models!

- programs assuming “stricter” model may not run correctly on machines implementing “looser” models
- stricter-model program will not have enough membars to reordering in looser-model machine

looser models generically called relaxed models

Relaxed models hard to program (but better performance)

Strictest model - easiest to program - called sequential consistency

- all memory ops done in program order and atomically

Speculation for Seq Consistency

Would Seq Consistency perform poorly?

No - use speculation

- Allow loads to go out of program order (loads are critical for performance)
- if there is an invalidation between the times the load completes and load commits, then the load may have seen an incorrect value => squash and re-execute load and later instructions
- if there is no invalidation, nobody saw me do it so I didn’t do it
- Preserves Ch. 2 and 4 (OoO issue, lockup-free, banking,...)
Coherence and Consistency

related but not same
- implementation issues put the two together
- after 565 you should NEVER use one word for the other!

coherence
- deals with ONE location across threads
- answers WHAT is kept coherent

consistency (problem even if no caches or correct coherence)
- deals with ordering of MULTIPLE locations in one thread
- answers BY WHEN coherence actions should be complete

Atomicsity, Memory Consistency, Ordering

atomicity
- informally - accesses to a variable occur one at a time and “update” ALL copies

sequential consistency
- informally - memory is a “single time-shared block”

relaxed consistency
- informally - only synch operations are consistent; high perf

ordering of events
- underlying implementation feature - sequential/relaxed

Atomicity

define
- an access by a processor i on var X is atomic if no other processor is allowed to access any copy of X while the access by i is in progress

hardware atomicity is not necessarily required
- only APPEAR to be atomic without literally updating ALL copies simultaneously

atomicity is key to synchronization operations
ordering problems still possible - refers to individual variables
- NOT ordering of updates to different variables
Coherence revisited

informally makes writes to one location appear in same order to other processors

focuses on single location rather than ordering among locations
  • caches as large buffers
  • with large delays

coherence is mechanism to provide appearance of write atomicity

Consistency models

model 1: sequentially consistent events are strongly ordered
  • chapter 3 invalid? NO - only APPEAR strongly ordered
  • but may be low performance (fixed with speculation!)

model 2: relaxed consistency - relax program order/atomicity
  • require order only for synchronization operations
  • non-synch and synchronizing operations may look alike
  • programmer marks synchronization
  • no reordering across synch (in compiler or hardware)
  • higher performance

Future

Future of computer systems
  • higher performance
  • but wiring limitations
  • power limitations
  • memory system limitations
  • multiprocessors
  • billion transistor chips
  • better processors, better caches, smarter software, ...
  • the next decade will be more interesting than the previous