Topics for Thread Level Parallelism (TLP)

- Parallelism (centered around … )
  - Instruction Level Parallelism
  - Data Level Parallelism
  - Thread Level Parallelism

- TLP Introduction
  - 5.1
- SMP and Snooping Cache Coherence Protocol
  - 5.2
- Distributed Shared-Memory and Directory-Based Coherence
  - 5.4
- Synchronization Basics and Memory Consistency Model
  - 5.5, 5.6

Finish in three sessions
Acknowledge and Copyright

- Slides adapted from
  - UC Berkeley course “Computer Science 252: Graduate Computer Architecture” of David E. Culler Copyright(C) 2005 UCB
  - UC Berkeley course Computer Science 252, Graduate Computer Architecture Spring 2012 of John Kubiatowicz Copyright(C) 2012 UCB
  - Computer Science 152: Computer Architecture and Engineering, Spring 2016 by Dr. George Michelogiannakis from UC Berkeley
  - Arvind (MIT), Krste Asanovic (MIT/UCB), Joel Emer (Intel/MIT), James Hoe (CMU), John Kubiatowicz (UCB), and David Patterson (UCB)
  - UH Edgar Gabrial, Computer Architecture Course: http://www2.cs.uh.edu/~gabriel/courses/cosc6385_s16/index.shtml

- https://passlab.github.io/CSE564/copyrightack.html
Consider:

/* each thread to update shared variable best_cost */

if (my_cost < best_cost)
    best_cost = my_cost;

- two threads,
- the initial value of best_cost is 100,
- the values of my_cost are 50 and 75 for threads t1 and t2

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (my_cost (50) &lt; best_cost)</td>
<td>if (my_cost (75) &lt; best_cost)</td>
</tr>
<tr>
<td>best_cost = my_cost;</td>
<td>best_cost = my_cost;</td>
</tr>
</tbody>
</table>

- The value of best_cost could be 50 or 75!
- The value 75 does not correspond to any serialization of the two threads.
Critical Section and Mutual Exclusion

- Critical section = a segment that must be executed by only one thread at any time

```c
if (my_cost < best_cost)
    best_cost = my_cost;
```

- Mutex locks protect critical sections in Pthreads
  - locked and unlocked
  - At any point of time, only one thread can acquire a mutex lock

- Using mutex locks
  - request lock before executing critical section
  - enter critical section when lock granted
  - release lock when leaving critical section
Mutual Exclusion using Pthread Mutex

```c
int pthread_mutex_lock (pthread_mutex_t *mutex_lock);
int pthread_mutex_unlock (pthread_mutex_t *mutex_lock);
int pthread_mutex_init (pthread_mutex_t *mutex_lock,
const pthread_mutexattr_t *lock_attr);

pthread_mutex_t cost_lock;

int main() {
    ...
    pthread_mutex_init(&cost_lock, NULL);
    pthread_create(&thhandle1, NULL, find_best, …);
    pthread_create(&thhandle2, NULL, find_best, …);
}

void *find_best(void *list_ptr) {
    ...
    pthread_mutex_lock(&cost_lock); // enter CS
    if (my_cost < best_cost)
        best_cost = my_cost;
    pthread_mutex_unlock(&cost_lock); // leave CS
}
```

**pthread_mutex_lock** blocks the calling thread if another thread holds the lock.

When **pthread_mutex_lock** call returns
1. Mutex is locked, enter CS
2. Any other locking attempt (call to thread_mutex_lock) will cause the blocking of the calling thread

When **pthread_mutex_unlock** returns
1. Mutex is unlocked, leave CS
2. One thread who blocks on thread_mutex_lock call will acquire the lock and enter CS
Role of Synchronization

- Types of Synchronization
  - Mutual Exclusion
  - Event synchronization
    » point-to-point
    » group
    » global (barriers)

- How much hardware support?
  - high-level operations?
  - atomic instructions?
  - specialized interconnect?
Components of a Synchronization Event

- **Acquire method**
  - Acquire right to the synch
    » enter critical section, go past event

- **Waiting algorithm**
  - Wait for synch to become available when it isn’t
  - busy-waiting, blocking, or hybrid

- **Release method**
  - Enable other processors to acquire right to the synch

- **Waiting algorithm is independent of type of synchronization**
  - makes no sense to put in hardware
Strawman Lock Implementation

lock:  ld  R1, mem[cost_lock]
       cmp  R1, #0
       bnz  lock
       st   mem[cost_lock], #1
       ret

unlock: st   mem[cost_lock], #0
         ret

Why doesn’t the acquire method work?

For example: when two threads (cores) try to acquire the lock, they both execute the ld instruction when the mem[cost_lock] = 0
Atomic Instructions

Exchange data between register and memory atomically

- Specifies a location, register, & atomic operation
  - 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
  - Value in location read into a specified register
  - Constant 1 stored into location
  - Successful if value loaded into register is 0
  - Other constants could be used instead of 1 and 0

- How to implement test&set in distributed cache coherent machine?
  - Wait until have write privileges, then perform operation without allowing any intervening operations (either locally or remotely)
T&S Lock Microbenchmark: SGI Chal.

lock:
    t&s R1, mem[cost_location]
    bnz lock /* if not 0, try again */
    ret /* return control to caller */
unlock:
    st mem[cost_location], #0/* write 0 to location */
    ret /* return control to caller */
Choices of Hardware Primitives for Synchronizations -- 1

- **Test&Set**
  ```c
  test&set (&address) {
    result = M[address];
    M[address] = 1;
    return result;
  }
  ```

- **Swap**
  ```c
  swap (&address, register) { /* x86 */
    temp = M[address];
    M[address] = register;
    register = temp;
  }
  ```
Choices of Hardware Primitives for Synchronizations -- 2

- **Compare and Swap (CAS)**
  
  ```c
  compare&swap (&address, reg1, reg2) {
    if (reg1 == M[address]) {
      M[address] = reg2;
      return success;
    } else {
      return failure;
    }
  }
  ```

- **Load-linked and Store-conditional**
  
  ```c
  load-linked&store conditional(&address) {
    loop:
    ll r1, M[address];
    movi r2, 1; /* Can do arbitrary comp */
    sc r2, M[address];
    beqz r2, loop;
  }
  ```

  [https://gcc.gnu.org/onlinedocs/gcc-4.1.0/gcc/Atomic-Builtins.html](https://gcc.gnu.org/onlinedocs/gcc-4.1.0/gcc/Atomic-Builtins.html)
Improved Hardware Primitives: LL-SC

- **Goals:**
  - Test with reads
  - Failed read-modify-write attempts don’t generate invalidations
  - Nice if single primitive can implement range of r-m-w operations

- **Load-Locked (or -linked), Store-Conditional**
  - LL reads variable into register
  - Follow with arbitrary instructions to manipulate its value
  - SC tries to store back to location
  - succeed if and only if no other write to the variable since this processor’s LL
    » indicated by condition codes;

- If SC succeeds, all three steps happened atomically
- If fails, doesn’t write or generate invalidations
  - must retry acquire
Simple Lock with LL-SC

```assembly
lock:    ll    R1, mem[cost_lock] /* LL location to reg1 */
        sc    mem[cost_lock], R2 /* SC reg2 into location */
        beqz R2, lock          /* if failed, start again */
        ret

unlock: st    mem[cost_lock], #0 /* write 0 to location */
        ret
```

- Can do more fancy atomic ops by changing what’s between LL & SC
  - But keep it small so SC likely to succeed
  - Don’t include instructions that would need to be undone (e.g. stores)

- SC can fail (without putting transaction on bus) if:
  - Detects intervening write even before trying to get bus
  - Tries to get bus but another processor’s SC gets bus first

- LL, SC are not lock, unlock respectively
  - Only guarantee no conflicting write to lock variable between them
  - But can use directly to implement simple operations on shared variables
Cost of Atomic and Hardware Locks

- Expensive, e.g. X86 lock could cause multiple 100 cycles

  "lock cmpxchg [rsp - 8], rdx" (both with comparison match and mismatch),
  "lock xadd [rsp - 8], rdx",
  "lock bts qword ptr [rsp - 8], 1"

- Hardware atomic increment/decrement could cost multiple 10 cycles

- Because it may involve locking the memory bus so no others can use
Mini-Instruction Set debate

- atomic read-modify-write instructions
  - IBM 370: included atomic compare&swap for multiprogramming
  - x86: any instruction can be prefixed with a lock modifier
  - High-level language advocates want hardware locks/barriers
    » but it’s goes against the “RISC” flow, and has other problems
  - SPARC: atomic register-memory ops (swap, compare&swap)
  - MIPS, IBM Power: no atomic operations but pair of instructions
    » load-locked, store-conditional
    » later used by PowerPC and DEC Alpha too
  - 68000: CCS: Compare and compare and swap
    » No-one does this any more

- Rich set of tradeoffs
Busy-wait Lock

- Also called spin lock
  - Keep trying to acquire lock until read
  - Very low latency/processor overhead!
  - Very high system overhead!
    » Causing stress on network while spinning
    » Processor is not doing anything else useful

lockit:  DADDUI R2, R0, #1
         EXCH R2, 0(R1)  ;atomic exchange
         BNEZ R2, lockit  ;already locked?

Spinning on memory read
Busy-wait Lock Leveraging Cache Coherence

- Also called spin lock
  - Keep trying to acquire lock until read
  - Very low latency/processor overhead!
  - Very high system overhead!

lockit: LD R2, 0(R1) ;load of lock
       BNEZ R2, lockit ;not available-spin
       DADDUI R2, R0, #1 ;load locked value
       EXCH R2, 0(R1) ;swap
       BNEZ R2, lockit ;branch if lock wasn’t 0

Spinning on cache read until cache miss (because of invalidation)
Busy-wait Lock Leveraging Cache Coherence

lockit: LDR2,0(R1) ; load of lock
BNEZR2,lockit ; not available-spin
DADDUIR2,R0,#1 ; load locked value
EXCHR2,0(R1) ; swap
BNEZR2,lockit ; branch if lock wasn't 0

<table>
<thead>
<tr>
<th>Step</th>
<th>P0</th>
<th>P1</th>
<th>P2</th>
<th>Coherence state of lock at end of step</th>
<th>Bus/directory activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Has lock</td>
<td>Begins spin, testing if lock = 0</td>
<td>Begins spin, testing if lock = 0</td>
<td>Shared</td>
<td>Cache misses for P1 and P2 satisfied in either order. Lock state becomes shared.</td>
</tr>
<tr>
<td>2</td>
<td>Set lock to 0</td>
<td>(Invalidate received)</td>
<td>(Invalidate received)</td>
<td>Exclusive (P0)</td>
<td>Write invalidate of lock variable from P0.</td>
</tr>
<tr>
<td>3</td>
<td>Cache miss</td>
<td>Cache miss</td>
<td>Shared</td>
<td></td>
<td>Bus/directory services P2 cache miss; write-back from P0; state shared.</td>
</tr>
<tr>
<td>4</td>
<td>(Waits while bus/ directory busy)</td>
<td>Lock = 0 test succeeds</td>
<td>Shared</td>
<td></td>
<td>Cache miss for P2 satisfied</td>
</tr>
<tr>
<td>5</td>
<td>Lock = 0</td>
<td>Executes swap, gets cache miss</td>
<td>Shared</td>
<td></td>
<td>Cache miss for P1 satisfied</td>
</tr>
<tr>
<td>6</td>
<td>Executes swap, gets cache miss</td>
<td>Completes swap: returns 0 and sets lock = 1</td>
<td>Exclusive (P2)</td>
<td></td>
<td>Bus/directory services P2 cache miss; generates invalidate; lock is exclusive.</td>
</tr>
<tr>
<td>7</td>
<td>Swap completes and returns 1, and sets lock = 1</td>
<td>Enter critical section</td>
<td>Exclusive (P1)</td>
<td></td>
<td>Bus/directory services P1 cache miss; sends invalidate and generates write-back from P2.</td>
</tr>
<tr>
<td>8</td>
<td>Spins, testing if lock = 0</td>
<td></td>
<td></td>
<td>None</td>
<td></td>
</tr>
</tbody>
</table>
Busy-wait vs Blocking Lock

- **Busy-wait: i.e. spin lock**
  - Keep trying to acquire lock until read
  - Very low latency/processor overhead!
  - Very high system overhead!
    » Causing stress on network while spinning
    » Processor is not doing anything else useful

- **Blocking:**
  - If can’t acquire lock, deschedule process (i.e. unload state)
  - Higher latency/processor overhead (1000s of cycles?)
    » Takes time to unload/restart task
    » Notification mechanism needed
  - Low system overhead
    » No stress on network
    » Processor does something useful

- **Hybrid:**
  - Spin for a while, then block
  - 2-competitive: spin until have waited blocking time
Blocking Lock

- while (!finished()) cpu_pause();
  - Pause CPU so not consuming cycles/energy, but still occupying the CPU
  - ISA support

- while (!finished()) sched_yield();
  - Yield the CPU from the kernel, i.e. give up the slice in time-sharing
  - API/Kernel support

- mutex_wait() and mutex_wake()
  - Completely surrender the CPU for doing other work
  - API/Kernel support via pthread_cond_t and pthread_mutex_t
    » Pthread Condition variable and mutex
Lock-Free Synchronization

- What happens if process grabs lock, then goes to sleep???
  - Page fault
  - Processor scheduling
  - Etc

- Lock-free synchronization:
  - Operations do not require mutual exclusion of multiple insts

- Nonblocking:
  - Some process will complete in a finite amount of time even if other processors halt

- Wait-Free (Herlihy):
  - Every (nonfaulting) process will complete in a finite amount of time

- Systems based on LL&SC can implement these
Synchronization Summary

- Rich interaction of hardware-software tradeoffs
- Must evaluate hardware primitives and software algorithms together
  - primitives determine which algorithms perform well
- Evaluation methodology is challenging
  - Use of delays, microbenchmarks
  - Should use both microbenchmarks and real workloads
- Simple software algorithms with common hardware primitives do well on bus
  - Will see more sophisticated techniques for distributed machines
  - Hardware support still subject of debate
- Theoretical research argues for swap or compare&swap, not fetch&op
  - Algorithms that ensure constant-time access, but complex
One of the most confusing topics (if not the most) in computer system, parallel programming and parallel computer architecture
Setup for Mem. Consistency

- Coherence ⇒ Writes to a location become visible to all in the same order
  - But when does a write become visible?
  - Immediately or visible when needed?

- How do we establish orders between a write and a read by different processors?
  - use event synchronization

- Typically use more than one location!
Under cache coherence, if write is immediately available
- Not possible for both L1 and L2 to be true

For cache-coherent systems, if write invalidates are delayed and processor allows to continue under delay
- It is possible for both L1 and L2 to be true
Another Example of Ordering?

- **What’s the intuition?**
  - Whatever it is, we need an ordering model for clear semantics
    - across different locations as well
    - so programmers can reason about what results are possible

  ```plaintext
  P_1                         P_2
  /*Assume initial values of A and B are 0*/
  (1a) A = 1; (2a) print B;
  (1b) B = 2; (2b) print A;
  ```

- Expect memory to respect order between accesses to different locations issued by a given process
  - to preserve orders among accesses to same location by different processes

- Coherence is not enough!
  - pertains only to single location
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

- Implications for both programmer and system designer
  - Programmer uses to reason about correctness and possible results
  - System designer can use to constrain how much accesses can be reordered by compiler or hardware

- Contract between programmer and system
Sequential Consistency

- Memory operations from a proc become visible (to itself and others) in program order

- There exists a total order, consistent with this partial order - i.e., an interleaving
  - the position at which a write occurs in the hypothetical total order should be the same with respect to all processors

- Said another way:
  - For any possible individual run of a program on multiple processors
  - Should be able to come up with a serial interleaving of all operations that respects
    - Program Order
    - Read-after-write orderings (locally and through network)
    - Also Write-after-read, write-after-write
Sequential Consistency

- Total order achieved by *interleaving* accesses from different processes
  - Maintains *program order*, and memory operations, from all processes, appear to [issue, execute, complete] atomically w.r.t. others
  - as if there were no caches, and a single memory

- “A multiprocessor 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.”
  [Lamport, 1979]
SC Example

- What matters is order in which operations appear to execute, not the chronological order of events.
- Possible outcomes for (A,B): (0,0), (1,0), (1,2).
- What about (0,2)?
  - Program order => 1a->1b and 2a->2b.
  - A = 0 implies 2b->1a, which implies 2a->1b.
  - B = 2 implies 1b->2a, which leads to a contradiction (cycle!).
- Since there is a cycle => no sequential order that is consistent!

/* Assume initial values of A and B are 0 */

P1
---
(1a) A = 1;
(1b) B = 2;

P2
---
(2a) print B;
(2b) print A;

A = 0
B = 2

A = 0
B = 2
Implementing SC

- Two kinds of requirements
  - **Program order**
    » memory operations issued by a process must appear to execute (become visible to others and itself) in program order
  - **Atomicity**
    » in the overall hypothetical total order, one memory operation should appear to complete with respect to all processes before the next one is issued
    » guarantees that total order is consistent across processes
    - tricky part is making writes atomic

- How can compilers violate SC?
  - **Architectural enhancements?**
Sequential Consistency

- Bus imposes total order on xactions for all locations
- Between xactions, procs perform reads/writes (locally) in program order
- So any execution defines a natural partial order
  - $M_j$ subsequent to $M_i$ if
    - (i) $M_j$ follows $M_i$ in program order on same processor,
    - (ii) $M_j$ generates bus xaction that follows the memory operation for $M_i$
- In segment between two bus transactions, any interleaving of local program orders leads to consistent total order
- Within segment writes observed by proc P serialized as:
  - Writes from other processors by the previous bus xaction P issued
  - Writes from P by program order
  - Insight: only one cache may have value in “M” state at a time…
Sufficient conditions

- **Sufficient Conditions**
  - issued in program order
  - after write issues, the issuing process waits for the write to complete before issuing next memory operation
  - after read is issues, the issuing process waits for the read to complete and for the write whose value is being returned to complete (globally) before issuing its next operation

- **Write completion**
  - can detect when write appears on bus (flush) appears

- **Write atomicity:**
  - if a read returns the value of a write, that write has become visible to all others already
    - Either: it is being read by the processor that wrote it and no other processor has a copy (thus any read by any other processor will get new value via a flush
    - Or: it has already been flushed back to memory and all processors will have the value