Synchronization in Shared Memory

Kathy Yelick
yelick@cs.berkeley.edu
www.cs.berkeley.edu/~yelick/cs194f07
Lecture Schedule for Next Two Weeks

• Fri 9/14    11-12:00
• Mon 9/17    10:30-11:30 Discussion (11:30-12 as needed)
• Wed 9/19    10:30-12:00 Decomposition and Locality
• Fri 9/21    10:30-12:00 NVIDIA Lecture
• Mon 9/24    10:30-12:00 NVIDIA lecture
• Tue 9/25    3:00-4:30 NVIDIA research talk in Woz (optional)
• Wed 9/26    10:30-11:30 Discussion
• Fri 9/27    11-12 Lecture topic TBD

• Why? Opportunity to hear David Kirk from NVIDIA on the CUDA processor/programming system
• I’m away at CSTB meeting (Sustaining the growth) and Petascale meeting (10^15) and Exascale (10^18) and …
162 Review: Multiprocessing / Multiprogramming

- What does it mean to run two threads “concurrently”?

- Why does this matter?
  - With multiprogramming (single processor), no true parallelism
    - Locks can be implemented by preventing context switches during to other threads
  - In a multiprocessing environment (without multiprogramming), you *may* have control over processor resources
    - Tying up processor (spinlocks) while waiting *may* be practical
162 Review: Atomic Operations

• To understand a concurrent program, we need to know what the underlying indivisible operations are!

• Atomic Operation: an operation that always runs to completion or not at all
  • It is indivisible: it cannot be stopped in the middle and state cannot be modified by someone else in the middle
  • Fundamental building block – if no atomic operations, then have no way for threads to work together

• On most machines, memory references and assignments (i.e. loads and stores) of words are atomic

• Many instructions are not atomic
  • VAX and IBM 360 had an instruction to copy a whole array

• See CS162 lectures slides on synchronization for more.
Role of Synchronization

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

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

• How much hardware support?
  • high-level operations?
  • atomic instructions?
  • specialized interconnect?
Mini-Instruction Set debate

• atomic read-modify-write instructions
  • IBM 370: 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

• Rich set of tradeoffs
Strawman Lock

lock:  ld  register, location  /* copy location to register */
cmp location, #0  /* compare with 0 */
bnz 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 */

Does the acquire method work?
Release method?
Simple Test&Set Lock

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

- Other read-modify-write primitives
  - Swap
  - Fetch&op
  - Compare&swap
    - Three operands: location, register to compare with, register to swap with
    - Not commonly supported by RISC instruction sets
Performance Criteria for Synch. Ops

- Latency (time per op)
  - How long does it take if you always win
  - Especially when light contention
- Bandwidth (ops per sec)
  - Especially under high contention
  - How long does it take (averaged over threads) when many others are trying for it
- Traffic
  - How many events on shared resources (bus, crossbar,...)
- Storage
  - How much memory is required?
- Fairness
  - Can any one threads be “starved” and never get the lock?
T&S Lock Microbenchmark: SGI Chal.

- Why does performance degrade?
- Bus Transactions on T&S?
- Hardware support in CC protocol?
Enhancements to Simple Lock

- Reduce frequency of issuing test&sets while waiting
  - Test&set lock with backoff
  - Don’t back off too much or will be backed off when lock becomes free
  - Exponential backoff works quite well empirically: $i^{th}$ time = $k \cdot c^i$
- Busy-wait with read operations rather than test&set
  - Test-and-test&set lock
  - Keep testing with ordinary load
    - cached lock variable will be invalidated when release occurs
    - When value changes (to 0), try to obtain lock with test&set
      - only one attempt will succeed; others will start testing again
- Use of other hardware primitives
  - Load-Linked (or –locked) Store Conditional
- Add queues for waiting threads for fairness
Point to Point Event Synchronization

- Producer-consumer patterns of parallelism are common
  - One thread computes a value, other(s) use it

- Software methods:
  - Busy-waiting: use ordinary variables as flags
  - Blocking: use semaphores (see CS162)

- Hardware support: \textit{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)

- Problem: flexiblity
  - multiple consumers, or multiple writes before consumer reads?
  - needs language support to specify when to use
  - composite data structures?

- Not available on most machines; too expensive to implement
Barriers

- Software algorithms implemented using locks, flags, counters
- 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
A Simple Centralized Barrier

- Shared counter maintains number of processes that have arrived
  - increment when arrive (lock), check until reaches numprocs
  - Problem?

```c
struct bar_type {int counter; struct lock_type lock;
                   int flag = 0;} bar_name;

BARRIER (bar_name, p) {
    LOCK(bar_name.lock);
    if (bar_name.counter == 0)
        bar_name.flag = 0; /* reset flag if first to reach*/
    mycount = bar_name.counter++;
    /* mycount is private */
    UNLOCK(bar_name.lock);
    if (mycount == p) {
        bar_name.counter = 0; /* reset for next barrier */
        bar_name.flag = 1; /* release waiters */
    }
    else while (bar_name.flag == 0) {}; /* busy wait for release */
}
```
A Working Centralized Barrier

- Consecutively entering the same barrier doesn’t work
  - Must prevent process from entering until all have left previous instance
  - Could use another counter, but increases latency and contention
- Sense reversal: wait for flag to take different value consecutive times
  - Toggle this value only when all processes reach

```c
BARRIER (bar_name, p) {
    local_sense = !(local_sense); /* toggle private sense variable */
    LOCK(bar_name.lock);
    mycount = bar_name.counter++; /* mycount is private */
    if (bar_name.counter == p)
        UNLOCK(bar_name.lock);
        bar_name.flag = local_sense; /* release waiters*/
    else
        { UNLOCK(bar_name.lock);
        while (bar_name.flag != local_sense) {}}; }
}
Centralized Barrier Performance

- Latency
  - Centralized has critical path length at least proportional to $p$
- Traffic
  - About $3p$ bus transactions
- Storage Cost
  - Very low: centralized counter and flag
- Fairness
  - Same processor should not always be last to exit barrier
  - No such bias in centralized
- Key problems for centralized barrier are latency and traffic
  - Especially with distributed memory, traffic goes to same node
Improved Barrier Algorithms for a Bus

Software combining tree

- Only $k$ processors access the same location, where $k$ is degree of tree

- Separate arrival and exit trees, and use sense reversal
- Valuable in distributed network: communicate along different paths
- On bus, all traffic goes on same bus, and no less total traffic
- Higher latency ($\log p$ steps of work, and $O(p)$ serialized bus xactions)
- Advantage on bus is use of ordinary reads/writes instead of locks
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