# CS184c: Computer Architecture [Parallel and Multithreaded]

Day 10: May 8, 2001
Synchronization




## Today

- Synchronization
  - Primitives
  - Algorithms
  - Performance
  - coarse vs. fine grained

## Problem

- If correctness requires an ordering between threads,
  - have to enforce it
- Was not a problem we had in the single-thread case
  - does occur in the multiple threads on single processor case



## Desired Guarantees

- Precedence
  - barrier synchronization
  - producer-consumer
- Atomic Operation Set
- Mutual exclusion

## Read/Write Locks?

- Try implement lock with r/w:

```
while (~done)
if (~A.lock)
      A.lock=true
      do stuff
      A.lock=false
      done=true
```



## Problem with R/W locks?

- Consider context switch between test (~A.lock=true?) and assignment (A.lock=true)

## Primitive Need

- Indivisible primitive to enabled atomic operations



## Original Examples

- Test-and-set
  - combine test of A.lock and set into single atomic operation
  - once have lock
    - can guarantee mutual exclusion at higher level
- Read-Modify-Write
  - atomic read…write sequence
- Exchange

# Examples (cont.)

- Exchange
  - Exchange true with A.lock
  - if value retrieved was false
    - this process got the lock
  - if value retrieved was true
    - already locked
    - (didn't change value)
    - keep trying
  - key is, only single exchanger get the false value



# Implementing...

- What required to implement?
  - Uniprocessor
  - Bus-based
  - Distributed

## Implement: Uniprocessor

- Prevent Interrupt/context switch
- Primitives use single address
  - so page fault at beginning
  - then ok, to computation (defer faults…)
- SMT?



## Implement: Snoop Bus

- Need to reserve for Write
  - write-through
    - hold the bus between read and write
  - write-back
    - need exclusive read
    - and way to defer other writes until written

## Implement: Distributed

- Can't lock down bus
- Exchange at memory controller?
  - Invalidate copies (force writeback)
  - after settles, return value and write new
  - don't service writes until complete



## Performance Concerns?

- Locking resources reduce parallelism
- Bus (network) traffic
- Processor utilization
- Latency of operation

## Basic Synch. Components

- Acquisition
- Waiting
- Release



## Possible Problems

- Spin wait generates considerable memory traffic
- Release traffic
- Bottleneck on resources
- Invalidation
  - can't cache locally…
- Fairness

# Test-and-Set

```
Try: t&s R1, A.lock
     bnz R1, Try
     return
```

- Simple algorithm generate considerable traffic
- p contenders
  - p try first, 1 wins
  - for o(1) time p-1 spin
  - …then p-2…
  - c*(p+p-1+p-2,,,)
  - $O(p^2)$



# Test-test-and-Set

```
Try:  ld R1, A.lock
      bnz R1, Try
      t&s R1, A.lock
      bnz R1, Try
      return
```

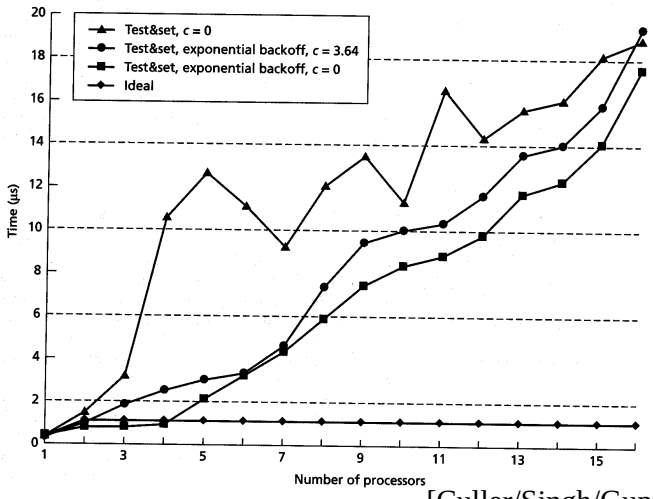
- Read can be to local cache
- Not generate bus traffic
- Generates less contention traffic

# Backoff

- Instead of immediately retrying
  - wait some time before retry
  - reduces contention
  - may increase latency
    - (what if I'm only contender and is about to be released?)



# Primitive Bus Performance




[Culler/Singh/Gupta 5.29]

## Bad Effects

- Performance Decreases with users
  - From growing traffic already noted



## Detecting atomicity sufficient

- Fine to detect if operation will appear atomic
- Pair of instructions
  - ll -- load locked
    - load value and mark in cache as locked
  - sc -- store conditional
    - stores value iff no intervening write to address
    - e.g. cache-line never invalidated by write

## LL/SC operation

```
Try: LL R1 A.lock
     BNZ R1, Try
     SC   R2, A.lock
     BEQZ Try
     return from lock
```



## LL/SC

- Pair doesn't really lock value
- Just detects if result would appear that way
- Ok to have arbitrary interleaving between LL and SC
- Ok to have capacity eviction between LL and SC
  - will just fail and retry

## LL/SC and MP Traffic

- Address can be cached
- Spin on LL not generate global traffic (everyone have their own copy)
- After write (e.g. unlock)
  - everyone miss -- O(p) message traffic
- No need to lock down bus during operation



## Ticket Synchronization

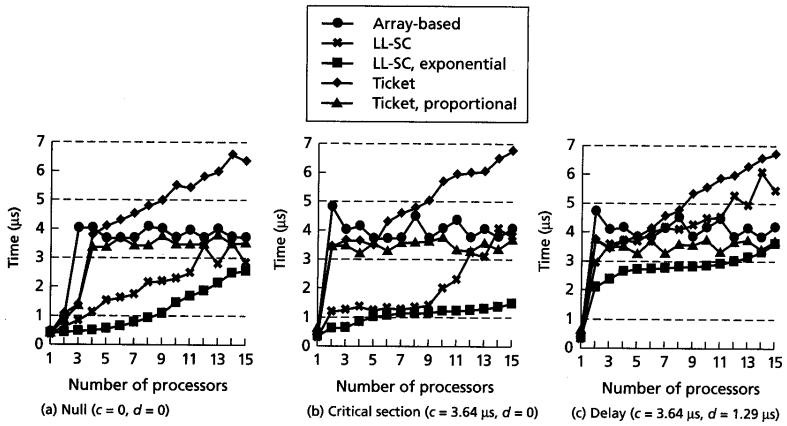
- Separate counters for place in line and current owner
- Use ll/sc to implement fetch-and-increment on position in line
- Simple read current owner until own number comes up
- Increment current owner when done
- Provides FIFO service (fairness)
- O(p) reads on change like ll/sc
- Chance to backoff based on expected wait time

# Array Based

- Assign numbers like Ticket
- But use numbers as address into an array of synchronization bits
- Each queued user spins on **different** location
- Set next location when done
- Now only O(1) traffic per invalidation



# Performance Bus



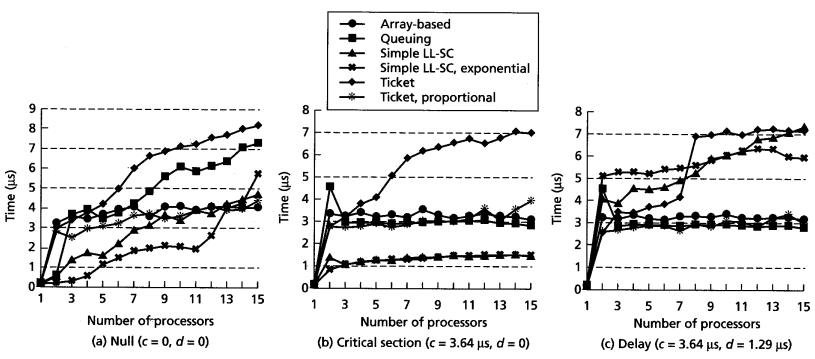

[Culler/Singh/Gupta 5.30]

# Queuing

- Like Array, but use queue
- Atomicly splice own synchronization variable at end of queue
- Can allocate local to process
- Spin until predecessor done
  - and splices self out



# Performance Distributed




[Culler/Singh/Gupta 8.34]

## Barrier Synchronization

- Guarantee all processes rendezvous at point before continuing
- Separate phases of computation



## Simple Barrier

- Fetch-and-Decrement value
- Spin until reaches zero
- If reuse same synchronization variable
  - will have to take care in reset
  - one option: invert sense each barrier

## Simple Barrier Performance

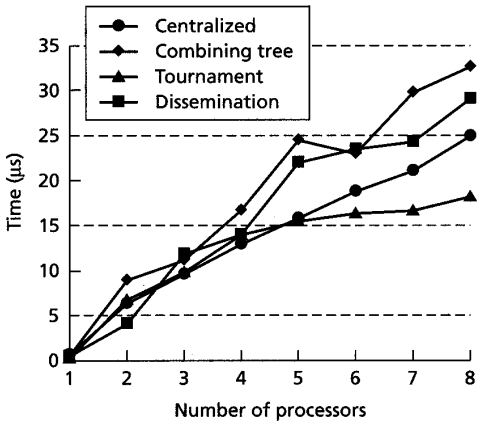
- Bottleneck on synchronization variable
- $O(p^2)$ traffic spinning
- Each decrement invalidates cached version



## Combining Trees

- Avoid bottleneck by building tree
  - fan-in and fan-out
- Small (constant) number of nodes rendezvous on a location
- Last arriver synchronizes up to next level
- Exploit disjoint paths in scalable network
- Spin on local variables
- Predetermine who passes up
  - "Tournament"

# Simple Bus Barrier




[Culler/Singh/Gupta 5.31]



# Coarse vs. Fine-Grained...

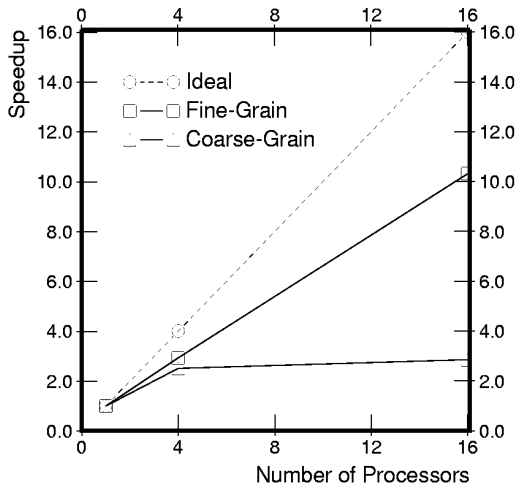
- Barriers are coarse-grained synchronization
  - all processes rendezvous at point
- Full-empty bits are fine-grained
  - synchronize each consumer with producer
  - expose more parallelism
  - less false waiting
  - many more synchronization events
    - and variables

# Alewife / Full Empty

- Experiment to see impact of synchronization granularity
- Conjugate Gradient computation
- Barriers vs. full-empty bits
- [Yeung and Agarwal PPoPP'93]



# Overall Impact



## Alewife provides

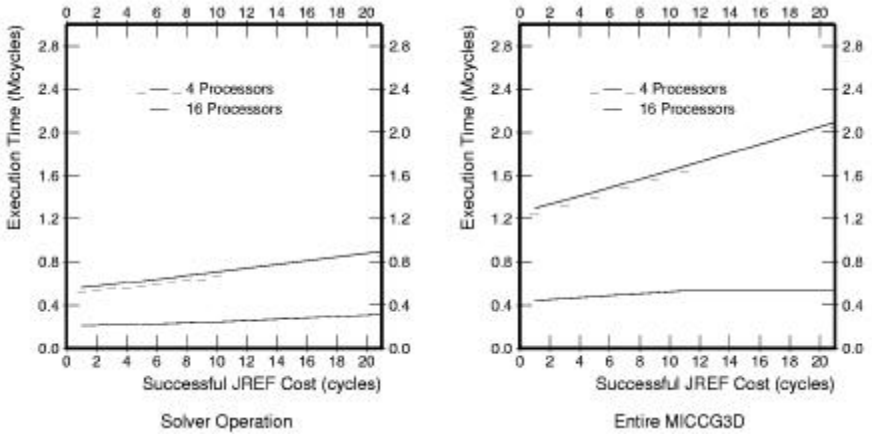
- Ability to express fine-grained synchronization with J-structures
- Efficient data storage (in directory)
- Hardware handling of data presence
  - so like memory op in common case that data is available



## Breakdown benefit

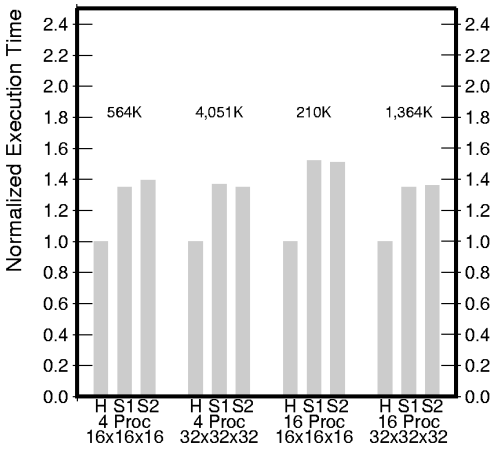
- How much of benefit from each?
  - Expressiveness
  - Memory efficiency
  - Hardware support

# Impact of Hardware Synch.



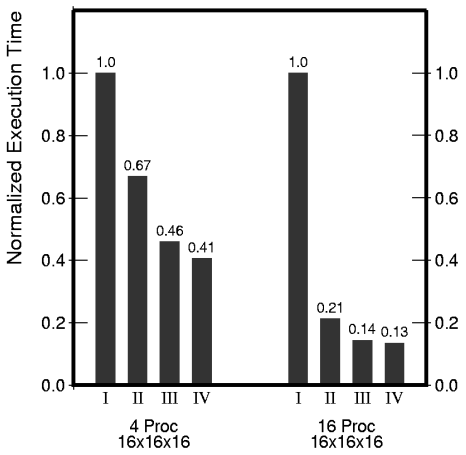



# Impact of Compact Memory



# Overall Contribution

II expression only
III + memory
full-empty
IV + fast bit ops






# Synch. Granularity

- Big benefit from expression
- Hardware can make better
  - but not the dominant effect

# Big Ideas

- Simple primitives
  - Must have primitives to support atomic operations
  - don't have to implement atomicly
    - just detect non-atomicity
- Make fast case common
  - optimize for locality
  - minimize contention



# Big Ideas

- Avoid bottlenecks
  - distribute load/work for scalable performance
    - what spin on for lock
    - combining tree
- Expose parallelism
  - fine-grain expressibility exposes most
  - cost can be manageable