

---

---

Global and high-contention operations:  
Barriers, reductions, and highly-  
contended locks

Katie Coons

April 6, 2006

---

---

# Synchronization Operations

- Locks
- Point-to-point event synchronization
- Barriers
- Global event notification
- Dynamic work distribution

---

---

# Locks - Desirable Characteristics and Potential Tradeoffs

- Low latency to acquire lock
- High bandwidth
- Minimal traffic at all stages
- Low storage cost
- Fairness - FIFO lock granting
- Perform well with distributed memory

---

---

# Test&set Lock

- **Acquire method:** test&set returns 0, sets to 1
- **Waiting algorithm:** spin on test&set until it returns 0
- **Release algorithm:** set to 0

---

---

# Disadvantages

- Excessive traffic
- Unfair
- Separate primitives needed for different operations
- Exponential backoff only helps somewhat

---

---

# Test-and-test&set

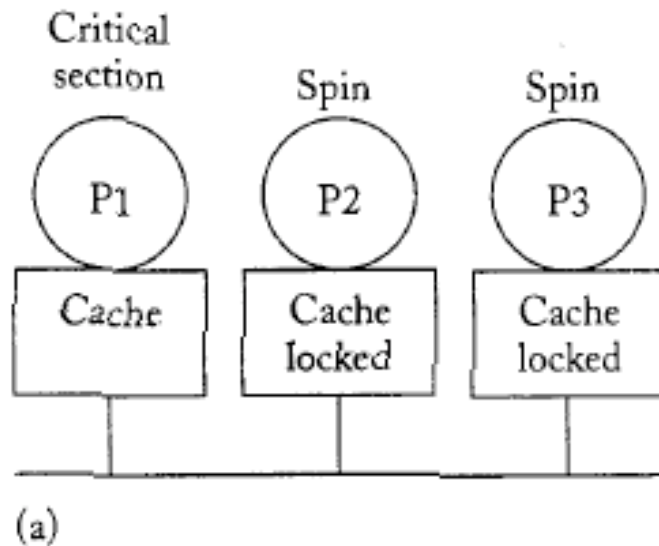
- Spin waiting protocol
- Spins on the read only
- Generates less bus traffic, but still  $O(p^2)$
- Failed attempts generate invalidations

---

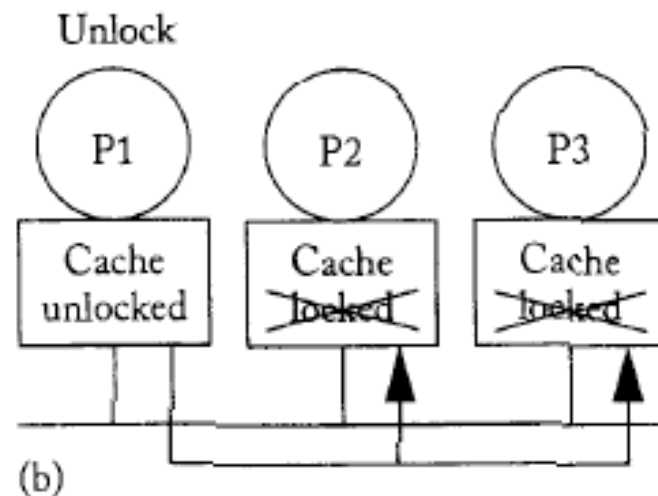
---

# Contended test&set spin locks

P1 holds the lock, P2 and P3 spin on the same variable



P1 releases the lock, P2 and P3 read miss

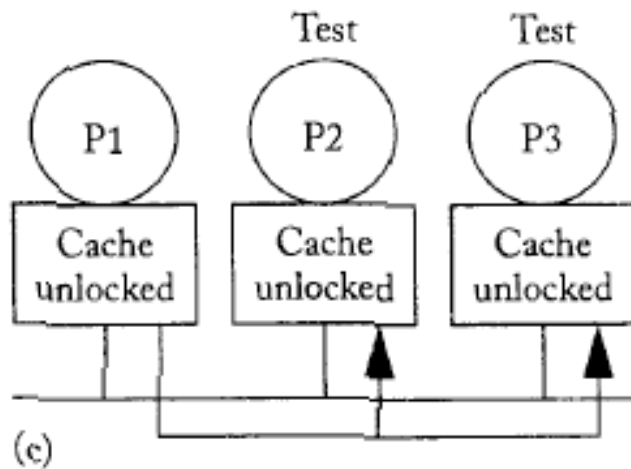


---

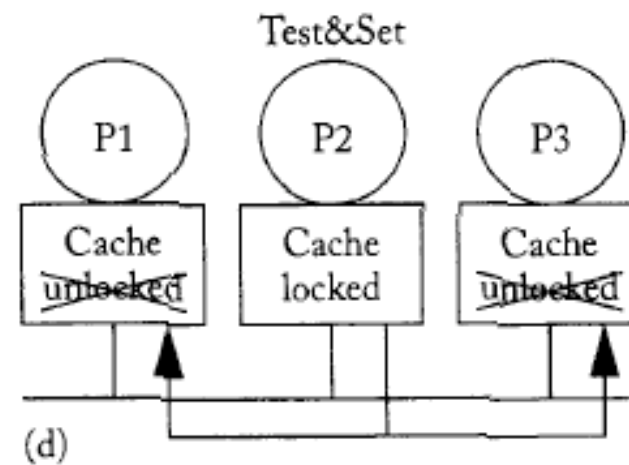
---

# Contended test&set spin locks

P2 and P3 try to reread lock - lock is temporarily unlocked



P2 and P3 attempt to test&set the lock to gain exclusive access

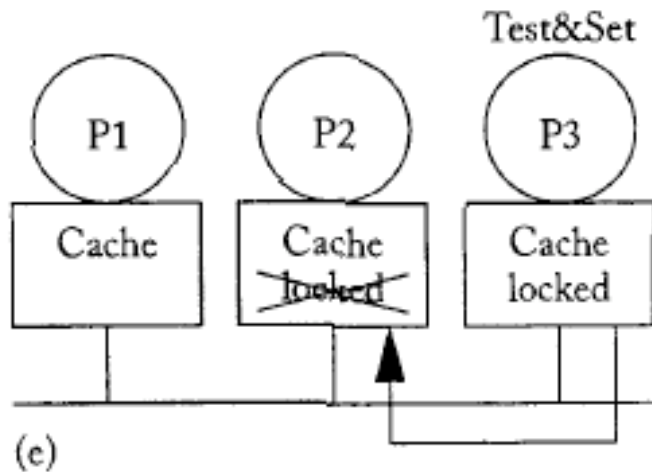


---

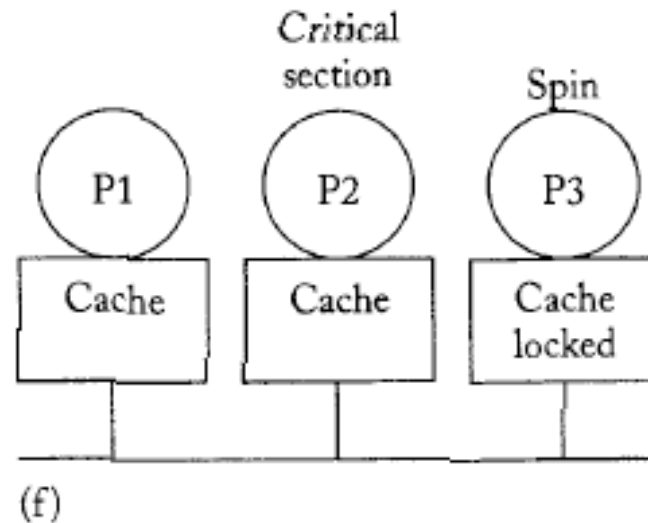
---

# Contended test&set spin locks

Causes additional  
invalidations and cache  
interference



Return to a), but now P2  
has the lock



---

---

# Load-linked, Store-conditional

- LL - Loads variable to register
- SC - Writes register to memory only if no intervening writes to that location occurred
- Together, they implement an atomic r-m-w
- Goals:
  - Test with reads only
  - No invalidations on failure
  - Single primitive for variety of r-m-w operations

---

---

# LL, SC Lock Implementation

```
lock:   ll   r1, location      ; read the value
        bnz  r1, lock         ; loop if not free

        sc   location, #1     ; try to store
        beqz lockit          ; start over if
                                ;   unsuccessful

unlock: st   location, #0     ; release the lock
```

SC fails if

- 1) Detects another write before bus request or
- 2) Loses bus arbitration

---

---

# Load-linked, Store-conditional

- Advantages
  - No bus traffic while spinning
  - Generates no invalidations on store failure
  - Primitive for various operations (test&set, fetch&op, compare&swap)
  - Improved traffic for lock acquisition -  $O(p)$

---

---

# Load-linked, Store-conditional

- Disadvantages
  - Heavy traffic when lock is released.
  - Invalidates caches for all waiting processors
  - $O(p)$  traffic per lock acquisition (could do better)
  - Not fair

---

---

# Contended Locks

- Problem: Release all waiting processors, but only one will get the lock!
- Solution: Notify only one processor

---

---

# Ticket Lock

- Two counters: `next-ticket` and `now-serving`
- Algorithm
  - **Acquire method:** atomic fetch&increment on `next-ticket` provides unique `my-ticket`
  - **Waiting algorithm:** check `now-serving` until it equals `my-ticket`
  - **Release method:** increment `now-serving`

---

---

# Ticket Lock

- Advantages:
  - Decreased traffic on lock release
  - Constant, small storage
  - Fair
  - Low latency with cacheable fetch&increment
- Drawbacks:
  - Traffic still not  $O(1)$  on release

---

---

# Array-Based Lock

- **Acquire method:** atomic fetch&increment provides unique *location* (address)
- **Waiting algorithm:** check *location* for ready, if not ready, check until a read miss occurs
- **Release method:** write to the next location in the array

---

---

# Array-Based Lock

- Advantages:
  - Only one invalidate on a release
  - Fair
  - Similar uncontended latency
  - No backoff needed
- Disadvantages:
  - $O(p)$  rather than  $O(1)$  space
  - Complications with distributed memory

---

---

# Synchronization with Distributed Memory

- Interconnect not centralized
  - Disjoint processors coordinate in parallel
  - Complicates synchronization primitives
- Physically distributed memory
  - Synchronization variable allocation important
  - Varies with cache implementation

---

---

# Synchronization with Distributed Memory

- Memory bandwidth
  - Limits scalability
  - Hot spot references are most severe cause
- Memory latency
  - Limits performance
  - Requires good cache and memory locality

---

---

# Array-Based Locks and Distributed Memory

- Problems
  - $O(p)$  storage
  - Impossible to always spin on local memory
- Spinning on remote locations undesirable
  - Increases traffic
  - Increases contention

---

---

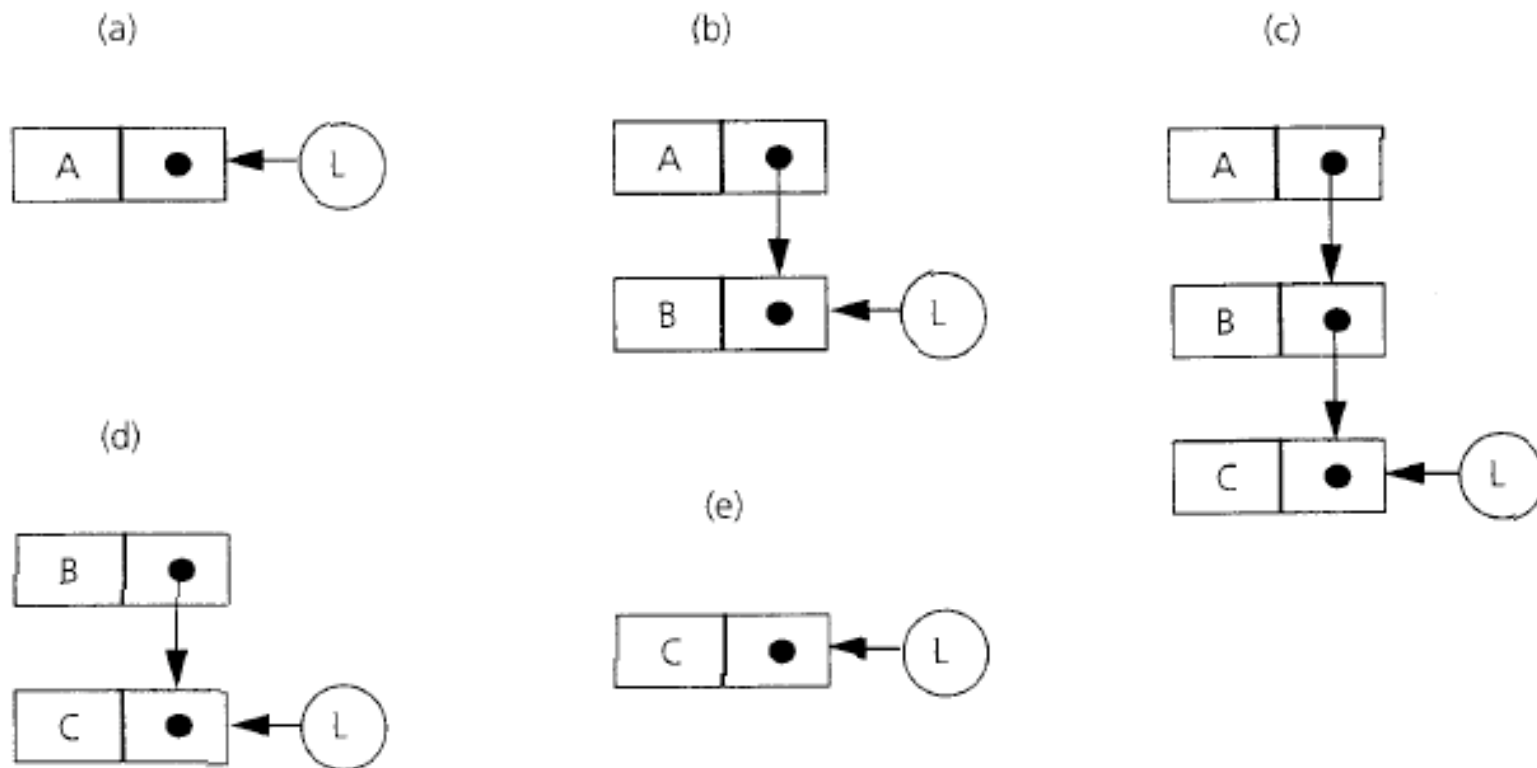
# Software Queuing Lock

- Goals:
  - Reduce space requirements
  - Always spin on locally allocated variables
- Distributed linked list of waiters
- Each node points to following node
- Tail pointer to last waiter

---

---

# Software Queuing Lock



---

---

# Software Queuing Lock

- Atomic changes to tail pointer
- Atomic *fetch&store*
  - Returns current value of 1st operand
  - Sets it to second operand
  - Returns only on success
  - Determines FIFO ordering for acquisition

---

---

# Software Queuing Lock

- Atomic check for last processor
- Atomic *compare&swap*
  - Compares first two operands
  - If equal, set first to third, return true
  - If not equal, return false
  - Difficult to implement (3 operands) - use LL,SC

---

---

# Software Queuing Lock

- Advantages
  - Space proportional to waiting processes
  - FIFO granting order
  - Processes spin on local variables
- Preferred lock for shared address space, distributed memory with no coherent caching

---

---

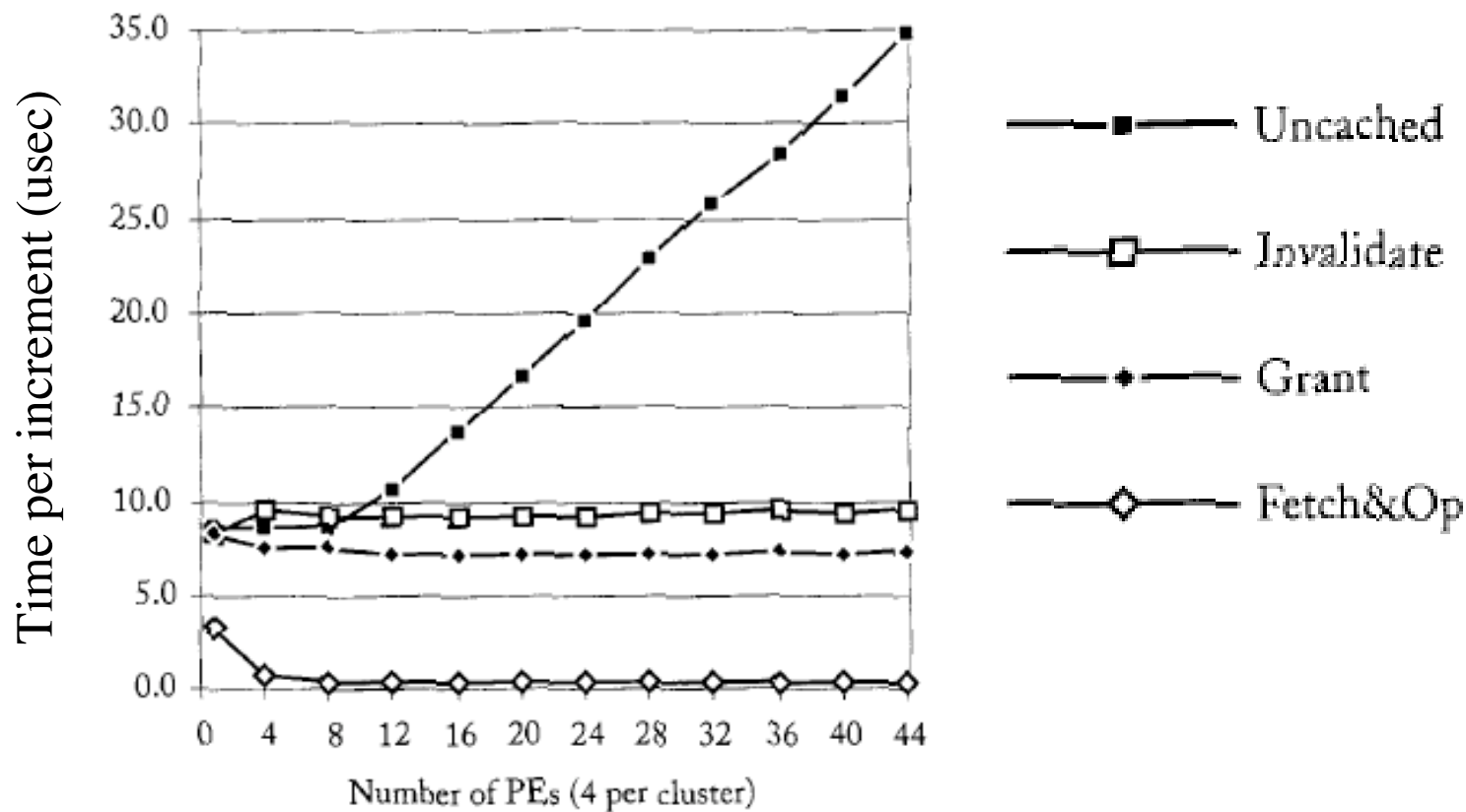
# Queue on Lock Bit (QOLB)

- Hardware primitive
- Incorporated in IEEE SCI protocol
- List of waiters in cache tags of spinning processors
- DASH - directory pointers approximate QOLB waiting list

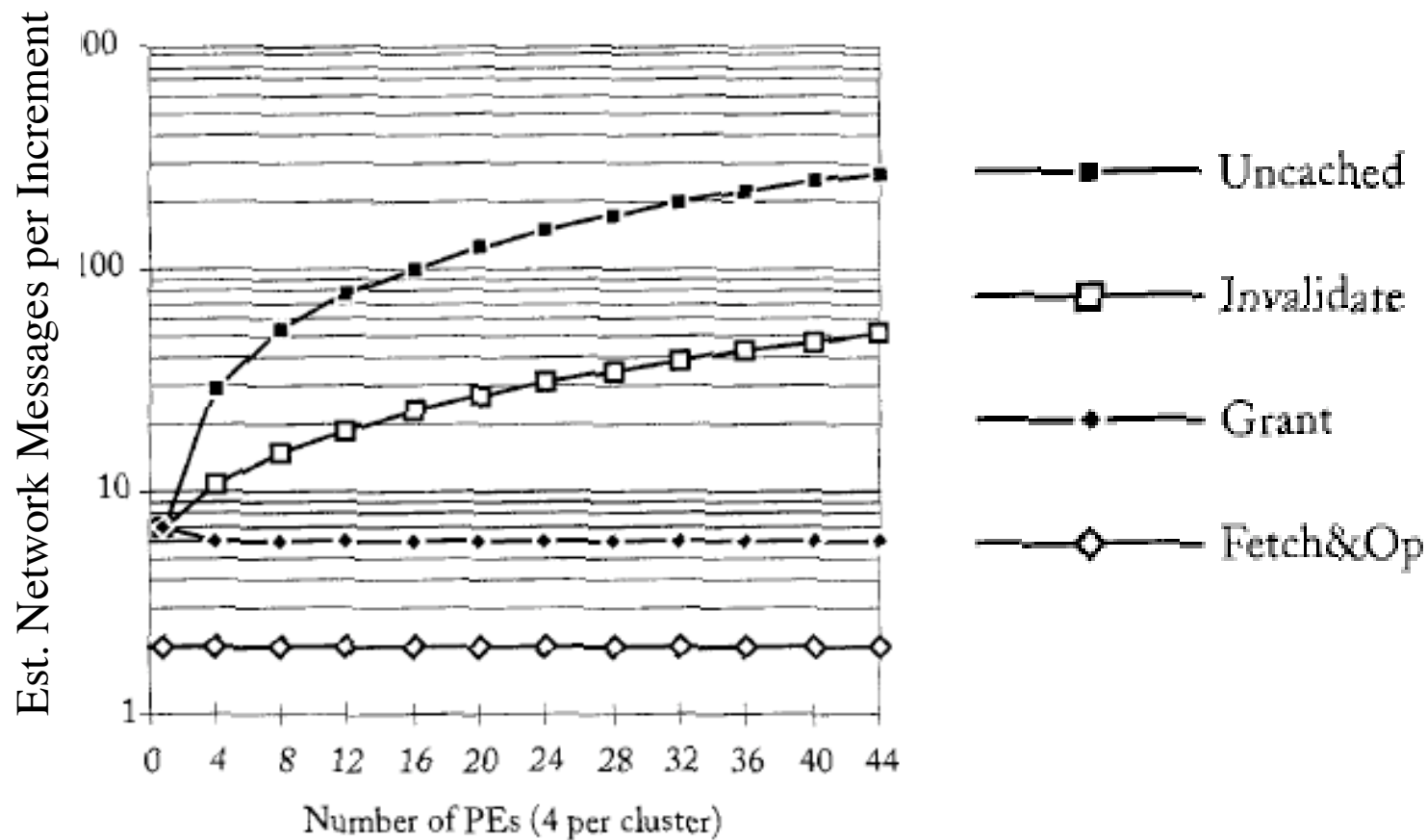
---

---

# Atomic Counter Increment Performance



# Atomic Counter Increment Network Usage



---

---

# Point-to-Point Event Synchronization

- Producer-consumer synchronization
- Software algorithms use flags -  $P_1$  tells  $P_2$  that a value is ready for  $P_2$  to use:

$P_1$

---

```
a = f(x); // set a
flag = 1;
```

$P_2$

---

```
while (flag is 0)
    do nothing;
b = g(a); // use a
```

---

---

## Full-Empty Bits

- Word-level, producer-consumer synchronization:
  - A *full-empty* bit is associated with each word in memory
  - Producer writes only if the *full-empty bit* is empty, and leaves it set to full
  - Consumer reads only if the *full-empty bit* is set to full, and leaves it set to empty

---

---

# Full-Empty Bits

- Advantages:
  - *Full-empty bit* preserves atomicity
  - Hardware support for fine-grained producer-consumer synchronization
- Disadvantages:
  - Inflexible
  - Imposes synchronization on all accesses
  - Hardware cost
- J-machine? M-machine?

---

---

# Global (Barrier) Event Synchronization

- No processes can go beyond the barrier until all processes have reached the barrier
- Arrival
- Wait for release
- Release

---

---

# Centralized Barrier

- Single, shared counter, and flag
- Counter: Number of arrived processes, increment on arrival to get `my-number`
- $p$  = Total number of processes
- If `my-number` =  $p$ , set release flag
- Otherwise, busy-wait on release flag

---

---

# Centralized Barrier

- Inefficient
  - Counter: incremented atomically by each arriving processor
  - Flag: all arrived processors busy-wait on the same flag

Correctness Problem: Consecutively entering the same barrier (use sense reversal)

---

---

---

---

## Centralized Barrier

- Latency: critical path proportional to  $p$
- Traffic: about  $3p$  bus transactions
- Storage: low cost (1 counter, 1 flag)
- Fairness: same processor may always be last to exit the barrier (unfair)
- Key problems: latency and traffic, especially with distributed memory!

---

---

# Barriers and Distributed Memory

- Why do we need better barrier algorithms for distributed memory?
  - Traffic, contention
  - Even bigger problem without cache coherence
  - Parallelization of communication now possible
  - Fine-grained parallelism often means frequent communication and synchronization

---

---

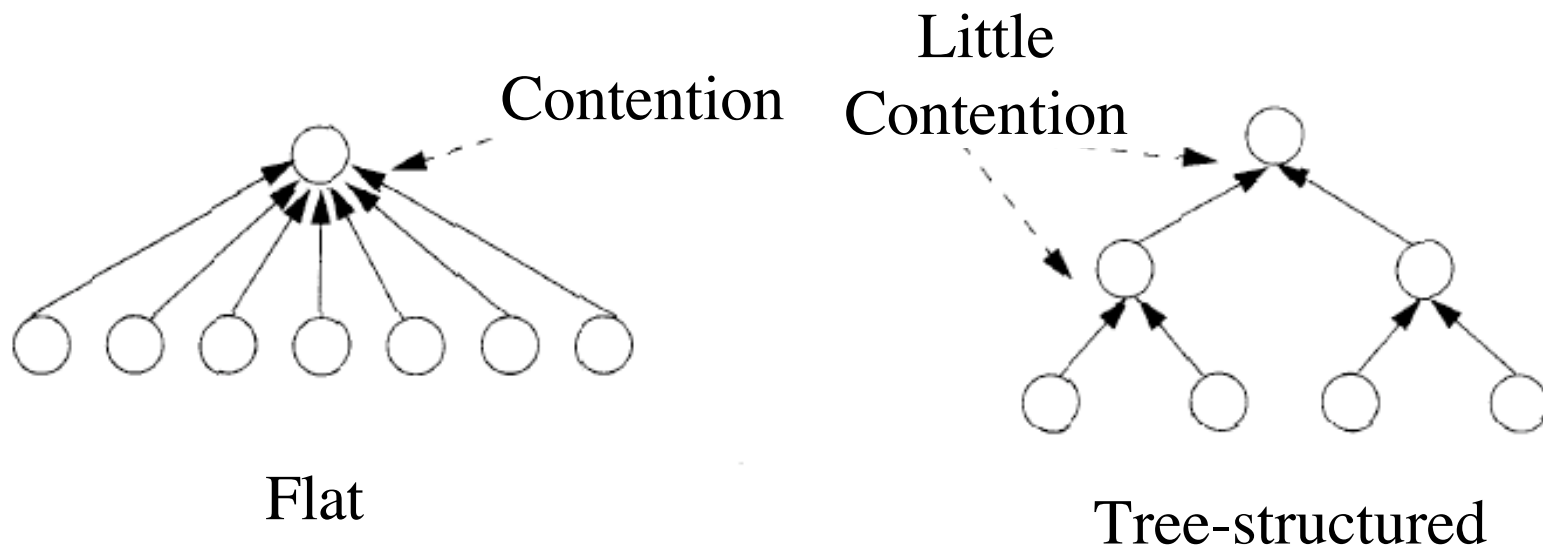
# Barriers and Distributed Memory

- Is special hardware support needed?
    - CM-5, special “control” network for barriers, reductions, broadcasts
    - CRAY T3D, M-machine
    - Potentially significant overhead in a large system
  - Are sophisticated software algorithms enough?
- 
-

---

---

# Software Combining Trees



---

---

# Software Combining Trees

- Same process for release
- Critical path length is  $O(\log_k p)$ 
  - $O(p)$  for centralized barrier
  - $O(p)$  for any barrier on a centralized bus
- Disadvantages:
  - Remote spinning problem
  - Heavy network traffic while spinning

---

---

# Tree Barriers with Local Spinning

- “Tournament barrier”
  - Predetermine which processor moves up
  - The other processor spins on a local variable
- P-node tree
  - A leaf writes to its parent’s arrival array
  - A parent waits for all arrivals, then writes to its parent’s arrival array
  - Separate arrival and release tree ok

---

---

# Tree Barriers with Local Spinning

- Separate arrival, release branching factor
  - Larger branching factor => more contention
  - Smaller branching factor => more network transactions
- Suited to scalable machines without coherent caching

---

---

# Global Event Notification

- Example uses:
  - Producer-consumer synchronization
  - Communicate global data to consumers (new global min/max, for example)
- Invalidation-based coherence - sufficient for low-frequency writes
- Update protocol - reduces communication latency, prevents remote read misses for consuming processors

---

---

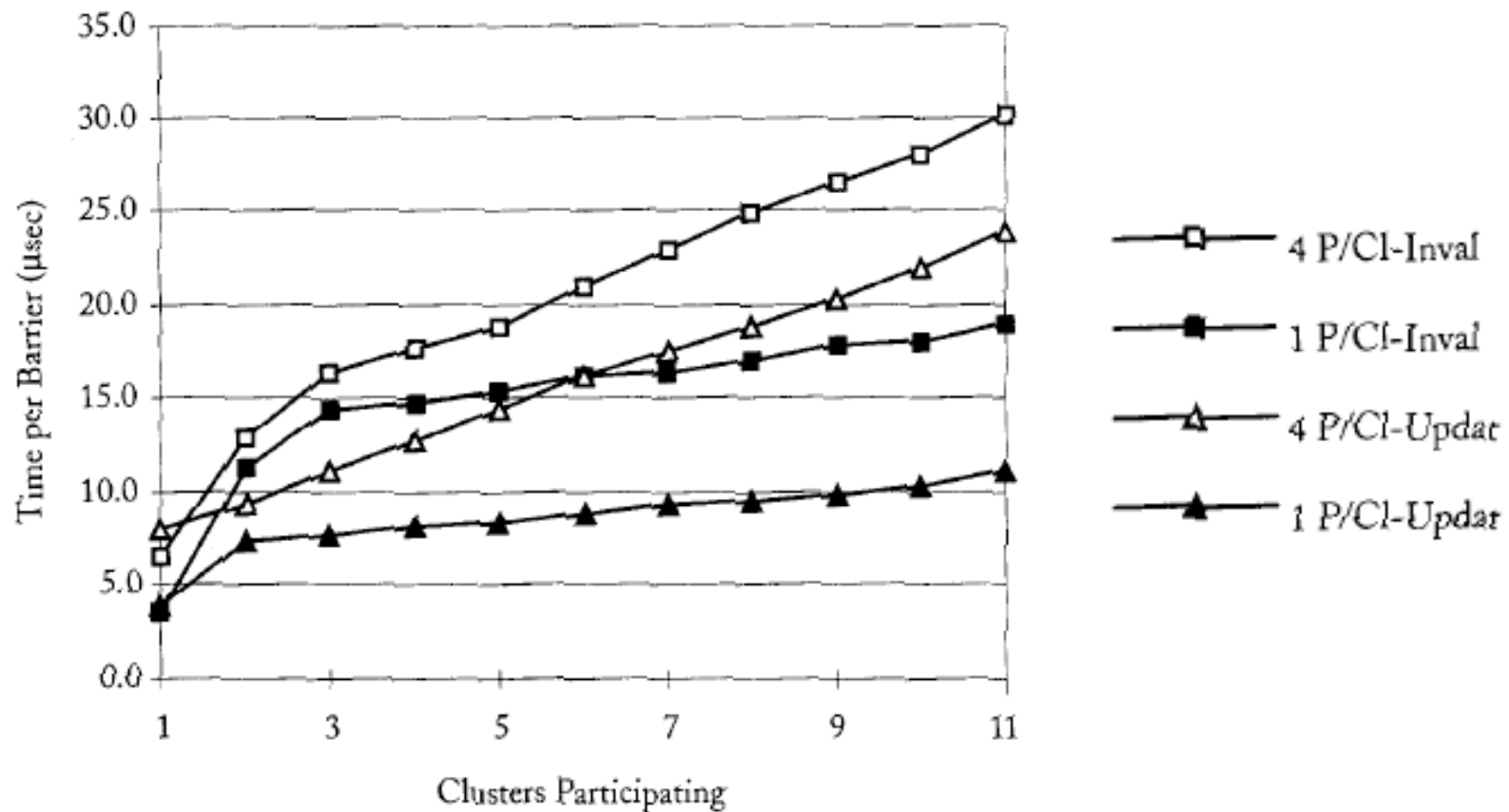
# Update-writes

- Consumer doesn't fetch data from producers cache
- Used for:
  - Small data items (coherence messages per word, not per cache line)
  - Items the consumer already has cached
  - Well-suited to implementing barrier release

---

---

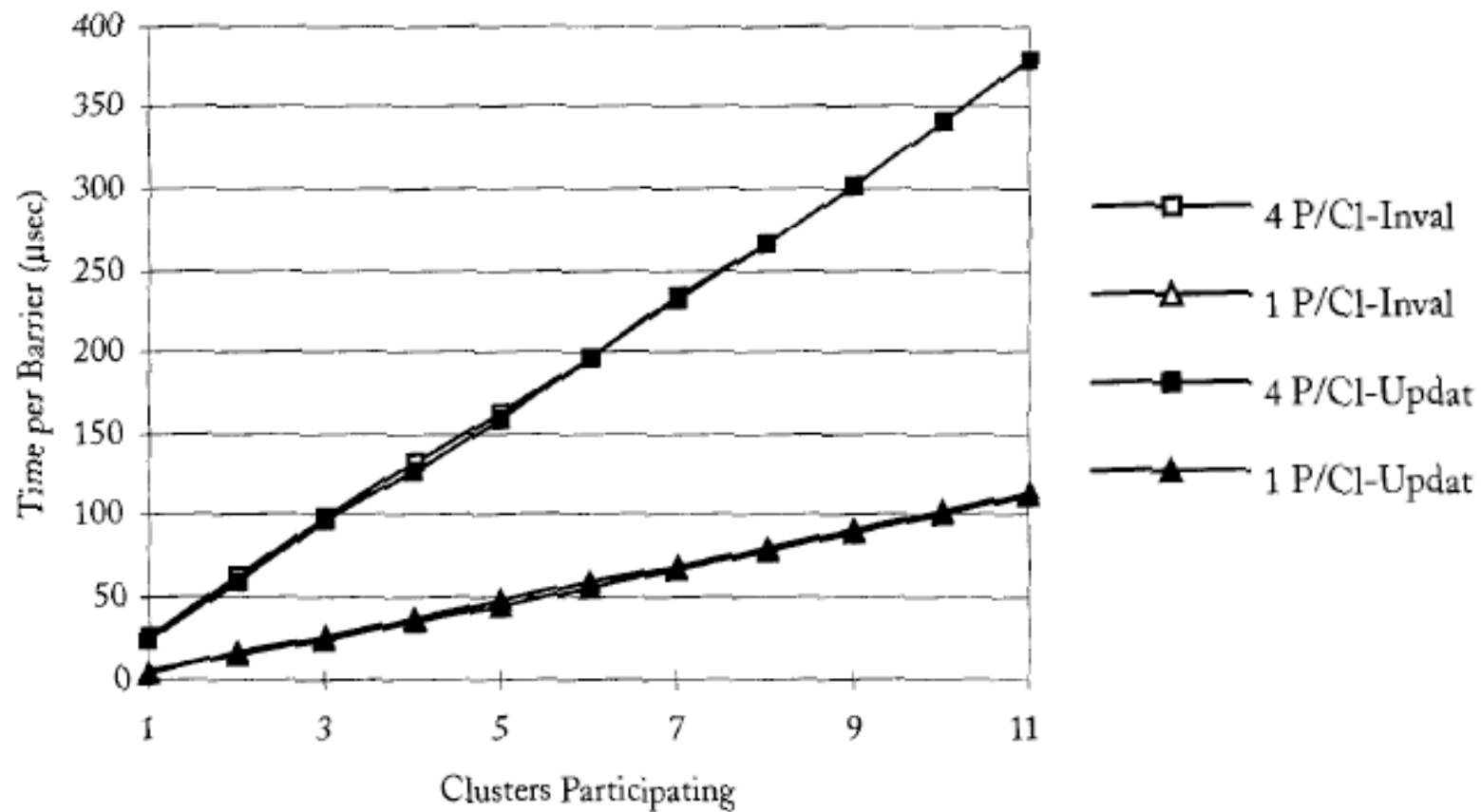
# Barrier Synchronization with Update Write and Fetch&Op



---

---

# Barrier Synchronization Without Fetch&Op



---

---

# Dynamic Work Distribution

- Allocate work to load-balance system, often using task queues
- Mutual exclusion => multiple remote memory accesses per update
- Instead, support Fetch&Op
- Fetch&Op operations can often be parallelized (combining tree)

---

---

# Parallel Prefix

- Synchronize by combining information
- Distribute a result based on that combination
- Carry-lookahead operator is an example
- Can calculate any associative function (sum, maximum, concatenate) in  $O(\log n)$  time

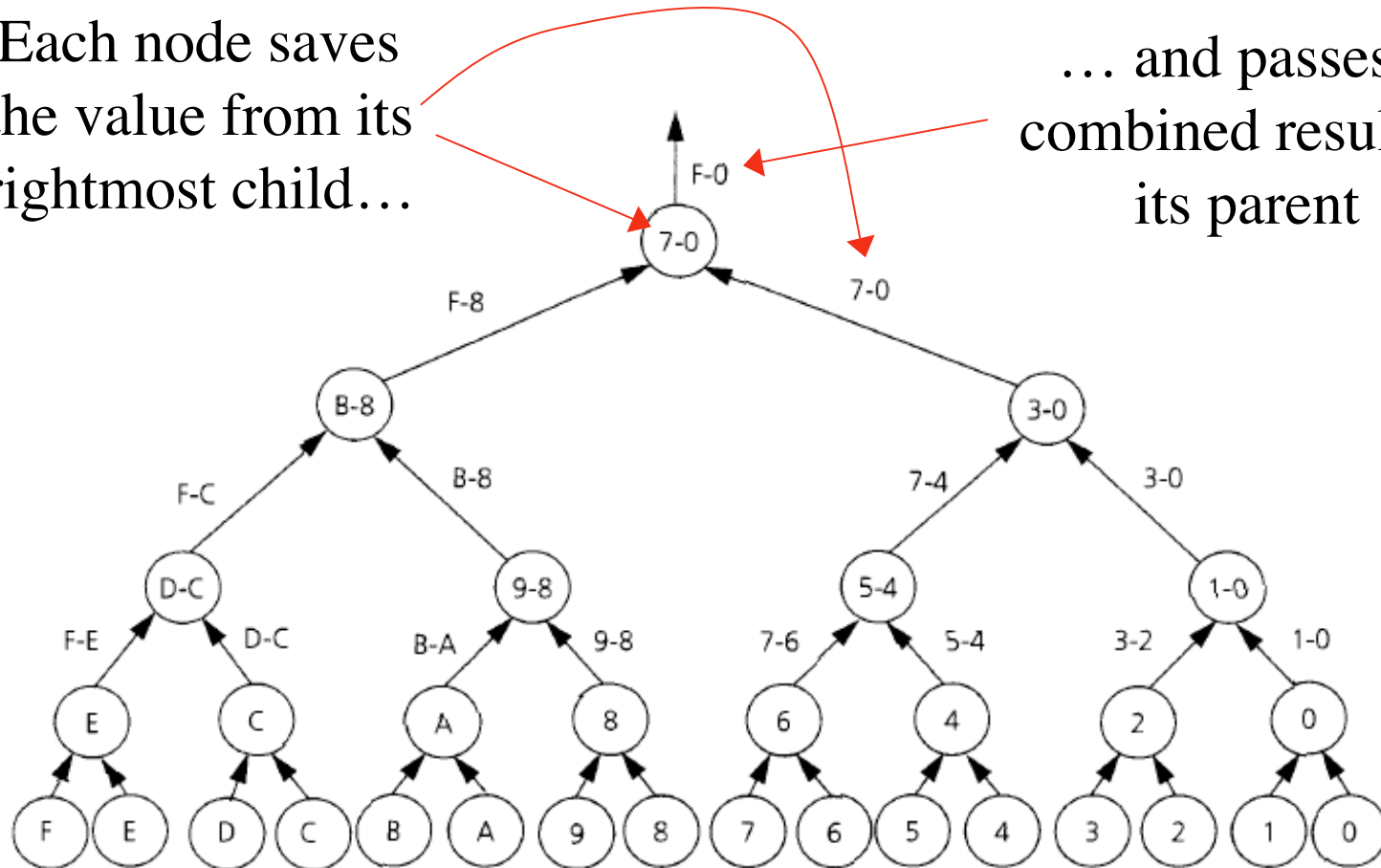
---

---

# Parallel Prefix - Upward Sweep

Each node saves the value from its rightmost child...

... and passes a combined result to its parent



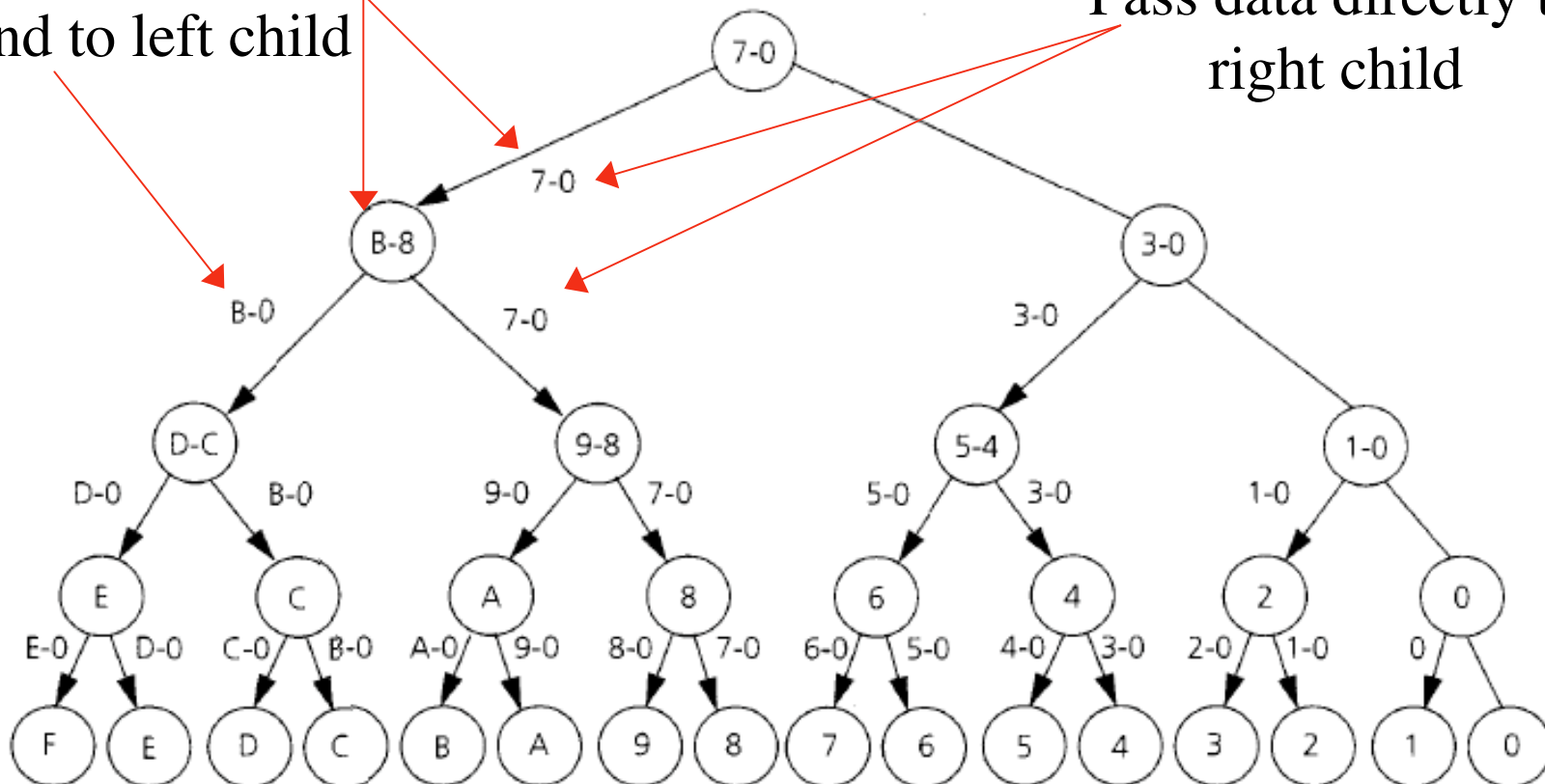
---

---

# Parallel Prefix - Downward Sweep

Combine values,  
send to left child

Pass data directly to  
right child



---

---

# Synchronization and Fine-Grained Parallelism

- How do these techniques apply to transactional memory?
- How do they differ for message-passing vs. shared memory?
- What mechanisms are worth implementing in hardware to support fine-grained parallelism?

---

---

# Questions?