Synchronization without Contention

John Mellor-Crummey and Michael Scott

Presented by Shoaib Kamil

Overview

- Review of some lock types
- MCS lock algorithm
- Barriers
- Empirical Performance
- Discussion

Review of Lock Types

- test&set
  - using a test&set instruction, poll a single memory location
  - acquire lock by changing flag from false to true
  - release by changing back
- test-and-test&set
  - reduce memory/interconnection contention
  - but only while lock is held!
  - exponential backoff helps

Review of Lock Types

- ticket lock
  - next ticket and currently-serving counters
  - acquire lock by fetch&increment on next ticket; own the lock if currently-serving equals our ticket
  - fair (FIFO order)
  - effective backoff
- Anderson
  - fetch&increment to obtain new location; spin on that location
  - previous owner of lock frees it by writing to next loc
  - reduces contention; polling on unique locations
  - but requires coherence & O(P*locks) static space

MCS Lock

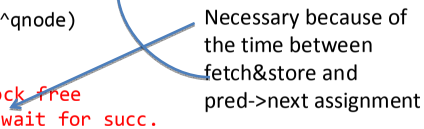
- Maintains a queue of requesters
- Each waiter has a local record that points to the next waiter
- Release gives the next waiter the lock

MCS Lock Pseudocode

```
type qnode = record
next : ^qnode// ptr to successor in queue
locked : Boolean // busy-waiting necessary
type lock = ^qnode// ptr to tail of queue

// I points to a queue link record allocated
// (in an enclosing scope) in shared memory
// locally-accessible to the invoking processor
procedure acquire_lock(L : ^lock; I : ^qnode)
varpred : ^qnode I->next := nil // initially, no successor
pred := fetch_and_store(L, I) // queue for lock
if pred != nil // lock was not free
        I->locked := true // prepare to spin
pred->next := I // link behind predecessor
        repeat while I->locked // busy-wait for lock

procedure release_lock(L * ^lock; I : ^qnode)
if I->next = nil // no known successor
        if compare-and-swap(L, I, nil)
            return // no successor, lock free
        repeat while I->next = nil // wait for succ.
I->next->locked := false // pass lock
```



MCS Lock (con't)

- Alternate release procedure doesn't use compare&swap
  - but doesn't guarantee FIFO order
- All spinning occurs on local data item
  - no unnecessary bus traffic while spinning

Barriers

- Previous work
  - central counter is incremented by each processor, then wait until count equals P
    - large amounts of contention
  - software combining uses groups of *k* organized into a *k*-ary tree, travel up tree to root then down. last one in each leaf is the one that travels up.
    - less contention, but still spinning on non-local location
  - tournament barriers use statically determined node to travel up (not last one to arrive)
    - not local-only spinning on DSM machines

MCS Barrier

- spins only on local-accessible flag variables
- requires O(P) space for P processors
- performs theoretical minimum number of network transactions
- performs O(log P) transactions in critical path
- uses two trees with different structures
  - one for arrival, one for wakeup

MCS Barrier

```
type treenode = record
wsense : Boolean
parentpointer : ^Boolean
childpointers : array [0. .1] of "Boolean
havechild : array [0. .3] of Boolean
cnotready : array [0. .3] of Boolean
        dummy : Boolean // pseudo-data

processor private vpid : integer
// a unique "virtual processor" index
processor private sense : Boolean

shared nodes : array [O..P-11 of treenode
        // nodes[vpid] is allocated in shared memory
        // locally-accessible to processor vpid

// for each processor i , sense is initially true
// in nodes [i] :
// havechild[j] = (4*i+j< P)
// parentpointer =
// hodes[floor((i-l)/4)] .cnotready [(i-1) mod 41
// or &dummy if i = 0
// childpointers [O] = (modes [2*i+l] .wsense,
// or &dummy if 2*i+l>= P
// childpointers [l] = (modes [2*i+2] .wsense,
// or &dummy if 2*i+2 >= P
// initially,
// cnotready = havechild and wsense = false
```

```
procedure tree-barrier
        with nodes[vpid] do
                repeat until cnotready =
                        [false, false, false, false]
cnotready := havechild // init for next ti
parentpointer^ := false // signal parent
                if vpid != 0
                        // not root, wait until parent wakes me
                        repeat until wsense = sense
        // signal children in wakeup tree
        childpointers[0]^ := sense
        childpointers[1]^ := sense
        sense := not sense
```

Results

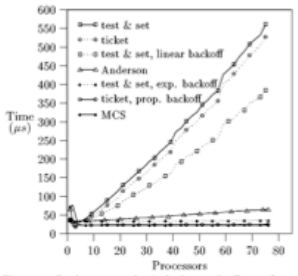


Figure 1: Performance of spin locks on the Butterfly.

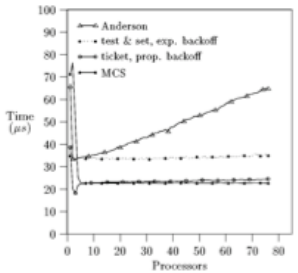


Figure 2: Performance of selected spin locks on the Butterfly.

- MCS scales best on Butterfly
- backoffs are effective
- peak is due to fact some parts of lock acquire/release occur in parallel
- note time to release MCS lock depends on whether there is a processor waiting

Results

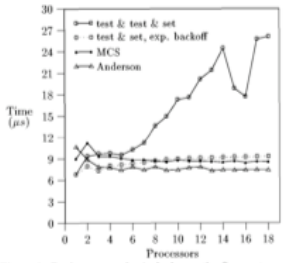


Figure 3: Performance of spin locks on the Symmetry.

Busy-wait Lock	Increase in Network Latency Measured From	
	Lock Node	Idle Node
test_and_set	1420%	96%
test_and_set w/ linear backoff	882%	67%
test_and_set w/ exp. backoff	32%	4%
ticket	992%	97%
ticket w/ prop. backoff	53%	8%
Anderson	75%	67%
MCS	4%	2%

- On Symmetry, MCS and Anderson are best
- Symmetry is more representative of actual lock costs?

Barrier Results

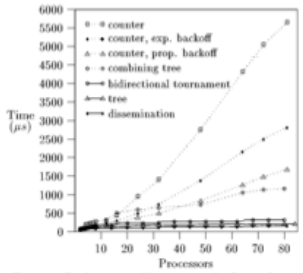


Figure 4: Performance of barriers on the Butterfly.

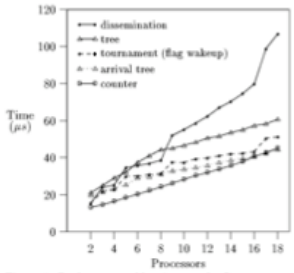


Figure 6: Performance of barriers on the Symmetry.

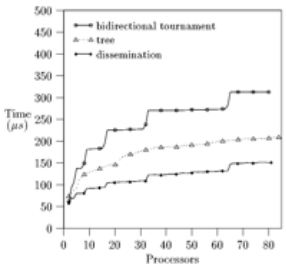


Figure 5: Performance of selected barriers on the Butterfly.

Shared Local Memory

- is good for performance
- helps because it lets processes spin on local items without going to main memory

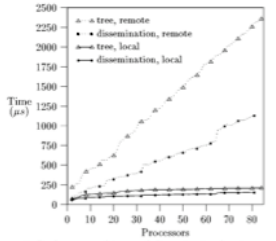


Figure 7: Performance of tree and dissemination barriers on the Butterfly with and without local access to shared memory.

Conclusions