Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors

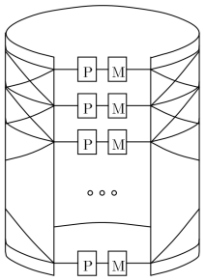
John M. Mellor-Crummey and Michael L. Scott

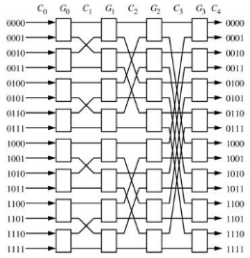
Presented by Charles Lehner and Matt Graichen

Hardware: BBN Butterfly

- shared-memory multiprocessor supporting up to 256 processor nodes
- each node contains an 8 MHz MC68000 and supports one to four MB of memory
- local memory access is direct
- remote memory access is done via a $\log_4$-depth butterfly network
- supports two 16-bit atomic operations
  - `fetch_and_clear_then_add`
  - `fetch_and_clear_then_xor`

Hardware: BBN Butterfly

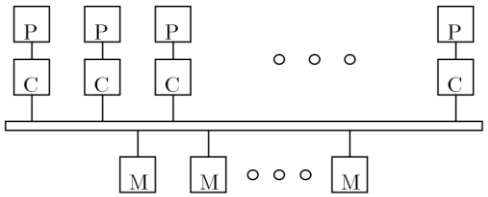




Hardware: Sequent Symmetry

- shared-bus multiprocessor supporting up to 30 processor nodes
- each processor node contains a 16 MHz Intel 80386 processor with 64 KB of two-way set-associative cache
- cache coherence achieved via snooping the shared-bus
- supports 1, 2, and 4 byte atomic `fetch_and_φ` operations
  - no genuine return value (operations only set condition codes)

Hardware: Sequent Symmetry



Spin Locks: Evaluation Criteria

- scalability and induced network load
- single-processor latency
- space requirements
- fairness/sensitivity to preemption
- implementability with given atomic operations

Spin Locks: test_and_set lock (with exp. backoff)

```
type lock = (unlocked, locked)

procedure acquire_lock(L : ^lock)
        delay : integer := 1
        while test_and_set(L) = locked          // returns old value
                pause (delay)                                   //
backoff
                delay := delay * 2

procedure release_lock(L : ^lock)
        lock^ := unlocked
```

Spin Locks: test_and_set lock (with exp. backoff)

- Pros
  - single processor latency
  - space efficiency
  - scales very well (only with exp. backoff!)
- Cons
  - no guarantee of fairness
- Required Atomic Operations
  - `test_and_set`

Spin Locks: ticket lock

```
type lock = record
      next_ticket : unsigned integer := 0
      now_serving : unsigned integer := 0
procedure acquire_lock (L : ^lock)
      my_ticket : unsigned integer := fetch_and_increment (&L->next_ticket)
            // returns old value arithmetic overflow is harmless
      loop
            pause (my_ticket - L->now_serving)
                  // consume this many units of time
                  // on most machines, subtraction works correctly despite overflow
            if L->now_serving = my_ticket
                  return
procedure release_lock (L : ^lock)
      L->now_serving := L->now_serving + 1
```

Spin Locks: ticket lock

- Pros
  - single processor latency
  - space efficiency
  - scales very well (only with prop. backoff!)
- Cons
  - all processes spin on one shared variable
- Required Atomic Operations
  - `fetch_and_increment`

Spin Locks: array-based queueing locks (Anderson's)

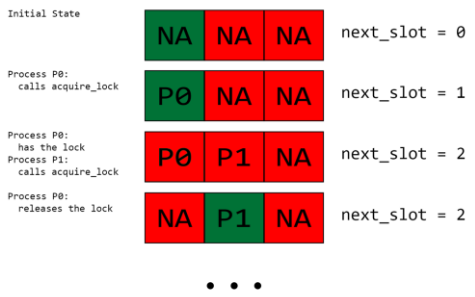
```
type lock = record
      slots : array[0..num_procs - 1] of (has_lock, must_wait)
            // each element of slots should lie either in a separate cache
      // line on cache coherent systems or different memory modules on
      // machines like the Butterfly
            // slots is initialized such that slots[0] = has_lock
      // and slots[1..num_proces - 1] = must_wait
            next_slot : integer := 0
```

Spin Locks: array-based queueing locks (Anderson's)

```
procedure acquire_lock(L : ^lock, my_place : ^integer)
         my_place^ := fetch_and_increment(&L->next_slot)
         if my_place^ mod num_procs = 0
                  // avoid overflow problems
                  atomic_add(&L->next_slot, -num_procs)
         my_place^ := my_place^ mod num_procs
         repeat while L->slots[my_place^] = must_wait   // spin
         L->slots[my_place^] := must_wait                        // init for next
time

procedure release_lock (L : ^lock, my_place : ^integer)
         // give next slot the lock
         L->slots[(my_place^ + 1) mod num_procs] := has_lock
```

Spin Locks: array-based queueing locks (Anderson's)



• • •

Spin Locks: array-based queueing locks

- Pros
  - each processor spins on a different location (memory module and/or separate cache line)
  - guaranteed FIFO order of lock acquisition
- Cons
  - worse single processor latency with respect to the other proposed lock algorithms
  - requires O(P) space where P is the number of processors

Spin Locks: MCS Lock

```
type qnode = record
             next : ^qnode
             locked : Boolean
type lock = ^qnode


// parameter I, below, points to a qnode record allocated in shared memory
// that should be locally-accessible to the invoking processor
procedure acquire_lock (L : ^lock, I : ^qnode)
             I->next := nil
             predecessor : ^qnode := fetch_and_store (L, I)
             if (predecessor != nil)                          // queue was non-empty
                          I->locked := true
                          predecessor->next := I
                          repeat while I->locked              // spin
```

Spin Locks: MCS Lock

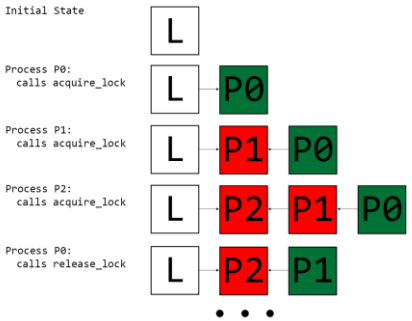
```
procedure release_lock (L : ^lock, I : ^qnode)
             if I->next = nil          // no known successor
                          // compare_and_swap returns true iff it swapped
                          // nil for L, which may only happen if L = I
                          if compare_and_swap (L, I, nil)
                                       return
                          // if the CAS failed, this means that some other processor is in the process of

                          // acquiring the lock, but the setting of their node's next field either hasn't

                          // propagated to this processor or hasn't happened yet. Therefore, we spin in
                          // order to make sure we don't miss setting their lock to false in the next

                          // statement (avoiding deadlock)
                          repeat while I->next = nil
             I->next->locked := false
```

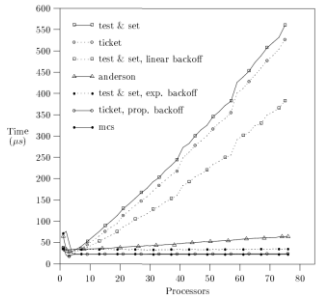
Spin Locks: MCS Lock



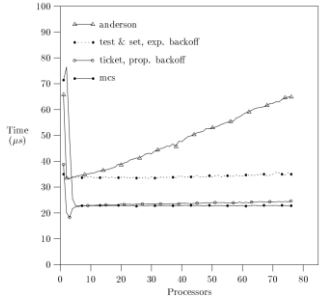
Spin Locks: MCS Lock

- Pros
  - processors spins on locally-accessible flag variables only
  - only O(1) network transactions per lock acquisition
  - requires only a small constant amount of space per lock
  - guaranteed FIFO order of lock acquisition
- Cons
  - worse single processor latency with respect to the other proposed lock algorithms

Spin Locks: Perf. on the Butterfly (empty critical section)



Spin Locks: Perf. on the Butterfly (empty critical section)

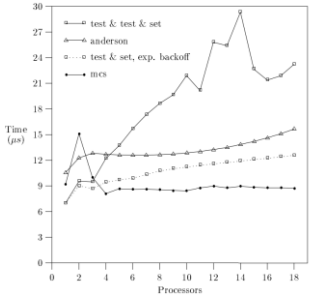


Spin Locks: Perf. on the Butterfly (Increase in Network Latency)

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%

Ir) on the it lock

Spin Locks: Perf. on the Symmetry (empty critical section)



Barriers: Evaluation Criteria

- length of critical path
- total number of network transactions
- space requirements
- implementability with given atomic operations

Barriers: centralized barriers

- each processor:
  - update shared variable on arrival
  - poll the shared variable to check when all have arrived
- problem: consecutive barriers could be skipped
- solution: sense reversal
- drawback: spinning on shared location may cause contention

Barriers: software combining tree barrier

- replace shared variable with tree of references
- each processor updates the state in its leaf
- propagate state up the tree

Barriers: dissemination barrier

```
type flags = record
       myflags : [array 0..1] of array [0..LogP-1] of Boolean
       partnerflags : [array 0..1] of array [0..LogP-1] of ^Boolean

processor private parity : integer := 0
processor private sense : Boolean := true
processor private localflags : ^flags
shared allnodes : array [0..P-1] of flags
       // allnodes[i] is allocated in shared memory
       // locally accessible to processor i
// on processor i, localflags points to allnodes[i]
// initially allnodes[i].myflags[r][k] is false for all i, r, k
// if j = (i+2^k) mod P, then for r = 0, 1:
//     allnodes[i].partnerflags[r][k] points to allnodes[j].myflags[r][k]

procedure dissemination_barrier
       for instance : integer := 0 to LogP-1
              localflags^.partnerflags[parity][instance]^ := sense
              repeat until localflags^.myflags[parity][instance] = sense
       if parity = 1
              sense := not sense
       parity := 1 - parity
```

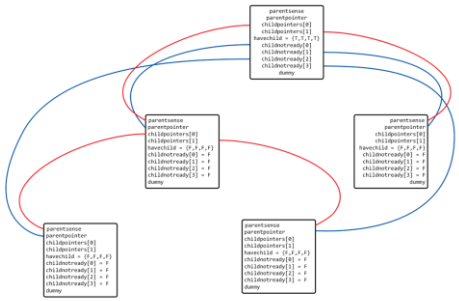
Barriers: “new tree-based barrier”

```
type treenode = record
              parentsense : Boolean
              parentpointer : ^Boolean
              childpointers : array [0..1] of ^Boolean
              havechild : array [0..3] of Boolean
              childnotready : array [0..3] of Boolean
              dummy : Boolean          // pseudo-data
// nodes[vpid] allocated in shared memory (locally accessible to processor vpid)
shared nodes : array [0..P-1] of treenode
processor private vpid : integer       // unique virtual processor index
processor private sense : Boolean
// initial state for processor i
// for node[i]:
//           havechild[j] = true if 4*i+j < P; otherwise false
//           parentpointer = &nodes[floor((i-1)/4].childnotready[(i-1) mod 4]
//                                or &dummy if i = 0
//           childpointers[0] = &nodes[2*i+1].parentsense, or &dummy if 2*i+1 >= P
//           childpointers[1] = &nodes[2*i+2].parentsense, or &dummy if 2*i+2 >= P
//           initially childnotready = havechild and parentsense = false
```

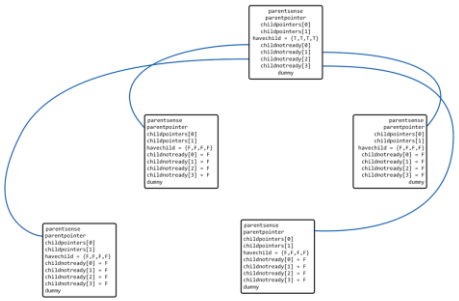
Barriers: “new tree-based barrier”

```
procedure tree_barrier
              with nodes[vpid] do
              repeat until childnotready = {false, false, false, false}
              childnotready := havechild             // prepare for the next barrier
                           parentpointer^ := false               // let parent know I'm ready
              // if not the root node, wait until my parent signals wakeup
              if vpid != 0
                           repeat until parentsense = sense
                           // signal children in wakeup tree
                           childpointers[0]^ := sense
                           childpointers[1]^ := sense
                           sense := not sense
```

Barriers: “new tree-based barrier”



Barriers: “new tree-based barrier”

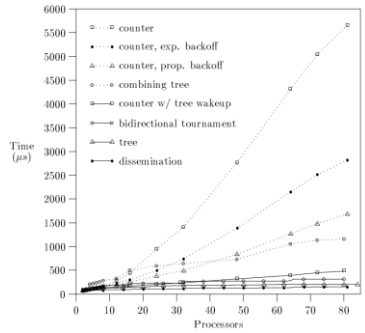


Barriers: “new tree-based barrier”

Barriers: “new tree-based barrier”

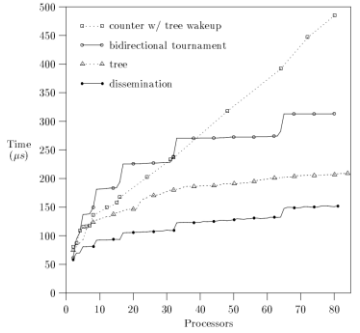
- Pros
  - spins on locally-accessible flags only
  - requires only O(P) space for P processors
  - performs minimum number of network transactions on machines without broadcast (2P - 2)
  - performs O(log P) network transactions on critical path
- Cons
  - useless optimizations for cache coherent, UMA machines like the Symmetry

Barriers: Perf. on the Butterfly

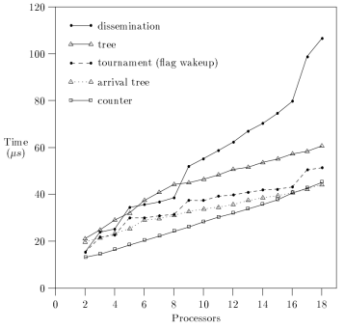


- shared counter leads to contention, linear performance.
- backoff decreases contention

Barriers: Perf. on the Butterfly

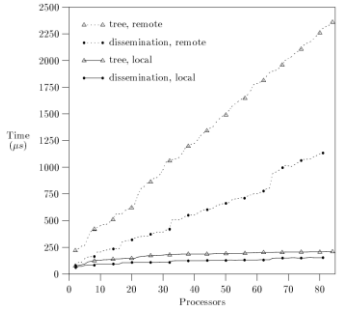


Barriers: Perf. on the Symmetry



coherent cache allows the counter to be effective

Barriers: Importance of Local Memory Access on the Butterfly



forcing memory accesses to traverse the interconnect led to linear performance

Questions?

Image Sources

Scott, Michael L; Mellor-Crummey, John M. *Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors*. ACM Trans. on Computer Systems. 1991

"CSC/ECE 506 Spring 2010/ch 12 PP." - *PG_Wiki*. N.p., n.d. Web. 01 Feb. 2015.