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: 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 \texttt{fetch_and}_{\varphi} operations
  - no genuine return value (operations only set condition codes)
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)

```plaintext
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

```plaintext
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
        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)

```plaintext
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
    next_slot : integer := 0
procedure acquire_lock(L : ^lock, my_place : ^integer)
    my_place^ := fetch_and_increment(&L->next_slot)
        // avoid overflow problems
    my_place^ := my_place^ mod num_procs
    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
    if L->next_slot = my_place
        L->slots[(my_place^ + 1) mod num_procs] := has_lock
```

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

```plaintext	next_slot := 0
procedure acquire_lock(L : ^lock, my_place : ^integer)
    my_place^ := fetch_and_increment(&L->next_slot)
        if my_place^ mod num_procs = 0
            atomic_add(&L->next_slot, -num_procs)
    my_place^ := my_place^ mod num_procs
    repeat
        while L->slots[my_place^] = must_wait
            // spin
        if L->next_slot = my_place
            L->slots[(my_place^ + 1) mod num_procs] := has_lock
```
Spin Locks: array-based queueing locks (Anderson’s)

- **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 definitions**

```plaintext
type qnode = record
  next : ^qnode
  locked : Boolean
end

type lock = ^qnode
```

**Procedure `acquire_lock`**

```plaintext
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

procedure acquire_lock (L : ^lock, I : ^qnode)
```

**Procedure `release_lock`**

```plaintext
if I-next = nil
  // no known successor
  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

procedure release_lock (L : ^lock, I : ^qnode)
```
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)

<table>
<thead>
<tr>
<th>Busy-wait Lock</th>
<th>Increase in Network Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>test and set</td>
<td>1400% 99%</td>
</tr>
<tr>
<td>test and set with linear backoff</td>
<td>882% 64%</td>
</tr>
<tr>
<td>test and set with exp. backoff</td>
<td>32% 4%</td>
</tr>
<tr>
<td>ticket with prop. backoff</td>
<td>6% 8%</td>
</tr>
<tr>
<td>Anderson</td>
<td>75% 64%</td>
</tr>
<tr>
<td>MCS</td>
<td>4% 2%</td>
</tr>
</tbody>
</table>

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

```pascal
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;
end;

shared allnodes : array [0..P-1] of flags;
// allnodes[i] is allocated in shared memory
// locally accessible to processor i
// on processor 1, myflags 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 do
  localflags^.partnerflags[parity][instance]^ := sense;
repeat until localflags^.myflags[parity][instance] = sense;
if parity = 1 then
  sense := not sense;
  parity := 1
end;
```

Barriers: “new tree-based barrier”

```pascal
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;
end;

shared nodes : array [0..P-1] of treenode;
// nodes[vpid] allocated in shared memory (locally accessible to processor vpid)
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[(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”

```pascal
procedure tree_barrier
with nodes[vpid] do
  repeat until childnotready = {false, false, false, false} do
    childnotready := havechild;
    parentpointer^ := false;
    // let parent know I'm ready
    if vpid != 0 then
      repeat until parentsense = sense do
        childpointers[0]^ := sense;
        childpointers[1]^ := sense;
        sense := not sense;
        // signal children in wakeup tree
        childpointers[i]^ := sense;
        // let parent know I'm ready
        repeat until parentsense = sense do
          childpointers[i]^ := sense;
          sense := not sense;
```
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 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