Mellor-Crummey and Scott, Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors, TOCS, Feb 1991.

why are we discussing this paper?
 illustrates interaction of interconnect, caching, and parallel code
 locks often important to parallel performance

why lock performance important
 good parallel prog won't spend much time in locks?
 but usually a few locks, e.g. TSP work list
 typical scenario:
  on few CPUs, just 5% in some lock, rarely two CPUs at same time
  a few years later, 32 CPUs, 5% means *often* multiple CPUs waiting
  danger: locks get less efficient with many waiters
   so 5% goes up to 50%
  you see this a lot
 paper investigates extreme case: all CPUs waiting for same lock

overall msg of paper
 don't need special h/w dedicated to locking
 s/w using simple atomic instrs can get good perf
 design details matter for performance

paper's h/w
 BBN butterfly: multistage interconnect, no caches, local memory
  [diagram: P, M, net]
  net wraps around
  each cpu on same board as 1/Nth of RAM
  4 us remote read (5x local)
  they had 80 CPUs!
  what were they thinking -- a machine with no cache?
   answer: RAM time ~~ instruction time
  68000, 8 mhz
  important points for us:
   no caches, one mem unit local to each CPU
  is no-cache architecture at all relevant any more?
   a lot like Beehive: cache but you must invalidate to see other changes
   relevant if you think c.c. may not scale up to huge # of cores
 symmetry: bus, cache, coherence
  similar to modern small-scale multiprocessors
  [diagram: P, C, bus, M]
  80386, 16 mhz
  write-back cache
  snoop for writes, invalidate

what are atomic operations?
 usually a single machine instruction
 test_and_set(addr)
  tmp = *addr

```
  *addr = 1
  return tmp
fetch_and_increment(addr)
  tmp = *addr
  *addr = *addr + 1
  return tmp
```

how does h/w make these operations atomic?
 bus-based machines, like symmetry:
  reserve bus
  read mem (or snoop from a cache)
  [add]
  write mem (+invalidate caches)
  release bus
 network interconnect, like Butteryfly:
  send special msg to mem unit
  mem unit does read/modify/write locally, returns result
  mem unit processes incoming msgs one at a time

so
 how to implement locks using atomic operations?
 how well do they perform?

test-and-set (t-s) lock
 (Algorithm 1 minus the exponential backoff)

```
 while(test_and_set(l) == 0)
   ;
```

 "spin lock"
 performance on butterfly?
  hot-spot memory unit
  some of net congested
 on symmetry?
  constant bus traffic
 the nub of the problem
  we don't care if waiting CPUs waste their own time
  we do care if waiting CPUs slow lock holder!
 will try to guess time for N CPUs to get lock
  assume dominant factor is interconnect messages
 if net/bus fair, holding time = O(N) waiters
  since holder only gets 1/Nth of bus capacity
 time for all N to get lock: O(N^2)
 the paper considers this non-scalable
 how to have fewer probes on net/bus from waiters?

t+s+exp:
 goal: probe less often
 paper's Algorithm 1
 why might exponential backoff work well?
  maybe fewer total probes, since less often

slow down lock holder less
why not constant delay?
hard to choose delay time
too large: waste
too small: everyone probes mult times/crit, so N^2
can we analyze # of probes?
if all cpus acquire at the same time:
there *is* a message-count problem
N episodes, in which every node probes, only one wins
so N^2 msgs
and an elapsed-time problem, backoff will overshoot
if cpus arrive spaced out:
as would be true for repeated benchmarks
then good chance one guy probes and wins, others are sleeping
not sure how to analyze
suppose takes time O(N) for all CPUs to succeed
how many probes does each CPU make in time N? logN
so total probes: N*logN
bus cc: N*logN
rem+loc: N*logN
unlikely to be fair!
some CPUs will have much lower delays than others
will win, and come back, and win again
some CPUs with have huge delays, will sit idle
doing no harm, but doing no work

t+t+s:
goal: reduce probe traffic using c.c.
```
while(1){
  while(L == locked)
    ;
  if(test_and_set(L) != locked)
    return;
}
```
use cache coherence to alert us when unlocked
spin in cache until then, generate no bus traffic
should be much better than t+s
should avoid overshoot/unfairness of t+s+exp
what happens during unlock? for c.c. bus machine
invalidate
everyone sees "unlocked"
everyone is going to run t+s
first core to run t+s wins
2nd core runs t+s, loses, goes to top
3rd core t+s, invalidates, goes to top
2nd core re-loads
4th core t+s, invalidates, goes to top
2nd and 3rd re-load
Nth core t+s, invalidates, goes to top

N-2 cores re-load
there were about N^2 re-loads!
so maybe N*N*N
best case:
winner runs t+s before any core sees unlock
so N-1 cores re-load (but don't t+s)
so N^2
bus cc: N^2 to N^3
rem+loc: N^2? like t-s
problem: N^3 is bad!

ticket:
(Linux uses ticket locks)
goal: fair, cheaper than t-t-s
idea: assign numbers, wake up one at a time
avoid repeated t-s after every release
Algorithm 2
ignore the pause
is it fair?
analysis for c.c. bus machine
waiting is cheap, spin in local caches
what happens in acquire?
maybe N fetch_and_increments
though spread out in steady state
each is just one bus xaction (no re-loads after invalidate)
what happens after release?
invalidate
N re-loads
so N^2
bus cc: N^2 (but NOT N^3)
rem+loc: N^2
problem: N^2 still blows up rapidly

anderson:
can we do better than N^2?
can we get rid of the N re-loads after release?
what if each core spins on a *different* location?
Algorithm 3
analyze for c.c. machine
acquire cost?
a fetch-and-increment (maybe two)
(local spin)
release cost?
invalidate someone else's slots[] element
they have to re-load
no other CPUs involved
so O(N)
O(N) is what the paper means by scalable!
does it have value on non-c.c. machines e.g. Butterfly?

  yes: may spread spinning over many memory units, no hot spot
 problem: high space cost
  often much more than size of protected object

how could we fix anderson idea to work well on Butterfly?
 each CPU spins in its local memory
 if you did that, O(N) on butterfly too
 that is what MCS lock does -- Algorithm 5

MCS
 each CPU has an anderson-style slot in local mem
 forms a linked list with fetch-and-store
 L points to most recent acquire() caller
 other end of list is current holder
 each CPU's I->next points to next in line
 acquire makes current last-in-line's I->next point to me
 release:
  tricky: race between release and acquire
    in case where previously no-one else was waiting
  look for L pointing to an I where I->next is zero
    means acquire() has done fetch_and_store but not pred->next := I
 cost?
  acquire does a few loads and stores
    invalidates L on c.c. machine, but no-one spinning on it
    spins on local memory, so butterfly interconnect happy
      BUT every mem bank still has someone beating on it -- local CPU
      may still slow down lock holder a little bit
  release
    modifies only one CPU's I->locked
    so only one CPU invalidates and re-reads on c.c. machine

what results do they see?
 are they what we expect?

Figure 4 -- Butterfly
 graph shows time per successful acquire
 scalable => flat
  any rise means expense grows
 MCS is at very bottom, looks pretty flat, they win!
 t-s and ticket have O(N^2)-looking cost, pretty bad, no surprise
 anderson much better but still rises a bit
  why so good on non-caching machine?
    spreads load over N mem units?
  why does it rise?
    waiters using higher and higher fraction of interconnect?
 t-s-exp very good, looks perfectly scalable rather than NlogN

Figure 7 -- Sequent Symmetry (c.c.)
 t-t-s worst, N^2, don't know why erratic

MCS, anderson, t-s-exp all scalable
as we expect

my results on 48-CPU modern c.c. AMD, slowest to fastest:
my results pretty compatible with Figure 7!
ticket (N^2 w/ high slope: fetch+add expensive???, same n=2 as anderson)
t-t-s (only looks good b/c grossly unfair)
t-s (drifts up, unfair but not as bad as t-t-s)
anderson (flat, start higher than t-s b/c fetch+add expensive??)
t-s-exp (flat. wins b/c unfair? some shut out, so N lower)

notice they are all super fast for one cpu
atomic instructions generate no interconnect traffic if cache line exclusive

if you think you won't have contention
use t-s
but if you later get contention
anderson?
probably not!
fix the underlying problem!

are Beehive locks scalable?
no: retries w/o backup

how could you make Beehive locks scale well?
have lock unit wait
if you see a release, flip a bit invalidating it
now you have lock
then just 1 ring msg per waiting CPU

what about barriers?

cost of Algorithm 7 on c.c machine?
all but last CPU:
each a few bus transactions
spins in cache until last CPU inverts sense
last CPU:
inverts sense
invalidate
N re-reads
so O(N)

Algorithm on BBN butterfly?
waiters use interconnect+mem capacity, slow down remaining executors
so more CPUs -> slower, bad news

how to fix for Butterfly?
again, everyone spins on local mem
everyone who enters needs to check if they are last CPU

do N reads of status of each other CPU?
no, that would have N^2 message cost

Algorithm 11

pre-arrange in tree
when you enter barrier
spin on local mem waiting for children to write childready
then write your childready in parent's local mem
spin on parentsense in local mem, to be written by parent on way down
write each child's parentsense
all spinning is on local memory
single writes up and down tree