Algorithms for Scalable Synchronization on Shared-memory Multiprocessors

John Mellor-Crummey

Department of Computer Science
Rice University

johnmc@rice.edu

Context: Synchronization

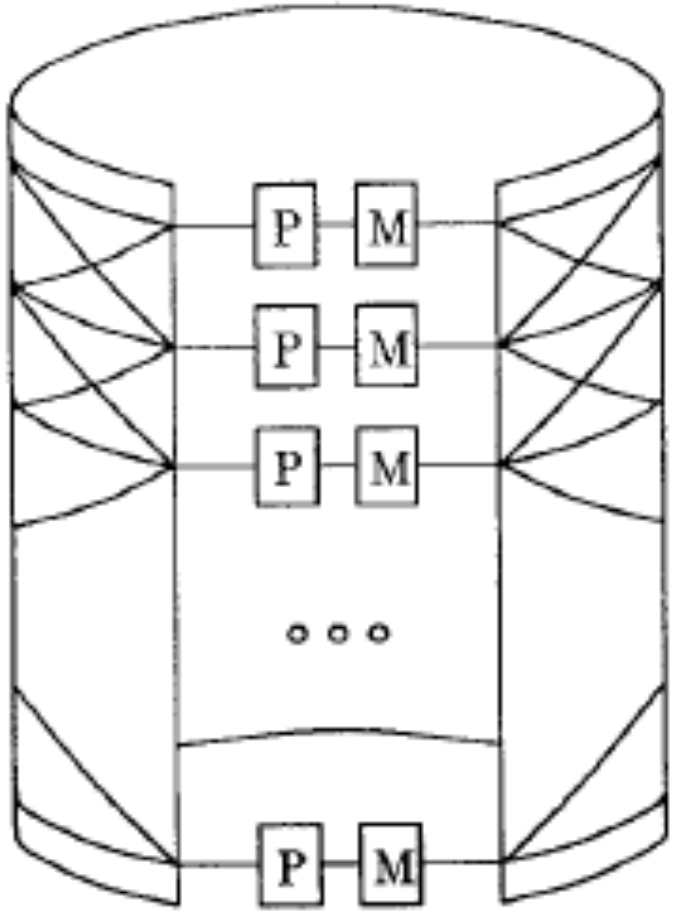
- Previous lecture: wait-free synchronization
- Today: locks and barriers
- Upcoming:
  - —synchronization on multicore
  - —transactional memory
  - —practical non-blocking concurrent objects

Outline

- Hardware then and now
- Locks
- Barriers
- Barrier performance

Hardware Then: BBN Butterfly

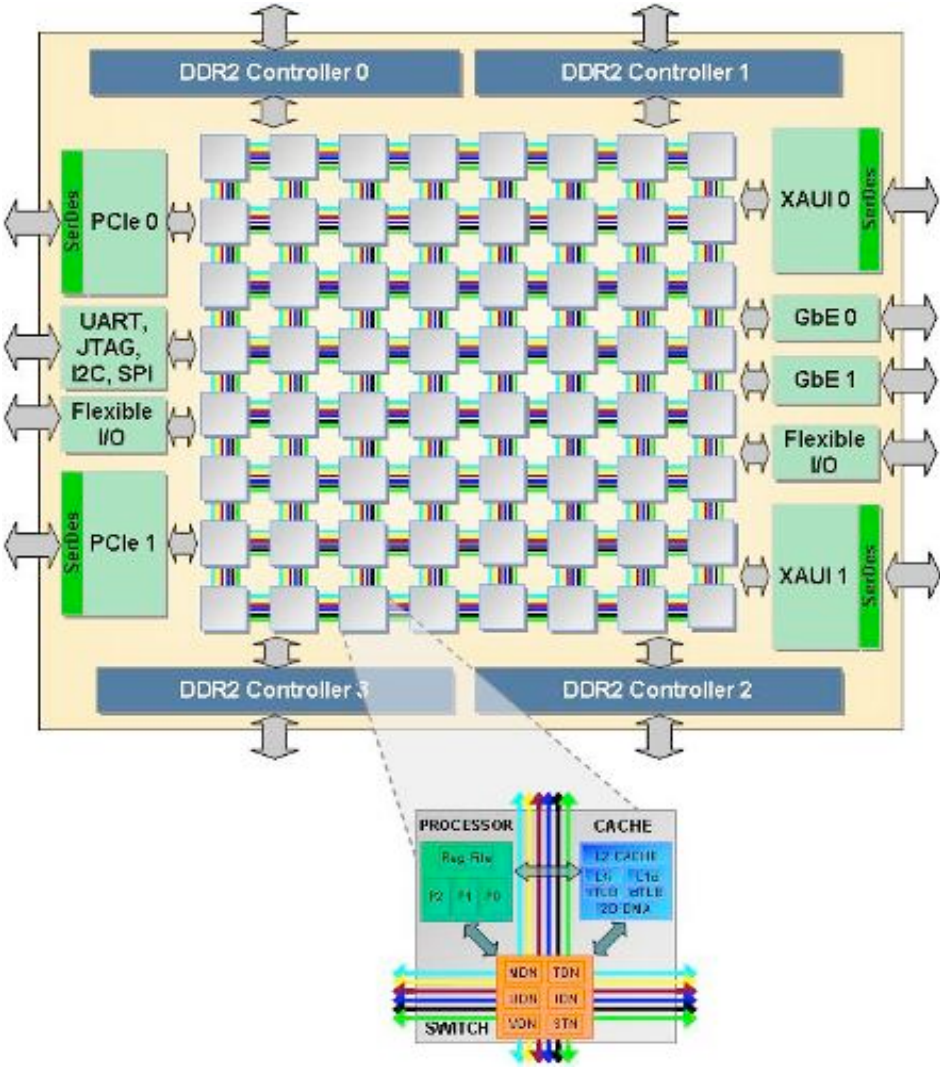
- 8 MHz MC68000
- 24-bit virtual address space
- 1-4 MB memory per PE
- $\log_4$ depth switching network
- Packet switched, non-blocking
- Remote reference
  - —4us (no contention)
  - —5x local reference
- Collisions in network
  - —1 reference succeeds
  - —others abort and are retried later
- 16-bit atomic operations
  - —fetch_clear_then_add
  - —fetch_clear_then_xor



Hardware Now: Tilera TilePro64 (2008)

- Tiled processor design with 64 tiles
- Each tile: processor, cache, switch
  - —32-bit 5-stage VLIW pipeline
  - —64 registers
  - —separate L1 I & D caches
  - —64KB unified L2 cache
  - —"home core" + directory coherence
- 8x8 mesh networks
  - —packets dynamically routed based on two-word packet header (analogous to IP, port pair)
  - —three networks (HW controlled)
    - – "memory movement"
    - – cache coherence
  - —other three networks (SW controlled)
    - – one for I/O and OS control
    - – two for applications: core-to-core communication

      user-level API to R/W remote registers



Berezeki et al. Many-core Key-value Store. 2nd Intl. Green Computing Conf, Orlando, FL. July 27, 2011.
http://bit.ly/n6IX8B

Synchronization Approaches

- **Busy waiting (spinning)**
  - —poll one or more variables to determine when to proceed
  - —best when # SW threads ≤ # HW threads
  - —may yield the processor to other threads
  - —advantage
    - – low latency
  - —disadvantage
    - – consumes core resources
- **Blocking**
  - —transition a thread to a BLOCKED state, to be awoken later
  - —especially useful when # SW threads ≥ HW threads
  - —advantages
    - – avoids squandering resources on thread that cannot make progress
    - – low power
  - —disadvantage
    - – high latency: high cost of context switching

Problems with Naïve Busy Waiting

- May produce large amounts of
  - —network contention
  - —memory contention
  - —cache thrashing
- Bottlenecks become more pronounced as applications scale
  - —hot spot: target of disproportionate share of network traffic
    - – busy waiting on synchronization variables can cause hot spots

      e.g. busy-waiting using test-and-set
  - —impact of hot spots
    - – Pfister and Norton

      presence of hot spots can severely degrade performance of all network traffic in multi-stage interconnection networks
    - – Agarwal and Cherian

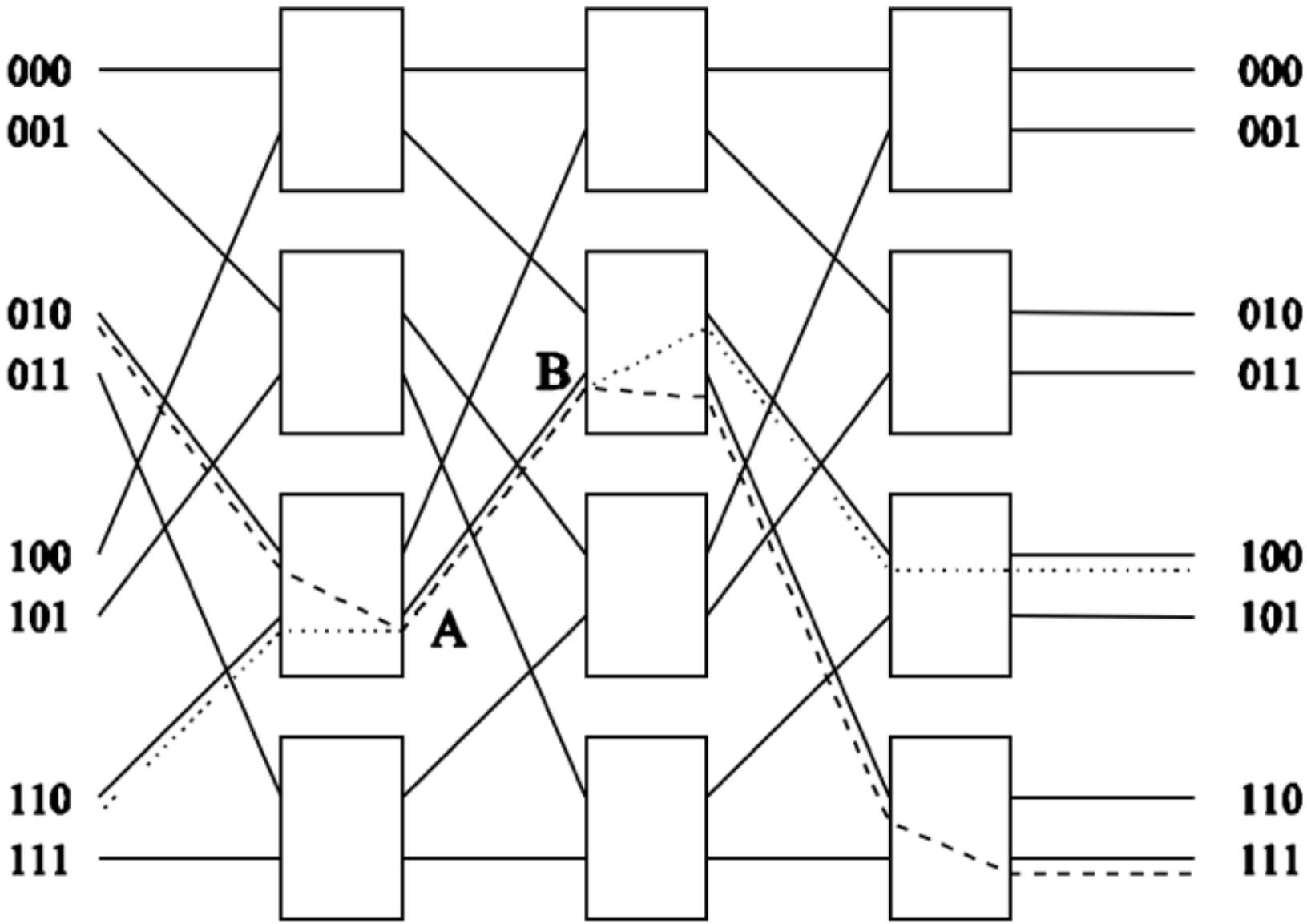
      studied impact of synchronization on overall program performance

      synch memory references cause cache-line invalidations more often than other memory references

      simulations of 64-processor dance-hall architecture

      synchronization accounted for as much as 49% of network traffic

Multistage Interconnect: Omega



Scalable Synchronization

Efficient busy-wait algorithms are possible

- **Each processor spins on a separate *locally-accessible* flag**
  - **may be locally-accessible via**
    - **coherent caching**
    - **allocation in *local* physically-distributed shared memory**
- **Another processor terminates the spin when appropriate**

Locks

- Why locks?
- Atomic operations
- Lock algorithms
  - —test and set
  - —test and set with exponential backoff
  - —ticket lock
  - —Anderson's array-based lock
  - —MCS queue-based locks
  - —CLH queue-based lock

Why Locks?

- **Not robust: if lock holder delayed, progress stalls**
- **Relationship between lock and data is implicit**
  - —preserved only through programmer discipline
  - —association between lock and data is a global property
    - – convention must be observed by all code accessing the data
- **Hard to use**
  - —coarse-grain locks shackle parallelism
  - —fine-grain locks admit possibility of deadlock
  - —priority inversion
  - —lack of composability
    - – calling into code you don't control is a recipe for deadlock
      - locks must be acquired in a fixed global order to avoid deadlock
    - – extensible frameworks often call virtual functions while holding lock

So then, why locks? Efficient in space and time.

Atomic Primitives for Synchronization

Atomic read-modify-write primitives

- **test_and_set(Word &M)**
  - —writes a 1 into M
  - —returns M's previous value
- **swap(Word &M, Word V)**
  - —replaces the contents of M with V
  - —returns M's previous value
- **fetch_and_Φ(Word &M, Word V)**
  - —Φ can be ADD, OR, XOR
  - —replaces the value of M with Φ(old value, V)
  - —returns M's previous value
- **compare_and_swap(Word &M, Word oldV, Word newV)**
  - —if (M == oldV) M $\leftarrow$ newV
  - —returns TRUE if store was performed
  - —universal primitive

Load-Linked & Store Conditional

- load_linked(Word &M)
  - —sets a mark bit in M's cache line
  - —returns M's value
- store_conditional(Word &M, Word V)
  - —if mark bit is set for M's cache line, store V into M, otherwise fail
  - —condition code indicates success or failure
  - —may spuriously fail if
    - – context switch, cache line eviction, interrupt, …
- Arbitrary read-modify-write operations with LL / SC

  loop forever

  load linked on M returns V

  execute sequence of instructions performing arbitrary computation on V and other values

  store conditional of V' into M

  if store conditional succeeded exit loop
- Supported on Alpha, PowerPC, MIPS, and ARM

Test & Set Lock

```
type lock = (UNLOCKED=0, LOCKED=1)

procedure acquire_lock (L : ^lock)
  loop
    // NOTE: test and set returns old value
    if test_and_set(L) = UNLOCKED
       return

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

Test & Set Lock Notes

- Space: n words for n locks and p processes
- Lock acquire properties
  - —spin waits using atomic read-modify-write
- Starvation theoretically possible; unlikely in practice
- Poor scalability
  - —continual updates to a lock cause heavy network traffic
    - on cache-coherent machines, each update causes an invalidation

Test & Set Lock with Exponential Backoff

```
type lock = (UNLOCKED=0, LOCKED=1)

procedure acquire_lock (L : ^lock)
  delay : integer := 1

  // NOTE: test and set returns old value
  while test_and_set (L) = LOCKED
    pause (delay)        // wait this many units of time
    delay := delay * 2   // double delay each time

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

Tom Anderson, IEEE TPDS, January 1990

Test & Set Lock with Exp. Backoff Notes

- Grants requests in unpredictable order
- Starvation is theoretically possible, but unlikely in practice
- Spins (with backoff) on remote locations
- Atomic primitives: test_and_set

- Pragmatics: need to cap probe delay to some maximum

IEEE TPDS, January 1990

Ticket Lock with Proportional Backoff

```
type lock = record
  next_ticket : unsigned integer := 0
  now_serving : unsigned integer := 0

procedure acquire_lock (L : ^lock)
  // NOTES: fetch_and_increment returns old value
  //        arithmetic overflow is harmless here by design
  my_ticket : unsigned integer :=
     fetch_and_increment(&L->next_ticket)
  loop
     // delay proportional to # customers ahead of me
     // NOTE: on most machines, subtraction works correctly despite overflow
     pause(my_ticket - L->now_serving)
     if (L->now_serving = my_ticket) return

procedure release_lock (L : ^lock)
  L->now_serving := L->now_serving + 1
```

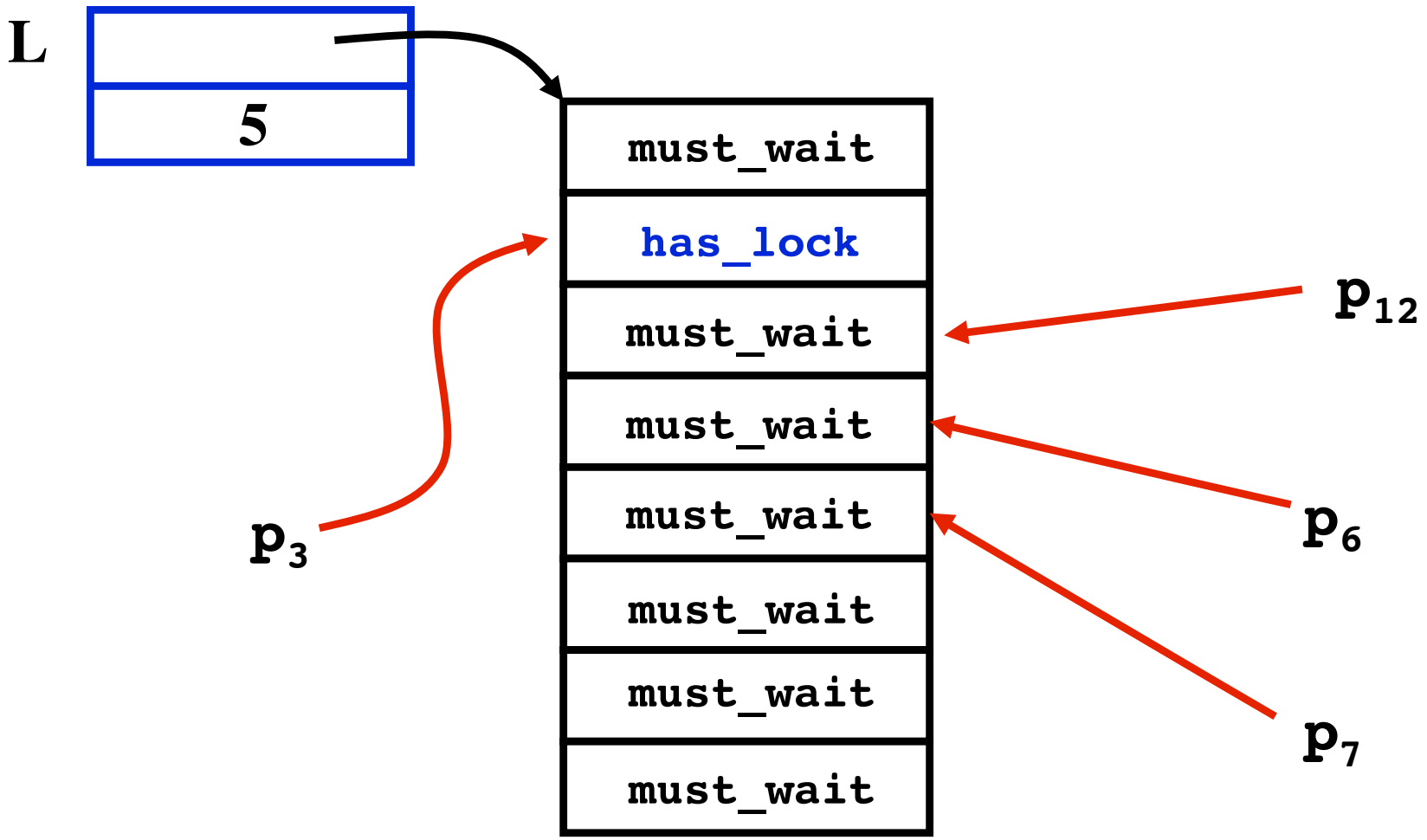
Ticket Lock Notes

- Grants requests in FIFO order
- Spins (with backoff) on remote locations
- Atomic primitives: fetch_and_increment

Anderson's Array-based Queue Lock

```
type lock = record
  slots: array [0..numprocs -1] of (has_lock, must_wait)
     := (has_lock, must_wait, must_wait, ..., must_wait)
  // each element of slots should lie in a different memory module or cache line
  next_slot : integer := 0

// parameter my_place, below, points to a private variable in an enclosing scope
procedure acquire_lock (L: ^lock, my_place: ^integer)
  my_place^ := fetch_and_increment (&L->next_slot)
  if my_place^ mod numprocs = 0
     // decrement to avoid problems with overflow; ignore return value
     atomic_add(&L->next_slot,-numprocs)
  my_place^ := my_place^ mod numprocs
  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)
  L->slots[(my_place^ + 1) mod numprocs] := has_lock
```

Anderson's Lock



Anderson's Lock Notes

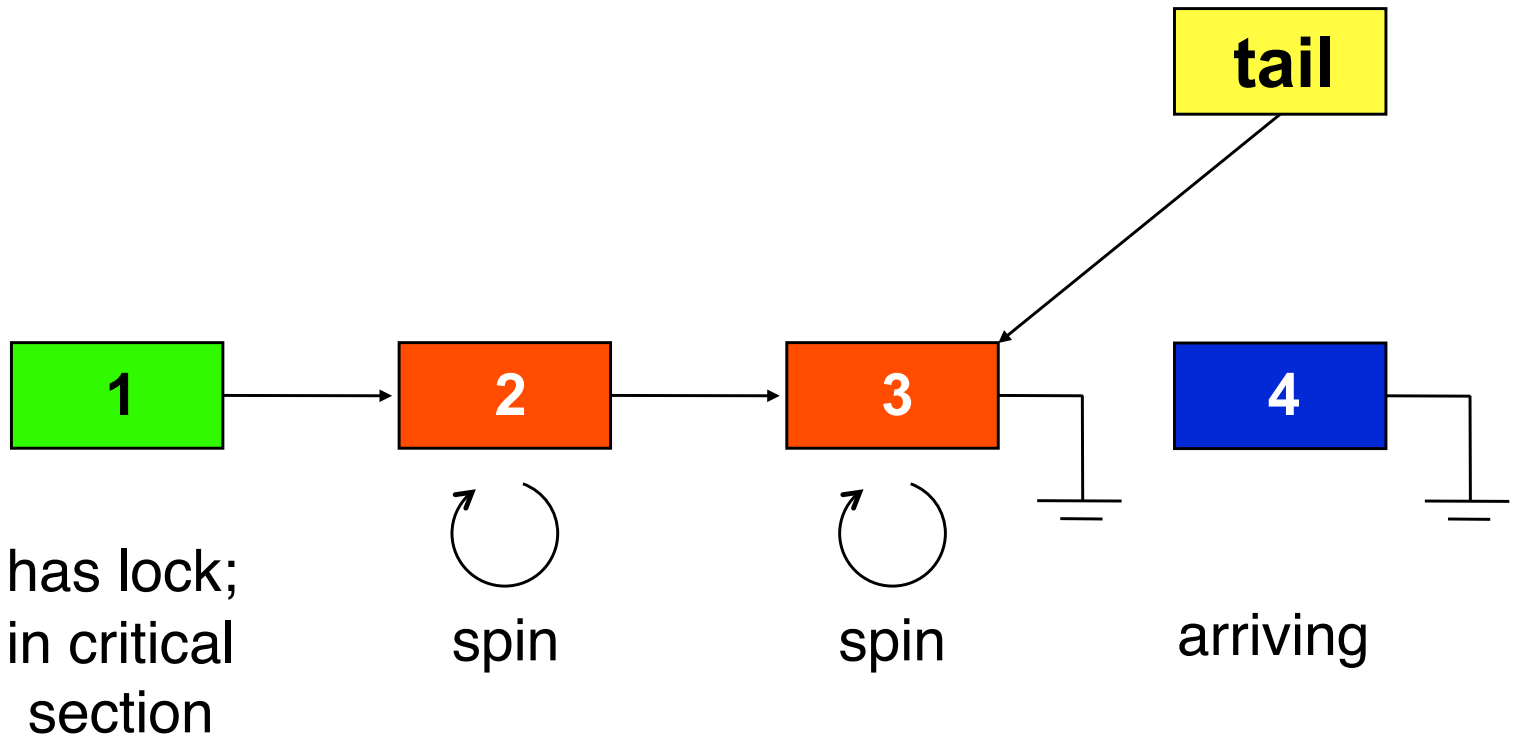
- **Grants requests in FIFO order**
- **Space: *O(pn)* space for *p* processes and *n* locks**
- **Spins only on local locations on a cache-coherent machine**
- **Atomic primitives: fetch_and_increment and atomic_add**

IEEE TPDS, January 1990

The MCS List-based Queue Lock

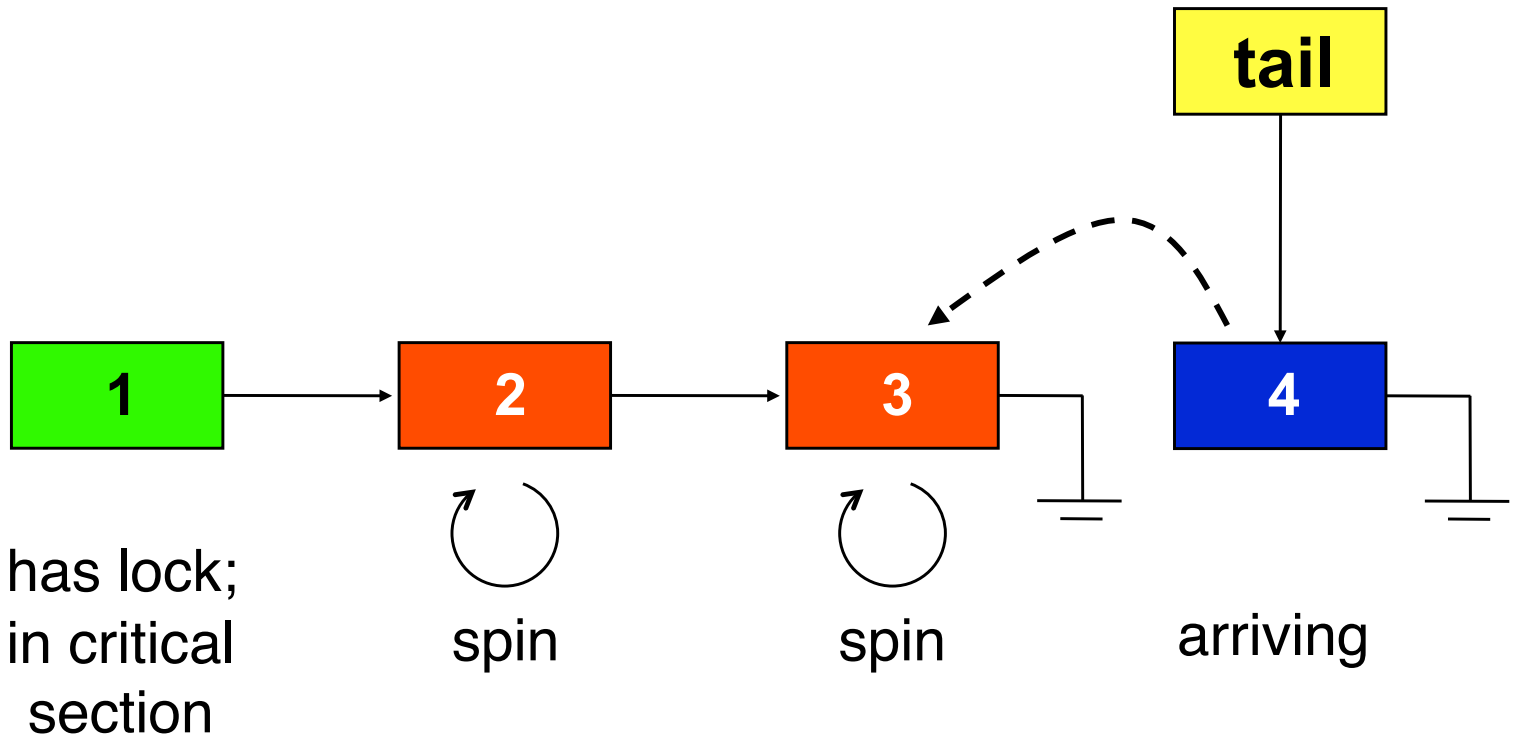
```
type qnode = record
  next : ^qnode
  locked : Boolean

type lock = ^qnode   // initialized to nil

// parameter I, below, points to a qnode record allocated (in an enclosing scope) in
// shared memory 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

procedure release_lock (L : ^lock, I: ^qnode)
  if I->next = nil             // no known successor
    if compare_and_swap(L, I, nil) return // I was still at tail of list
    repeat while I->next = nil // spin
  I->next->locked := false
```

MCS Lock In Action - I



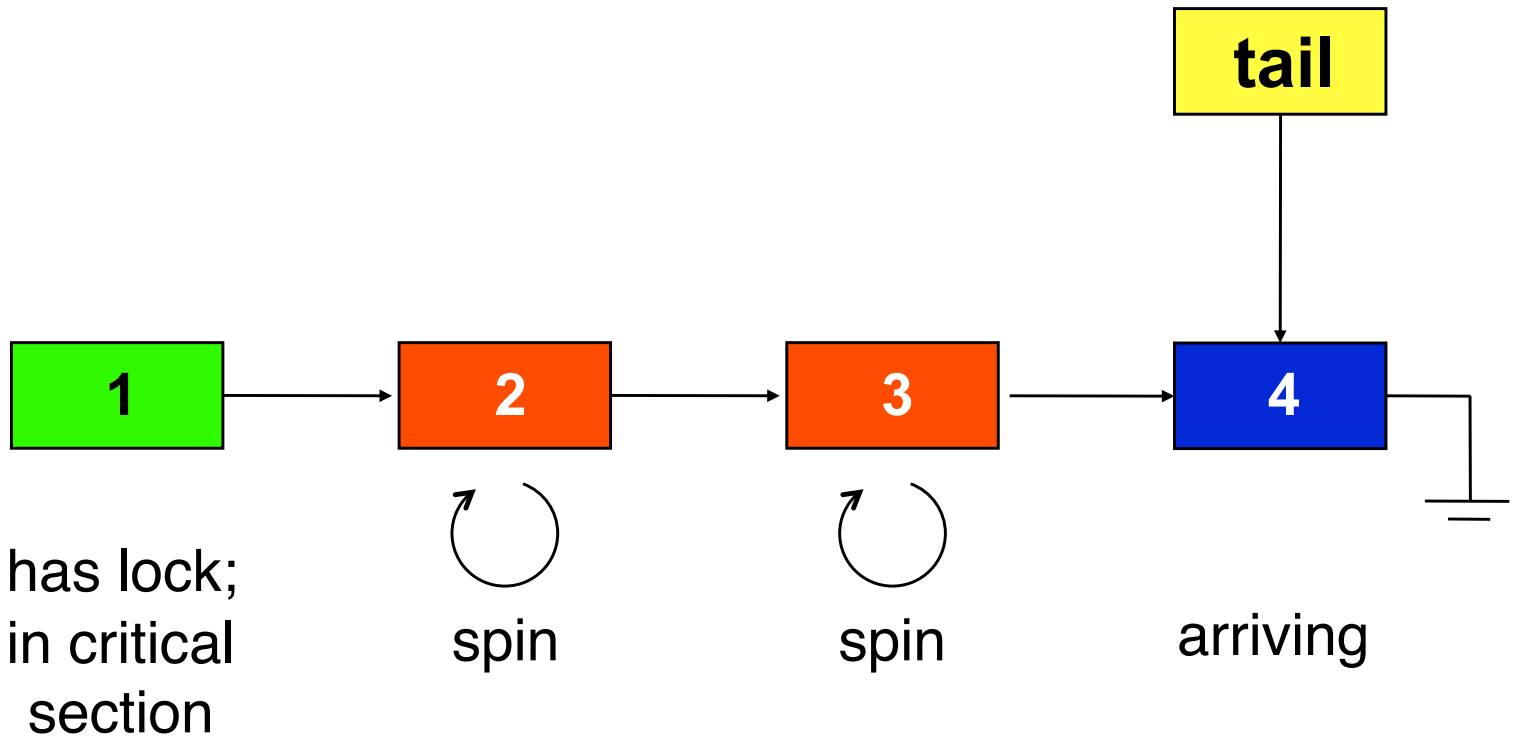
Process 4 arrives, attempting to acquire lock

MCS Lock In Action - II



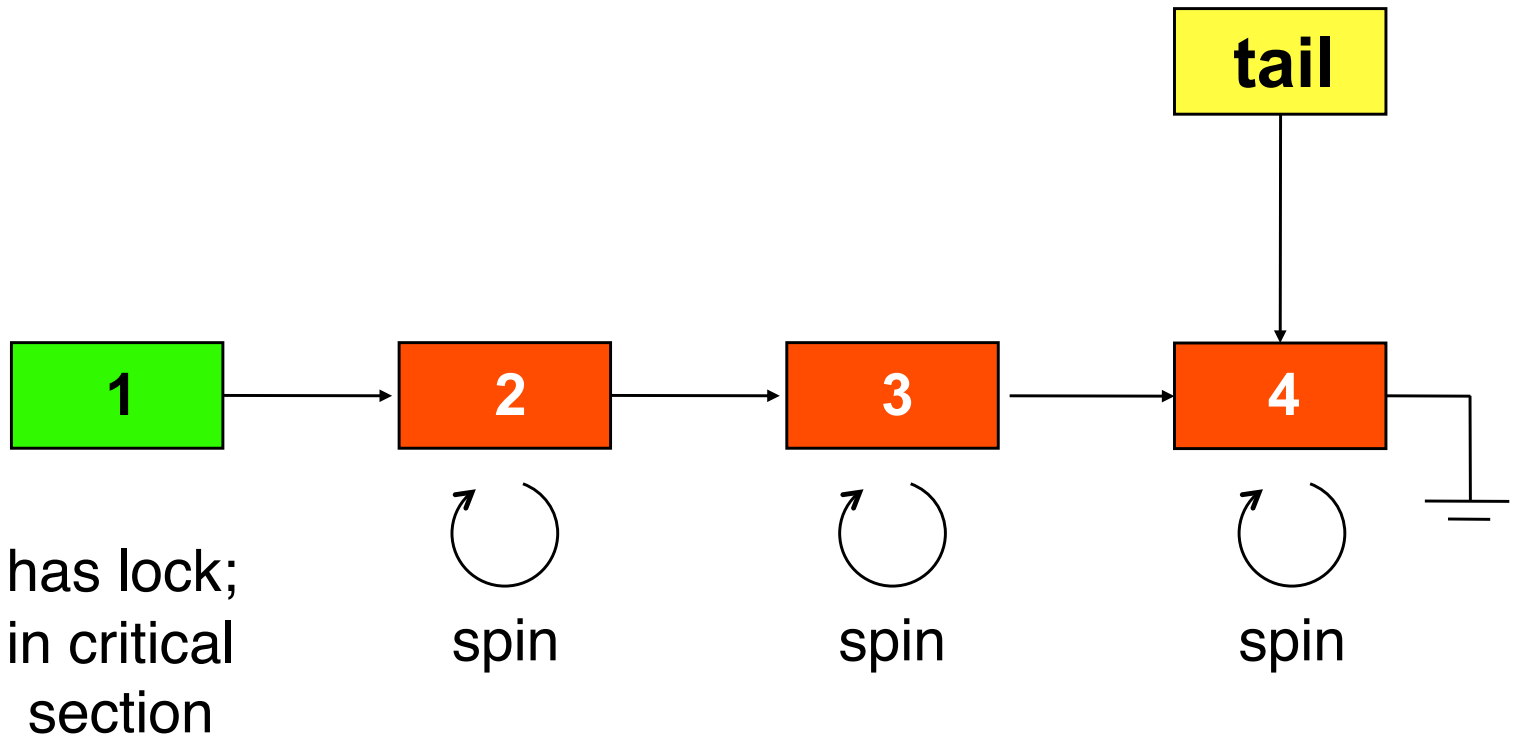
- **Process 4 swaps self into tail pointer**
- **Acquires pointer to predecessor (3) from swap on tail**
- **3 can't leave without noticing that one or more successors will link in behind it because the tail no longer points to 3**

MCS Lock In Action - III



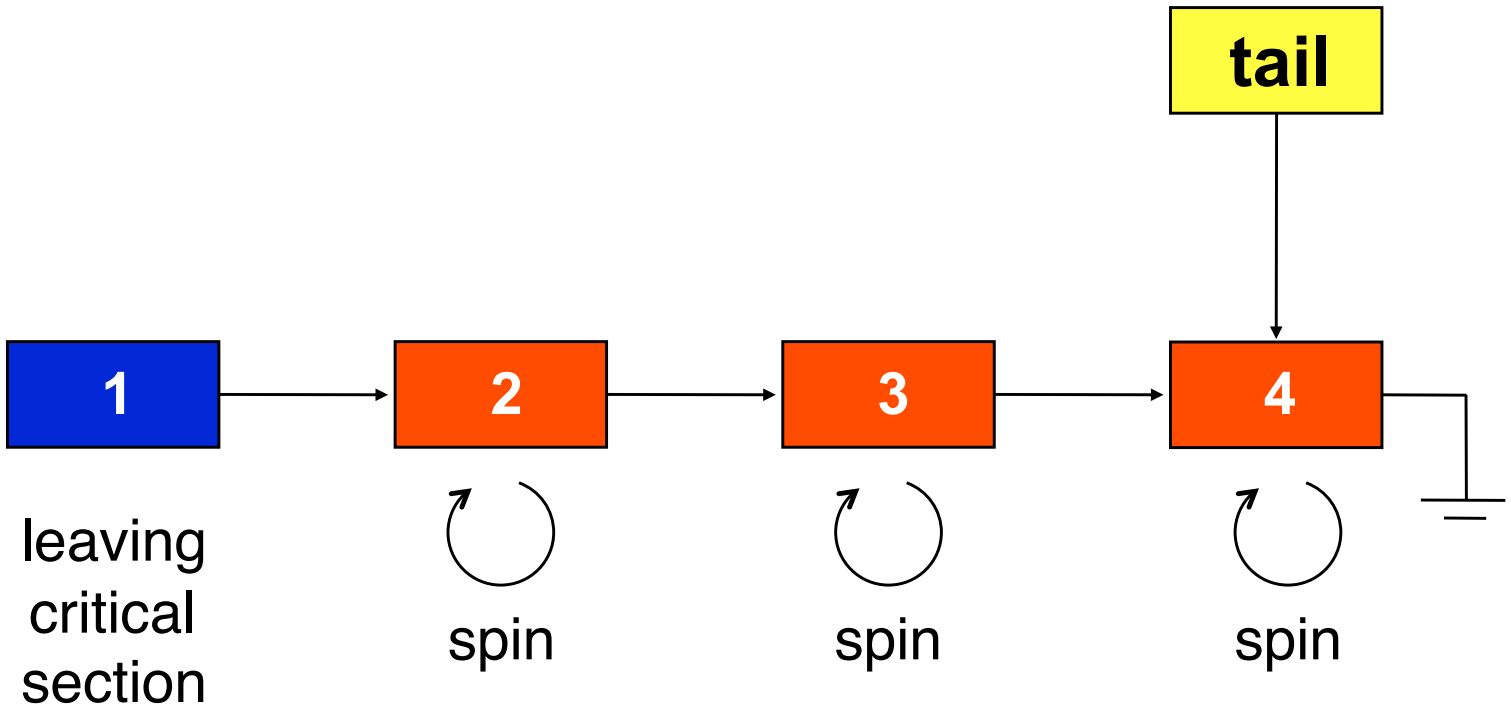
4 links behind predecessor (3)

MCS Lock In Action - IV



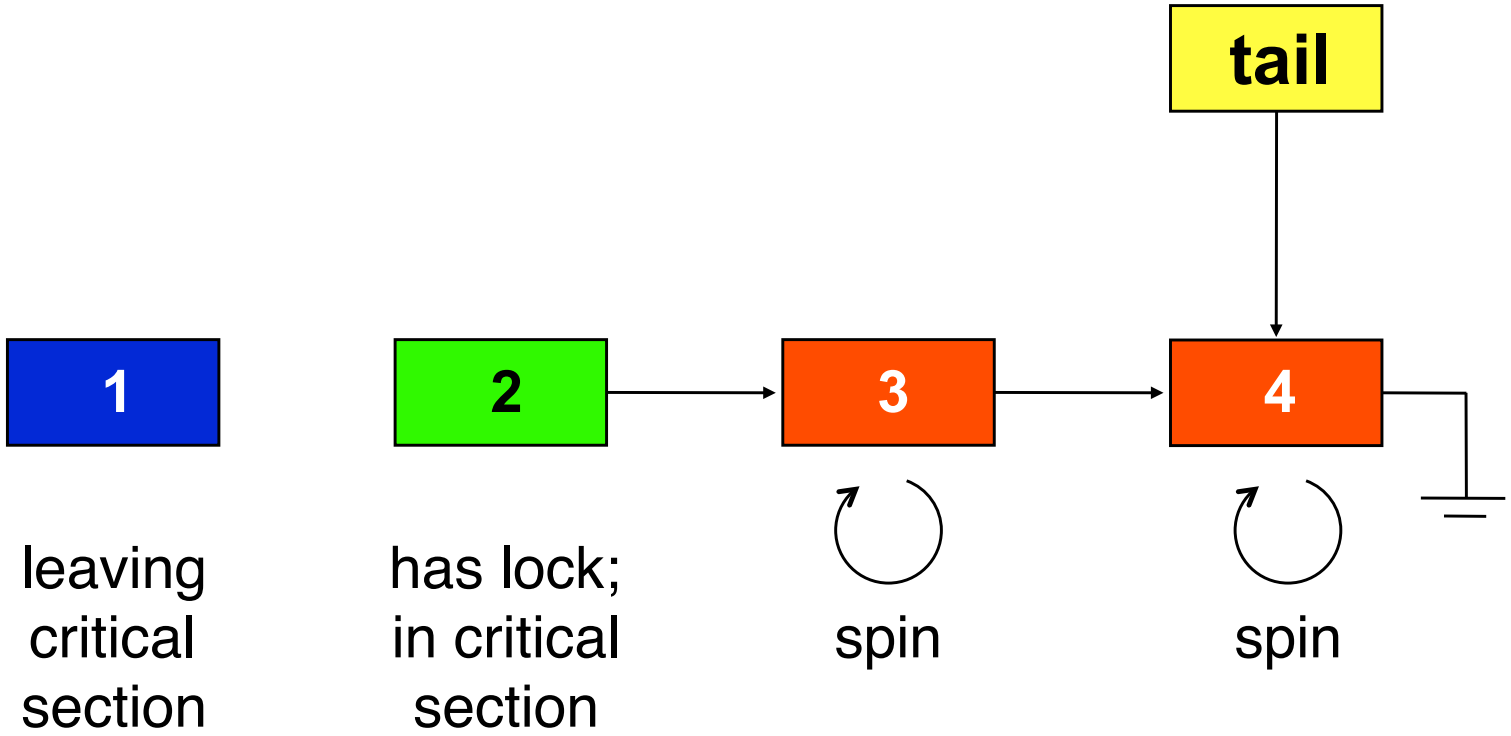
4 links now spins until 3 signals that the lock is available by setting a flag in 4's lock record

MCS Lock In Action - V



- **Process 1 prepares to release lock**
  - **—if it's next field is set, signal successor directly**
  - **—suppose 1's next pointer is still null**
    - **attempts a compare_and_swap on the tail pointer**
    - **finds that tail no longer points to self**
    - **waits until successor pointer is valid (already points to 2 in diagram)**
    - **signals successor (process 2)**

MCS Lock In Action - VI



MCS Lock Notes

- Grants requests in FIFO order
- Space: *2p* + *n* words of space for *p* processes and *n* locks
- Requires a local "queue node" to be passed in as a parameter
  - —alternatively, additional code can allocate these dynamically in acquire_lock, and look them up in a table in release_lock).
- Spins only on local locations
  - —only four “remote” references
  - —cache-coherent and non-cache-coherent machines
- Atomic primitives
  - —fetch_and_store and (ideally) compare_and_swap

ASPLOS, April 1991
ACM TOCS, February 1991

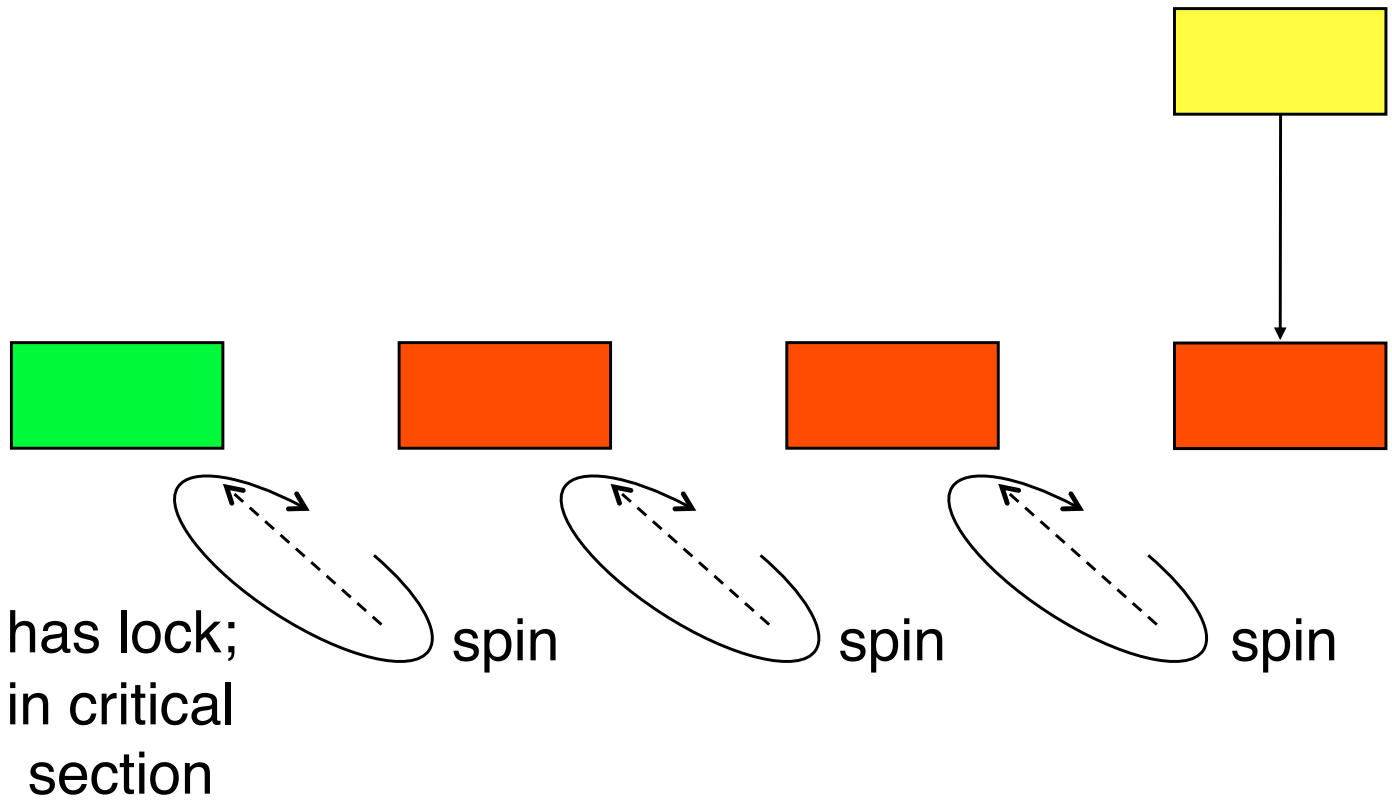
Impact of the MCS Lock

- **Local spinning technique bounds remote memory traffic**
  - —influenced virtually all practical synchronization algorithms since

- **2006 Edsger Dijkstra Prize in distributed computing**
  - —"probably the most influential practical mutual exclusion algorithm ever"
  - —"vastly superior to all previous mutual exclusion algorithms"
  - —fast, scalable, and fair in a wide variety of multiprocessor systems
  - —avoids need to pre-allocate memory for a fixed, maximum # of threads
  - —widely used
    - Linux kernel uses MCS since version 3.15
    - queue locks in Intel's Thread Building Blocks are based on MCS
    - queue locks in Intel's OpenMP runtime are based on MCS

CLH List-based Queue Lock

```
type qnode = record
  prev : ^qnode
  succ_must_wait : Boolean

type lock = ^qnode   // initialized to point to an unowned qnode

procedure acquire_lock (L : ^lock, I : ^qnode)
  I->succ_must_wait := true
  pred : ^qnode := I->prev := fetch_and_store(L, I)
  repeat while pred->succ_must_wait

procedure release_lock (ref I : ^qnode)
  pred : ^qnode := I->prev
  I->succ_must_wait := false
  I := pred          // take pred's qnode
```



CLH

CLH Queue Lock Notes

- Discovered twice, independently
  - —Travis Craig (University of Washington)
    - TR 93-02-02, February 1993
  - —Anders Landin and Eric Hagersten (Swedish Institute of CS)
    - *IPPS*, 1994
- Space: $2p + 3n$ words of space for p processes and n locks
  - —MCS lock requires $2p + n$ words
- Requires a local "queue node" to be passed in as a parameter
- Spins only on local locations on a cache-coherent machine
- Local-only spinning possible when lacking coherent cache
  - —can modify implementation to use an extra level of indirection
    (local spinning variant not shown)
- Atomic primitives: fetch_and_store

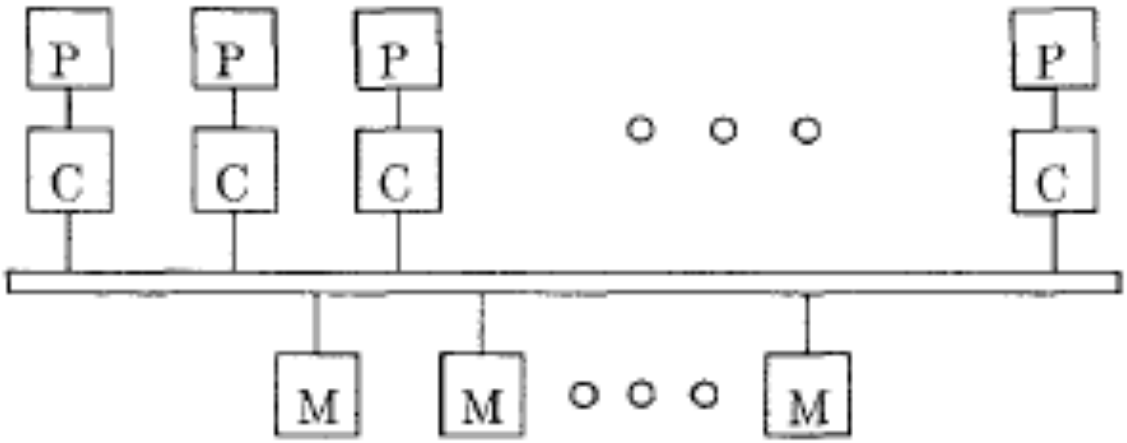
Case Study:

Evaluating Lock Implementations for the BBN Butterfly and Sequent Symmetry

J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Transactions on Computer Systems, 9(1):21-65, Feb. 1991.

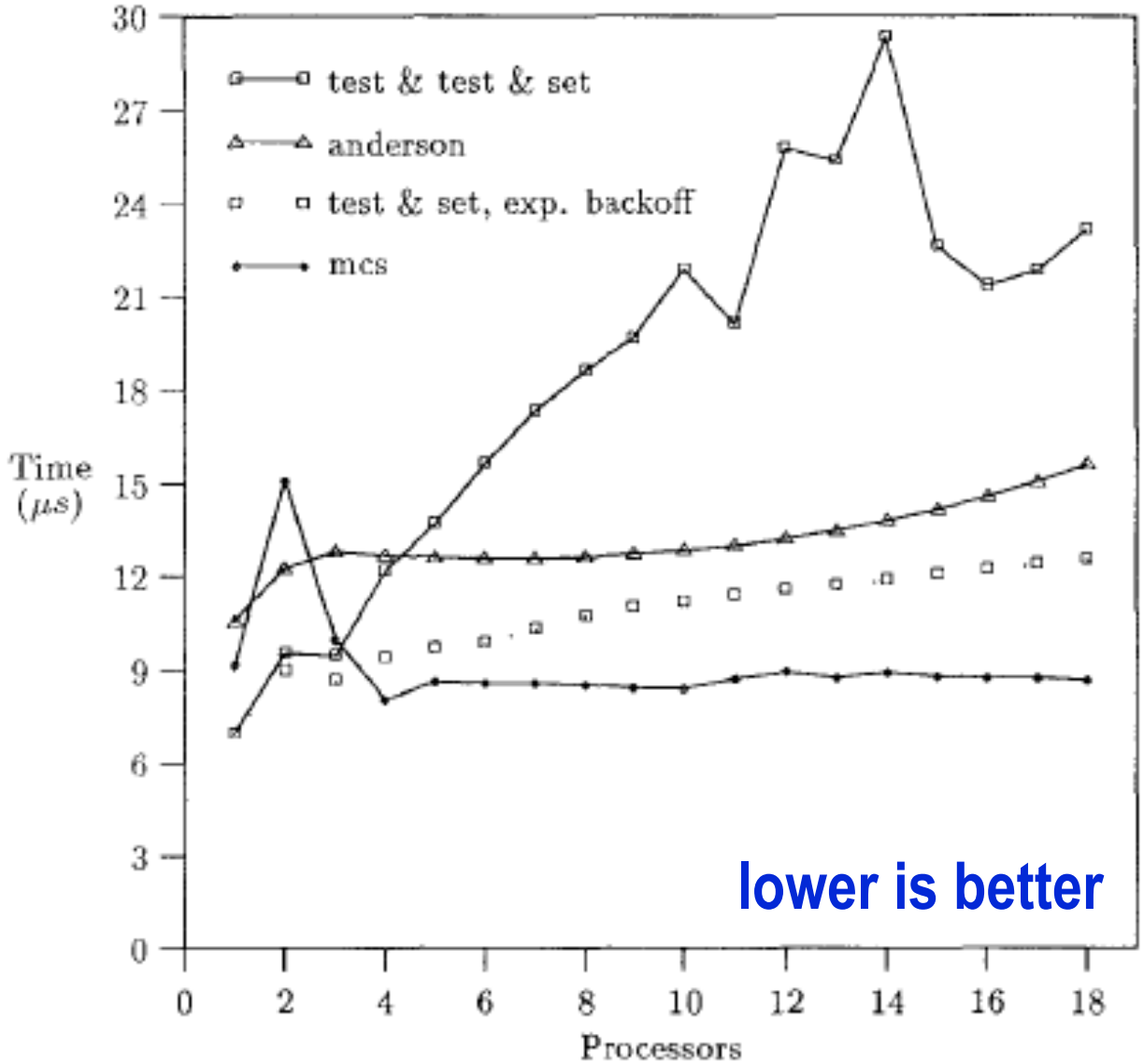
Sequent Symmetry

- 16 MHz Intel 80386
- Up to 30 CPUs
- 64KB 2-way set associative cache
- Snoopy coherence
- various logical and arithmetic ops
  - —no return values, condition codes only



Lock Comparison (Selected Locks Only)

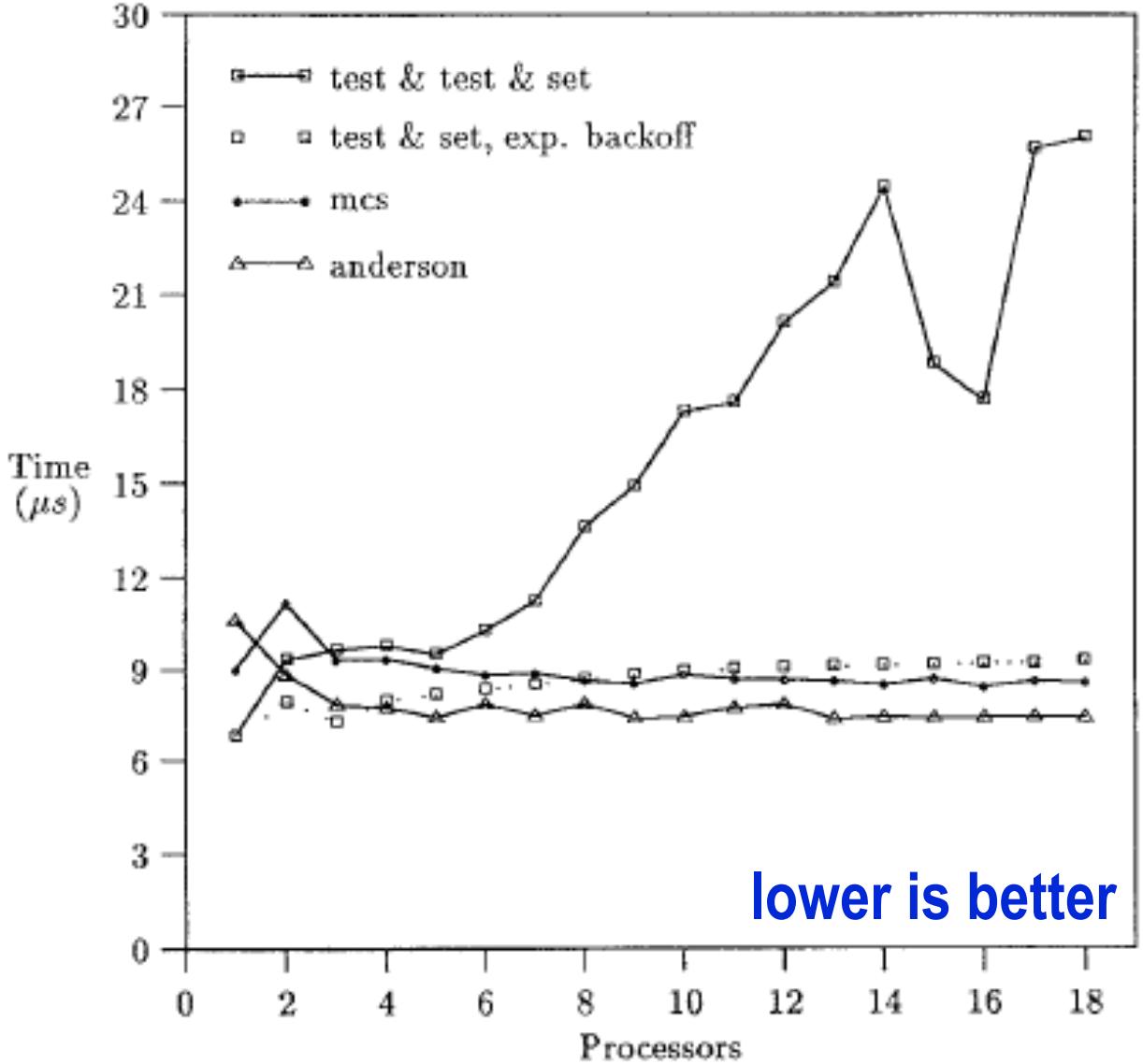
Sequent Symmetry: shared-bus, coherent caches



empty critical section

Lock Comparison (Selected Locks Only)

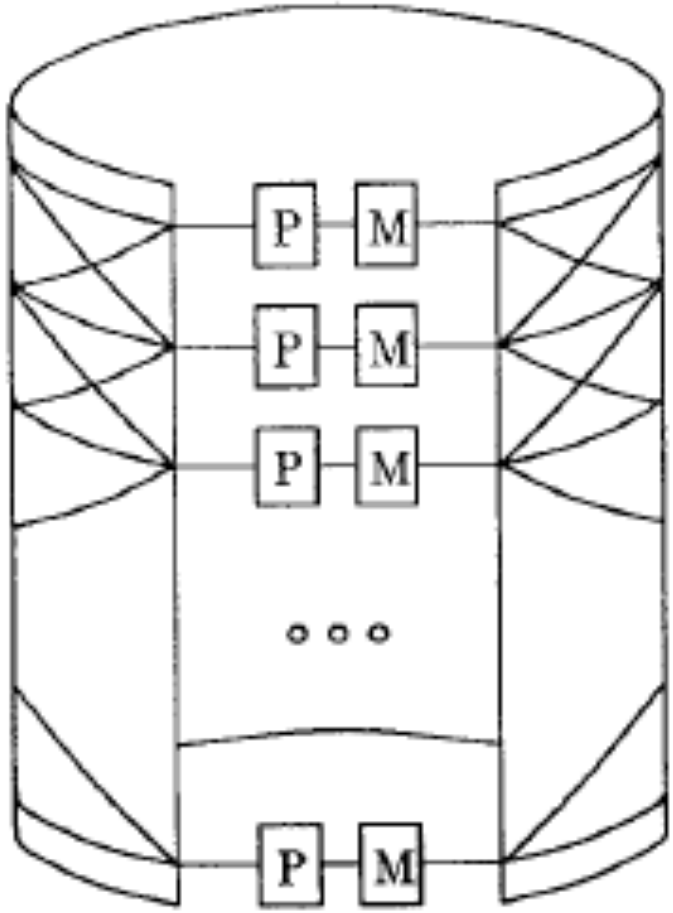
Sequent Symmetry: shared-bus, coherent caches



small critical section

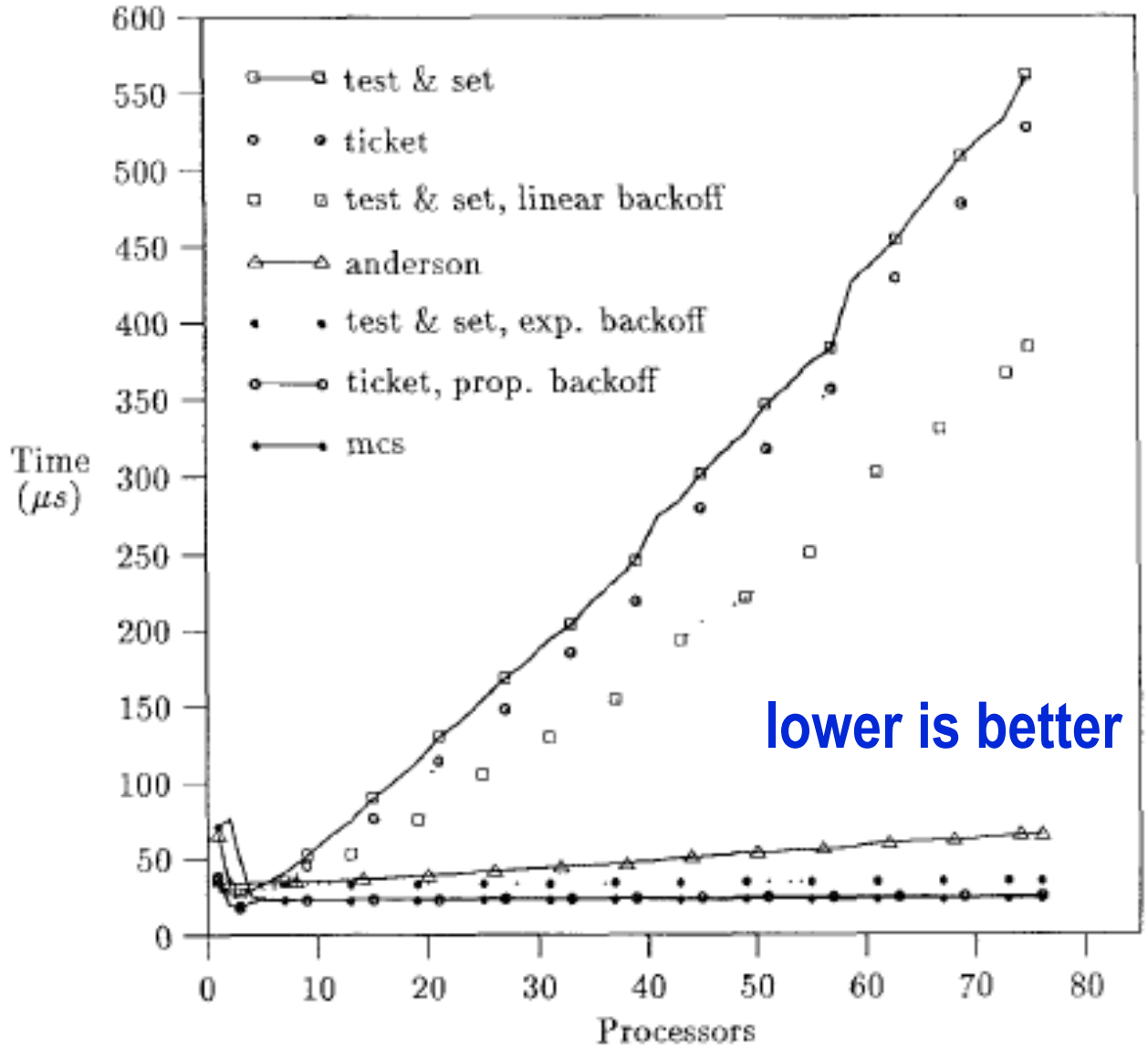
BBN Butterfly

- 8 MHz MC68000
- 24-bit virtual address space
- 1-4 MB memory per PE
- $\log_4$ depth switching network
- Packet switched, non-blocking
- Remote reference
  - —4us (no contention)
  - —5x local reference
- Collisions in network
  - —1 reference succeeds
  - —others aborted and retried later
- 16-bit atomic operations
  - —fetch_clear_then_add
  - —fetch_clear_then_xor



Lock Comparison

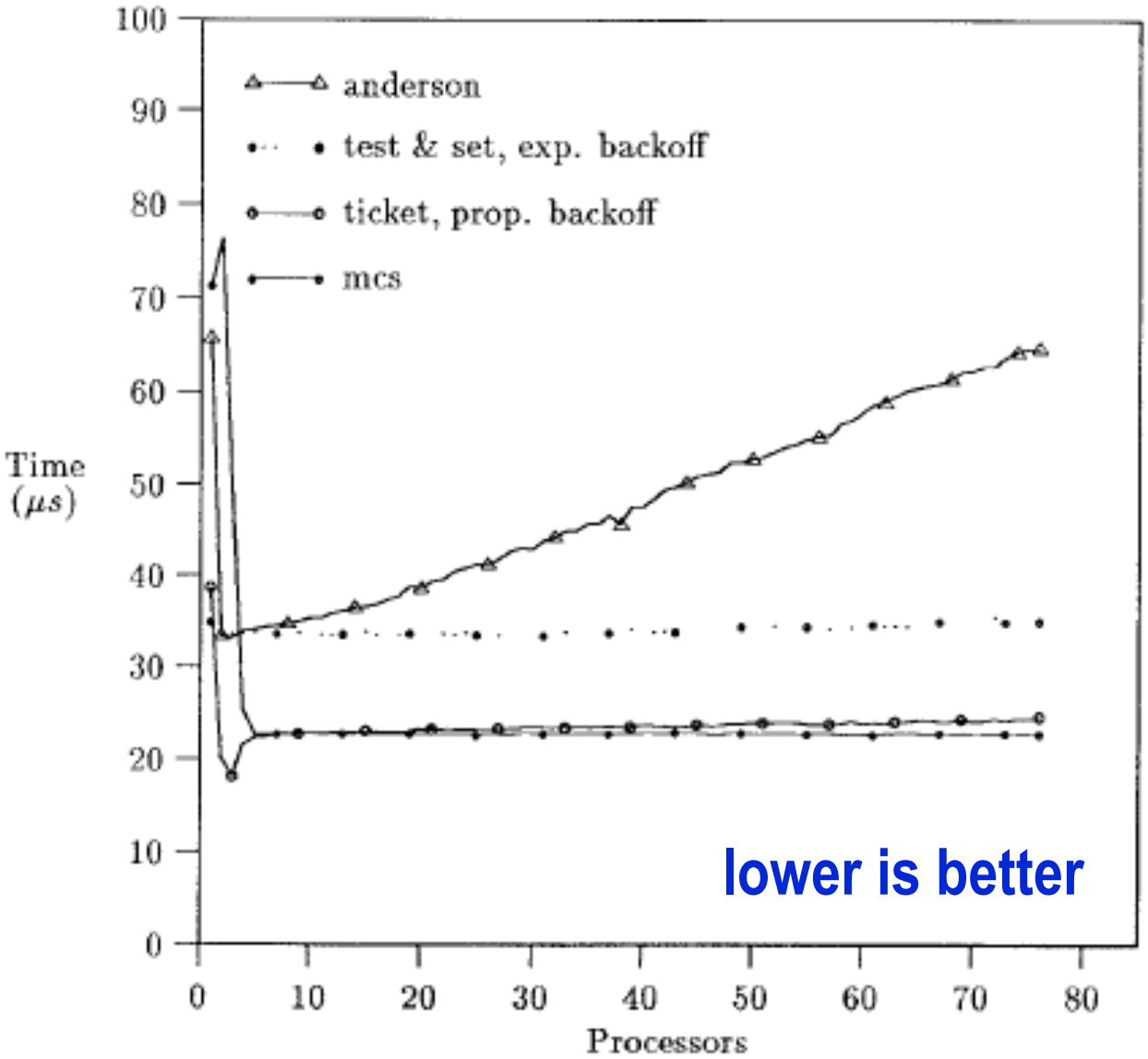
BBN Butterfly: distributed memory, no coherent caches



empty critical section

Lock Comparison (Selected Locks Only)

BBN Butterfly: distributed memory, no coherent caches



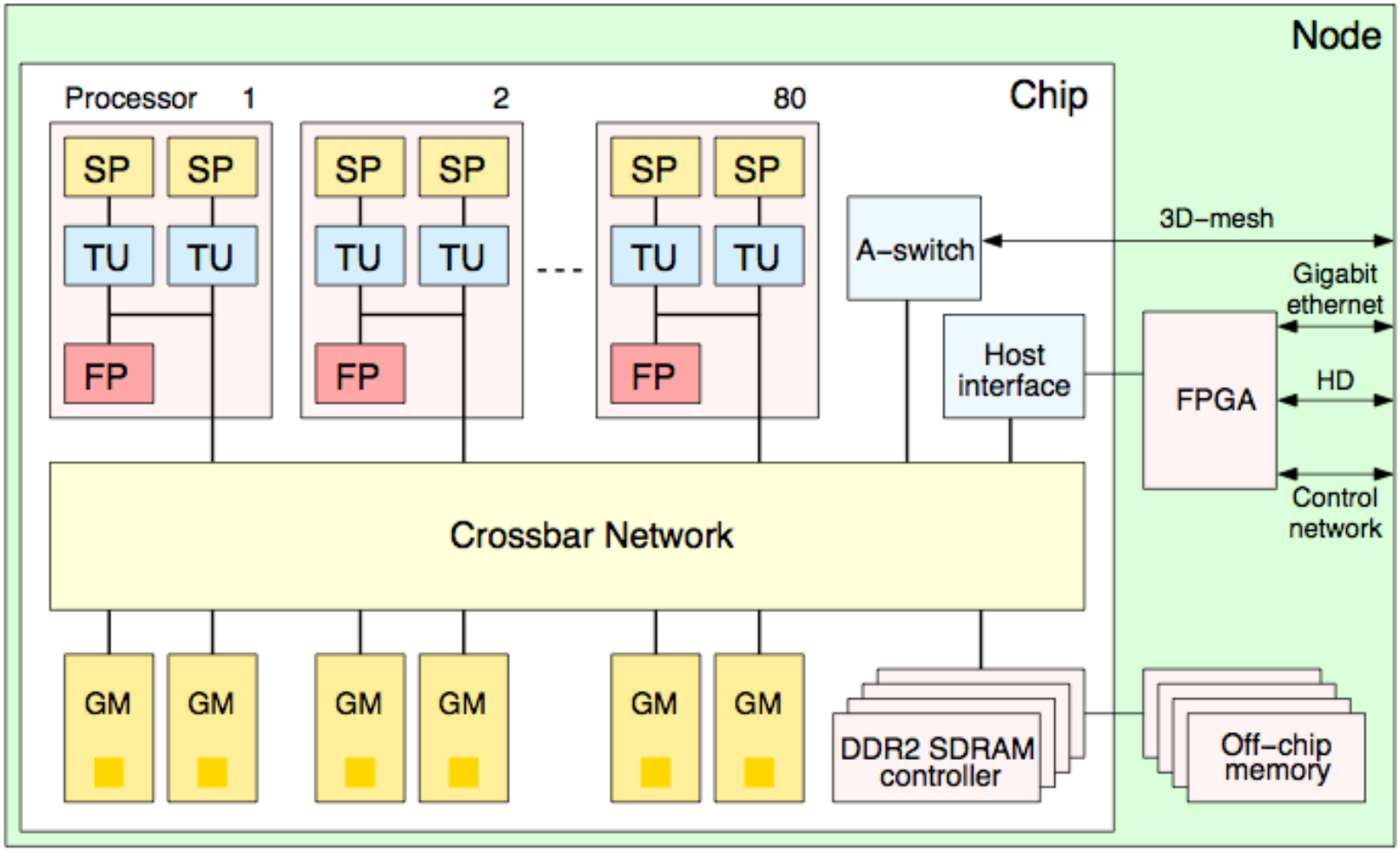
empty critical section

Case Study:

Evaluating Lock Implementations for the IBM Cyclops C64

Juan del Cuvillo, Weirong Zhu, and Guang R. Gao. Landing OpenMP on Cyclops-64: an efficient mapping of OpenMP to a many-core system-on-a-chip, ACM International Conference on Computing Frontiers, May 2-5, 2006, Ischia, Italy.

IBM Cyclops-64 Node



- **80 processors: 2 thread units (TU) each, 1 floating point (FP)**
- **No data cache; 32K instruction cache per 5 PEs**
- **Scratchpad memory (SP) + global memory (GM)**

Lock Implementations on the C64

- Test-and-set: threads spin on values in global memory
  - —plain (TS) and exponential backoff (TS-exp)
- Ticket: threads spin on values in global memory
- MCS: threads spin locally in scratch pad memory
- MCS with sleep/wakeup (MCS-SW)
  - —thread suspends after adding itself to queue
  - —constant # instructions per acquire/release pair
    - – independent of # threads contending for lock

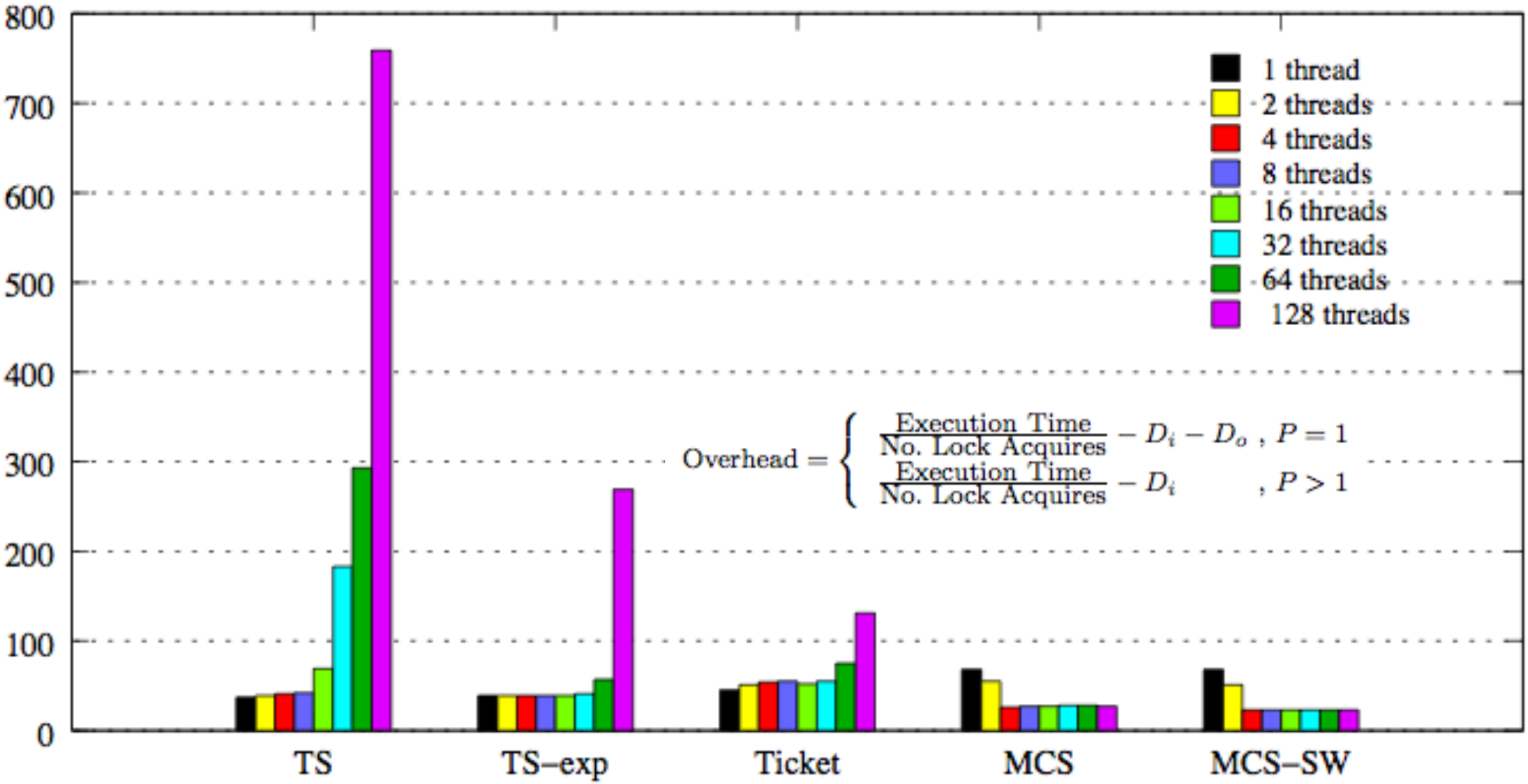
Evaluation Strategy

- Each thread performs 1K acquires and releases
- Evaluation benchmarks
  - —lock-null: no delays
  - —lock-delay: fixed delays
    - delay inside critical section = 3 x delay outside critical section
- Evaluation metrics
  - —overhead

$$\text{Overhead} = \begin{cases} \dfrac{\text{Execution Time}}{\text{No. Lock Acquires}} - D_i - D_o \ , & P = 1 \\ \dfrac{\text{Execution Time}}{\text{No. Lock Acquires}} - D_i \ , & P > 1 \end{cases}$$

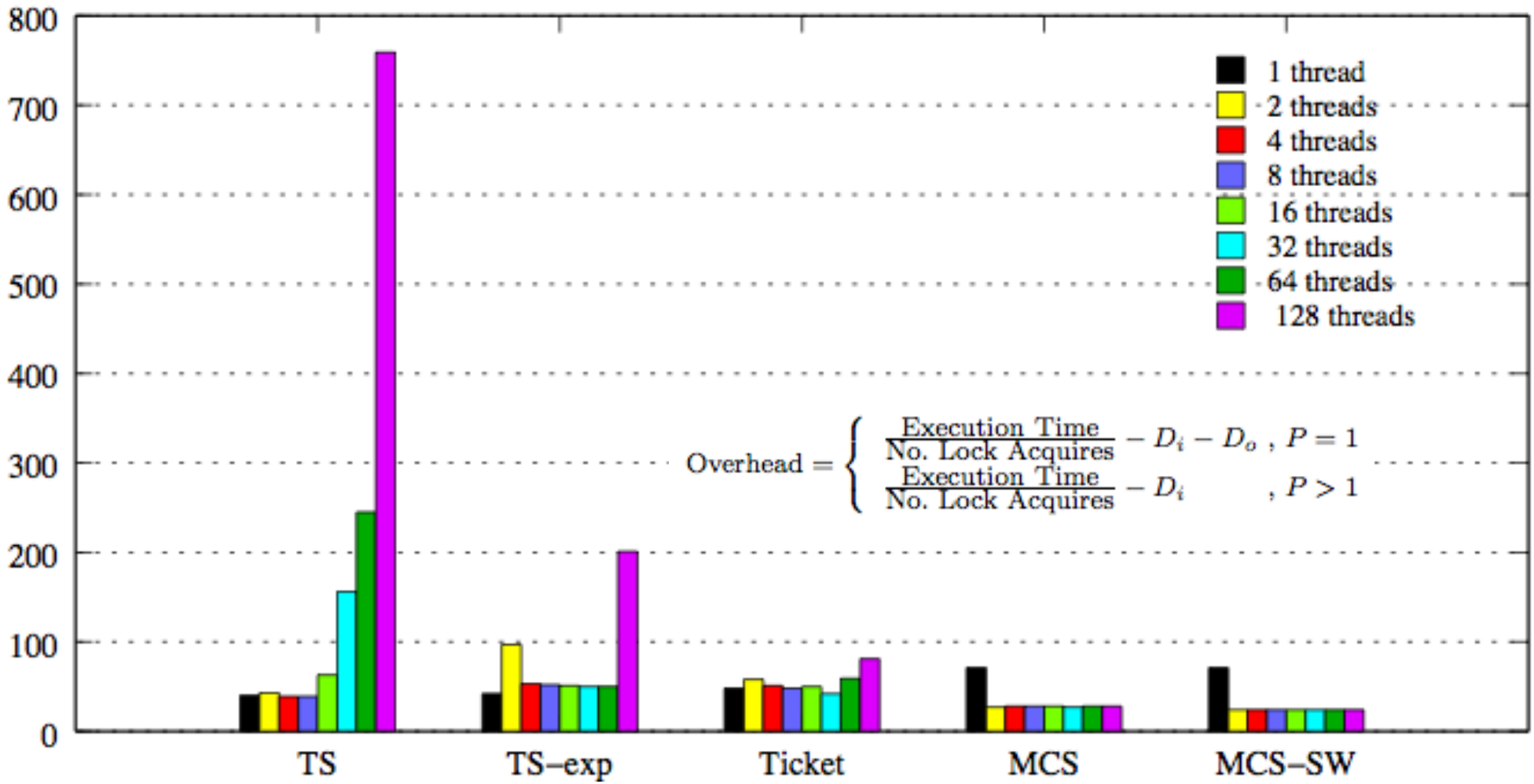
  - —contention
    - measured with software simulator
      - two or more threads compete for same resource in same cycle
        - contention counter incremented
    - important because it affects execution as well as lock acquisition

C64 lock-null Overhead



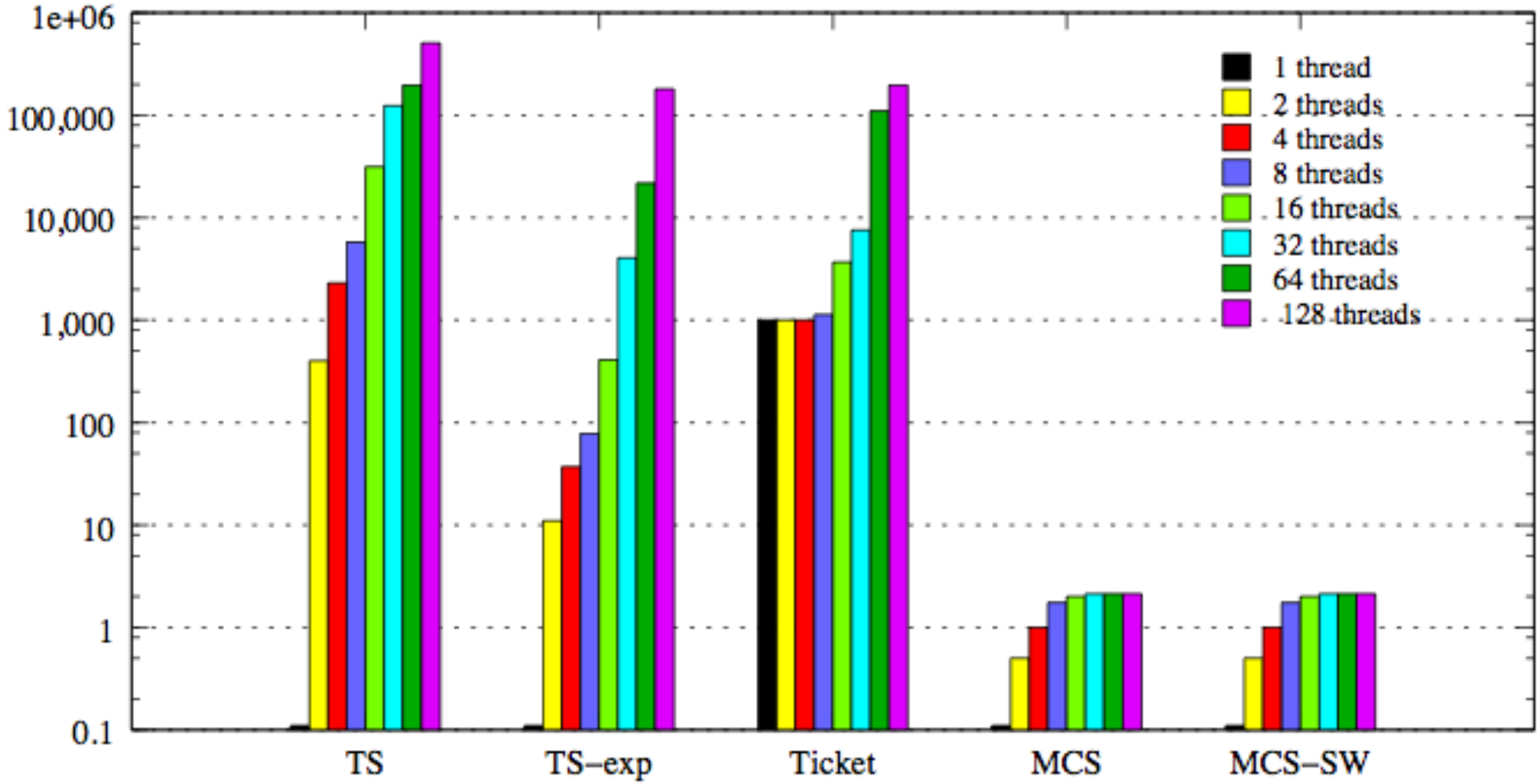
- **MCS and MCS-SW have lowest overhead**
- **MCS and MCS-SW have perfect scalability**

C64 lock-delay Overhead



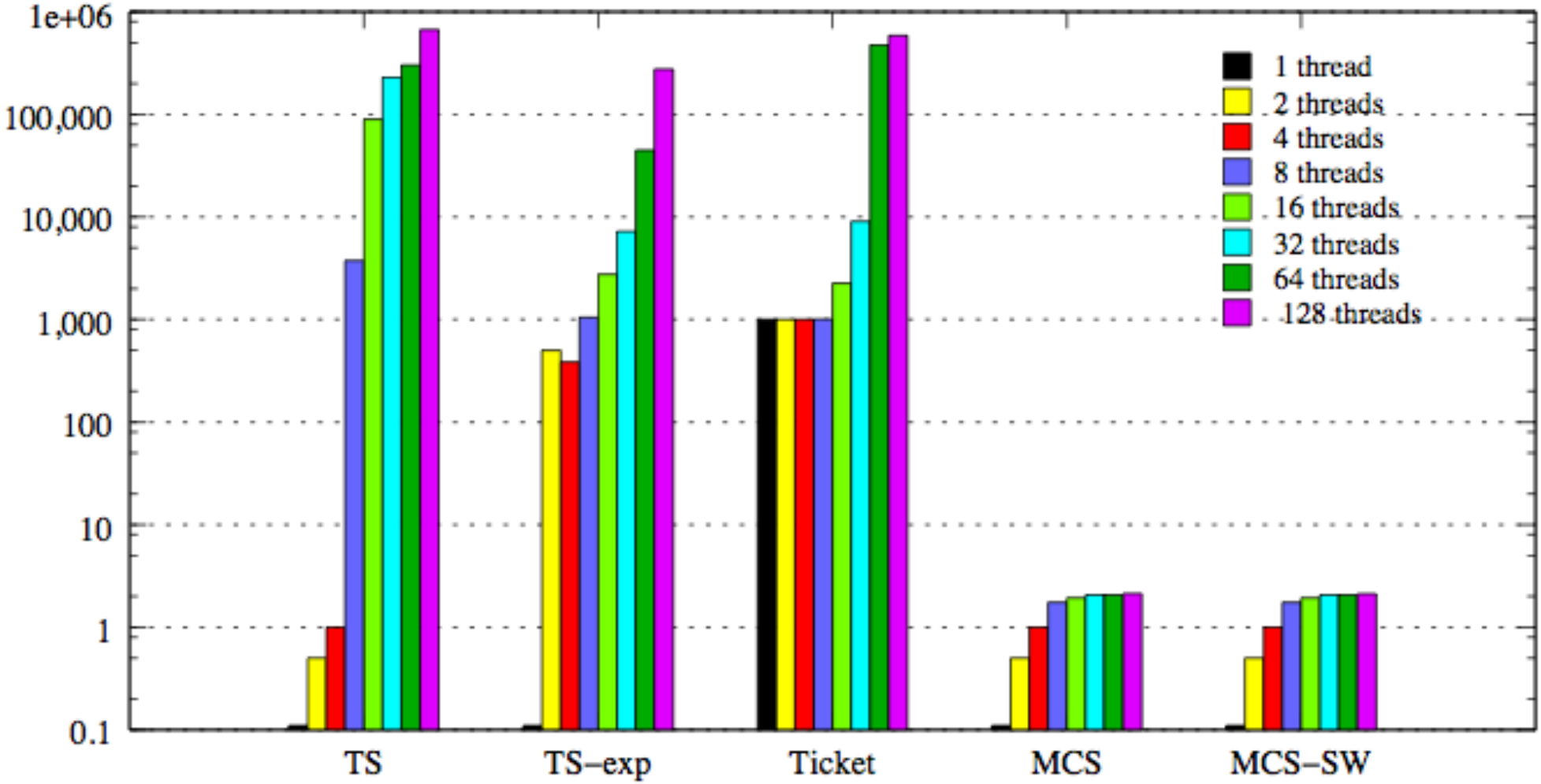
- **MCS and MCS-SW have lowest overhead**
- **MCS and MCS-SW have perfect scalability**

C64 lock-null Contention



- **Contention**
  - —2 or more threads compete for same resource in same cycle
  - —normalized by the number of threads
- **MCS and MCS-SW have lowest contention**

C64 lock-delay Contention



- **Contention**
  - —2 or more threads compete for same resource in same cycle
  - —normalized by the number of threads
- **MCS and MCS-SW have lowest contention**

C64 Lock Evaluation Summary

- **MCS and MCS-SW are superior**
- **MCS-SW is preferable**
  - **—for > 1 thread, MCS-SW overhead < MCS overhead by few cycles**
  - **—sleeping instead of spin waiting reduces power consumption**

Locks in Linux

Non-scalable Locks are Dangerous

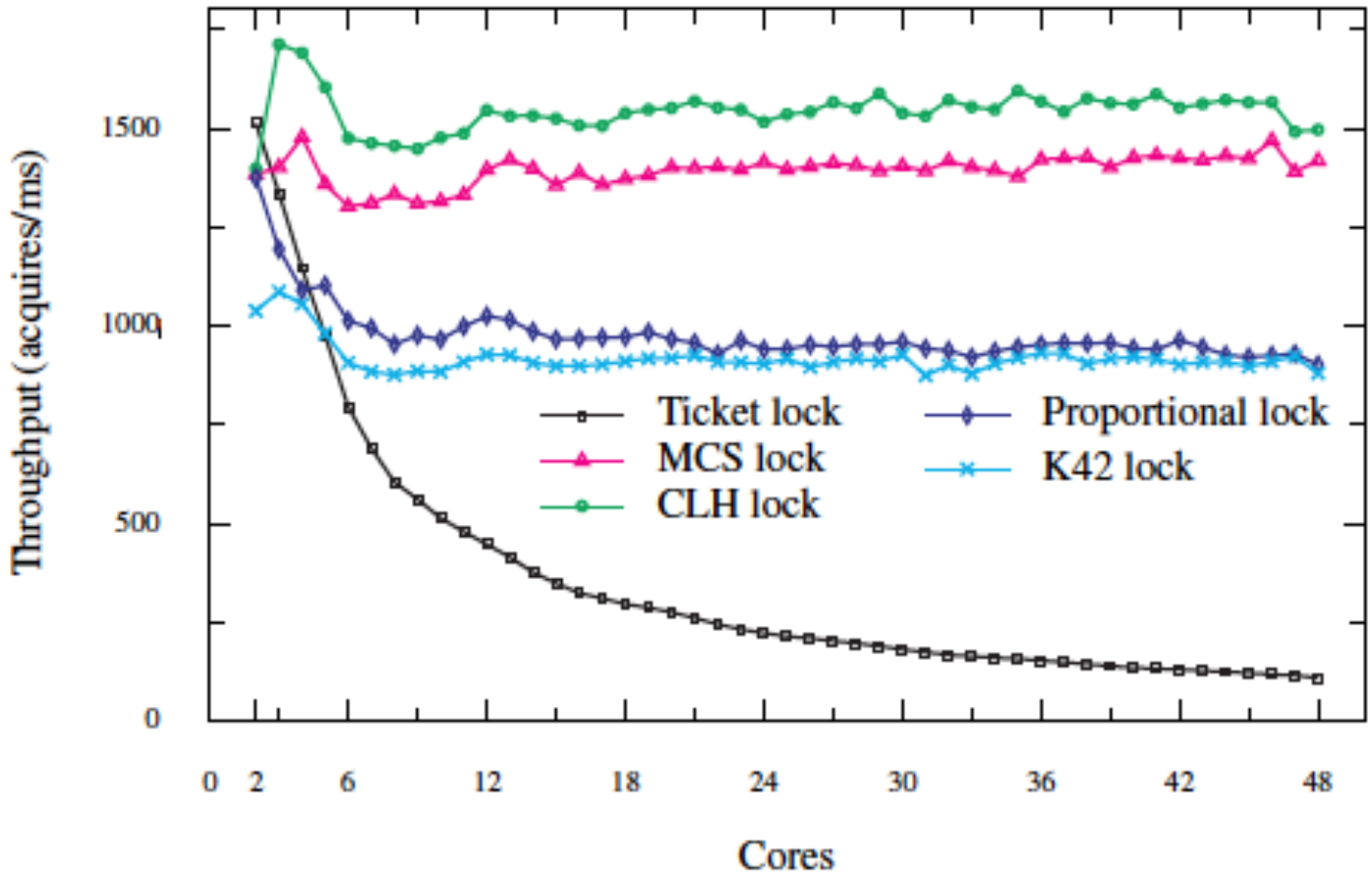


Figure 10: Throughput for cores acquiring and releasing a shared lock. Results start with two cores.

Non-scalable locks are dangerous. Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. *In the Proceedings of the Linux Symposium, Ottawa, Canada, July 2012.*

Linux Benchmarks

Benchmark	Operation time (cycles)	Top lock instance name	Acquires per operation	Average critical section time (cycles)	% of operation in critical section
FOPS	503	d_entry	4	92	73%
MEMPOP	6852	anon_vma	4	121	7%
PFIND	2099 M	address_space	70 K	350	7%
EXIM	1156 K	anon_vma	58	165	0.8%

Figure 3: The most contended critical sections for each Linux microbenchmark, on a single core.

FOPS creates a single file and starts one process on each core. Each thread repeatedly opens and closes the file.

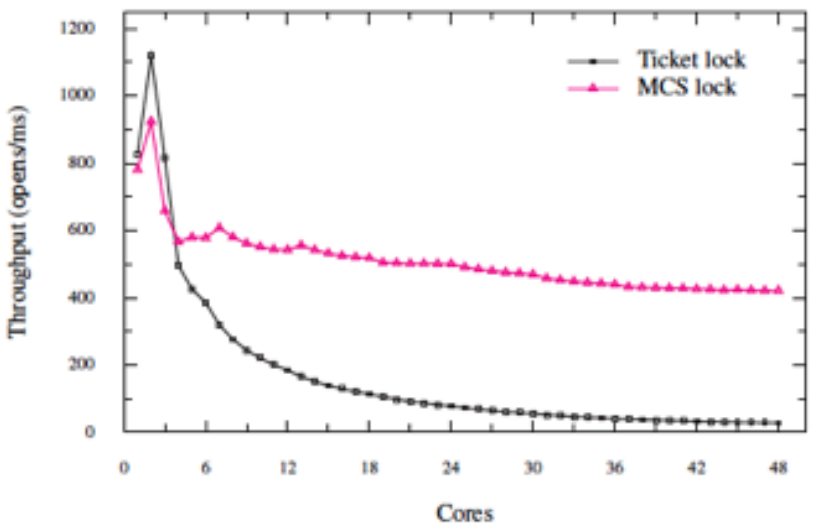
PFIND searches for a file by executing several instances of the GNU find utility. PFIND takes a directory and filename as input, evenly divides the directories in the first level of input directory into per-core inputs, and executes one instance of find per core, passing in the input directories. Before we execute the PFIND, we create a balanced directory tree so that each instance of find searches the same number of directories.

MEMPOP creates one process per core. Each process repeatedly mmaps 64 kB of memory with the MAP_POPULATE flag, then munmaps the memory. MAP_POPULATE instructs the kernel to allocate pages and populate the process page table immediately, instead of doing so on demand when the process accesses the page.

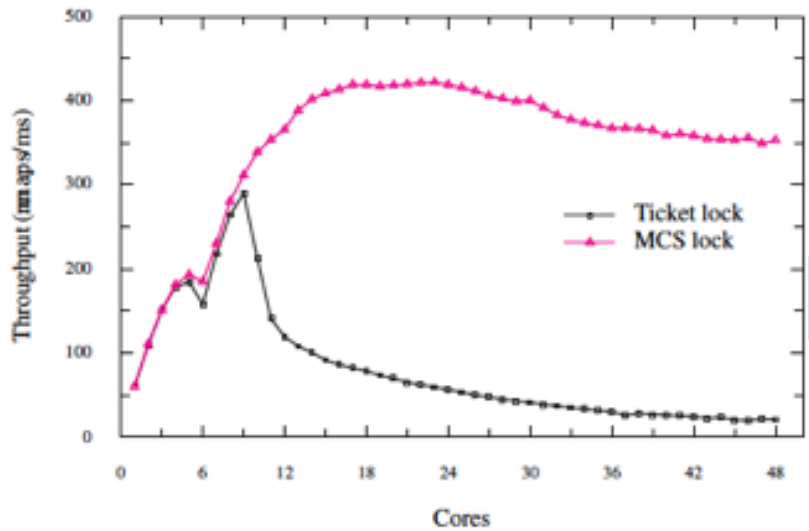
EXIM is a mail server. A single master process listens for incoming SMTP connections via TCP and forks a new process for each connection, which accepts the incoming message. We use the version of EXIM from MOSBENCH [3].

Non-scalable locks are dangerous. Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. *In the Proceedings of the Linux Symposium, Ottawa, Canada, July 2012.*

MCS vs. Ticket Lock in Linux

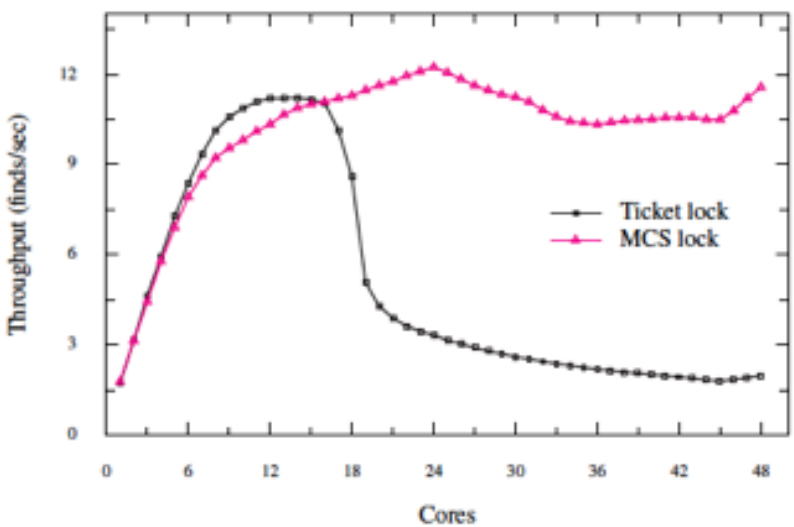


(a) Performance for FOPS.

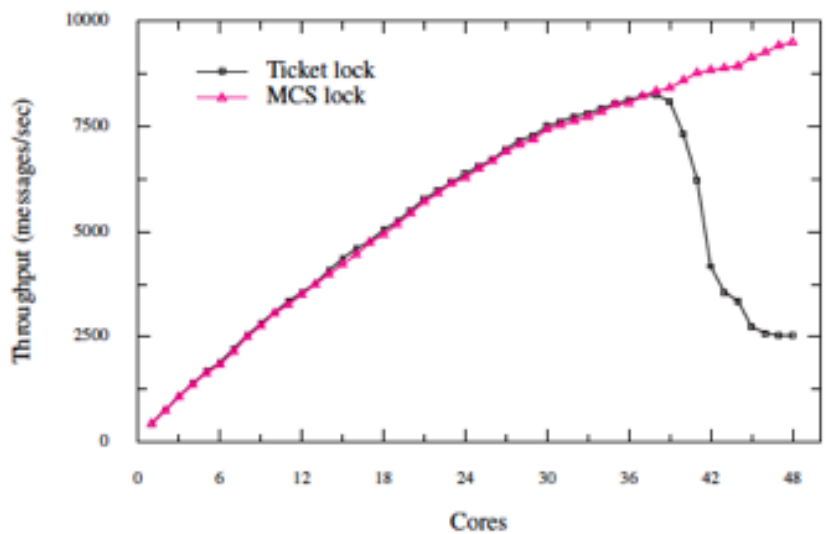


(b) Performance for MEMPOP.

16.6x



(c) Performance for PFIND.



(d) Performance for EXIM.

3.7x

Non-scalable locks are dangerous. Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. *In the Proceedings of the Linux Symposium, Ottawa, Canada, July 2012.*

Locks on Blue Gene/Q

Blue Gene Q Atomic Primitives

- **Core atomics**
  - —load linked
  - —store conditional
- **L2 atomic operations**
  - —L2_ATOMIC_OPCODE_LOAD
  - —L2_ATOMIC_OPCODE_LOAD_CLEAR
  - —L2_ATOMIC_OPCODE_LOAD_INCREMENT
  - —L2_ATOMIC_OPCODE_LOAD_DECREMENT
  - — L2_ATOMIC_OPCODE_LOAD_INCREMENT_BOUNDED
  - —L2_ATOMIC_OPCODE_LOAD_DECREMENT_BOUNDED
  - —L2_ATOMIC_OPCODE_LOAD_INCREMENT_IF_EQUAL
  - —L2_ATOMIC_OPCODE_STORE
  - —L2_ATOMIC_OPCODE_STORE_TWIN
  - —L2_ATOMIC_OPCODE_STORE_ADD
  - —L2_ATOMIC_OPCODE_STORE_ADD_COHERENCE_ON_ZERO
  - —L2_ATOMIC_OPCODE_STORE_OR
  - —L2_ATOMIC_OPCODE_STORE_XOR
  - —L2_ATOMIC_OPCODE_STORE_MAX
  - —L2_ATOMIC_OPCODE_STORE_MAX_SIGN_VALUE

Blue Gene Q L2 Ticket Lock

```
__INLINE__ void L2_LockAcquireNoSync(L2_Lock_t *l)
{
  uint64_t ticket=L2_AtomicLoadIncrement(&l->ticket);
  while(l->serving != ticket);
}

__INLINE__ void L2_LockAcquire(L2_Lock_t *l)
{
  L2_LockAcquireNoSync(l);
  isync();
}

__INLINE__ void L2_LockReleaseNoSync(L2_Lock_t *l)
{
  L2_AtomicStoreAdd(&l->serving, 1);
}

__INLINE__ void L2_LockRelease(L2_Lock_t *l)
{
  ppc_msync();
  L2_LockReleaseNoSync(l);
}
```

WakeUp Unit

- **Enables SMT threads to be suspended, waiting for an event**
  - **—lighter weight than wake on interrupt: no context switching**
  - **—reduces power and issue slot consumption**
- **Wait instruction provides wakeup under software control**
- **Idiom**
  - **—reserve a cache line with "load linked"**
  - **—wakeup when reservation is canceled by write to the cache line**
  - **—if no reservation present, wait has no effect**
  - **—interrupts also end a wait; not automatically resumed**

```
loop:
    lwarx       r4, 0, r3     # Load and reserve.
    cmpwi       r4, 0         # If it is already non-zero then exit.
    bne-        exit
    wait 1                    # Wait until the reservation is lost.
    stwcx.      r4, 0, r3     # Store old value if still reserved.
    beq-        loop          # Loop if reservation exists.

exit:           ...
```

r3 contains address of flag initialized to zero

Each thread can have a unique cache line for this reservation wake-up.

Figure credit: A2 Processor User's Manual for Blue Gene/Q

CLH Queue Lock for BG/Q

```
type qnode = record
  qnode *prev
  Boolean succ_must_wait

type qnode *Lock  // initialized to point to an unowned qnode

procedure acquire_lock(Lock *L, qnode *I)
  I->succ_must_wait := true
  qnode *pred
  loop   // pred = fetch_and_store(L, I)
    pred := LL(L)
    if SC(L,I)
      break
  I->prev := pred
  loop
    Boolean flag := LL(pred->succ_must_wait)
    if flag = false
      return
    waitrsv  // for reservation to be cleared
```

MCS Acquire with WAIT

```
0x1001e60   li      r0,0
    0x1001e64   std     r0,8(r4)
    0x1001e68   lwsync
    0x1001e6c   ldarx   r5,0,r3
    0x1001e70   stdcx.  r4,0,r3
    0x1001e74   bne     0x1001e6c
    0x1001e78   cmpdi   r5,0
    0x1001e7c   beq     0x1001ea4
    0x1001e80       li      r0,1
    0x1001e84       std     r0,0(r4)
    0x1001e88       lwsync
    0x1001e8c       std     r4,8(r5)
    0x1001e90       ldarx   r0,0,r4
    0x1001e94           cmpdi   r0,1
    0x1001e98           bne     0x1001ea4
    0x1001e9c           waitrsv
    0x1001ea0           b       0x1001e90
    0x1001ea4   isync
    0x1001ea8   blr
```

l->next = nil

remote spin

pred = fetch_and_store(L, l)

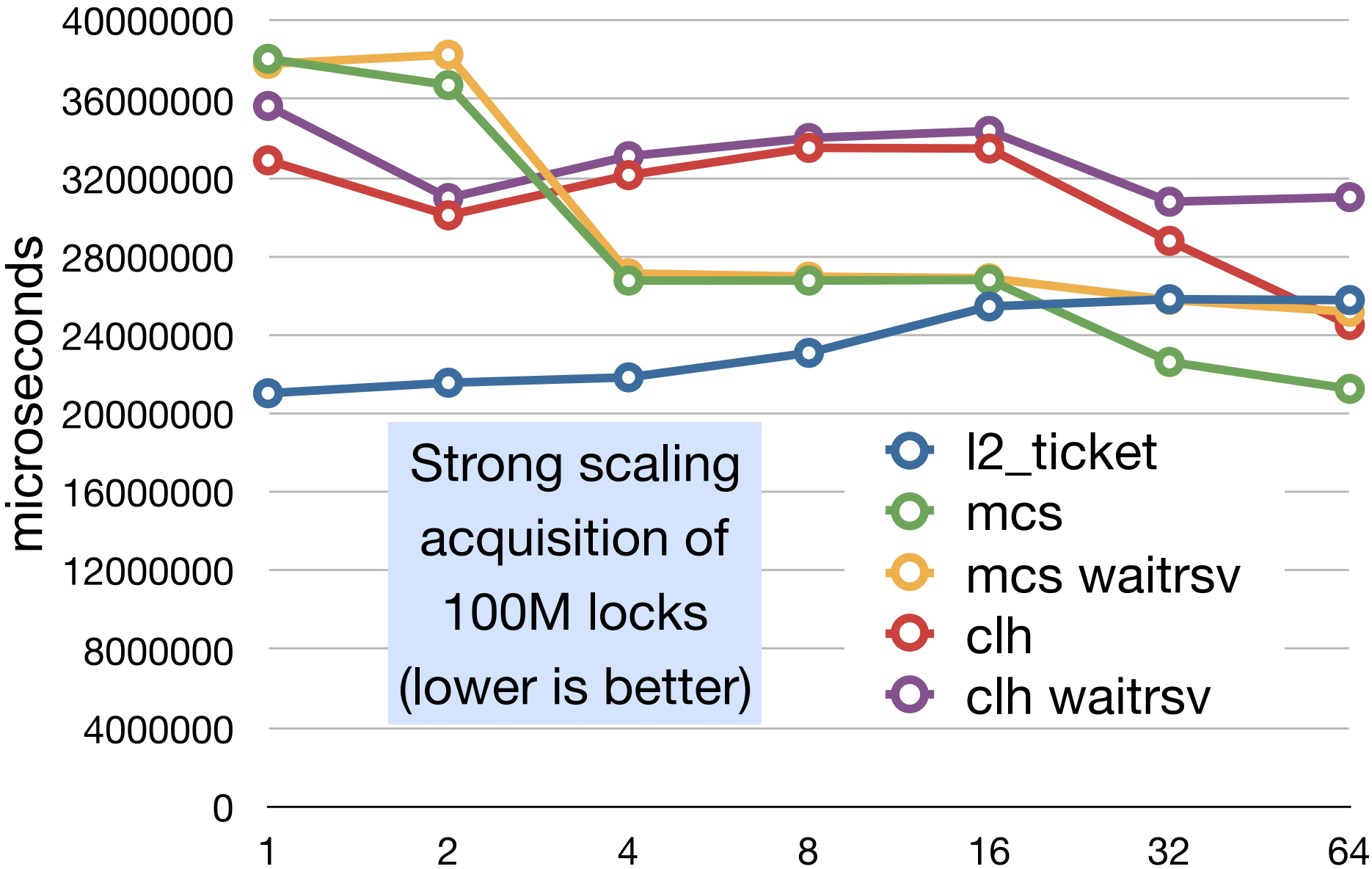
if pred != nil

l->locked = true

pred->next = l

loop
    locked = LL(&(l->locked)
    if (locked != true) break
    waitrsv(); // wait for change

Blue Gene Q Lock Scalability



John Mellor-Crummey, March 2013

Barriers

- Motivating barriers
- Barrier overview
- Performance issues
- Software barrier algorithms
  - —centralized barrier with sense reversal
  - —combining tree barrier
  - —dissemination barrier
  - —tournament barrier
  - —scalable tree barrier

Barriers

- **Definition**
  - —wait for all to arrive at point in computation before proceeding
- **Why?**
  - —separate phases of a multi-phase algorithm
- **Duality with mutual exclusion**
  - —include all others, rather than exclude all others

Technique 1: Sense Reversal

- **Problem: reinitialization**
  - —each time a barrier is used, it must be reset
- **Difficulty: odd and even barriers can overlap in time**
  - —some processes may still be exiting the kth barrier
  - —other processes may be entering the $k+1^{st}$ barrier
  - —how can one reinitialize?
- **Solution: sense reversal**
  - —terminal state of one phase is initial state of next phase
  - —e.g.
    - – odd barriers wait for `flag` to transition from true to false
    - – even barriers wait for `flag` to transition from false to true

Sense-reversing Centralized Barrier

```
shared count : integer := P
shared sense : Boolean := true

processor private local_sense : Boolean := true

procedure central_barrier
   // each processor toggles its own sense
   local_sense := not local_sense
   if fetch_and_decrement (&count) = 1
      count := P
      sense := local_sense  // last processor toggles global sense
   else
      repeat until sense = local_sense
```

Centralized Barrier Operation

- **Each arriving processor decrements count**
- **First** P-1 **processors**
  - **—wait until sense has a different value than previous barrier**
- **Last processor**
  - **—resets count**
  - **—reverses sense**
- **Argument for correctness**
  - **—consecutive barriers can't interfere**
    - **all operations on count occur before sense is toggled to release waiting processors**

Barrier Evaluation Criteria

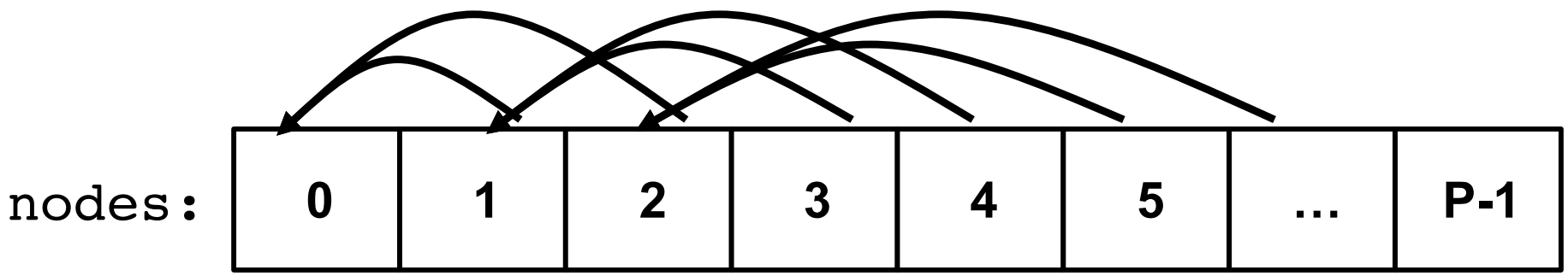
- Length of critical path: how many operations
- Total number of network transactions
- Space requirements
- Implementability with given atomic operations

Assessment: Centralized Barrier

- $\Omega(p)$ operations on critical path
- All spinning occurs on single shared location
- # busy wait accesses typically >> minimum
  - —process arrivals are generally staggered
  - —on cache-coherent multiprocessors
    - – spinning may be OK
  - —on machines without coherent caches
    - – memory and interconnect contention from spinning may be unacceptable
- Constant space
- Atomic primitives: fetch_and_decrement
- Similar to
  - —code employed by Hensgen, Finkel, and Manber (*IJPP*, 1988)
  - — sense reversal technique attributed to Isaac Dimitrovsky
    - – *Highly Parallel Computing*, Almasi and Gottlieb, Benjamin / Cummings, 1989

Software Combining Tree Barrier 1/2

```
type node = record
  k : integer                    // fan-in of this node
  count : integer                // initialized to k
  nodesense : Boolean            // initially false
  parent : ^node                 // pointer to parent node; nil if root

shared nodes : array [0..P-1] of node
// each element of nodes allocated in a different memory module or cache line
```




```
processor private sense : Boolean  := true
processor private mynode : ^node  // my group's leaf in the tree
```

Yew, Tzeng, and Lawrie. *IEEE TC*, 1987.

Software Combining Tree Barrier 2/2

```
procedure combining_barrier
  combining_barrier_aux(mynode)          // join the barrier
  sense := not sense                     // for next barrier

procedure combining_barrier_aux(nodepointer : ^node)
  with nodepointer^ do
    if fetch_and_decrement(&count) = 1  // last to reach this node
      if parent != nil
        combining_barrier_aux(parent)
      count := k                         // prepare for next barrier
      nodesense := not nodesense         // release waiting processors
    repeat until nodesense = sense
```

Operation of Software Combining Tree

- Each processor begins at a leaf node
- Decrements its leaf count variable
- Last descendant to reach each node in the tree continues upward
- Processor that reaches the root begins wakeup
  - —reverse wave of updates to nodesense flags
- When a processor wakes
  - —retraces its path through tree
  - —unblocking siblings at each node along path
- Benefits
  - —can significantly decrease memory contention
    - – distributes accesses across memory modules of machine
  - —can prevent tree saturation in multi-stage interconnect
    - – form of network congestion in multi-stage interconnects
- Shortcomings
  - —processors spin on locations not statically determined
  - —multiple processors spin on same location

Assessment: Software Combining Tree

- Ω*(log p)* operations on critical path
- *Total remote operations*
  - —*O(p)* on cache-coherent machine
  - —unbounded on non-cache-coherent machine
- *O(p)* space
- Atomic primitives: fetch_and_decrement

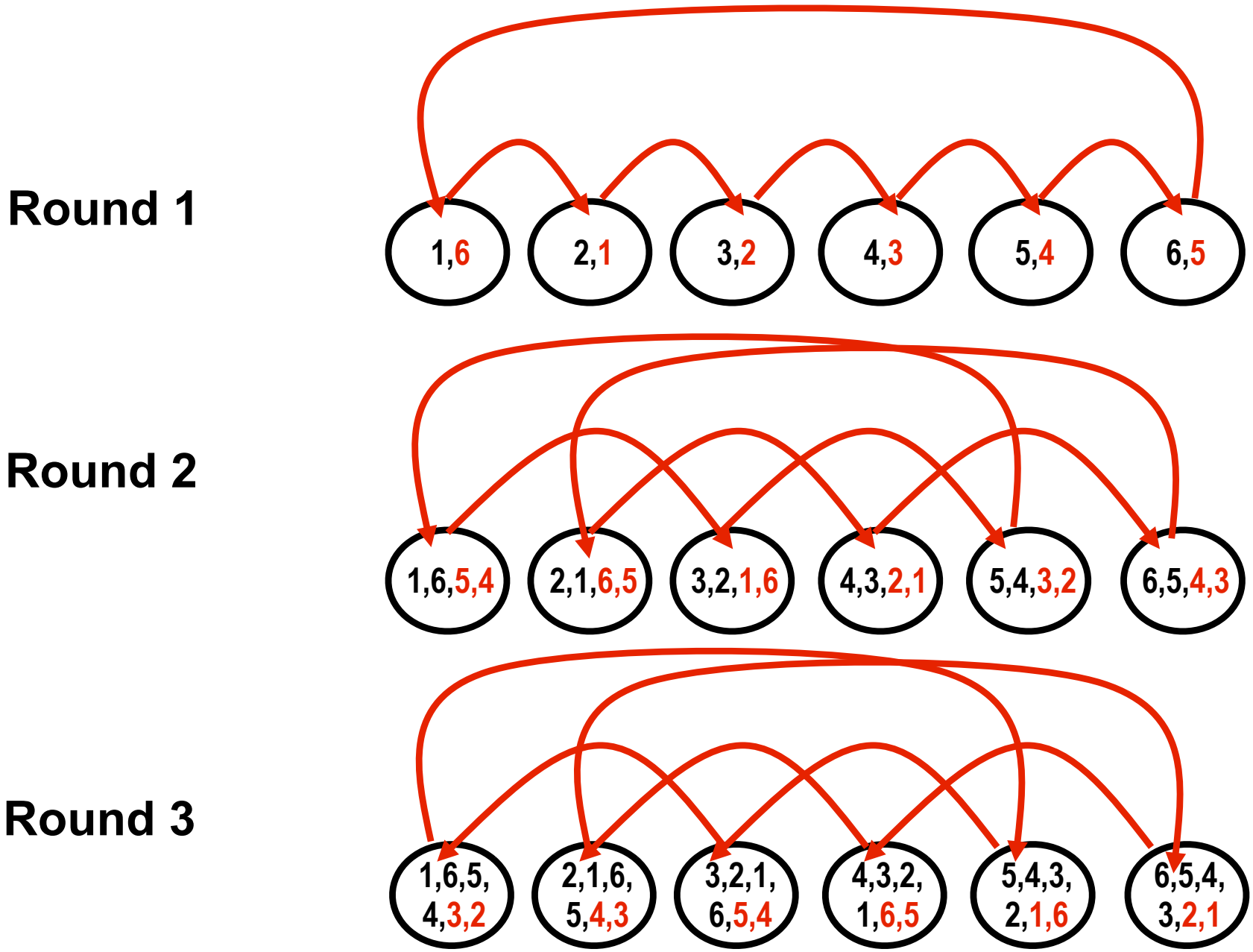
Dissemination Barrier Algorithm

- for k = 0 to ceiling($\log_2$ P)
  - —processor i signals processor $(i + 2^k) \bmod P$
    - synchronization is <u>not</u> pairwise
  - —processor i waits for signal from $(i - 2^k) \bmod P$

- Does <u>not</u> require $P = 2^k$

Hensgen, Finkel, Manber. IJPP 1988.

Dissemination Barrier in Action



Technique 2: Paired Data Structure

- **Use alternating sets of variables to avoid resetting variables after each barrier**

Dissemination Barrier Algorithm

```
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
```

Assessment: Dissemination Barrier

- $\Theta(\log p)$ **operations on critical path**
- $\Theta(p \log p)$ **total remote operations**
- $O(p \log p)$ **space**
- **Atomic primitives: load and store**

Tournament Barrier with Tree-based Wakeup

```
type round_t = record
role : (winner, loser, bye, champion, dropout)
opponent : ^Boolean
flag : Boolean  shared rounds :
    array [0..P-1][0..LogP] of round_t
   // row vpid of rounds is allocated in shared memory
   // locally accessible to processor vpid  processor

private sense : Boolean := true
processor private vpid : integer  // a unique index
```

Tournament Barrier Idea

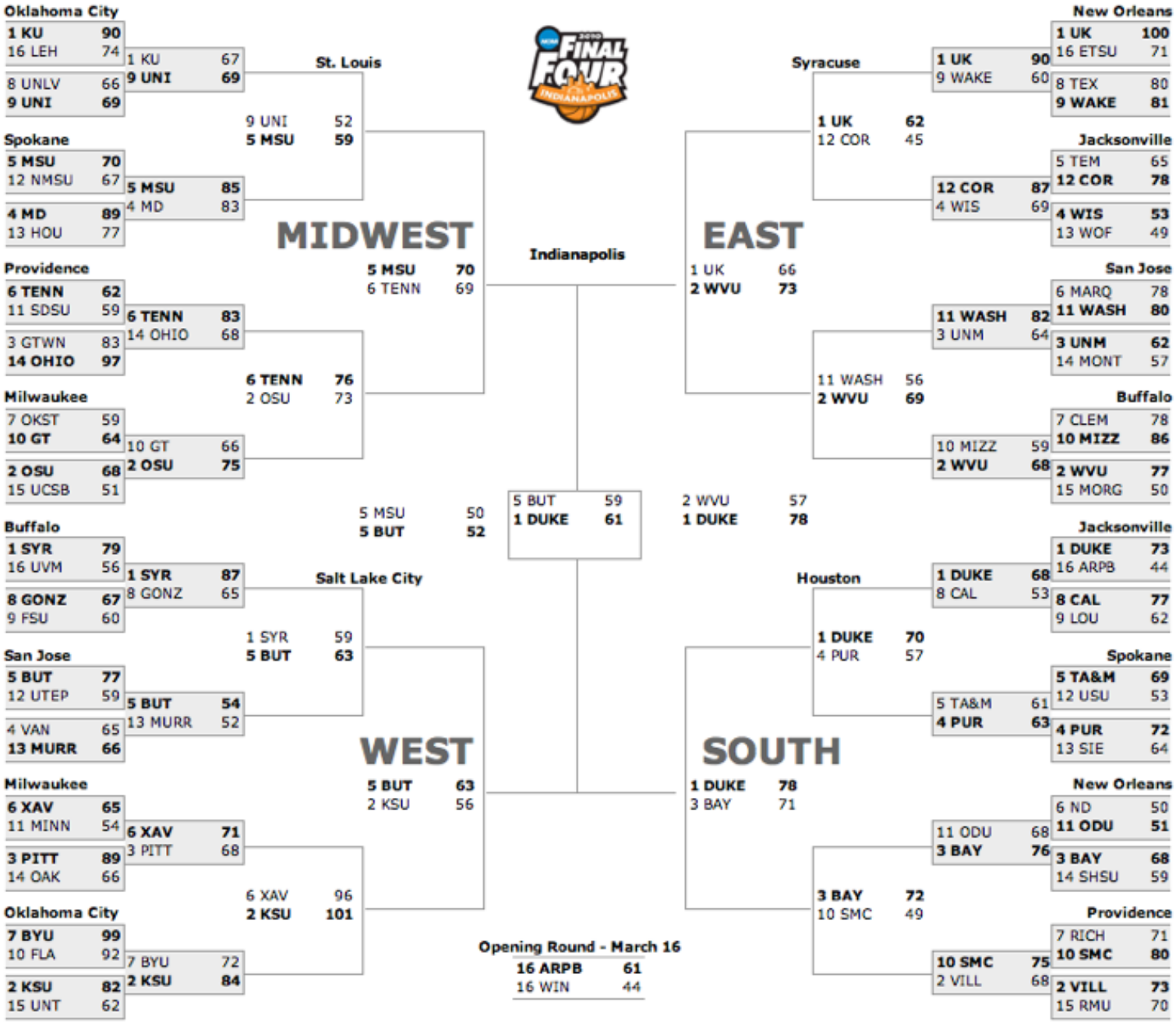


Figure credit: http://espn.go.com/mens-college-basketball/tournament/bracket

Tournament Barrier

```
procedure tournament_barrier
  round : integer := 1
  loop                          // arrival
    case rounds[vpid][round].role of
      loser:
        rounds[vpid][round].opponent^ := sense
        repeat until rounds[vpid][round].flag = sense
        exit loop
      winner:
        repeat until rounds[vpid][round].flag = sense
      bye:          // do nothing
      champion:
        repeat until rounds[vpid][round].flag = sense
        rounds[vpid][round].opponent^ := sense
        exit loop
    round := round + 1
```

Tournament Barrier Wakeup

```
loop                    // wakeup
  round := round - 1
  case rounds[vpid][round].role of
    loser:              // impossible
    winner:
      rounds[vpid][round].opponent^ := sense
    bye:                // do nothing
    champion:           // impossible
    dropout:
      exit loop
sense := not sense
```

Assessment: Tournament Barrier

- $\Theta(\log p)$ **operations on critical path**
  - **—larger constant than dissemination barrier**
- $\Theta(p)$ **total remote operations**
- $O(p \log p)$ **space**
- **Atomic primitives: load and store**

Scalable Tree Barrier

```
procedure tree_barrier
  with nodes[vpid] do
    repeat until childnotready =
       {false, false, false, false}
    childnotready := havechild      // prepare for next barrier
    parentpointer^ := false         // let parent know I'm ready

    // if not root, 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
```

Assessment: Scalable Tree Barrier

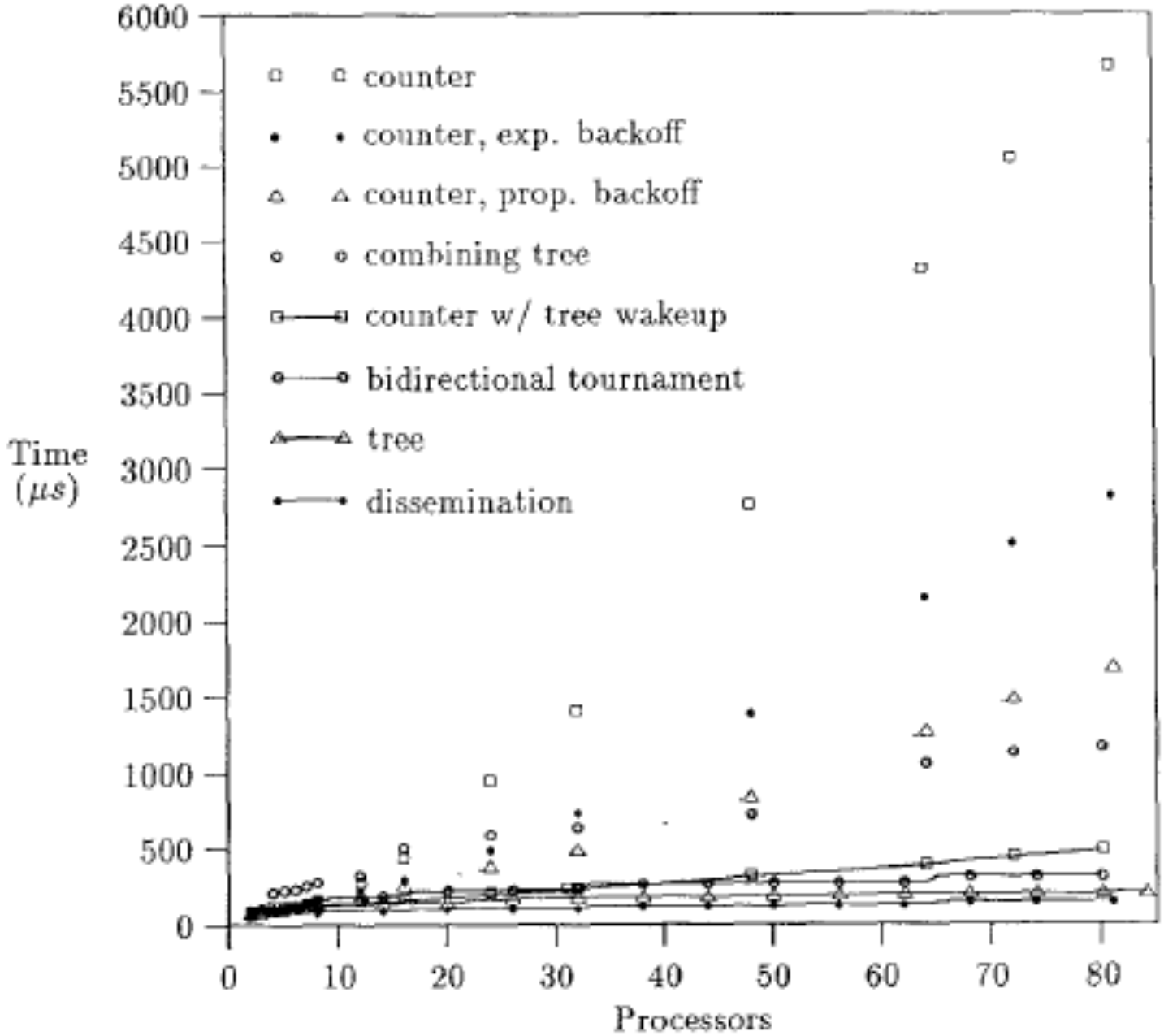
- $\Theta(\log p)$ **operations on critical path**
- *2p-2* **total remote operations**
  - **—minimum possible without broadcast**
- $O(p)$ **space**
- **Atomic primitives: load and store**

Case Study:

Evaluating Barrier Implementations for the BBN Butterfly and Sequent Symmetry

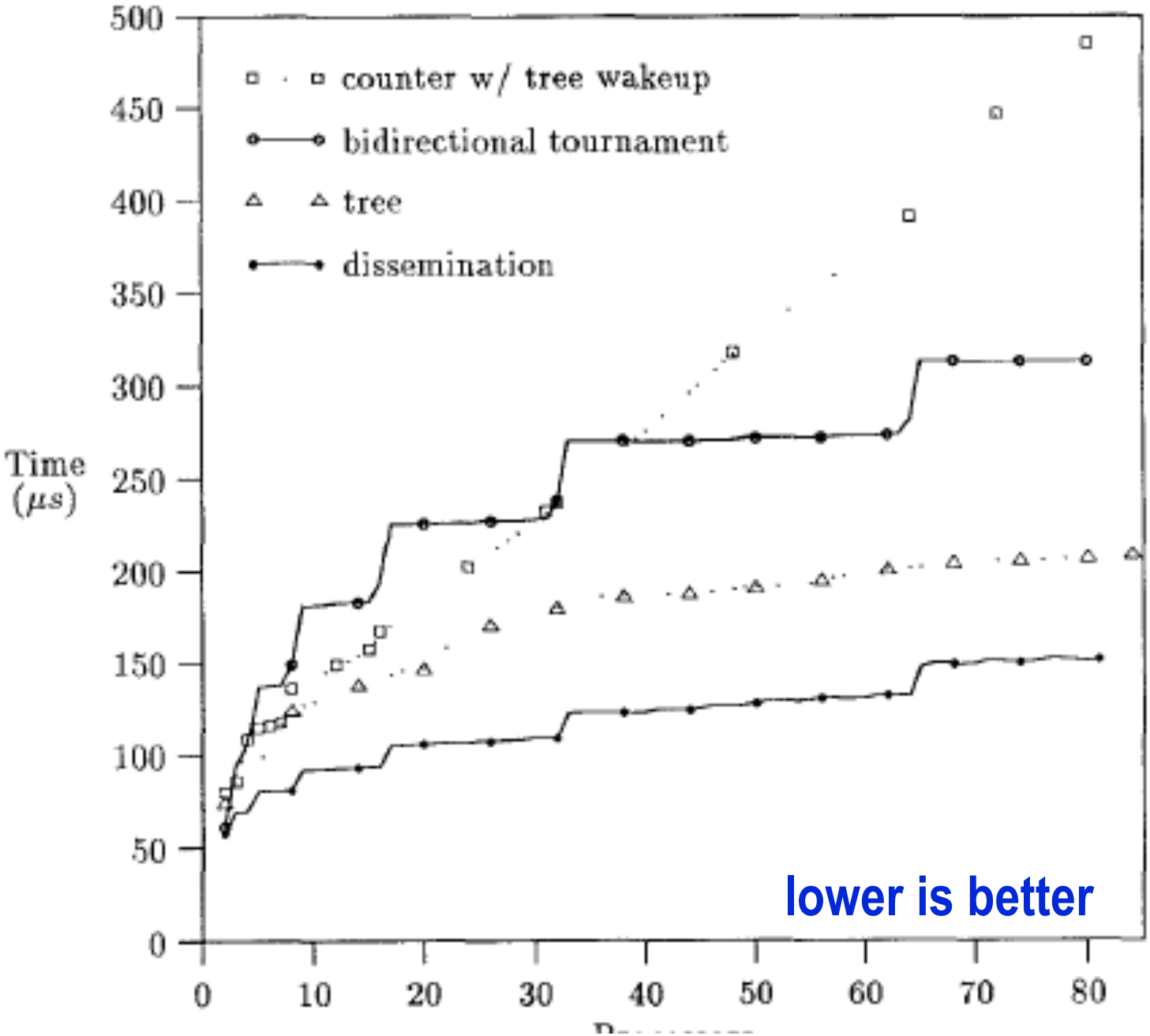
J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Transactions on Computer Systems, 9(1):21-65, Feb. 1991.

Butterfly: All Barriers

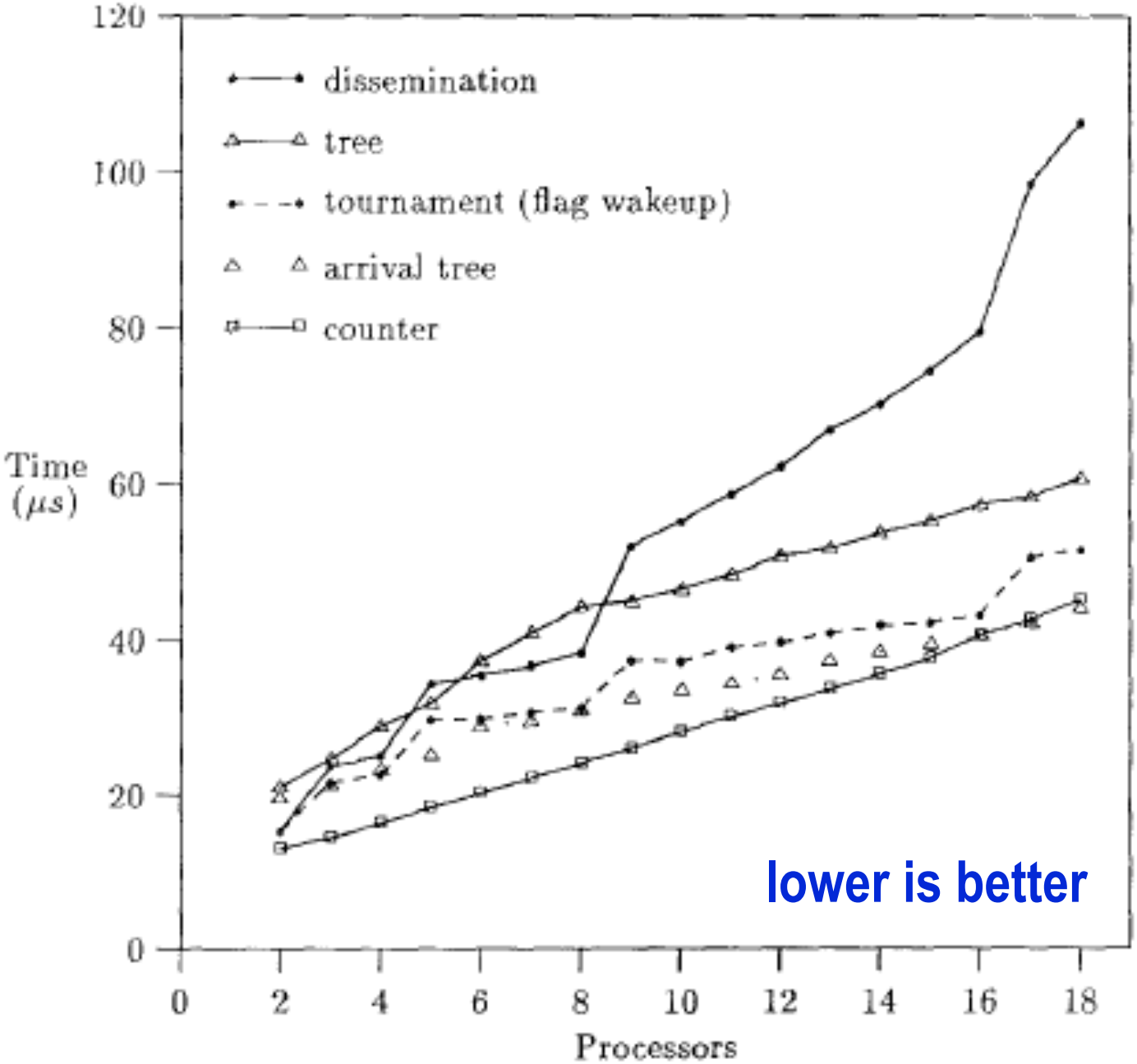


lower is better

Butterfly: Selected Barriers



Sequent: Selected Barriers



Take Away Points

- The atomic primitives available make a difference
- Scalable locks are important for stable performance
- Hardware requirements for constructing scalable synchronization algorithms for locks and barriers
  - —HW support for local spinning or blocking
  - —sufficient atomic operations
    - swap, compare-and-swap
- Key characteristic of scalable synchronization algorithms
  - —bounded number of remote operations
    - local spinning or blocking

Implications for Hardware Design

- **Special-purpose synchronization hardware can offer only**
  - **—at best a logarithmic improvement for barriers**
  - **—constant factor improvement for locks**
- **Locality matters: local-spinning algorithms provide a case against dance-hall architectures**
  - **—dance-hall = uncached shared-memory equally far from all processors**

Trends

- **Hierarchical systems**
- **Hardware support for barriers**

Hierarchical Systems

- **Layers of hierarchy**
  - —multicore processors
  - —SMP nodes in NUMA distributed shared-memory multiprocessor
    - e.g. SGI Ultraviolet: dual-socket Nehalem EX nodes
- **Require hierarchical algorithms for top efficiency**
  - —use hybrid algorithm
    - one strategy within an SMP
      - a simple strategy might work fine
    - a second scalable strategy across SMP nodes

Hardware Support for Barriers

Wired OR in IBM Cyclops 64-core chip

- **Special-purpose register(SPR) implements wired-or**
  - —8-bit register; 2 bits per barrier (4 independent barriers)
  - —reads of SPR reads the ORed value of all thread's SPRs
- **Each thread writes its own SPR independently**
  - —threads not participating leave both bits 0
  - —threads initialize bit for current barrier cycle to 1
  - —when a thread arrives at a barrier
    - – atomically: current barrier bit ← 0; next barrier bit ← 1
- **Busy-wait for a barrier to complete**

  while (current barrier bit in SPR != 0) {}

Hardware Support for Barriers II

IBM Blue Gene/L

- Hardware support for barriers to reduce latency
- Hardware barrier network
  - —four independent channels
  - — effectively a global OR over all nodes
    - – individual signals are combined in hardware
    - – propagate to the physical top of a combining tree
    - – resultant signal is broadcast back down the tree
  - —global AND can be achieved by inverting logic
- Global AND used as barrier
- Global OR used as global interrupt
- Barrier network is partitionable on same midplane boundaries as other Blue Gene networks

A. Gara et al. Overview of the IBM Blue Gene/L System Architecture.
IBM J. RES. & DEV. VOL. 49 NO. 2/3 MARCH/MAY 2005

References

- J. Mellor-Crummey, M. L. Scott: Synchronization without Contention. ASPLOS, 269-278, 1991.
- J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Transactions on Computer Systems, 9(1): 21-65, Feb. 1991.
- T. E. Anderson, The performance of spin lock alternatives for shared-memory multiprocessors. IEEE Transactions on Parallel and Distributed Systems, 1(1): 6-16, Jan. 1990.
- Travis Craig, Building FIFO and priority queuing spin locks from atomic swap. University of Washington, Dept. of Computer Science, TR 93-02-02, Feb. 1993.
- Anders Landin and Eric Hagersten. Queue locks on cache coherent multiprocessors. International Parallel Processing Symposium, pages 26-29, 1994.
- Juan del Cuvillo, Weirong Zhu, and Guang R. Gao. Landing OpenMP on Cyclops-64: an efficient mapping of OpenMP to a many-core system-on-a-chip, ACM International Conference on Computing Frontiers, May 2-5, 2006, Ischia, Italy.