Scalable Synchronization on Shared Memory Multiprocessors

- Paper discusses several algorithms for implementing spinlocks and barriers; includes performance discussion on two types of hardware

Intro

- Why need synchronization contructs in the kernel?
- Can a mutex be sufficient?

The lock in GTTreads

```
int actual_lock(int* mutex) {

        int mutex_val = 1;

    while(mutex_val)
    {
        asm("movl $0x1, %%eax;"
            "movl %1, %%ebx;"
            "xchg %%eax, (%%ebx);"
            "movl %%eax, %0;"
            :"=m"(mutex_val)
            :"m"(mutex)
            :"%eax", "%ebx");
    }
        return 0;
}
```

It’s spinning!
Can do it differently so it blocks…

Basic sync constructs

- Blocking – may be too heavy
- Busy-waiting
  - Benefits when?
- Built on top of some atomic instructions supported by hardware
  - Different on different platforms
    - test-and-set(<mem>)
    - fetch-and-store(R, <mem>)
    - fetch-and-add(<mem>, <value>)
    - compare-and-swap(<mem>, <cmp-val>, <stor-val>)
  - Execution costly – typically atomic instruction costs more cycles than a regular one

Basic spin-lock

- Using atomic test_and_set operation

```
init      lock = clear;
lock      while(t&s(lock) == busy); /* spin */
unlock    lock = clear;
```

- Where is lock?
- Do we have to spin on the shared variable (and the atomic instruction)?
- What is the architecture like?
  - Bus-based vs. Interconnection network
  - Local caches? Cache-coherency?
    - Write-update (Broadcast)
    - Write-invalidate

Algorithm Objectives

- reduce latency
  - how quickly can I get the lock in the absence of competition
- reduce waiting time – delaying the application
  - time between lock is freed and another processor acquires it
- reduce contention / bandwidth consumption
  - other processors need to do useful work, process holding lock needs to complete asap…
  - how to design the waiting scheme to reduce interconnection network contention

Any conflicts here??

Reduce contention approach – Backoff delay

```
type lock = (unlocked, locked)
procedure acquire_lock(L:^lock)
   delay : integer :=1
   while test_and_set(L) = locked
        pause(delay)
        delay:=delay*2
procedure release_lock(L:^lock)
   lock^:=unlocked
```

- Different processors back-off different amount of time when they see lock was busy
- Exponential backoff, static backoff…
- Delay approaches work for bus-based and interconnection network based architectures (“dance-hall”)
- Problem -?

Reduce waiting time/delay

- Can you find out immediately that lock is freed without going to memory location?
- Caches! & cache-coherent architectures
- Idea – spin on locally cached value, act when you see a change
- Test_and_test&set

```
 lock     while (lock == busy) spin;
          if (t&s(lock) == busy) goto lock;
```

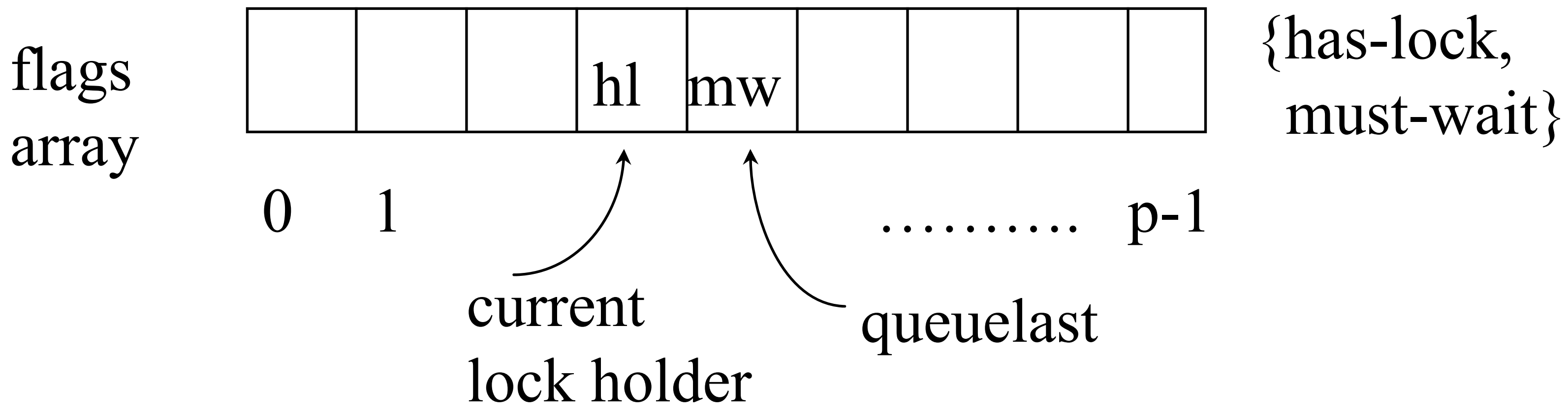
- What's behaviour like in Write-Update vs. Write-Invalidate?
- What if you don't have hardware supported coherency? Can all local "lock" copies be changed?

Ticket lock

```
type lock = record
    next_ticket : unsigned integer := 0
    now_serving : unsigned integer := 0
                                                         What if not supported?
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)  <-  Backoff time good/bad?
            // consume this many units of time
            // on most machines, subtraction works correctly despite over
        if L->now_serving = my_ticket  <-  private
            return
                                           shared

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

Can we spin on local and private?

Queuing locks



- a unique ticket for each arriving processor
- get ticket in waiting queue
- go into CS when you have lock
- signal next waiter when you are done
- assume a primitive op r&inc(mem)

Anderson's array-based queuing 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)
        // returns old value
    if my_place^ mod numprocs = 0
        atomic_add (&L->next_slot, -numprocs)
        // avoid problems with overflow; return value ignored
    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
```

again!
approach: replace with
Regular test_and_set

Noone is spinning on next_slot

make sure it's
in different cache
line/Mm module

Graunke-Thakkar's array-based lock

```
type lock = record
    slots : array [0..numprocs -1] of Boolean := true
        // each element of slots should lie in a different memory module
        // or cache line
    tail : record
        who_was_last : ^Boolean := 0
        this_means_locked : Boolean := false
        // this_means_locked is a one-bit quantity.
        // who_was_last points to an element of slots.
        // if all elements lie at even addresses, this tail "record"
        // can be made to fit in one word
processor private vpid : integer // a unique virtual processor index

procedure acquire_lock (L : ^lock)                          avoids read&inc
    (who_is_ahead_of_me : ^Boolean, what_is_locked : Boolean)
        := fetch_and_store (&L->tail, (&slots[vpid], slots[vpid]))
    repeat while who_is_ahead_of_me^ = what_is_locked

procedure release_lock (L : ^lock)
    L->slots[vpid] := not L->slots[vpid]
```

Issues with these locks

- Proc: FIFO ordering
  - Delay, resource contention – good
    - (not in ticket without CCwBroadcast)
  - Though may be dangerous if process in the queue gets context switched!
  - Ticket lock also guarantees FIFO.
- Cons:
  - lock structure size proportional to # of procs.
  - Latency issue, especially if no r&inc in Anderson, ticket

MCS queuing lock

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

// 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
            // compare_and_swap returns true iff it swapped
        repeat while I->next = nil         // spin
    I->next->locked := false
```

No known successor?!

- Time difference between fetch_and_store in acquire_lock & predecessor->next = I
- If we think noone’s behind us, we must atomically (1) verify this, and (2) set lock to NIL
  - Compare_and_swap solves this

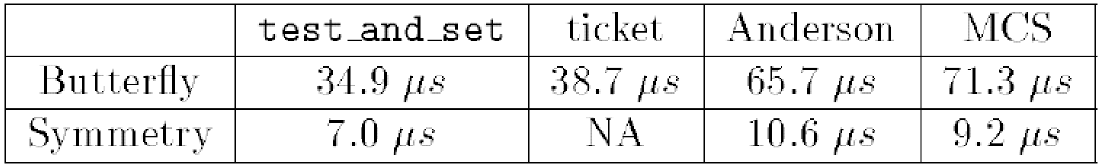
No compare_and_swap

```
procedure release_lock (L : ^lock, I : ^qnode)
    if I->next = nil        // no known successor
        old_tail : ^qnode := fetch_and_store (L, nil)
        if old_tail = I     // I really had no successor
            return
        // we have accidentally removed some processor(s) from the queue;
        // we need to put them back
        usurper := fetch_and_store (L, old_tail)
        repeat while I->next = nil          // wait for pointer to victim list
        if usurper != nil
            // somebody got into the queue ahead of our victims
            usurper->next := I->next        // link victims after the last usurper
        else
            I->next->locked := false
    else
        I->next->locked := false
```

- Doable – but now 2 atomic operations – they are expensive!

Performance comparisons

- BBN Butterfly
  - Shared mm multiprocessor, up to 256 processor nodes, $\log_4$-depth interconnection network
  - Atomic ops fetch_and_clear_then_add and fetch_and_clear_then_xor; mask used to implement other ops, including fetch_and_add;
  - Atomic ops very expensive compared to regular operations
- Sequent Symmetry
  - Shared bus multiprocessor, up to 30 nodes
  - Write-invalidate cache-coherency
  - Fetch_and_store supported; other logical/arithm ops can be applied atomically but do not return a value (e.g., incr vs. read_and_incr)
  - Atomics cheaper
- Neither supports compare_and_swap
- Single processor #s:



	test_and_set	ticket	Anderson	MCS
Butterfly	34.9 μs	38.7 μs	65.7 μs	71.3 μs
Symmetry	7.0 μs	NA	10.6 μs	9.2 μs

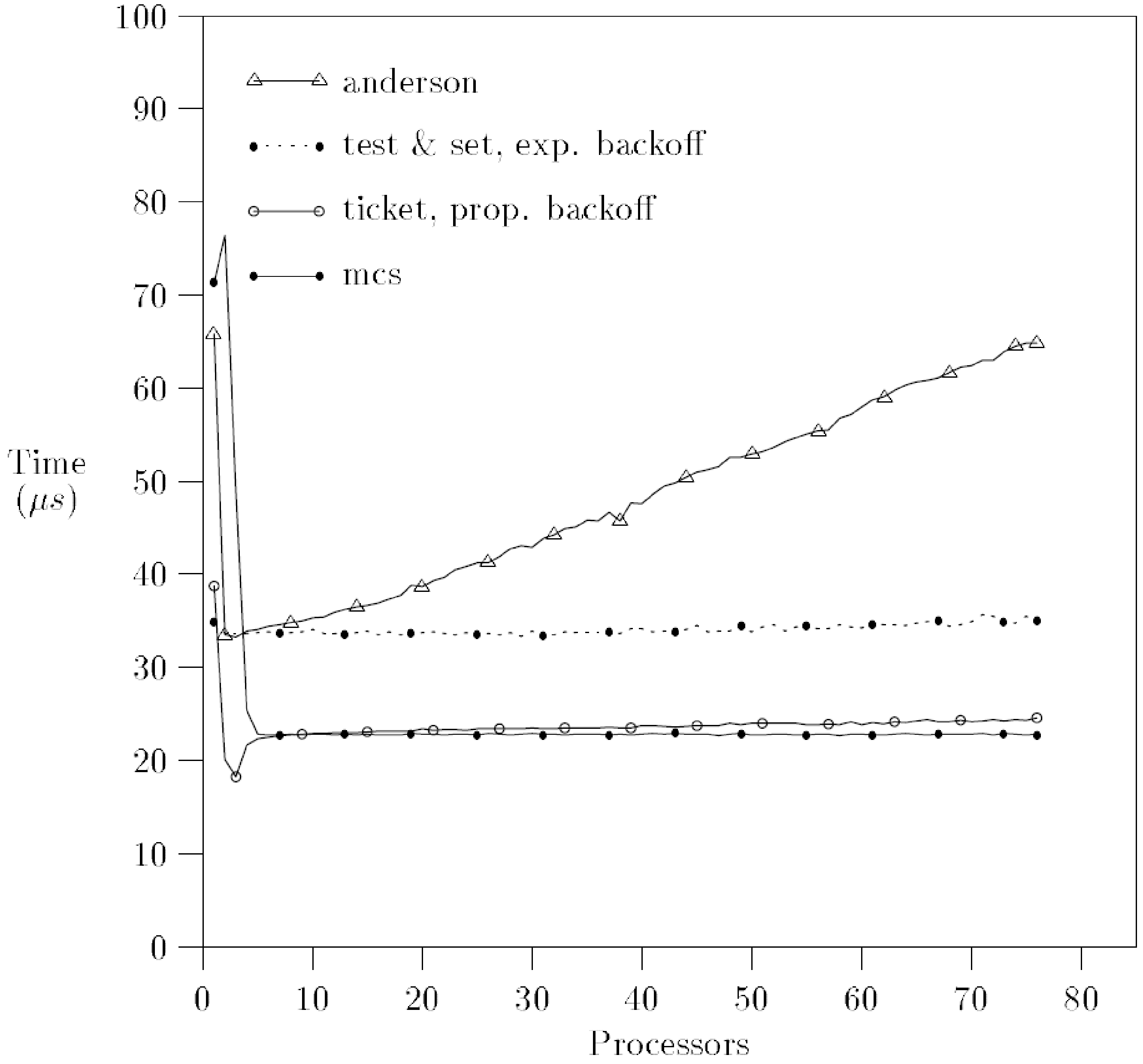


Figure 5: Performance of selected spin locks on the Butterfly (empty critical section)

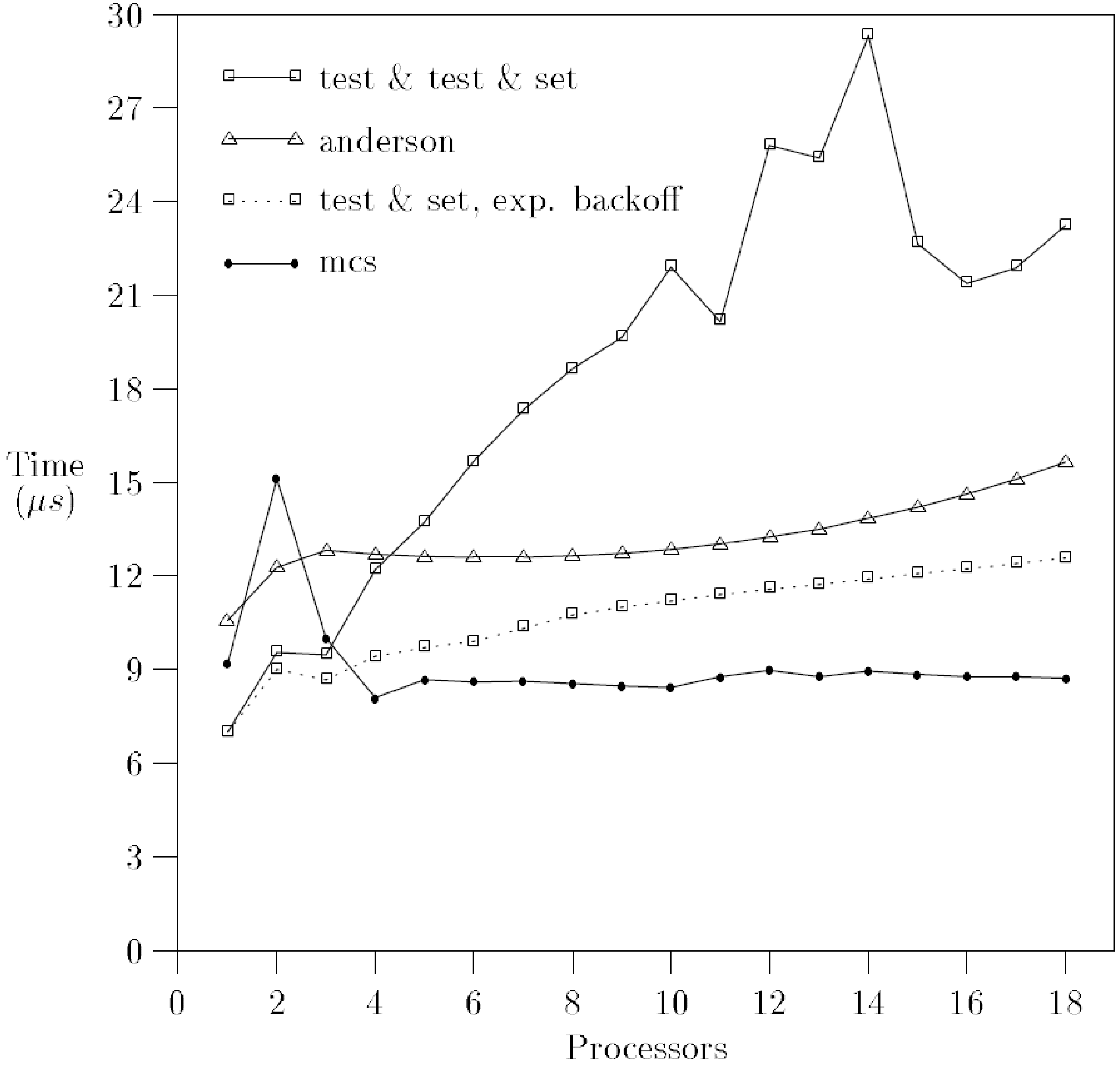


Figure 6: Performance of spin locks on the Symmetry (empty critical section)

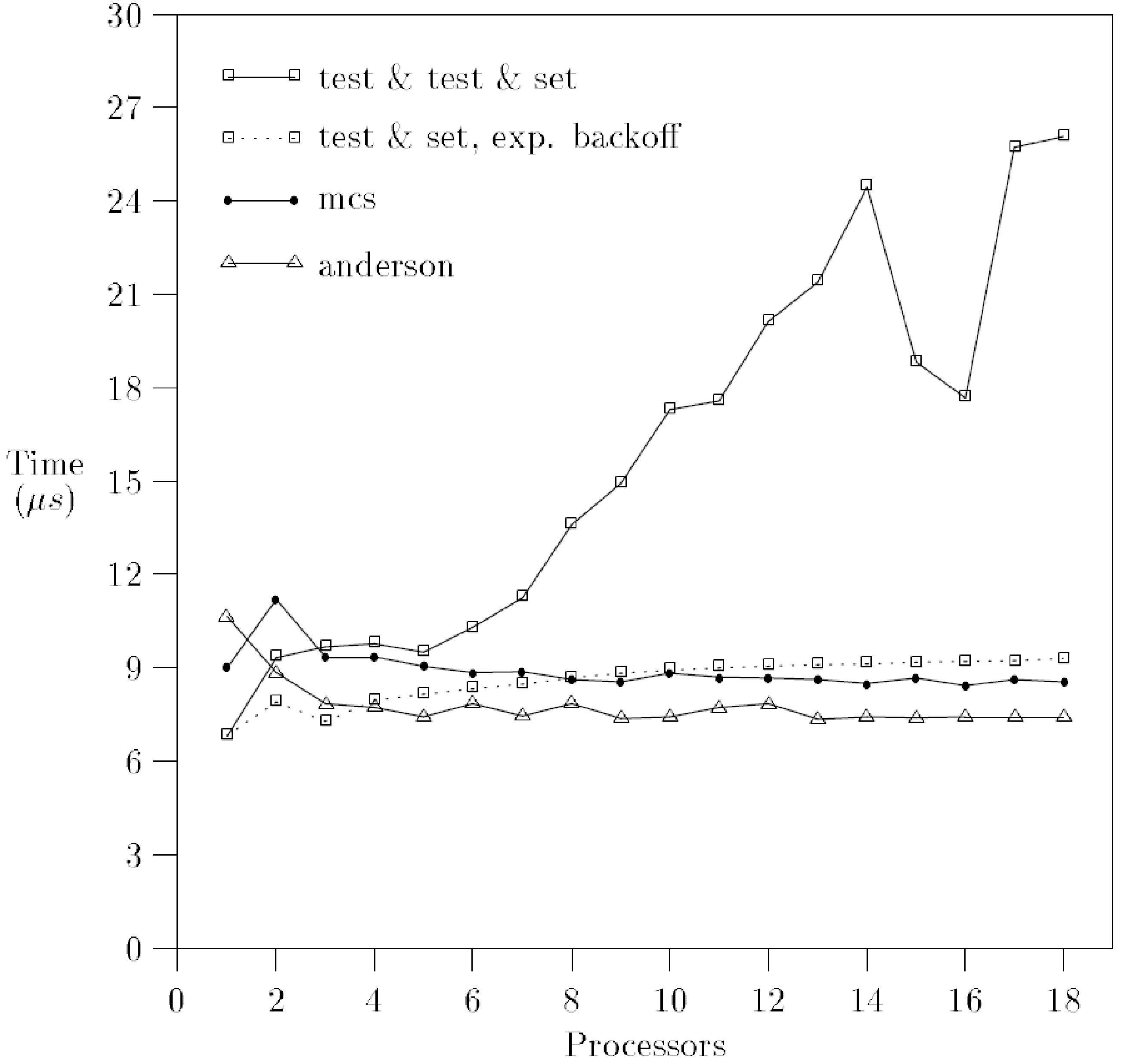
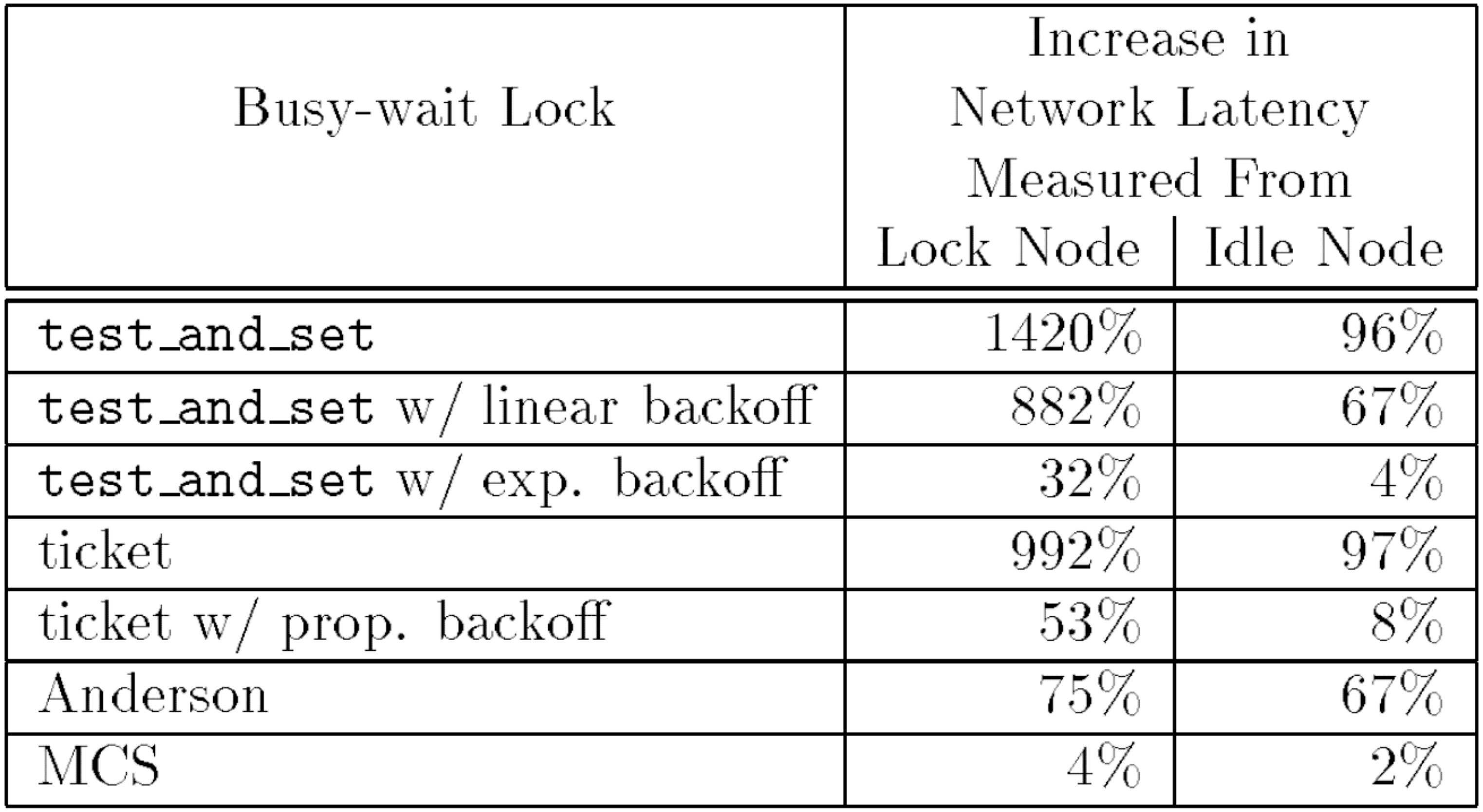


Figure 7: Performance of spin locks on the Symmetry (small critical section).

Network contention on interconnect-based multiprocessors

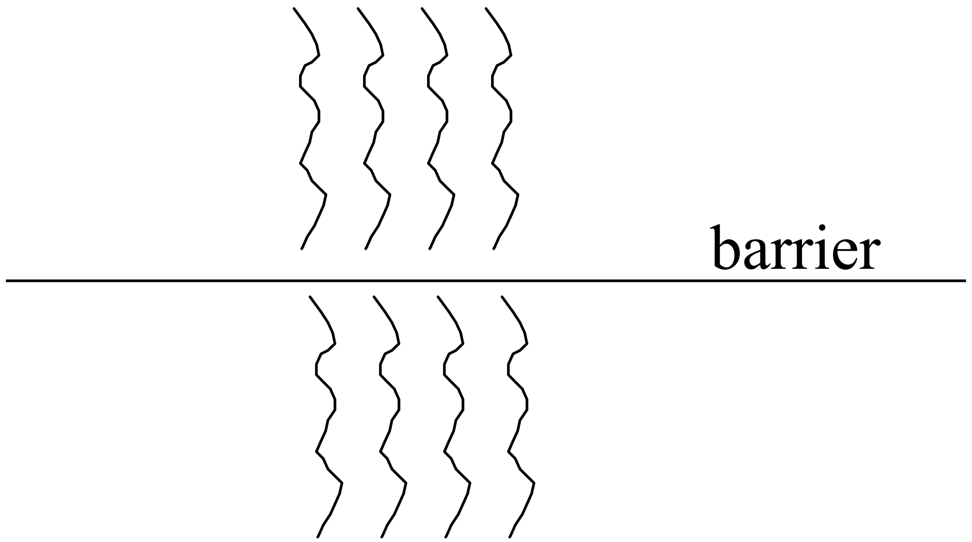


Busy-wait Lock	Increase in Network Latency Measured From	
	Lock Node	Idle Node
test_and_set	1420%	96%
test_and_set w/ linear backoff	882%	67%
test_and_set w/ exp. backoff	32%	4%
ticket	992%	97%
ticket w/ prop. backoff	53%	8%
Anderson	75%	67%
MCS	4%	2%

- BBN with 60 processors

Barriers

- User in parallel apps, to make sure no process advances beyond a point until all processes in the group reach the barrier



Basic sense-reversing centralized barrier

```
shared count : integer := P
shared sense : Boolean := true
processor private local_sense : Boolean := true

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

- What’s the purpose of the sense variable?
- Can we implement barriers without sense reversing?
- Spins on shared sense variable
  - when is this ok?
    - Hint: only one processor is updating
- Does inserting delays, like with spinlocks, help?

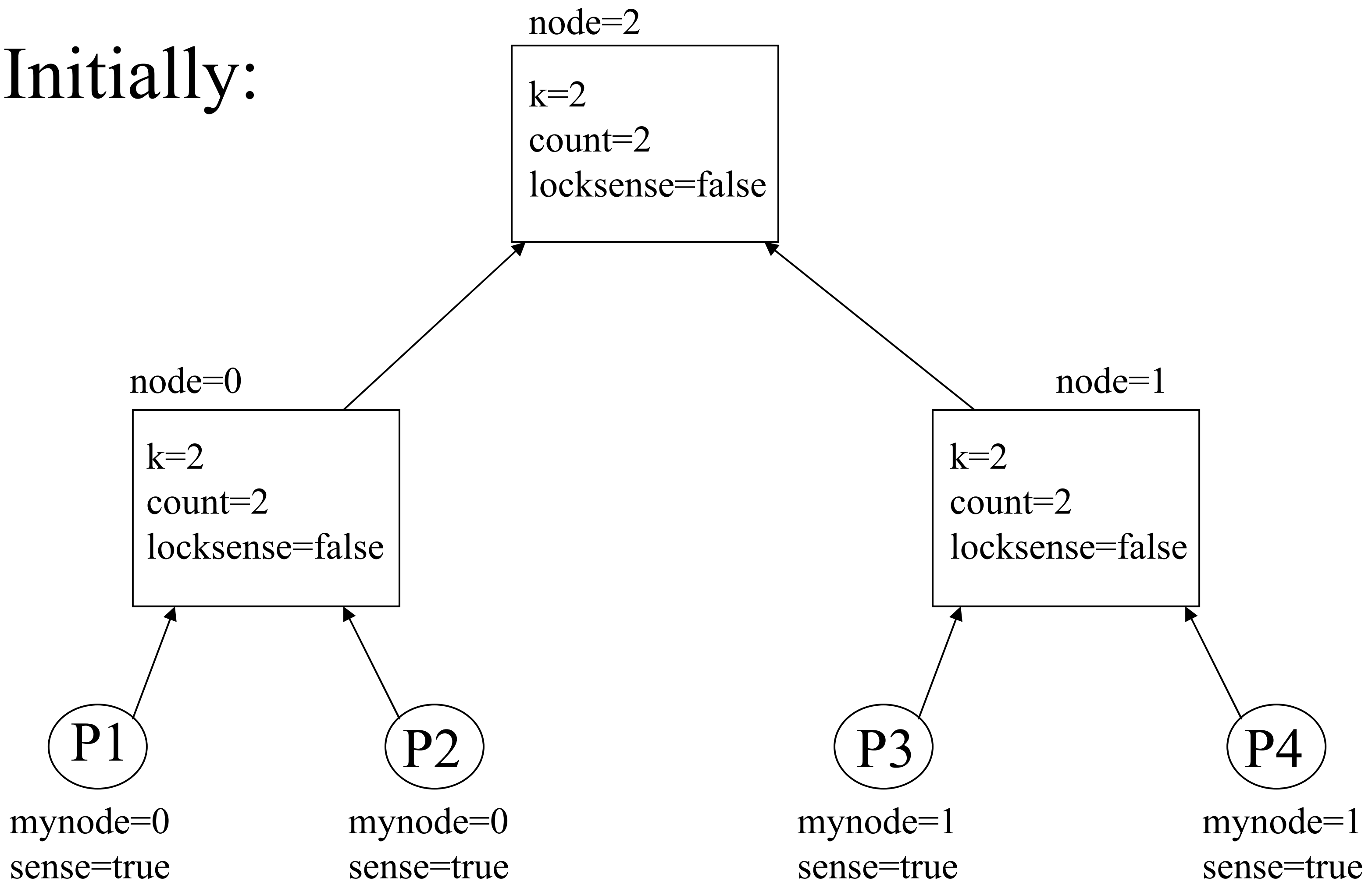
Software Combining Tree Barrier

```
type node = record
    k : integer              // fan-in of this node
    count : integer          // initialized to k
    locksense : 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 combining tree

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 one to reach this node
            if parent != nil
                combining_barrier_aux (parent)
            count := k                          // prepare for next barrier
            locksense := not locksense          // release waiting processors
        repeat until locksense = sense
```

(illustration – a tree wouldn't necessarily look like this binary tree)

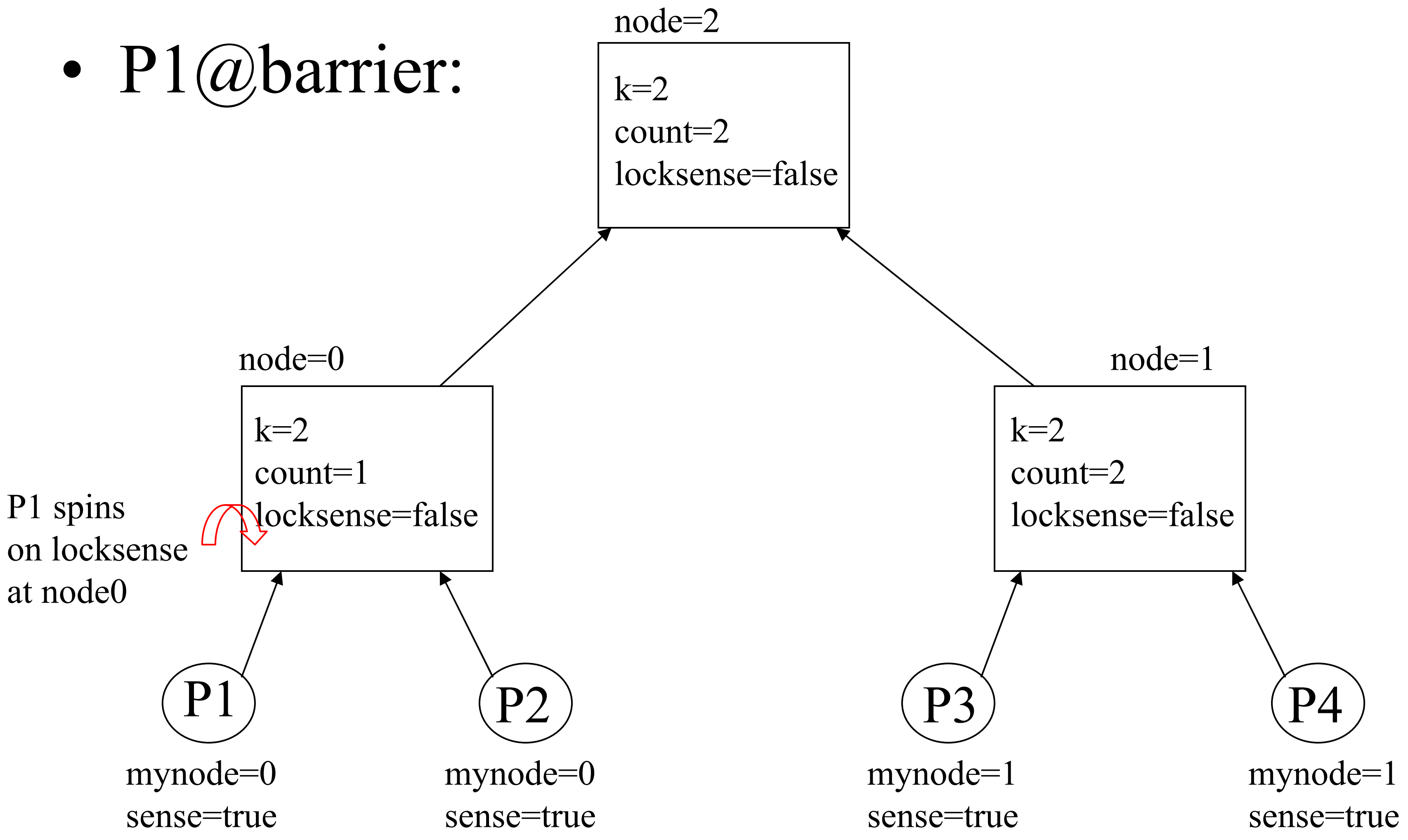
Software Combining Tree Barrier

- Initially:



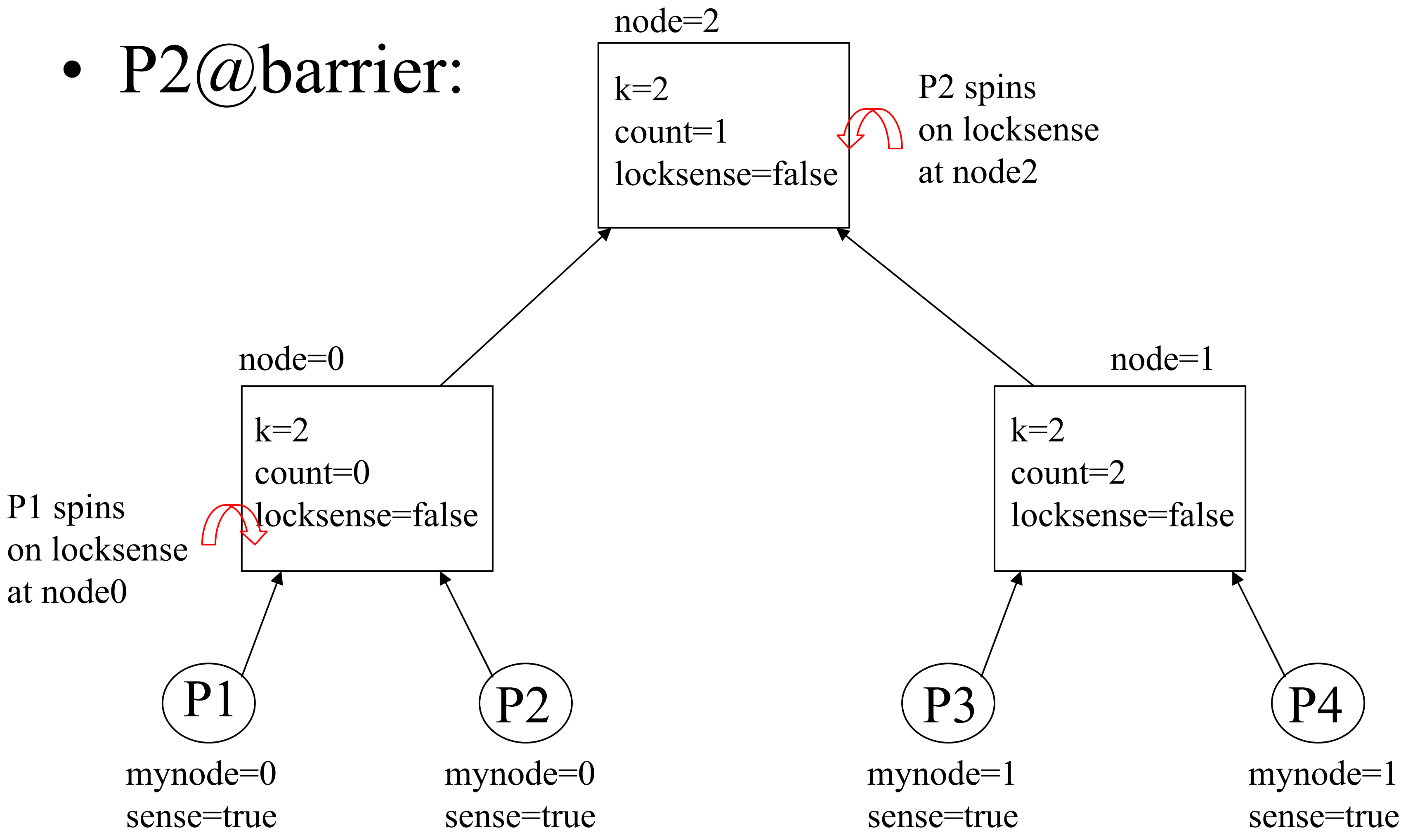
Software Combining Tree Barrier

- P1@barrier:



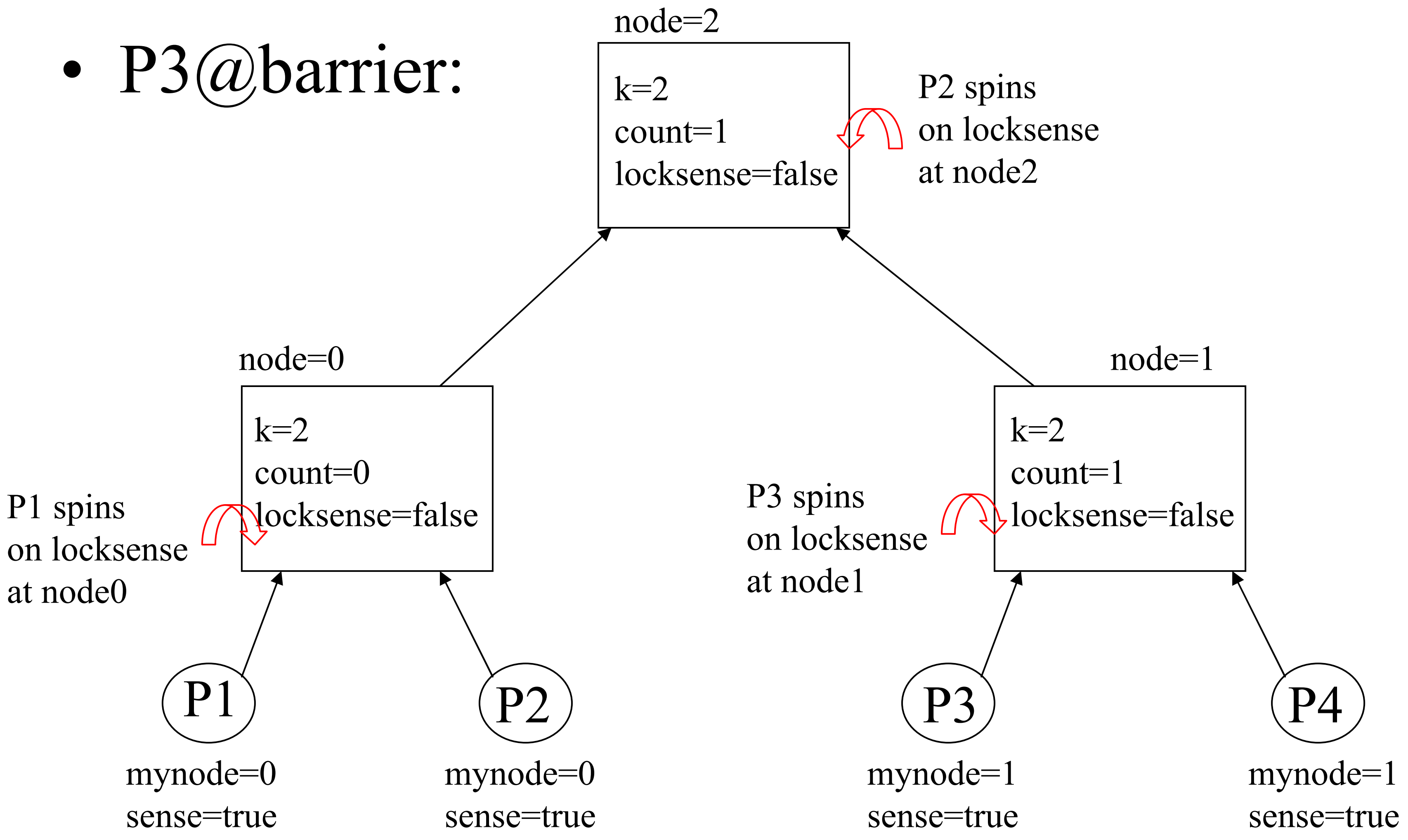
Software Combining Tree Barrier

- P2@barrier:



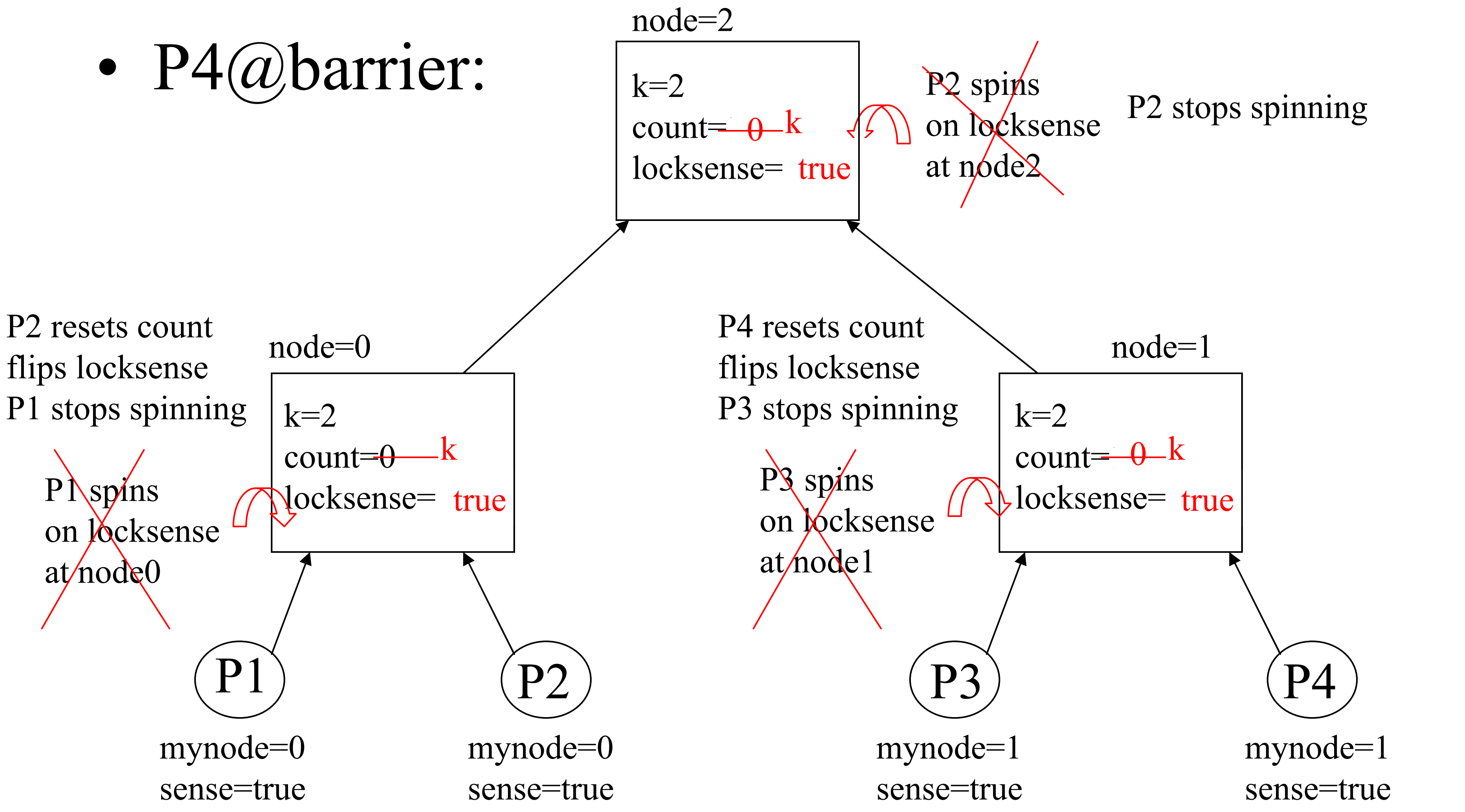
Software Combining Tree Barrier

- P3@barrier:



Software Combining Tree Barrier

- P4@barrier:



SCT barrier

- Good for other collectives too
- On CC arch., may spin locally, otherwise on remote locations
- Spin location not statically known – depends on arrival of processes – ultimately unbound # network transactions on nonCCwBroadcast
- Length of critical path
- Space

Other barrier versions in paper

- Dissemination:
  - Each proc notifies logP procs it reached a barrier, and also waits on logP procs
  - Complex/sizeable data structure O(PlogP), simple signaling operations but O(PlogP)
  - Local spinning
- Tournament:
  - Like combining tree, but predetermined which processor goes up to the next level – statically known spinning locations
  - O(PlogP) space, O(P) network transactions
  - May spin on global flag for CCwBroadcast architectures
- Proposed barrier:
  - Again tree based, smaller data structure O(P), and minimizes number of network transactions – still O(P);
  - Spin optimization as above
- Each processor updates a dedicated (set of) locations – no need for atomic ops!

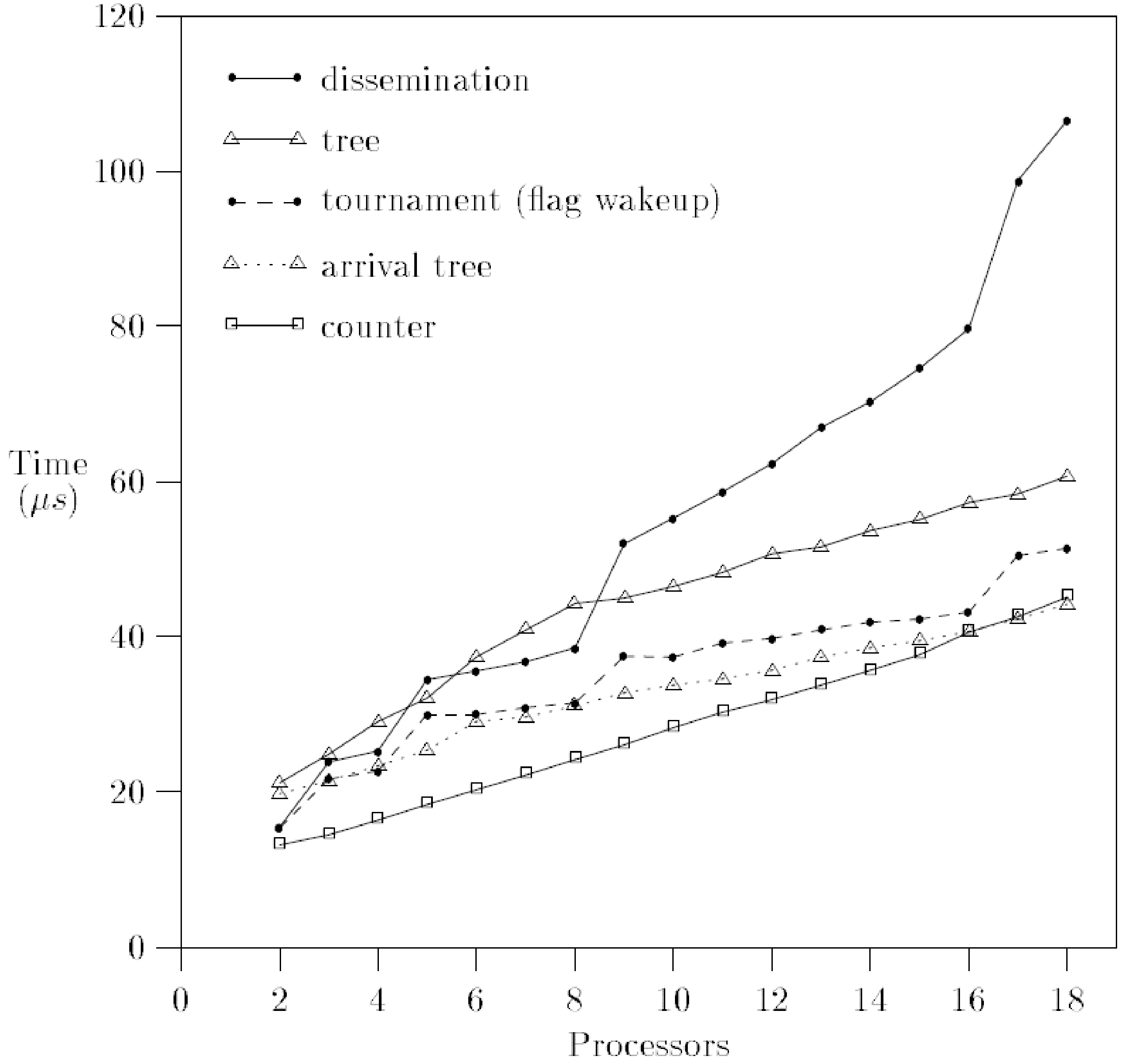


Figure 10: Performance of barriers on the Symmetry.

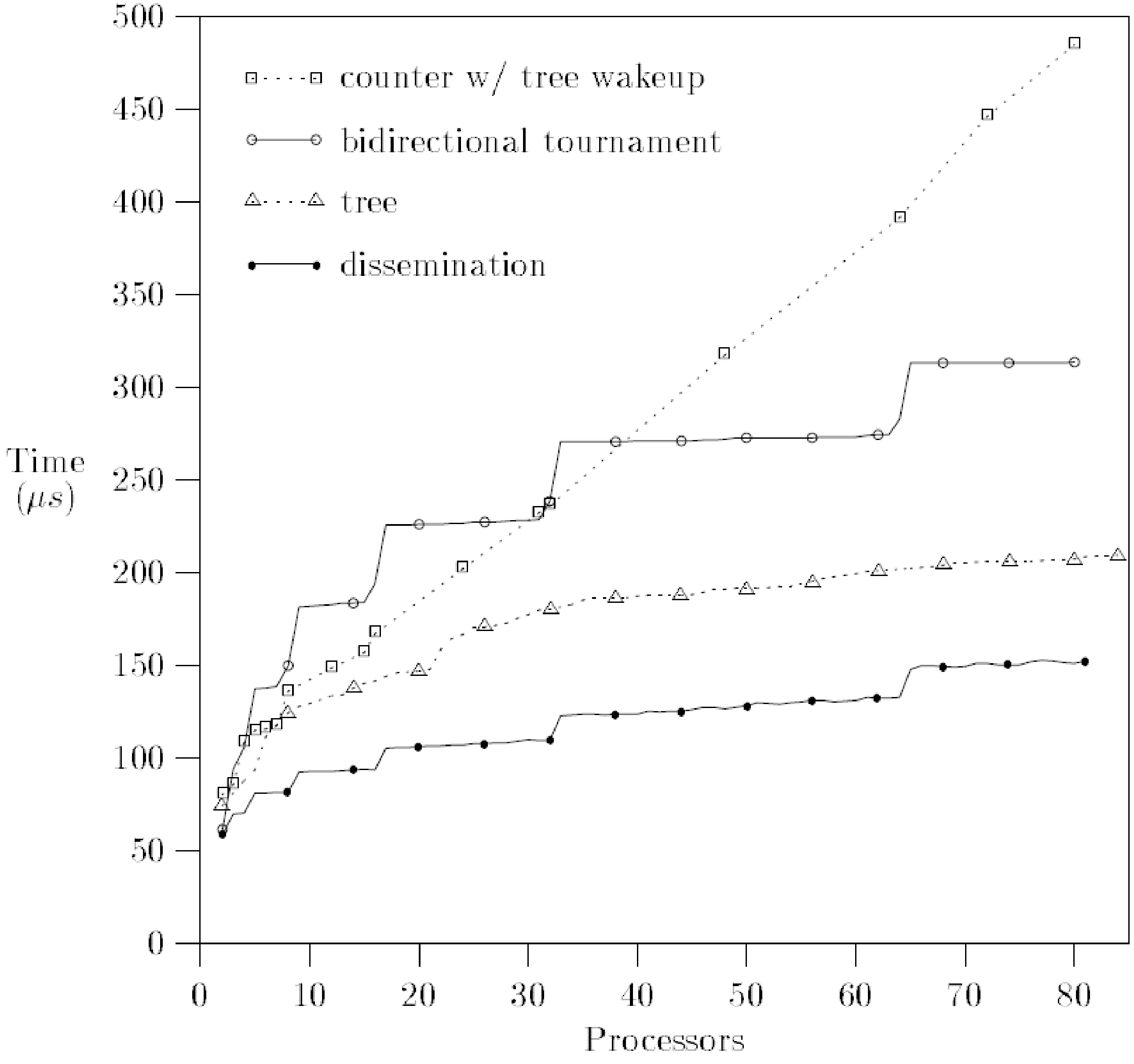


Figure 9: Performance of selected barriers on the Butterfly.