Spinlocks & Barriers

Henry Moncure IV and Pavlo Pastaryev

1

Evaluated hardware

- Sequent Symmetry
  - Shared bus
  - Coherent caches
- BBN Butterfly
  - Each processor has local memory
  - Can access memory of others via a switching network

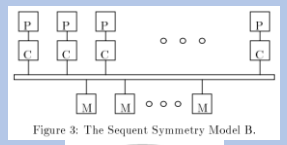


Figure 3: The Sequent Symmetry Model B.

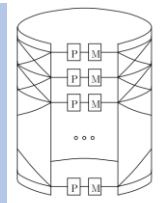


Figure 2: The BBN Butterfly 1.

2

Motivation for Locking

- Synchronize access to shared data
- Enforce:
  - Mutual exclusion
  - Forward progress
  - Bounded waiting

3

Performance Goals

- Low latency, short critical path
- Low traffic (on interconnect network)
- Scalability
- Low storage cost
- Fairness

4

What Are Our Choices?

- Scheduler-Based (Blocking)
  - Tell the OS we are not ready to run right now
  - Give up CPU
  - OS can wake us up later when conditions are met
- Busy-Waiting
  - "Are we there yet? Are we there yet? Are we there yet?…"

5

Why Bother With Busy-Wait?

- Blocking seems great - let the OS take care of our problems
- Blocking does have some downsides though
  - OS kernel can't make use of it
  - Scheduling overhead
  - Processor may not actually be needed by other tasks
- Busy-waiting is appropriate when:
  - Scheduling overhead time is greater than the expected wait time to acquire the lock
  - The processor is not needed for other tasks
  - Blocking inappropriate or impossible (e.g. OS kernel)
- Spin locks usually protect a small critical section and may be executed many, many times
  - Lock performance is important

6

Are we there yet?" - Downsides of Busy-Waiting

- Major downside: memory and interconnect bus contention
- "Hot Spot" - many processors all busy-waiting on one synchronization variable
  - Can degrade performance for all interconnect traffic, not just the traffic related to synchronization

7

Potential Solutions

- Hardware based
  - Specialized interconnect designs - combine concurrent accesses, synchronization in the interconnect
  - Special cache hardware to maintain a queue of processors waiting for the same lock
  - Hardware is a) expensive and b) less flexible
- Software based
  - Focus of this paper and presentation

8

Software Fix

- We want each processor to only spin on memory local to that processor
  - And that is not the spin target of any other processor
  - This way, we will greatly reduce traffic across the interconnect and to memory
- How to communicate between processors?
  - To be addressed later, varies between lock designs and hardware architectures

9

Test-and-Set Lock

- Processors repeatedly executing test-and-set atomic instructions to try to acquire the lock
  - And as fast as possible
- Expensive: causes many remote cache invalidations as well as interconnect contention
- Some optimizations do exist
  - Test-and-test-and-set: Only do the expensive test-and-set if a previous read indicates it's likely to succeed
  - Backoff strategies - constant delay or exponential

10

Ticket Lock

- Reduces the number of fetch-and-op operations to 1 per lock acquisition
- Ensures FIFO service
- Two counters: request counter and release counter
- Improvement: Only read operations in the spin, no expensive writes
  - Still fairly expensive on the interconnect
- Potential issues with backoff and overshooting

11

Ticket Lock – Pseudocode (From Paper)

```
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
```

12

Array-Based Queueing Lock

- Atomic fetch-and-increment/swap to obtain the address of an array index
- Spin on that array index
- Each processor spins on a location in a different cache line
- Major disadvantage: linear space requirements w.r.t. the number of processors, per lock

13

MCS Lock

- Named for authors' initials
- Guarantees:
  - FIFO ordering of acquisitions
  - Spins on locally accessible flags only
  - O(1) space per lock
  - O(1) interconnect transactions per lock acquisition
    - On machine with and without coherent caches
- Idea: each processor stores the address of the next processor in line to wake up
- Forms a queue
- The only operation that involves non-local memory is 1 write for the lock release

14

MCS Lock: Benefits

- Massively reduced interconnect traffic
  - On machines that are cache coherent and machines that are not!
- Highly scalable
  - Adds a completely minuscule amount of overhead per processor competing
- Constant space per lock
  - Each processor can only be waiting on one lock at a time, so even with multiple locks potentially able to be acquired, there is only one qnode structure and it's used in the queue for whichever lock the processor is currently waiting on

15

Motivation for Barriers

- Used to separate phases of computation
- Processes arriving at a barrier wait for all others, only then pass the barrier
- Less overhead than fork-join – no process destruction/creation

16

Centralized Barrier

- Shared counter, state variable (sense)
- Each arriving process decrements the counter, spins until sense has a different value than in the previous barrier
- Last arriving process resets counter and reverses sense

```
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
```

17

Centralized Barrier Analysis

- Works, but…
- All threads spin on the same location
  - If a system has coherent caches, all spin on local copies in the cache
  - If a system has no coherent caches, this will generate a lot of traffic
  - Could apply backoff to reduce the amount of traffic
- If multiple processes arrive at the same time, they all try to access one memory location

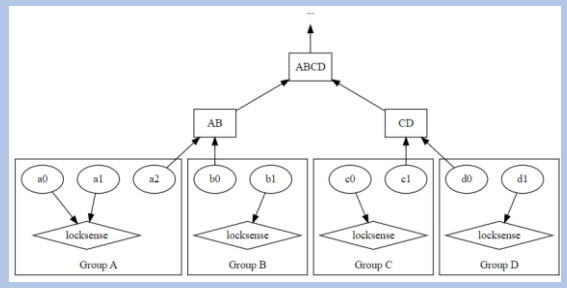
18

Software Combining Tree Barrier

- Split processes into groups
- Arrange counter, sense variables into a tree
- Process groups are the leaves
- Each arriving process decrements the counter, spins until sense has a different value than in the previous barrier (same as in Centralized Barrier)
- However, last processor now instead updates the sense and counter in the parent
- The processes pass the barrier when root is reached

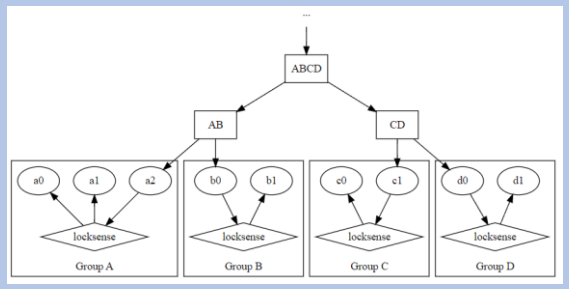
19

SCT Barrier – Arrival



20

SCT Barrier – Release



21

SCT Barrier – Code

```
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
```

22

SCT Barrier Analysis

- Similar problems to Centralized Barrier
- Each processor spins on shared variables
  - On cache-coherent systems can create local copies
  - On other systems, generates a lot of traffic
- Better when multiple processes arrive at the same time
  - They now are likely to access separate memory locations

23

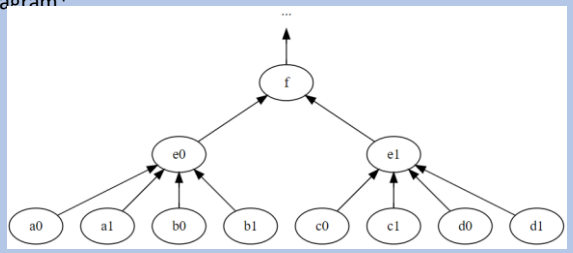
MCS Barrier

- Arrange processes into a tree
- Each arriving process waits for children to finish
- Then signals to the parent and spins on a local sense (if not root)
- When it is time to wake up, a process modifies local sense of children

24

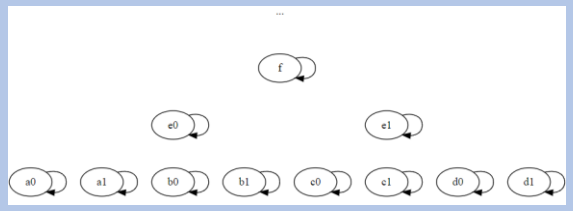
MCS Barrier – Arrival I

- *Diagram*



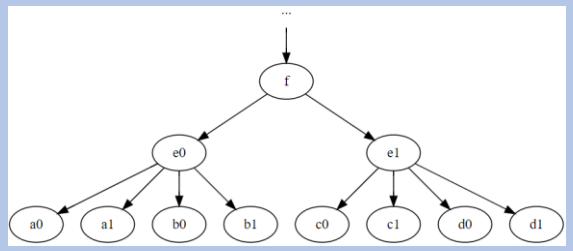
25

MCS Barrier – Arrival (Cont.)



26

MCS Barrier – Wakeup



27

MCS Barrier – Code

```
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

shared nodes : array [0..P-1] of treenode
    // nodes[vpid] is allocated in shared memory
    // locally accessible to processor vpid
processor private vpid : integer     // a unique virtual processor index
processor private sense : Boolean

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
```

28

MCS Barrier Analysis

- All processes spin on local flags
- Performs theoretically lowest amount of bus transactions
- Takes logarithmic time to wake up all processes

29

References

- Slides taken directly or modified slightly from last year's presentation, as well as class slides
- Diagrams and phrasing taken from paper

30