Barrier Synchronization

John Mellor-Crummey

Department of Computer Science
Rice University

johnmc@cs.rice.edu
Topics for Today

• Motivating barriers
• Barrier overview
• Performance issues
• Software barrier algorithms
  — centralized barrier with sense reversal
  — combining tree barrier
  — dissemination barrier
  — tournament barrier
  — scalable tree barrier
• Performance study
• Architectural trends
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
Exercise: Design a Simple Barrier
Shared-memory Barrier

- Each processor indicates its arrival at the barrier — *updates shared state*
- **Busy-waits** on shared state to determine when all have arrived
- Once all have arrived, each processor is allowed to continue
Problems with Naïve Busy Waiting

- May produce large amounts of
  - network contention
  - memory contention
  - cache thrashing

- Bottlenecks become more pronounced as applications scale
Hot Spots

• 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

• Research results about 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
Scalable Synchronization

Efficient busy-wait algorithms are possible

- Each processor spins on a separate *locally-accessible* flag variable
  - may be locally-accessible via
    - coherent caching
    - allocation in *local* physically-distributed shared memory

- Some other processor terminates the spin when appropriate
Barrier Design Issues

• Naïve formulation
  —each instance of a barrier begins and ends with identical values for state variables

• Implication
  —each processor must spin twice per barrier instance
    – once to ensure that all processors have left the previous barrier
      without this, a processor can mistakenly pass through current barrier because of state still being used by previous barrier
    – again to ensure that all processors have arrived at current barrier
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 $k^{th}$ 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

• Benefits
  —reduce number of references to shared variables
  —eliminate one of the spinning episodes
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

nodes: 0 1 2 3 4 5 ... P-1

processor private sense : Boolean  := true
processor private mynode : ^node  // my group's leaf in the 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 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
• $\Omega(\log p)$ 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 $\text{ceiling}(\log_2 P)$
  - processor $i$ signals processor $(i + 2^k) \mod P$
    - synchronization is not pairwise
  - processor $i$ waits for signal from $(i - 2^k) \mod P$

- Does not require $P = 2^k$

Technique 2: Paired Data Structure

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

processor private localflags : ^flags

shared allnodes : array [0..P-1] of flags
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

- Improves on earlier "butterfly" barrier of Brooks (*IJPP*, 1986)
- $\Theta(\log p)$ 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 Structure

// initially, rounds[i][k].flag = false for all i,k

// rounds[i][k].role =
// winner if k > 0, i mod 2^k = 0, i + 2^(k-1) < P, and 2^k < P
// bye if k > 0, i mod 2^k = 0, and i + 2^(k-1) >= P
// loser if k > 0 and i mod 2^k = 2^(k-1)
// champion if k > 0, i = 0, and 2^k >= P
// dropout if k = 0
// unused otherwise; value immaterial
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
Tournament Barrier Wakeup

```c
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
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
// on processor i, sense is initially true
// in nodes[i]:
//    havechild[j] = true if 4*i+j < P; otherwise false
//    parentpointer = &nodes[floor((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
procedure tree_barrier
    with nodes[vpid] do
        repeat until childnotready =
            {false, false, false, false}
        childnotready := havechild   // prepare for next barrier
data pointer^ := false        // let parent know I'm ready

        // if not root, wait until my parent signals wakeup
        if vpid != 0
            repeat until parentsense = 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

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
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
Butterfly: All Barriers

![Graph showing time in microseconds for different barrier algorithms](image)
Butterfly: Selected Barriers

![Graph showing selected barriers with time (µs) vs. counter with tree wakeup, bidirectional tournament, tree, and dissemination.]
Sequent: Selected Barriers
Implications for Hardware Design

• Special-purpose synchronization hardware can offer only
  —at best a logarithmic improvement for barriers

• Feasibility of local-spinning algorithms provides a case
  against dance-hall architectures
  —dance-hall = 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 Origin: dual-CPU 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
References
