Barrier Methods

Based on “Algorithms for Scalable Synchronization on Shared Memory Multiprocessors”, by John Mellor-Crummey and Michael Scott

Presentation by Jonathan Pearson

Conventions

- In code snippets, the following may be referenced
  - P = number of processors
  - fetch_and_decrement(address) = function that atomically decrements the value at “address”, returning its value prior to the decrement
  - A ‘shared’ modifier indicates that what follows is shared among all processors
  - A private modifier indicates that each processor has its own copy
  - The ** operator indicates exponentiation
  - The log(x, y) function returns log, x

Centralized Barrier

- The most obvious approach, not attributed to any single person
- Benefits:
  - Simple to program and debug
- Drawbacks:
  - Spinning occurs on a single, centralized location
  - Does not scale well to large numbers of processors

How it Works

- As each processor enters the barrier, it checks to see if it is the last
- If it is, it signals all waiting processors and returns
- If not, it increments a counter and spins on a shared variable until it is signaled

Pitfalls

- Unless done carefully, a processor may enter the barrier before others finish leaving, causing problems

Enhancements

- Sense-Reversing
  - Solves the re-entrant problem
  - Processors each have a ‘sense’ flag, which is compared to a global sense value
  - When different, the barrier is finished and the local sense is flipped
- Geometric Delay
  - Used to save bandwidth by spinning fewer times on the shared location
  - Saves bandwidth, but increases latency for processors leaving the barrier
Code

Code for a sense-reversing, centralized barrier
shared int count = P;
shared bool sense = true;
private bool local_sense = true;

void central_barrier() {
  local_sense = !local_sense;
  if (fetch_and_decrement(&count) == 1) {
    sense = local_sense;
  } else {
    while (sense != local_sense);
  }
}

Pros and Cons

Benefits
- Significantly reduces memory contention
- Prevents tree saturation
  - This is a problem in multistage interconnection networks where too many accesses are going to the same memory module

Drawbacks
- Processors spin on dynamically-determined memory locations
- Multiple processors spin on the same location
  - Not a problem for broadcast-based cache-coherent machines, but otherwise uses significant interconnection bandwidth

How it Works

- Groups of processors are assigned to the leaf nodes of a tree
- As a processor enters the barrier, it checks if it is the last to enter its leaf
  - If not, it spins on a flag in the leaf
  - If so, it moves to the parent and repeats the same operation
- When the last processor reaches the root, it sets the flags in the first layer down, and as nodes awake, they set the flags of their children to wake them

Software Combining Tree Barrier

- Designed by Yew, Tzeng, and Lawrie
- Reduces 'hot-spot' contention for synchronization variables
- Basic idea is to put processors in a tree such that the root is reached only when all processors have entered the barrier

How it Works

- Groups of processors are assigned to the leaf nodes of a tree
- As a processor enters the barrier, it checks if it is the last to enter its leaf
  - If not, it spins on a flag in the leaf
  - If so, it moves to the parent and repeats the same operation
- When the last processor reaches the root, it sets the flags in the first layer down, and as nodes awake, they set the flags of their children to wake them

Code

struct node {
  int k; //number of children/processors assigned here
  int count; //initialized to k
  bool locksense = false;
  struct node *parent;
};

shared struct node *nodes[P]; //distributed widely in mem
private bool sense = true;
private struct node *myNode; //init. to this proc.'s leaf

Code cont'd

void combining_barrier() {
  combining_barrier_aux(myNode);
  sense = !sense; //remember, this is local
}

static void combining_barrier_aux(struct node *n) {
  if (fetch_and_decrement(&n->count) == 1) { //last in
    if (n->parent != NULL)
      combining_barrier_aux(n->parent);
    n->count = n->k;
    n->locksense = (!n->locksense);
  } while (n->locksense != sense);
}
Dissemination Barrier

- Based on “Butterfly Barrier” by Brooks
  - On round k (from 0), processor i syncs with processor i XOR (2 ** k)
  - If P is not a power of 2, existing processors stand in for the missing ones
  - Designed by Hensgen, Finkel, and Manber

How it Works

- In round k, proc i signals proc (i + 2 ** k) % P
  - Requires ceil(log(P)) synchronization ops
- As each processor enters the barrier, it sets a flag for each of a set of ‘partner’ processors
- After setting a flag, the processor waits for its own corresponding flag to be set
- Parity is used to ensure that two instances of the barrier do not interfere

Pros and Cons

Benefits

- Location of flags for processors to spin on are automatically determined
  - Thus spinning is local (on a shared memory/coherent cache machine)
- No two processors spin on the same flag

Drawbacks

- (None known)

Code

```c
struct flags {
    bool myFlags[2][log(P, 2)];
    bool *partnerFlags[2][log(P, 2)];
};

private int parity = 0;
private bool sense = true;
private struct *localFlags = (struct flags *) malloc(sizeof(struct flags));
shared struct flags allNodes[P]; // [i] is in shared mem. accessible to processor i
```

Memory Layout

Initial memory layout

The numbers/colors represent processors and what they own.

The colors also represent relationships between which processors signal which others.
Tournament Barrier
- Designed by Hensgen, Finkey, Manber, and Lubachevsky
- Very similar to the Software Combining Tree Barrier

How it Works
- In round \( k \) (from 0), processor \( i \) sets a flag that is being (or will be) waited on by processor \( j \)
  \[ j = \left(2^m \times k \right) \mod \left(2^m (k + 1)\right) \]
- \( j = \lfloor 2^m k \rfloor \)
- After setting the flag, processor \( i \) drops out and spins on a global flag indicating the state of the barrier
- Processor \( j \) continues up the tree, tournaments continue
- The processor to reach the root sets the global flag
- Complete tournament consists of \( \text{ceil}(\log_2 P) \) rounds

Pros and Cons
- Benefits
  - The processor that continues up the tree is statically determined
  - Avoids fetch* and * operations
  - Two versions, more discussed later
- Drawbacks
  - Two versions, more discussed later

CREW and EREW
- Lubachevsky proposed two versions
  - CREW (Concurrent Read, Exclusive Write) is described above
  - Benefits
    - Appropriate for multiprocessors that use broadcast to main cache consistency
  - Drawbacks
    - DO NOT USE on machines without broadcast-based coherent caches (all processors spinning on the same variable)

CREW and EREW cont’d
- EREW (Exclusive Read, Exclusive Write)
  - More similar to the Software Combining Tree Barrier in that it uses a wakeup tree instead of a global variable
  - Benefits
    - Appropriate for any cache-coherent multiprocessor, even without broadcast
  - Drawbacks
    - Each processor spins on a set of non-contiguous array elements (a real problem without coherent-cache)

Code
- (See handout, very long)
MCS Barrier

- Designed by John M. Mellor-Crummey and Michael Scott
- Benefits
  - Spins only on locally-accessible flags
  - Requires O(P) space
  - Performs the theoretical minimum number of network transactions on machines without broadcast (2P - 2)
  - Performs O(Log P) network transactions on its critical path (path that needs to be sequential)

How it Works

- A tree of P nodes is built, each processor gets a node
- Each node contains a few flags
  - childNotReady – one for each child node, always clear on leaf nodes (they have no children not to be ready)
  - parentSense – toggled when the barrier is complete and being released
  - childPointer – one for each child, points to the parentSense flag in each child node
  - parentPointer – points to the childNotReady flag in the parent corresponding to this node
  - haveChild – one for each possible child, indicates whether it is present

How it Works cont’d

- Upon entering the barrier, a processor spins on all of its childNotReady flags until they are clear (returns instantly for leaf nodes)
  - When they are clear, they are reset to the values in haveChild (sets only ones that must be cleared next time the barrier is run)
  - parentPointer is cleared
  - Processor (not root) spins on parentSense
  - Root sets its childPointer flags to signal its children, and then returns from the barrier
  - All nodes that see their parentSense flags get signaled do the same

Pitfalls

- The MCS barrier uses two separate trees (one for entering, and one for wakeup/exit)
  - For efficiency, the fan-in (number of processors assigned to each leaf of the entrance tree) is 4, but the fan-out (number of children of each node in the wakeup tree) is 2

Code

- (See handout, very long)

Performance – Space

- Centralized Barrier: O(1)
- Software Combining Tree: O(P)
- Dissemination: O(P log P)
- Tournament CREW: O(P log P)
- Tournament EREW: O(P)
- MCS: O(P)
Performance – Critical Path
- Centralized: $O(P)$
- Software Combining Tree: $O(\log P)$
- Dissemination: $O(\log P)$ (shortest by a constant factor)
- Tournament CREW: $O(\log P)$
- Tournament EREW: $O(\log P)$
- MCS: $O(\log P)$

Which to Use: Centralized
- Outperforms all others for small numbers of processors
- Smallest space requirement
- Easily adapts to different numbers of processors in subsequent rounds

Which to Use: Combining Tree
- Linear space requirement
- Short critical path
- Simplest to program of the complex barriers

Which to Use: Dissemination
- Fastest, once past the simplicity of the centralized barrier
- Shortest critical path

Which to Use: Tournament
- Can outperform the dissemination barrier in some situations
  - ‘Sequent Balance’ multiprocessor machine
    (no mention of the specifications of this machine)

Which to Use: MCS
- Linear space requirement
- Short critical path
- Should be the fastest on large-scale multiprocessors that use broadcast to maintain cache coherence
- $O(P)$ updates to shared variables
  - Compare to $O(P \log P)$ in the dissemination barrier
- Uses simple writes, as opposed to atomic combination operations that may not be supported
- Less space than the tournament barrier, code is simpler, and slightly less local work when $P$ is not a power of 2
Real Performance

- Measured on two machines
  - BBN Butterfly
    - Up to 256 processors
    - Each is an 8 MHz MC68000
    - 24-bit virtual addresses
    - 1M of local memory
    - Non-local memory access takes about 4µs, 5x longer than local
    - Supports two 16-bit atomic operations
      - fetch_and_clear_then_add
      - fetch_and_clear_then_xor
    - Each takes three 16-bit arguments
      - destination
      - mask
      - source

BBN Butterfly cont’d

- What they do
  - destination is ANDed with 1’s complement of mask
  - then ADDed or XORed with source
  - result replaces original value in destination
  - original value is returned
- These can be used to build atomic operations such as
  - fetch_and_add
  - fetch_and_store (swap)
  - fetch_and_or

Sequent Symmetry

- The other machine was the Sequent Symmetry
  - Up to 30 processors on a shared bus
  - Each processor is a 16 MHz Intel 80386
  - 64KB 2-way set associative cache
  - Each line includes a flag indicating whether there are other copies
  - Writes are only broadcast if there are other copies of the line
  - Supports fetch_and_store (swap), along with atomic logic operations
  - Works on any 1, 2, or 4-byte quantity
  - Logic operations do not have return values, reducing their usefulness

Performance Graphs: Butterfly

- Supports fetch and store (swap)
- Atomic logic operations

Performance Graph: Symmetry

- Graph showing performance comparison for different operations.
- Key: count, count, count, count, count...