NON-BLOCKING DATA STRUCTURES
AND TRANSACTIONAL MEMORY

Tim Harris, 21 Oct 2016
Lecture 1/3

- Introduction
- Basic spin-locks
- Queue-based locks
- Hierarchical locks
- Reader-writer locks
- Reading without locking
- Flat combining
- Recent research: parallel work distribution
Overview

- Building shared memory data structures
  - Lists, queues, hashtables, ...
- Why?
  - Used directly by applications (e.g., in C/C++, Java, C#, ...)  
  - Used in the language runtime system (e.g., management of work, implementations of message passing, ...)  
  - Used in traditional operating systems (e.g., synchronization between top/bottom-half code)
- Why not?
  - Don’t think of “threads + shared data structures” as a default/good/complete/desirable programming model  
  - It’s better to have shared memory and not need it...
What do we care about?

Correctness: What does it mean to be correct? e.g., if multiple concurrent threads are using iterators on a shared data structure at the same time?

Ease to write: What do we care about? Does it matter? Who is the target audience? How much effort can they put into it? Is implementing a data structure an undergrad programming exercise? …or a research paper?

When can it be used? How well does it scale? How fast is it? Between threads in the same process? Between processes sharing memory? Within an interrupt handler? With/without some kind of runtime system support?

Suppose I have a sequential implementation (no control at all): is the new implementation 5% slower? 100x slower?

How does performance change as we increase the number of threads? When does the implementation add or avoid synchronization?
What do we care about?

- Ease to write
- Correctness
- When can it be used?
- How fast is it?
- How well does it scale?
What do we care about?

1. Be explicit about goals and trade-offs
   - A benefit in one dimension often has costs in another
   - Does a perf increase prevent a data structure being used in some particular setting?
   - Does a technique to make something easier to write make the implementation slower?
   - Do we care? It depends on the setting

2. Remember, parallel programming is rarely a recreational activity
   - The ultimate goal is to increase perf (time, or resources used)
   - Does an implementation scale well enough to out-perform a good sequential implementation?
Suggested reading

- “The art of multiprocessor programming”, Herlihy & Shavit – excellent coverage of shared memory data structures, from both practical and theoretical perspectives
- “Transactional memory, 2\textsuperscript{nd} edition”, Harris, Larus, Rajwar – recently revamped survey of TM work, with 350+ references
- “NOrec: streamlining STM by abolishing ownership records”, Dalessandro, Spear, Scott, PPoPP 2010
- “Simplifying concurrent algorithms by exploiting transactional memory”, Dice, Lev, Marathe, Moir, Nussbaum, Olszewski, SPAA 2010
- Intel “Haswell” spec for SLE (speculative lock elision) and RTM (restricted transactional memory)
Basic spin-locks
bool testAndSet(bool *b) {
    bool result;
    atomic {
        result = *b;
        *b = TRUE;
    }
    return result;
}
Test and set

• Suppose two threads use it at once

Thread 1:

\[ \text{testAndSet}(b) \rightarrow \text{true} \]

Thread 2:

\[ \text{testAndSet}(b) \rightarrow \text{false} \]
Test and set lock

void acquireLock(bool *lock) {
    while (testAndSet(lock)) {
        /* Nothing */
    }
}

void releaseLock(bool *lock) {
    *lock = FALSE;
}

FALSE => lock available
TRUE => lock held

Each call tries to acquire the lock, returning TRUE if it is already held

NB: all this is pseudo-code, assuming SC memory
Test and set lock

void acquireLock(bool *lock) {
    while (testAndSet(lock)) {
        /* Nothing */
    }
}

void releaseLock(bool *lock) {
    *lock = FALSE;
}

lock: TRUE

Thread 1

Thread 2
What are the problems here?

testAndSet implementation causes contention
Contention from testAndSet

- Single-threaded core
- L1 cache
- L2 cache
- Main memory

Non-blocking data structures and transactional memory
Multi-core h/w – separate L2

testAndSet(k)

Single-threaded core

L1 cache

L2 cache

Main memory

Single-threaded core

L1 cache

L2 cache

Non-blocking data structures and transactional memory
Multi-core h/w – separate L2

- Single-threaded core
- L1 cache
- L2 cache
- Main memory

- Single-threaded core
- L1 cache
- L2 cache

- Main memory

- testAndSet\(k\)
Does this still happen in practice? Do modern CPUs avoid fetching the line in exclusive mode on failing TAS?
What are the problems here?

- testAndSet implementation causes contention
- Only supports mutual exclusion: not reader-writer locking
- No control over locking policy
- Spinning may waste resources while waiting
General problem

- No logical conflict between two failed lock acquires
- Cache protocol introduces a physical conflict
- For a good algorithm: only introduce physical conflicts if a logical conflict occurs
  - In a lock: successful lock-acquire & failed lock-acquire
  - In a set: successful insert(10) & failed insert(10)
- But not:
  - In a lock: two failed lock acquires
  - In a set: successful insert(10) & successful insert(20)
  - In a non-empty queue: enqueue on the left and remove on the right
Test and test and set lock

```c
void acquireLock(bool *lock) {
    do {
        while (*lock) {
        }
    } while (testAndSet(lock));
}

void releaseLock(bool *lock) {
    *lock = FALSE;
}
```

FALSE => lock available
TRUE => lock held

Spin while the lock is held... only do testAndSet when it is clear
Performance

Based on Fig 7.4, Herlihy & Shavit, “The Art of Multiprocessor Programming”
**Stampedes**

```c
void acquireLock(bool *lock) {
    do {
        while (*lock) { }
    } while (testAndSet(lock));
}

void releaseLock(bool *lock) {
    *lock = FALSE;
}
```
Back-off algorithms

1. Start by spinning, watching the lock for “s” iterations
2. If the lock does not become free, wait locally for “w” (without watching the lock)

What should “s” be?
What should “w” be?
Time spent spinning on the lock “s”

- **Lower values:**
  - Less time to build up a set of threads that will stampede
  - Less contention in the memory system, if remote reads incur a cost
  - Risk of a delay in noticing when the lock becomes free if we are not watching

- **Higher values:**
  - Less likelihood of a delay between a lock being released and a waiting thread noticing
Local waiting time “w”

- **Lower values:**
  - More responsive to the lock becoming available

- **Higher values:**
  - If the lock doesn’t become available then the thread makes fewer accesses to the shared variable
Methodical approach

- For a given workload and performance model:
  - What is the best that could be done (i.e. given an “oracle” with perfect knowledge of when the lock becomes free)?
  - How does a practical algorithm compare with this?
- Look for an algorithm with a bound between its performance and that of the oracle
- “Competitive spinning”
Rule of thumb

- Spin on the lock for a duration that’s comparable with the shortest back-off interval
- Exponentially increase the per-thread back-off interval (resetting it when the lock is acquired)
- Use a maximum back-off interval that is large enough that waiting threads don’t interfere with the other threads’ performance
Systems problems

- The threads need to “wait efficiently”
- Not consuming processing resources (contending with lock holder) & not consuming power
- “monitor” / “mwait” operations – e.g., SPARC M7
Systems problems

- Spinning gets in the way of other s/w threads, even if done efficiently
- For long delays, may need to actually block and unblock
- ...as with back-off, how long to spin for before blocking?
Queue-based locks
Queue-based locks

- Lock holders queue up: immediately provides FCFS behavior
- Each spins *locally* on a flag in their queue entry: no remote memory accesses while waiting
- A lock release wakes the next thread directly: no stampede
MCS locks

Lock identifies *tail*

QNode 1 -> QNode 2 -> QNode 3

Local flag

FALSE

Head

Tail
void acquireMCS(mcs *lock, QNode *qn) {
    QNode *prev;
    qn->flag = false;
    qn->next = NULL;
    while (true) {
        prev = lock->tail;
        /* Label 1 */
        if (CAS(&lock->tail, prev, qn)) break;
    }
    if (prev != NULL) {
        prev->next = qn; /* Label 2 */
        while (!qn->flag) { } // Spin
    }
}
MCS lock release

```c
void releaseMCS(mcs *lock, QNode *qn) {
    if (lock->tail = qn) {
        if (CAS(&lock->tail, qn, NULL)) return;
    }
    while (qn->next == NULL) { }
    qn->next->flag = TRUE;
}
```

If we were at the tail then remove us

Wait for next lock holder to announce themselves; signal them
Hierarchical locks
Hierarchical locks

- Core 1
- Core 2
- Core 3
- Core 4
- Shared L2 cache

- Core 5
- Core 6
- Core 7
- Core 8
- Shared L2 cache

Memory bus

- Memory

Non-blocking data structures and transactional memory
Hierarchical locks

Core 1
Core 2
Core 3
Core 4
Core 5
Core 6
Core 7
Core 8
Shared L2 cache
Shared L2 cache
Memory bus
Memory

Non-blocking data structures and transactional memory
Hierarchical locks

Pass lock “nearby” if possible

Call this a “cluster” of cores

Non-blocking data structures and transactional memory
Hierarchical TATAS with backoff

void acquireLock(bool *lock) {
    do {
        holder = *lock;
        if (holder != -1) {
            if (holder == MY_CLUSTER) {
                BackOff(SHORT);
            } else {
                BackOff(LONG);
            }
        } while (!CAS(lock, -1, MY_CLUSTER));
    }
}
Hierarchical locks: unfairness v throughput

Avoid this cycle repeating, starving 5 & 7...
Lock cohorting

Lock cohorting

- Lock acquire, uncontended

(1) Acquire local lock

(2) Acquire global lock

Core 1 -> Core 2
Core 3 -> Core 4
Core 5 -> Core 6
Core 7 -> Core 8

Per-NUMA-domain lock SA
Per-NUMA-domain lock SB

System-wide arbitration lock G
Lock cohorting

- Lock acquire, contended

(1) Wait for local lock (e.g., MCS)
Lock cohorting

- Lock release, with successor

(1) Pass global lock to successor
Lock cohorting, requirements

- Global: “thread oblivious” (acq one thread, release another)
- Local lock: “cohort detection” (can test for successors)
Reader-writer locks
Reader-writer locks (TATAS-like)

lock: 0

-1 => Locked for write
0 => Lock available
+n => Locked by n readers

void acquireWrite(int *lock) {
do {
    if ((*lock == 0) &&
        (CAS(lock, 0, -1))) {
        break;
    } while (1);
} }

void acquireRead(int *lock) {
do {
    int oldVal = *lock;
    if ((oldVal >= 0) &&
        (CAS(lock, oldVal, oldVal+1))) {
        break;
    } } while (1);
} }

void releaseWrite(int *lock) {
    *lock = 0;
}

void releaseRead(int *lock) {
    FADD(lock, -1); // Atomic fetch-and-add
}
The problem with readers

```c
int readCount() {
    acquireRead(lock);
    int result = count;
    releaseRead(lock);
    return result;
}
```

```c
void incrementCount() {
    acquireWrite(lock);
    count++;
    releaseWrite(lock);
}
```

- Each `acquireRead` fetches the cache line holding the lock in exclusive mode
  - Again: `acquireRead` are not logically conflicting, but this introduces a physical conflict
- The time spent managing the lock is likely to vastly dominate the actual time looking at the counter
- Many workloads are read-mostly...
Keeping readers separate

Acquire write on core i: CAS the owner from 0 to i

...then spin until all of the flags are clear

Acquire read on core i: set own flag to true...

...then check that the owner is 0 (if not then clear own flag and wait)
Keeping readers separate

- With care, readers do not need to synchronize with other readers
  - Extend the flags to be whole cache lines
  - Pack multiple locks flags for the same thread onto the same line
  - Exploit the cache structure in the machine: Dice & Shavit’s TLRW byte-lock on SPARC Niagara
- If “N” threads is very large..
  - Dedicate the flags to specific important threads
  - Replace the flags with ordinary multi-reader locks
  - Replace the flags with per-NUMA-domain multi-reader locks
Other locking techniques

- **Affinity**
  - Allow one thread fast access to the lock
  - “One thread” – e.g., previous lock holder
  - “Fast access” – e.g., with fewer / no atomic CAS operations
  - Mike Burrows “Implementing unnecessary mutexes” (Do the assumptions hold? How slow is an uncontended CAS on a modern machine? Are these techniques still useful?)
Other locking techniques

- **Affinity**
  - Allow one thread fast access to the lock
  - “One thread” – e.g., previous lock holder
  - “Fast access” – e.g., with fewer / no atomic CAS operations
  - Mike Burrows “Implementing unnecessary mutexes”
    (Do the assumptions hold? How slow is an uncontended CAS on a modern machine? Are these techniques still useful?)

- **Inflation**
  - Start out with a simple lock for likely-to-be-uncontended use
  - Replace with a “proper” lock if contended
  - David Bacon (thin locks), Agesen *et al* (meta-locks)
  - Motivating example: standard libraries in Java
Where are we

- Amdahl’s law: to scale to large numbers of cores, we need critical sections to be rare and/or short
- A lock implementation may involve updating a few memory locations
- Accessing a data structure may involve only a few memory locations too
- If we try to shrink critical sections then the time in the lock implementation becomes proportionately greater
- So:
  - try to make the cost of the operations in the critical section lower, or
  - try to write critical sections correctly without locking
Reading without locking
What if updates are very rare

- Modest number of updates: could use reader-writer locks
- No updates at all: no need for locking
Version numbers

Sequential data structure with write lock

Per-data-structure version number
Version numbers: writers
Version numbers: writers

Writers:
1. Take write lock
2. Increment version number
Version numbers: writers

Writers:
1. Take write lock
2. Increment version number
3. Make update
Version numbers: writers

Writers:
1. Take write lock
2. Increment version number
3. Make update
4. Increment version number
5. Release write lock
Version numbers: readers

Writers:
1. Take write lock
2. Increment version number
3. Make update
4. Increment version number
5. Release write lock

Readers:
1. Wait for version number to be even
Version numbers: readers

**Writers:**
1. Take write lock
2. Increment version number
3. Make update
4. Increment version number
5. Release write lock

**Readers:**
1. Wait for version number to be even
2. Do operation
Version numbers: readers

**Writers:**
1. Take write lock
2. Increment version number
3. Make update
4. Increment version number
5. Release write lock

**Readers:**
1. Wait for version number to be even
2. Do operation
3. Has the version number changed?
4. Yes? Go to 1
Why do we need the two steps?

**Writers:**
1. Take write lock
2. Increment version number
3. Make update
4. Increment version number
5. Release write lock

**Readers:**
1. Wait for version number to be even
2. Do operation
3. Has the version number changed?
4. Yes? Go to 1
Read-Copy-Update (RCU)
Read-Copy-Update (RCU)

1. Copy existing structure
Read-Copy-Update (RCU)

1. Copy existing structure
2. Update copy
Read-Copy-Update (RCU)

1. Copy existing structure
2. Update copy
3. Install copy with CAS on root pointer
Read-Copy-Update (RCU)

- Use locking to serialize updates (typically)
  - ...but allow readers to operate concurrently with updates
- Ensure that readers don’t go wrong if they access data mid-update
  - Have data structures reachable via a single root pointer: update the root pointer rather than updating the data structure in-place
  - Ensure that updates don’t affect readers – e.g., initializing nodes before splicing them into a list, and retaining “next” pointers in deleted nodes
  - Exact semantics offered can be subtle (ongoing research direction)
- Memory management problems common with lock-free data structures
When will these techniques be effective?

- Update rate low
  - So the need to serialize updates is OK
- Readers behaviour is OK mid-update
  - E.g., structure small enough to clone, rather than update in place
  - Readers will be OK until a version number check (not enter endless loops / crash / etc.)
- Deallocation or re-use of memory can be controlled
Flat combining
Flat combining

- “Flat Combining and the Synchronization-Parallelism Tradeoff”, Hendler et al

Intuition:
- Acquiring and releasing a lock involves numerous cache line transfers on the interconnect
- These may take hundreds of cycles (e.g., between cores in different NUMA nodes)
- The work protected by the lock may involve only a few memory accesses...
- ...and these accesses may be likely to hit in the cache of the previous lock holder (but miss in your own)
- So: if a lock is not available, request that the current lock holder does the work on your behalf
Flat combining

<table>
<thead>
<tr>
<th>Lock</th>
<th>Sequential data structure</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Request / response table</td>
</tr>
<tr>
<td></td>
<td>Thread 1</td>
</tr>
<tr>
<td></td>
<td>Thread 2</td>
</tr>
<tr>
<td></td>
<td>Thread 3</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
Flat combining: uncontended acquire

1. Write proposed op to req/resp table
2. Acquire lock if it is free
3. Process requests
4. Release lock
5. Pick up response
Flat combining: contended acquire

1. Write proposed op to req/resp table
2. See lock is not free
3. Wait for response
4. Pick up response
Recent research: Parallel work distribution
The following is intended to provide some insight into a line of research in Oracle Labs. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. Oracle reserves the right to alter its development plans and practices at any time, and the development, release, and timing of any features or functionality described in connection with any Oracle product or service remains at the sole discretion of Oracle. Any views expressed in this presentation are my own and do not necessarily reflect the views of Oracle.
PageRank inner loop
PageRank inner loop
PageRank inner loop
PageRank inner loop
Batch size / load imbalance trade-off

Divide into large batches of vertices
Reduce overheads
Risk load imbalance

Divide into small batches of vertices
Increase overheads distributing work
Achieve better load balance
Batch size / load imbalance trade-off

Variable amount of work per iteration

(Actual data – #out-edges of the top 1000 nodes in the SNAP Twitter dataset)
Example performance
Complete PageRank execution, SNAP LiveJournal data set

8-socket SPARC T5
16 cores per socket
8 h/w threads per core
Batch size / load imbalance trade-off

Typically, choose manually – but getting this right depends on (1) algorithm, (2) machine, (3) data

Our approach: efficient fine-grained distribution

Divide into large batches of vertices
Reduce overheads
Risk load imbalance

Divide into small batches of vertices
Increase overheads distributing work
Achieve better load balance
Consider distributing 0..16000 vertices, batch size 10

8 sockets
Consider distributing 0..16000 vertices, batch size 10

- 8 sockets
- Distribute vertices at start of loop down to per-core counters
- 16 cores per socket
- 8 h/w threads per core
Consider distributing 0..16000 vertices, batch size 10

- 8 sockets
- Distribute vertices at start of loop down to per-core counters
- 16 cores per socket
- Aggregate requests upwards within a core
- 8 h/w threads per core

Per-thread request flags
Consider distributing 0..16000 vertices, batch size 10

- 8 sockets
- Distribute vertices at start of loop down to per-core counters
- 16 cores per socket
- Aggregate requests upwards within a core
- 8 h/w threads per core
Consider distributing 0..16000 vertices, batch size 10

- 8 sockets
- Distribute vertices at start of loop down to per-core counters
- 16 cores per socket
- Aggregate requests upwards within a core
- 8 h/w threads per core

0..125
125..250
Consider distributing 0..16000 vertices, batch size 10

- 8 sockets
  - Distribute vertices at start of loop down to per-core counters
- 16 cores per socket
- 8 h/w threads per core
  - Aggregate requests upwards within a core
  - 3*10
Consider distributing 0..16000 vertices, batch size 10

- 8 sockets
- Distribute vertices at start of loop down to per-core counters
- 16 cores per socket
- Aggregate requests upwards within a core
- 8 h/w threads per core
PageRank – SNAP LiveJournal (4.8M vertices, 69M edges)

Before

Threads

32
64
128
256
512
1024

Batch size

1024 256 64 16 4

After

Threads

32
64
128
256
512
1024

Batch size

1024 256 64 16 4

Normalized execution time

1.0 1.5 2.0 2.5 3.0 3.5 4.0
More details

- Callisto-RTS: Fine-Grain Parallel Loops
- Tim Harris, Stefan Kaestle, *USENIX ATC 2015*