Lecture 5: Fast practical locks, lock-free, consensus, and scalable locks

Teaching assistant: Salvatore Di Girolamo

Motivational video: https://www.youtube.com/watch?v=qx2dRIQXnbs

Nondeterminism in [most] performance measurements!

Same code executed 1000 times. Two metrics measured each time.

How do we report measurements showing high variation?

- The other — not at all!
- One is amazingly stable.

Rule 5: Report if the measurement values are deterministic. For nondeterministic data, report confidence intervals of the measurement.

Administrivia

- Intermediate project presentation: next Monday 10/29 during lecture
- Report will be due in January!
- At the start of class:
  - Basic information
  - The order of talks will be randomized for fairness

Review of last lecture

- Memory models in practical parallel programming
  - Synchronized programming
  - How locks synchronize processes and memory!
- Proving program correctness
  - Pre- / postconditions — sequential
  - Lifting to parallel
  - How to prove locked programs correct (nearly trivial)
- Lock implementation
  - Peterson lock — proof of correctness (using read / write histories, program and visibility orders)
  - With x86 memory model!
- Lock performance
  - Single x86 — how much does memory model correctness cost?

DPHPC Overview

- DPHPC: locality, parallelism, vector ISA, shared memory, distributed memory
- Memory: models
  - locks, wait free, linearizability
- Group communication
  - algorithms
- I/O: complexity
  - balance principles I, balance principles II
  - Little’s Law
Goals of this lecture

- Fast and scalable practical locks!
  - Based on atomic operations
  - Why do we need atomic operations?
- Recap lock-free and wait-free programming
  - Proof that wait-free consensus is impossible without atomics
  - Valence argument: a proof technique similar to showing that atomics are needed for locks
- Locks in practical setting
  - How to block?
  - When to block?
  - How long to block?
  - Simple proof of competitiveness
- Case study: large-scale distributed memory locking
  - Problems and outline to next class

Back to Peterson in Practice ... on x86

- Implement and run our little counter on x86
  - 100000 iterations
  - What is the problem?
  - No sequential consistency for W(I) and R(I)
  - Why?
  - Still 1.3 - 10% errors
  - Goals of this lecture

Peterson in Practice ... on x86

- Implement and run our little counter on x86
  - 1.6 - 10% errors
  - What is the problem?
  - No sequential consistency for W(I) and R(I)
  - Why?
  - Still 1.3 - 10% errors
  - Goals of this lecture

Correct Peterson Lock on x86

- Unoptimized (naïve sprinkling of mfences)
- Performance:
  - No mfence
  - mfence in lock
  - mfence in unlock
  - Two mfence
- Goals of this lecture

Peterson in Practice ... on x86

- Implement and run our little counter on x86
  - 1.6 - 10% errors
  - What is the problem?
  - No sequential consistency for W(I) and R(I)
  - Why?
  - Still 1.3 - 10% errors
  - Goals of this lecture

The compiler may inline this function 😃
**Hardware atomic operations:**
- **Test&Set**
  - Write const to memory while returning the old value
  - Atomic swap
  - Atoms read memory and register
- **Fetch&Op**
- **Compare&Swap**
  - Loads value from memory, allows operations, commits only if no other updates committed
- **Intel TSX (transactional synchronization extensions)**
  - Hardware-TM (roll your own atomic operations)
- **Load-Linked/Store-Conditional (LL/SC)**
  - Loads value from memory, if condition is true!
- **Cacheline contention (or: why I told you about MESI and friends)**
  -’on ARM)
- **Atomic swap**
  - Locks have C=1

**Test-and-Set semantics**
- Assume `TASval` indicates "locked"
- Write something else to indicate "unlocked"
- TAS until return value is `!= TASval` in practice?

**Test-and-Set locks**
- Assume `TASval` indicates "locked"
- TAS until return value is `!= TASval` (1 in this example)

**Relative Power of Synchronization**
- Design-Problem I: Multi-core Processor
  - Which atomic operations are useful?
- Design-Problem II: Complex Application
  - What atomic should I use?

**Test-and-Set locks**
- TAS, Swap, Fetch&Op have C=2
- CAS, LL/SC, TM have C=∞

**Cacheline contention (or: why I told you about MESI and friends)**
- On x86, the XCHG instruction is used to implement TAS
- x86 lock is implicit in `xchg!`
- Cacheline is read and written
  - Dots up in exclusive state, invalidates other copies
  - Cacheline is "thrown" around uselessly
- High load on memory subsystem
  - x86 lock is essentially a full-memory barrier

**Test-And-Set semantics**
- Memoize old value
- Set fixed value `TASval (true)`
- Return old value

**TestAndSet**
```c
bool TestAndSet (bool *flag) {
    bool old = *flag;
    *flag = true;
    return old;
} // all atomic!
```

**Test-And-Set (TATAS) Locks**
- Spinning in TAS is not a good idea
- Spin on cache line in shared state
  - All threads at the same time, no cache coherency/memory traffic
- **Danger!**
  - Efficient but use with great care
  - Generalizations are very dangerous

**TestAndSet**
```c
volatile int lck = 0;
void lock() {
    while (TestAndSet(&lck) == 1); // all atomic!
}
void unlock() {
    lck = 0;
}
```
Warning: Even Experts get it wrong!

- Example: Double-Checked Locking

Problem: Memory ordering leads to race-conditions!

Contention?

- Do TATAS locks still have contention?
- When lock is released, k threads fight for cache line ownership
  - One gets the lock, all get the CL exclusively (serially!)
- What would be a good solution? (think "collision avoidance")

Exponential backoff eliminates contention statistically

Locks granted in unpredictable order

Starvation possible but unlikely

Maximum waiting time makes it less likely

Improvements?

- Are TAS locks perfect?
  - What are the two biggest issues?
  - Cache coherence traffic (contending on same location with expensive atomics)
    - or --
  - Critical section underutilization (waiting for backoff times will delay entry to CR)
  - What would be a good fix for that?
    - How is this solved at airports and shops (often at least)?
  - Queue locks – Threads enqueue
    - Learn from predecessor if it’s their turn
    - Each thread spins at a different location
    - FIFO fairness

Exponential backoff

- volatile int lck = 0;
  void lock() {
    do {
      while (TestAndSet(&lck) == 1);
    } while (time *= 2); // double waiting time
  }
  void unlock() {
    lck = 0;
  }

Queue locks - Threads enqueue

- volatile int lck = 0;
  void lock() {
    do {
      while (TestAndSet(&lck) == 1); 
    } while (time *= 2); // double waiting time
  }
  void unlock() {
    lck = 0;
  }

Perofrmance of Locks


TAS Lock with Exponential Backoff

- Exponential backoff eliminates contention statistically
- Locks granted in unpredictable order
- Starvation possible but unlikely
- Maximum waiting time makes it less likely

Array to implement queue

- Tail-pointer shows next free queue position
- Each thread spins on own location
- 

CLH Lock (1993)

- Qnode objects represent thread state!
- succ_blocked == 1 if waiting or acquired lock
- succ_blocked == 0 if released lock
- List is implicit!
- One node per thread
- Spin location changes
- NUMA issues (cacheless)
- Can we do better?

CLH Lock (1993)

- List-based (same queue principle)
- Discovered twice by Craig, Landin, Hagersten 1993/94
- 2N+M words
- N threads, M locks
- Requires thread-local qnode pointer
  - Can be hidden!

MCS Lock (1991)

- Make queue explicit
  - Acquire lock by appending to queue
  - Spin on own node until locked is reset
- Similar advantages as CLH but
  - Only 2N + M words
  - Spinning position is fixed!
- Benefits cacheless NUMA
- What are the issues?
  - Releasing lock spins
  - More atomics!

Lessons Learned!

- Key Lesson:
  - Reducing memory (coherency) traffic is most important!
  - Not always straightforward (need to reason about CL states)
- MCS: 2006 Dijkstra Prize in distributed computing
  - "An outstanding paper on the principles of distributed computing, whose significance and impact on the theory and/or practice of distributed computing has been evident for at least a decade"
  - "probably the most influential practical mutual exclusion algorithm ever"
  - "satisfies superior to all previous mutual exclusion algorithms"
- fast, fair, scalable widely used, always compared against!

Time to Declare Victory?

- Down to memory complexity of 2N+M
  - Probably close to optimal
- Only local spinning
  - Several variants with low expected contention
- But: we assumed sequential consistency
  - Reality causes trouble sometimes
  - Sprinkling memory fences may harm performance
- Open research on minimally-spinning algorithms!
  - Come and talk to me if you’re interested
Fighting CPU waste: Condition Variables
- Allow threads to yield CPU and leave the OS run queue
- Other threads can get them back on the queue!
- `cond_wait(cond, lock)` – yield and go to sleep
- `cond_signal(cond)` – wake up sleeping threads
- Wait and signal are OS calls
  - Often expensive, which one is more expensive?
  - Wait, because it has to perform a full context switch

When to Spin and When to Block?
- Spinning consumes CPU cycles but is cheap
  - "Steals" CPU from other threads
- Blocking has high one-time cost and is then free
  - Often hundreds of cycles (trap, save TCB ...)
  - Wakeup is also expensive (restore)
  - Also cache pollution
- Strategy:
  - Poll for a while and then block
  - But what is a "while"??

Competitive Spinning
- If T is the overhead to process a wait, then a locking algorithm that spins for time T before it blocks is 2-competitive!
- If randomized algorithms are used, then $e/(e-1)$-competitiveness ($\approx 1.58$) can be achieved
  - See paper above!

Remember: lock-free vs. wait-free
- A lock-free method guarantees that infinitely often some method call finishes in a finite number of steps
- A wait-free method guarantees that each method call finishes in a finite number of steps (implies lock-free)
- Synchronization instructions are not equally powerful!
  - Indeed, they form an infinite hierarchy; no instruction (primitive) in level x can be used for lock-/wait-free implementations of primitives in level z+1.

Concept: Consensus Number
- Each level of the hierarchy has a "consensus number" assigned.
  - Is the maximum number of threads for which primitives in level x can solve the consensus problem
- The consensus problem:
  - Has single function: `decide(v)`
  - Each thread calls it at most once, the function returns a value that meets two conditions: consistency: all threads get the same value validity: the value is some thread's input
- Simplification: binary consensus (inputs in \{0,1\})
Can a particular class solve n-thread consensus wait-free?

A class C solves n-thread consensus if there exists a consensus protocol using any number of objects of class C and any number of atomic registers.

The protocol has to be wait-free (bounded number of steps per thread).

The consensus number of a class C is the largest n for which that class solves n-thread consensus (may be infinite).

Assume we have a class D whose objects can be constructed from objects out of class C. If class C has consensus number n, what does class D have?

Binary consensus with two threads (A, B):

Each thread moves until it decides on a value

May update shared objects

Protocol state = state of threads + state of shared objects

Initial state = state before any thread moved

Final state = state after all threads finished

States form a tree, wait-free property guarantees a finite tree

Example with two threads and two moves each!

Understanding Consensus

Theorem [Herlihy'91]: Atomic registers have consensus number one

I.e., they cannot be used to solve even two-thread consensus! Really?

Proof outline:

Assume arbitrary consensus protocol, thread A, B
Run until it reaches critical state where next action determines outcome [show that it must have a critical state first]
Show all options using atomic registers and show that they cannot be used to determine one outcome for all possible executions:

1) Any thread reads (other thread runs solo until end)
2) Threads write to different registers (order doesn’t matter)
3) Threads write to same register (solo thread can start after each write)

Corollary: It is impossible to construct a wait-free implementation of any object with consensus number of >1 using atomic registers

"perhaps one of the most striking impossibility results in Computer Science" (Herlihy, Shavit)

We need hardware atomics or Transactional Memory!

Proof technique borrowed from:

Very influential paper, always worth a read!

Nicely shows proof techniques that are central to parallel and distributed computing!

Simple RMW operations (Test&Set, Fetch&Op, Swap, basically all functions where the op commutes or overwrites) have consensus number 2!

Similar proof technique (bivalence argument)

CAS and TM have consensus number ∞

Constructive proof!

CAS provides an infinite consensus number

Machines providing CAS are asynchronous computation equivalents of the Turing Machine

Compare and Set/Swap Consensus

const int first = -1;
volatile int thread = -1;
int proposed[n];

int decide(v) {
  proposed[tid] = v;
  if(CAS(thread, first, tid))
    return v; // I won
  else
    return proposed[thread]; // thread won
}
Now you know everything 😊

- Not really … ;-)
- We’ll argue more about performance now!
- But you have all the tools for:
  - Efficient locks
  - Efficient lock-based algorithms
  - Efficient lock-free algorithms (or even wait-free)
  - Reasoning about parallelism!
- What now?
  - A different class of problems
  - Impact on wait-free/lock-free on actual performance is not well understood
  - Relevant to HPC, applies to shared and distributed memory
  - Group communications

Case study: Fast Large-scale Locking in Practice

Locks: Challenges

- We need intra- and inter-node topology-awareness
- We need to cover arbitrary topologies
- We need to distinguish between readers and writers
- We need flexible performance for both types of processes

We need intra- and inter-node topology-awareness

What will we use in the design?
Ingredient 1 - MCS Locks

- Pointer to the queue tail
- Proc
  - Can enter
  - Next proc
- Proc
  - Cannot enter
  - Next proc

Ingredient 2 - Reader-Writer Locks

- Remote Memory Access (RMA) Programming

- How to manage the design complexity?
- How to ensure tunable performance?
- What mechanism to use for efficient implementation?

Remote Memory Access Programming

- Implemented in hardware in NICs in the majority of HPC networks (RDMA support).

RMA-RW - Required Operations

- Memory
  - Proc
    - put
    - get
    - flush
  - Proc
    - put
    - get
    - Fetch-and-Add (FAA)
    - replace
    - Compare-and-Swap (CAS)
MPI RMA: More in the recitation sessions
- Windows expose memory
  - Created explicitly
  - Remote accesses
  - Put, get
  - Atomics (also atomic Put)
  - Get, accumulate (also atomic Get)
  - Fetch and update (faster single-word get, accumulate)
- Synchronization
  - Two modes: passive and active target
    - Passive target: similar to shared memory!
  - Synchronization: flush, flush_local
- Memory model
  - Unified (coherent) and separate (not coherent) view - it's complicated but versatile

Distributed MCS Queues (DQs) - Throughput vs Fairness

- Each element has its own distributed MCS queue (DQ) of writers
- Modular design
- A distributed counter (DC) for every structure
- How to manage the design complexity?
- Each DQ: The maximum number of lock passings within a DQ at level i, before it is passed to another DQ at level i.
- Larger \( T_{L,i} \): more throughput at level i. Smaller \( T_{L,i} \): more fairness at level i.

Distributed Tree of Queues (DT) - Throughput vs Fairness

- Each DQ: The fairness vs throughput of writers
  - \( J_i \): a parameter for the fairness of readers vs writers
  - \( B_i \): a parameter for the throughput of readers vs writers
- DT: The maximum number of consecutive lock passings within a DQ at level i.
**Distributed Counter (DC) - Latency of readers vs writers**

DC: every \( k \)th compute node hosts a partial counter, all of which constitute the DC.

\[ T_{DC} = 1 \]
\[ T_{DC} = 2 \]

锁获取协议（读者）：仅有一个原子的 fetch-and-add (FAA) 操作。

一个写者持有锁

读者到达Cs

读者离开Cs

**Design space**

\[ T_{LI} \]

\[ T_{R} \]


ewer throughput of writers vs readers

\[ T_{L}, i \]

Locality vs fairness (for writers)

Design A

Design B

**Evaluation - Distributed Counter Analysis**

- CSCS Pi Daint (Cray XC30)
- 5272 compute nodes
- 8 cores per node
- 169TB memory
- Microbenchmarks: acquire/release; latency, throughput
- Distributed hashtable

**Evaluation**

- Throughput, 2% writers
- Single-operation benchmark

![Graph showing throughput vs number of MPI processes](image)
Evaluation - Reader Threshold Analysis

Throughput, 0.2% writers, Empty-critical-section benchmark

Evaluation - Comparison to the State-of-the-Art

Evaluation - Distributed Hashtable

Another application area - Databases
- MPI-RMA for distributed databases?
Another application area - Databases

- MPI-RMA for distributed databases on Piz Daint

![Graph showing network vs. compute dominating](image1)

C. Barthels, et al.: Distributed Join Algorithms on Thousands of Cores presented in Munich, Germany, VLDB Endowment, Aug. 2017

Another application area - Databases

- MPI-RMA for distributed databases on Piz Daint

![Graph showing hash join performance](image2)

C. Barthels, et al.: Distributed Join Algorithms on Thousands of Cores presented in Munich, Germany, VLDB Endowment, Aug. 2017