CSC2/458 Parallel and Distributed Systems
PPMI: Synchronization Preliminaries

Sreepathi Pai

February 15, 2018

URCS

Outline

Outline

Embarrassingly Parallel Programs

What are the characteristics of programs that scale linearly?

Embarrassingly Parallel Programs

No serial portion.

I.e., no communication and synchronization.

Critical Sections

Why should critical sections be short?

[A critical section is a region of code that must be executed by a single thread at a time.]

Locks

```
tail_lock.lock()        // returns only when lock is obtained

tail = tail + 1

list[tail] = newdata

tail_lock.unlock()
```

Outline

The Promise of Transactional Memory

```
transaction {
   tail += 1;
   list[tail] = data;
}
```

- Wrap critical sections with transaction markers
- Transactions succeed when no conflicts are detected
- Conflicts cause transactions to fail
    - Policy differs on who fails and what happens on a failure

Implementation (High-level)

- Track reads and writes
  - inside transactions (weak atomicity)
  - everywhere (strong atomicity)
- Conflict when
  - reads and writes "shared" between transactions
  - these may not correspond to programmer-level reads/writes
- Eager conflict detection
  - every read and write checked for conflict
  - aborts transaction immediately on conflict
- Lazy conflict detection
  - check conflicts when transaction end
- May provide abort path
  - taken when transactions fail

Actual Implementations

How can we use cache coherence protocols to implement transactional memory?

Outline

Mutual Exclusion

How do n processes co-ordinate to achieve exclusive access to one or more resources for themselves?

Some strategies

- Take turns
  - Tokens
  - Time-based
- Queue
- Assume you have exclusive access and detect and resolve conflicts

Evaluating Strategies: Correctness

Show that mutual exclusion is achieved (under all possible orderings).

- Does strategy deadlock?
  - What are the conditions for deadlock?
- Does strategy create priority inversions?
  - What is a priority inversion?

Evaluation: Performance

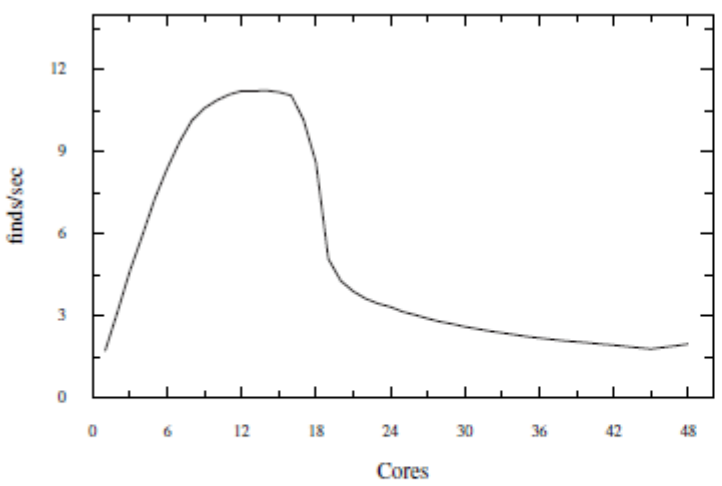
How do we evaluate performance of, say, a particular implementation strategy for locks?

- Use execution time for locking and unlocking?

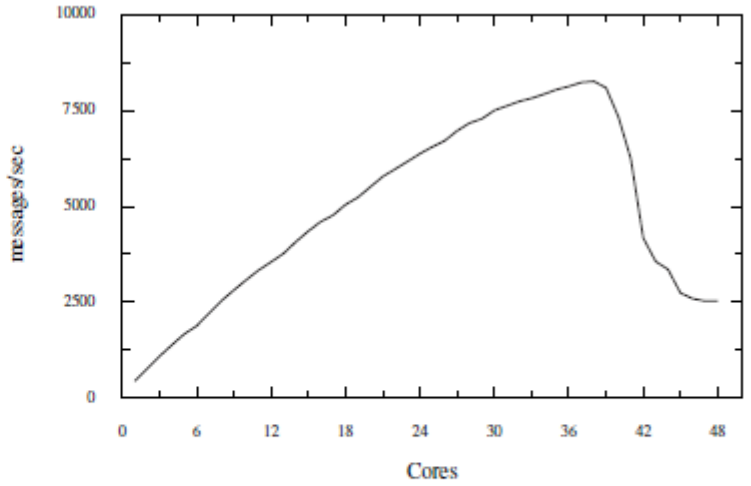
Evaluation: Performance

- Use throughput: Operations/Second
- Vary degree of contention
  - I.e. change number of parallel workers
  - "Low contention" vs "High contention"
- Operations can either be:
  - Application-level operations
  - Lock/Unlock operations

Collapse of Ticket Locks in the Linux kernel



(c) Collapse for PFIND.

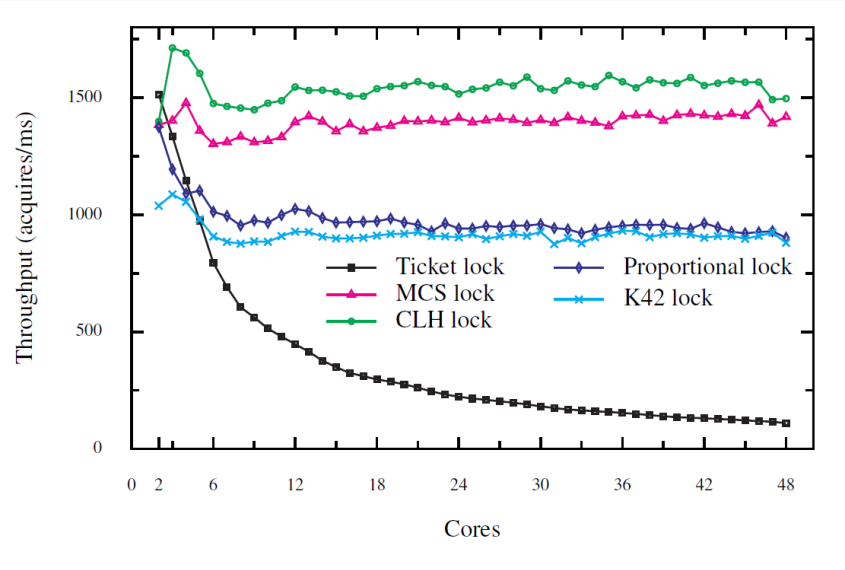


(d) Collapse for EXIM.

Figure 2: Sudden performance collapse with ticket locks.

Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich, "Non-scalable Locks are Dangerous"

Lock Performance



Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich, "Non-scalable Locks are Dangerous"

Evaluation: Fairness/Starvation

Will all workers that need access to a resource get it?

Consider scheduler queues with shortest-job-first scheduling.

Evaluation: Efficiency

- How much storage is required?
- How many operations are used?
    - How much do those operations cost?
- Should you yield or should you spin?

Evaluation: Other Notions

We will examine these notions in more detail in next two lectures:

- Progress
  - System-wide progress ("lock-free")
  - Per-thread ("wait-free")
- Resistance to failure of workers

Outline

Can this happen?

```
   T0                    T1
a = -5                a = 10
```

A later read of a returns -10.

Implementation of Locks

All of the below algorithms require only read/write instructions(?):

- Peterson's Algorithm (for $n = 2$ threads)
- Filter Algorithm (> 2 threads)
- Lamport's Bakery Algorithm

Limitations

- for n threads, require n memory locations
- between a write and a read, another thread may have changed values

Atomic Read–Modify–Write Instructions

- Combine a read–modify–write into a single "atomic" action
- Unconditional
  - *type* `__sync_fetch_and_add (`*type* `*ptr,` *type* `value, ...)`
- Conditional
  - `bool __sync_bool_compare_and_swap (`*type* `*ptr,` *type* `oldval,` *type* `newval`, ...)
  - *type* `__sync_val_compare_and_swap (`*type* `*ptr,` *type* `oldval,` *type* `newval`, ...)
- See GCC documentation
  - `__sync` functions are replaced by `__atomic` functions

AtomicCAS

- (Generic) Compare and Swap
  - `atomic_cas(ptr, old, new)`
  - writes *new* to `ptr` if `ptr` contains *old*
  - returns *old*
- *Only atomic primitive really required*

```
atomic_add(ptr, addend) {
   do {
       old = *ptr;
   } while(atomic_cas(ptr, old, old + addend) != old);
}
```

Locks that spin/Busy-waiting locks

- Locks are initialized to `UNLOCKED`

```
lock(l):
   while(atomic_cas(l, UNLOCKED, LOCKED) != UNLOCKED);

unlock(l):
   l = UNLOCKED;
```

- This is a poor design
  - Why?
- Suitable only for very short lock holds
  - Use random backoff otherwise (e.g. `sleep` or `PAUSE`)

Locks that yield during spinning

- Locks are initialized to `UNLOCKED`

```
lock(l):
   while(atomic_cas(l, UNLOCKED, LOCKED) != UNLOCKED) {
      sched_yield(); // relinquish CPU
   }
```

Performance tradeoffs of spin locks

Operation	Atomics
Lock	unbounded
Unlock	0

- Remember every atomic must be processed serially!

An alternative lock – ticket lock

- Each lock has a ticket associated with it
- Locks and tickets are initialized to 0

```
lock(l):
    // atomic_add returns previous value
    my_ticket = atomic_add(l.ticket, 1);

    while(l != my_ticket);

unlock(l):
    l += 1;   // could also be an atomic_add
```

Performance tradeoffs of ticketlocks

Operation	Atomics	Reads/Writes
Lock	1	unbounded
Unlock	0	1

- Variations on ticket locks are used as high-performance locks today
- We'll study some of these in next lecture.