KAIST SoC
Grad. Algorithms

Concurrent Algorithms

Jeehoon Kang
Concurrency and Parallelism Laboratory
KAIST School of Computing

Introduction

What we want: parallelism

- Definition of **parallelism: spatially coexisting**
- Parallel algorithms: using “spatially coexisting” computations (e.g. “A || B”)
- Parallel architecture: utilizing “spatially coexisting” devices (e.g. cores, memory, PCIe)
- Purpose: better time complexity & energy consumption
- Q: how to **communicate** among spatially coexisting things? (e.g. job-stealing deque)

Communication channels

- FIFO queue
- Network (e.g. system bus, ethernet, …)
- Database (e.g. key-value store, RDBMS)
- **Shared memory** (dominant for software)

- All channels are “**shared mutable state**”
- Summary: parallel things communicate among each other
  via shared mutable states

Communication channels

- FIFO queue
- Network (e.g. system bus, ethernet, …)
- Database (e.g. key-value store, RDBMS)
- **Shared memory** (dominant for software)

Central problem

- All channels are “**shared mutable state**”
- Summary: parallel things communicate among each other via shared mutable states

What we need: concurrency

- Problem: how shared mutable states work
  in the presence of temporally coexisting accesses?
- Definition of **concurrency: temporally coexisting**
- Parallelism requires concurrency
  - e.g. “(A || B); C” requires a channel from A & B to C
  - e.g. Parallel cores require shared-memory concurrency
  - e.g. job-stealing scheduler for parallel jobs require
    job-stealing concurrent deque

Concurrent algorithms

- Algorithms for synchronizing (/coordinating/orchestrating) concurrent accesses to shared mutable states
  - Extremely nondeterministic & complex due to multiple threads of execution
- Usually in the form of **concurrent data structures**
  - e.g. job-stealing deque, concurrent hashmap, spin lock
- Goal: studying a few concurrent algorithms in shared-memory concurrency (dominant for SW)

Shared-memory concurrency

- Dominant concurrency abstraction for modern software
- Shared mutable state: memory
- Concurrent agents: multiple threads of execution
- Synchronization via memory location
  - e.g. message passing (passing value to another thread)

X=42; **F=1**	‖	**while (F==0) { };** **assert(X==42);**

Shared-memory concurrency

- Dominant concurrency abstraction for modern software
- Shared mutable state: memory
- Concurrent agents: multiple threads of execution
- Synchronization via memory location
  - e.g. message passing (passing value to another thread)

Wrong **(store hoisting)**

X=42;	‖	**while (F==0) { };**
F=1		**assert(X==42);**

Relaxed behavior

- Strange things happen **due to HW/compiler optimizations**

Relaxed behavior

- Strange things happen **due to HW/compiler optimizations**

  - Thread A: X=1; a=Y
    Thread B: Y=1; b=X
    (is a=b=0 possible?)

Relaxed behavior

- Strange things happen **due to HW/compiler optimizations**

    - Thread A: X=1; a=Y
      Thread B: Y=1; b=X
      (is a=b=0 possible?)

Load hoisting

Relaxed behavior

- Strange things happen **due to HW/compiler optimizations**

  - Thread A: X=1; a=Y
    Thread B: Y=1; b=X
    (is a=b=0 possible?)

  Load hoisting

  - Thread A: a=Y; X=1
    Thread B: b=X; Y=1
    (is a=b=1 possible?)

Relaxed behavior

- Strange things happen **due to HW/compiler optimizations**

  - Thread A: X=1; a=Y
    Thread B: Y=1; b=X
    (is a=b=0 possible?)

    Load hoisting

  - Thread A: a=Y; X=1
    Thread B: b=X; Y=1
    (is a=b=1 possible?)

    Store hoisting

Relaxed behavior

- Strange things happen **due to HW/compiler optimizations**
  - Thread A: X=1; a=Y
    Thread B: Y=1; b=X
    (is a=b=0 possible?)

    Load hoisting

  - Thread A: a=Y; X=1
    Thread B: b=X; Y=1
    (is a=b=1 possible?)

    Store hoisting

  - Thread A: X++
    Thread B: X++
    (is X=1 possible?)

Relaxed behavior

- Strange things happen **due to HW/compiler optimizations**
  - Thread A: X=1; a=Y
    Thread B: Y=1; b=X
    (is a=b=0 possible?)

    Load hoisting

  - Thread A: a=Y; X=1
    Thread B: b=X; Y=1
    (is a=b=1 possible?)

    Store hoisting

  - Thread A: X++
    Thread B: X++
    (is X=1 possible?)

    Non-atomicity

Common wisdom: locking

- **lock():** disallowing hoisting of later instructions
  - Impossible: lock(); A -> A; lock()
  - Possible: A; lock() -> lock(); A
- **unlock():** not hoisted across earlier instructions
  - Impossible: A; unlock() -> unlock(); A
  - Possible: unlock(); A -> A; unlock()

Common wisdom: locking

- "**Data-race freedom**" theorem (DRF):
  if all concurrent accesses are protected by locks,
  then no relaxed behaviors are observed.

Common wisdom: locking

- “**Data-race freedom**” theorem (DRF):
  if all concurrent accesses are protected by locks,
  then no relaxed behaviors are observed.

  - e.g. message passing w/ proper locking:

    Thread A: X=42; **lock(); F=1; unlock()**
    Thread B: do { **lock(); f=F; unlock()** } while(f==0);
    assert(X==42)

Common wisdom: locking

- “**Data-race freedom**” theorem (DRF):
  if all concurrent accesses are protected by locks,
  then no relaxed behaviors are observed.

  - e.g. message passing w/ proper locking:

    Thread A: X=42; **lock(); F=1; unlock()**
    Thread B: do { **lock(); f=F; unlock()** } while(f==0);
    assert(X==42)

  - e.g. counter w/ proper locking (X=1 impossible):

    Thread A: **lock(); X++; unlock()**
    Thread B: **lock(); X++; unlock()**

Common wisdom: locking

- **Universal**: all sequential data structures
  turn into concurrent when protected with locks.

- **Deadlock-prone**: deadlock may happen if
  multiple threads try to acquire different locks

- **Inscalable**: locking achieves safety basically by
  removing parallelism

Common wisdom: locking

- **Universal**: all sequential data structures turn into concurrent when protected with locks.
- **Deadlock-prone**: deadlock may happen if multiple threads try to acquire different locks
- **Inscalable**: locking achieves safety basically by removing parallelism

Lock-free concurrent programming

Summary

- Parallelism: spatially coexisting
  Concurrency: temporally coexisting

- Parallelism requires concurrent **shared mutable state**

- Shared memory is a dominant SMS for SW,
  but it admits very strange relaxed behaviors
  (trading simplicity for performance)

  - Locking works universally, but it is inscalable

  - Lock-free concurrency is hard due to relaxed behaviors

Key ideas of lock-free programming

Lock-free programming

Complex relaxed behaviors due to nondeterminism & reordering

- **Key challenge 1:** how to tame nondeterminism?
  - **Solution:** by using read-modify-update instructions
- **Key challenge 2:** how to tame reordering?
  - **Solution:** by using acquire load & release store

Lock-free programming

Complex relaxed behaviors due to nondeterminism & reordering

- **Key challenge 1:** how to tame nondeterminism?
  - **Solution:** by using read-modify-update instructions

- **Key challenge 2:** how to tame reordering?
  - **Solution:** by using acquire load & release store

Taming nondeterminism

- Example: counter (# of dropped packets in network sys.)

Taming nondeterminism

- Example: counter (# of dropped packets in network sys.)
    - if (packet.is_dropped()) X++;

Taming nondeterminism

- Example: counter (# of dropped packets in network sys.)
  - if (packet.is_dropped()) X++;

Wrong (nonatomic)

Taming nondeterminism

- Example: counter (# of dropped packets in network sys.)
  - if (packet.is_dropped()) X++;

    Wrong (nonatomic)
  - if (packet.is_dropped()) lock(&D); X++; unlock(&D);

Taming nondeterminism

- Example: counter (# of dropped packets in network sys.)
  - if (packet.is_dropped()) X++;

    Wrong (nonatomic)
  - if (packet.is_dropped()) lock(&D); X++; unlock(&D);

    Lock

Taming nondeterminism

- Example: counter (# of dropped packets in network sys.)
  - if (packet.is_dropped()) X++;

    Wrong (nonatomic)
  - if (packet.is_dropped()) lock(&D); X++; unlock(&D);

    Lock
  - if (packet.is_dropped()) **fetch_add**(&X, 1);

Taming nondeterminism

- Example: counter (# of dropped packets in network sys.)
  - if (packet.is_dropped()) X++; **Wrong (nonatomic)**
  - if (packet.is_dropped()) lock(&D); X++; unlock(&D); **Lock**
  - if (packet.is_dropped()) **fetch_add**(&X, 1); **Lock-free**

Taming nondeterminism

- Example: counter (# of dropped packets in network sys.)
  - if (packet.is_dropped()) X++; **Wrong (nonatomic)**
  - if (packet.is_dropped()) lock(&D); X++; unlock(&D); **Lock**
  - if (packet.is_dropped()) **fetch_add**(&X, 1); **Lock-free**
    - **Read-modify-update (RMW)** instructions:
      Read and write a memory location in a single instruction
    - **Taming nondeterminism by removing "critical" one** (interleaving btw. read & write).

Spinlock (w/o reordering)

- The most basic lock implementation

- struct spinlock {
      lock: atomic<bool>;
  };

- spinlock::lock(spinlock &l) {
      while (!l.lock.swap(true)) {};
  }

- spinlock::unlock(spinlock &l) {
      l.lock.store(false);
  }

Lock-free programming

- **Key challenge 1:** how to tame nondeterminism?
  - **Solution:** by using read-modify-update instructions

- **Key challenge 2:** how to tame reordering?
  - **Solution:** by using acquire load & release store

Lock-free programming

- **Key challenge 1:** how to tame nondeterminism?
  - **Solution:** by using read-modify-update instructions
- **Key challenge 2:** how to tame reordering?
  - **Solution:** by using acquire load & release store

Lock prevents reordering

- **lock():** disallowing hoisting of later instructions
- **unlock():** not hoisted across earlier instructions

Lock prevents reordering

- **lock():** disallowing hoisting of later instructions
- **unlock():** not hoisted across earlier instructions

- **X.load(acquire):** disallowing hoisting of later instructions
- **X.store(42, release):** not hoisted across earlier instructions

Lock prevents reordering

- **lock():** disallowing hoisting of later instructions
- **unlock():** not hoisted across earlier instructions

- **X.load(acquire):** disallowing hoisting of later instructions
- **X.store(42, release):** not hoisted across earlier instructions

Taming reordering by acquire/release instructions

Spinlock (incorrect)

- The most basic lock implementation
- struct spinlock {
    lock: atomic<bool>;
};
- spinlock::lock(spinlock &l) {
    while (!l.lock.swap(true)) {};
}
- spinlock::unlock(spinlock &l) {
    l.lock.store(false);
}

Spinlock (incorrect)

- The most basic lock implementation

- struct spinlock {
    lock: atomic<bool>;
  };

- spinlock::lock(spinlock &l) {
    while (!l.lock.swap(true)) {};
  }

- spinlock::unlock(spinlock &l) {
    l.lock.store(false);
  }

Reordering possible

Spinlock (correct)

- The most basic lock implementation

- struct spinlock {
    lock: atomic<bool>;
  };

- spinlock::lock(spinlock &l) {
    while (!l.lock.swap(true, **acquire**)) {};
  }

- spinlock::unlock(spinlock &l) {
    l.lock.store(false, **release**);
  }

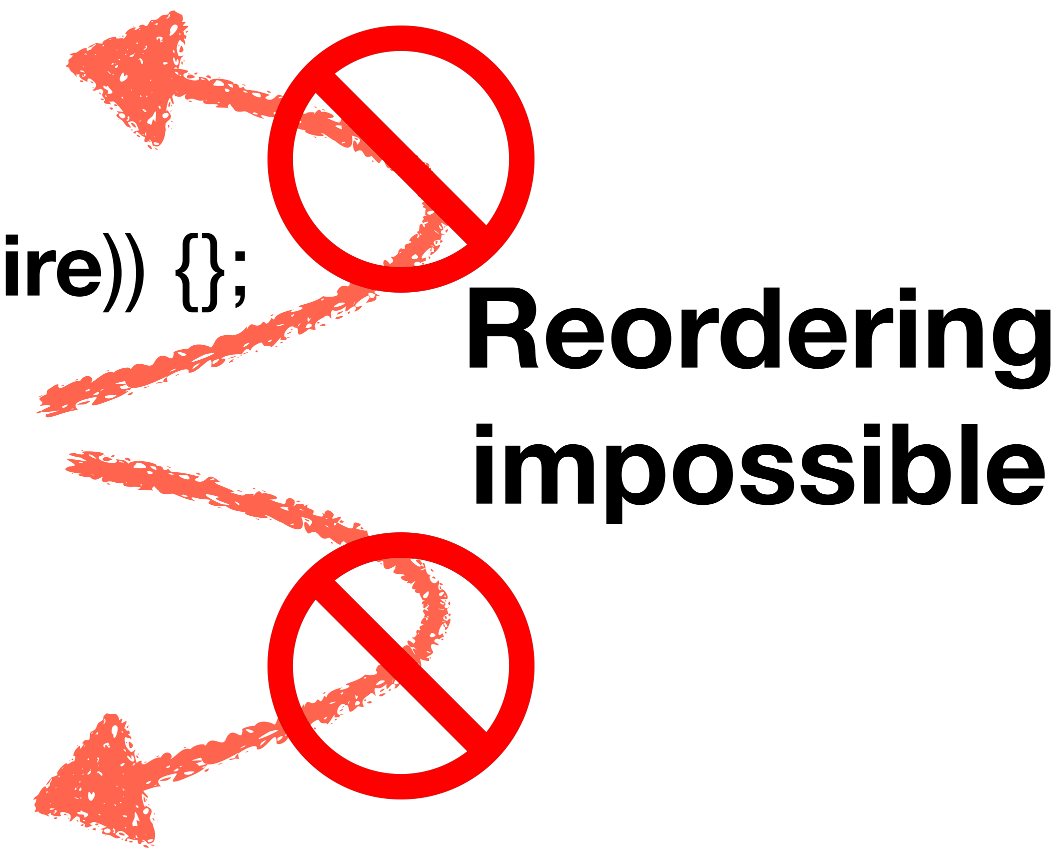
Spinlock (correct)

- The most basic lock implementation

- struct spinlock {
  lock: atomic<bool>;
  };

- spinlock::lock(spinlock &l) {
  while (!l.lock.swap(true, **acquire**)) {};
  }

- spinlock::unlock(spinlock &l) {
  l.lock.store(false, **release**);
  }



Lock-free programming

- **Key challenge 1:** how to tame nondeterminism?
    - **Solution:** by using read-modify-update instructions

- **Key challenge 2:** how to tame reordering?
    - **Solution:** by using acquire load & release store

Versatility of RMW & RA

- We can implement spinlock w/ RMW & release/acquire.
- Actually, we can implement **most** concurrent data structures w/ RMW & release/acquire.
  - Concurrent stack, queue, hash table, trie, b-tree, balanced tree (AVL and red-block trees), …
- Next class: implementing a concurrent stack

Concurrent stack

TODO