Concepts and Models of Parallel and Data-centric Programming

Shared Memory VI

Lecture, Summer 2020

Dr. Christian Terboven <terboven@itc.rwth-aachen.de>

i12 High Performance Computing | RWTH AACHEN UNIVERSITY

Outline

Implementation of a lock

First approach: naïve lock

```
bool flag[2];   // initialization w/ false
void lock() {
  flag[i] = true;
  while (flag[j]) {}
}
```

- Illustration for two threads (`i` and `j`) only
  - Each thread has a flag
  - i = 0, j = 1
  - `unlock()`: set thread's flag back to `false`

- Would this work?

First approach: naïve lock

```
bool flag[2];    // initialization w/ false
void lock() {
   flag[i] = true;
   while (flag[j]) {}
}
```

set flag

Wait for other flag to become false

- Illustration for two threads (`i` and `j`) only
  - Each thread has a flag
  - i = 0, j = 1
  - `unlock()`: set thread's flag back to `false`

- Would this work?

Peterson’s Algorithm

- Lock algorithm
  - that does not fail when concurrent
  - that does not deadlock when sequential

```
void lock() {
 flag[i] = true;
 victim  = i;
 while (flag[j] && victim == i) {};
}
void unlock() {
 flag[i] = false;
}
```

Peterson’s Algorithm

- Lock algorithm
  - that does not fail when concurrent
  - that does not deadlock when sequential

```
void lock() {
  flag[i] = true;
  victim  = i;
  while (flag[j] && victim == i) {};
}
void unlock() {
  flag[i] = false;
}
```

Announce I am interested

Peterson’s Algorithm

- Lock algorithm
  - that does not fail when concurrent
  - that does not deadlock when sequential

```
void lock() {
  flag[i] = true;
  victim  = i;
  while (flag[j] && victim == i) {};
}
void unlock() {
  flag[i] = false;
}
```

Announce I am interested

Defer to other

Peterson’s Algorithm

- Lock algorithm
  - that does not fail when concurrent
  - that does not deadlock when sequential

```
void lock() {
  flag[i] = true;
  victim  = i;
  while (flag[j] && victim == i) {};
}
void unlock() {
  flag[i] = false;
}
```

Announce I am interested

Defer to other

Wait while other is interested and I am the victim

Peterson's Algorithm

- Lock algorithm
  - that does not fail when concurrent
  - that does not deadlock when sequential

```
void lock() {
  flag[i] = true;
  victim  = i;
  while (flag[j] && victim == i) {};
}
void unlock() {
  flag[i] = false;
}
```

Announce I am interested

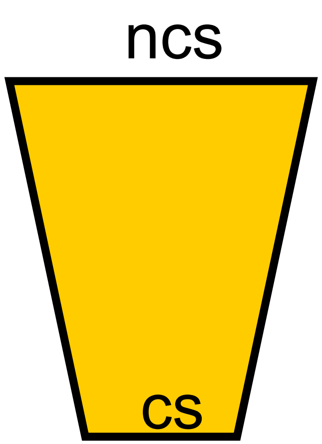
Defer to other

Wait while other is interested and I am the victim

I am no longer interested

Remark on generalization

- Generalization for Peterson’s Algorithms for n threads: Filter Alg.
  - The Filter lock creates n-1 ”waiting rooms” (levels) that a thread must traverse before acquiring the lock
    - At each level
      - At least one enters level
      - At least one blocked if many try
    - Only one thread makes it through



- Issue of Filter Alg.: Fairness
  - A thread can be overtaken arbitrary number of times
  - One solution: bakery algorithm with first come first serve property
  - But impractical efficiency!!!

Test-and-set locks

Overview

- The previous examination of a lock protocol was
  - helpful in understanding the foundations
  - meant to illustrate the underlying problems

Overview

- The previous examination of a lock protocol was
  - helpful in understanding the foundations
  - meant to illustrate the underlying problems

- However, in the “real” world, these protocols are not efficient

- Modern processors: MIMD (Multiprocessors)
  - Multiple instruction, multiple data
  - Provide a test-and-set instruction …
  - … that can be used to implements locks efficiently

Test-and-set Instruction

- Modern architectures provide a test-and-set instruction
  - Write 1 to a memory location
  - Return its old value
  - Both is done as a single *atomic* operation

Test-and-set Instruction

- Modern architectures provide a test-and-set instruction
  - Write 1 to a memory location
  - Return its old value
  - Both is done as a single *atomic* operation

- Atomic Operation:
  - A **set of operations** that appears to the rest of the system to occur at once without being interrupted (it is uninterruptable and indivisible).

Test-and-set Instruction

- Modern architectures provide a test-and-set instruction
  - Write 1 to a memory location
  - Return its old value
  - Both is done as a single *atomic* operation

- Atomic Operation:
  - A **set of operations** that appears to the rest of the system to occur at once without being interrupted (it is uninterruptable and indivisible).

- General: Test-and-set (TAS)
  - Swap **true** with current value
  - Return value tells if prior value was **true** or **false**

Test-and-set Lock

- Locking
  - Lock is free: value is false
  - Lock is taken: value is true
- Acquire lock by calling TAS
  - If result is false, you win
  - If result is true, you lose
- Release lock by writing false

- Why is this more efficient than the Peterson Algorithm?

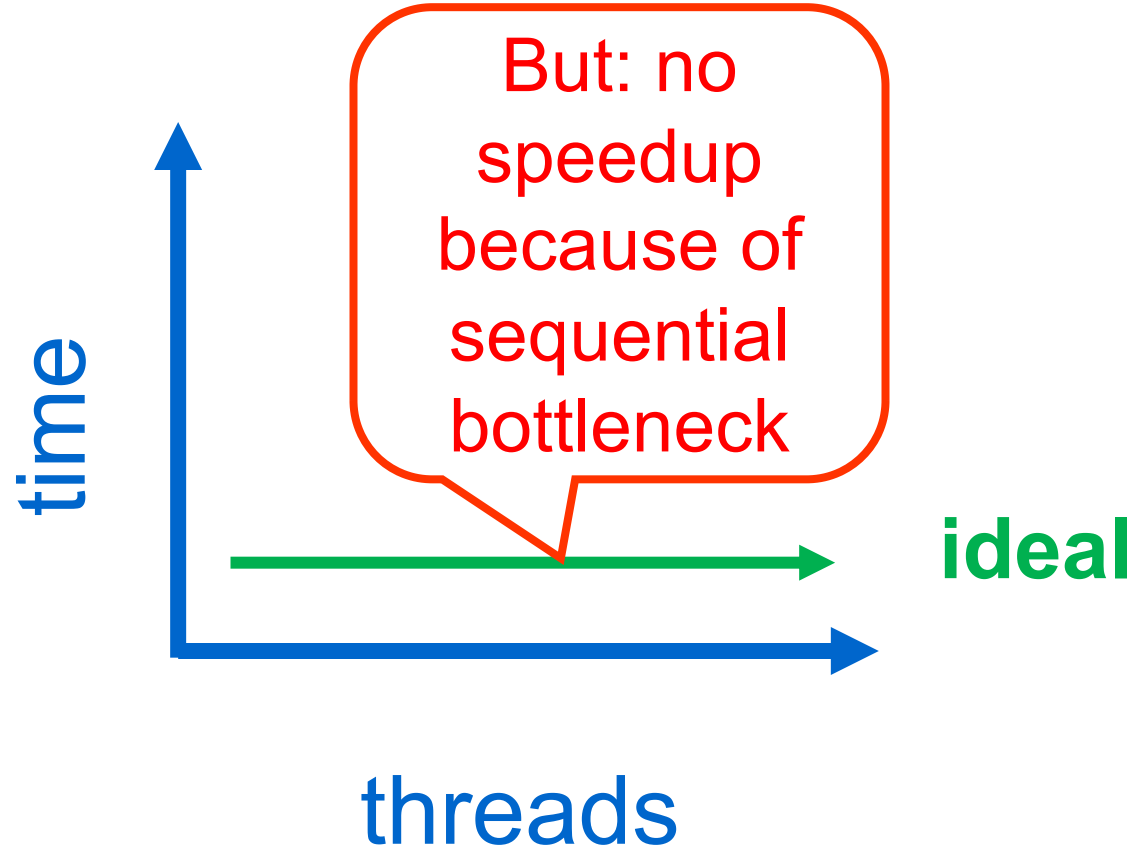
Test-and-set Lock

- Locking
  - Lock is free: value is false
  - Lock is taken: value is true
- Acquire lock by calling TAS
  - If result is false, you win
  - If result is true, you lose
- Release lock by writing false

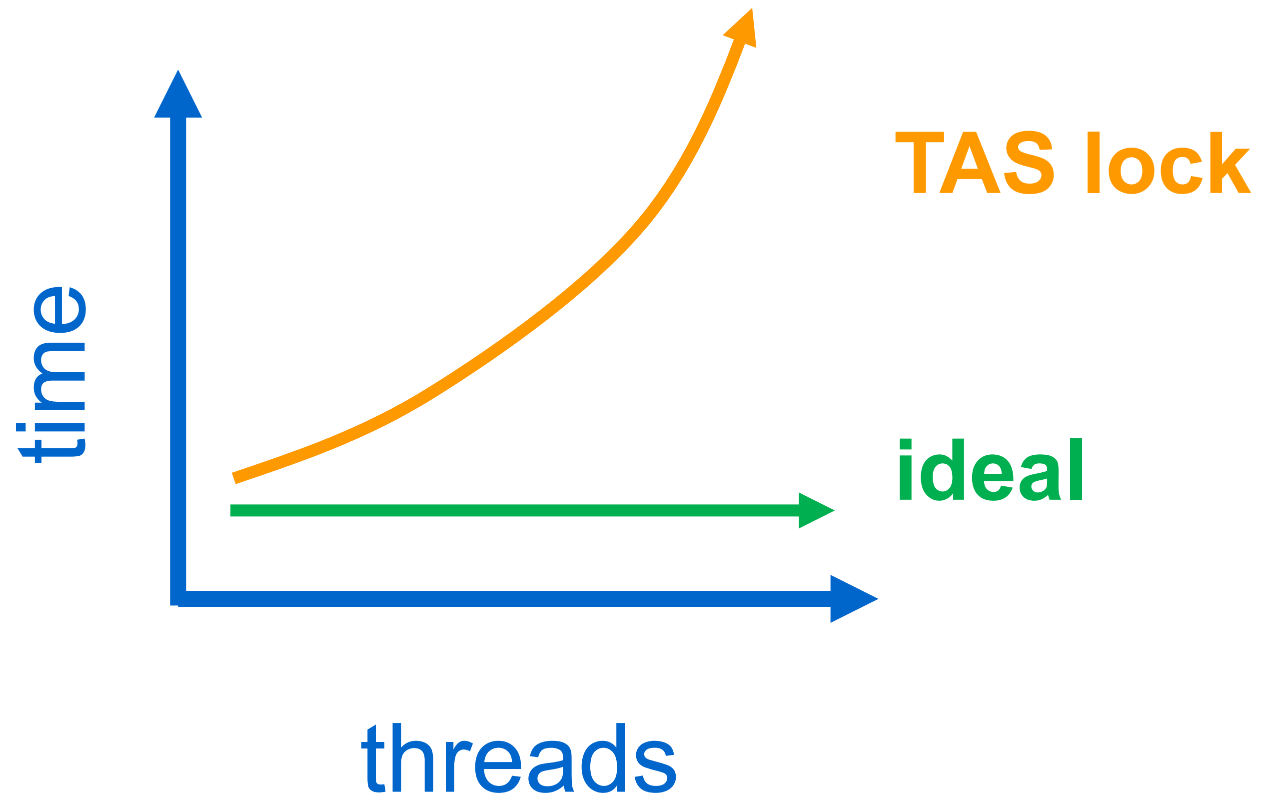
- Why is this more efficient than the Peterson Algorithm?
- Because: O(1) space / instances (as opposed to O(n))!

Remark: do-it-yourself vs. standard implementations

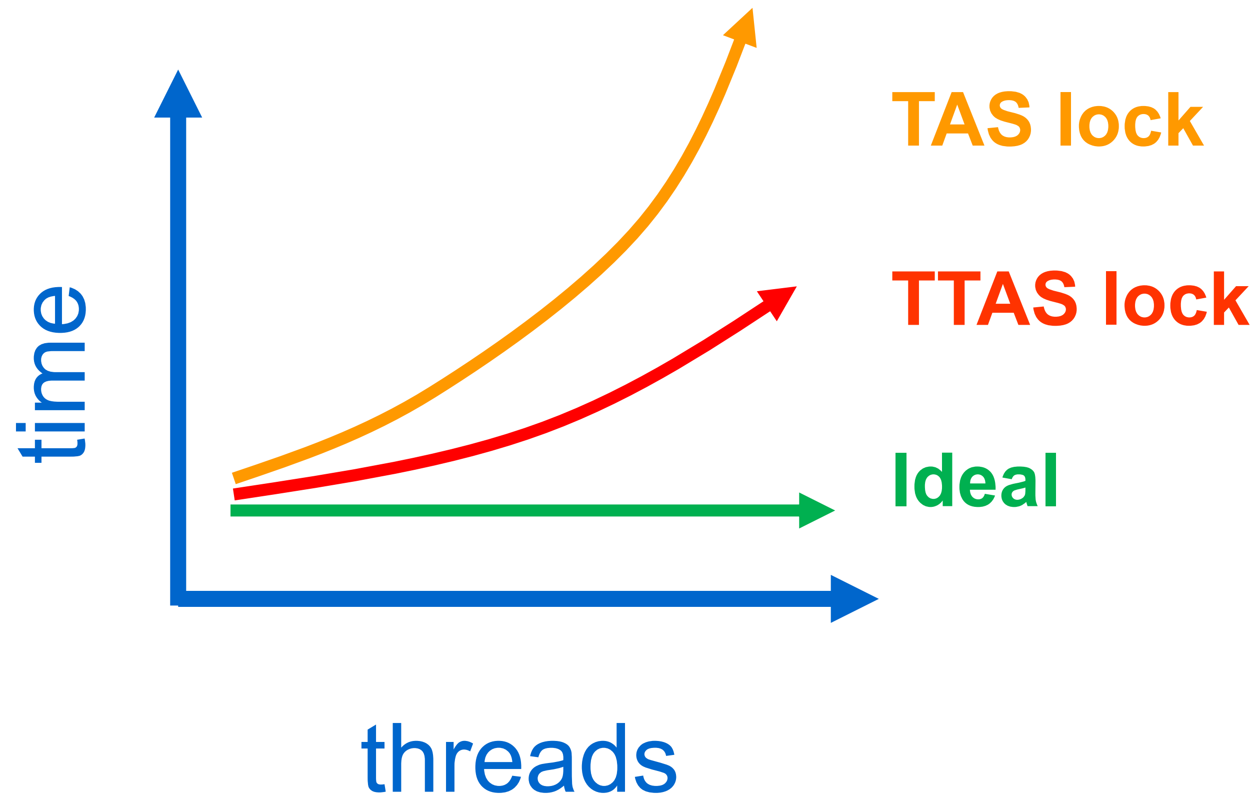
Performance Considerations / 1



Performance Considerations / 2



Performance Considerations / 3



- TTAS: Locking is split up into two phases:
  - Waiting until the lock is available … before trying to acquire the lock

Simplified Explanation

- TAS invalidates cache lines
  - Spinners
    - Miss in cache
    - Go to bus
  - Thread wants to release lock: delayed behind spinners

Simplified Explanation

- TAS invalidates cache lines
  - Spinners
    - Miss in cache
    - Go to bus
  - Thread wants to release lock: delayed behind spinners
- TTAS waits until lock “looks” free
  - Spin on local cache
  - No bus use while lock busy
  - Problem: when lock is released: invalidation storm …

Simplified Explanation

- TAS invalidates cache lines
  - Spinners
    - Miss in cache
    - Go to bus
  - Thread wants to release lock: delayed behind spinners
- TTAS waits until lock “looks” free
  - Spin on local cache
  - No bus use while lock busy
  - Problem: when lock is released: invalidation storm …
- Solution 1: learn (much) more about system architecture

Simplified Explanation

- TAS invalidates cache lines
  - Spinners
    - Miss in cache
    - Go to bus
  - Thread wants to release lock: delayed behind spinners
- TTAS waits until lock “looks” free
  - Spin on local cache
  - No bus use while lock busy
  - Problem: when lock is released: invalidation storm …
- Solution 1: learn (much) more about system architecture
- Solution 2: use lock methods provided by runtime system ☺