University of Leeds

Lecture 7: Lock and mute

Previous lecture

In the last lecture we saw how critical regions of code could be **seria**

-
- Avoids **data races**.
- Can incur a significant **performa**
- Implemented in OpenMP as #pra
- Single arithmetic instructions can be optimised by using **atomic** instructions (`#pragma omp atomic`).

Today's lecture

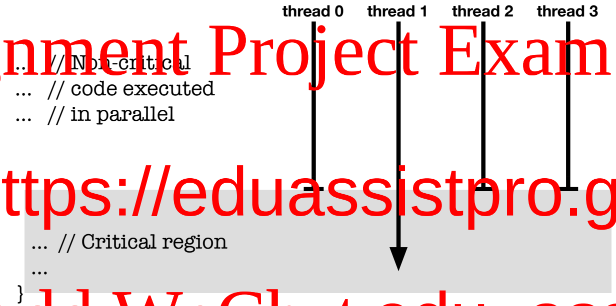
For today's final lecture on shared memory parallelism, we will look at what is going on 'behind the scenes'.

- Thread coordination performed using **locks**, sometimes known
-
- ess.
- However, multiple locks can give rise to

This lecture is largely theoretical[1] a courseworks, but the material may appear in the exam.

[1] There **are** code examples for this lecture, and a question on Worksheet 1.

Recap: Critical regions



- Instructions before #pragma omp cri **concurrently** (*e.g.* if in a parallel loop).
- Instructions in the scope ('{' to '}') only executed by **one thread at a time**.
- Other threads blocked from entering; they are **idle**.

Thread coordination with locks

This **synchronisation** is performed using a **lock**:

-
-
-
    - Also known as **acquiring the lock**.
    - This thread is said to be the lock's

- No other threads can enter the region ('acqu
  until it becomes unlocked.
- The owning thread **unlocks** (or **releases**) it when leaving the region, allowing another thread to take over ownership.

Critical region using a lock

OpenMP:

```
// Multiple threads exec-
// uting co
// (e.g. pa

#pragm
{
   ...
   ...   // Critical code.
   ...
}
```


Lock pseudocode:

```
// All threads access a

..
..     // C
..
re
```


regionLock.lock() does not return until the thread has **acquired** the lock; it is said to be **blocking**.

Implementations of locks

Most parallel APIs support **locks**, although they are sometimes called **mutexes** as they control **mutual exclusion**:

- cks).
-
- `pthread_mutex_t` in the `pthreads` library (C/C++).

When implemented as classes, they are typically

- The user does not have access to instance variables or details of the implementation.

Locks in OpenMP

OpenMP runtime library also supports locks:


```
#include <omp.h>

// Initi
omp_l
omp_i
...                                  // (in parallel).
omp_set_lock(&regionLock);           // LOCK.
...                                  // (critical code).
omp_unset_lock(&regionLock);         // UNLOCK.
...
// Deallocate the lock.
omp_destroy_lock(&regionLock);
```


You *could* implement your own critical region this way, although it is easier to use `#pragma omp critical`.

Programming locks

Note there is no **explicit** link between the lock `regionLock` and the critical region of code, or data structure, that it is trying to prot

> It is do
> associated block of critical code, or data

This gives greater **flexibility**, but also great programming errors.

- Could use a `struct` or `class` to keep the lock with the data it is protecting, with the lock private/protected.

Lock mistakes (1): Forgetting to `lock()`


```
lock_t regionLock;
...
//regionLock.lock();  // Forgot to lock()!
...
...  // Criti
...
regi
```


This is precisely the situation we were trying to avoi

- **All** threads enter the critical region.
- **Race conditions** become a possibilit

`unlock()` will have no effect, except possibly a small performance overhead[1].

[1] Generally, this depends on the API: In C++11, attempting to unlock a `std::mutex` that is **not** locked leads to undefined behaviour.

Lock mistakes (2): Forgetting to `unlock()`


```
lock_t regionLock;
...
regio
...
...    // Criti
...
//regionLock.unlock();  // Forgot to unlock!
```


- The first thread **exclusively** enter
- It never **releases** the lock.
- Therefore no other thread can **acquire** the lock.
- **All other threads remain idle at** `lock()`.

RAII = Resource Acquisition Is Initialisation.

This second mistake is easier to make than it seems:

- The critical code may throw an **exception** (C++/Java).
- ck().

May b
leave their scope.

- If defined at start of a routine, automatically r
  of routine **however it reached there**

- *e.g.* `std::lock_guard<std::mutex>` in C++11.

This mechanism is generally known as RAII, for Resource Acquisition Is Initialisation.

Multiple locks

Code on Minerva: `multipleLockCopy.c`

Suppose we want to copy randomly-selected elements of an array data of size N to another randomly-selected element.


```
#prag
for( n=0
{
  i = rand() % N;
  j = rand() % N;

  omp_set_lock( &entireLock );     // Lock.
  data[j] = data[i];               // Safe copy.
  omp_unset_lock( &entireLock );   // Unlock.
}
```

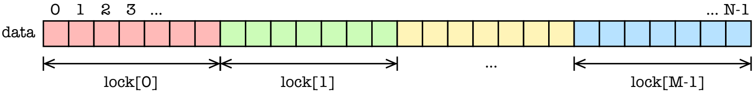
Multiple locks for memory access

This works, but is very inefficient.

- Only one thread can access the array data at a time.
-

Better to use **multiple locks** spanning t

- Different threads can write to different regi **simultaneously**.
- Less **idle time** spent waiting for a lock to be released.



Using multiple locks is measurably faster *(try the code)*:


```
omp_lock_t partialLocks[M];

// Initialise M locks near start of code.
...
// Ident
int lock = M*
omp_s
data[
omp_unset_lock( &partialLocks[lock] );
...
// Destroy all locks at end of code
```


Note we only lock for the **write** to element j

- Recall that just reading does **not** invoke a data race.

Multiple locks for swapping

Code on Minerva: `multipleLockSwap.c`

Suppose now we want to **swap** elements i and j.

- Want to protect **each write** during the swap.

If acce
woul


```
omp_set_lock( &entireLock );

// Writes to both data[i] and data[j].
float temp = data[i];
data[i] = data[j];
data[j] = temp;

omp_unset_lock( &entireLock );
```


However, performance would again be poor.

Multiple locks for swapping

We might think of using two locks, one for each region of the array being written to:


```
int lock_i = M*i/N;
int lock

omp_s
omp_s

float temp = data[i];
data[i] = data[j];
data[j] = temp;

omp_unset_lock( &partialLocks[lock_i] );
omp_unset_lock( &partialLocks[lock_j] );
```

Try this out!

Why does this fail? Deadlock

Suppose one thread tries to lock `lock_i` then `lock_j` and **simultaneously** another tries to lock `lock_j` *then* `lock_i`.

- _ _
- _ _

Sinc

release the lock they own. **They will bot**

> Threads waiting for synchronisation events th
> known as **deadlock**.

The ‘forgetting to `unlock()`’ example earlier is also **deadlock**.

Nested critical regions

Code on Minerva: `nestedCriticalRegion.c`

Another problem is when `lock_i==lock_j`. A simpler example where this occurs is for **nested critical regions**:


```
// Outer c
omp_s

// Inner c
omp_set_lock( &lock );
...
omp_unset_lock( &lock );   // End of inner region

omp_unset_lock( &lock );   // End of outer region
```


In OpenMP, this will also **deadlock**.

- A thread that **owns** a lock cannot **re-acquire** the lock.

Nested #pragma omp critical

OpenMP does **not** allow nested critical regions:

```
#pragma omp critical
{
  ...
  #pragma omp critical
  {
    ...
  }
}
```


. . . will **not** compile.

- The **same** lock is being used by **both** critical sections.
- The same problem as in the previous slide.

Named critical regions

This can be resolved by using **named** critical regions:


```
#pragma omp critical (OUTER)
{
  ..
  #p
  {

  }
}
```


- OUTER and INNER are user-defined la
- Each unique label corresponds to a unique lock.
- You are implicitly using a different lock for each critical region, so no thread tries to re-acquire a lock it already owns.

Reacquiring locks

OpenMP code **deadlocks** if a thread tries to reacquire a lock it already owns.

-

This is

- necessary.

Not all parallel/concurrent APIs impose the sam need to check the documentation!

- *e.g.* For C++11's `std::mutex`, the behaviour is undefined.
- Should also check documentation if attempting to `unlock` a lock that was **not** acquired.

Summary of shared memory systems

Lecture	Content	Key points
	Arch	rnel
	Data par	inism; s.
4	Theory	Amdahl's law (strong scaling); Gustafs
5	Data races	Loop parallelism; data dependencies.
6	Critical regions	Thread coordination; thread safety; serialisation; atomics.
7	Locks/mutexes	Performance costs for locks; deadlock; named critical regions.

Next lecture

Next time we will start to look at **distributed memory systems**:

Not surprisingly, data races are *not* an is
aspects we have covered *are*:

- Non-determinism, scaling, deadlock, d
  parallelism, . . .