XJCO3221 Parallel Computation

Peter Jimack

University of Leeds

Lecture 7: Lock and mutexes

Previous lecture

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

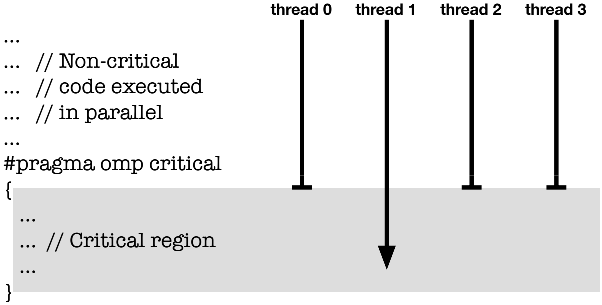
- Only one thread can enter the region at a time, a form of **coordination**.
- Avoids **data races**.
- Can incur a significant **performance penalty**.
- Implemented in OpenMP as `#pragma omp critical`
- Single arithmetic instructions can be optimised by using **atomic** instructions (`#pragma omp atomic`).

This lecture

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

- Thread coordination performed using **locks**, sometimes referred to as **mutexes**.
- Locks can control access to **data structures**.
- Multiple locks can improve **performance** of memory access.
- However, multiple locks can give rise to **deadlock**.

Recap: Critical regions



- Instructions before `#pragma omp critical` executed **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**:

- Single lock for the critical region.
- Can be in one of two states: **locked** and **unlocked**.
- The first thread to reach the critical region **locks** it.
    - Also known as **acquiring the lock**.
    - This thread is said to be the lock's **owner**.
- No other threads can enter the region ('acquire the lock') 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 concurrently,
// (e.g. parallel loop).

#pragma omp critical
{
  ...
  ...  // Critical code.
  ...
}
```

Lock psuedocode:

```
// All threads access a
// single lock object.
lock_t regionLock;

regionLock.lock();

...
...  // Critical code.
...
regionLock.unlock();
```

`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 **mutex**es as they control **mutual exclusion**:

- Java's `Lock` interface (in `java.util.concurrent.locks`).
- `std::mutex` in C++11 (in `<mutex>`).
- `pthread_mutex_t` in the `pthreads` library (C/C++).

When implemented as classes, they are typically **opaque**:

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

Locks in OpenMP

OpenMP also supports locks.

```
#include <omp.h>

// Initialise lock (opaquely).
omp_lock_t regionLock;
omp_init_lock(&regionLock);
...                                   // (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 protect.

> It is down to the **programmer** to correctly link each lock with its associated block of critical code, or data structure.

This gives greater **flexibility**, but also greater scope for 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()!
...
...  // Critical code
...
regionLock.unlock();
```

This is precisely the situation we were trying to avoid!

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

`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;
...
regionLock.lock();
...
...  // Critical code
...
//regionLock.unlock();  // Forgot to unlock!

```

- The first thread **exclusively** enters the critical region.
- 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).
- A `break` or `continue` command may jump over `unlock()`.

May be support for locks that automatically release when they leave their scope.

- If defined at start of a routine, automatically released at end 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.

- Decide to use a **lock** to control data writes.

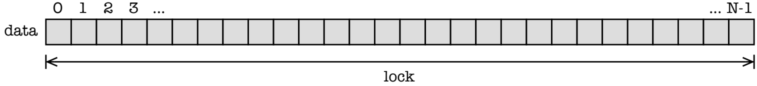
```
#pragma omp parallel for
for( n=0; n<N; n++ )
{
  i = rand() % N;
  j = rand() % N;

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

Multiple locks for memory access

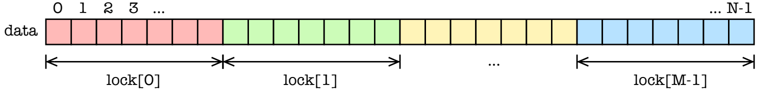
This works, but is very inefficient.

- Only one thread can access the array `data` at a time.
- Even though only writing to one value.



Better to use **multiple locks** spanning the array:

- Different threads can write to different regions of the array **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.
...
// Identify lock for writing to array.
int lock = M*j/N;
omp_set_lock( &partialLocks[lock] );
data[j] = data[i];
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 access to the whole array was governed by a single lock, this would be straightforward to implement:

```
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_j = M*j/N;

omp_set_lock( &partialLocks[lock_i] );
omp_set_lock( &partialLocks[lock_j] );

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`.

- Thread 1 **owns** `lock_i`, **waits** for `lock_j` to be released.
- Thread 2 **owns** `lock_j`, **waits** for `lock_i` to be released.

Since each thread is waiting for the other lock, they will never release the lock they own. **They will both wait forever**.

> Threads waiting for synchronisation events that will never occur is 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 critical region
omp_set_lock( &lock )

// Inner critical region
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)
{
  ...
  #pragma omp critical (INNER)
  {
    ...
  }
}
```

- OUTER and INNER are user-defined labels.
- 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.

- Just as it does not support nested critical regions.

This is primarily due to performance considerations:

- Would incur an execution overhead that would not always be necessary.

Not all parallel/concurrent APIs impose the same requirement - 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
2	Architectures and OpenMP	Cache coherency; false sharing; kernel *versus* user threads.
3	Data parallel problems	Fork-join construct; non-determinism; embarrassingly parallel problems.
4	Theory	Amdahl's law (strong scaling); Gustafson-Barsis law (weak scaling).
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**:

- Multiple **processes** with their own **heap memory**.
- Examples: Clusters, super-computers.

Not surprisingly, data races are *not* an issue, but many of the other aspects we have covered *are*:

- Non-determinism, scaling, deadlock, data and loop parallelism, . . .