

# XJCO3221 Parallel Computation

Peter Jimack

University of Leeds

Lecture 6: Critical regions and atomics

## Previous lectures

In the last lecture we looked at **data races** and **loop parallelism**:

- If different threads access the same memory location, with at least one having write access, there is a potential **race condition**.
- Leads to **non-deterministic** behaviour.
- Can arise in loops as **data dependencies**.
- Often possible to remove these dependencies, sometimes at the expense of increased **parallel overheads**.

# This lecture

In this lecture we will look at an important concept in parallel programming: **synchronisation**.

- Represents some form of **coordination between processing units** (threads *etc.*).
- Will arise in many contexts throughout this module.
- Briefly mentioned in Lecture 4 in the context of **fork-join**.

We will focus on using synchronisation to avoid data races.

- Define **critical regions** which can only be accessed by one thread at a time.
- **Atomics**: Critical regions specialised to single operations.

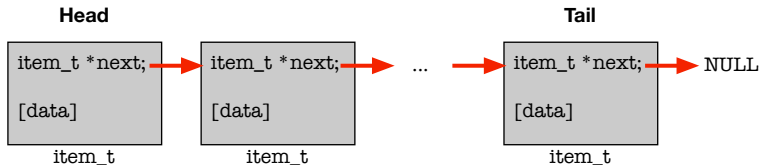
We will say more about atomics in Lecture 18.

# Singly linked lists

Serial code on Minerva: `linkedList.c`

Linked lists are a form of **container** in which each item is linked together in a **chain**:

- First item is the **head**, with a global pointer `item_t *head`.
- Last item is the **tail**, with `next=NULL`.



Note this is a *singly* linked list - a **doubly** linked list has arrows 'both ways,' i.e. a field `item_t *prev;`.

# List storage

For today's example, the data for each item is just a single integer.

To link into a list, convenient to use a struct in C:

```
1 typedef struct item {  
2     int data;  
3     struct item *next; // Next in the list  
4 } item_t;  
5  
6 item_t *head = NULL; // First item
```

New items are added using addToList:

```
1 for( i=0; i<numAdd; i++ )  
2     addToList( i );
```

# Implementation of addToList()

```
1 void addToList( int data )
2 {
3     item_t *newItem = (item_t*) malloc(sizeof(item_t));
4     newItem->data = data;
5     newItem->next = NULL;
6
7     // Find tail and add newItem to it.
8     if( head==NULL ) // i.e. list is empty.
9     {
10         head = newItem;
11     }
12     else // Find end of list and add to it.
13     {
14         item_t *tail = head;
15         while( tail->next != NULL ) tail = tail->next;
16         tail->next = newItem;
17     }
18 }
```

## Linked lists in parallel

Suppose we want to add multiple items **in parallel** for speed.

The obvious thing is to create multiple threads, and have each thread add items simultaneously.

In OpenMP, this could be achieved as follows:

```
1 #pragma omp parallel for
2 for( i=0; i<numAdd; i++ )
3     addToList( i );
```

- Multiple threads created at start of loop (*'fork'*).
- Each thread calls `addToList` some fraction of `numAdd` times.
- Check with `printList()` after the loop is complete (*'join'*).

## Failure of `addToList()` when called in parallel

The items will not be added in the same order as in serial, but this may not be a problem as long as they are *somewhere* on the list.

Some items are being **lost**, especially when many are added.

- May differ run by run, *i.e.* non-deterministic.

Also, the memory allocated for lost items is never reclaimed.

- This **memory leak** could cause problems if the application was run for a long time (e.g. server).

The implementation of `addToList()` does not work in a **multithreaded context**. We say it is not **thread safe**.



# Thread safety

A routine, library, class *etc.* is called **thread safe** if it works ‘as advertised’ in a **multithreaded context**.

If an API specification does not state whether or not a routine is thread safe, **assume that it is not!**

Note that being thread safe does **not** necessarily mean it is **efficient** in parallel.

- May have used a ‘lazy’ method to make a routine thread safe, but very slow.
- In this case may need to find an alternative that *does* scale in parallel, or develop your own solution.

# The need for synchronisation

`item_t *head` is global and therefore **shared** between threads.

This generates **data races** when adding a new item:

- ❶ Multiple threads can have '`if(head==NULL)`' evaluate as true. Only one (the last) will become the new head.
- ❷ When traversing the list, multiple threads may reach the tail at the same time. Again, only one is added.
  - That is, a thread can read '`tail->next!=NULL`' fractionally before another thread sets '`tail->next`' to its `item_t`.

If also *removing* (or 'popping') items, similar considerations would apply, although things would be more complicated.

# Critical regions

It is not easy to remove the data dependencies in this case.

- Cannot make `head` local/private (*unique to the entire list*).
- Similarly, cannot make the current **end** of the list private.

An alternative strategy is to ensure only one thread can access **critical regions** of code at a time.

- Implemented in OpenMP as `#pragma omp critical`
- Called **lock synchronisation**, for reasons that will become clear next lecture.

# #pragma omp critical

## Critical region

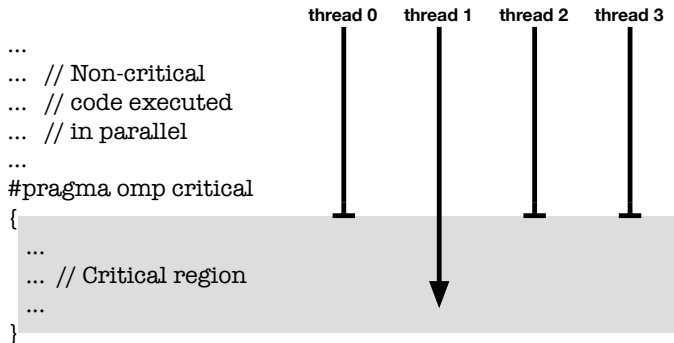
Only one thread is allowed to be in a critical region at a time.  
Until it leaves, **no other threads** are allowed to enter.

A **critical region** is defined in OpenMP as follows:

```
1 #pragma omp critical
2 {
3     ...    // Critical region
4 }
```

The critical region is defined by the **scope**, *i.e.* from '{' to '}'.

## Example for 4 threads



- Thread 1 reaches the critical region **first**.
- **No other thread** can enter until it leaves.
- **Exactly one** thread may then enter.

# Performance

There can be a significant performance penalty for critical regions:

- ① Need some mechanism to **synchronise** the threads on entering and leaving the region [*cf. Lecture 7*].
- ② Threads 'blocked' at the start will be idle. This leads to poor **load balancing** [*cf. Lecture 13*].
- ③ The scheduler may **suspend** idle threads in favour of another (not necessarily yours!). Suspension and restart incur penalties.

# Serialisation

Since only one thread can be in a critical region at any time, the critical code is **executed in serial**.

This is known as **serialisation**.

**Amdahl's law** and the **Gustafson-Barsis law** from Lecture 4:

- Maximum speed-up  $S$  in terms of the **serial fraction**  $f$ .
- By serialising regions of code we are **increasing** the value of  $f$ .
- The maximum speed-up  $S$  is **reduced**, *especially* for Amdahl's law (*i.e.* strong scaling), which predicts  $S \leq 1/f$ .

It is therefore important to restrict the **number** and **size** of critical regions to ensure reasonable parallel performance.

## First attempt: Serialise calls to addToList()

```
1 #pragma omp parallel for
2 for( i=0; i<numAdd; i++ )
3     #pragma omp critical
4     {
5         addToList( i );
6     }
```

This works, but parallel performance is poor.

- Essentially the whole loop has been serialised.
- Would be better off leaving it in serial, *i.e.*

```
1 for( i=0; i<numAdd; i++ )
2     addToList( i );
```



## Attempt 2: Serialise list traversal

Note that the start of `addToList()` has no data dependencies:

```
1 item_t *newItem = (item_t*) malloc(sizeof(item_t));  
2 newItem->data = data;  
3 newItem->next = NULL;
```

- `newItem` is local/private to each thread.
- Each thread will create its own item **independently of other threads**.
- Each value of loop counter `i` will be mapped one-to-one to a value of `data`.

This is the behaviour we want! (so far...)

The data dependencies in the remainder of addToList() can be removed by placing this portion in a critical region:

```
1 #pragma omp critical
2 {
3     if( head==NULL )
4     {
5         head = newItem;
6     }
7     else
8     {
9         item_t *tail = head;
10        while( tail->next != NULL ) tail = tail->next;
11        tail->next = newItem;
12    }
13 }
```

Performance *slightly* improved compared to the previous attempt.

# Making routines thread safe

Note the strategy followed for making `addToList()` **thread safe**:

- ① **Identify** data dependencies.
- ② Enclose in **critical regions**.
- ③ **Reduce** size and/or number of critical regions until required performance achieved.
- ④ If further scaling benefits required, may need to rethink the algorithm completely.

# Atomics

Often the data dependency is only a **single arithmetic operation**.

For instance, counting the number of items in an array of size  $n$  that obey some condition:

```
1 int count = 0;
2 for( i=0; i<n; i++ )
3 {
4     ...
5     if( condition[i] ) count++;
6 }
```

The command `count++` is a **data race**:

- Two threads may read the **old** value of `count` simultaneously.
- New count may not be the old value +2 [*cf. Lecture 5*].

## Critical region

Using a critical region *will* work ...

```
1 int count = 0;
2 #pragma omp parallel for
3 for( i=0; i<n; i++ )
4 {
5     ...
6     if( condition[i] )
7     #pragma omp critical
8     {
9         count++;
10    }
11 }
```

...but has the usual overheads of a critical region.

# Atomic instructions

Because this is a common situation, most compilers/hardware can perform the necessary synchronisation efficiently.

- **Very low overhead.**

These are known as **atomic instructions**.

Can think of atomics as a special type of critical region specialised to single arithmetic operations that exploits hardware primitives.

GPUs also support atomic instructions; we will look at these more closely in Lecture 18.

# Atomics in OpenMP

Atomics are implemented in OpenMP as follows:

```
1 int count = 0;
2 #pragma omp parallel for
3 for( i=0; i<n; i++ )
4 {
5     ...
6     if( condition[i] )
7     #pragma omp atomic
8         count++;
9 }
```

Note there is no scope ('{'...'}') after `#pragma omp atomic`.

- Only works on single instructions.

## Summary and next lecture

In this lecture we have looked at using **critical regions**:

- Can avoid data races by **serialising** blocks of code.
- Corresponding performance loss.
- Reduce overhead for single arithmetic instructions by using **atomics**.

Next time will look in more detail at how this synchronisation is achieved.