University of Leeds

Lecture 6: Critical regions and a

Previous lecture

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

-
- Leads to **non-deterministic** beha
- Can arise in loops as **data depend**
- Often possible to remove these dependenc the expense of increased **parallel overheads**.

This lecture

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

- **ing**

- Briefly mentioned in Lecture 4 in the context of **fork-join**.

Now we will focus on using synchronisation to avoi

- Define **critical regions** which can on thread at a time.
- **Atomics**: Critical regions specialised to single operations.

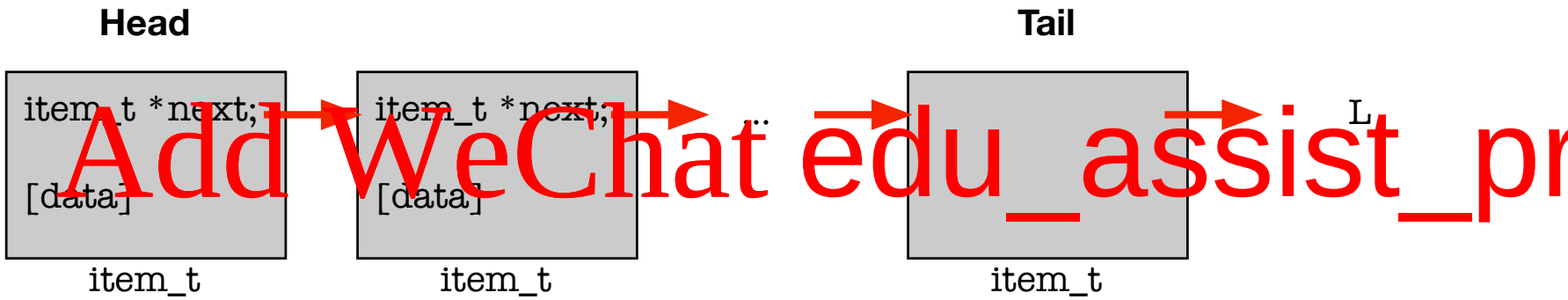
We will say more about atomics in Lecture 18.

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**:

- _ ead.
- https://eduassistpro.github.i



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:

```
typed
  in
  st
} item_t;

item_t *head = NULL; // First item
```


New items are added using `addToList`:

```
for( i=0; i<numAdd; i++ )
  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    ne
6
7    //Fi
8    if  h
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 thre

In Op

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


- Multiple threads created at start of loop (‘
- 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.

Som

Also, the memory allocated for lost items is never re

- This **memory leak** could cause prob
  was run for a long time (e.g. serv

The implementation of `addToList()` does not work in a **mulithreaded 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 A

Note that being thread safe does **not** **efficient** in parallel.

- May have used a 'lazy' method to make a routi
  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 g

1.
2. When traversing the list, multiple threads may reach the tail at the same time. Again, only one is added.
3. A thread can read `tail->next` ! another thread sets '`tail->next`'

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.

-
-

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

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

#pragma omp critical

Critical region

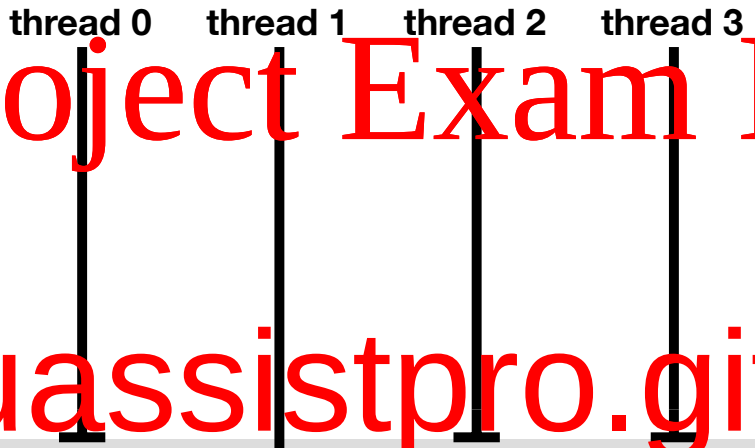
O

A **critical region** is defined in OpenMP as foll


```
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




```
...
... // Non-critical
... // code executed
...
...
... // Critical region
...
}
```

- 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.
2. **load balancing** *[cf. Lecture 13]*.
3. The scheduler may **suspend** idle th (not necessarily yours!). Suspension and r penalties.

Serialisation

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

Am

- Maximum speed-up S in terms of the
- By serialising regions of code we are f.
- The maximum speed-up S is **red** l'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
6   }
```


This works, but parallel performance is poor.

- Essentially the whole loop has been serialis
- Would be better off leaving it in serial,


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

Attempt 2: Serialise list traversal

Note that the start of addToList() has no data dependences

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


- Each thread will create its own item **independently of other threads**.
- Each value of loop counter i will b 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   h
4    {
5
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 t **d safe**:

1.
2.
3. **Reduce** size and/or number of critical re
performance achieved.
4.
If further scaling benefits required, may nee
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 o


```
1 int coun 0
2 for( i=0
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  #prag
3  for( i=0
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.

Can think of atomics as a special type of critical regio
to single arithmetic operations that exploits har

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 #prag
3 for( i=0
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

Today we have looked at using **critical regions**:

-
-
-

atomics.

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