XJCO3221 Parallel Computation

Peter Jimack

University of Leeds

Lecture 5: Loop parallelism and data races

Previous lectures

In Lecture 3 we saw how two problems could be parallelised:

1. **Vector addition**, where two one-dimensional arrays were added together element-by-element.
2. **Mandelbrot set**, where pixel colours were calculated to generate a 2D image.

However, neither of these problems have any **data dependencies**.

1. Each vector element was calculated **independently** of other elements.
2. Each pixel colour was calculated **independently** of the others.

Today's lecture

Today we will look how to parallelise problems that **do** have data dependencies.

- Can lead to **data races** on shared memory systems.
- Behaviour then becomes **non-deterministic**.
- May require algorithm re-design to remove dependencies.

We will then look at three examples of **loop parallel** problems and how their dependencies can be resolved.

Example of a data race

Consider the following pseudecode[1] for two concurrent threads, where each thread accesses the same variable x. x=0 at the start of the code segment.

Thread 0:

```
a = x;
a += 1.0;
x = a;
```

Thread 1:

```
b = x;
b += 2.0;
x = b;
```

What value does x take at the end?

[1]From §2.6 of McCool *et al.*, *Structured parallel programming* (Morgan-Kaufman, 2012).

Non-determinism

The result may differ each time the program is executed.

- An example of **non-determinism** *[Lecture 3]*.

The scheduler runs threads depending on various factors, including other applications and OS tasks.

- Cannot predict which thread is **launched** first.
- The OS may **suspend** threads to make way for other tasks *(pre-emptive multitasking)*.
- The instructions may become **interleaved**.

Interleaved instructions (example)

Recall x=0 initially.

```
a  = x;    // Thread 0: a now 0
a += 1;    // Thread 0: a now 1
b  = x;    // Thread 1: b now 0
x  = a;    // Thread 0: x now 1
b += 2;    // Thread 1: b now 2
x  = b;    // Thread 1: x now 2
```

In this example, x=2 at the end.

- Possible to get x=1 or x=3 by different interleaving of instructions *(check left as exercise)*.

Race conditions

This is known as a **data race** or a **race condition**.

- Result of calculation depends on which thread reaches its instructions first.
- **Non-deterministic**, which is usually undesirable.

Only an issue for **shared memory**.

- If each thread had its own `x`, there would be no race.
- In this example, if each thread had its own `x`, `x`=1 for thread 0 and `x`=2 for thread 1 at the end, regardless of any interleaving.

Read-only does not lead to a data race

For a race condition to arise, **at least one** thread must **write** to `x`.

- No race if all threads just **read** `x`.

There is no race for the following example, as both threads only read `x`:

Thread 0:

```
a = x;
a += 1.0;
```

Thread 1:

```
b = x;
b += 2.0;
```

For this example, `x`=0 (and `a`=1 and `b`=2) at the end.

Sequential consistency

Have assumed each thread executes its instructions in order.

- *i.e.* have assumed **sequential consistency**.

Modern compilers often rearrange instructions to improve performance.

- *e.g.* bring forward memory accesses, combine operations *etc.*
- The result is the same **in serial**.
- However, **multithreading** can confuse compilers.
- Can lead to unexpected results!

The `volatile` keyword

You may read that the way to solve this is to declare variables as `volatile` (in C/C++). However, this is only partially correct[1].

- `volatile` is for **special memory**, such as that read/written by external device *(memory-mapped I/O)*.
- The way compilers handle such variables is **not guaranteed** to work for multithreading.

If this might be an issue, should use features **specific** to concurrent programming.

- *e.g.* memory fences, `std::atomic<>` in C++11 *etc.*
- Will come to atomics next lecture and Lecture 18.

[1]S. Meyers, *Effective modern C++* (O'Reilly, 2015).

Loop parallelism

Often we are required to parallelise **loops**.

- Known as **loop parallelism**.
- If there are **data dependencies**, may have implicit **data races** within the loop.

There is no **systematic** way to parallelise loops.

- How to remove a dependency depends on context.
- Consider extra resources or loops to reach the correct solution.

For the remainder of this lecture, will give examples of loops with data dependencies, and how to overcome them.

Example 1: Redundant variable

Consider the following serial code:

```
float temp, a[n], b[n], c[n];
...   // Initialise arrays b and c

int i;
for( i=0; i<n; i++ )
{
  temp = 0.5f*( b[i] + c[i] );
  a[i] = temp;
}
```

Here, `temp` is being used as a temporary variable.

- Sometimes useful to make (more complex) code easier to read.

Need to make temp a private (or local) variable:

```
#pragma omp parallel for
for( i=0; i<n; i++ )
{
  float temp = 0.5f*( b[i] + c[i] );
  a[i] = temp;
}
```

Can also use OpenMP's private clause:

```
#pragma omp parallel for private(temp)
for( i=0; i<n; i++ )
{
  temp = 0.5f*( b[i] + c[i] );
  a[i] = temp;
}
```

cf. the inner loop counter in Lecture 3's Mandelbrot set example.

Example 2: Shift dependency
Code on Minerva: `shiftDependency.c`

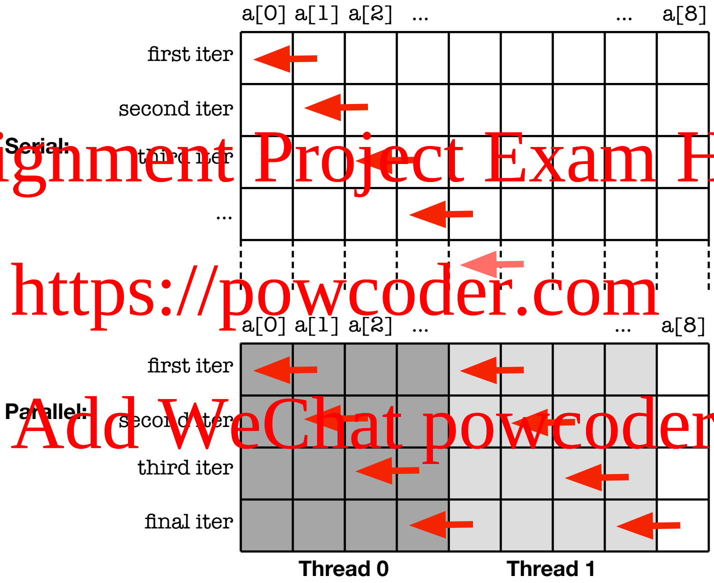
Consider a shift dependency:

```c
float a[n];
... // initialise array a

int i;
for( i=0; i<n-1; i++ )
  a[i] = a[i+1];
```

Naive parallelisation does not *quite* work:

```c
#pragma omp parallel for
for( i=0; i<n-1; i++ )
  a[i] = a[i+1];
```



a[0] a[1] a[2] ... ... a[8]
first iter
second iter
Serial:
third iter
...
a[0] a[1] a[2] ... ... a[8]
first iter
Parallel:
second iter
third iter
final iter
Thread 0
Thread 1

A solution here is to **copy** the array a **before** the loop:

```
float atemp[n];

#pragma omp parallel for
for( i=1; i<n; i++ )
  atemp[i] = a[i];

#pragma omp parallel for
for( i=0; i<n-1; i++ )
  a[i] = atemp[i+1];
```

This comes at the expense of **additional resources**:

- **Memory** for the array atemp.
- **CPU time** to copy a to atemp.

Examples of **parallel overheads**.

Example 3: Red-black Gauss-Seidel

Code on Minerva: `redBlackGaussSeidel.c`

Finally consider this:

```
for( i=1; i<n-1; i++ )
  a[i] = 0.5f * ( a[i+1] + a[i-1] );
```

Each element takes the average of the elements either side of it.

- Can be used to **smooth** vector a, *e.g.* in image transformations ('blurring') with a 2D nested loop over pixels.
- Also arises in numerical computation - the **diffusion** or **heat equation** solved using the **Gauss-Seidel** method.

We **could** make a copy `atemp` as before.

- In numerical computation, this is the **Jacobi method**.

However, this is undesirable in some situations:

- Typically repeat the loop until 'reaching' the solution, which is faster (*takes fewer iterations*) when not using a copy.

Instead we consider a **modified serial** variant which is more amenable to parallelisation.

- Known as **red-black Gauss-Seidel**, as it bears a resemblance to red and black squares on a chessboard (in 2D).

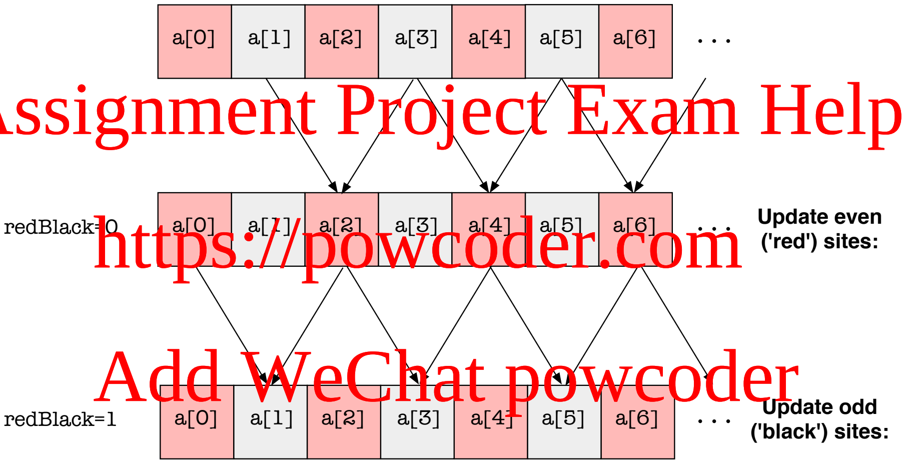
Update **even** elements first, then **odd** elements[1]:

```c
int redBlack;
for( redBlack=0; redBlack<2; redBlack++ )
  for( i=1; i<n-1; i++ )
    if( i%2 == redBlack )
    {
      a[i] = 0.5f * ( a[i-1] + a[i+1] );
    }
```

The outer `redBlack` loop only loops over 2 values, 0 or 1.

- When `redBlack=0`, only the elements of `a[i]` with `i` **even** are updated.
- Similarly, only the **odd** are updated when `redBlack=1`.

[1]Recall `i%2` gives the remainder of division by 2, so *e.g.* `i%2==0` if `i` is even.


a[0]
a[1]
a[2]
a[3]
a[4]
a[5]
a[6]
...
redBlack=0
Update even ('red') sites:
redBlack=1
Update odd ('black') sites:

Note that for each loop, the calculations are **now independent**.

- We have **removed the dependency**, albeit by slightly changing the serial algorithm.

Now clear how to parallelise:

```
for( redBlack=0; redBlack<2; redBlack++ )
  #pragma omp parallel for
  for( i=1; i<n-1; i++ )
    if( i%2 == redBlack )
    {
      a[i] = 0.5f * ( a[i-1] + a[i+1] );
    }
```

There are no dependencies **within** the `i`-loop, because `a[i-1]` and `a[i+1]` were/will be updated in the other `redBlack` loop.

The ‘best’ parallel algorithm?

Notice that we changed Gauss-Seidel to the red-black variant to make it more efficient in parallel.

- For Gauss-Seidel, this is the standard parallel variant.

Perfectly acceptable to do this if:

1. The solution reached is still ‘correct.’
2. Parallel scaling is good (for large arrays).

As a general observation, the ‘best’ parallel algorithm need **not** be directly related to the ‘best’ serial algorithm.

Summary and next lecture

Today we have seen how multiple threads with at least one writing to shared memory can lead to **data races**.

- Outcome is not predictable *(non-deterministic)*.
- Can be a challenge to remove **data dependencies** from loops.
- May require additional resources not present in the serial version, *i.e.* **parallel overheads**.

Next time we will look at a way to **synchronise** threads within a parallel program and apply it to a linked list.