University of Leeds

Lecture 3: Data parallel pro

Previous lectures

In the l

para

- Separate processing units (cores) share some levels of **memory cache**.
- Various frameworks for programming S
- Widely-implemented standard: **O**

Today's lecture

Today we are going to look at a some actual problems.

- Examples of a **data parallel** problems, where the same
-
-
- How to parallelise **nested loops**.
- Parallel code can be **non-determ**
  code is **deterministic**.

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

Vector addition

An **n-vector a** can be thought of as an **array** of n numbers: $\mathbf{a} = (a_1, a_2, \ldots, a_n)$.

If two v

gene

$$\begin{array}{ccccccccc} & & & 1 & & 2 & & 3 & & & n \\ & + & & + & & + & & + & & + & \\ \mathbf{b} = (& b_1, & & b_2, & & b_3, & & \ldots, & & b_n &) \\ & \downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow & \\ \mathbf{c} = (& c_1, & & c_2, & & c_3 & & \ldots, & & c_n &) \end{array}$$

Or:

$$c_i = a_i + b_i \quad , \quad i = 1 \ldots n.$$

Serial vector addition

Code on Minerva: `vectorAddition_serial.c`


```c
#define n 100

int main()
{
  flo

  ...  // in

  int i;
  for( i=0; i<n; i++ )
    c[i] = a[i] + b[i];

  return 0;
}
```


Note that indices usually start at 0 for most languages, but 1 for the usual mathematical notation (also FORTRAN, MATLAB).

Vector addition in parallel

Code on Minerva: `vectorAddition_parallel.c`

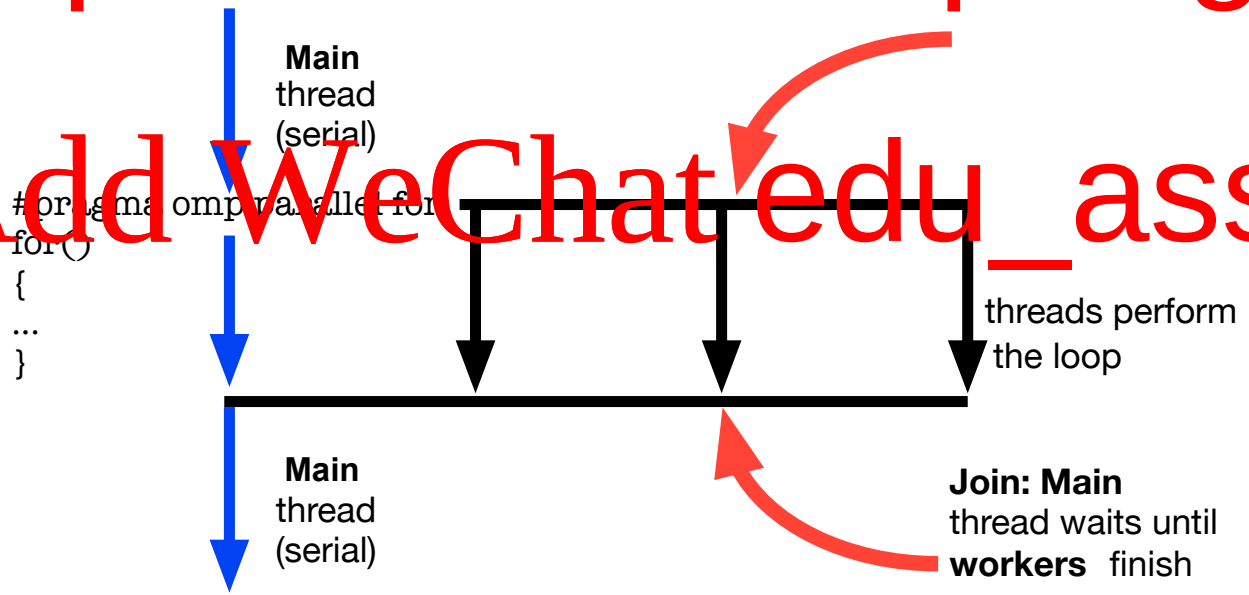
Add `#pragma omp parallel for` just before the loop


```
1  #define n 100
2
3  int main
4  {
5    flo
6
7    ...  // Initialise a[n] and b[n]
8
9    int i;
10   #pragma omp parallel for
11   for( i=0; i<n; i++ )
12     c[i] = a[i] + b[i];
13
14   return 0;
15 }
```


This only parallelises this one loop, not any later ones!

Fork-and-join

- When the executable reaches `#pragma omp parallel for`, it spawns multiple **threads**.
  - Each thread computes **part** of the loop.
- This is



Example: Four threads in total

Pseudocode for the **main** thread:

```
// Main thread starts in serial
// Initialise arrays a, b; allocate c.
...
// REACH
// FORK: Cr
worker    f
worker    f
worker    f

// Perform 1/4 of the total loop.
for( i=0; i<n/4; i++ )
  c[i] = a[i] + b[i];

// JOIN: Wait for other threads to finish.
worker1.join();
worker2.join();
worker3.join();

// Continue in serial after the loop
```

Worker thread 1:

```
1 // CREATED BY MAIN ('fork')
2 // Perform second 1/4 of loop
3 for( i=n/4; i<n/2; i++ ) c[i] = a[i] + b[i];
4 // FINISH ('join')
```

Wor

```
1 // CREAT
2 // Perfo
3 for( i=n/2; i<3*n/4; i++ ) c[i] = a[i] + b[i];
4 // FINISH ('join')
```

Worker thread 3:

```
1 // CREATED BY MAIN ('fork')
2 // Perform final 1/4 of loop.
3 for( i=3*n/4; i<n; i++ ) c[i] = a[i] + b[i];
4 // FINISH ('join')
```

Notes

- The four threads are **not** being executed one after the other.
- Each thread runs **concurrently**, hopefully on separate cores,

Each thread performs the **same** opera

- Would be **SIMD** in Flynn's taxonom
  implemented **in software** on a MIM

Have assumed n is divisible by the number of threads for clarity.

- Generalising to arbitrary n is not difficult, but obscures the parallel aspects.

#pragma omp parallel for

The total loop range was evenly divided between all threads.

- ched.
- start

- The start, end and stride must be
- Cannot break from the loop.
- **Cannot apply to 'while...do' or 'do...whil**

Data parallel and embarrassingly parallel

This is an example of a **data parallel problem** or a **map**.

-
-
-

In fact, this example is so straightforward to parall
sometimes referred to as an **embarras**

- Easy to get working **correctly** in parallel.
- May still be a challenge to achieve good parallel **performance**.

Mandelbrot set generator

Code on Minerva: `Mandelbrot.c`, `makefile`

Classic computationally intensive problem in two dimensions that used to be used as a **benchmark** for processor speeds:

-
  double loop.
- Colour of each pixel calculated **independently** of all other pixels.
- Each colour calculation requires many **floating point operations**.

Code snippet

The part of the code that interests us here is shown below:

```
1 // Change the colour arrays for the whole image.
2 int i, j;
3 for( j=0  j
4   for
5   {
6
7     setPixelColour( i, j );
8   }
```


Note the `i`-loop is nested inside the `j`-

The graphical output is performed in OpenGL/GLFW. Since including and linking is different between Linux and Macs, a simple `makefile` has been provided.

What `setPixelColour` does

Purely for background interest, here's how the colours are calculated:

1. Each **pixel** `i,j` is converted to **floating point numbers** c_x,
2.
3. $x \quad y$ 4, or a maximum number of iterations `m`

$$(z_x, z_y) \to (z_x^2 - z_y^2 +$$

4. The colour is selected based on the number of iterations.

[1] More concisely represented as **complex numbers** c and z [with *e.g.* $z_x = \Re(z)$], then the iteration is just $z \to z^2 + c$.

Parallel Mandelbrot: First attempt

Parallelise **only** the **inner** loop.

```
1 int i, j;
2 for( j=0; j<numPixels_y; j++ )
3   #pr
4   for
5   {
6
7   }
```


This works, but may be slower than serial .

Multiple possibilities for this:

- The **fork-join** is **inside** the `j`-loop, so threads are created and destroyed `numPixels_y` times, which incurs an **overhead**.
- This problem suffers from poor **load balancing**; see later.

Parallel Mandelbrot: Second attempt

Parallelise only the **outer** loop, so there is only a single **fork** event and a single **join** event.

```
1 int i, j;
2 #pragm   o
3 for( j=0; j
4   for
5   {
6     setPixelColour( i, j );
7   }
```


This is faster ... but **wrong**!

- A **distorted** image results.
- The distortion is **different each time** the program is executed.

The **same** variable `i` for the inner loop counter is being **updated by all threads**:

- When one thread completes a calculation, it increments `i`.
- Therefore other threads will skip at least one pixel.

Parallel Mandelbrot: Third attempt

Make the inner loop variable `i` **private** to each thread:

```
int j;
#pragma omp parallel for
for( j=0; j<numPixels_y; j++ )
{
  int i
  for
  {

  }
}
```


. . . or (although strictly this is C++):

```
#pragma omp parallel for
for( int j=0; j<numPixels_y; j++ )
  for( int i=0; i<numPixels_x; i++ )
  {
    setPixelColour( i, j );
  }
```

The private clause

A third way to solve this is to use OpenMP's `private` **clause**:

```
1 int i, j;
2 #pragma omp parallel for private(i)
3 for( j=0  j
4   for
5   {
6
7   }
```


- Creates a **copy** of `i` for each thread.
- Multiple variables may be listed,

The code now works . . . but is no faster for more than 2 threads!

- The primary overhead is poor **load balancing**. We will look at this next lecture briefly, and detail in Lecture 13.

Determinism and non-determinism

Notice that the incorrect images were slightly different each time.

e.g. 1. *e.*

The pixels plotted depend on the order in which the
the shared variable `i`, which depends on the thread **scheduler**.

- Will be influenced by factors outside our control.
- *e.g.* the various **background tasks** that every OS must run.

Our serial code was **deterministic**, *i.e.* produced the same results each time it was run.

By co **inistic**.

Ofte

- Some algorithms, often in science and engi about non-deterministic errors **as I**
- Strictly imposing determinism may resul overheads and performance loss.

However, for this module we will try to develop parallel algorithms whose results match that of the serial equivalent.

Summary and next lecture

Today we have looked at **data parallel** problems or **maps**, where the sa

- https://eduassistpro.github.i

Next week we will start looking at more complex pr
which the calculations on different threads are .

Before then, we need to learn the vocabulary of parallel theory, which is the topic of next lecture.