COMP3221 Parallel Computation

David Head

University of Leeds

Lecture 3: Data parallel problems

Previous lectures

In the last lecture we started looking at **shared memory parallelism** (SMP):

- Relevant to **multi-core CPUs**.
- Separate processing units (cores) share some levels of **memory cache**.
- Various frameworks for programming SMP systems.
- Widely-implemented standard: **OpenMP**.

Today's lecture

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

- Examples of a **data parallel** problems, where the same operation is applied to multiple data elements.
- Also known as a **map**[1].
- **Multi-threading** solution employs a **fork-join** pattern.
- How to parallelise **nested loops**.
- Parallel code can be **non-deterministic**, even when the serial 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 vectors **a** and **b** are the same size, they can be added to generate a new *n*-vector **c**:

$$\begin{array}{lcccccl}
\mathbf{a} = (& a_1, & a_2, & a_3, & \ldots, & a_n &) \\
+ & + & + & + & & + & \\
\mathbf{b} = (& b_1, & b_2, & b_3, & \ldots & b_n &) \\
\downarrow & \downarrow & \downarrow & \downarrow & & \downarrow & \\
\mathbf{c} = (& c_1, & c_2, & c_3 & \ldots, & c_n &)
\end{array}$$

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()
{
  float a[n], b[n], c[n];

  ...  // Initialise a[n] and b[n]

  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:

```c
#define n 100

int main()
{
  float a[n], b[n], c[n];

  ...  // Initialise a[n] and b[n]

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

  return 0;
}
```

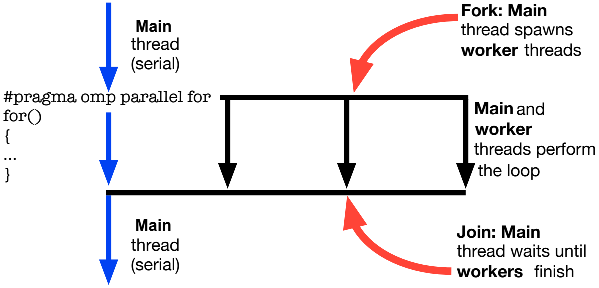
This only parallelises this one loop, not any later ones!

Fork-and-join

When the executable reaches `#pragma omp parallel for`, it spawns multiple **threads**.

- Each thread computes **part** of the loop.
- The extra threads are **destroyed** at the end of the loop.

This is known as a **fork-join** construct:


Main thread (serial)
Fork: Main thread spawns worker threads
#pragma omp parallel for
for()
{
...
}
Main and worker threads perform the loop
Main thread (serial)
Join: Main thread waits until workers finish

Example: Four threads in total

Pseudocode for the **main** thread:

```
// Main thread starts in serial
// Initialise arrays a, b; allocate c.
...
// REACHES #pragma omp parallel for
// FORK: Create three new threads.
worker1 = fork(...);
worker2 = fork(...);
worker3 = fork(...);

// 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:

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

Worker thread 2:

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

Worker thread 3:

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

Notes

The four threads are **not** being executed one after the other:

- Each thread runs **concurrently**, hopefully on separate cores, *i.e.* in **parallel**.
- Cannot be understood in terms of serial programming concepts.

Each thread performs the **same** operations on **different** data.

- Would be **SIMD** in Flynn's taxonomy, except this is implemented **in software** on a MIMD device.

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.

- Happens as soon as `#pragma omp parallel for` reached.
- The **trip count** (*i.e.* loop range) **must** be known at the **start** of the loop.
- The start, end and stride must be **constant**.
- Cannot `break` from the loop.
- **Cannot apply to 'while. . . do' or 'do. . . while' loops**

Data parallel and embarrassingly parallel

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

- Array elements distributed evenly over the threads.
- Same operation performed on all elements.
- Suitable for the SIMD model.

In fact, this example is so straightforward to parallelise that is also sometimes referred to as an **embarrassingly parallel problem**.

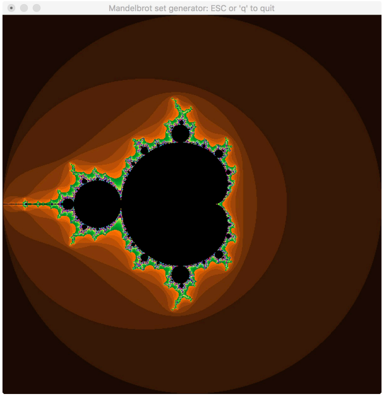
- Easy to get working **correctly** in parallel.
- May still be a challenge to achieve good parallel **performance**.

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:

- Loops over **pixels**, *i.e.* a **two dimensional**, **nested** 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:

```
// Change the colour arrays for the whole image.
int i, j;
for( j=0; j<numPixels_y; j++ )
  for( i=0; i<numPixels_x; i++ )
  {
    // Set the colour of pixel (i,j), i.e. modify the values
     of red[i][j], green[i][j], and/or blue[i][j].
    setPixelColour( i, j );
  }
```

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

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, c_y, both in the range -2 to 2.
2. Two other floats z_x and z_y are initialised to zero.
3. The following iteration[1] is performed until $z_x^2 + z_y^2 \geq 4$, or a maximum number of iterations `maxIters` is reached:

$$(z_x, z_y) \rightarrow (z_x^2 - z_y^2 + c_x \, , \, 2z_x z_y + c_y)$$

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 $z \rightarrow z^2 + c$.

Parallel Mandelbrot: First attempt

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

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

This works, but is not much faster than serial – and may even be slower *(check on your system)*.

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.

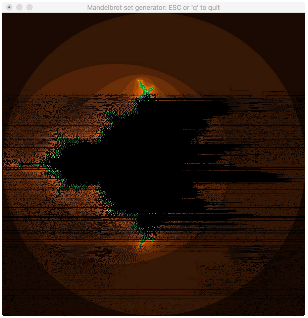
```
int i, j;
#pragma omp parallel for
for( j=0; j<numPixels_y; j++ )
  for( i=0; i<numPixels_x; i++ )
  {
    setPixelColour( i, j );
  }
```

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.
- Threads do **not** calculate the full line of pixels.



Parallel Mandelbrot: Third attempt

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

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

. . . or (for compilers following the C99 standard):

```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**:

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

- Creates a **copy** of `i` for each thread.
- Multiple variables may be listed, *e.g.* `private(i,a,b,c)`

The code now works, but is still not much faster than serial.

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

The collapse clause

The collapse clause replaces 2 or more nested loops with a single loop, at the expense of additional internal calculations.

```
#pragma omp parallel for collapse(2)
for( int j=0; j<numPixels_y; j++ )
  for( int i=0; i<numPixels_x; i++ )
    setPixelColour( i, j );
```

is equivalent to (but more readable than)

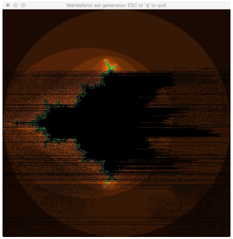
```
#pragma omp parallel for
for( int k = 0; k < numPixels_x * numPixels_y; k++ )
  {
    int
      i = k % numPixels_x,
      j = k / numPixels_x;
    setPixelColour( i, j );
  }
```

This is principally intended for **short** loops that cannot be equally distributed across all threads.

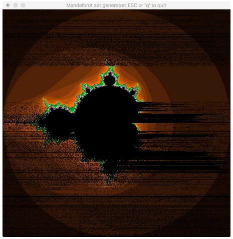
Determinism and non-determinism

Notice that the incorrect images were slightly different each time:

e.g. 1.



e.g. 2.



The pixels plotted depend on the order in which threads update 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 contrast, our (incorrect) parallel code was **non-deterministic**.

Often this is the result of an error, but can sometimes be useful:

- Some algorithms, often in science and engineering, do not care about non-deterministic errors **as long as they are small**.
- Strictly imposing determinism may result in additional 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 look at **data parallel** problems or **maps**, where the same operation is applied to multiple data members.

- Distribute data evenly across threads.
- Sometimes referred to as **embarrassingly parallel**.

In two lectures time we will start looking at more complex problems for which the calculations on different threads are **not** independent.

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