3. Shared Memory Parallel Programming

Parallel Programming

Dr Hamidreza Khaleghzadeh
School of Computing
University of Portsmouth

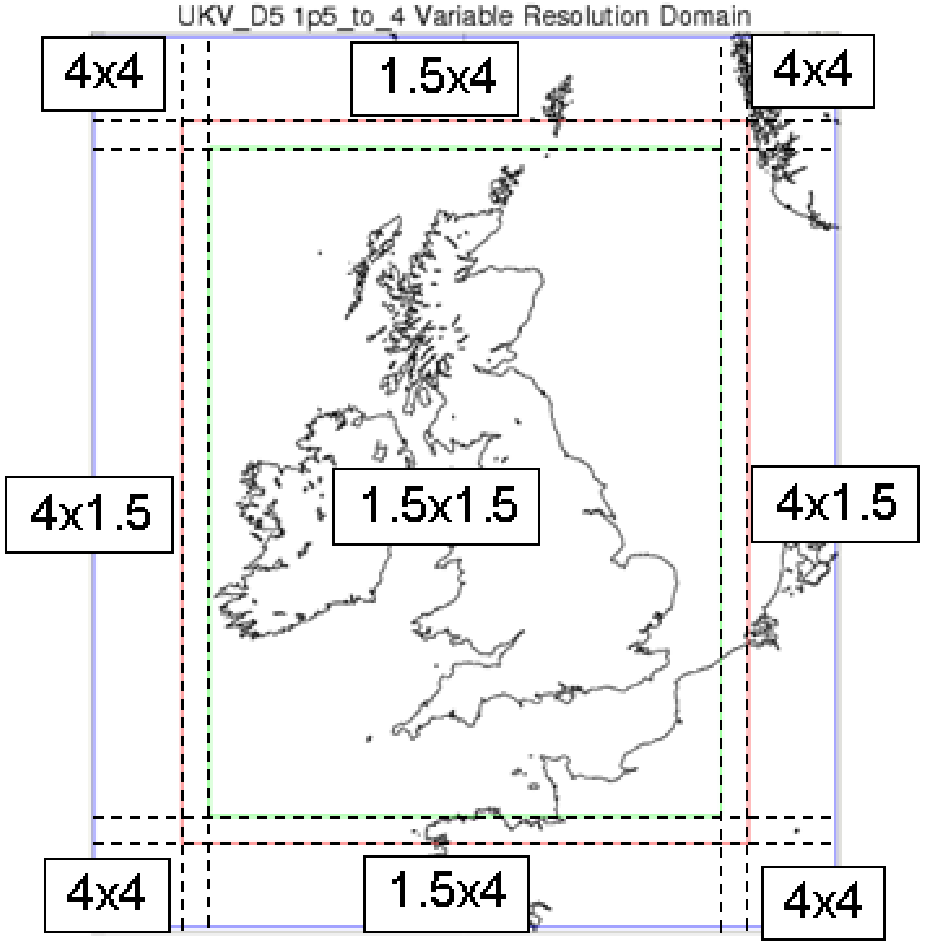
Goals

- Begin with general discussion of differences between "embarrassingly parallel" problems, and harder problems like simulations.
- In the context of the simple simulation introduced in this week's lab script, introduce a fundamental kind of synchronization for shared-memory parallel programming.
- Brief overview of OpenMP – one of the established frameworks for shared memory programming.

Embarrassing Parallelism

- The Mandelbrot set was an example of an *Embarrassingly Parallel* problem, where each point is calculated independently of every other.
- Actually such problems are quite common – general situation is where a large problem decomposes into completely independent tasks.
- But many important problems *don't* have this property.

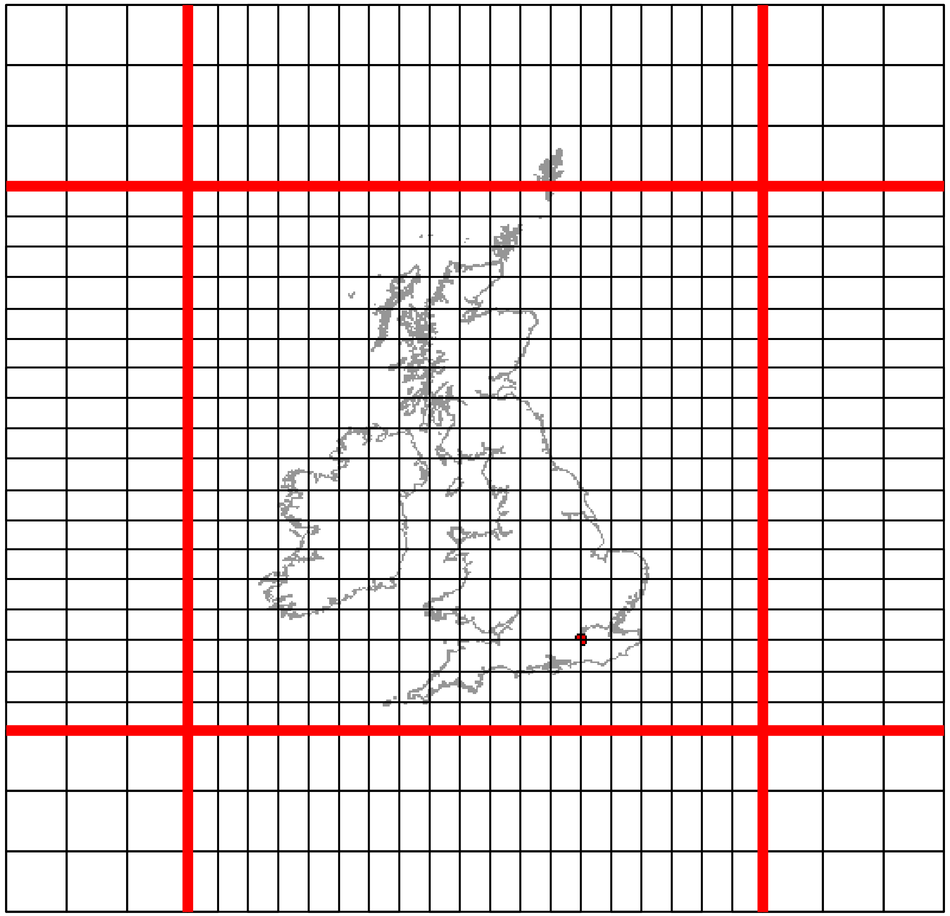
Weather Forecasting



- Size of individual cells (in kilometres) in Met Office's *UKV model*† of UK weather.

†https://www.metoffice.gov.uk/research/approach/modelling-systems/unified-model/weather-forecasting

Schematic of Grid



UKV Model

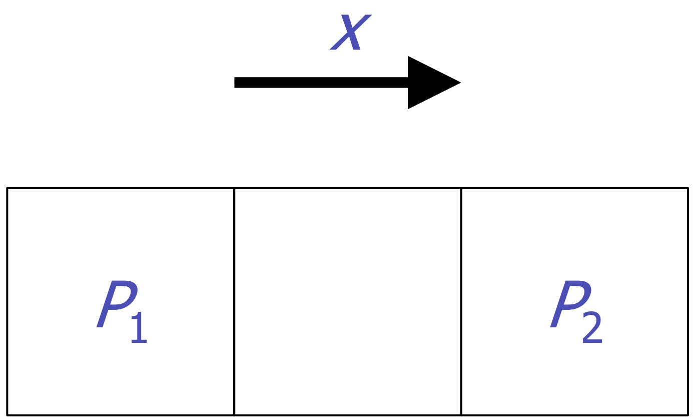
- Area of the UK is divided into 1.5km square cells horizontally, and into 70 layers vertically – a 3D grid (total 744 x 928 x 70 grid points)
- Numerical modelling techniques used:
  - Atmospheric variables (wind speed, temperature, pressure, humidity, ...) stored for each grid box.
  - Equations are solved for each grid box to predict the values at that point a short time later.
  - Repeat until reach forecast time required.
  - For UKV, time step is 50 seconds, and forecast goes out to 36 hours.

Running Weather Simulation

- Dividing the grid over processors gives opportunity for exploiting massive parallelism
  - As of 2017, Met office runs a Cray XC40 supercomputer with total 241,920 cores that was #15 in Nov 2017 TOP500.
- But not "embarrassingly parallel" – equations at each grid point will depend on atmospheric variables stored at *neighbouring* points.
  - Each element of atmosphere moved by pressure of neighbouring elements, is warmed or cooled by them, etc.

Simplified Dynamics

- Net force on volume of air in *central* cell (tending to cause air to accelerate in x-direction), is proportional to difference in air pressures in *adjacent* cells, $P_1 - P_2$.



SIMULATIONS

Simpler Models

- In this unit we can't do weather forecasting.
  - Equations for updating variables within grid points involve complex physics that is out of scope.
- But we *can* look at simpler problems that illustrate the features affecting parallelization.
- The Conway's Game of Life example in this week's lab already incorporated some of these
  - Update of an individual cell according to some local rule, which however depends of values of neighbours.

"Life"

- Cellular automaton on 2D grid, where each cell takes value 0 or 1.
- In a time step, new value of cell depends on it's old value and old values of it's neighbouring cells.
- For current purposes, close enough analogy with large scale simulations on grids (even 3D) where local update depends on neighbours.

Life Animation†

†From https://www.conwaylife.com/wiki/Conway%27s_Game_of_Life

Update Rule for Cell

Sum of neighbours = 3

1	0	0
0	1	1
0	0	1

⇒

	1	

1	0	0
0	0	1
0	0	1

⇒

	0	

Stay the same!

Sum of neighbours = 2

0	1	0
0	1	0
1	0	0

⇒

	1	

0	1	0
0	0	0
1	0	0

⇒

	1	

Live!

Sum of neighbours anything else

1	0	0
0	1	1
1	1	0

⇒

	0	

1	0	0
0	0	1
1	1	0

⇒

	0	

Die!

13

Sequential Life

- Pseudocode – note two phases, “sum” and “update”:

```
while(true) {
    for(int i = 0 ; i < N ; i++)
        for(int j = 0 ; j < N ; j++)
            sums [i] [j] = sum of cells values neighbouring (i, j) ;
    for(int i = 0 ; i < N ; i++)
        for(int j = 0 ; j < N ; j++)
            cells [i] [j] = update(cells [i] [j], sums [i] [j]) ;
}
```

- Main array is cells; auxiliary array sums holds sum of cell elements neighbouring (i, j)

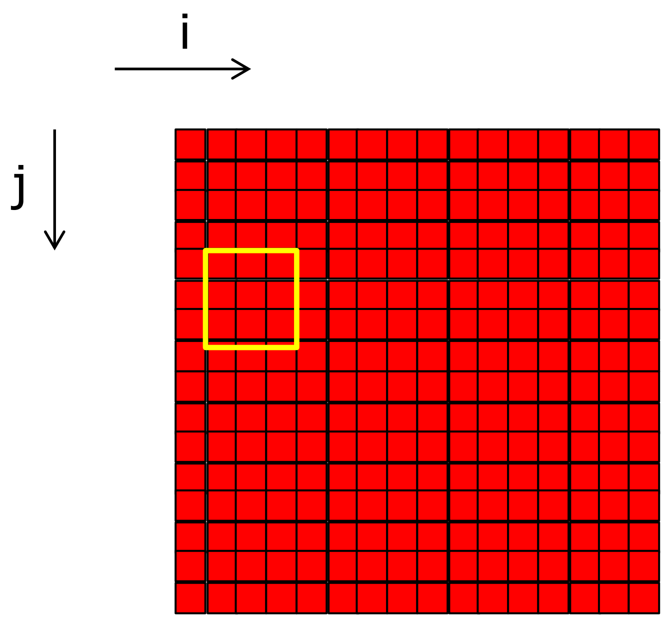
Aside

- Why do we need *two* sets of for loops here – why couldn't we have, say?:

```
while(true) {
   for(int i = 0 ; i < N ; i++)
      for(int j = 0 ; j < N ; j++) {
         sum = sum of cells values neighbouring (i, j) ;
         cells [i] [j] = update(cells [i] [j], sum) ;
      }
}
```

Updating cells array in iteration t

- Old cell value – after iteration t - 1
- New cell value – after iteration t



- Iterate through array, updating cells.
- With code on previous slide, sum for next (yellow) cell contains 4 green values and 4 red values.
- Wrong! Update of cell in iteration t should only depend on values of cells in *previous* generation.

Parallel Attempt

```
Class SumThread {
   void run() {
      for(int i = begin ; i < end ; i++)
         for(int j = 0 ; j < N ; j++)
            sums [i] [j] = sum of cells values neighbouring (i, j) ;
   }
}
Class UpdateThread {
   void run() {
      for(int i = begin ; i < end ; i++)
         for(int j = 0 ; j < N ; j++)
            cells [i] [j] = update(cells [i] [j], sums [i] [j]) ;
   }
}
```

Parallel Attempt (continued)

- Main program:

```
while(true) {
    create P instances of  SumThread, start them, and wait
                for completion ;
    create P instances of  UpdateThread, start them, and
                wait for completion ;
}
```

- On previous slide we left *begin* and *end* undefined, but may assume a *block-wise decomposition* of i loops.
  - See previous lecture.

Comments

- The approach given over the previous two slides is clearly correct, and follows approaches taken in labs for parallelizing (i, j) loops in Mandelbrot Set.
- *But*, the *overhead of creating new threads* in every iteration of the main loop is likely to lead to *poor performance*
    - Inner (i, j) loops here typically smaller and contain much less work than Mandelbrot set.

Second Attempt

```
Class LifeThread {
   void run() {

      while(true) {
         for(int i = begin ; i < end ; i++)
            for(int j = 0 ; j < N ; j++)
               sums [i] [j] = sum of cells values neighbouring (i, j) ;
         for(int i = begin ; i < end ; i++)
            for(int j = 0 ; j < N ; j++)
                cells [i] [j] = update(cells [i] [j], sums [i] [j]) ;
      }
   }
}
```

Second Attempt (continued)

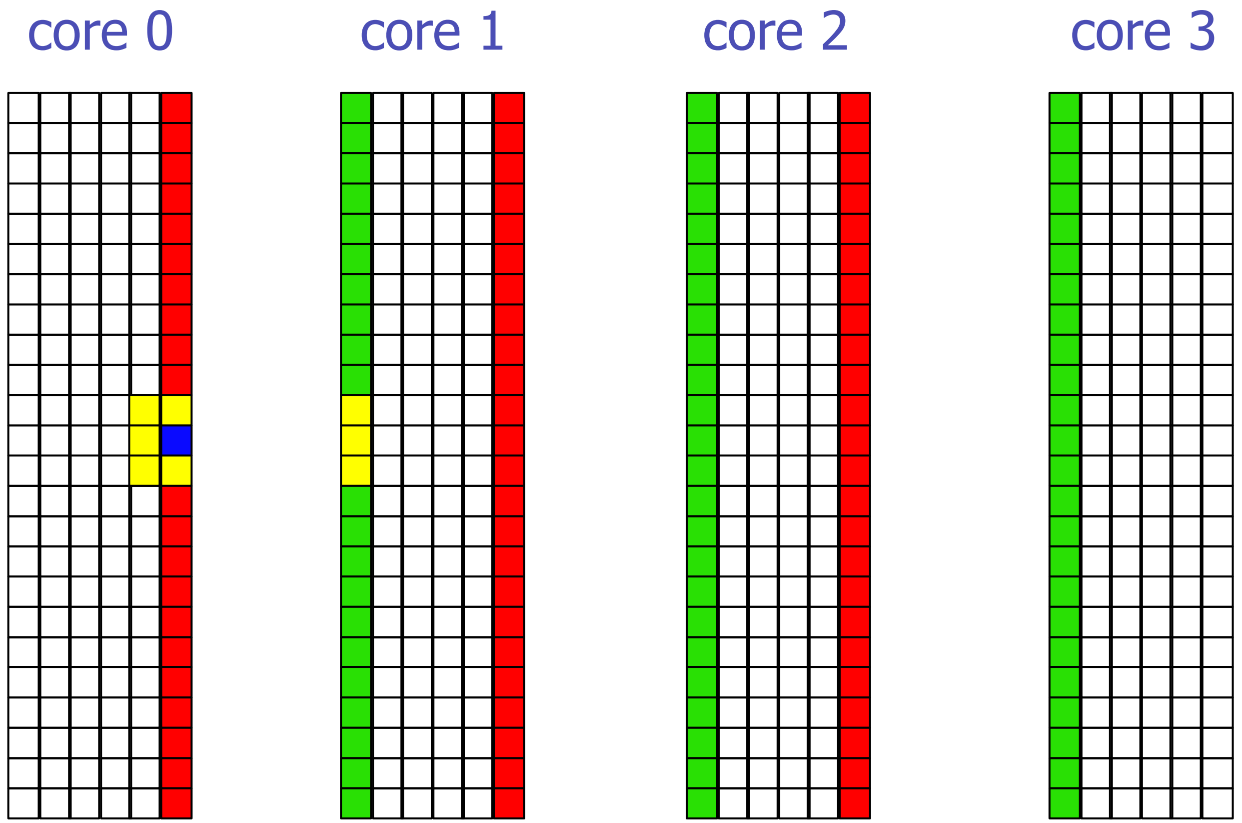
- Main program:

  create P *instances of* LifeThread, *start them, and wait*
  for completion ;

- This version creates the threads only once at the start of the program, and runs the whole of the main loop over time in each thread.
  - *Relatively* , much smaller overheads from thread creation – much more work in each thread.
- But is it correct??

A Problem

- The problem occurs for the cells at the edges of each thread's *domain* (cells it updates):



Shared Access

- The blue cell is updated by core 0. But the decision rule for its update depends on all the yellow cells.
- Hence core zero is *writing* the blue cell, and *reading* all the yellow cells.
  - In general the green cells, while written by the thread they "belong" to, are also read by the thread to their left.
  - Red cells are written by the "owning" thread, but also read by the thread to their right.
- These array elements are *shared variables*.

Outcome

- It is very likely that some threads will run ahead, maybe by several generations, of other threads.
- When a thread is updating cells on the edges of its domain in terms of cells updated by an adjacent thread, it may well be reading the wrong generation of cells.
- Effect is unpredictable and non-deterministic – it is an example of what kind of condition in a concurrent program?

  Race condition.

A New Kind of Synchronization

- In the 2nd year OS unit when we encountered race conditions, a common solution was to employ the form of *thread synchronization* called *mutual exclusion*.
- Here we need a more general, global form of synchronization, that keeps all threads in *lock-step*.
- The mechanism we need is *barrier synchronization*.

BARRIERS

The Barrier

- We will regard a *barrier* as an object that *P* threads share access to.
- It has a single synchronization method which we will call await().
- All *P* threads must call the await() method on the barrier.
  - In general the await() call *blocks*, and will not complete in *any* thread until the *last* of the *P* threads have made the call.
  - The await() call then completes in all *P* threads, and all can continue.

Life Thread with Barriers

```
Class LifeThread {
   void run() {

      while(true) {
        for(int i = begin ; i < end ; i++)
           for(int j = 0 ; j < N ; j++)
              sums [i] [j] = sum of cells values neighbouring (i, j) ;
        barrier.await() ;
        for(int i = begin ; i < end ; i++)
           for(int j = 0 ; j < N ; j++)
               cells [i] [j] = update(cells [i] [j], sums [i] [j]) ;
        barrier.await() ;
      }
   }
}
```

Comments

- In this version, the first barrier synchronization call makes sure that no thread goes on to modify the cells array until all threads have finished using it's old value to calculate their elements of the sums array.
- The second barrier synchronization makes sure that no thread goes on to calculate the sums array in the *next* generation until all threads have finished calculating the new values of the cells array.

Being Safe

- If in doubt, put a barrier synchronization after each "decomposed loop" in the thread.
- Notice that with the barriers the behaviour of the program is now very similar to our first attempt, with separately created threads for each individual set of parallelized loops.
- joining with the main program followed by spawning new threads has a very similar effect to a barrier – but at greater cost!

Life in C, and OpenMP

- So long as we are using an appropriate version of C, pseudocode can look much the same:

```
while(1) {
   for(int i = 0 ; i < N ; i++)
      for(int j = 0 ; j < N ; j++)
         sums [i] [j] = sum of cells values neighbouring (i, j) ;
   for(int i = 0 ; i < N ; i++)
      for(int j = 0 ; j < N ; j++)
         cells [i] [j] = update(cells [i] [j], sums [i] [j]) ;
}
```

- ... elided code for calculating sums and doing update also can be identical to Java.

OpenMP Parallel Version

```
#include <omp.h>

while(1) {

   #pragma omp parallel for
   for(int i = 0 ; i < N ; i++)
      for(int j = 0 ; j < N ; j++)
         sums [i] [j] = sum of cells values neighbouring (i, j) ;

   #pragma omp parallel for
   for(int i = 0 ; i < N ; i++)
      for(int j = 0 ; j < N ; j++)
         cells [i] [j] = update(cells [i] [j], sums [i] [j]) ;
}
```

Comments

- Impressively, this does *everything* our parallel Java version does – but *automatically*.
- It creates *P* threads, where by default *P* is the available number of cores.
- The for(int i = 0 ; i < N ; i++) loops after the parallel for pragma are automatically decomposed with an index subrange for each thread.
- By default a *barrier synchronization* is added after each parallel loop.

Summary

- Next week: Topics in *distributed memory and MPI intro*.

Further Reading

- See links embedded in these slides.