Parallelism

Threads and OpenMP

Parallelism

Threads and OpenMP

- Some useful resources:
  - Tim Mattson OpenMP notes (https://www.openmp.org/wp-content/uploads/Intro_To_OpenMP_Mattson.pdf) and videos (https://www.youtube.com/playlist?list=PLLX-Q6B8xqZ8n8bwjGdzBJ25X2utwnoEG)
  - ARCHER2 Youtube courses (https://www.archer2.ac.uk/training/courses/201006-openmp/)
  - Many other online lecture courses

Parallelism

Threads and OpenMP

- Around 20 years ago, the clock rate for CPUs stalled - we can no longer speed up code simply by running on newer chips
- We must now make use of multiple chips to *parallelise* our code
- This means to provide access for one program to run on many cores
- We first concentrate on *threads* - independent instances of code within a process which *share the same memory*

Parallelism

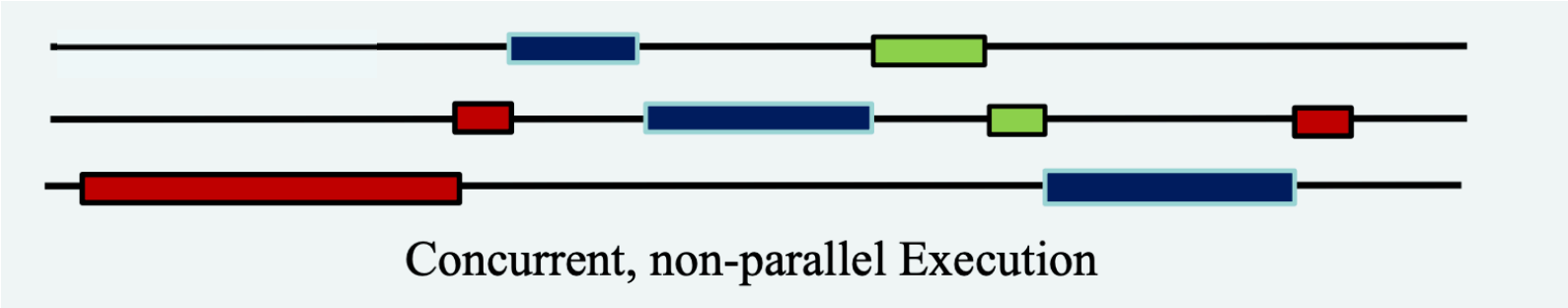
Threads and OpenMP

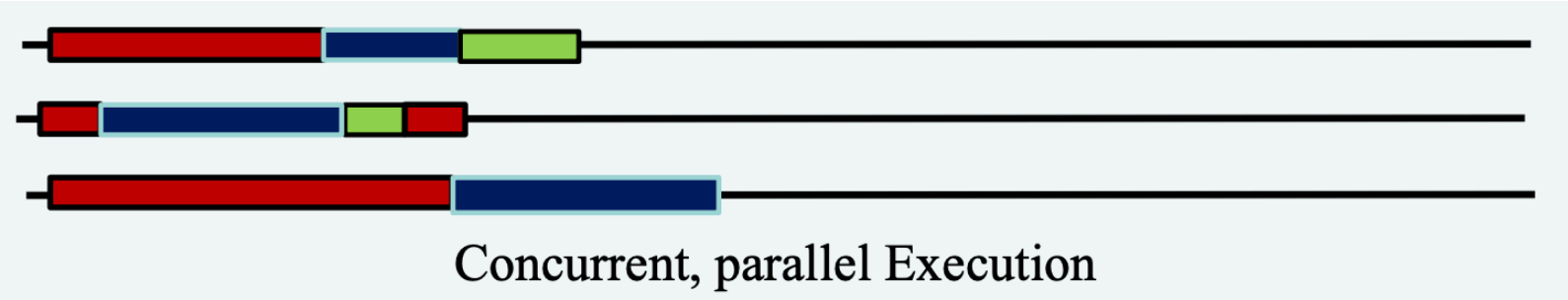
- An important distinction:
  - *Concurrency* - multiple tasks are *logically* active at the same time but *not running* at the same time
    - E.g. checking emails while watching Netflix
    - E.g. C++ `<thread>` library
    - Concurrency *can* reduce wasted clock cycles (reduce latency) - for example, while the program, is fetching something from memory, a program can implement some other part of the program
    - However it is not always associated with speedup

Parallelism

Threads and OpenMP

- *Parallelism* - multiple tasks are *actually* active at the same time
  - Implemented to speed up execution





https://www.openmp.org/wp-content/uploads/Intro_To_OpenMP_Mattson.pdf

Parallelism

Threads and OpenMP

- Another important concept in parallelism is *scaling* - this is how efficiently the program scales to using more cores
- There are two types of scaling:
  - *Strong scaling* - how does the solution time vary with processor number for a *fixed problem*
    - Eg. When we want to *speed up* an existing serial program
  - *Weak scaling* - how does the solution time vary with processor number for a fixed problem size *per processor*
    - Eg. If we want to run a larger problem in the *same time*
- You will use these terms frequently if you do any kind of HPC development

Parallelism

Threads and OpenMP

- OpenMP is an API that supports shared-memory parallelisation via *multithreading*, developed in ~1990s
- The computational workload is divided up between threads that can run at the same time
- Practically:
  - Most directives start with `#pragma omp`
  - We usually `#include <omp.h>`

Parallelism

Threads and OpenMP

Example to demonstrate speedup:

- Set up the Docker container and `connect to running container'
- Compile and run `openmp.cpp` as normal, then time the program with `time`
- Uncomment the line containing `#pragma`
- Compile again, but this time with the flag `-fopenmp`
- Run and time again - also try with different numbers of threads, set using `export OMP_NUM_THREADS=number`
- (But note: are we getting the right answer..?)

Parallelism

Threads and OpenMP

Exercise: Write a multithreaded program that prints “hello world”

```
#include <stdio.h>

int main() {

  int ID = 0;

  printf(“hello(%d)”, ID);

  printf(“world(%d)\n”, ID);

  }
```

Parallelism

Threads and OpenMP

Exercise: Edit to run on multiple threads (remember the compiler flag `-fopenmp`)

```
#include <stdio.h>

int main() {

#pragma omp parallel

   {

   int ID = 0;

   printf("hello(%d)", ID);

   printf("world(%d)\n", ID);

   }

}
```

Parallelism

Threads and OpenMP

Exercise: Edit so we know which output comes from which thread

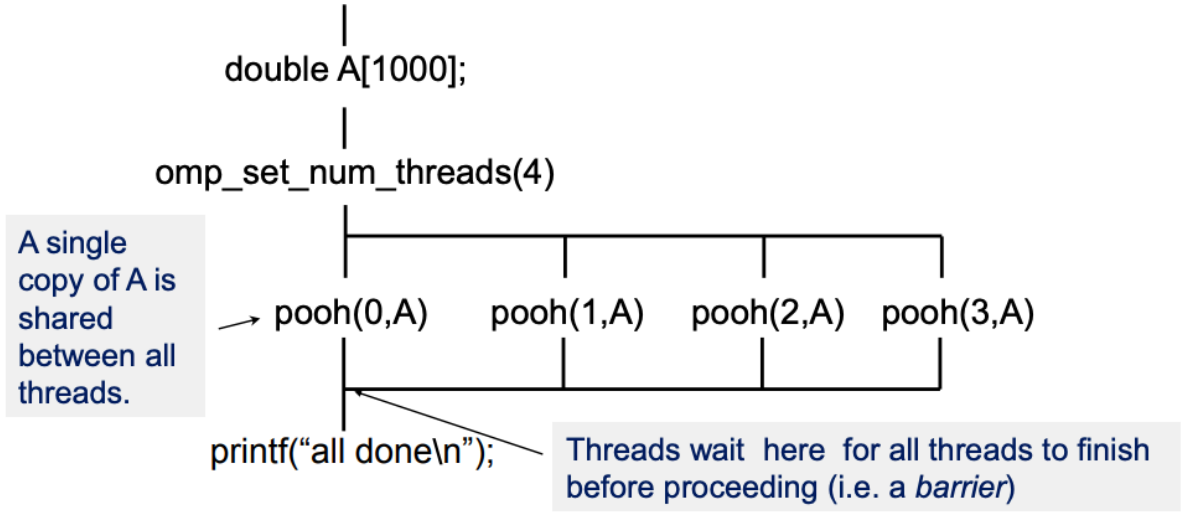
```
#include <stdio.h>
#include <omp.h>


int main() {
#pragma omp parallel
   {
   int ID = omp_get_thread_num();
   printf("hello(%d)", ID);
   printf("world(%d)\n", ID);
   }
}
```

Parallelism

Threads and OpenMP

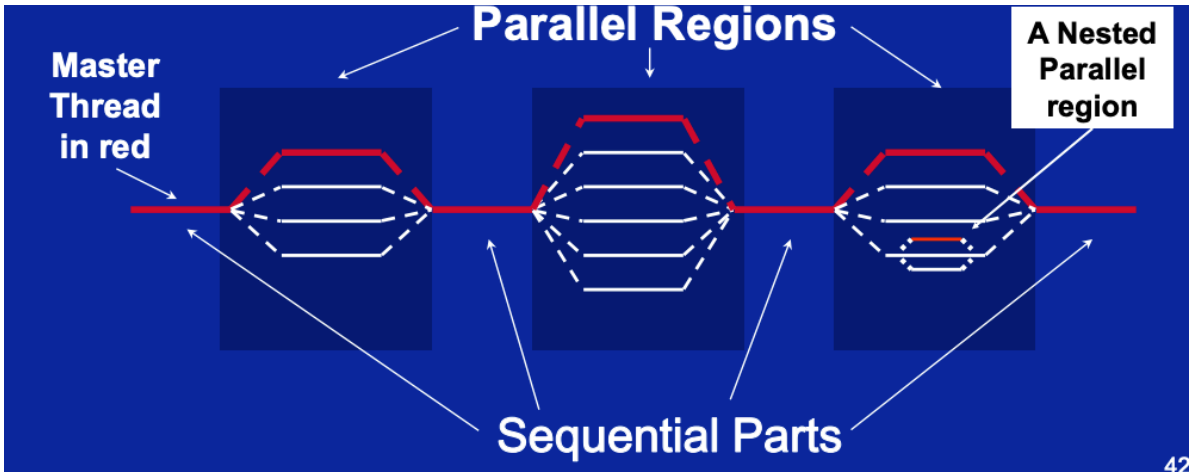
- Here we have the desired output, but it is very muddled
- As we know, OpenMP threads share an address space
- This can lead to *race conditions* - this is where the output of the program changes due to the threads running/finishing at different times



Parallelism

Threads and OpenMP

- We have seen how OpenMP creates a team of threads from the main thread using `#pragma omp parallel`



https://www.openmp.org/wp-content/uploads/Intro_To_OpenMP_Mattson.pdf

- If we want to set the number of threads in the code instead of using environment variables, we use `omp_set_num_threads(num)` before the pragma

- OR we can edit the pragma to `#pragma omp parallel num_threads(num)`

Parallelism

Threads and OpenMP

Exercise: Write a multithreaded program to perform a numerical integration of:

$$\int_0^1 \frac{4}{1+x^2} \mathrm{d}x$$

- We start with the serial version `numerical_integration.cpp`
- How do we want to split this up?
- Solution in `numerical_integration_solution.cpp`
  - Important: pay attention to the local and global variables
- Try timing both of these with more steps

Parallelism

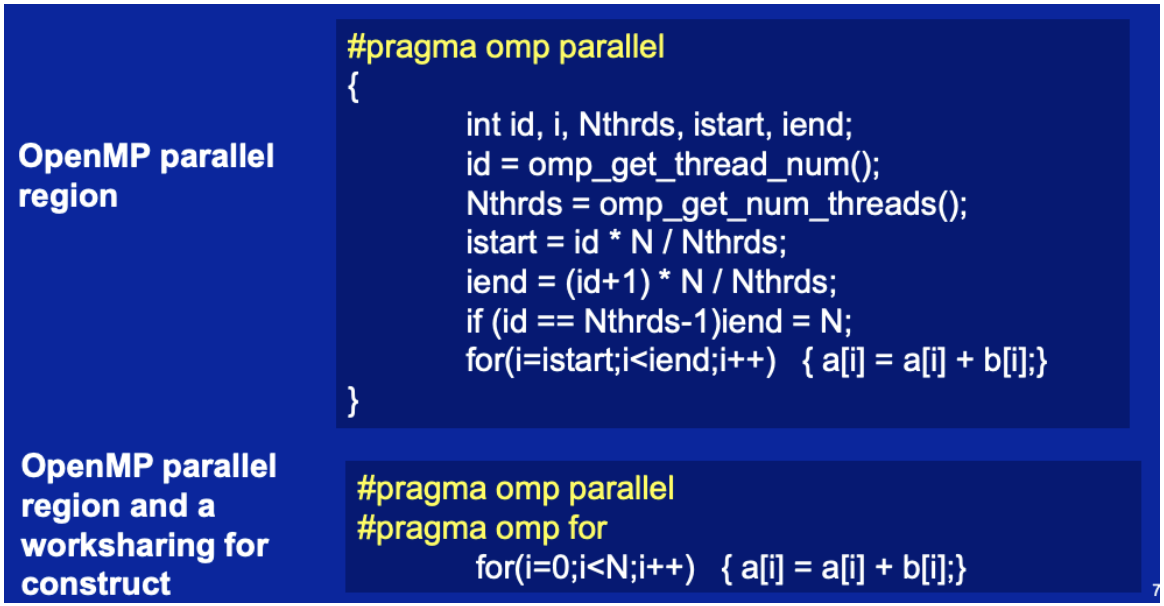
Threads and OpenMP

- This is an example of a Single Program Multiple Data (SPMD) algorithm - uses the thread ID to control which tasks to run
- If we time this example, we currently don't get good scaling
- This could be due to *false sharing* - this occurs if independent data elements happen to sit on the *same cache line*
  - This is common in OpenMP when we promote a scalar to an array to index by the thread number
  - The array elements are contiguous in memory, so share cache lines - poor scalability
- One way around this is to *pad* the arrays so that we are on distinct cache lines - note though that this is architecture dependent

Parallelism

Threads and OpenMP

- As we saw briefly at the beginning, we can parallelise `for` loops using `#pragma omp parallel for` (or in Fortran, `#pragma omp parallel do`)
- This splits up the loop across threads in a team
- Significantly improves the readability of the code



Parallelism

Threads and OpenMP

- Must make sure that the loop iterations are *independent*
- We can also use this for *nested loops*

  ```
  #pragma omp parallel for collapse(2)
  ```

  where here, the number of loops is 2
- If we do have a loop where we are gathering values into one variable, such as eg. `sum += value[i]`, we can use a *reduction*
- In this case, a local copy of each list variable is made and updates, then local copies are reduced to a single value and combined

Parallelism

Threads and OpenMP

Example:

```
double sum=0., average, values[max];
int i;
for(i=0; i<max; i++)

{

  sum += values[max];

}

average = sum/max;
```

Parallelism

Threads and OpenMP

Example:

```
double sum=0., average, values[max];
int I;
#pragma omp parallel for reduction (+:sum)


for(i=0; i<max; i++)

{
    sum += values[max];
}

average = sum/max;
```

Exercise for outside class - parallelise the integration program with a loop construct

Parallelism

Threads and OpenMP

- Returning to *race conditions* - how do we stop these from happening with multiple threads?
  - Barriers - force each thread to wait until all of the threads have finished executing

    `#pragma omp barrier`

  - Mutual exclusion - define a block of the code that only one thread at a time can execute (sort of defeats the point of adding the parallelism…)

    `#pragma omp critical` or

    `#pragma omp atomic` (for memory updates only)

    - We can also use `#pragma omp critical` to stop false sharing