Parallel Computing with GPUs

OpenMP
Part 1 - OpenMP Overview

The University Of Sheffield.

Dr Paul Richmond
http://paulrichmond.shef.ac.uk/teaching/COM4521/

This Lecture (learning objectives)

- Introducing OpenMP
  - Identify the language purpose and approach
- OpenMP "Hello World"
  - Recognise the basic structure of an OpenMP directive
  - Examine output from a parallel application
  - Present the fork and join model

OpenMP

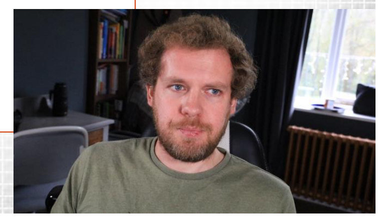
- Open Multi-Processing Standard
  - An API that supports shared memory programming in C, C++ and FORTRAN
  - Cross platform support using native threading
    - Higher level than OS models and portable
  - Is not suitable for distributed computing (look at MPI)
- It is not an automatic parallel programming language
  - Parallelism is explicitly defined and controlled by the programmer
  - Requires compiler directives, a runtime, environment variables

Application	Compiler	Environment
OpenMP Runtime		
Platform threading model (e.g. Windows threading or pthreads)		

OpenMP Compiler Directives

- Use of `#pragmas`
  - If not understood by the compiler then they are ignored
  - Does not require serial code to be changed
  - Allows behaviour to be specified which are not part of the C standard specification

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

int main()
{
#pragma omp parallel
    {
        printf("Hello World\n");
    }
    return 0;
}
```



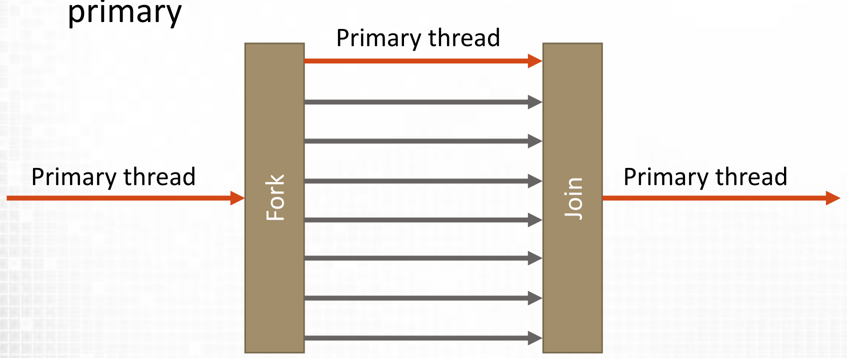
Extending OpenMP Hello World

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

int main()
{
#pragma omp parallel
    {
        int thread = omp_get_thread_num();
        int max_threads = omp_get_max_threads();
        printf("Hello World (Thread %d of %d)\n", thread, max_threads);
    }
return 0;
}
```

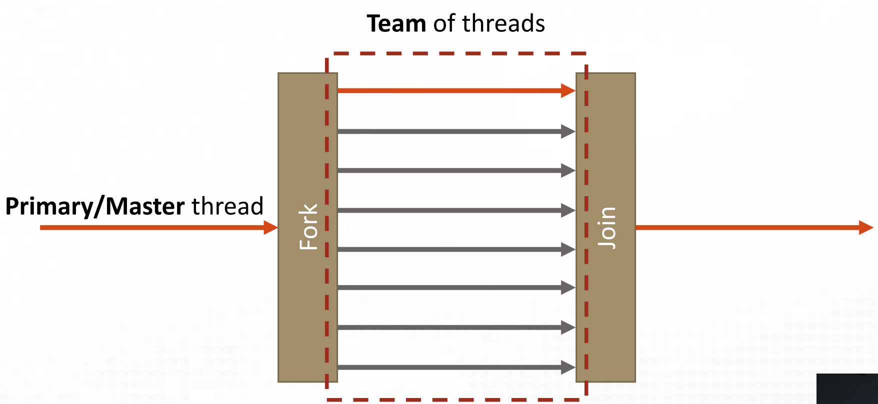
```
Hello World (Thread 5 of 8)
Hello World (Thread 6 of 8)
Hello World (Thread 2 of 8)
Hello World (Thread 7 of 8)
Hello World (Thread 1 of 8)
Hello World (Thread 0 of 8)
Hello World (Thread 3 of 8)
Hello World (Thread 4 of 8)
```

Fork and Join

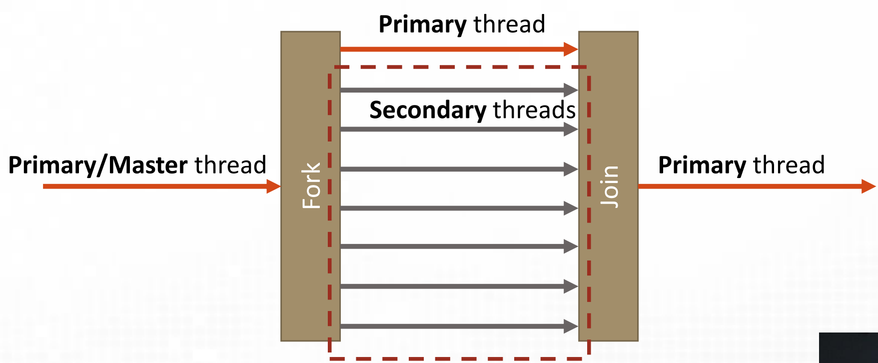
- OpenMP uses a fork a join model
  - Fork: Creates a number of parallel threads from a primary thread
    - Primary thread is always thread 0
    - No guarantee of order
  - Join: Synchronises thread termination and returns program control to primary

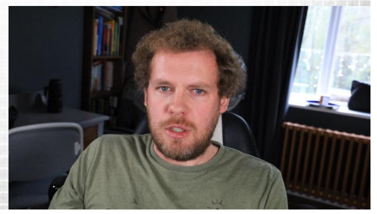


Terminology



Terminology





Summary

- ❑Introducing OpenMP
  - ❑Identify the language purpose and approach
- ❑OpenMP “Hello World”
  - ❑Recognise the basic structure of an OpenMP directive
  - ❑Examine output from a parallel application
  - ❑Present the fork and join model

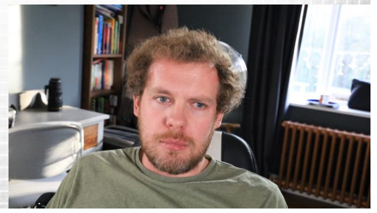
- ❑Next Lecture: Loops and Critical Sections

Parallel Computing with GPUs

OpenMP
Part 2 – Loops & Critical Sections

The University Of Sheffield.

Dr Paul Richmond
http://paulrichmond.shef.ac.uk/teaching/COM4521/



This Lecture (learning objectives)

- ❑Parallelising Loops
  - ❑Assign parallel section of code from loops to threads within OpenMP
- ❑Critical Sections
  - ❑Identify the potential for race conditions in parallel code
  - ❑Examine a range of solutions for different race conditions

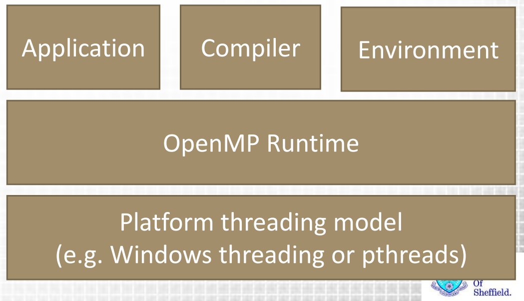
OpenMP Syntax

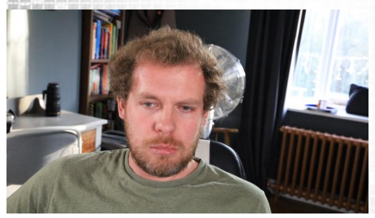
- ❑Parallel region directive
  - ❑`#pragma omp parallel [clause list] {structured block}`
  - ❑Spawns a number of parallel threads
- ❑Clauses
  - ❑Are used to specify modifications to the parallel directive e.g.
    - ❑Control scoping of variables in multiple threads
    - ❑Dictate the number of parallel threads (example below)
    - ❑Conditional parallelism

```
#pragma omp parallel num_threads(16)
    {
        int thread = omp_get_thread_num();
        int max_threads = omp_get_max_threads();
        printf("Hello World (Thread %d of %d)\n", thread, max_threads);
    }
```

`num_threads()`

- Without this clause `OMP_NUM_THREADS` will be used
  - This is an environment variable
  - Set to the number of cores (or hyperthreads) on your machine
  - This can be set globally by `omp_set_num_threads(int)`
  - Value can be queried by `int omp_get_num_threads();`
- `num_threads` takes precedence over the environment variable
- `num_threads()` does not guarantee that the number requested will be created
  - System limitations may prevent this
  - However: It almost always will





`parallel for`

- `#pragma omp for`
  - Assigns work units to the team
  - Divides loop iterations between threads
- For can be combined e.g. `#pragma omp parallel for`
  - Threads are spawned and then assigned to loop iterations

```c
int n;
#pragma omp parallel for
for (n = 0; n < 8; n++){
    int thread = omp_get_thread_num();
    printf("Parallel thread %d \n", thread);
}
```

```c
#pragma omp parallel
{
  int n;
  for (n = 0; n < 8; n++){
    int thread = omp_get_thread_num();
    printf("Parallel thread %d \n", thread);
  }
}
```

```c
#pragma omp parallel
{
int n;
#pragma omp for
  for (n = 0; n < 8; n++){
    int thread = omp_get_thread_num();
    printf("Parallel thread %d \n", thread);
  }
}
```

Which is the odd one out?

`parallel for`

- `#pragma omp for`
  - Assigns work units to the team
  - Divides loop iterations between threa
- For can be combined e.g. `#pragm`
  - Threads are spawned and then assigr

```
Parallel thread 0
Parallel thread 0
Parallel thread 0
Parallel thread 0
Parallel thread 0
Parallel thread 0
Parallel thread 0
Parallel thread 0
Parallel thread 2
Parallel thread 2
Parallel thread 2
Parallel thread 2
Parallel thread 2
Parallel thread 2
Parallel thread 2
Parallel thread 2
Parallel thread 5
Parallel thread 5
Parallel thread 5
Parallel thread 5
Parallel thread 4
Parallel thread 4
Parallel thread 3
Parallel thread 3
Parallel thread 1
…
```

```c
#pragma omp parallel
{
  int n;
  for (n = 0; n < 8; n++){
    int thread = omp_get_thread_num();
    printf("Parallel thread %d \n", thread);
  }
}
```

What is wrong with this code?

- Consider a problem such as Taylor series expansion for *cos* function
  - $\cos(x) = \sum_{n=0}^{\infty}(-1)^{n-1}\frac{x^{2n-1}}{(2n)!}$
  - $\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} + \frac{x^6}{6!} \ldots$

```c
int n;
double result = 0.0;
double x = 1.0;

#pragma omp parallel for
for (n = 0; n < EXPANSION_STEPS; n++){
    double r = pow(-1, n - 1) * pow(x, 2 * n - 1) / fac(2 * n);
    result -= r;
}

printf("Approximation of x is %f, value is %f\n", result, cos(x));
```

Critical sections

- Consider a problem such as Taylor series expansion for *cos* function
  - $\cos(x) = \sum_{n=0}^{\infty} (-1)^{n-1} \frac{x^{2n-1}}{(2n)!}$
  - $\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} + \frac{x^6}{6!} \ldots$

```c
int n;
double result = 0.0;
double x = 1.0;

#pragma omp parallel for
for (n = 0; n < EXPANSION_STEPS; n++){
    double r = pow(-1, n - 1) * pow(x, 2 * n - 1) / fac(2 * n);
    result -= r;
}

printf("Approximation of x is %f, value is %f\n", result, cos(x));
```

Race Condition: Multiple threads try to write to the same value!
(undefined behaviour and unpredictable results)

Critical sections

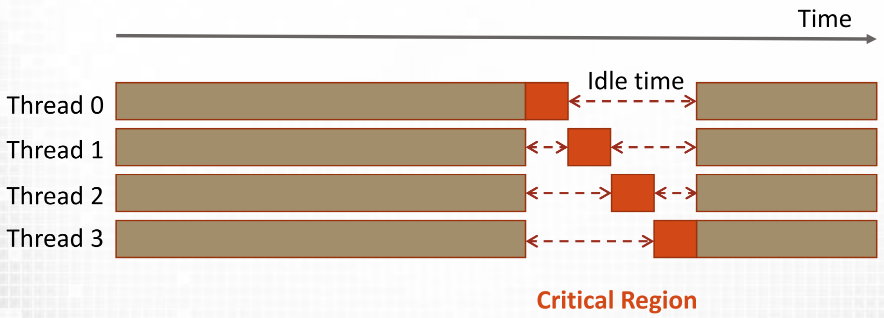
- Consider a problem such as Taylor series expansion for *cos* function
  - $\cos(x) = \sum_{n=0}^{\infty} (-1)^{n-1} \frac{x^{2n-1}}{(2n)!}$
  - $\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} + \frac{x^6}{6!} \ldots$

```c
int n;
double result = 0.0;
double x = 1.0;

#pragma omp parallel for
for (n = 0; n < EXPANSION_STEPS; n++){
    double r = pow(-1, n - 1) * pow(x, 2 * n - 1) / fac(2 * n);
    #pragma omp critical
    {
        result -= r;
    }
}

printf("Approximation of x is %f, value is %f\n", result, cos(x));
```

Solution: Define as a critical section

Critical sections

- `#pragma omp critical [name]`
  - Ensures mutual exclusions when accessing a shared value
  - Prevents race conditions
  - A thread will wait until no other thread is executing a critical region (with the same name) before beginning
  - Unnamed critical regions map to the same unspecified name



Atomics

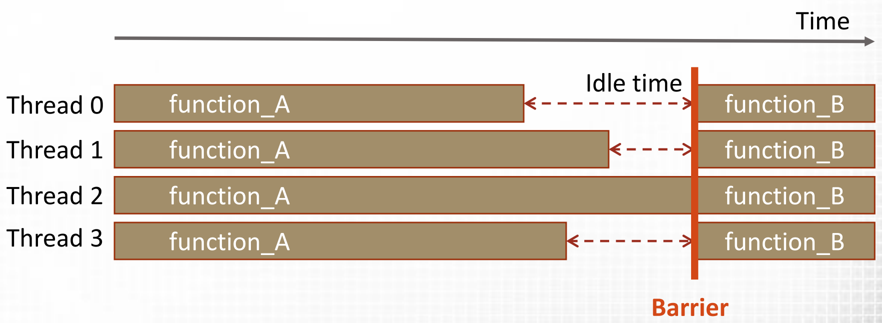
- Atomic operations can be used to safely increment a shared numeric value
  - For example summation
  - Atomics only apply to the immediate assignment
- Atomics are usually faster than critical sections (benchmark to confirm)
  - Critical sections can be applied to general blocks of code (atomics can not)
- Example
  - Compute histogram of random values for a given range
  - Random is an `int` array of size `NUM_VALUES` with random value within `0:RANGE`
  - Histogram is an `int` array of size `RANGE` with `0` values;

```c
#pragma omp parallel
{
    int i;
#pragma omp for
    for (i = 0; i < NUM_VALUES; i++){
        int value = randoms[i];
#pragma omp atomic
        histogram[value]++;
    }
}
```

Barriers

- #pragma omp barrier
  - Synchronises threads at a barrier point
  - Parallel regions have an implicit barrier
  - Can be used to ensure execution of particular code is complete
    - E.g. data read by `function_B`

```
#pragma omp parallel
{
    function_A()
#pragma omp barrier
    function_B();
}
```



Single and Master Sections

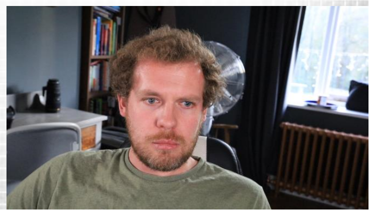
- #pragma omp single { … }
  - Used to ensure that only a single thread executes a region of a structured block
  - Useful for I/O and initialisation
  - First available thread will execute the defined region
    - No control over which this is
  - Will cause an implicit barrier (after structured block) unless a `nowait` clause is used
    - E.g. `#pragma omp single nowait`
    - `nowait` will remove an implied barrier and can also be applied to parallel for loops
- #pragma omp master { … }
  - Similar to `single` but will always use the primary/master thread
  - Preferable to single (usually faster)
  - Does not have an implicit barrier

Master example

```
int t, r;
int local_histogram[THREADS][RANGE];

zero_histogram(local_histogram);

#pragma omp parallel num_threads(THREADS)
  {
  int i;
#pragma omp for
  for (i = 0; i < NUM_VALUES; i++){
    int value = randoms[i];
    local_histogram[omp_get_thread_num()][value]++;
  }
#pragma omp barrier
#pragma omp master
  for (t = 0; t < THREADS; t++){
    for (r = 0; r < RANGE; r++){
      histogram[r] += local_histogram[t][r];
    }
  }
}
```

Same result as the atomic version

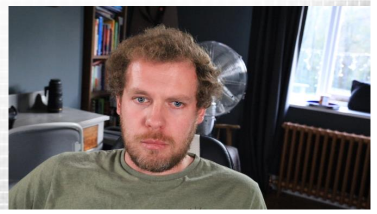
Benchmark to understand performance!



Summary

- Parallelising Loops
  - Assign parallel section of code from loops to threads within OpenMP
- Critical Sections
  - Identify the potential for race conditions in parallel code
  - Examine a range of solutions for different race conditions

- Next Lecture: Scoping and Tasks

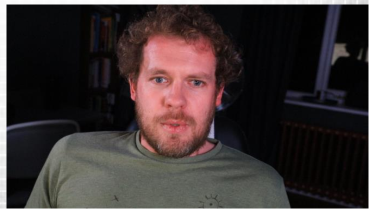


Parallel Computing with GPUs

OpenMP
Part 3 – Scoping & Task Parallelism

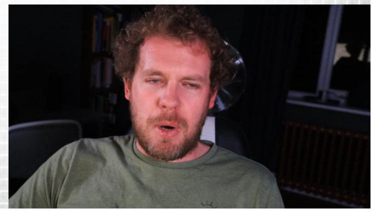
The University Of Sheffield.

Dr Paul Richmond
http://paulrichmond.shef.ac.uk/teaching/COM4521/



This Lecture (learning objectives)

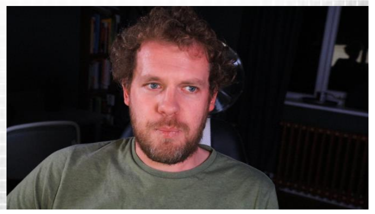
- ❑Scoping
  - ❑Determine appropriate scope for OpenMP variables
  - ❑Label variable explicitly using scope clauses
- ❑Task Parallelism
  - ❑Develop programs using a task parallel model



Scoping

- ❑Scope refers to the part of the program in which a variable can be used
- ❑OpenMP has different scoping to serial programming
  - ❑We must specify if a variable is private or shared between threads

- ❑**Shared**: A variable can be accessed by all threads in the team
  - ❑All variables declared outside of a parallel loop are shared by default
- ❑**Private**: A Variable is local to a single thread and can only be accessed by this thread within the structured block it is defined
  - ❑All variables declared inside a structured block are private by default



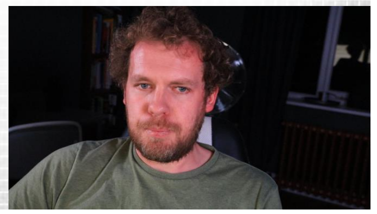
Scoping

```
int t, r;
int local_histogram[THREADS][RANGE];

zero_histogram(local_histogram);

#pragma omp parallel num_threads(THREADS)
  {
  int i;
#pragma omp for
  for (i = 0; i < NUM_VALUES; i++){
    int value = randoms[i];
    local_histogram[omp_get_thread_num()][value]++;
  }
#pragma omp barrier
#pragma omp master
  for (t = 0; t < THREADS; t++){
    for (r = 0; r < RANGE; r++){
      histogram[r] += local_histogram[t][r];
    }
  }
}
```

Shared

But what about i?

Private



Scoping

```
int t, r;
int local_histogram[THREADS][RANGE];

zero_histogram(local_histogram);

#pragma omp parallel num_threads(THREADS)
  {
  int i;
#pragma omp for
  for (i = 0; i < NUM_VALUES; i++){
    int value = randoms[i];
    local_histogram[omp_get_thread_num()][value]++;
  }
#pragma omp barrier
#pragma omp master
  for (t = 0; t < THREADS; t++){
    for (r = 0; r < RANGE; r++){
      histogram[r] += local_histogram[t][r];
    }
  }
}
```

Shared

`i` is private as it is the counter of the parallel for loop

Private

Explicit scoping

- Why is explicit scoping required?
  - It is possible to use implicit scoping as in previous example
    - Although it is good practice to use shared for any shared variables
  - The clause default(shared or none) is helpful in ensuring you have defined variables scope correctly
    - By changing the default scope from shared to none it enforces explicit scoping of variables and will give errors if scoping is not defined
  - `const` variables can not be explicitly scoped (always shared) - more
    - Not enforced in windows but this is against the spec

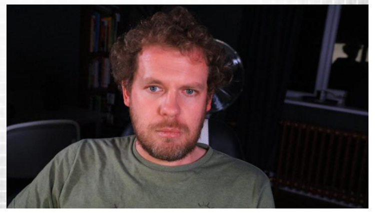
```
int a, b = 0;
#pragma omp parallel default(none) shared(b)
{
    b += a;
}
```

error C3052: 'a' : variable doesn't appear in a data-sharing clause under a default(none) clause

Explicit scoping

- Why is explicit scoping required?
  - Older C programming (C89) style has variable declarations before definitions and statements (including loops)
    - Requires declarations to be made explicitly private for the parallel structured block
    - E.g. Consider our atomic histogram example

```
void calculate_histogram()
{
    int i;
    int value;
#pragma omp parallel for private(value)
    for (i = 0; i < NUM_VALUES; i++){
        value = randoms[i];
#pragma omp atomic
        histogram[value]++;
    }
}
```



Advanced private scoping

- If you want to pass the value of a variable outside of a parallel structured block then you must use the `firstprivate` clause
  - Private variables will be initialised with the value of the master thread before the parallel directive
- If you want to pass a private value to a variable outside of the parallel for loop you can use the `lastprivate` clause
  - This will assign the value of the **last iteration** of the loop

```
int i = 10;
#pragma omp parallel private(i)
{
    printf("Thread %d: i = %d\n", omp_get_thread_num(), i);
}
```

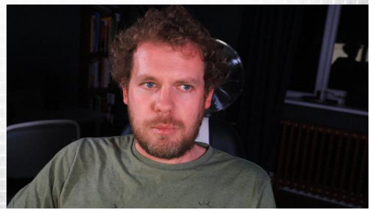
```
Thread 0: i = 0
Thread 2: i = 0
Thread 1: i = 0
Thread 3: i = 0
```

```
int i = 10;
#pragma omp parallel firstprivate(i)
{
    printf("Thread %d: i = %d\n", omp_get_thread_num(), i);
}
```

```
Thread 0: i = 10
Thread 2: i = 10
Thread 1: i = 10
Thread 3: i = 10
```

Data vs Task Parallelism

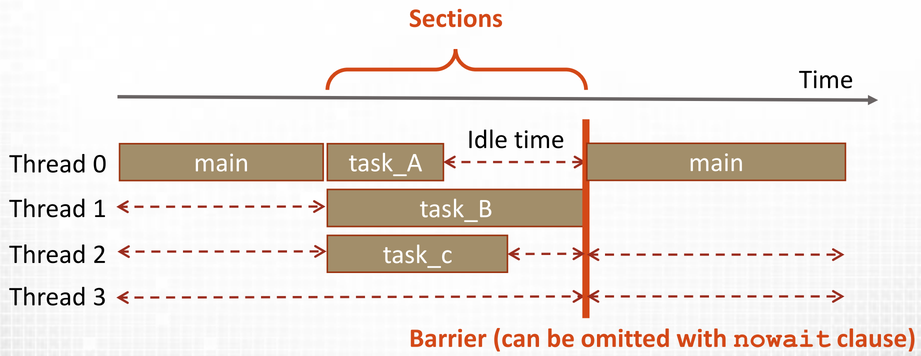
- Parallelism over loops is data parallelism. i.e.
  - The task is the same (the loop) – OpenMP model
  - Parallelism is over the data elements the loop refers to
- What about task parallelism?
  - Task Parallelism: Divide a set of tasks between threads
  - This is supported by sections
  - Further task parallelism is supported by OpenMP tasks
    - This is OpenMP 3.0 spec and not supported in Visual Studio 2017
    - Very similar to sections

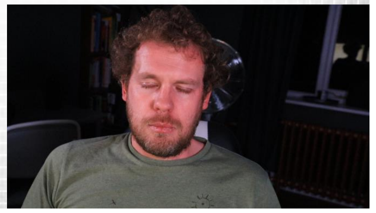


Sections (task parallelism OpenMP <3.0)

- `#pragma omp sections [clauses]`
  - Defines a code region where individual sections can be assigned to individual threads
  - Each section is executed exactly once by one thread
  - Unused threads wait for **implicit barrier**

```
#pragma omp parallel

#pragma omp sections
{
#pragma omp section
    task_A();
#pragma omp section
    task_B();
#pragma omp section
    task_C();
}
```

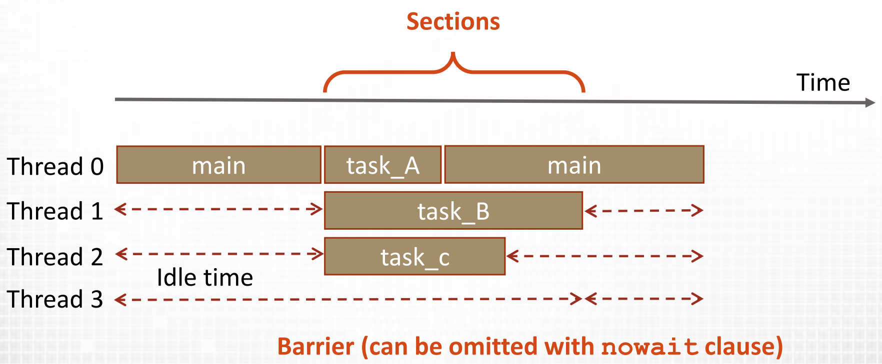




Sections

- If `nowait` clause is used then sections omit the barrier
  - will immediately enter other parallel sections

```
#pragma omp parallel

#pragma omp sections nowait
{
#pragma omp section
    task_A();
#pragma omp section
    task_B();
#pragma omp section
    task_C();
}
```



Summary

- Scoping
  - Determine appropriate scope for OpenMP variables
  - Label variable explicitly using scope clauses
- Task Parallelism
  - Develop programs using a task parallel model