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

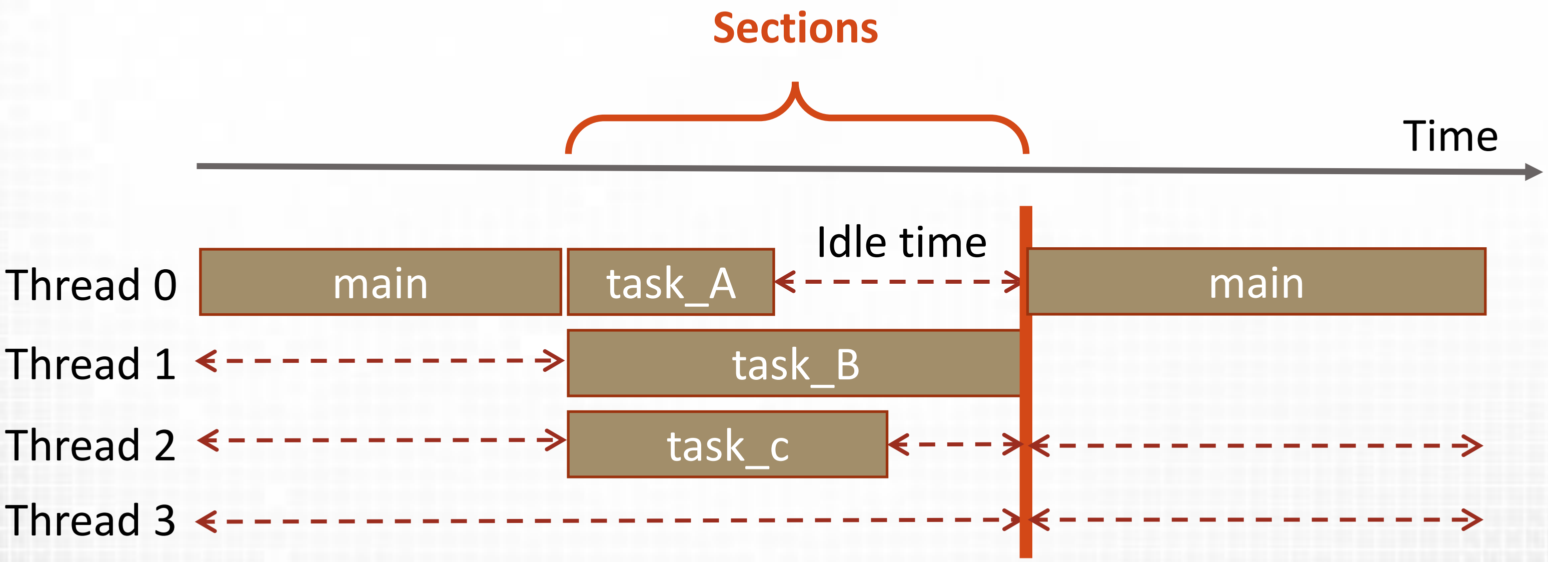
Sections (task parallelism OpenMP <3.0)

❑ #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();
}
```



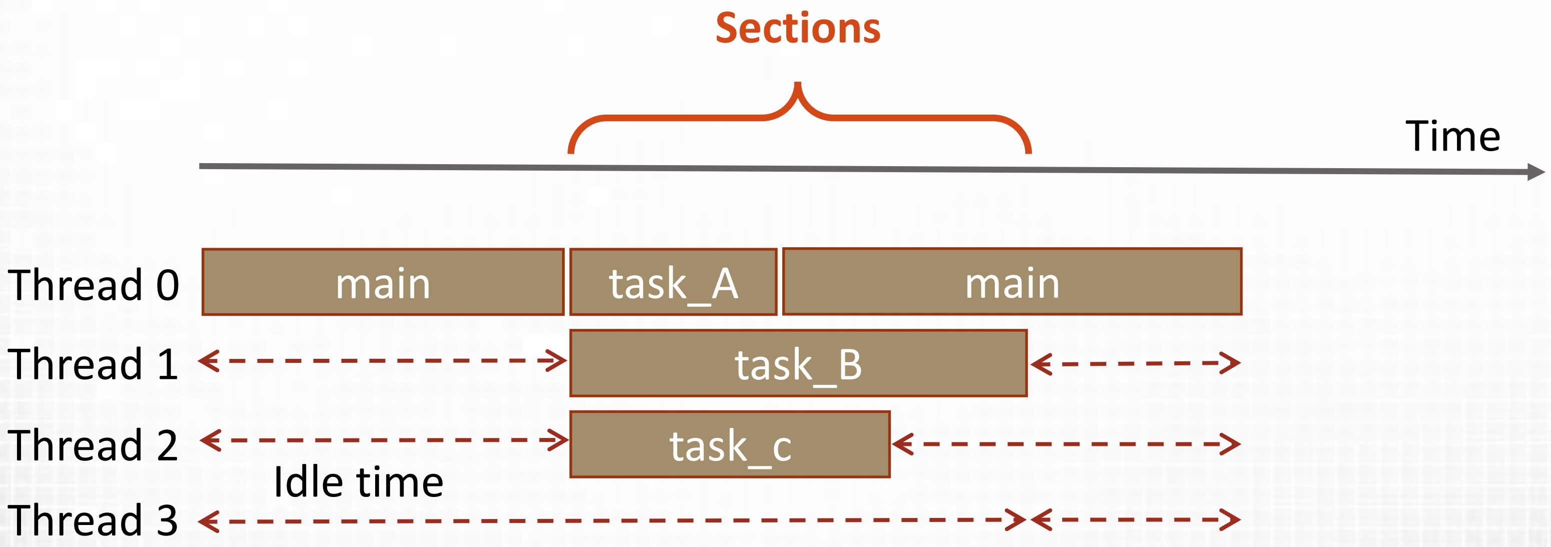
Barrier (can be omitted with nowait clause)

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();
}
```



Barrier (can be omitted with `nowait` clause)

Summary

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