Parallel Computing with GPUs: OpenMP 2

Dr Mozhgan Kabiri Chimeh
http://mkchimeh.staff.shef.ac.uk/teaching/COM4521

- ❑OpenMP Timing
- ❑Parallel Reduction
- ❑Scheduling
- ❑Nesting
- ❑Summary

The problem with clock()

- `clock()` function behaviour
  - In windows: represents a measure of real time (wall clock time)
  - Linux: represents a cumulative measure of time spent executing instructions
    - Cumulative over core = not good for measuring parallel performance
- Open MP timing
  - omp_get_wtime() – cross platform wall clock timing

```
double begin, end, seconds;
begin = omp_get_wtime();

some_function();

end = omp_get_wtime();
seconds = (end - begin);

printf("Sum Time was %.2f seconds\n", seconds);
```

The University Of Sheffield.

NVIDIA GPU RESEARCH CENTER

- ❑OpenMP Timing
- ❑Parallel Reduction
- ❑Scheduling
- ❑Nesting
- ❑Summary

Parallel Reduction

- A Reduction is the combination of local copies of a variable into a single copy
  - Consider a case where we want to sum the values of a function operating on a vector of values;

```
void main(){
    int i;
    float vector[N];
    float sum;

    init_vector_values(vector);
    sum = 0;

    for (i = 0; i < N; i++){
        float v = some_func(vector[i]);
        sum += v;
    }
    printf("Sum of values is %f\n", sum);
}
```

Candidate for parallel reduction…

NBody calculation with OpenMP

```
void main(){
    int i;
    float vector[N];
    float sum;

    init_vector_values(vector);
    sum = 0;

#pragma omp parallel for reduction(+: sum);
    for (i = 0; i < N; i++){
        float v = some_func(vector[i]);
        sum += v;
    }
    printf("Sum of values is %f\n", sum);
}
```

Without reduction we would need a critical section to update the shared variable!

The University Of Sheffield.

NVIDIA GPU RESEARCH CENTER

OpenMP Reduction

- Reduction is supported with the reduction clause which requires a reduction variable
  - E.g. `#pragma omp parallel reduction(+: sum_variable) {…}`
  - Reduction variable is implicitly private to other threads
- OpenMP implements this by;
  - Creating a local (private) copy of the (shared) reduction variable
  - Combining local copies of the variable at the end of the structured block
  - Saving the reduced value to the shared variable in the master thread.
- Reduction operators are `+`, `-`, `*`, `&`, `|`, `&&` and `||`
  - `&`: bitwise and
  - `|`: bitwise or
  - `&&`: logical and
  - `||`: logical or

- ❑OpenMP Timing
- ❑Parallel Reduction
- ❑Scheduling
- ❑Nesting
- ❑Summary

Scheduling

- OpenMP by default uses static scheduling
  - Static: schedule is determined at compile time
  - E.g. `#pragma omp parallel for` **`schedule(static)`**
- In general: `schedule(type [, chunk size])`
  - `type=static`: Iterations assigned to threads before execution (preferably at compile time)
  - *`type=dynamic`: iterations are assigned to threads as they become available*
  - *`type=guided`: iterations are assigned to threads as they become available (with reducing chunk size)*

  - `type=auto`: compiler and runtime determine the schedule
  - `type=runtime`: schedule is determined at runtime by an environment variable, illegal to specify chunk size

What would be a use case where static scheduling is a bad choice?

Static scheduling chunk size

- chunk size
  - Refers to the amount of work assigned to each thread
  - By default chunk size is to divide the work by the number of threads
    - Low overhead (no going back for more work)
    - Not good for uneven workloads
    - E.g. consider our last lectures Taylor series example (updated to use reduction)

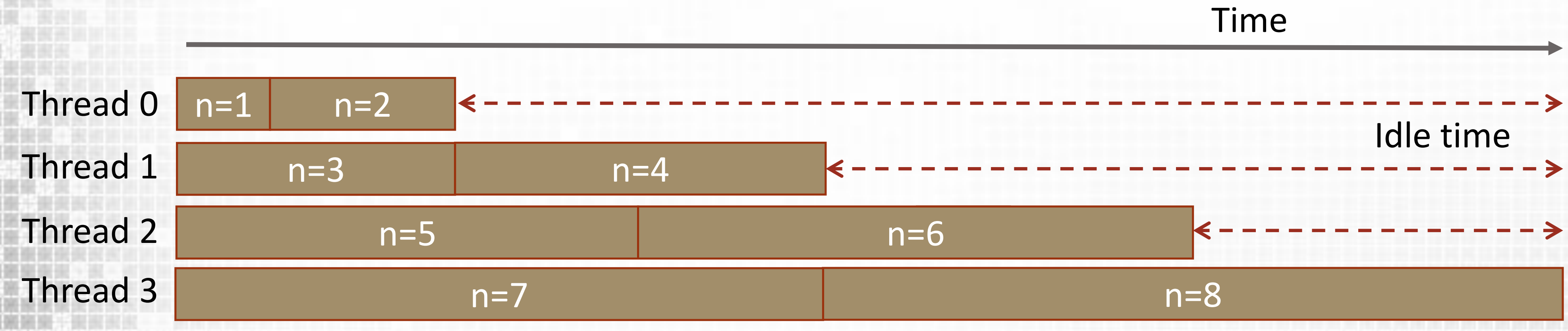
```
int n;
double result = 0.0;
double x = 1.0;

#pragma omp parallel for reduction(-: result)
  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 is %f, value is %f\n", result, cos(x));
```

Uneven workload

Scheduling Workload

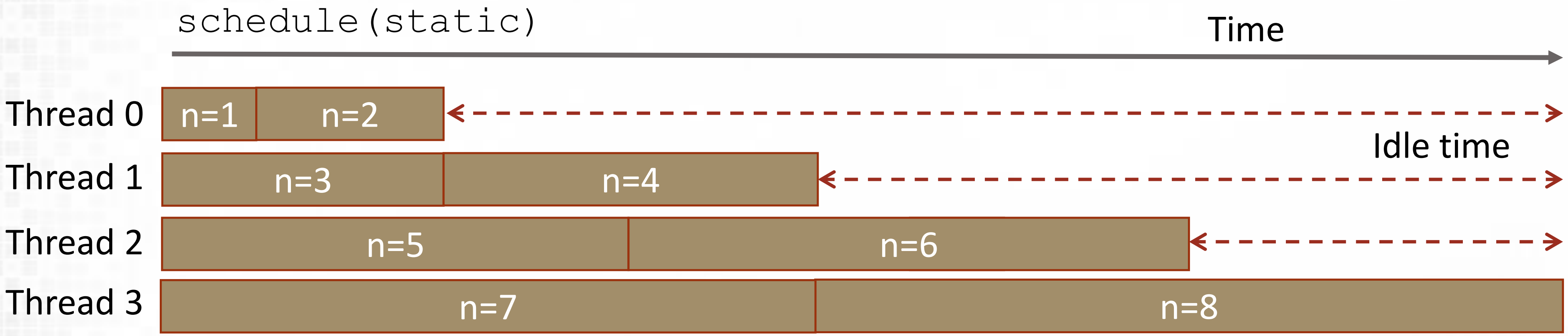
```
long long int factorial(int n)
{
  if (n == 0)
    return 1;
  else
    return(n * factorial(n - 1));
}
```

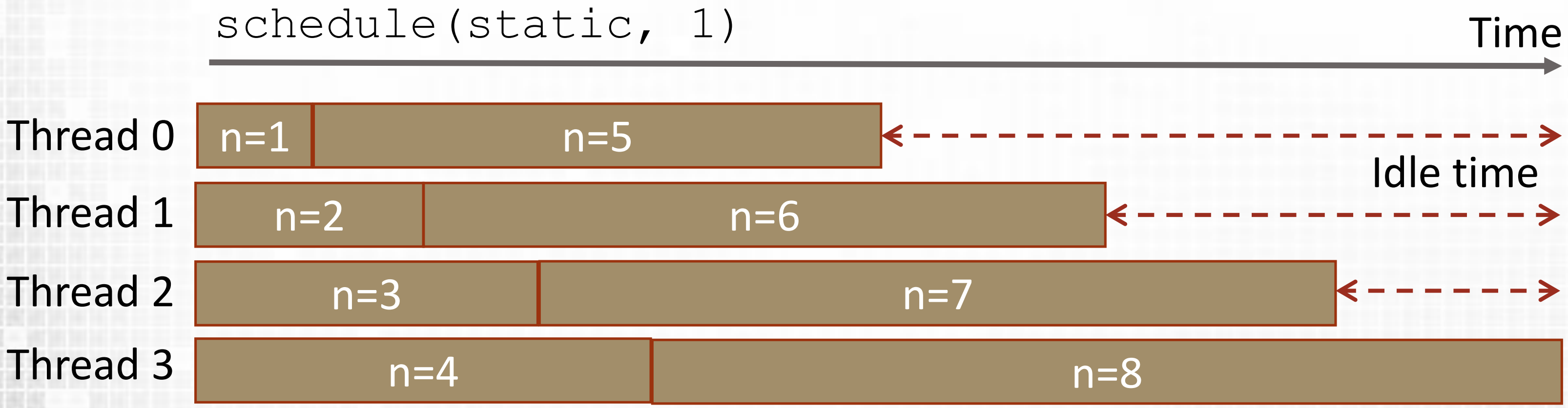
- ❑Uneven workload amongst threads
  - ❑Increase in *n* leads to increased computation
  - ❑E.g. `EXPANSION_STEPS=8, num_threads(4), schedule(static)`



Cyclic Scheduling

- It would be better to partition the workload more evenly
  - E.g. Cyclic scheduling via chunk size





Cyclic Scheduling

```
#pragma omp for num_threads(4)
for (i = 0; i < 16; i++)
```

`schedule(static, 1)`

Thread 0	0	4	8	12
Thread 1	1	5	9	13
Thread 2	2	6	10	14
Thread 3	3	7	11	15

`schedule(static, 2)`

Thread 0	0	1	8	9
Thread 1	2	3	10	11
Thread 2	4	5	12	13
Thread 3	6	7	14	15

`schedule(static, 4)`

Thread 0	0	1	2	3
Thread 1	4	5	6	7
Thread 2	8	9	10	11
Thread 3	12	13	14	15

Default case

- Default chunk size is `n/threads`
  - where `n` is the number of iterations

Dynamic and Guided Scheduling

- Dynamic (med overhead)
  - **Iterations are broken down by chunk size**
  - Threads request chunks of work from a runtime queue when they are free
  - Default chunk size is 1
- Guided (high overhead)
  - **Chunks of the workload grow exponentially smaller**
  - Threads request chunks of work from a runtime queue when they are free
  - Chunk size is the size which the workloads decrease to
    - with the exception of last chunk which may have remainder
- Both
  - Requesting work dynamically creates overhead
    - Not well suited if iterations are balanced
  - Overhead vs. imbalance: How do I decide which is best?
    - **Benchmark all to find the best solution**

- OpenMP Timing
- Parallel Reduction
- Scheduling
- Nesting
- Summary

Nesting

- Consider the following example…
  - How should we parallelise this example?

```
for (i = 0; i < OUTER_LOOPS; i++){
    for (j = 0; j < INNER_LOOPS; j++){
        printf("Hello World (Thread %d)\n", omp_get_thread_num());
    }
}
```

Nesting

- Consider the following example…
  - How should we parallelise this example?

```
#pragma omp parallel for
for (i = 0; i < OUTER_LOOPS; i++){
    for (j = 0; j < INNER_LOOPS; j++){
        printf("Hello World (Thread %d)\n", omp_get_thread_num());
    }
}
```

- What if OUTER_LOOPS << number of threads
  - E.g. `OUTER_LOOPS = 2`

Nesting

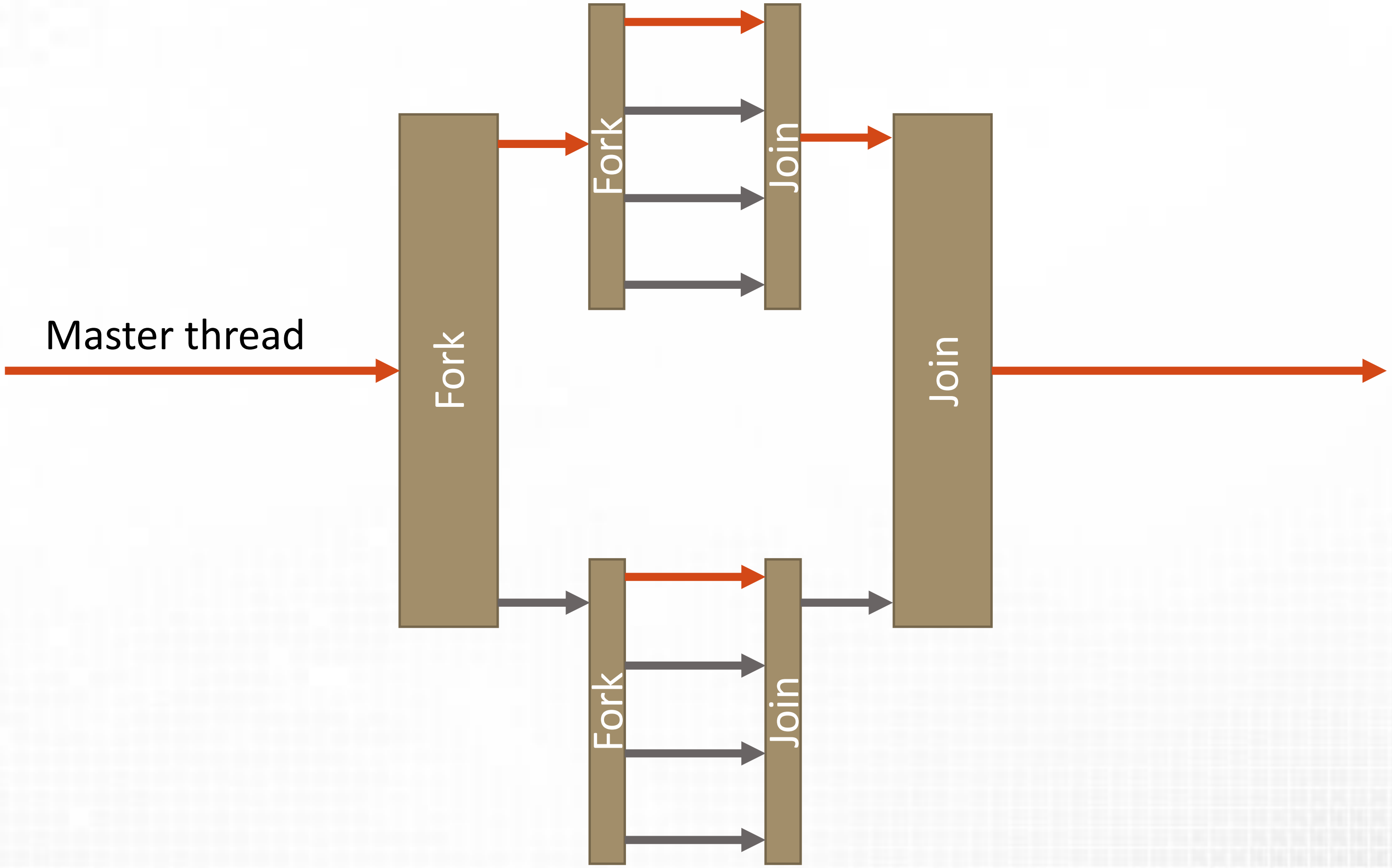
- We can use parallel nesting
  - Nesting is turned off by default so we must use `omp_set_nested()`
  - When inner loop is met each outer thread creates a new team of threads
  - Allows us to expose higher levels of parallelism
    - *Only useful when outer loop does not expose enough*

```
omp_set_nested(1);

#define OUTER_LOOPS 2
#define INNER_LOOPS 4

#pragma omp parallel for
  for (i = 0; i < OUTER_LOOPS; i++){
    int outer_thread = omp_get_thread_num();
      #pragma omp parallel for
        for (j = 0; j < INNER_LOOPS; j++){
          int inner_thread = omp_get_thread_num();
          printf("Hello World (i T=%d j T=%d)\n", outer_thread, inner_thread);
        }
  }
```

```
Hello World (i T=0 j T=0)
Hello World (i T=0 j T=1)
Hello World (i T=0 j T=3)
Hello World (i T=1 j T=2)
Hello World (i T=1 j T=1)
Hello World (i T=1 j T=0)
Hello World (i T=0 j T=2)
Hello World (i T=1 j T=3)
```

Nesting Fork and Join

- Every parallel directive creates a fork (new team)
  - In this case each `omp parallel` is used to fork a new parallel region



Collapse

- Only available in OpenMP 3.0 and later (**not VS2017**)
  - Can automatically collapse multiple loops
  - Loops must not have statements or expressions between them

```
#pragma omp parallel for collapse(2)
  for (i = 0; i < OUTER_LOOPS; i++){
    for (j = 0; j < INNER_LOOPS; j++){
      int thread = omp_get_thread_num();
      printf("Hello World (T=%d)\n", thread);
    }
  }
```

Work around…

```
#pragma omp parallel for
  for (i = 0; i < OUTER_LOOPS* INNER_LOOPS; i++){
    int thread = omp_get_thread_num();
    printf("Hello World (T=%d)\n", thread);
  }
```

- OpenMP Timing
- Parallel Reduction
- Scheduling
- Nesting
- Summary

Clauses usage summary

Clause	Directive: #pragma omp …					
	parallel	for	sections	single	parallel for	parallel sections
if						
private						
shared						
default						
firstprivate						
lastprivate						
reduction						
schedule						
nowait						

Performance

- Remember ideas for general C performance
  - Have good data locality (good cache usage)
  - Combine loops where possible
- Additional performance criteria
  - Minimise the use of barriers
    - Use `nowait` but only if it is safe to do so!
  - Minimise critical sections
    - High overhead. Can you use reduction or atomics?

Summary

- Parallel reduction is very helpful in combining data
  - It will use the OS most efficient method to implement the combination
- Scheduling can be static or dynamic
  - Static is good for fixed work sizes
  - Dynamic is good for varying work sizes
  - Benchmarking is important to find the best approach
- Nested parallelism can improve performance for outer loops with poor parallelism
- To get good performance try to avoid critical sections and barriers

Further reading

- https://software.intel.com/en-us/articles/32-openmp-traps-for-c-developers