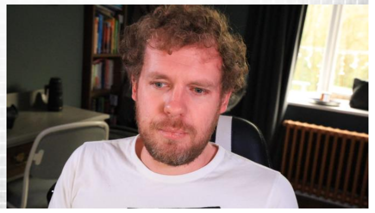
Parallel Computing with GPUs

Advanced OpenMP
Part 1 - Parallel Reduction

The University Of Sheffield.

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

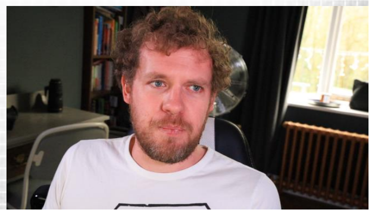


This Lecture (learning objectives)

- ❑Reduction
  - ❑Perform a parallel reduction using the reduction clause
  - ❑Recognise the limitations of the reduction functionality

- ❑What do we need to look out for when considering applying OpenMP to this example?

```
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);
}
```

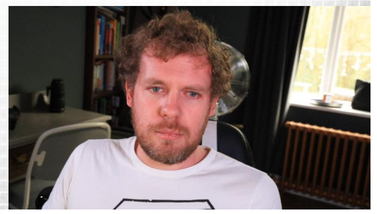


Parallel Reduction

- ❑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…



Reduction clause

```c
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!

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 **in parallel** by;
  - Creating a local (private) copy of the (shared) reduction variable
  - Combining (merging) 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

Summary

- Reduction
  - Perform a parallel reduction using the reduction clause
  - Recognise the limitations of the reduction functionality
- Next Lecture: Scheduling

Parallel Computing with GPUs

Advanced OpenMP Part 2 – Scheduling

The University Of Sheffield.

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

This Lecture (learning objectives)

- ❑Scheduling
  - ❑Compare and contrast different scheduling approaches to understand the benefits and limitations of each
  - ❑Identify how scheduling parameters may impact cache utilisation

Scheduling clause

- ❑OpenMP by default uses static scheduling
  - ❑Static: schedule is determined at compile time
  - ❑`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 env variable

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)

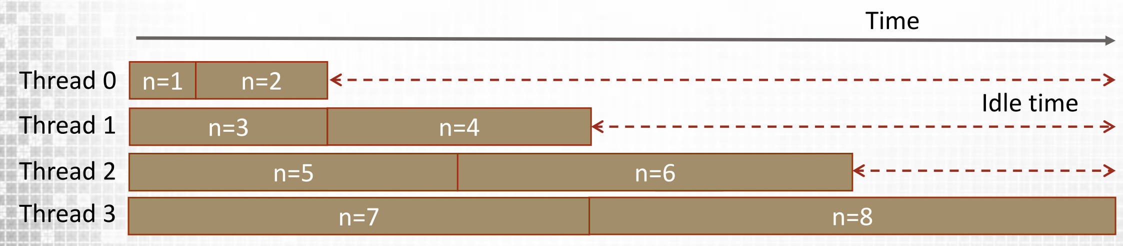
```
int n;
double result = 0.0;
double x = 1.0;

#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));
```

Recursive uneven workload

Scheduling Workload

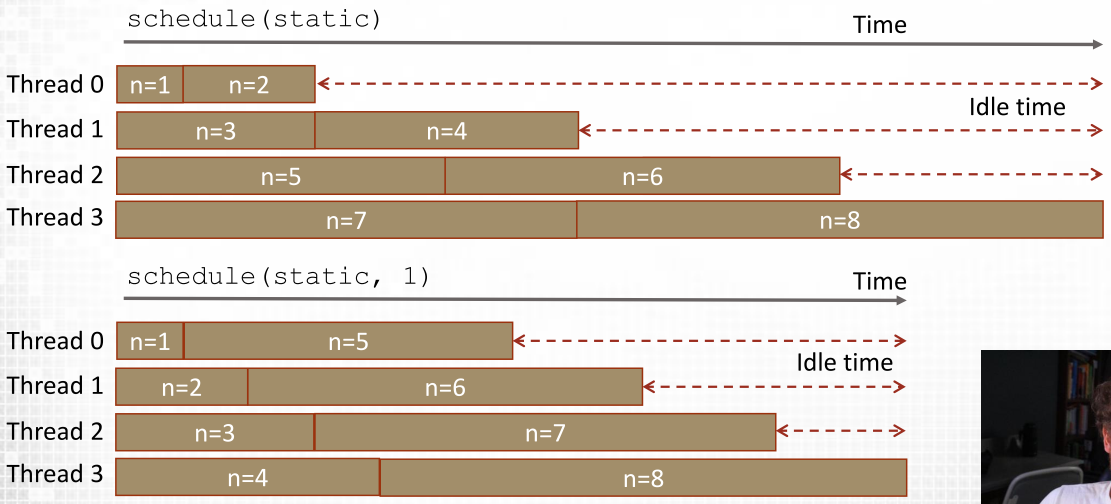
```
long long int fac(int n)
{
  if (n == 0)
    return 1;
  else
    return(n * fac(n - 1));
}
```

- ❑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)`

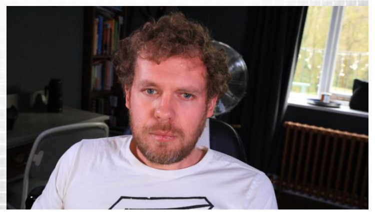
Thread 0	0	1	2	3
Thread 1	4	5	6	7
Thread 2	8	9	10	11
Thread 3	12	13	14	15

Default case

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

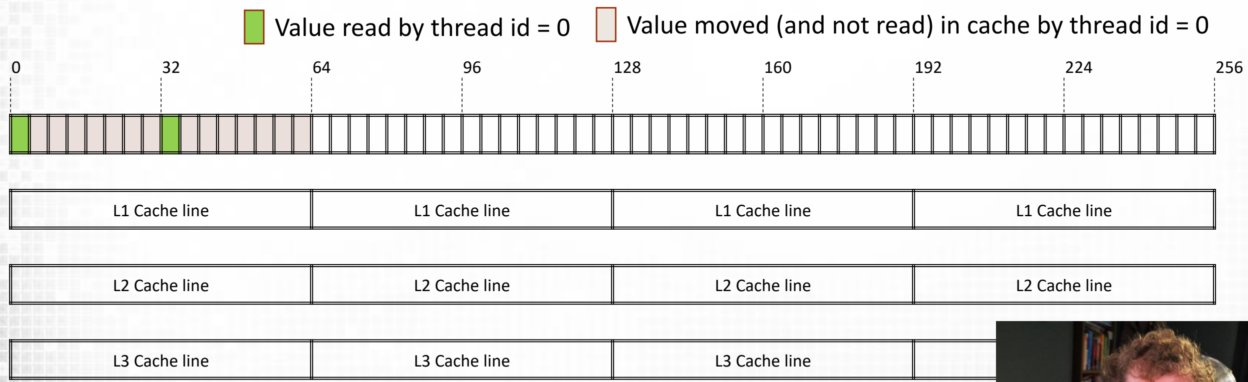
Dynamic and Guided Scheduling

- Dynamic
  - **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
  - **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**



Cache Efficiency

```
#pragma omp parallel for schedule(static,1) num_threads(8)
    for (int i=0; i<64; i++) {
        something(array[i]);
    }
```

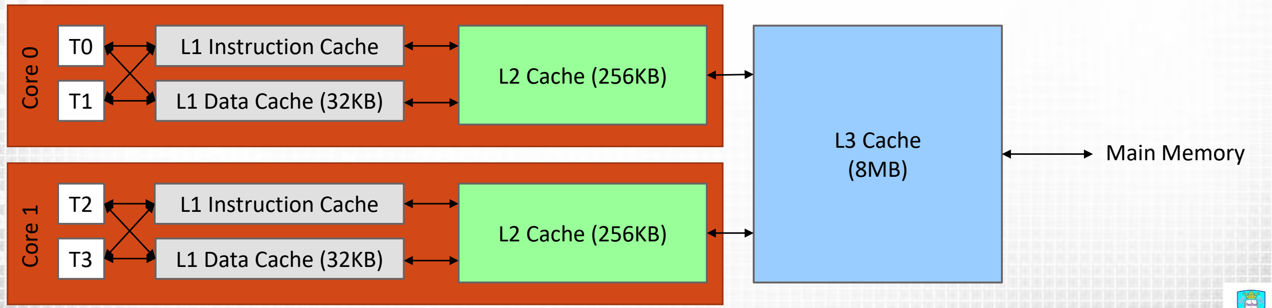


- Chunk size may effect cache utilisation

False Sharing

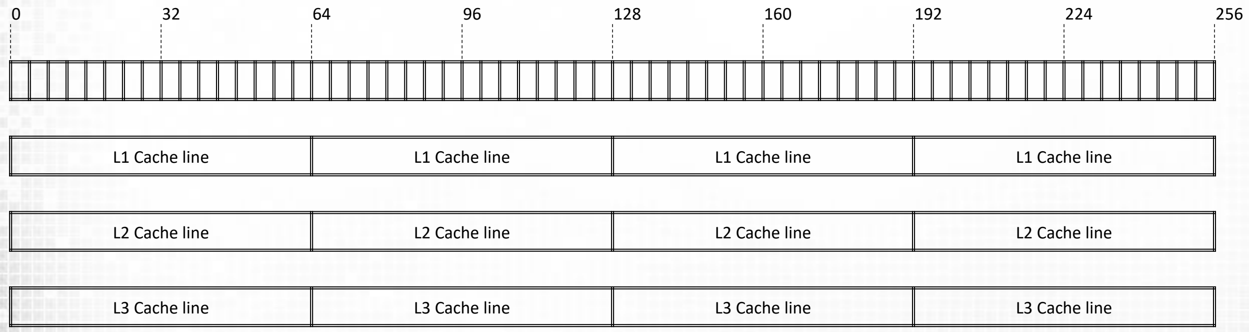
- ❑Changing a single value causes a whole cache line to be invalid
  - ❑Invalid lines must be re-cached
  - ❑Chunk size case effect the amount of times a line is invalid

```
#pragma omp parallel for schedule(static,1)
    for (int i=0; i<64; i++) {
        array[i]++;
    }
```



False Sharing (worked example)

```
#pragma omp parallel for schedule(static,1) num_threads(8)
    for (int i=0; i<64; i++) {
        array[i]++;
    }
```



Summary

- ❑Scheduling
  - ❑Compare and contrast different scheduling approaches to understand the benefits and limitations of each
  - ❑Identify how scheduling parameters may impact cache utilisation

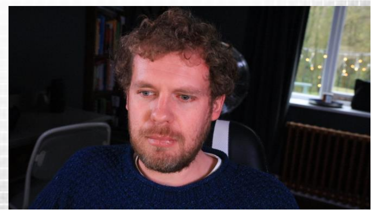
❑Next Lecture: Nesting Loops and OpenMP Summary

Parallel Computing with GPUs

Advanced OpenMP
Part 3 - Nesting and Summary

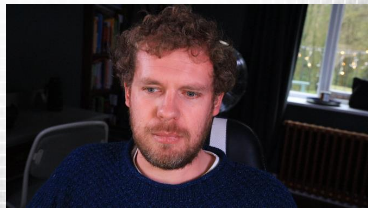
The University Of Sheffield.

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



This Lecture (learning objectives)

- Nesting
  - Operate on nested loops using OpenMP
  - Compare the performance implications of different approaches for nesting
- Summary
  - Classify permitted use of the various OpenMP clauses



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());
    }
}
```

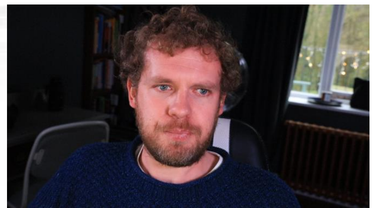
Nesting

- 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



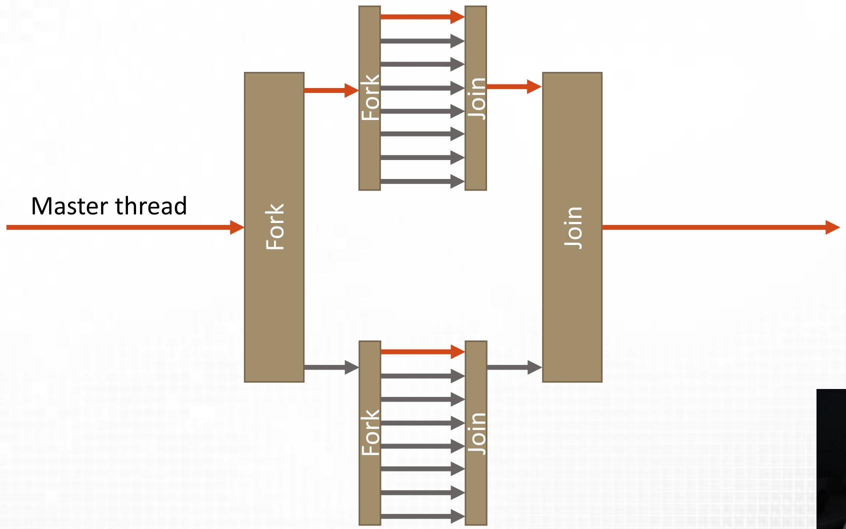
- We can use parallel nesting
  - Nesting is turned off by default so we must use `omp_set_nested()`
  - When inner loop is met each outer thread creates a new team of threads
  - Allows us to expose higher levels of parallelism
    - *Only useful when outer loop does not expose enough parallelism*

```
omp_set_nested(1);

#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 `omp parallel` is used to fork a new parallel region



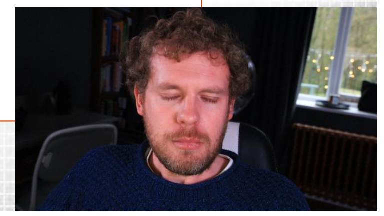
Collapse

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

```c
#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);
    }
  }
```

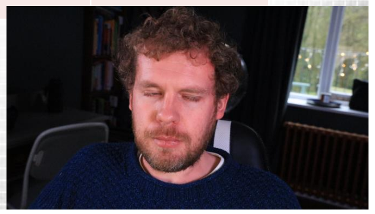
Work around…

```c
#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);
  }
```



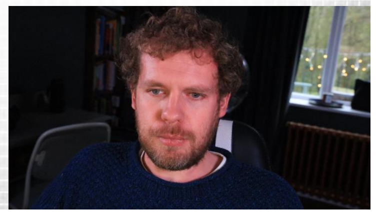
Clauses usage summary

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



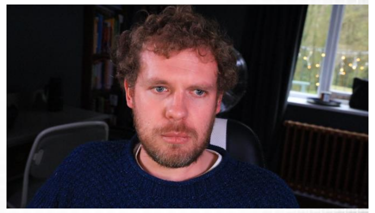
Performance

- ❑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!
  - ❑Especially minimise critical sections
    - ❑High overhead. Can you use reduction or atomics?
    - ❑Benchmark to find best solution
  - ❑Experimentally try out different scheduling approaches and chunk sizes



Summary

- ❑Nesting
  - ❑Operate on nested loops using OpenMP
  - ❑Compare the performance implications of different approaches for nesting
- ❑Summary
  - ❑Classify permitted use of the various OpenMP clauses



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