Durham University
Department of Computer Science

Core I—Introduction to HPC

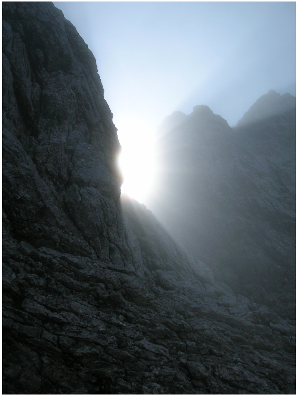
Session V: OpenMP (1/2)

Dr. Weinzierl

Michaelmas term 2019

Outline

Durham University
Department of Computer Science



OpenMP
Thread communication

OpenMP is a . . .

- abbreviation for *Open Multi Processing*
- allows programmers to annotate their C/FORTRAN code with parallelism specs
- portability stems from compiler support
- standard defined by a consortium (`www.openmp.org`)
- driven by AMD, IBM, Intel, Cray, HP, Nvidia, . . .
- old thing currently facing its fourth generation (1st: 1997, 2nd: 2000; 3rd: 2008; 4th: 2013)

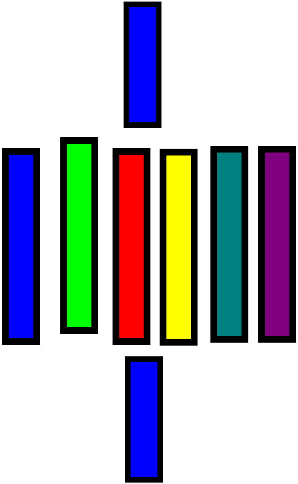
A first example

```
const int size = ...
int a[size];
#pragma omp parallel for
for( int i=0; i<size; i++ ) {
  a[i] = i;
}
```

- OpenMP for C is a preprocessor(pragma)-based extension (annotations, not API)
  - Implementation is up to the compiler (built into recent GNU and Intel compilers; before additional precompiler required)
  - Implementations internally rely on libraries (such as pthreads)
  - Annotations that are not understood should be ignored (don't break old code)
- Syntax conventions
  - OpenMP statements always start with `#pragma omp`
  - Before GCC 4, source-to-source compiler replaced only those lines
- OpenMP originally written for BSP/Fork-Join applications
- OpenMP usually abstracts from machine characteristics (number of cores, threads, ...)

A first example

Durham University
Department of Computer Science



```
const int size = ...
int a[size];
#pragma omp parallel for
for( int i=0; i<size; i++ ) {
  a[i] = i;
}
```

Compilation and execution:

- GCC: `-fopenmp`
- Intel: `-openmp`
- Some systems require `#include <omp.h>`
- Set threads: `export OMP_NUM_THREADS=2`

What happens—usage

Durham University
Department of Computer Science

```
>icc -fopenmp test.cpp
test.cpp(22): (col. 1) remark: \
OpenMP DEFINED REGION WAS PARALLELIZED.

>export OMP_NUM_THREADS=4
>./a.out
```

```
const int size = ...
int a[size];
#pragma omp parallel for
for( int i=0; i<size; i++ ) {
  a[i] = i;
}
```

- Code runs serially until is hits the pragma
- System splits up for loop into chunks (we do not yet know how many)
- Chunks then are deployed among the four threads
- All threads wait until loop has terminated on all threads, i.e. it synchronises the threads $\Rightarrow$ bulk synchronous processing (bsp)
- Individual threads may execute different instructions from the loop concurrently (designed for MIMD machines)

OpenMP execution model

Durham University
Department of Computer Science

Explicit scoping:

```
#pragma omp parallel for
{
  for( int i=0; i<size; i++ ) {
    a[i] = i;
  }
}
```

Implicit scoping:

```
#pragma omp parallel for
for( int i=0; i<size; i++ ) {
  a[i] = i;
}
```

- Master thread vs. worker threads
- Fork/join execution model with implicit synchronisation (barrier) at end of scope
- Nested parallel loops possible (though sometimes very expensive)
- Shared memory paradigm (everybody may access everything)

Some OMP functions

Durham University
Department of Computer Science

```
int numberOfThreads = omp_get_num_procs();

#pragma omp parallel for
for( int i=0; i<size; i++ ) {

  int thisLineCodeIsRunningOn = omp_get_thread_num();

}
```

- No explicit initialisation OpenMP required in source code
- Abstract from threads—setting thread count is done by OS
- Error handling (to a greater extent) not specified by standard
- Functions almost never required (perhaps for debugging)

Parallel loops in action

Durham University
Department of Computer Science

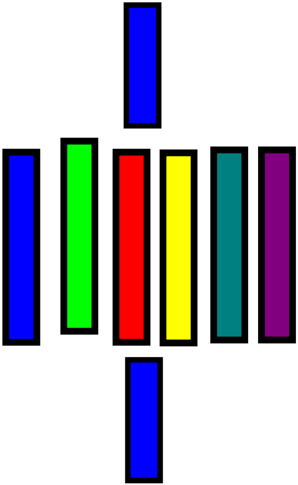
```
#pragma omp parallel
{
  for( int i=0; i<size; i++ ) {
    a[i] = a[i]*2;
  }
}


#pragma omp parallel for
{
  for( int i=0; i<size; i++ ) {
    a[i] = a[i]*2;
  }
}
```

Observations:

- Global loop count is either `size` or *threads*·`size`
- We run into race conditions
- These result from dependencies (read-write, write-read, write-write) on `a[i]`

Parallel loops and BSP

Durham University
Department of Computer Science



```
#pragma omp parallel
{
  #pragma omp for
  for( int i=0; i<size; i++ ) {
    a[i] = a[i]*2;
  }
}
```

- `omp parallel` triggers fork technically, i.e. spawns the threads
- `for` decomposes the iteration range (logical fork part)
- `omp parallel for` is a shortcut
- BSP's join/barrier is done implicitly at end of the parallel section

Requirements for parallel loops

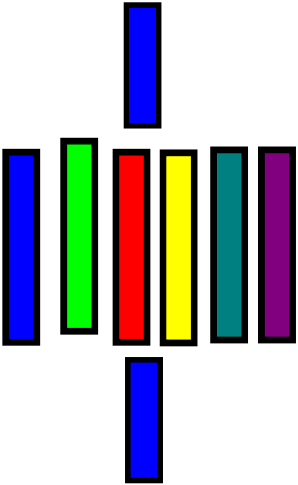
Durham University
Department of Computer Science

```
#pragma omp parallel for
{
  for( int i=0; i<size; i++ ) {
    a[i] = a[i]*2;
  }
}
```

- Loop has to follow plain initialisation-condition-increment pattern:
  - Only integer counters
  - Only plain comparisons
  - Only increment and decrement (no multiplication or any arithmetics)
- Loop has be countable (otherwise splitting is doomed to fail).
- Loop has to follow single-entry/single-exit pattern.

Data consistency in OpenMP's BSP model

Durham University
Department of Computer Science



```
#pragma omp parallel
{
  #pragma omp for
  for( int i=0; i<size; i++ ) {
    a[i] = a[i]*2;
  }
}
```

- No assumptions which statements run technically concurrent
- Shared memory without any consistency model
- No inter-thread communication (so far)

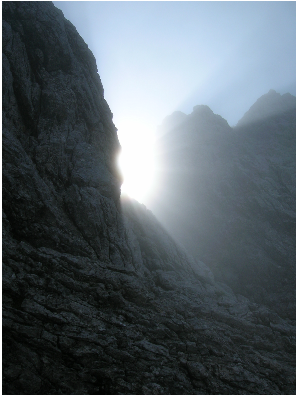
⇒ Data consistency is developer's responsibility

Concept of building block: OpenMP Introduction

Durham University
Department of Computer Science

- Content
  - OpenMP syntax basics
  - OpenMP runtime model
  - OpenMP functions
- Expected Learning Outcomes
  - The student can *translate* and use an application with OpenMP support
  - The student can *explain* with the OpenMP execution model

Outline



OpenMP
Thread communication

Thread communication

Durham University
Department of Computer Science

We distinguish two different communication types:

- Communication through the join
- Communication inside the BSP part (not “academic” BSP)

Critical section: Part or code that is ran by at most one thread at a time.

Reduction: Join variant, where all the threads reduce a value into one single value.

⇒ Reduction maps a vector of $(x_0, x_1, x_2, x_3), \ldots, x_{p-1}$ onto one value x, i.e. we have a all-to-one data flow

⇒ Inter-thread communication realised by data exchange through shared memory

⇒ Fork can be read as one-to-all information propagation (done implicitly by shared memory)

Critical sections

Durham University
Department of Computer Science

```
#pragma omp critical (mycriticalsection)
{
  x *= 2;
}
...
#pragma omp critical (anothersection)
{
  x *= 2;
}
...
#pragma omp critical (mycriticalsection)
{
  x /= 2;
}
```

- Name is optional (default name i.e. does not block with other sections with a name)
- Single point of exit policty ⇒ return, break, ... not allowed within critical section
- For operations on built-in/primitive data types, an *atomic operation* is usually the better, i.e. faster choice (cmp. CCS module)

Recap: Requirements for parallel loops


```
#pragma omp parallel for
{
  for( int i=0; i<size; i++ ) {
    a[i] = a[i]*2;
  }
}
```

- Loop has to follow plain initialisation-condition-increment pattern:
  - Only integer counters
  - Only plain comparisons
  - Only increment and decrement (no multiplication or any arithmetics)
- Loop has be countable (otherwise splitting is doomed to fail).
- Loop has to follow single-entry/single-exit pattern.
- Loop copies all share the memory.

Consistency observation:

- All attributes are shared
- Besides the actual loop counter (otherwise splitting wouldn't work)

$\Rightarrow$ There has to be support for non-shared data

The shared default clause


```
double result = 0;
#pragma omp parallel for
for( int i=0; i<size; i++ ) {
  result += a[i] * b[i]; // race, but doesn't bother us here
}
```

```
double result = 0;
#pragma omp parallel for shared(result)
for( int i=0; i<size; i++ ) {
  result += a[i] * b[i];
}
```

- By default, all OpenMP threads share all variables, i.e. variables declared outside are visible to threads
- This sharing can be made explicit through the clause `shared`
- Explicit shared annotation is kind of good style (improved readability)

Thread-local variables

Durham University
Department of Computer Science

```
double result = 0;
#pragma omp parallel for
for( int i=0; i<size; i++ ) {
  double result = 0.0;
  result += a[i] * b[i];
}
```

Without OpenMP pragma:

- C/C++ allows us to “redefine” variable in inner scope
- Hides/shadows outer `result`
- We may not forget the second initialisation; otherwise garbage

With OpenMP pragma:

- OpenMP introduces (concurrent) scope of its own
- Scope-local variables are thread-local variables
- These are *not* shared

shared vs. private clauses

Durham University
Department of Computer Science

```
double result = 0;
#pragma omp parallel for private(result)
for( int i=0; i<size; i++ ) {
  result += a[i] * b[i];
}
```

- `private` is the counterpart of `shared`, i.e. each thread works on its own copy
- Basically, the copying is similar to the following fragment:

```
double result = 0;
#pragma omp parallel
{
  double result;
  #pragma omp for
  for( int i=0; i<size; i++ ) {
    result += a[i] * b[i];
  }
}
```

⇒ In this example, `result` within thread is not initialised (garbage)!

⇒ In this example, data within `result` is lost throughout join!

Example

```
int x;
x=40;
#pragma omp parallel for private (x)
for (int i=0; i<10; i++) {
 x = i; // try to remove this line
 std::cout << "x=" << x << " on thread "
           << omp_get_thread_num() << std::endl;
}
std::cout << "x=" << x << std::endl;
```

Example


```
int x;
x=40;
#pragma omp parallel for private (x)
for (int i=0; i<10; i++) {
 x = i;  // try to remove this line
 std::cout << "x=" << x << " on thread "
           << omp_get_thread_num() << std::endl;
}
std::cout << "x=" << x << std::endl;
```

Observations:

- If we comment out `x=i`, `x` is not properly initialised.
- `x` in the last line always equals 40.

Now remove initialisation and write `firstprivate(x)`.

Copy policies

Durham University
Department of Computer Science

- `default` Specifies default visibility of variables
- `firstprivate` Variable is private, but initialised with value from surrounding
- `lastprivate` Variable is private, but value of very last iteration is copied over to outer variable

Case study: scalar product

Serial starting point:

```
double result = 0;
for( int i=0; i<size; i++ ) {
  result += a[i] * b[i];
}
```

Parallel variant:

```
double result = 0.0;
#pragma omp parallel
{
  double myResult = 0.0;
  #pragma omp for
  for( int i=0; i<size; i++ ) {
    myResult += a[i] * b[i];
  }
  #pragma omp critical
  result += myResult;
}
```

Observations:

- Avoid excessive synchronisation
- Type of operation is called *reduction* (as defined before)
- We may not use `result` to accumulate because of races
- We may not hide `result` as we then loose access to outer variable

Concept of building block

Durham University
Department of Computer Science

- Content
  - Introduce three types of data flow/usage of shared memory: making memory available to all threads throughout fork, sharing data throughout computation, reducing data at termination
  - Introduce private variables
  - Study semantics and usage of critical sections
- Expected Learning Outcomes
  - The student *can use* critical sections
  - The student *knows difference* between private and shared variables
  - The student can *identify race conditions* and *resolve* them through critical sections for given code snippet