4️⃣

OpenMP 1

Files	01 IntroductionToOpenMP.pdf
Link	
Status	Done
Type	Lecture

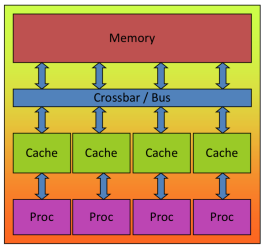
OpenMP vs MPI

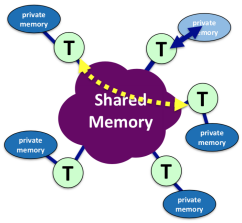
MPI: message passing, every process is working on its own memory space in isolation from others

OpenMP: all the threads have access to all the data (same memory space)

It is not uncommon for an application to utilise both.

OpenMP Intro





- All threads have access to the same globally shared memory
- Each thread has its private memory as well
- Data transfer happens in the shared memory and is 100% transparent to everyone

Hello World

- Exactly one entry and one exit point to the region
- Branching in and out is not allowed
- Terminating the program inside the block is not allowed
- Number of threads is specified as an env variable or in the pragma
- (Similar to MPI) can not make any assumptions about which threads will run in which schedule as the OS schedules them

```
#pragma omp parallel
```

All threads executes the code on the block

(but no partition/division of work need `for` or another directive)

Functions

Aa Definition	Description
omp_set_num_threads(int)	The specified number of threads will be used for the parallel region encountered next.
int omp_get_num_threads	Returns the number of threads in the current team.

Definition	Description
int omp_get_thread_num()	Returns the number of the calling thread in the team, the Master has always the id 0.

For loop

```
#pragma omp parallel for
```

The work in the for loop is distributed over the threads, dividing the number of iterations over the threads (0 does 0-24, 2 does 25-49 etc.)

This is called **worksharing**

> 📌 Not all for loops can be parallelised:
> Simple test is will the execution backwards produce a different result (but this test is not enough)

Influence scheduling

Definition	Description	Comments
schedule(static [, chunk])	Divide in chunk size and assign in round robin	Works well if all the objects in the loop will take roughly the same time
schedule(dynamic [, chunk])	Default chunk=1, assigned when a worker thread is done with current task	Why not default? Because you want to organise the data as to profit from the cache of neighbouring calculations
schedule(guided [, chunk])	Similar to dynamic, but block size starts with implementation-defined value, then is decreased exponentially down to chunk.	This is to decrease overhead at the beginning when probably the chunks will take similar time since there is many anyways then once it matters and its almost done its more granular

Synchronisation

Constructs

Definition	Description	Comments	Example
#pragma omp critical (name)	Executed by all threads, but only 1 at a time		
#pragma omp barrier	All threads wait until the whole team has reached the barrier	All worksharing constructs have an implicit barrier at the end	
#pragma omp single [clause]	One thread only does it, up to runtime which one	useful for I/O or Memory allocation and deallocation, etc. (in general: setup work)	
#pragma omp master [clause]	Executed only by the master thread of a team	no worksharing construct and does not contain an implicit barrier at the end	
#pragma omp sections	Structured blocks that are to be distributed among and executed by the team of threads		
#pragma omp parallel for ordered	Specify order		
#pragma omp taskwait	Waits on the completion of child tasks (not all descendants, only the immediate children)		
#pragma omp task if(expression)	True: task creation proceeds as usual, False: the task has to be executed immediately (undeferred task)		

Data Scoping

Dont really understand it but discusses which variables are private vs shared

→ *private*-list and *shared*-list on Parallel Region

→ *private*-list and *shared*-list on Worksharing constructs

→ General default is *shared* for Parallel Region, *firstprivate* for Tasks.

→ Loop control variables on *for*-constructs are *private*

Tasks are introduced later

→ Non-static variables local to Parallel Regions are *private*

→ *private*: A new uninitialized instance is created for the task or each thread executing the construct

→ *firstprivate*: Initialization with the value before encountering the construct

→ *lastprivate*: Value of last loop iteration is written back to Master

→ Static variables are *shared*

■ **Global / static variables can be privatized with the *threadprivate* directive**

→ One instance is created for each thread

→ Before the first parallel region is encountered

→ Instance exists until the program ends

→ Does not work (well) with nested parallel region

→ Based on thread-local storage (TLS)

→ TlsAlloc (Win32-Threads), pthread_key_create (Posix-Threads), keyword __thread (GNU extension)

Really: try to avoid the use of threadprivate and static variables!

C/C++

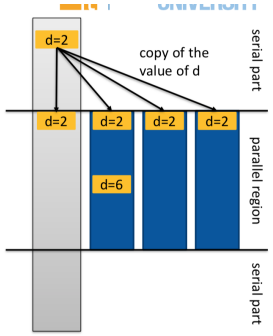
```
int i;
#pragma omp threadprivate(i)
```

Fortran

```
SAVE INTEGER :: i
!$omp threadprivate(i)
```

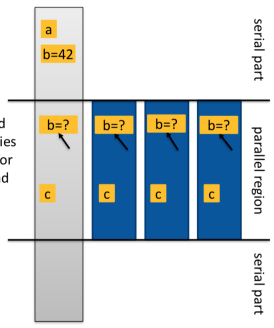


```
int d=2;
.
.
#pragma omp parallel firstprivate(d)
{
#pragma omp single
{d=6;}
.
}
.
.
.
```




```
int a
int b = 42;
.
#pragma omp parallel shared(a) //
  private(b)
{
.
int c;
.
}
.
.
.
```

uninitialized private copies of b and c for every thread



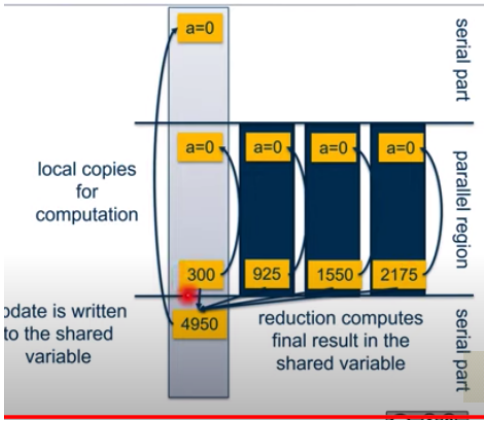
Reduction

`reduction(operator:list)`

Creates a private variable for each of the processes then calculates the final answer based on the individual answers.

C/C++

```
#pragma omp parallel for reduction(+:s)
for(i = 0; i < 99; i++)
{
    s = s + a[i];
}
```



▼ Operators

+

-

*

&

|

&&

||

^

min

max

Tasking

Independent unit of work that can be executed straight away when it is encountered or later by another thread

```
#pragma omp task [clauses]
```

Keeping granularity vs overhead in mind is important, there is an overhead to task creation and retrieval.

Important technique to avoid this overhead: cut off strategy, once the "n" is less than a number, probably more efficient to do serially.

Data Scoping in Tasks

```
 1  int a = 1;
 2  void foo()
 3  {
 4      int b = 2, c = 3;
 5      #pragma omp parallel private(b)
 6      {
 7          int d = 4;
 8          #pragma omp task
 9          {
10              int e = 5;
11
12              // Scope of a: shared,            value of a: 1
13              // Scope of b: firstprivate,      value of b: 0 / undefined
14              // Scope of c: shared,            value of c: 3
15              // Scope of d: firstprivate,      value of d: 4
16              // Scope of e: private,           value of e: 5
17  } } }
```