Getting started with OpenMP
## OpenMP – overview

### Wednesday

<table>
<thead>
<tr>
<th>Time</th>
<th>Session</th>
</tr>
</thead>
<tbody>
<tr>
<td>13:00-13:45</td>
<td>Introduction to OpenMP</td>
</tr>
<tr>
<td>13:45-14:30</td>
<td>Exercises</td>
</tr>
<tr>
<td><strong>14:30-14:45</strong></td>
<td><strong>Coffee break</strong></td>
</tr>
<tr>
<td>14:45-15:30</td>
<td>Thread synchronization</td>
</tr>
<tr>
<td>15:30-16:30</td>
<td>Exercises</td>
</tr>
<tr>
<td><strong>16:30-17:00</strong></td>
<td><strong>Wrap up and further topics</strong></td>
</tr>
</tbody>
</table>
BASIC CONCEPTS: PROCESS AND THREADS
Processes and threads

Process
- Independent execution units
- Have their own state information and *own address spaces*

Thread
- A single process may contain multiple threads
- Have their own state information, but share the address space of the process
Processes and threads

**Process**
- Long-lived: spawned when parallel program started, killed when program is finished
- Explicit communication between processes

**Thread**
- Short-lived: created when entering a parallel region, destroyed (joined) when region ends
- Communication through shared memory
WHAT IS OPENMP?
OpenMP

- A collection of compiler directives and library routines that can be used for multi-threaded shared memory parallelization
- Fortran 77/9X/03 and C/C++ are supported
- Most recent version of the standard is 4.0 (July 2013)
  - Includes support for attached devices
  - Includes thread affinity support
  - Not all compilers do not yet support the newest standard
Why would you want to learn OpenMP?

- OpenMP parallelized program can be run on your many-core workstation or on a node of a cluster
- Enables one to parallelize one part of the program at a time
  - Get some speedup with a limited investment in time
  - Efficient and well scaling code still requires effort
- Serial and OpenMP versions can easily coexist
- Hybrid programming
Three components of OpenMP

- Compiler directives, i.e., *language extensions* for shared memory parallelization
  - Syntax: directive, construct, clauses
    - C/C++: `#pragma omp parallel shared(data)`
    - Fortran: `!$omp parallel shared(data)`

- Runtime library routines (Intel: libiomp5, GNU: libgomp)
  - Conditional compilation to build serial version

- Environment variables
  - Specify the number of threads, thread affinity, etc.
OpenMP directives

Sentinels precede each OpenMP directive
- C/C++: `#pragma omp`
- Fortran free form: `!$omp`

Old Fortran programs may still use fixed form formatting
- Sentinel: `c$omp`
- Space in sixth column begins directive
- No space depicts continuation line
Compiling an OpenMP program

Compilers that support OpenMP usually require an option that enables the feature

- PGI: `-mp[=nonuma,align,allcores,bind]`
- Cray: `-h omp` (on by default, `-h noomp` disables)
- GNU: `-fopenmp`
- Intel: `-openmp, -qopenmp`

Without these options a serial version is compiled!
Conditional compilation with \_OPENMP macro:

```c
#ifdef _OPENMP
   Thread specific code
#else
   Serial code
#endif
```

Fortran free form guard sentinels: !$
- Fortran fixed form guard sentinels: !\$ \*\$ c\$
Example: Helloworld with OpenMP

```
program hello
    use omp_lib
    integer :: omp_rank

    !$omp parallel private(omp_rank)
    omp_rank = omp_get_thread_num()
    print *, 'Hello world! by &
        thread ', omp_rank
    !$omp end parallel
end program hello
```

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char argv[]){
    int omp_rank;

    #pragma omp parallel private(omp_rank)
    {
        omp_rank = omp_get_thread_num();
        printf("Hello world! by 
            thread %d", omp_rank);
    }
}
```

```
> ftn -homp omp_hello.f90 -o omp
> aprun -n 1 -d 4 -e OMP_NUM_THREADS=4
./omp
Hello world! by thread  0
Hello world! by thread  2
Hello world! by thread  3
Hello world! by thread  1
```

```
> cc -homp omp_hello.c -o omp
> aprun -n 1 -d 4 -e OMP_NUM_THREADS=4
./omp
Hello world! by thread  2
Hello world! by thread  3
Hello world! by thread  0
Hello world! by thread  1
```
PARALLEL REGIONS AND DATA SHARING
Defines a parallel region

C/C++:

```
#pragma omp parallel [clauses]
```

Fortran:

```
!$omp parallel [clauses]
```

- Prior to it only one thread, master
- Creates a team of threads: master+slave threads
- At end of the block is a barrier and all shared data is synchronized
How do the threads interact?

Because of the shared address space threads can *communicate* using *shared* variables.

Threads often need some private work space together with shared variables:
- For example the index variable of a loop.

Visibility of different variables is defined using *data-sharing clauses* in the parallel region definition.
Default storage

Most variables are *shared* by default

Global variables are shared among threads
- C: `static` variables, file scope variables
- Fortran: `SAVE` and `MODULE` variables, `COMMON` blocks
- Both: dynamically allocated variables

Private by default:
- Stack variables of functions called from parallel region
- Automatic variables within a block
Default storage

```c
int main(void) {
    int B[2];
    #pragma omp parallel
do_things(B);
    return 0;
}

void do_things(int *var) {
    double wrk[10];
    static int status;
    ...
}
```

Shared between threads

Private copy on each thread
omp parallel: data-sharing clauses

private(list)

- Private variables are stored in the private stack of each thread
- Undefined initial value
- Undefined value after parallel region

firstprivate(list)

- Same as private variable, but with an initial value that is the same as the original objects defined outside the parallel region
omp parallel: data-sharing clauses

shared(list)

- All threads can write to, and read from a shared variable
- Variables are shared by default

Race condition = a thread accesses a variable while another writes into it
omp parallel: data-sharing clauses

default(private/shared/none)

- Sets default for variables to be shared, private or not defined
- In C/C++ default(private) is not allowed
- default(none) can be useful for debugging as each variable has to be defined manually
WORK SHARING CONSTRUCTS
Work sharing

Parallel region creates an “Single Program Multiple Data” instance where each thread executes the same code

How can one split the work between the threads of a parallel region?

– Loop construct
– Single/Master construct
– Sections
– Task construct (in OpenMP 3.0 and above)
Loop construct

- Directive instructing compiler to share the work of a loop
  C/C++: `#pragma omp for [clauses]`
  Fortran: `!$omp do [clauses]`
  - The construct must followed by a loop construct. To be active it must be inside a parallel region
  - Combined construct with parallel:
    `#pragma omp parallel for / !$omp parallel do`

- Loop index is private by default

- Work sharing can be controlled with the `schedule` clause
Restrictions of loop construct

For loops in C/C++ are very flexible, but loop construct can only be used on limited set of loops of a form

```
for(init ; var comp a ; incr)
```

where

- `init` initializes the loop variable `var` using an integer expression
- `comp` is one of `<`, `<=`, `>`, `>=` and `a` is an integer expression
- `incr` increments `var` by an integer amount standard operator
Thread synchronization
REDUCTIONS
Race condition

Race conditions take place when multiple threads read and write a variable simultaneously, for example:

```
 asum = 0.0d0
 !$OMP PARALLEL DO SHARED(x,y,n,asum) PRIVATE(i)
  do i = 1, n
       asum = asum + x(i)*y(i)
  end do
 !$OMP END PARALLEL DO
```

Random results depending on the order the threads access `asum`

We need some mechanism to control the access
Reductions

Summing elements of array is an example of reduction operation

\[ S = \sum_{j=1}^{N} A_j = \sum_{j=1}^{\left\lfloor \frac{N}{2} \right\rfloor} A_j + \sum_{j=\left\lceil \frac{N}{2} \right\rceil+1}^{N} A_j = B_1 + B_2 = \sum_{j=1}^{2} B_j \]

OpenMP provides support for common reductions within parallel regions and loops with the *reduction* -clause
reduction(operator:list)
  – Performs reduction on the (scalar) variables in list
  – Private reduction variable is created for each thread’s partial result
  – Private reduction variable is initialized to operator’s initial value
  – After parallel region the reduction operation is applied to private variables and result is aggregated to the shared variable
# Reduction operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>*</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operator</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>.AND.</td>
<td>.true.</td>
</tr>
<tr>
<td>.OR.</td>
<td>.false.</td>
</tr>
<tr>
<td>.NEGV.</td>
<td>.false.</td>
</tr>
<tr>
<td>.IEOR.</td>
<td>0</td>
</tr>
<tr>
<td>.IOR.</td>
<td>0</td>
</tr>
<tr>
<td>.IAND.</td>
<td>All bits on</td>
</tr>
<tr>
<td>.EQV.</td>
<td>.true.</td>
</tr>
<tr>
<td>MIN</td>
<td>max pos.</td>
</tr>
<tr>
<td>MAX</td>
<td>min neg.</td>
</tr>
</tbody>
</table>

**C/C++ only**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>~0</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>0</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Fortran only**
Race condition example revisited

```c
!$OMP PARALLEL DO SHARED(x,y,n) PRIVATE(i) REDUCTION(+:asum)
  do i = 1, n
    asum = asum + x(i)*y(i)
  end do
!$OMP END PARALLEL DO
```
EXECUTION CONTROLS AND SYNCHRONIZATION
Execution controls

Sometimes a part of parallel region should be executed only by the master thread or by a single thread at time

– IO, initializations, updating global values, etc.
– Remember the synchronization!

OpenMP provides clauses for controlling the execution of code blocks
Execution control constructs

barrier

- When a thread reaches a barrier it only continues after all the threads in the same thread team have reached it
  - Each barrier must be encountered by all threads in a team, or none at all
  - The sequence of work-sharing regions and barrier regions encountered must be same for all threads in team
- Implicit barrier at the end of: do, parallel, sections, single, workshare
Execution control constructs

**master**

- Specifies a region that should be executed only by the master thread
- Note that there is no implicit barrier at end

**single**

- Specifies that a region should be executed only by a single (arbitrary) thread
- Other threads wait (implicit barrier)
Execution control constructs

critical[(name)]
  – A section that is executed by only one thread at a time
  – Optional name specifies global identifier for critical section
  – Unnamed critical sections are treated as the same section

flush[(name)]
  – Synchronizes the memory of all threads
  – Implicit flush at
    ▪ All explicit and implicit barriers
    ▪ Entry to / exit from critical section and lock routines
Execution control constructs

**atomic**

- Strictly limited construct to update a single value, cannot be applied to code blocks
- Only guarantees atomic update, does not protect function calls
- Can be faster on hardware platforms that support atomic updates
Example: reduction using critical section

```fortran
!$OMP PARALLEL SHARED(x,y,n,asum) PRIVATE(i, psum)
psum = 0.0d
!$OMP DO
do i = 1, n
  psum = psum + x(i)*y(i)
end do
!$OMP END DO
!$OMP CRITICAL(dosum)
asum = asum + psum
!$OMP END CRITICAL(dosum)
!$OMP END PARALLEL DO
```
Example: updating global variable

```c
int global_max = 0;
int local_max = 0;
#pragma omp parallel firstprivate(local_max) private(i)
{
#pragma omp for
    for (i=0; i < 100; i++) {
        local_max = MAX(local_max, a[i]);
    }
#pragma omp critical(domax)
    global_max = MAX(local_max, global_max);
}
```
OpenMP and execution environment

OpenMP provides several means to interact with the execution environment. These operations include
– Setting the number of threads for parallel regions
– Requesting the number of CPUs
– Changing the default scheduling for work-sharing clauses
– etc.

Improves portability of OpenMP programs between different architectures (number of CPUs, etc.)
Environment variables

- OpenMP standard defines a set of environment variables that all implementations have to support.

- The environment variables are set before the program execution and they are read during program start-up.
  - Changing them during the execution has no effect.

- We have already used **OMP_NUM_THREADS**
Runtime functions

Runtime functions can be used either to read the settings or to set (override) the values.

Function definitions are in

- C/C++ header file `omp.h`
- `omp_lib` Fortran module (`omp_lib.h` header in some implementations)

Two useful routines for distributing work load:

- `omp_get_num_threads()`
- `omp_get_thread_num()`
Parallelizing a loop with library functions

```c
#pragma omp parallel private(i,nthrds,thr_id)
{
    nthrds = omp_get_num_threads();
    thrd_id = omp_get_thrd_num();
    for (i=thrd_id; i<n; i+=nthrds) {
        ...
    }
}
```
FURTHER TOPICS
OpenMP programming best practices

- Maximize parallel regions
  - Reduce fork-join overhead, e.g. combine multiple parallel loops into one large parallel region
  - Potential for better cache re-usage

- Parallelize outermost loops if possible
  - Move \texttt{PARALLEL DO} construct outside of inner loops

- Reduce access to shared data
  - Possibly make small arrays private
Things that we did not cover

- Other work-sharing clauses
  - task
  - sections, workshare, simd (OpenMP 4.0)
  - teams, distribute (both OpenMP 4.0)

- More advanced ways to reduce synchronization overhead with nowait and flush
- threadprivate, copyin, cancel
- Support for attached devices with OpenMP 4.0 target
OpenMP summary

OpenMP is an API for thread-based parallelization
- Compiler directives, runtime API, environment variables
- Relatively easy to get started but specially efficient and/or real-world parallelization non-trivial

Features touched in this intro
- Parallel regions, data-sharing attributes
- Work-sharing and scheduling directives
Web resources

- OpenMP homepage
  http://openmp.org/

- Good online tutorial:
  https://computing.llnl.gov/tutorials/openMP/

- More online tutorials:
  http://openmp.org/wp/resources/#Tutorials