EPL372
Lab Exercise 5:
Introduction to OpenMP

References:

https://computing.llnl.gov/tutorials/openMP/

http://openmp.org/wp/openmp-specifications/

http://openmp.org/mp-documents/OpenMP-4.0-C.pdf

http://openmp.org/mp-documents/OpenMP4.0.0.Examples.pdf

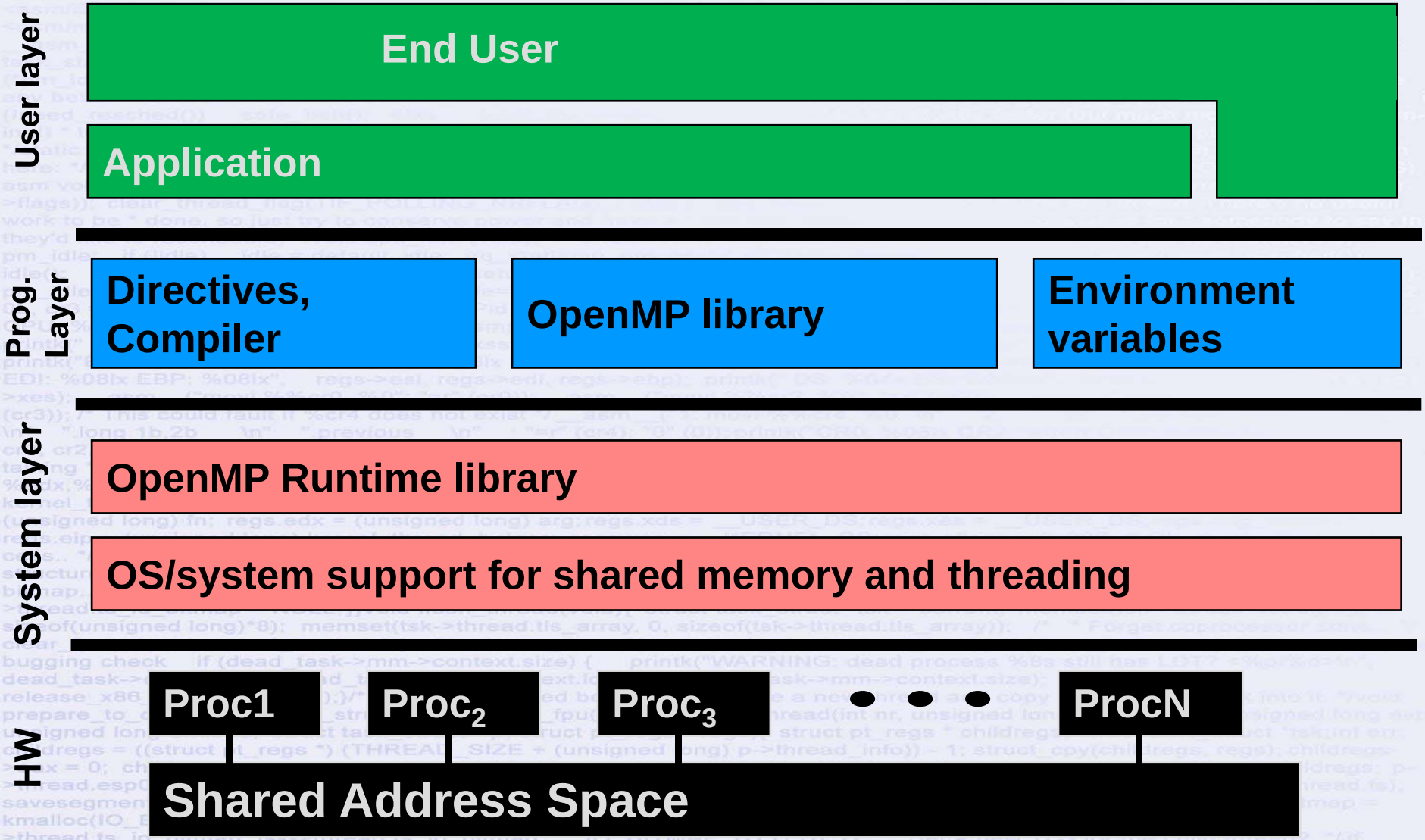
http://www.training.prace-ri.eu/uploads/tx_pracetmo/omp_tutorial2.pdf

What is OpenMP

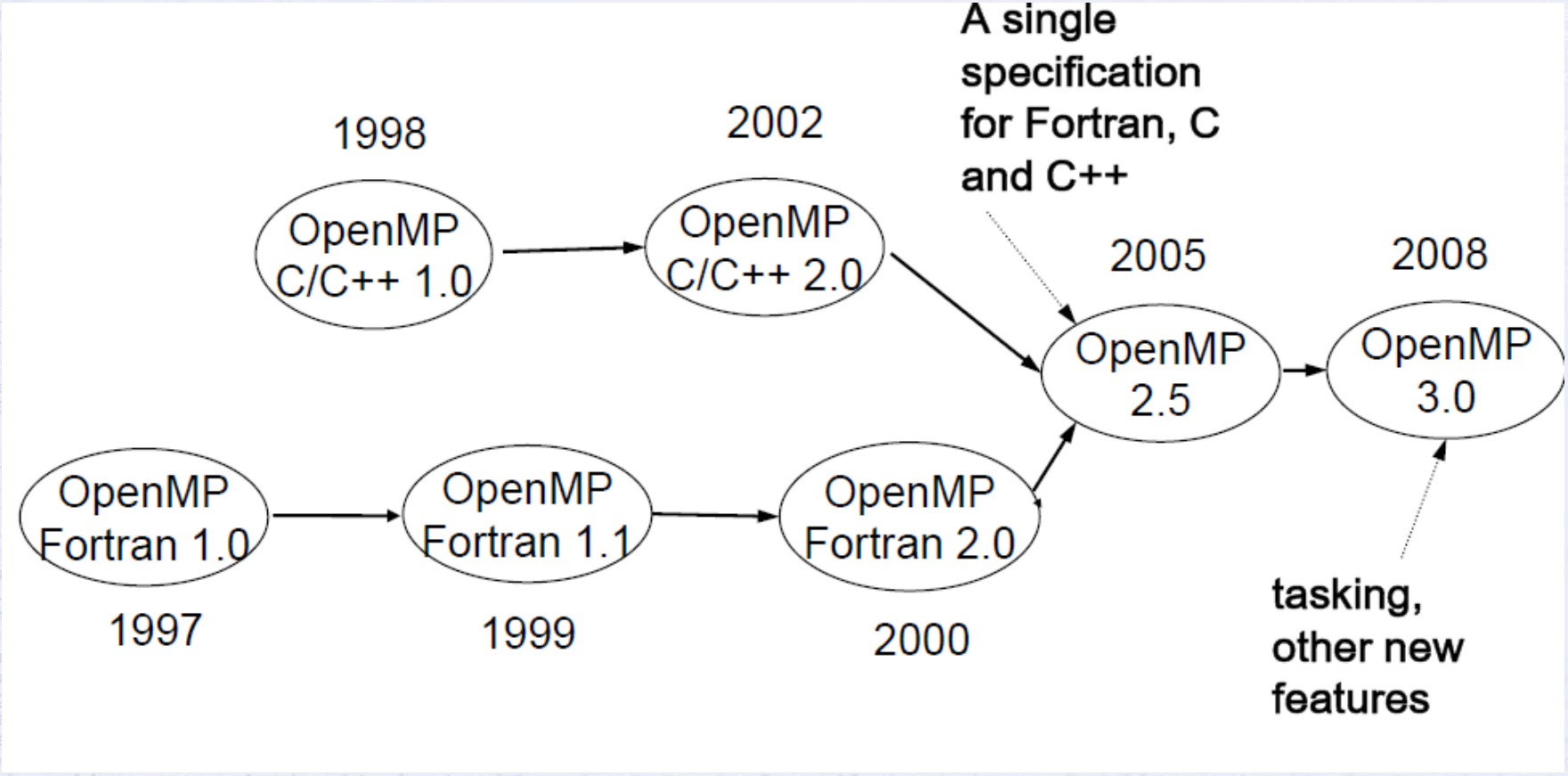
OpenMP: An API for Writing Multithreaded Applications

- *A set of compiler directives and library routines for parallel application programmers*
- *Greatly simplifies writing multi-threaded (MT) programs in Fortran, C and C++*
- *Standardizes last 20 years of SMP practice*
- *Current Version 4.0 (http://openmp.org/wp/openmp-specifications/)*

OpenMP Solution Stack



History of OpenMP



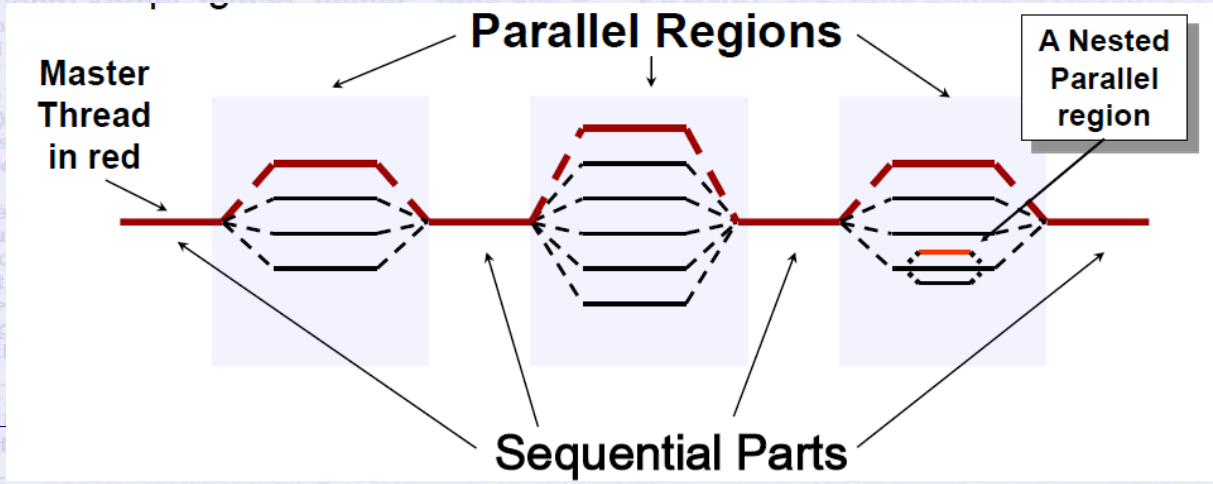
Today version 4.0 http://openmp.org/mp-documents/OpenMP-4.0-C.pdf

Programming Execution Model

Shared Memory, Thread Based Parallelism:
OpenMP is based upon the existence of multiple threads in the **shared memory programming** paradigm. A shared memory process consists of multiple threads.

Explicit Parallelism:
OpenMP is an **explicit** (not automatic) programming model, offering the programmer full control over parallelization.

Fork - Join Model:
- OpenMP uses the fork-join model of parallel execution
- All OpenMP programs begin as a single process: the **master thread**. The master thread executes sequentially until the first **parallel region** construct is encountered
- ***FORK*****:** the master thread then creates a ***team*** **of parallel threads**
- The statements in the program that are enclosed by the parallel region construct are then executed in parallel among the various team threads
- ***JOIN*****:** When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread



The essence of OpenMP

- **Create threads that execute in a shared address space:**
  – The only way to create threads is with the “parallel construct”
  – Once created, all threads execute the code inside the construct.

- **Split up the work between threads by one of two means:**
  – SPMD (Single program Multiple Data) … all threads execute the same code and you use the thread ID to assign work to a thread.
  – Workshare constructs split up loops and tasks between threads.

- **Manage data environment to avoid data access conflicts**
  – Synchronization so correct results are produced regardless of how threads are scheduled.
  – Carefully manage which data can be private (local to each thread) and shared.

OpenMP core syntax

- Most of the constructs in OpenMP are compiler directives.

  ```
  #pragma omp construct [clause [clause]…]
  ```

  - Example

    ```
    #pragma omp parallel num_threads(4)
    ```

- Function prototypes and types in the file:

  ```
  #include <omp.h>
  ```

- Most OpenMP* constructs apply to a “structured block”.
  - Structured block: a block of one or more statements with one point of entry at the top and one point of exit at the bottom.
  - It’s OK to have an exit() within the structured block.

C / C++ - General Code Structure

```
#include <omp.h>
main ()  {
int var1, var2, var3;
Serial code
        .
Beginning of parallel section. Fork a team of threads.
Specify variable scoping
#pragma omp parallel private(var1, var2) shared(var3)
  {
  Parallel section executed by all threads
        .
  Other OpenMP directives
        .
  Run-time Library calls
        .
  All threads join master thread and disband
  }
Resume serial code
        .
}
```

Specifying the number of threads

The number of threads is controlled by an internal control variable (**ICV**) called **nthreads-var**.

When a parallel construct is found a parallel region with a maximum of **nthreads-var** is created

Parallel constructs can be nested creating nested parallelism

The **nthreads-var** can be modified through

- the **omp_set_num_threads** API called (**omp_set_num_threads (2)**)
- the **OMP_NUM_THREADS** environment variable

Additionally, the **num_threads** clause causes the implementation to ignore the ICV and use the value of the clause for that region.

Other useful routines – API calls

int **omp_get_num_threads**()

Returns the number of threads in the current team

int **omp_get_thread_num**()

Returns the id of the thread in the current team

int **omp_get_num_procs**()

Returns the number of processors in the machine

int **omp_get_max_threads**()

Returns the maximum number of threads that will be used in the next parallel region

double **omp_get_wtime**()

Returns the number of seconds since an arbitrary point in the past

Data environment

shared the variable inside the construct is the same as the one outside the construct.

private the variable inside the construct is a new variable of the same type with an undefined value

Firstprivate the variable inside the construct is a new variable of the same type but it is initialized to the original variable value.

parallel [2.5] [2.4]

Forms a team of threads and starts parallel execution.

#pragma omp parallel *[clause[[,]clause] ...]*
structured-block

clause:

if(*scalar-expression*)
num_threads(*integer-expression*)
default(**shared** | **none**)
private(*list*)
firstprivate(*list*)
shared(*list*)
copyin(*list*)
reduction(*reduction-identifier*: *list*)
4.0 **proc_bind**(**master** | **close** | **spread**)

loop [2.7.1] [2.5.1]

Specifies that the iterations of associated loops will be executed in parallel by threads in the team in the context of their implicit tasks.

#pragma omp for *[clause[[,]clause] ...]*
for-loops

clause:

private(*list*)
firstprivate(*list*)
lastprivate(*list*)
reduction(*reduction-identifier*: *list*)
schedule(*kind*[, *chunk_size*])
collapse(*n*)
ordered
nowait

kind:

- **static:** Iterations are divided into chunks of size *chunk_size* and assigned to threads in the team in round-robin fashion in order of thread number.
- **dynamic:** Each thread executes a chunk of iterations then requests another chunk until none remain.
- **guided:** Each thread executes a chunk of iterations then requests another chunk until no chunks remain to be assigned.
- **auto:** The decision regarding scheduling is delegated to the compiler and/or runtime system.
- **runtime:** The schedule and chunk size are taken from the *run-sched-var* ICV.

REDUCTION Clause

Purpose:

The REDUCTION clause performs a reduction on the variables that appear in its list.

A private copy for each list variable is created for each thread. At the end of the reduction, the reduction variable is applied to all private copies of the shared variable, and the final result is written to the global shared variable.

```
#include <omp.h>
main () { int i, n, chunk;
float a[100], b[100], result;
/* Some initializations */
 n = 100; chunk = 10; result = 0.0;
 for (i=0; i < n; i++)
{ a[i] = i * 1.0; b[i] = i * 2.0; }
#pragma omp parallel for \
default(shared) private(i) \
schedule(static,chunk) \
 reduction(+:result)
for (i=0; i < n; i++)
result = result + (a[i] * b[i]);
printf("Final result= %f\n",result); }
```

reduction(*reduction-identifier*:*list*)

Specifies a *reduction-identifier* and one or more list items. The *reduction-identifier* must match a previously declared *reduction-identifier* of the same name and type for each of the list items.

Operators for reduction (initialization values)			
+	(0)	|	(0)
*	(1)	^	(0)
-	(0)	&&	(1)
&	(~0)	||	(0)
max (Least representable number in **reduction** list item type)			
min (Largest representable number in **reduction** list item type)			

OpenMP Examples

Compile and Run ExampleOMP1.c
Compile and Run ExampleOMP2.c
Compile and Run ExampleOMP3.c

Debuging OpenMP Threads

gcc -fopenmp -Wall -Werror example3OMP.c **-g** -o a.out

gdb ./a.out

b 47

r

thread

info threads