Brief intro to OMP

OpenMP (Open specifications for Multi Processing) is an API for shared-memory parallel computing. It was developed as an open standard for portable and scalable parallel programming, primarily designed for Fortran and C/C++. It is a flexible and easy to implement solution, which offers a specification for a set of compiler directives, library routines and environment variables.

As of 2021, the latest version of OpenMP API is 5.2. The OpenMP API is comprised of three components:

- Compiler Directives
- Runtime Library Routines
- Environment Variables

Many compilers (proprietary or open source) allow compilation of OpenMP directives in C or Fortran codes. Before using any of them one should check which OpenMP version the compiler's version supports.

Compiler Directives

Compiler directives are in the form of comments in the source code and are taken into account at compile time only if an appropriate compiler flag is specified. We use OpenMP compiler directives for:

- spawning a parallel region
- dividing blocks of code among threads
- distributing loop iterations between threads
- serializing sections of code
- synchronization of work among threads

The syntax of the compiler directives is as follows:

```
sentinel  directive-name  [clause, ...]
```

In the step Hello World! you have already learned the syntax of the OpenMP compiler directive in C and Fortran, i.e., for the directive name (construct) *parallel*.

Runtime Library Routines

These routines can be used for:

- setting and querying:
  - number of threads
  - dynamic threads feature
  - nested parallelism
- querying:
  - thread ID
  - thread ancestor's identifier
  - thread team size
  - wall clock time and resolution
  - a parallel region and its level

- setting, initializing and terminating:
    - locks
    - nested locks

All the runtime library routines in C/C++ are subroutines, while in Fortran some are functions, e.g., the runtime library routine for querying the number of threads in C is a subroutine:

```
int omp_get_num_threads(void)
```
while in Fortran it is a function:

```
INTEGER FUNCTION OMP_GET_NUM_THREADS()
```
Also note, that in C/C++ a specific header has to be generally included and that, contrary to Fortran, C/C++ routines are case sensitive:

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

- setting:
    - number of threads
    - thread stack size
    - thread wait policy
    - maximum levels of nested parallelism
- specifying how loop interations are divided
- binding threads to processors
- enabling/disabling:
    - nested parallelism
    - dynamic threads

The OpenMP environment variables are set like any other environment variables, depending on the shell used, e.g., you can set the number of OpenMP threads in *bash* with:

```
export OMP_NUM_THREADS=2
```
and in *csh* with:

```
setenv OMP_NUM_THREADS 2
```
Compiling codes with OpenMP directives

You have already seen in the step Hello World! how to compile C and Fortran codes with OpenMP directives:

```
!gcc hello_world.c -o hello_world -fopenmp
!gfortran hello_world.f90 -o hello_world.exe -fopenmp
```
We used GNU C and Fortran compilers, `gcc` and `gfortran`, respectively, with the compiler flag `-fopenmp` to tell the compiler to take OpenMP directives into account. This flag is dependent on the compiler used, the following table shows which flags have to be used by typical compilers for Unix systems.

1.10 OpenMP memory, programming and execution model

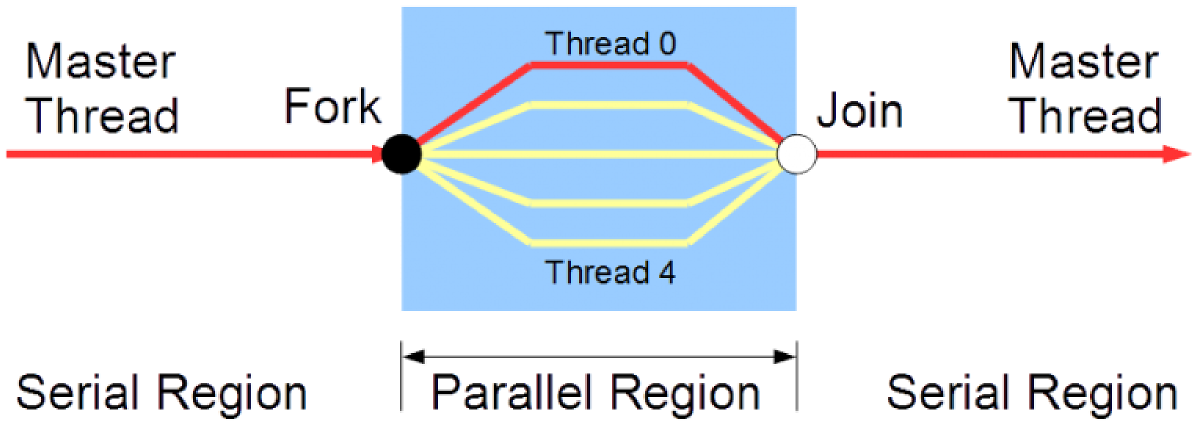
OpenMP is based on the shared memory model of multi-processor or multi-core machines. The shared memory type can be either Uniform Memory Access (UMA) or

Non-Uniform Memory Access (NUMA). In OpenMP, programs accomplish parallelism exclusively with the use of threads, so called thread-based parallelism.

A thread is the smallest unit of processing that can be scheduled. Threads can exist only within the resources of a single process. When the process is finished, the threads also vanish. The maximum number of threads is equal to the number of processor cores times threads per core available. The actual number of threads used is defined by the user or application used.

In the introduction, we referred to OpenMP as an easy approach for doing "automatic" parallelization. In reality, OpenMP is an explicit *programming model*, which offers the user full control over parallelization. Although not automatic in a strict sense, parallelization is simply achieved by inserting compiler directives in a serial program and hence "automatically" transforming it into a parallel program. Of course, OpenMP also offers complex programming approaches such as inserting subroutines to set multiple levels of parallelism, locks and nested locks, etc.

The basis of OpenMP's *execution model* is the **fork-join model** of parallel execution. OpenMP programs begin as a *master thread* that is executed sequentially until the first parallel region construct is encountered. The master thread then creates a team of parallel threads - a *fork*. The executable statements that are inside the parallel region construct are executed in parallel by the team threads. After the team threads finish execution of the statements in the parallel region construct, synchronization among them occurs and finally their termination results in a *join*, with the master thread as the only thread left.



You can compile OMP files with extension *.c/ *.cpp with CODEBLOCK compiler on your laptop else use COLAB

```
%env OMP_NUM_THREADS=3

 ls –l

 !g++ bubble1.cpp -o bubble1 -fopenmp !./bubble1
```