Practical Session 1 : Introduction

Objectives

The objective of this lab is to build some understanding of the *OpenMP* library. You should write your code using the *C++* language.

- know the architecture of your hardware.
- to manipulate the environment variable to control your parallel execution.
- to compile a code that uses *OpenMP*.
- write directives.
- verify at runtime some information about the execution.

1 – Environment

Question

1. The command `cat /proc/cpuinfo` provides information about the architecture of the system.
   Find the number of processor that can be used to perform parallel processing on your computer.
2. In order to change the number of process associated with the execution of the program you can change the shell variable `OMP_NUM_THREADS`.
   Modify the program `hello.c` to display information about the different threads.

Remark 1

In order to carry out this exercise, you must use the following elements

- include the correct header (`openmp.h`),
- compile with the correct options (`-fopenmp`)
- define a parallel region (`#pragma omp parallel`)
- obtain the running thread id (`int omp_get_thread_num()`)

2 – Numerical Integration

The value of an integral can be approximated by computing its Riemann sum

$$\int_a^b f(x)dx \approx \sum_{i=1}^{n} f(\tilde{x}_i)dx$$

where $dx = x_i - x_{i-1} = (b-a)/n$, and $\tilde{x}_i$ is some point in the interval $[x_{-1}, x_i]$, $x_0 = a$ and $x_n = b$

One of the possible approximation is the left rectangle rule. In this exercise, we will use the rectangle rule to compute an approximation of π by using the following definition

$$\int_0^1 \frac{4}{1+x^2}dx = \pi$$

Question

- Run the sequential program `pi.cxx` and modify some parameters to observe the performances.

- Create a parallel version of the pi program using a parallel construct : `#pragma omp parallel`.

Remark 2

In order to carry out this exercise, you will also need the runtime library routines

- `int omp_get_num_threads();`
- `int omp_get_thread_num();`
- `double omp_get_wtime();`
- `int omp_set_num_threads();`

While implementing your program in parallel, you should pay attention on how to divide the loop iterations between threads (use the thread ID and the number of threads) and create an accumulator for each thread to hold partial sums that you can later combine to generate the global sum.

3 – Mandelbrot set

In this exercise, we supply a program in C++, that computes the area of a Mandelbrot set (`mandel.cxx`).
The program has been parallelized with OpenMP, but some errors were made during the parallelizing process.

Question

- Identify and fix the errors by looking more precisely on how the data are shared among the different threads.
- Once the parallel program is working, try to optimize it
  - Try different schedules on the parallel loop.
  - Try different mechanisms to support mutual exclusion
  - Do the efficiencies change ?

4 – Computation using Random number

In this exercise, we supply a program in C++, that computes the value of π using Monte Carlo simulation (`pi_serial.cxx`).

Question

- Write a parallel version of the same program using OpenMP.
- Compare the performances of the sequential program and of your parallel program.
- After observing the results for different number of iteration, try to draw some conclusion and provide some explanation.