EXERCISE 1

Write an OpenMP program that calculates the value of PI Greek

► The request is to translate the MPI program that has been given for Assignment I into an OpenMP one.

After that, study its behaviour on a node of Ulysse:

1. establish its weak and strong scalability;
2. estimate the parallel overhead;
3. compare the performance of your OpenMP version and of the MPI version, in terms of time-to-solution and of parallel efficiency.
   Run the MPI version with N_c processes (i.e. N_c = the largest number of physical threads that you have on the node) both on the single node that you use for the OpenMP version and on multiple nodes (keeping constant the number of processes). That should allow you to understand the impact of the network and how good is the shared-memory implementation of the MPI library.

> note: Appendix I here below may be useful

EXERCISE 2

Binary search with OpenMP

► The task in this exercise is to is OpenMP-ize the binary search program that you have seen in the lecture about the prefetching.

Then, you have to report about the performance comparison of the serial and the parallel version.
Performance is about: time-to-solution, parallel overhead, any other relevant metrics you think is informative.

> note: Appendix II may offer an useful tool. Nevertheless, it is not neither mandatory nor necessary to use it.

EXERCISE 3

Prefix sum with OpenMP

The "prefix sum", also known as "scan", is a very common computational pattern that has many important applications; scan pattern can be used in resource allocation, polynomial evaluation, string comparison, radix sort, quick sort, tree operations, an many others. Recursion, in general, is likely to be parallelized as a parallel scan computation.

Moreover, scan pattern is a linear algorithm, and then it is remarkably work-efficient. On one hand, that fact stresses its importance in computation. On the other hand, it makes it evident how difficult is to design an adequate parallelization without an excessive overhead.

An *inclusive scan* operates on an input array the *n* elements $[x_0, x_1, \ldots, x_{n-1}]$ through a binary associative operator $\otimes$, ending up in the following output:

$$[x_0, (x_0 \otimes x_1), \ldots, (x_0 \otimes x_1 \otimes \ldots \otimes x_{n-1})]$$

In this exercise, $\otimes$ is the $+$ addition operator and the x_i elements are numbers, then the prefix sum of the input arrays

[9 5 1 12 3 7]

[3.1415926, 5.6703e-8, 2.718282, 1.6180, 6.6742e-11, 0.7071068, 1.414213]

is

[9 14 15 27 30 37]

[3.1415926, 3.141592656703, 5.859874656703, 7.477874656703, 7.477874656769742, 8.184981456769742, 9.599194456769742],

respectively.

► The request is to develop an efficient version of a prefix sum with the $+$ operator, and also to OpenMP-ize it. Then, to compare the serial and the parallel version, reporting at least about the time-to-solution and the parallel overhead.
Initialize the input array to your liking.

EXERCISE 4

The Mandelbrot set with OpenMP

► In this exercise, you are required to implement a parallel code that iteratively calculates Eq. ??? for a given section of the complex plane (or, in other words, that computes the Mandelbrot set).

The Mandelbrot set is generated on the complex plane $\mathbb{C}$ by iterating the complex function $f_c(z)$ whose form is

$$f_c(z) = z^2 + c$$

for a complex point $c = x + iy$ and starting from the complex value $z = 0$ so to obtain the series

$$z_0 = 0,\ z_1 = f_c(0),\ z_2 = f_c(z_1),\ \ldots,\ f_c^n(z_{n-1})$$

The $Mandelbrot\ Set\ \mathcal{M}$ is defined as the set of complex points c for which the above sequence is bounded. It may be proved that once an element i of the series is more distant than 2 from the origin, the series is then unbounded.
Hence, the simple condition to determine whether a point c is in the set $\mathcal{M}$ is the following

$$|z_n = f_c^n(0)| < 2 \ \text{ or } \ n > I_{max}$$

where I_{max} is a parameter that sets the maximum number of iteration after which you consider the point c to belong to $\mathcal{M}$ (the accuracy of your calculations increases with I_{max}, and so does the computational cost).

Given a portion of the complex plane, included from the bottom left corner $c_L = x_L + iy_L$ and the top right one $c_R = x_R + iy_R$, an image of $\mathcal{M}$, made of $n_x \times n_y$ "pixels" can be obtained deriving, for each point c_i in the plane, the sequence $z_n(c_i)$ to which apply the condition (???), where

$$
\begin{aligned}
c_i &= (x_L + \Delta x) + i(y_L + \Delta y) \\
\Delta x &= (x_R - x_L)/n_x \\
\Delta y &= (y_R - y_L)/n_y \,.
\end{aligned}
$$

In practice, you define a 2D matrix `M` of integers, whose entries `[j][i]` correspond to the image's pixels. What pixel of the complex plane $\mathbb{C}$ corresponds to each element of the matrix depends obviously on the parameters $(x_L, y_L), (x_R, y_R), n_x, \text{ and } n_y$.

Then you give to a pixel `[j][i]` either the value of 0, if the corresponding c point belongs to $\mathcal{M}$, or the value n of the iteration for which

$$
|z_n\,(c)| > 2
$$

(n will saturate to I_{max}).

This problem is obviously embarrassingly parallel, for each point can be computed independently of each other and the most straightforward implementation would amount to evenly subdivide the plane among concurrent processes (or threads). **However, in this way you will possibly find severe imbalance problems because the $\mathcal{M}$'s inner points are computationally more demanding than the outer points, the frontier being the most complex region to be resolved**.

Requirements:

1. your code have to accept $I_{max}, c_L, c_R, n_x \text{ and } n_y$ as arguments. Specifically, the compilation must produce an executable whose execution has a proper default behaviour and accept argument as follows:
   `./executable n_x  n_y  x_L  y_L  x_R  y_R  I_max`
2. the size of integers of your matrix `M` shall be either `char` (1 byte; $I_{max} = 255$) or `short int` (2 bytes; $I_{max} = 65535$).
3. your code must produce a unique output file, using MPI I/O if you choose to implement the MPI version. The output must be in one of the 2 following format:
   - simple binary format, as follows:
     4 double: x_L, y_L, x_r, y_R
     3 integers (4 bytes long): n_x, n_y and I_{max}
     The exact dump in row-major order of the 2D matrix `M` that you have calculated.
   - you may **directly produce an image file** using the very simple format `.pgm` that contains a grey-scale image. You find a function to do that, and the relative simple usage instructions, in Appendix III at the end of this page.
     In this way you may check in real time and by eye whether the output of your code is meaningful.
4. you have to determine the strong and weak scaling of your code, accordingly to the choice of developing an MPI or OpenMP (or hybrid) version.

> **Note 1:** Mandelbrot set lives roughly in the circular region centered on $(-0.75, 0)$ with a radius of ~ 2.

> **Note 2:** the multiplication of 2 complex numbers is defined as
> $(x_1 + iy_1) \times (x_2 + iy_2) = (x_1x_2 - y_1y_2) + i(x_1y_2 + x_2y_1)$

Appendix I - random numbers in parallel

For some of the exercises proposed above, you will need pseudo-random numbers.
However, the generation of "true" pseudo-random number in parallel is an active field of research and a difficult issue. So, you're of course not required to solve that, but just to pay attention: it is not correct to use the standard routines you have used so far (they rely on a status register that is shared).

You may want to use `rand_r()`:

```
#pragma omp parallel
{
    int myid   = omp_get_thread_num();
    int myseed = function_of_myid( myid );
    int random_number = rand_r();
}
```

Although `rand_r()` satisfies only elementary statistical properties, it would be enough for your work.
If you want to adopt something more sophisticated but still simple, you can think to the `drand48_r()` family, though they are *only* GNU extensions:

```
#pragma omp parallel
{
    int myid   = omp_get_thread_num();
    struct drand48_data myrng;
    // here you initialize the 3 entries of mystatus
    // with some function_of_myid( myid );
    long int myseed = FUNCTION_THAT_GET_UNIQUE_SEED_per_THREADID(myid);
    srand48_r( myseed, &myrng );
    double random_number;
    drand48_r(&myrng, &random_number);
}
```

Appendix II - how to exit from a parallel region

OpenMP $\geq$ 4.0 **has** a cancellation construct that causes a loop (or other constructs, see below) to exit before its natural end.

> NOTE : `gcc` up to v.9.3 does not yet support this construct. Recent enough `icc`, `pgi` and `clang` compilers do.

Once a thread meets the exit condition, it breaks issuing a cancel request to all other threads. The other threads are checking whether such a request has been issued, and, if so, they also exit the loop.

For the cancellation directive to work, The environmental variable `OMP_CANCELLATION` must be set to `true`:

```
export OMP_CANCELLATION=TRUE
```

In the cancellation model, there are 2 fundamental point:

1. the `cancel` construct, at which a thread breaks the loop
2. the `cancellation` point, at which a thread checks whether there is a request for cancellation

The cancellation construct exists for the following parallel regions:

- `parallel`
- `for`
- `sections`
- `taskgroup`

Example:

```
#pragma omp for
for ( i = 0; i < N; i++ )
  {
    do_some_work( ... );

    if ( i == key )
      {
        #pragma omp cancel for
      }
    #pragma omp cancellation point for
  }
```

You find this code above in a more comprehensive and commented example in the assignment directory, `cancel_construct.c`.

Appendix III - how to write a .pgm file

The `PGM` image format, companion of the `PBM` and `PPM` formats, is a quite simple and portable one.
It consists in a small header, written in ASCII, and in the pixels that compose the image written all one after the others as integer values. A pixel's value in `PGM` corresponds to the grey level of the pixel.
Even if also the pixels can be written in ASCII format, we encourage the usage of a binary format.

The header is a string that can be formatted like the following:

`printf( "P5\n%d %d\n%d\n", width, height, maximum_value );`

where "`P5`" is a magic number, `width` and `heigth` are the dimensions of the image in pixels, and `maximum_value` is a value smaller than `65536`.
If `maximum_value < 256`, then 1 byte is sufficient to represent all the possible values and each pixel will be stored as 1 byte. Instead, if `256 <= maximum_value < 65536`, 2 bytes are needed to represent each pixel (that is why in the description of Exercise 1 we asked you that the matrix `M` entries should be either of type `char` or of type `short int`).

In the sample file `write_pgm_image.c` that you find the `Assignment03` folder, there is the function `write_pgm_image()` that you can use to write such a file once you have the matrix `M`.

In the same file, there is a sample code that generate a square image and write it using the `write_pgm_image()` function.
It generates a vertical gradient of N_x x N_y pixels, where N_x and N_y are parameters. Whether the image is made by single-byte or 2-bytes pixels is decided by the maximum colour, which is also a parameter.
The usage of the code is as follows

```
cc -o write_pgm_image write_pgm_image.c
./write_pgm_image [ max_val] [ width height]
```

as output you will find the image `image.pgm` which should be easily rendered by any decent visualizer .

Once you have calculated the matrix `M`, to give it as an input to the function `write_pgm_image()` should definitely be straightforward.