TPs MPI

Exercise 1

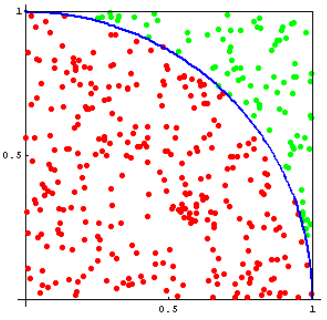
Run the **`test_graphic.py`** and **`test_mpi.py`** from the TP.zip archive

- **`python3 test_graphique.py`**
- **`mpirun -n 2 python3 test_mpi.py`**

Run the codes in **Example of MPI code from the Lesson slides**

- Run and analyze at least **bcast.py**, **gather.py**, **scatter.py** and **reduce.py**

Exercise 2: Monte Carlo Simulation to compute π



Surface of circle is $\pi * r^2$ we will use this equation to compute π

1. Run the code in **`monte_carlo.py`**
2. Implement a parallel version of the same algorithm where
   a. There is no communication at the beginning
   b. There is only one communication at the end (no **`send`**/**`receive`**, only collective operations)
   c. Only processus of rank 0 prints the result
3. Print the value of 'inside' for each process? Do not forget to initialize the random generator differently for each process.
4. Compare the time for 1, 2 and 4 processes
5. Same with higher value of nb taking at least 2 seconds for 1 process

Exercise 3: Contrast stretching

The goal is to stretch the contrast of images converted to greyscale.

1. Run the code in **`stretching-base.py`**
   In this code, two different methods are used to stretch the contrast (**`f_one`** and **`f_two`**): Test both. You have to close the color picture to go to the next step. You have to close the gray picture to finish the execution.
2. Parallelize the code
   - Only process with **`rank==0`** loads and saves the image
   - Compute *max* and *min* in parallel
   - Use the *stretch functions* (**`f_one`** and **`f_two`**) in parallel
   - Make it so that even ranked processus use **`f_one`**, while odd processus use **`f_two`**
   - Compare the time for 1, 2 and 4 processes

The result for the input image on the right (in color) should be like the one in grayscale for 4 processes.

Exercise 4: Maximum number of divisors

The number of divisors of a number `nb` is the total number of integer i such as

nb % i == 0

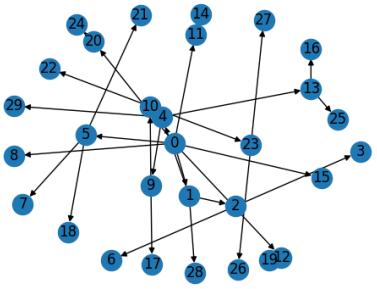
The goal will be to compute the maximum number of divisors for all numbers below N, i.e. for all nb from 1 to N. For all versions print also the time needed to do the computation (both total time, and time just for the loop without the collective operation).

1. Run the sequential program **`primes.py`**
2. Implement the parallel version with blocs of work (only using collective operations)

3. Implement the parallel version by distributing the work one by one (only using collective operations)

4. Compare the time of each version and explain

Exercise 5: Unreachable part of graphs



The goal is to check if all nodes of a graph are reachable

1. Run the code in graph-base
2. Parallelize the code so that new generations are well balanced
    a. For each iteration of the **`while`**, show the number of nodes managed by each process.

On computers from university install networkx package

```
python3 -m pip install --user networkx
```

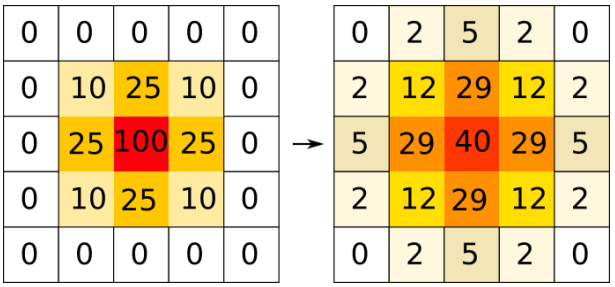
Exercise 6: Heat propagation

To compute heat propagation in a matrix representing an object, we have to apply the following formula for all points:

$V_{k+1}(i,j) = (V_k(i-1, j) + V_k(i, j-1) + V_k(i, j) + V_k(i+1, j) + V_k(i, j+1)) / 5$

The code in **`heat.py`** generates such a matrix and makes it evolve over several iterations (representing time).

Modify the code to make it run in parallel. Check the signature and plot time in function of the number of processes.



Exercice 7: N-bodies problem (to upload on moodle)

The basic file is called **`n-bodies-base.py`**. It is launched with the command:

```
mpirun -n 3 python3 n-bodies-base.py 12 1000
```

where the first parameter is the number of bodies (n-bodies), and the second one will be the number of iterations we want to see.

You should NOT touch the instructions at the beginning of the file, all the code you need to add should be added at the end of the file.
The file provided:

- does the imports you need
- defines a number of constants and utility functions. Among them:
    - `init_world(n)`: this function returns a list containing all n bodies (n is passed as parameter). Each body has several attributes: position, speed, weight. However you should not have to manipulate them directly.
    - `update(d,f)` : this function takes a body d and a force f as parameters and returns the body d whose position and speed have been modified by the force f
    - `interaction(body1, body2)`: this function returns a list with two elements representing the force on each of the X and Y axes, resulting from the interaction of `body2` on `body1`
    - `display(m, l)` : this function allows to display the parameters of the bodies contained in the list l by prefixing this display with the message m
    - `displayPlot(data)` : display the bodies which are in data on the graphical window
    - `signature(data)` : compute a value based on the characteristics of the bodies, which represents the signature of this world. Used after the evolution iterations, to check that the evolution of the world calculated in sequential and in parallel is the same.
    - `split(x, size)` : returns a list which is a split of the list x into a list of size lists. Potentially the last list contains less elements than the others, if it doesn't others, if it doesn't split right.
    - `unsplit(x)` : creates a list from a list x of lists
    - the other utility functions and the class are not intended to be used directly.

1. Start by writing the sequential algorithm in python, which will therefore only run on one processor (run by **`python3 n-bodies-base.py 12 1000`**). Note the signature value given for the world at the end of the simulation.
    a. **Reminder:** interaction returns a tuple (as the force is a vector of two components). During the TD/Practical Lab we used a **`!+!`** to add vectors. Here you will need several lines to add the components of the force: `[fx,fy] = force[i]; [dfx, dfy] = interaction(data[i], data[j]); force[i] = [fx+dfx, fy+dfy]`
    b. It might be the same for initializing forces: `forces = [[0,0] for _ in range(N)]`
2. Using the MPI library, propose a parallel version of the code. Compare the signature value given for the world at the end of the simulation with the value of the sequential version.
3. Add into the code something to measure the execution time (`comm.Wtime()`). Start the execution using 1, 2, 4, 8 instances, on 1, 2, 4 and 8 processes (if possible on your machine) with 1024 bodies and 10 iterations. Draw the speedup graph.

Upload the code on Moodle in an archive including:

- The **python** code of first parallel version
- The **python** code of parallel version using the fact that Fij = -Fji
- The explanation of the limits (in a **text** file)
- As a *bonus*, a version of the **python** code solving the problem
- A **.pdf** or **.png** file with the speedup curve