Università della Svizzera italiana | **Institute of Computing CI**

High-Performance Computing **2022**

Student: Qianbo Zang Discussed with: Mazio Lunghi, Anthony Bugatto, Luca Pernigo

Solution for Project 7

Due date: 21.12.2022, 23:59

1. Parallel Space Solution of a nonlinear PDE using MPI [in total 35 points]

Please see my code.

1.1. Initialize and finalize MPI [5 Points]

Please see my code.

1.2. Create a Cartesian topology [5 Points]

Please see my code.

1.3. Extend the linear algebra functions [5 Points]

Please see my code.

1.4. Exchange ghost cells [10 Points]

Please see my code. And after this step, run plotting.py.

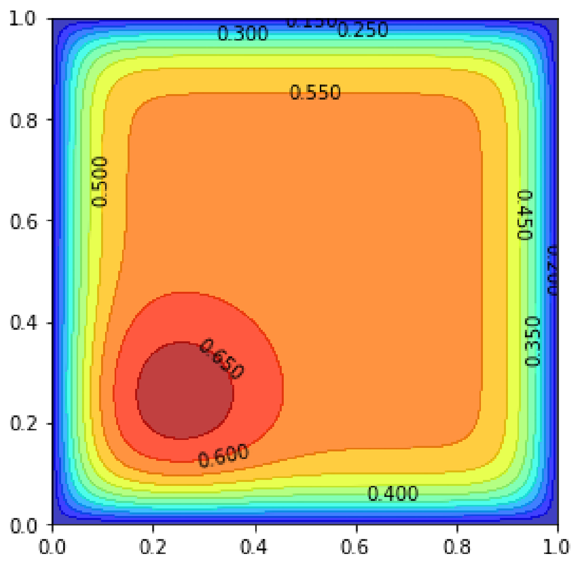


Figure 1: Visualisation of output

1.5. Scaling experiments [10 Points]

1.5.1. Plot

Plot the time to solution using 1-20 MPI ranks on 1 compute node

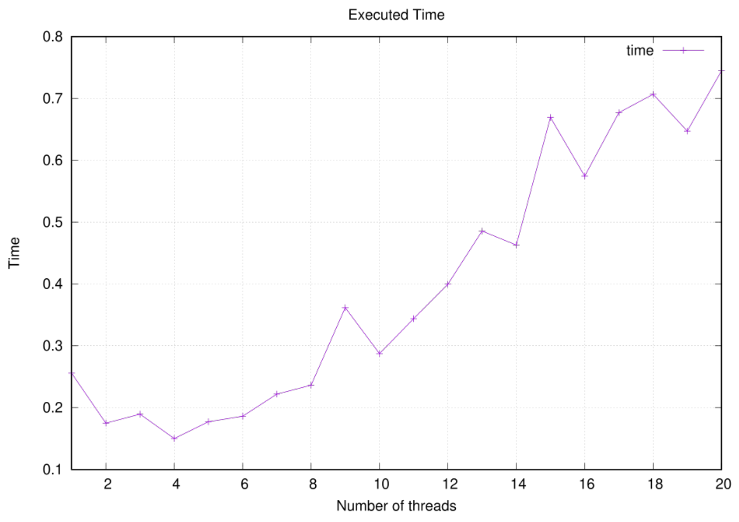


Figure 2: 128 x 128

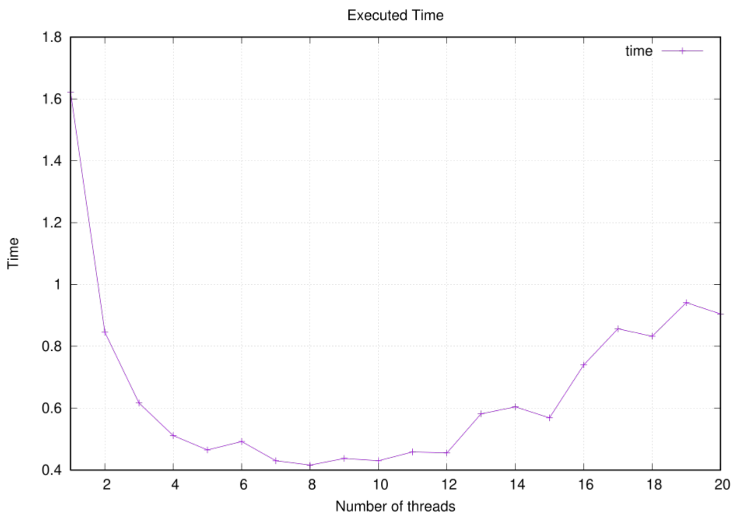


Figure 3: 256 x 256

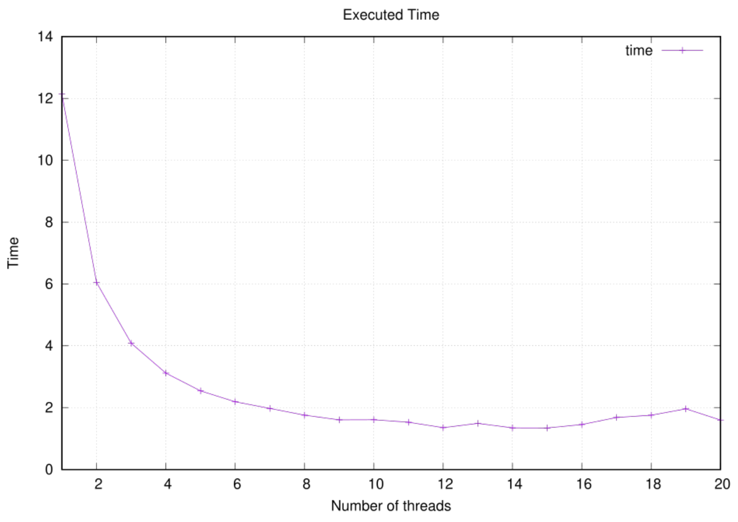


Figure 4: 512 x 512

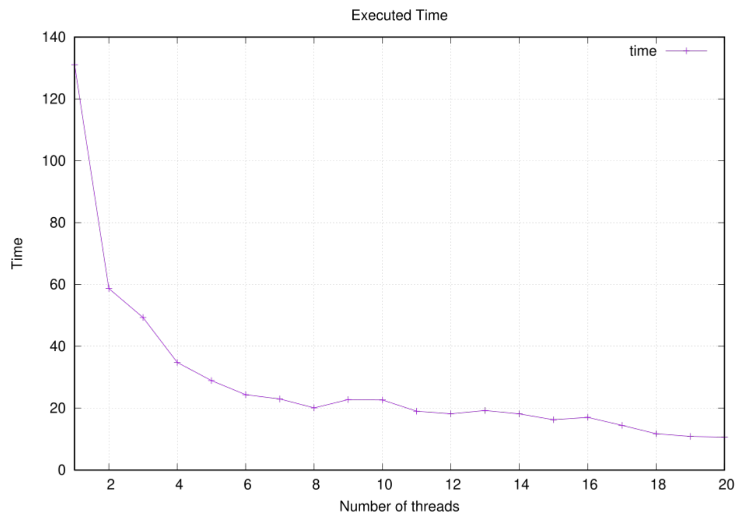


Figure 5: 1024 x 1024

Plot the 1 process per node

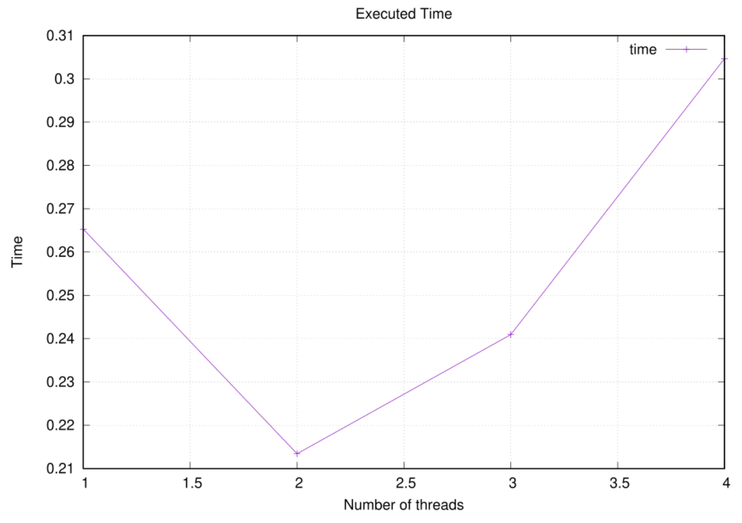


Figure 6: 128 x 128

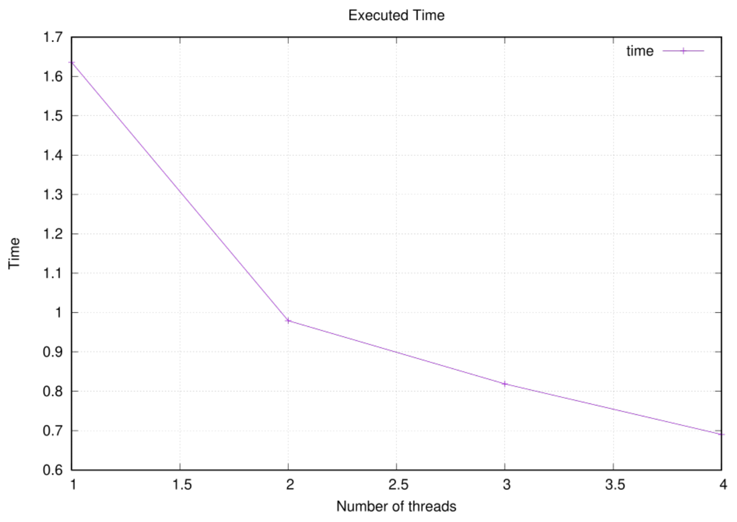


Figure 7: 256 x 256

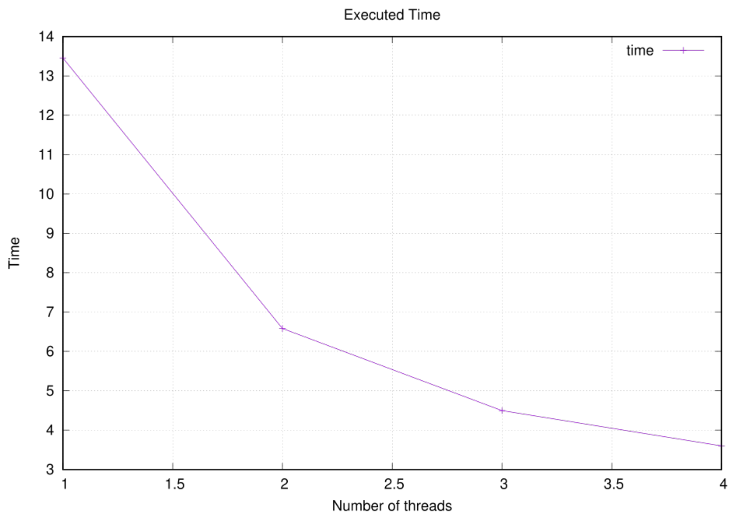


Figure 8: 512 x 512

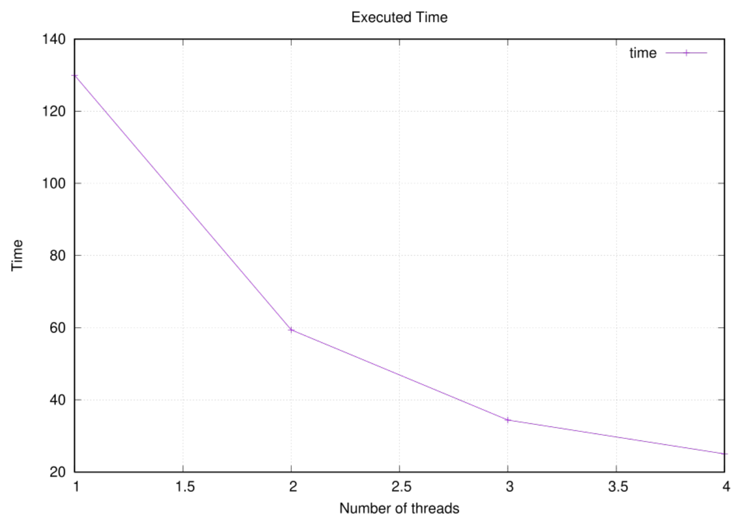


Figure 9: 1024 x 1024

1.5.2. Analysis

Strong
In the case of using one compute node, when the resolution is lower, adding processes instead increases the runtime. This is because the communication of processes also takes time. while the higher the resolution, the better it is to use a large number of processes. We can conclude that for low resolution images, the introduced overhead is greater than the speed gain in execution, leading to a performance degradation. For high resolutions, the opposite is true, the introduced overhead is worth the speed gain in execution, leading to better performance.
Comparing the performance of parallelizing the same problem using openmp, I can notice that openmp has a much higher overhead using threads in the case of high resolution. Whereas threads perform better for low resolution images and the number of threads is high (up to 24), and using processes in this case incurs high overhead.
Weak
Even if the workload per process remains constant, the runtime increases as the number of processes increases. This is due to the fact that the increase in the amount of communication between processes increases also generates additional time. Also, too many problems can cause iterations to stop collecting.

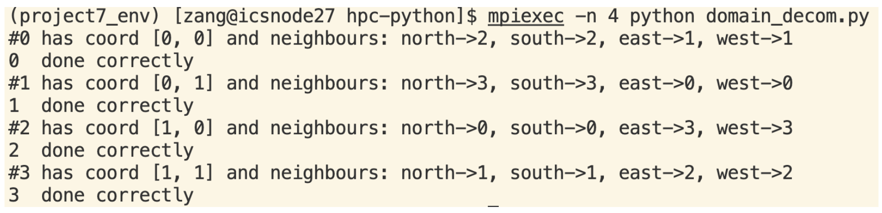
2. Python for High-Performance Computing (HPC) [in total 50 points]

2.1. Sum of ranks: MPI collectives [5 Points]

Use comm.reduce() methods. Please see my code.

2.2. Domain decomposition: Create a Cartesian topology [5 Points]

Use comm.Create_cart(), comm_cart.Get_coords() and comm_cart.Shift(). Please see my code. When ranking 4, the topology is here.



```
(project7_env) [zang@icsnode27 hpc-python]$ mpiexec -n 4 python domain_decom.py
#0 has coord [0, 0] and neighbours: north->2, south->2, east->1, west->1
0  done correctly
#1 has coord [0, 1] and neighbours: north->3, south->3, east->0, west->0
1  done correctly
#2 has coord [1, 0] and neighbours: north->0, south->0, east->3, west->3
2  done correctly
#3 has coord [1, 1] and neighbours: north->1, south->1, east->2, west->2
3  done correctly
```

2.3. Exchange rank with neighbours [5 Points]

Use comm_cart.isend(), and comm_cart.recv(). Please see my code.

2.4. Change linear algebra functions [5 Points]

Use comm.Allreduce(). Please see my code.

2.5. Exchange ghost cells [5 Points]

Use comm.Isend() and comm.Irecv(). Please see my code.

2.6. Scaling experiments [5 Points]

2.6.1. Plot

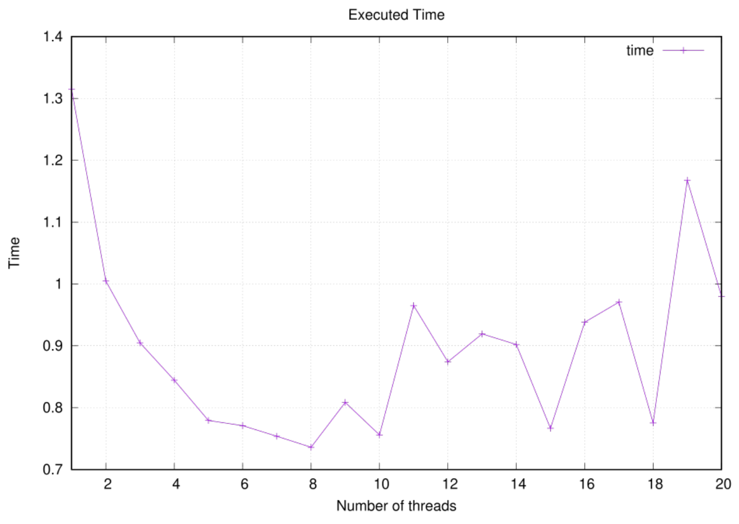


Figure 10: 128 x 128

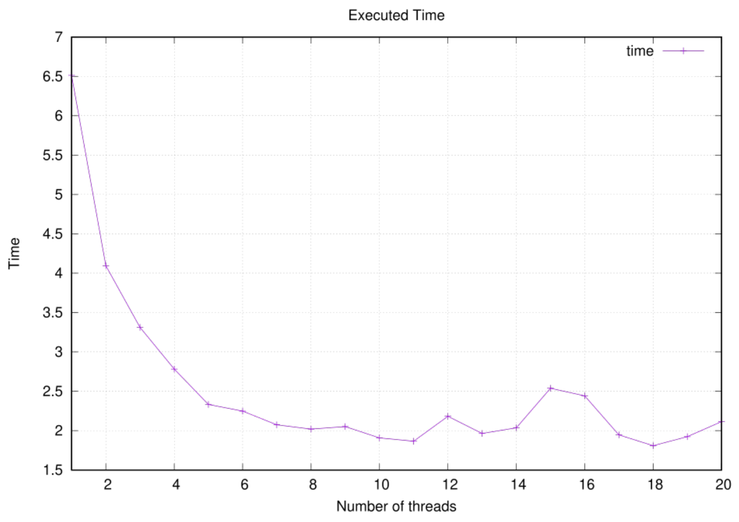


Figure 11: 256 x 256

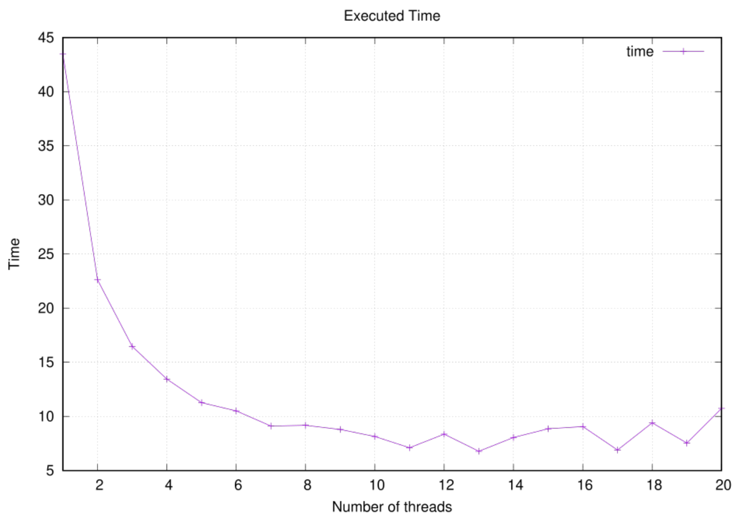


Figure 12: 512 x 512

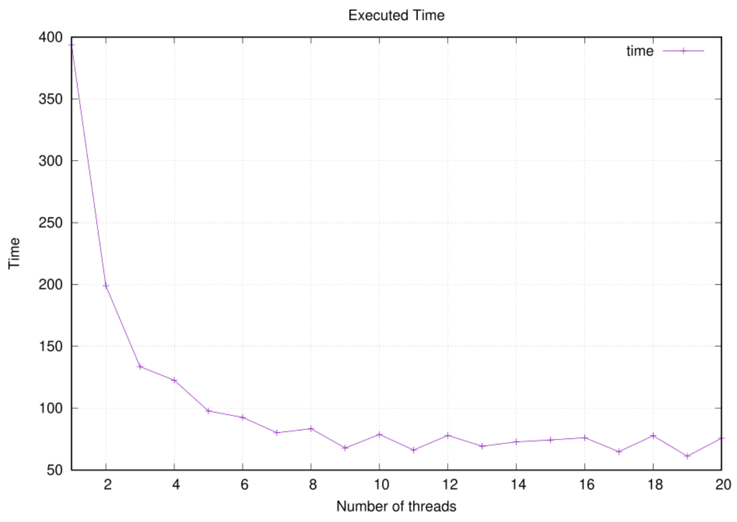


Figure 13: 1024 x 1024

2.6.2. Analysis

In the python experiments, the runtime and threaded functions follow the same trajectory as in C++. However, the total running time of the python experiments is almost three times longer than the C++ implementation. This can be explained by the general slowness of Python programs. As before, too large a pixel can cause iterations to stop converging.

2.7. A self-scheduling example: Parallel Mandelbrot [20 Points]

2.7.1. Plot

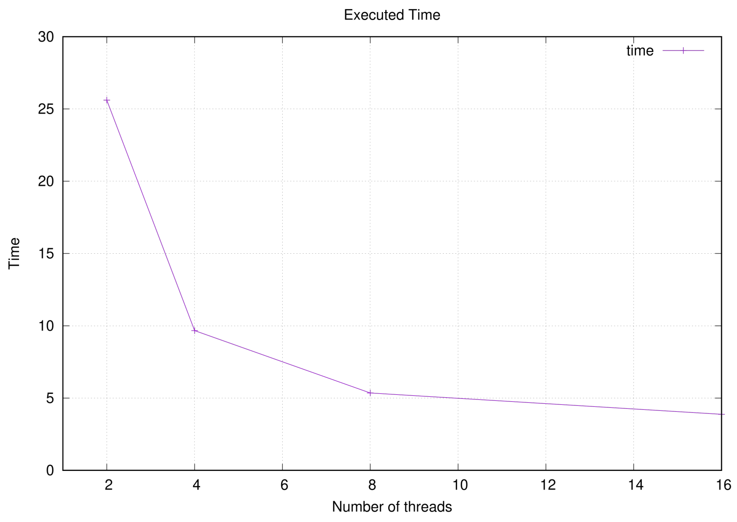


Figure 14: workload into 50

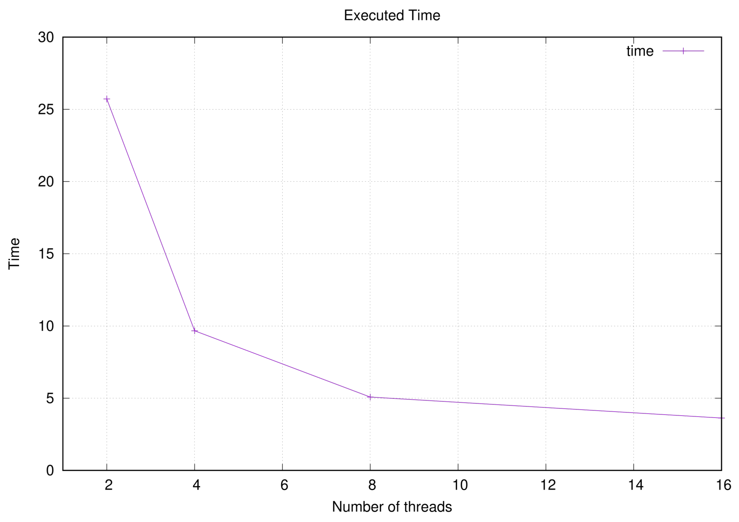


Figure 15: workload once into 100

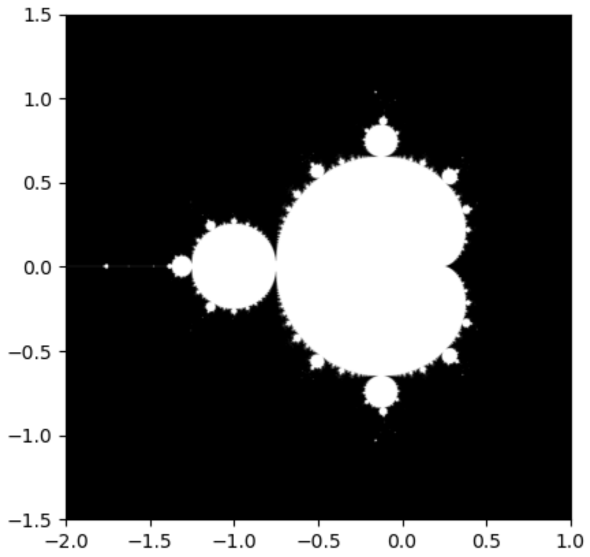


Figure 16: visualisation of mandelbrot

2.7.2. Analysis

By strong scaling analysis, the total running time is approximately the same for 50 and 100 tasks, and decreases as the number of processes increases. This indicates that this task is extremely well suited for parallel computing.

3. Task: Quality of the Report [15 Points]

Each project will have 100 points (out of which 15 points will be given to the general quality of the written report).

Additional notes and submission details

Submit the source code files (together with your used `Makefile`) in an archive file (tar, zip, etc.), and summarize your results and the observations for all exercises by writing an extended Latex report. Use the Latex template from the webpage and upload the Latex summary as a PDF to iCorsi.

- Your submission should be a gzipped tar archive, formatted like project_number_lastname_firstname.zip or project_number_lastname_firstname.tgz. It should contain:
  - all the source codes of your solutions;
  - your write-up with your name project_number_lastname_firstname.pdf.
- Submit your .tgz through Icorsi.