Image Loader and Visualizer

Niccolò Mazzi - 7076194

January 27, 2023

Abstract

In this report we are going to illustrate the implementation of an image loader both in a sequential (single-core) and a parallel (multi-core) way.

1 Introduction

The presented implementation consists in a non-blocking image loader which allows to load in memory some images and subsequently visualize them.

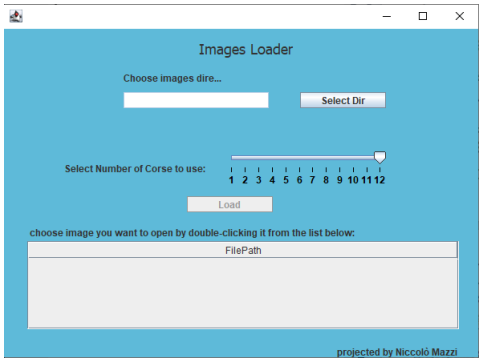
The test which leaded to the presented results have been executed on the following HW setup:

- **CPU**: 11th Gen Intel i5-11400F (6 cores, 12 threads)
- **RAM**: 16 GB DDR4
- **SSD**: Crucial P2 CT2000P2SSD8 2TB **M2**

The dataset is composed by 229 images with a complexive size of 60,4 MB.

1.1 GUI structure

To interact with the project we use an interface generated using WindowsBuilder.



Through this interface it is possible to choose the directory in which the images are saved and, by clicking on the "Load" button, we can load them in the memory.
We can choose the number of cores to use by moving the slide we can see in the interface from 1 to 12.
If the number of used cores is set to 1, the program will be executed in the sequential way.

1.2 Working logic

The whole program is based on a fork-join paradigm.
Based on the number of cores we are going to use, the program will divide the job in a corresponding number of tasks.
Tasks will be initialized as **loadPics** objects, and their main objective will be to load the assigned pictures into a shared buffer.
To divide the picures among all the tasks, each task will be initialized with a start index and an end index.
The start and end indexes will tell the task which subset of images to load.

2 Implementation overview

The program is divided in two packages. The first one, called "interface", contains the **main** class, which defines the instructions to generate and handle the GUI. The second package, called "utilities", contains the classes that will be used to divide the work to do in different tasks and to execute them.

2.1 classes overview

1. **Package "interface"**:
   - **main**: This class contains the WindowsBuilder definition of the GUI and, furthermore, it contains the inizialization of the pictures buffer where the images will be loaded.
2. **Package "utilities"**
   - **Draw**: Defines a "draw" object's methods
   - **loadPics**: An element of this class represents a single task. To create a task we use the constructor of the class.

     When all the tasks have finished their work, the entire set of images will have been loaded.
   - **ParallelProcesses**: In this class we divide the work in equal parts to assign the job to the different tasks.
   - **Picture**: Picture object, which will be wisualized through th GUI.
   - **Utilities**: Class which contains various methods that will be used in the program.

3 Performance Analysis

3.1 Cost Analysis

The single-core program will execute a sequential loading of the pictures. Then, for N images, the loading time will be:

$$\sum_{i=1}^{N}$$

The multiple execution will divide instead the cost by the total number of cores. So, supposing that we will use X cores, each core will have to load $\frac{N}{X}$ images.

In this case tge execution time will become

$$\sum_{i=1}^{[\frac{N}{X}]+N(modC)} t_i$$

Given these formulas, the speedup will be given by calculating:

$$Speedup = \frac{T_{sequential}}{T_{parallel}}$$

If we want to define the full loading time for the set of pictures:

- **Sequential case**: $t_l * N$, where t_l is the maximum loading time for an image
- **Parallel case**: $t_l * ([\frac{N}{X}] + N(modC))$

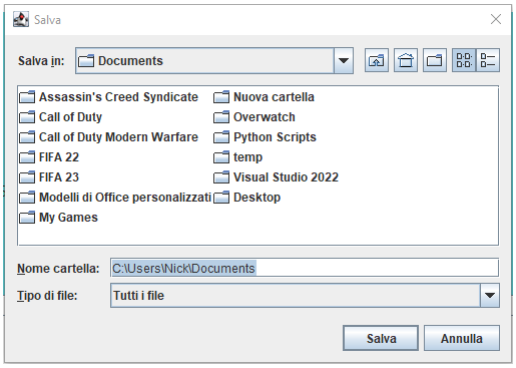
When we execute the program, we obtain the following results:

cores number	execution time (ms)	speedup
1	3193	1
2	1726	1,84
3	1300	2,45
4	1132	2,82
5	988	3,23
6	926	3.44

4 Execution test

An execution test of the program is now provided:

1. **Choose the directory** containing the images you want to open:



Once the list is selected, the list of images will get loaded will appear in the table below:

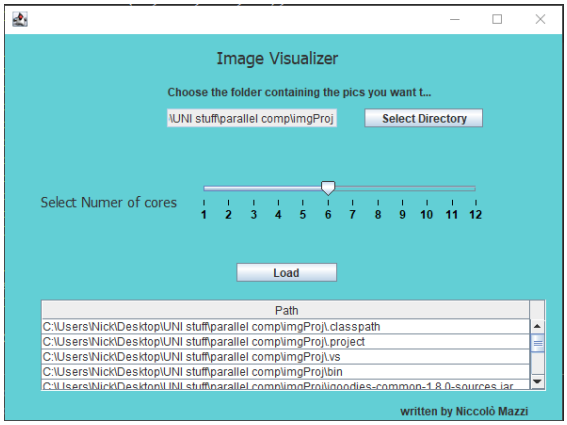
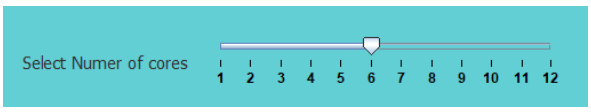


Figure 1: Caption

2. **Choose the number of cores** you want to use to load the images:



3. **Press "load" button** and wait for execution to terminate.

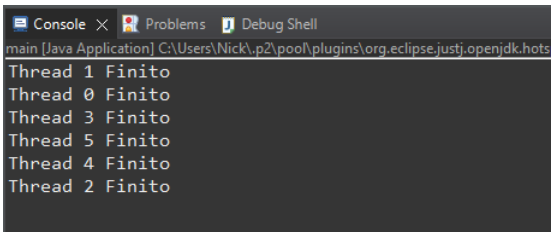
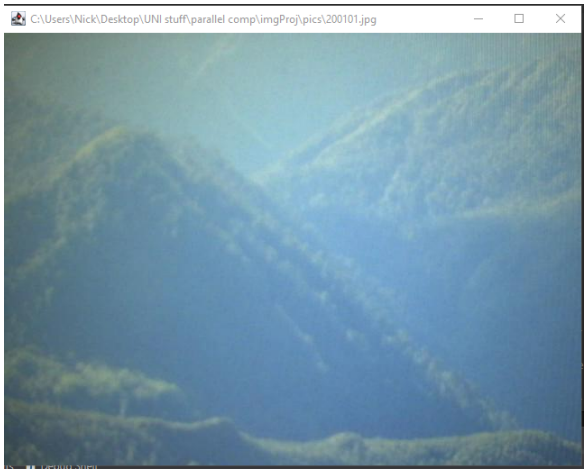


Figure 2: Console will notify when a thread finishes his task

4. **Click on the choosen figure in the table to visualize it**



5 Final considerations

The presented program doesn't present any deadlock condition or critical section simultaneous access problem. This is ensured by the fact that the program will for sure terminate his execution, by the absence of circular waiting conditions and by the fact that the only critical section is the shared data structure, which will therefore be sinchronized.