Sequential vs OpenMP Image Reader

Luca Leuter
E-mail address
luca.leuter@stud.unifi.it

Abstract

In this paper, we will analyze the performance of a program for reading images, in both its sequential and parallel versions, the latter is implemented using the OpenMP library.

1. Introduction

One of the most commonly performed operations on a PC is to read images and perform operations on them. This project focuses on the first part, following two approaches for reading JPEG images: a sequential and a parallel one.

The language used is C++. In particular, the following libraries were used:
OPENCV: an open-source library that provides functions for reading images
OPENMP: a library that allows code parallelization

2. Implementation

Several tools were used for the implementation of the code. In particular, was essential the use of the *imread* function of the OPENCV library, which allows loading an image into an object of type *Mat*, which contains a matrix with the pixel values of each image. It was used inside another method, to allow the saving of multiple images in an array. Once they have been read, they are made available by invoking another method.

To evaluate the performance of the two different approaches, the standard libraries ⟨CHRONO⟩ were used to keep track of time and ⟨FILESYSTEM⟩ to perform operations with system folders.

2.1. Sequential Approach

The sequential implementation is very simple. The function *loadImagesSequential()* loads the images onto an empty array of type Mat using the *imread()* function inside a for loop that stops when all the images in the *imageStrings* string have been loaded.

```
Mat* images;
void loadImagesSequential(vector<string>
    imageStrings){
   images = new Mat[imageStrings.size()];
   for(int i=0; i<imageStrings.size(); i++)
      images[i] = imread(imageStrings[i]);
}
```

The function *getImagesSequential()* returns the loaded images when they are needed.

```
Mat* getImagesSequential(){
   return images;
}
```

2.2. Parallel Approach

Thanks to the OPENMP library and the way the code is structured, even this approach becomes easy to implement. Analyzing the sequential code, we can note that the problem is embarrassingly parallel. In fact, the portion of the code to be parallelized using the PRAGMA OMP PARALLEL directive of the OPENMP library is the for loop for loading the images.

```
Mat* images1;
void loadImagesParallel(vector<string>
    imgStrings){
#pragma omp parallel private()
   // Parallel loading of images1
   images1 = new Mat[imgStrings.size()];
#pragma omp parallel for
   for(int i=0; i<imgStrings.size(); i++)
      images1[i] = imread(imageStrings[i]);
}
```

As with the sequential approach, here too there is a function *getImagesParallel()* which returns the loaded images.

```
Mat* getImagesParallel(){
   return images;
}
```

3. Performance and Test

To evaluate the performance, two types of tests were implemented: the first one by increasing the number of threads used in the parallel implementation; the second one by increasing the number of loaded images.

3.1. Test 1

For the first test, the first 500 images of the dataset are considered. A vector of strings containing the file names is created using the function *directory_iterator()*, which will then be used by the *imread* function to read the images.

```
int maxThreads = omp_get_max_threads();
for (auto &p:
    filesystem::directory_iterator(imagePath))
   imageStrings.emplace_back(p.path().string());
   vector<string> imageStringsSubset(
   imageStrings.begin(),imageStrings.begin()
       + subsetDimension);
   int sequentialTime = 0;
```

Then *nTest* are performed on the sequential loading of images. The stopwatch is started and the time in milliseconds to perform the loading is calculated. Then an average is taken and the result is saved in a log file.

```
for (int i = 0; i < nTest; i++) {
   Mat *images;
   auto start =
       chrono::system_clock::now();
   loadImagesSequential(imageStringsSubset);
   images = getImagesSequential();
   auto end =
       std::chrono::system_clock::now();
   auto elapsed =
       chrono::duration_cast<chrono::milliseconds>(end
       - start);
   sequentialTime += elapsed.count();
   delete[] images;
}
logFile << "Tempo medio trascorso per il
    caricamento delle immagini in modo
    sequenziale:\n"
      << sequentialTime / nTest << "\n";
```

Finally, the test on parallel loading is executed. The first for loop allows to perform tests by increasing the number of threads each time using the function *omp_set_num_threads()*. For each of these loops, *nTests* are executed on parallel loading and the times of executions in milliseconds are calculated. The average is then computed and the result is logged in a file.

```
for(int k = 2; k < (4*maxThreads + 1);
    k++) {
   int parallelTime = 0;
   // Cambia il numero di thread
   omp_set_num_threads(k);
   // Caricamento parallelo delle immagini
   for (int i = 0; i < nTest; i++) {
      Mat *images;
      auto start =
          std::chrono::system_clock::now();
      loadImagesParallel(imageStringsSubset);
      images = getImagesParallel();
      auto end =
          std::chrono::system_clock::now();
      auto elapsed =
          std::chrono::duration_cast<std::chrono::millise
          - start);
      parallelTime += elapsed.count();
      delete[] images;
   }
   cout << "Tempo medio trascorso per il
       caricamento delle immagini in modo
       parallelo: "
      << parallelTime / nTest << "\n";
   logFile << parallelTime / nTest << "\n";
}
```

3.2. Test 2

For the second test a subset of images is considered each time until all the images are loaded. For the sequential version of the test, we start with 400 images and increase by 50 each time until the last one. The loading time is calculated, and the results are saved in a log file.

```
for (int i =
    (int)imageStrings.size()/2.5; i <
    imageStrings.size(); i+=50) {
   Mat *images;
   sequentialTime = 0;
   vector<string>
       imageStringsTest(imageStrings.begin(),
       imageStrings.begin() + i);
   auto start =
       chrono::system_clock::now();
   loadImagesSequential(imageStringsTest);
   images = getImagesSequential();
   auto end =
       chrono::system_clock::now()
   auto elapsed =
       chrono::duration_cast<chrono::milliseconds>
       (end - start);
   sequentialTime += elapsed.count();
   delete[] images;
   cout << "Tempo trascorso per il
       caricamento di " << i << "
       immagini in modo sequenziale: "
       << sequentialTime << "\n";
   logFile2 << sequentialTime << "\n";
}
```

For the parallel version of the test, we set the number of threads used to the maximum possible and execute the same loop as the sequential test: we start from 400 images and increase by 50 every time until the last one. The loading time is then calculated, and the results are saved in a log file.

```
omp_set_num_threads(maxThreads);
for (int i =
    (int)imageStrings.size()/2.5; i <
    imageStrings.size(); i+=25) {
   int parallelTime = 0;
   Mat *images;
   vector<string>
       imageStringsTest(imageStrings.begin(),
       imageStrings.begin() + i);
   auto start =
       chrono::system_clock::now();
   loadImagesParallel(imageStringsTest);
   images = getImagesParallel();
   auto end =
       chrono::system_clock::now();
   auto elapsed =
       chrono::duration_cast<chrono::milliseconds>
      (end - start);
   parallelTime += elapsed.count();
   delete[] images;
   logFile2 << parallelTime<< "\n";
}
```

3.3. Setup

The tests were run on a PC with an AMD A8-6410 processor with 4 cores and 4 threads. The images are 1000 and have a resolution ranging from 720p to 1080p.

3.4. Results

The results of the first test, in which the number of images is fixed at 500 and the number of threads is increased incrementally, are shown in Figure 1.

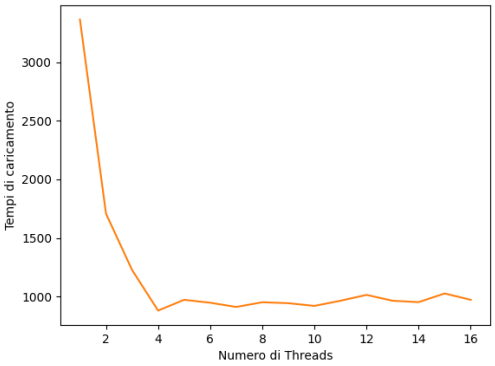

Figure 1. Comparison of loading times for 5000 images while increasing the number of threads

The test produced several speedups which are visible in Figure 2.

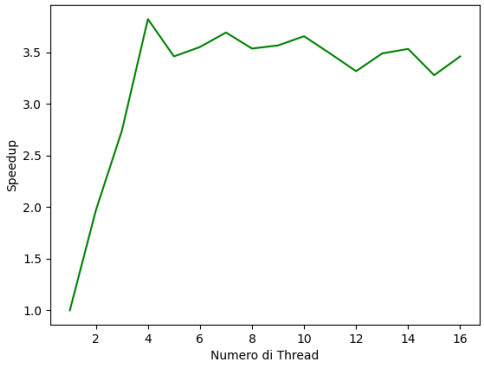

Figure 2. Speedup of the test increasing the number of threads.

The graph shows that the speedup initially increases as the number of threads increases, reaching its maximum when there are 4 threads (which happens to be the maximum number for the machine). Afterwards, it decreases, oscillating between various values, always better than the initial ones but lower than when the number of threads is 4.

The results of the second test, where the sequential loading and the parallel loading with the number of threads set to the maximum (in this case 4) are compared by increasing the number of images, are shown in Figure 3.

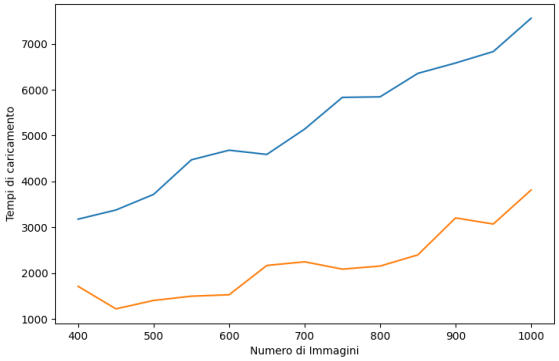

Figure 3. Comparison between loading times of an increasing number of images

The test produced various speedups, visible in Figure 4.

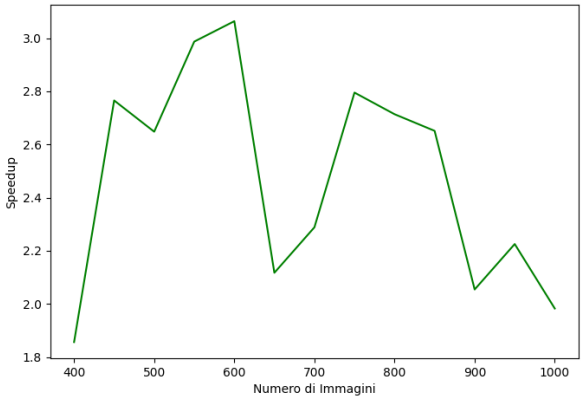

Figure 4. Speedup of the test increasing the number of images

We can observe from the graph that the speedup does not follow a precise logic but oscillates in values ranging from 1.5 to 3.2.

4. Conclusion

In this project, a comparison has been done between a sequential version and a parallel version for loading images using the OpenCV library. From the results of the tests, it was found that increasing the number of threads beyond the limit does not lead to an improvement, therefore it is always better to keep them at the maximum possible for the machine. Furthermore, it has also been seen that increasing the number of images does not necessarily result in an improvement in speedup. The project is available at this link: `https://github.com/Luca0079CM/Image_Reader`