Sequential vs OpenMP Image Reader

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

Abstract

In questo paper verranno analizzate le performace di un programma per leggere delle immagini, nella sua versione sequenziale e parallela. Quest'ultima implementata usando la libreria OpenMP

1. Introduction

Una delle operazioni più eseguite su un PC è quella di leggere immagini e di eseguire operazioni su di esse. Questo progetto si concentra sulla prima parte, seguendo 2 approcci per la lettura di immagini JPEG: una sequenziale e una parallela.

Il linguaggio utilizzato è il C++. In particolare sono state usate le librerie:

OPENCV: un libreria opensource che fornisce funzioni per la lettura di immagini

OPENMP: una libreria che permette la parallelizzazione del codice

2. Implementation

Per l'implementazione del codice sono stati usati diversi tool. In particolare è risultato fondamentale l'uso della funzione *imread* della libreria OPENCV, la quale permette il caricamento di un immagine all'interno di un oggetto di tipo *Mat*, che contiene una matrice con i valori dei pixel di ogni immagine. Essa è stata usata all'interno di un altro metodo, per permettere il salvataggio di più immagini all'interno di un array. Una volta che esse sono state lette vengono rese disponibili tramite l'invocazione di un altro metodo.

Per valutare le performance dei due diversi approcci sono state usate le librerie standard ⟨CHRONO⟩ per tenere traccia del tempo e ⟨FILESYSTEM⟩ per effettuare operazioni all'interno delle cartelle.

2.1. Sequential Approach

L'implementazione sequenziale è molto semplice.
La funzione *loadImagesSequential()* carica su un array vuoto di tipo Mat le immagini usando la funzione *imread()* all'interno di un ciclo for che si ferma quando tutte le immagini all'interno della stringa *imageStrings* sono state caricate.

```
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]);
}
```

La funzione *getImagesSequential()* restituisce le immagini caricate quando si ha bisogno di esse

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

2.2. Parallel Approach

Grazie alla libreria OPENMP e il modo in cui è stato strutturato il codice, anche questo approc-

cio diventa semplice da implementare. Analizzando il codice sequenziale si nota che il problema è imbarazzantemente parallelo. Infatti la porzione di codice da parallelizzare usando la direttiva PRAGMA OMP PARALLEL della libreria OPENMP è il ciclo for per il caricamento delle immagini.

```
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]);
}
```

Come per l'approccio sequenziale anche qua si ha una funzione *getImagesParallel()* la quale restituisce le immagini caricate

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

3. Performance and Test

Per valutare le performance sono stati eseguiti 2 tipologie di test: il primo all'aumentare del numero di Core utilizzati nell'implementazione parallela; il secondo all'aumentare del numero di immagini caricate.

3.1. Test 1

Per il primo test vengono considerate le prime 200 immagini del dataset. Viene creato una vettore di stringhe contenenti i nomi dei file usando la funzione *directory_iterator()*, che poi verrà utilizzato dalla funzione *imread* per leggere le immagini.

```
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;
```

Successivamente vengono eseguiti *nTest* test sul caricamento sequenziale delle immagini. Viene fatto partire il cronometro e calcolato il tempo in millisecondi per eseguire il caricamento. Viene poi fatta una media e il risultato salvato in un file di log

```
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";
```

Infine viene eseguito il test sul caricamento parallelo. Il primo ciclo for permette di eseguire i test aumentando ogni volta il numero di thread con la funzione *omp_set_num_threads()*. Per ognuno di questi cicli vengono eseguiti *nTests* test sul caricamento parallelo e vengono calcolati i tempi in millisecondi. Viene poi fatta la media e il risultato caricato in un file di log

```
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

Per il secondo test invece viene considerato ogni volta un sottoinsieme delle immagini fino ad arrivare a tutte le immagini. Per la versione sequenziale del test si parte da 400 immagini e si aumenta ogni volta di 50 fino all'ultima. Si calcola il tempo di caricamento e si salvano i risultati in un file di log

```
    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";
    }
```

Per la versione parallela del test si setta il numero di core usati al massimo possibile e si esegue la stesso ciclo del test sequenziale: si parte da 400 immagini e si aumenta ogni volta di 50 fino all'ultima. Viene calcolato poi il tempo di caricamento e si salvano i risultati in un file di log

```
    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

I test sono stati eseguiti su un pc con un processore AMD A8-64101 avente 4 core e 4 thread. Le immagini sono 1000 e hanno una risoluzione che varia dai 720p ai 1080p.

3.4. Results

I risultati del primo test, in cui si tiene fisso il numero di immagini a 500 e si aumenta via via il numero di thread è visibile nella Figura 1.

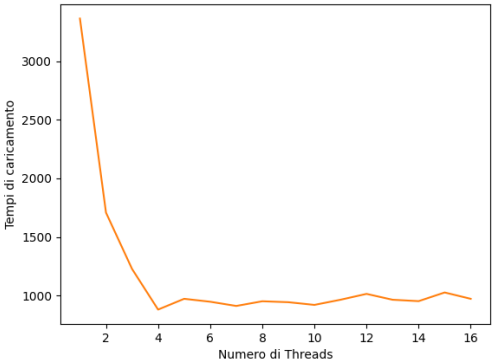

Figure 1. Confronto tra i tempi di caricamento di 5000 immagini aumentando il numero di thread

Il test ha prodotto diversi speedup che sono visibili nella Figura 2.

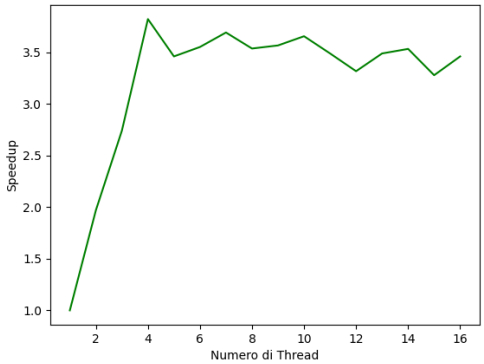

Figure 2. SpeedUp del test aumentando il numero di thread

Si nota dal grafico che lo speedup inizialmente cresce aumentando il numero di thread, fino a raggiungere il massimo quando essi sono 4 (che risulta essere il numero massimo per la macchina). Successivamente decresce oscillando tra vari valori, sempre migliori dei primi ma inferiori rispetto a quando il numero di thread è 4.

I risultati del secondo test, in cui si confronta il caricamento sequenziale e il caricamento con il numero di thread settato al massimo (in questo caso 4) aumentando il numero di immagini è visibile in Figura 3.
Il test ha prodotto diversi speedup, visibili in Figura 4.

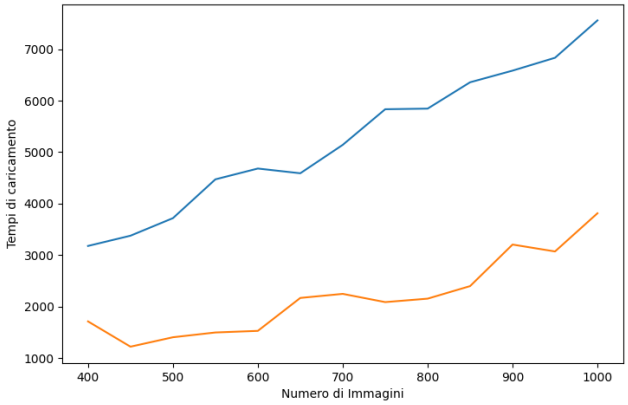

Figure 3. Confronto tra i tempi di caricamento di un numero di immagini crescente

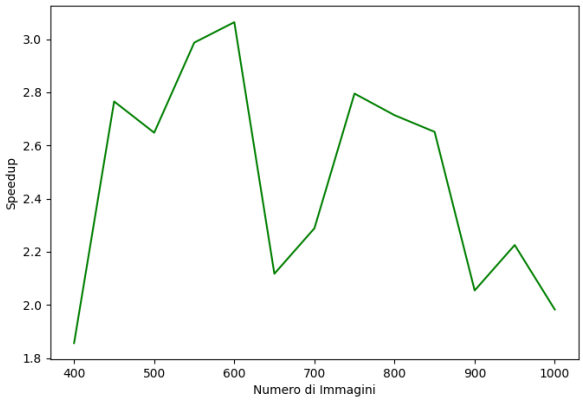

Figure 4. SpeedUp del test aumentando il numero di immagini

Si nota dal grafico che lo speedup non segue una logica precisa ma oscilla in valori compresi tra 1.5 e 3.2.

4. Conclusion

In questo progetto è stato fatto il confronto tra una versione sequenziale e una versione parallela per il caricamento delle immagini, usando la libreria OpenCV. Dai test risulta che aumentando il numero di thread oltre il limite non si ottiene un miglioramento, dunque conviene sempre tenerli al massimo possibile della macchina. Inoltre si è visto anche che aumentare il numero di immagini non si traduce in un miglioramento dello speedup.