UNIVERSITÀ
DEGLI STUDI
FIRENZE

Parallel Image Composition

Nicolò Pollini, Francesco Fantechi

A.A. 2022-2023

UNIVERSITÀ DEGLI STUDI FIRENZE

Indice

Obiettivo

- Parallelizzare un algoritmo di Data Augmentation che effettua composizioni di immagini
- Utilizzando:
  - Framework OpenMP
  - Librerie Python che permettono il multiprocessing
- Confrontare i vari metodi valutando gli speedup ottenuti rispetto alle loro esecuzioni sequenziali

Data Augmentation

- Insieme di processi atti ad aumentare gli elementi di un dataset senza raccogliere nuovi dati
- Tecnica che trova un largo impiego nell'addestramento delle reti neurali quando i dati a disposizione non sono numericamente sufficienti

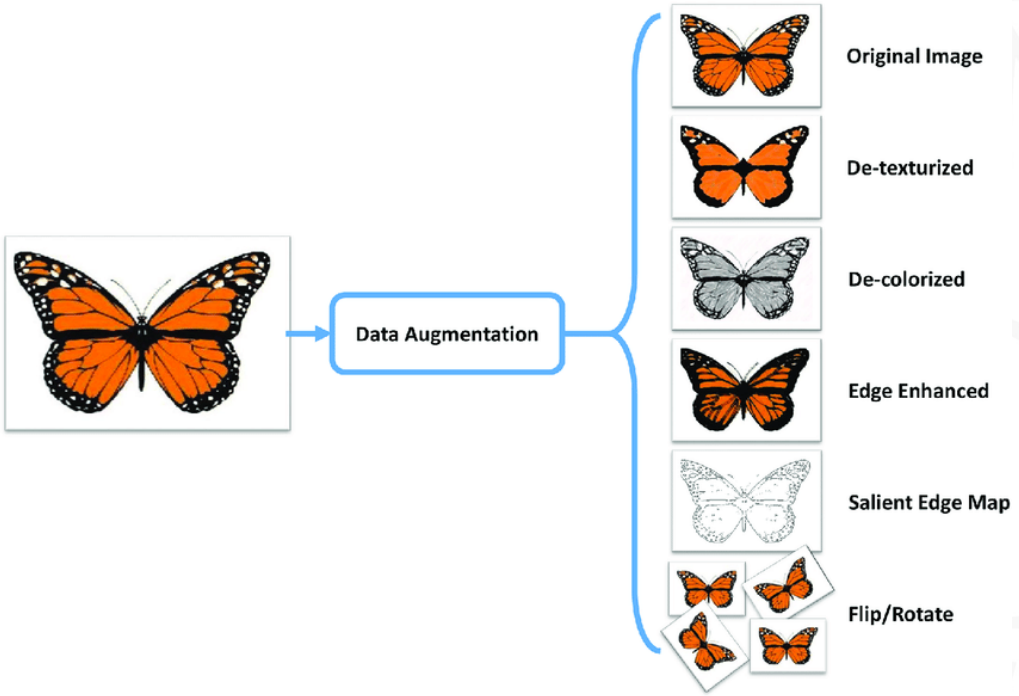
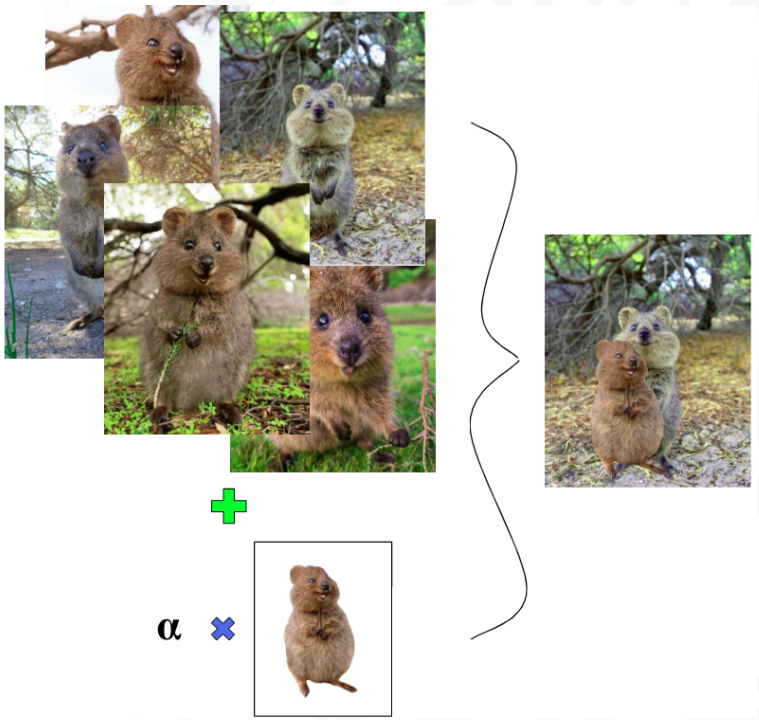


Image Composition

Nel nostro esperimento:

- Dataset: 5 immagini di background di dimensione 483×594 pixel raffiguaranti dei quokka
- Data Augmentation: mediante Image Composition applicando un'ulteriore immagine di dimensione 298×350 pixel



Codice e implementazione

- Progetto implementato in linguaggio C++ e Python
- Codice versionato tramite la piattaforma GitHub:
  - `https://github.com/francesco-ftk/Image_Composition_OpenMP.git`
  - `https://github.com/francesco-ftk/Image_Composition_Multiprocessing.git`
- Eseguito su due diverse macchine con due diversi OS:

Machine 1			
OS	CPU	Number of Core with Hyper-Threading	RAM
Windows 10	Intel(R) Core(TM) i7-8750H	12	16 GB
Machine 2			
OS	CPU	Number of Core with Hyper-Threading	RAM
Ubuntu 20.04.5	Intel(R) Core(TM) i7-1165G7	8	16 GB

UNIVERSITÀ DEGLI STUDI FIRENZE

Codice e implementazione

- Il progetto è stato suddiviso in tre parti:
  - Implemenatazione dell'agoritmo di Image Composition sequenziale
  - Parallelizzazione dell'algoritmo tramite il framework OpenMP in C++
  - Parallelizzazione dell'algoritmo tramite le librerie di multiprocessing in Python

Image Composition Sequenziale

- Parametro *"transformations"* indicativo del numero di nuove immagini volute
- Per *"transformations"* volte:
  - Presa un'immagine fra quelle possibili di background in modo randomico
  - Aggiunta in posiszione randomica l'immagine di foreground con una trasparenza uniformemente scelta nell'intervallo $[128, 255]$
  - Immagini lette, editate e salvate tramite la libreria OpenCV

Estratto Codice Python

Image Composition Sequenziale

```python
def data_augmentation(foreground, backgrounds, transformations) -> int:
    # generate the directory for saving ...
    for i in range(0, transformations, 1):
        index = random.randint(0, len(backgrounds) - 1)
        background = copy(backgrounds[index])
        # check the dimensions between foreground and background ...
        if background.shape[0] - foreground.shape[0] == 0:
            row = 0
        else:
            row = random.randint(0, background.shape[0] - foreground.shape[0] - 1)
        if background.shape[1] - foreground.shape[1] == 0:
            col = 0
        else:
            col = random.randint(0, background.shape[1] - foreground.shape[1] - 1)
        alpha = random.randint(128, 255)
        alpha_correction_factor = float(alpha) / 255.0
        beta_correction_factor = 1.0 - alpha_correction_factor
        for j in range(0, foreground.shape[0], 1):
            for k in range(0, foreground.shape[1], 1):
                f_pixel = foreground[j, k]
                b_pixel = background[row + j, col + k]
                f_alpha = float(f_pixel[3]) / 255.0
                if f_alpha > 0.9:
                    background[row + j, col + k] = [float(b_pixel[0]) * beta_correction_factor + float(
                        f_pixel[0]) * alpha_correction_factor * f_alpha,
                                                    float(b_pixel[1]) * beta_correction_factor + float(
                                                        f_pixel[1]) * alpha_correction_factor * f_alpha,
                                                    float(b_pixel[2]) * beta_correction_factor + float(
                                                        f_pixel[2]) * alpha_correction_factor * f_alpha, 255]
        # save the new image ...
```

UNIVERSITÀ DEGLI STUDI FIRENZE

OpenMP

- Framework OpenMP
- Permette di parallelizzare una porzione di codice in modalità implicit threading attraverso delle direttive “pragma”
- Ogni thread genera una porzione delle immagini di output richieste tramite il parametro *“transformations”*
- Algoritmo imbarazzantemente parallelo
- Numero di thread usati calcolato aggiungendo un 50% al numero complessivo di Core posseduti dalla macchina utilizzata considerando l’Hyper-Threading

UNIVERSITÀ DEGLI STUDI FIRENZE

Estratto Codice C++

OpenMP

```
#pragma omp parallel default(none) shared(foreground, backgrounds) firstprivate(transformations, outputFolderPath)
{
    #pragma omp for
    for (int count = 0; count < transformations; ++count) {
        // copy a random background from the shared pool
        int index = (int) (uniform(gen) % backgrounds.size());
        cv::Mat background = backgrounds.at(index).clone();
        // pick a random position for the foreground
        int row;
        if (background.rows - foreground.rows == 0) { row = 0; }
        else { row = (int) (uniform(gen) % (background.rows - foreground.rows)); }
        int col;
        if (background.cols - foreground.cols == 0) { col = 0; }
        else { col = (int) (uniform(gen) % (background.cols - foreground.cols)); }
        // pick a random alpha value
        int alpha = 128 + uniform(gen) % 128;
        float alphaCorrectionFactor = (float) alpha / 255;
        float betaCorrectionFactor = 1 - alphaCorrectionFactor;

        for (int i = 0; i < foreground.rows; ++i) {
            for (int j = 0; j < foreground.cols; ++j) {
                cv::Vec4b &f_pixel = foreground.at<cv::Vec4b>(i, j);
                cv::Vec4b &b_pixel = background.at<cv::Vec4b>(row + i, col + j);
                float f_alpha = (float) f_pixel[3] / 255;
                if (f_alpha > 0.9) {
                    b_pixel[0] = (unsigned char) ((float) b_pixel[0] * betaCorrectionFactor +
                                                  (float) f_pixel[0] * alphaCorrectionFactor * f_alpha);
                    b_pixel[1] = (unsigned char) ((float) b_pixel[1] * betaCorrectionFactor +
                                                  (float) f_pixel[1] * alphaCorrectionFactor * f_alpha);
                    b_pixel[2] = (unsigned char) ((float) b_pixel[2] * betaCorrectionFactor +
                                                  (float) f_pixel[2] * alphaCorrectionFactor * f_alpha);
                }
            }
        }
        std::string outputPath = outputFolderPath + "/out_" + std::to_string(count) + ".png";
        cv::imwrite(outputPath, background);
    }
}
```

UNIVERSITÀ DEGLI STUDI FIRENZE

Multiprocessing

- Interprete Python GIL (Global Interpretr Lock)
- In Python non è possibile eseguire piú di un thread contemporaneamente
- $\rightarrow$ Sottoprocessi al posto dei thread
- Molto piú costosi rispetto a far partire dei thread
- Due metodi differenti per istanziare i sottoprocessi:
  - Libreria standard di Python Multiprocessing
  - Libreria Joblib
- Numero di processi istanziati calcolato aggiungendo un 50% al numero complessivo di Core posseduti dalla macchina utilizzata considerando l'Hyper-Threading

UNIVERSITÀ DEGLI STUDI FIRENZE

Multiprocessing

- Libreria Multiprocessing
- Permette di istanziare e far partire manualmente i processi su una certa funzione target
- Lavoro equamente diviso fra i vari processi
- Parametro *“transformations_for_process”* ottenuto dividendo il numero di nuove immagini volute per il numero di processi istanziati
- Due implementazioni equivalenti del suo utilizzo implementate:
  - Multiprocessing
  - Multiprocessing con Pooling

Estratto Codice Python

Multiprocessing

```
print("START Multiprocessing Algorithm")
print("Using " + str(PROCESSES) + " processes")

trasformations_for_process = math.ceil(TRANSFORMATIONS / PROCESSES)
local_date = datetime.datetime.now()
new_dir_path = 'output/' + str(local_date)
os.mkdir(new_dir_path)
processes = [Process(target=data_augmentation_multiprocessing,
                     args=(foreground, backgrounds, new_dir_path, trasformations_for_process,)) for i in
             range(PROCESSES)]
start = time.time()
[p.start() for p in processes]
[p.join() for p in processes]
end = time.time()

print(f'Multiprocessing Running took {end - start} seconds.')
```

UNIVERSITÀ DEGLI STUDI FIRENZE

Estratto Codice Python

Pooling Multiprocessing

```
# MULTIPROCESSING IMAGE COMPOSITION
print("START Pool Multiprocessing Algorithm")

# pool_size = multiprocessing.cpu_count() * 2
pool_size = PROCESSES
print("Using " + str(pool_size) + " processes")
pool = multiprocessing.Pool(
    processes=pool_size,
)
trasformations_for_process = math.ceil(TRANSFORMATIONS / pool_size)
start = time.time()
local_date = datetime.datetime.now()
new_dir_path = 'output/' + str(local_date)
os.mkdir(new_dir_path)
# prepare arguments iterable for pool.starmap
args = [(foreground, backgrounds, new_dir_path, trasformations_for_process) for i in range(pool_size)]
pool.starmap(data_augmentation_multiprocessing, args)

pool.close()  # no more tasks
pool.join()  # wrap up current tasks

end = time.time()
print(f'Pool Multiprocessing Running took {end - start} seconds.')
```

UNIVERSITÀ DEGLI STUDI FIRENZE

Joblib

- Libreria Joblib
- Processi eseguiti in modo asincrono sulla propria funzione target suddividendosi il lavoro in modo autonomo
- Ad ogni nuova esecuzione di un processo gli argomenti posseduti devono essere ripassati in ingresso alla funzione
- Per ridurre i costi ogni processo genera una nuova immagine alla volta e riceve già negli argomenti l'immagine di background da utilizzare e non l'intero dataset da cui scegliere

Estratto Codice Python

Joblib

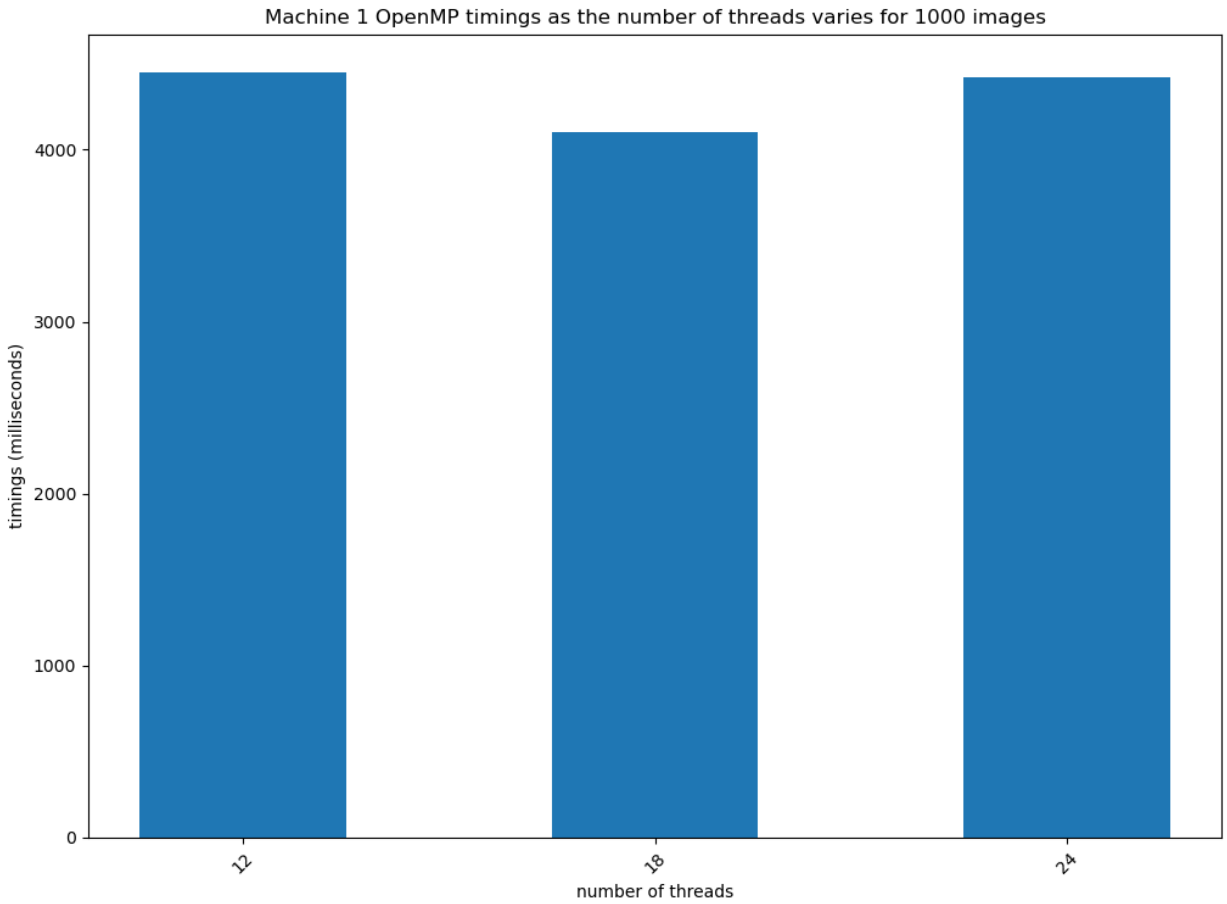
```
# JOBLIB MULTIPROCESSING IMAGE COMPOSITION
print("START Multiprocessing Joblib Algorithm")

start = time.time()
local_date = datetime.datetime.now()
new_dir_path = 'output/' + str(local_date)
os.mkdir(new_dir_path)
index = random.randint(0, len(backgrounds) - 1)
background = backgrounds[index]
Parallel(n_jobs=PROCESSES)(
    delayed(data_augmentation_multiprocessing_one_at_a_time)(foreground, background, new_dir_path, i) for i
    in range(TRANSFORMATIONS))
end = time.time()

print(f'Multiprocessing Joblib Running took {end - start} seconds.')
```

Risultati C++

Timing OpenMP Macchina 1 al variare del numero dei thread



Risultati C++

Timings e Speedup Macchina 1 OpenMP

- 18 thread

Table: C++ Timings

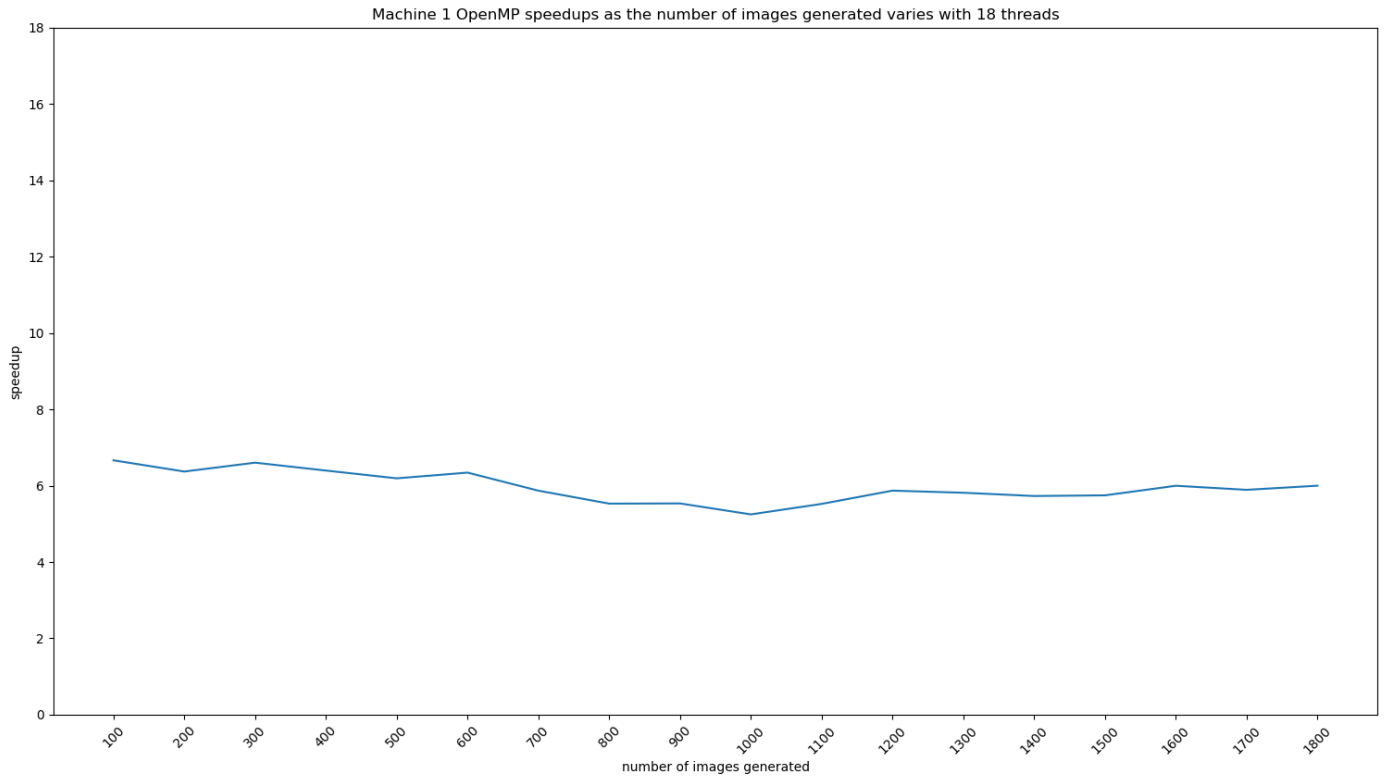
Machine 1		
Number of Images	Sequential	OpenMP
1800	88974*ms*	14832*ms*

Table: Speedups

Machine 1
OpenMP
6.0

Risultati C++

Speedup OpenMP Macchina 1 al variare del numero di immagini generate con 18 thread



Risultati Python

Timings e Speedup Macchina 2 Multiprocessing

- 12 processi

Table: Python Timings

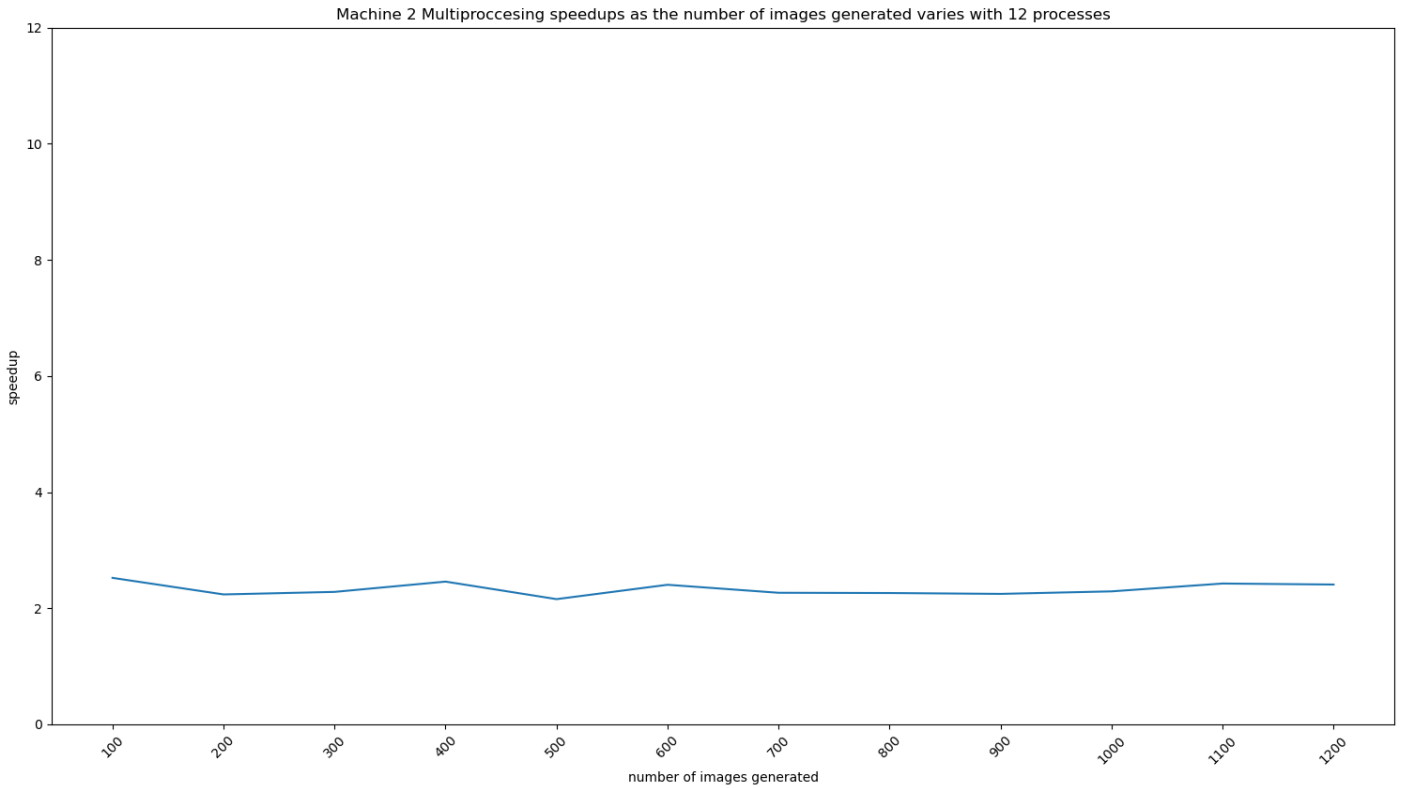
Machine 2				
Number of Images	Sequential	Joblib	Pool Multiprocessing	Multiprocessing
1200	122.1*s*	143.9*s*	52.1*s*	50.7*s*

Table: Speedups

Machine 2		
Joblib	Pool Multiprocessing	Multiprocessing
0.9	2.3	2.4

Risultati Python

Speedup Multiprocessing Macchina 2 al variare del numero di immagini generate con 12 processi



UNIVERSITÀ DEGLI STUDI FIRENZE

Conclusioni

- Speedup sublineare sia con OpenMP che con la libreria Multiprocessing
- Speedup con la libreria Multiprocessing nettamente inferiore a quello ottenuto con OpenMP
- Nessuno speedup tramite l'utilizzo della libreria Joblib