Studio dei benefici ottenuti applicando asincronia su un lettore di immagini

Jacopo Sabatino
jacopo.sabatino@edu.unifi.it

Abstract

In questo articolo è stato implementato un lettore di immagini JPEG. L'obiettivo è verificare se l'applicazione di asincronia sulle operazioni di I/O del lettore comportano benefici prestazionali. Le prestazioni ottenute sono state confrontate con le corrispettive di un lettore in versione sequenziale.

1. Introduzione

Questo elaborato ha l'obiettivo di sviluppare un lettore di immagini JPEG che tramite l'utilizzo dell'asincronia sia più efficiente sulle operazioni di I/O.

Sono quindi state implementate due versioni del lettore:

- sequenziale: utile come base su cui applicare asincronia e come punto di confronto prestazionale;
- parallela: dove le operazioni I/O effettuate nella versione sequenziale sono parallelizzate con l'asincronia.

2. Implementazione

Il lettore è realizzato come classe in cui le immagini sono mantenute in una lista. Ogni istanza del lettore conserva anche la locazione delle immagini da leggere. L'indirizzo è passato durante la creazione dell'istanza.

I metodi sella classe sono:

- *read_images*: legge le immagini dalla locazione salvata, le converte in formato bitmap e le inserisce nella lista. Come argomento si può indicare il numero di immagini che si desidera leggere dalla memoria. Se l'argomento non è passato, sono lette tutte le immagini presenti all'indirizzo;
- *get_all_images*: restituisce la lista di immagini;
- *show_images*: mostra a schermo le immagini della lista. Come argomento si può indicare il numero di immagini che si desidera visualizzare. Se l'argomento non è passato, saranno visualizzate tutte le immagini;
- *save_images*: salva le immagini della lista all'indirizzo passato come argomento;
- *close_images*: chiude tutte le immagini mantenute nella lista.

Per l'operazione di lettura sono stati presi degli accorgimenti che ne hanno migliorato le prestazioni:

- le immagini sono aperte in formato di lettura binario con la funzione built-in Python *open()* prima di essere aperte con la libreria **PIL**;
- le immagini sono trasformate in un in-memory binary stream prima di essere aperte con **PIL**.

2.1. Versione parallela

Il parallelismo è stato implementato solo per le operazioni di lettura e salvataggio, che sono di tipo I/O. Per queste operazioni, il parallelismo si ottiene con l'asincronia, cioè facendo richiesta dell'operazione e permettendo al core di svolgere altri compiti nell'attesa del completamento. Per fare ciò è stata utilizzata la libreria Python **asyncio**.

Grazie a questa libreria, i due metodi che implementano lettura e salvataggio sono stati trasformati in coroutine. Le coroutine sono un oggetto **awaitable** su cui altre coroutine possono mettersi in attesa. Nei test effettuati, una coroutine *main()* è stata messa in attesa delle due operazioni. A loro volta, le due operazioni sono in attesa del completamento di tutte le operazioni I/O che devono effettuare.

Per gestire le immagini, è stata utilizza la libreria **PIL**. I metodi di questa libreria sono bloccanti, perciò mettersi in **await** su questi non permette di ottenere asincronia.

Per superare questo problema ed ottenere quindi asincronia, è stata utilizzata la libreria **multiprocessing**. Con questa libreria è stato possibile generare un pool di processi a cui far eseguire le operazioni I/O. Utilizzare un pool di processi, permette di sfruttare più core logici contemporaneamente. Così facendo è possibile eseguire più operazioni in parallelo anche se queste risultano bloccanti, perché ciò che viene bloccato è il singolo core logico utilizzato dal processo. Naturalmente questa soluzione ha un parallelismo ottenibile limitato al numero dei core logici

presenti sulla macchina. Le operazioni I/O che i processi devono eseguire sono racchiuse in **future** ed inserite nell'event loop **asyncio** dalla funzione *run_in_executor*. I **future** sono raccolti da una lista passata ad un *gather*. Fare ciò permette di mettersi in attesa dell'oggetto restituito dal *gather*, il quale è **awaitable**.

Tutti i passaggi illustrati sono state applicati allo stesso modo per entrambe le operazioni parallelizzate.

Il lettore nella versione parallela ha un attributo in più rispetto al corrispondente sequenziale: **num_workers**. Questo attributo è inizializzato alla creazione dell'oggetto e indica il numero di processi da generare nel pool quando si effettua una delle due operazioni asincrone.

La lista contenente le immagini è istanziata utilizzando un **multiprocessing.Manager()**. Tale **Manager** è un processo che permette di creare e mantenere strutture dati condivise fra processi. Una struttura dati creata in questo modo, è gestita in maniera tale che non si verifichino conflitti fra gli aggiornamenti dei processi.

3. Test e confronti

Ogni configurazione dei test è stata eseguita dieci volte, cosicché i valori utilizzati nei confronti fra le due versioni siano più vicini al caso medio.

Le immagini utilizzate durante i test sono una composizione dei seguenti dataset:

- **JPEG Melanoma 128x128**
- **Concrete & Pavement Crack Dataset**
- **Gemstones Images**

3.1. Test al variare del numero di immagini

I test sono stati effettuati con le seguenti quantità di immagini: 5k, 10k, 25k, 40k, 60k.

Il numero di processi generati dalla versione parallela è stato fissato a 20, pari al numero dei core logici della macchina utilizzata.

Dalla **Figura 1** si può notare che, quantità minime di immagini non fanno registrare un guadagno in prestazioni da parte della versione parallela, ma anzi le peggiorano (speedup < 1). Ciò è causato dal costo aggiuntivo dovuto alla creazione e gestione dei processi, il quale prevale sul beneficio di un loro utilizzo.

Per quantità di immagini elevate invece la versione parallela fa registrare un buon guadagno prestazionale. L'operazione di lettura, essendo molto costosa, beneficia maggiormente di una parallelizzazione rispetto all'operazione di salvataggio.

3.2. Test al variare del numero di processi

I test sono stati effettuati con le seguenti quantità di processi: 5, 10, 15, 20, 30, 40.

La quantità di immagini è stata fissata a 60k. Avere un'elevata quantità di immagini aumenta i tempi di esecuzione dei metodi, permettendo così di valutare con maggior efficacia la differenza fra le configurazioni.

Dalla **Figura 2** si può notare che l'andamento dello speedup cresce fino ad un numero di processi uguale a 10 per poi diminuire. Il motivo per cui l'ottimo si ottiene per un numero di processi inferiore a quello dei core logici (20), potrebbe essere dovuto a processi non generati dal lettore. Sulla macchina solitamente eseguono anche processi non generati dal lettore. Quindi i processi del lettore, oltre a poter entrare in concorrenza fra loro, potrebbero entrare in concorrenza anche con questi altri processi.

Lo speedup per l'operazione di lettura ha valori più alti rispetto a quello di scrittura perché, come già detto, essendo un'operazione più costosa ed essendo effettuata su una elevata quantità di dati, permette di osservare maggiori benefici sulle prestazioni.

4. Conclusioni

I test effettuati ci permettono di affermare che l'utilizzo dell'asincronia sulle operazioni I/O di un lettore di immagini permette di migliorarne le prestazioni.

Perché il guadagno sia effettivo, la quantità di dati su cui si applica asincronia deve essere elevata, così da giustificare il costo di generazione dei processi.

Il numero di processi per ottenere maggior benefici dipende dal numero di core logici e dal numero di processi che girano sulla macchina al momento di esecuzione del lettore. Per i test effettuati, che sono stati eseguiti sempre in condizioni il più simili possibile, il numero di processi ideale risulta essere pari a 10.

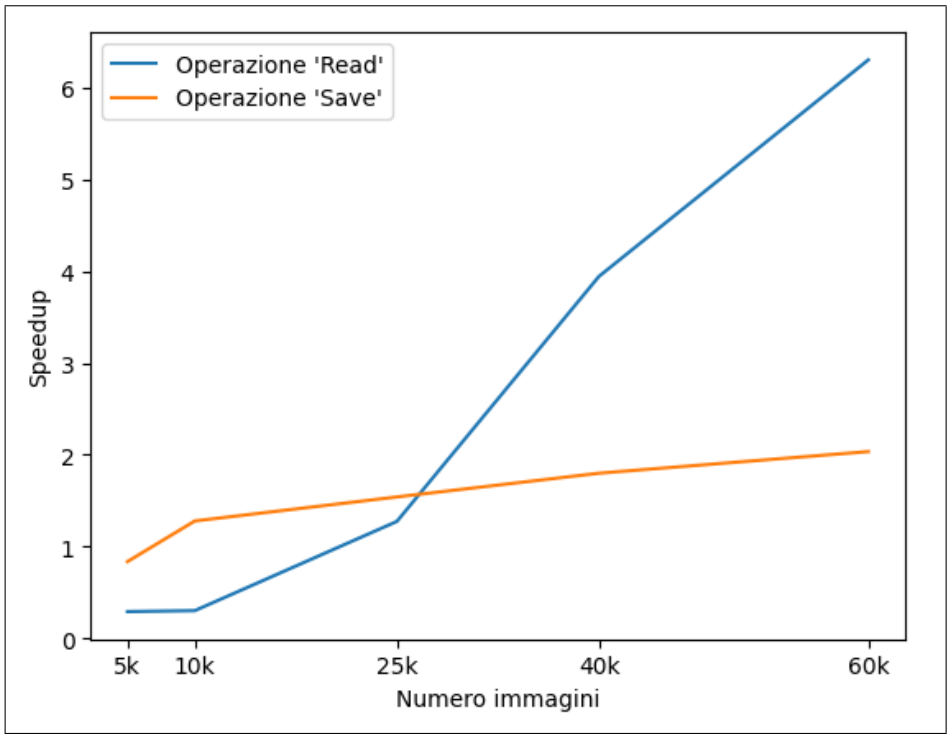

Figura 1: Speedup fra versione sequenziale e parallela al variare del numero di immagini. Numero di processi fissato a 20.

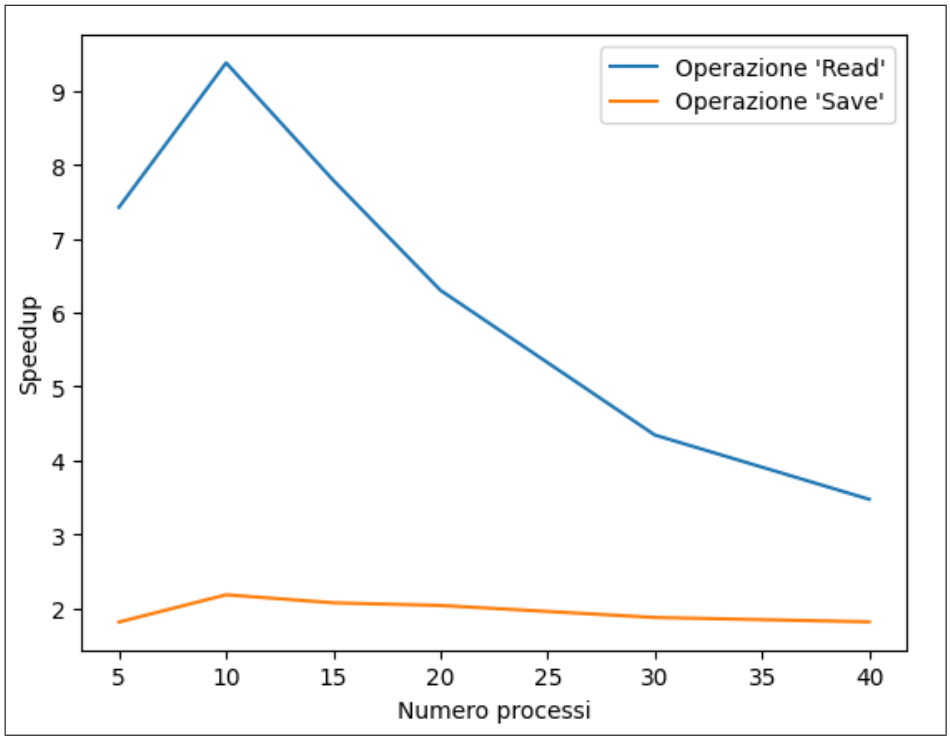

Figura 2: Speedup fra versione sequenziale e parallela al variare del numero di processi. Quantità di immagini fissata a 60k.