Integral Image Parallelo

Mattia Matucci
E-mail address
mattia.matucci@stud.unifi.it

Abstract

L'algoritmo di Integral Image presa in input una matrice di numeri restituisce il suo integrale. Analizzando la definizione di integrale di una matrice ci siamo accorti di come sia possibile implementare un algoritmo parallelo che ricalca la definizione con estrema facilità. Questa implementazione è tuttavia poco efficiente, per questo motivo presentiamo un'alternativa, che nella sua versione sequenziale ha già prestazioni migliori. La parallelizzazione di questo algoritmo alternativo è tuttavia meno efficiente di quella sequenziale, in quanto richiede ai processi di accedere ripetutamente in scrittura in memoria condivisa. Per migliorare le prestazioni abbiamo quindi deciso di parallelizzare a livello di matrice, ovvero facciamo partire un pool di processi, dove ciascun processo integra sequenzialmente una matrice dopo che è stata copiata nella propria memoria.

1. Introduzione

L'algoritmo di Integral Image presa in input una matrice di numeri restituisce il suo integrale. Definiamo integrale di una matrice X una nuova matrice Y che ha in ciascuna cella $Y_{i,j}$ la somma delle celle contenute nella sottomatrice di X di dimensione $\mathbb{R}^{i \times j}$.

Dalla definizione appena data possiamo intuire che ogni cella della matrice in output può essere calcolata indipendentemente dalle altre. Inoltre possiamo osservare che gli accessi sulla matrice in input sono solo in lettura, visto che in output è restituita una nuova matrice. Grazie a queste osservazioni, l'algoritmo che possiamo definire ricalcando la definizione può essere parallelizzato molto semplicemente.

In questa relazione invece, andiamo a proporre un algoritmo che ha una complessità lineare. Tale guadagno comporta una parallelizzazione più difficile dal momento che le celle della matrice in output non sono più indipendenti tra loro.

L'intuizione dietro l'algoritmo che proponiamo è riportata nella figura 1. L'idea è quella di calcolare la cella $Y_{i,j}$ a partire dalle celle precedenti di Y.

Nella figura di sinistra abbiamo evidenziato le celle necessarie per calcolare il valore in output della cella blu. Il valore è ottenibile sommando al valore contenuto nella cella stessa quello delle celle arancione e gialla, sottraendo il valore contenuto nella cella verde. Riscrivendo quanto detto in termini di i, j otteniamo:

$$y_{i,j} = y_{i,j} + y_{i-1,j} + y_{i,j-1} - y_{i-1,j-1}$$

Utilizzando la figura di destra possiamo spiegare il motivo del corretto funzionamento di questa formula. Come prima cosa dobbiamo dire che il valore della cella arancione della figura a sinistra è ottenuto sommando i valori delle celle verdi e arancioni della figura di destra, analogamente otteniamo il valore della cella gialla della figura di sinistra. È quindi necessario sottrarre al risultato parziale il valore delle celle verdi per considerarle un'unica volta, il quale è contenuto nella cella verde dell'immagine sinistra.

Per essere parallelizzato questo algoritmo bisogna attraversare la matrice in diagonale, in quanto tutti i valori delle celle di una diagonale possono essere calcolati a partire dalle due diagonali precedenti.

Senza addentrarci troppo nelle scelte implementative per parallelizzare questo algoritmo, diciamo subito che il tempo d'esecuzione della versione parallela di questo algoritmo è molto maggiore a quello dell'esecuzione sequenziale. Tra le cause ciò, la necessità per i processi di lavorare su un unica matrice condivisa, di conseguenza la necessità di sincronizzarsi.

La parallelizzazione è quindi avvenuta a livello di matrice, cioè ciascun processo lavora sequenzialmente in modo indipendente dagli altri processi su una matrice. In questo modo non è necessario lavorare con processi su una memoria condivisa, e la sincronizzazione è richiesta una volta terminata l'esecuzione su tutte le matrici.

In questo progetto abbiamo utilizzato la libreria JobLib per parallelizzare le operazioni CPU bound e la libreria Asyncio per quelle I/O bound, ad esempio per leggere il dataset. Le altre librerie utilizzate nel codice sono Numpy, per rappresentare le matrici, e Pillow per la gestione delle immagini.

Figura 1. Nell'immagine di sinistra sono evidenziate le celle della matrice utilizzate per calcolare il valore della cella blu. Nell'immagine di destra è riportata la matrice in input, che è utilizzata per spiegare

La scrittura e l'esecuzione dell'algoritmo proposto e stata realizzata su un sistema Windows 11 con CPU Intel i7-10710U con 6 core fisici e 12 thread e una RAM da 16 GB. La versione di Python utilizzata è la 3.10.

1.1. Dataset

Dal momento che ogni processo lavora indipendentemente su una matrice, abbiamo deciso di utilizzare un dataset di immagini per generare le matrici su cui lavorare. Le immagini utilizzate sono contenute nella cartella *images* all'interno della directory del progetto.

2. Implementazione

In questa sezione andremo a spiegare dettagliatamente le scelte implementative adottate, prestando particolare attenzione alle differenze tra versione sequenziale e parallela. Inoltre accenneremo anche altre scelte inizialmente perseguite, che però non hanno portato ai risultati previsti. Questa trattazione sarà effettuata per tutte le operazioni compiute nel codice a partire dalla lettura delle immagini contenute nel dataset.

2.1. Lettura del dataset

La prima operazione che descriviamo è la lettura delle immagini del dataset. Questa operazione è stata implementata utilizzando la libreria Pillow, la quale è stata anche usata per convertire le immagini da RGB alla scala dei grigi. Fatto ciò le immagini sono state interpretate come un array di Numpy.

Nel codice abbiamo definito due metodi distinti per compiere questa operazione, uno per la versione sequenziale e uno per quella parallela. Abbiamo dovuto definire due funzioni distinte poiché nella versione parallela abbiamo usato la libreria Asyncio, che permette di effettuare operazioni di input/output utilizzando il multithreading.

All'interno della funzione parallela viene definito un task per ciascuna immagine. Il task avvolge la coroutine, la quale preso il percorso dell'immagine restituisce la matrice associata. Per lanciare i tasks abbiamo utilizzato la funzione *gather*, che ha il vantaggio di raccogliere i risultati di tutte le coroutine, restituendoli in una lista.

2.2. Funzione Integral Image

Il passo successivo, una volta ottenuta la lista delle matrici, è quello di passare ogni matrice alla funzione di Integral Image. Abbiamo già visto nell'introduzione, che esistono numerosi algoritmi per ottenere l'integrale di una matrice. Adesso presentiamo quello che è stato implementato.

Prima di iniziare ricordiamo che abbiamo scelto di applicare contemporaneamente a tutte le matrici l'algoritmo sequenziale, quando eseguiamo il codice parallelo. Questa precisazione è importante poiché ci permette di sapere in anticipo qual è l'ordine di attraversamento di una matrice. Grazie a ciò possiamo procedere per righe e non per diagonale.

Oltre a poter attraversare la matrice per righe, parallelizzare a livello di matrice permette ai processi di non lavorare sulla memoria condivisa. JobLib crea per ogni processo una copia della matrice in input nella memoria del processo. In questo modo otteniamo un vantaggio a livello di prestazioni, dal momento che non si verifica race condition per l'accesso alla memoria, tuttavia la copia generata non può essere acceduta in scrittura. Per ovviare a questo problema copiamo all'interno della funzione Inegral Image la matrice in input e lavoriamo su questa, applicando la formula sottostante ad ogni cella della matrice.

$$y_{i,j} = y_{i,j} + y_{i-1,j} + y_{i,j-1} - y_{i-1,j-1}$$

3. Esecuzione

In questa sezione analizzeremo l'esecuzione del codice, con particolare attenzione allo speedup ottenuto dall'esecuzione parallela. Nel corso della sezione sarà divisa la trattazione della parallelizzazione della lettura del dataset, da quella dell'operazione di Integral Image. Eviteremo di trattare nuovamente la correttezza della formula utilizzata nel codice, così come il risultato complessivo, che viene testato all'interno del codice.

3.1. Lettura Dataset

La prima operazione che abbiamo parallelizzato è la lettura del dataset. Le dimensioni del dataset scelto non hanno permesso registrare, in media, tempi di esecuzione differenti tra la versione sequenziale e parallela della funzione.

Abbiamo poi testato solo questa parte con datasets più ampi, i tempi registrati questa volta differivano di un fattore di circa 6. I due dataset testati avevano rispettivamente 1000 e 10000 immagini al loro interno.

3.2. Integral Image

Come abbiamo già spiegato prima, abbiamo scelto di parallelizzare a livello di matrice, perché i risultati ottenuti parallelizzando l'algoritmo erano peggiori a causa dell'accesso in contemporanea di più processi alla memoria condivisa. In questa parte non verranno presentati i risultati raccolti con questa prima soluzione, ma solo quelli ottenuti eseguendo parallelamente su più matrici l'algoritmo sequenziale. I tempi di esecuzione registrati e lo speedup ottenuto sono riportati nelle figure 2 e 3 rispettivamente.

In entrambe le figure possiamo osservare tre linee: la blu riporta i risultati misurati sulle prime 10 immagini del dataset, l'arancione rappresenta i risultati ottenuti sulle prime 15 immagini del datast e la verde che mostra i risultati dell'intero dataset.

I tempi e lo speedup sono stati calcolati solo sulle configurazioni ritenute più interessanti. Le configurazioni parallele sono state scelte considerando la macchina utilizzata, abbiamo quindi deciso di utilizzare 2, 6, 10, 12, 13, 18 jobs. Abbiamo usato l'algoritmo sequenziale per calcolare i valori rappresentati nei grafici con 1, mentre gli altri sono stati misurati variando il valore della variabile *nJobs* parametro dell'istruzione *Parallel*.

Analizzando il grafo dei tempi di esecuzione possiamo osservare che le linee hanno un andamento simile, questo implica che il tempo di esecuzione dipende dal numero di immagini e che il tempo di esecuzione scala allo stesso modo nonostante cambi la dimensione del dataset.

Dal grafico dello speedup è infatti possibile vedere che questo è massimo quando il numero di jobs è prossimo a quello dei thread del computer. Un comportamento inatteso è la comunque buona performance nel momento in cui definiamo un numero di jobs superiori a quello dei thread del computer.

3.3. Conclusioni

Concludendo possiamo osservare che nonostante non sia possibile migliorare le prestazioni quando in input abbiamo una sola matrice, possiamo in caso di più matrici parallelizzare molto semplicemente attraverso la libreria JobLib, visto che l'esecuzione è indipendente per ogni matrice.

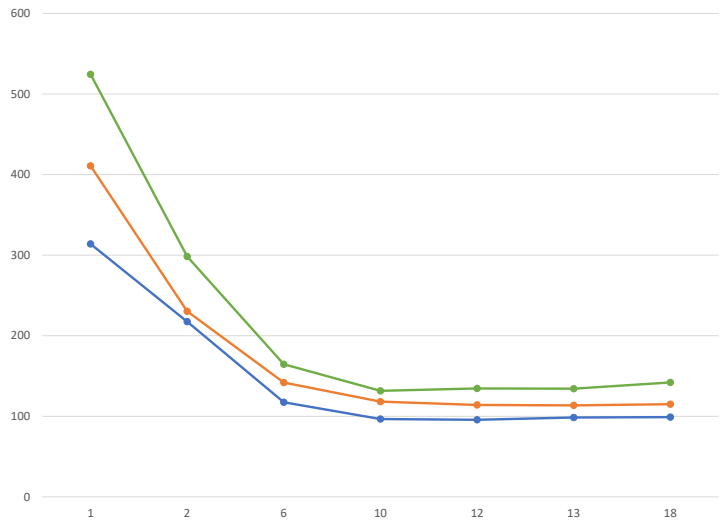

Figura 2. Tempo d'esecuzione registrato su 10 (blu), 15 (arancione) e 20 (verde) immagini

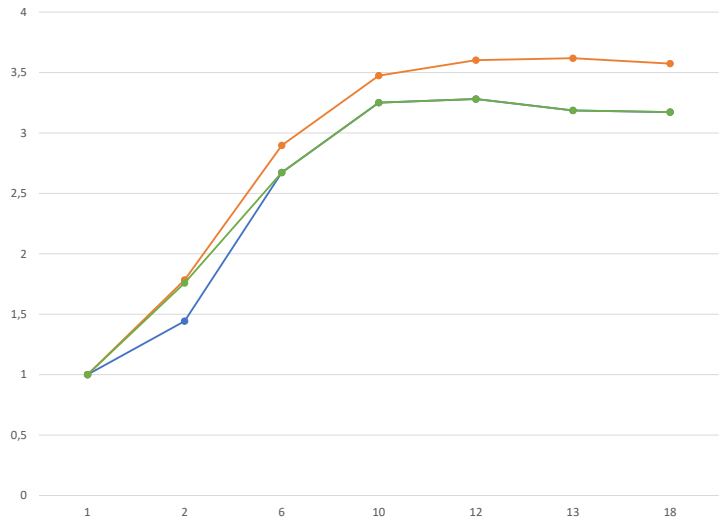

Figura 3. SpeedUp calcolato su 10 (blu), 15 (arancione) e 20 (verde) immagini