PC-2020/21 Elaborato Mid-Term

Andrea Neri
andrea.neri1@stud.unifi.it

Abstract

Il progetto “Mid-Term” prevede l’implementazione di un Image Reader in cui l’utente possa definire una cartella da cui prelevare le immagini facendo sì che il software si occupi di caricarle in modo non bloccante per renderle visibili. L’elaborato è stato sviluppato in Java e prevede una implementazione sequenziale e una parallela, in modo da verificarne tempistiche di esecuzione e SpeedUp tra le due versioni.

1. Introduzione

Le specifiche del progetto erano quelle di realizzare un algoritmo non bloccante per il caricamento e la visualizzazione di immagini contenute all’interno di una cartella.

In particolare, la richiesta era quella di realizzare il progetto sia in versione sequenziale sia in versione parallela (multithread).

L’utente una volta avviato il caricamento può richiedere di visualizzare una certa immagine del set.

Nel caso in cui l’immagine sia già caricata nella memoria principale la potrà visualizzare, altrimenti riceverà un errore che lo invita a riprovare più tardi.

Il progetto è stato realizzato in Java (JDK 15), per la realizzazione della GUI è stato utilizzato Swing.

2. Interfaccia grafica

L’utente può interagire con il programma attraverso una GUI visibile in Figura 1 e Figura 2.

Grazie all’interfaccia l’utente può specificare da quale cartella intende caricare le foto, specificare con quanti thread avviare il caricamento e, all’interno della tabella, potrà vedere l’elenco di tutte le foto.

Nel dettaglio il funzionamento del programma è il seguente:

1) Si specifica la cartella in cui sono presenti le immagini
2) Si specificano, tramite il menù a tendina, il numero di thread con cui si andrà ad eseguire il programma
3) Nella tabella verranno visualizzati tutti i path assoluti delle foto
4) Quando verrà cliccato il tasto load, il programma inizia a caricare le foto. Durante il caricamento l’utente può aprire le foto con un doppio click su una riga della tabella.
5) Se la foto è già stata caricata in memoria principale la foto verrà visualizzata (Figura 3)

5-Bis) Se la foto ancora non è presente in memoria principale l’utente riceverà un messaggio di errore, che lo invita a riprovare più tardi (Figura 4).

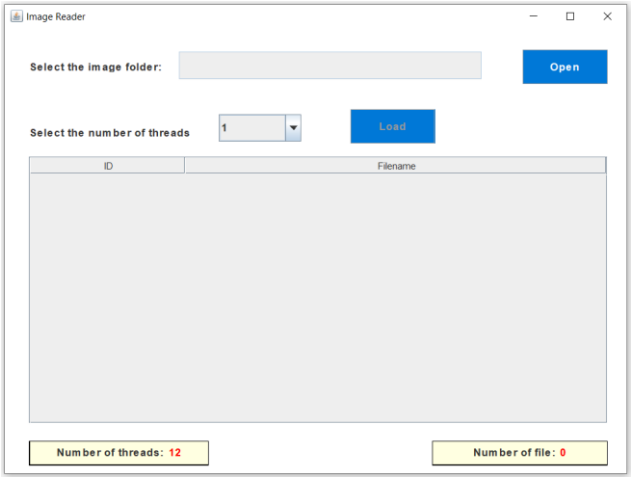


Figura 1: Interfaccia del programma all’apertura

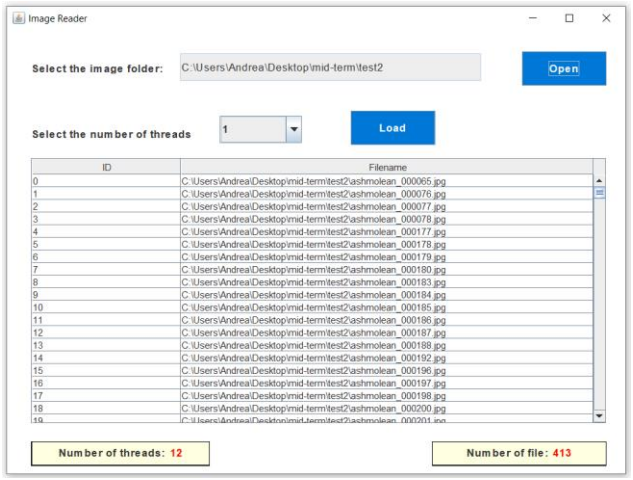


Figura 2: interfaccia del programma quando viene selezionata la cartella contenente le immagini

Nella parte bassa della finestra sono presenti due comode label che riportano il numero di thread disponibili sulla macchina e il totale delle immagini presenti nella cartella selezionata.

Figura 3: visualizzazione di un'immagine

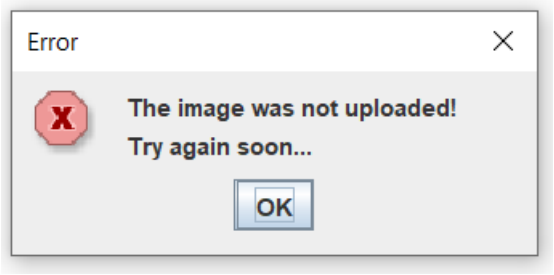


Figura 4: messaggio di errore nel caso l'immagine non sia stata ancora caricata

3. Implementazione

L'implementazione del programma si basa sul paradigma di programmazione parallela *Fork-Join*.

Una volta che l'utente specifica la cartella da cui caricare le immagini e il numero di thread da utilizzare, la classe Worker ha il compito di creare tanti Task quanti sono i thread scelti.

Ogni task riceverà due indici *startIndex* e endIndex, che rappresentano gli estremi del sottoinsieme creato a partire dall'insieme delle immagini totali. Ogni singolo Task avrà il compito di caricare all'interno del buffer condiviso e sincronizzato (di tipo *synchronizedList*) le immagini contenute nel proprio subset.

Prima di riportare con precisione l'implementazione di ogni singola classe, per rendere più chiara la struttura implementativa si riporta la struttura dei vari package:

- `core`
  - `Image`
  - `Task`
  - `Worker`
- `main`
  - `Main`
  - `MainGUI`
- `utils`
  - `Utils`

3.1. Classe Image

La classe si occupa della creazione di un oggetto di tipo Image con due attributi:

- *path (String)* che rappresenta il path assoluto di un'immagine all'interno del file system
- *icon (ImageIcon)* utile per inserire il titolo nel frame che mostrerà l'immagine.

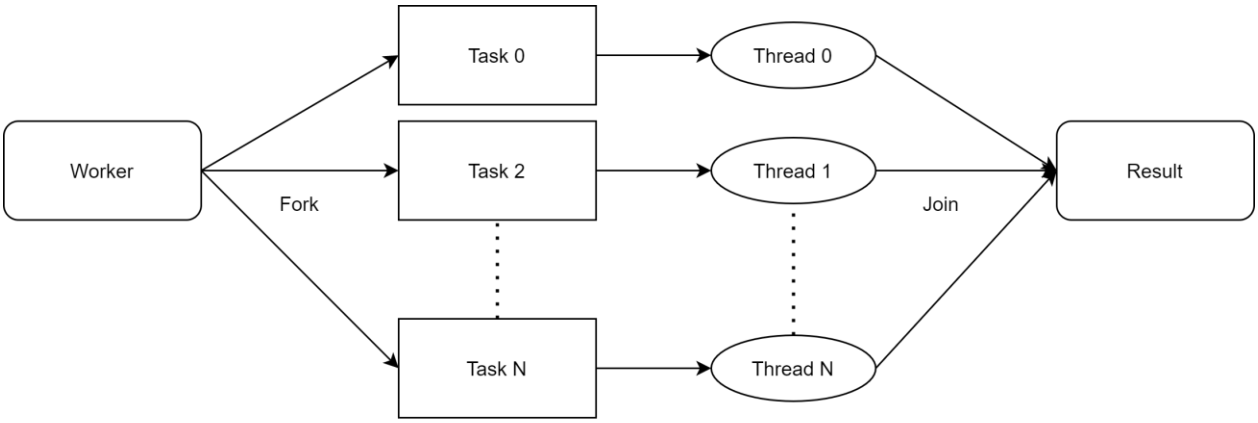


Figura 5: paradigma fork-join

L'inizializzazione del path con la stringa *empty* è utile per riuscire a capire se un'immagine è stata realmente caricata o se ancora non lo è.

Oltre alla creazione, la classe si occupa di costruire il frame su cui visualizzare un'immagine quando viene richiesta dall'utente con il metodo *draw*().

3.2. Classe Task

La classe estende RecursiveTask che permette di implementare e definire un singolo Task, ovvero un compito all'interno del paradigma Fork-Join.

Ogni Task ha il compito di caricare un certo numero di immagini predefinito all'interno di un buffer condiviso e sincronizzato tra tutti i task.

Attributi:

- *ID (int)* rappresenta l'identificativo del Task
- *path(File[])* l'array condiviso tra tutti i task che contiene tutte le immagini da caricare (cartella del file system). Ogni task accederà solo alla porzione assegnata.
- *sharedBuffer(List<Image>)* buffer condiviso tra tutti i task in cui verranno caricate le immagini
- *startIndex(int), endIndex(int)* rispettivamente l'indice iniziale e finale del sottoinsieme delle immagini che dovrà caricare uno specifico Task.

Il metodo fondamentale della classe è l'override del metodo *compute(),* grazie a questo metodo ogni Task andrà a caricare nel buffer condiviso le immagini e restituirà *True* non appena avrà concluso il proprio lavoro.

```
protected Boolean compute() {
    for (int i = startIndex; i < endIndex; i++){
        this.sharedBuffer.set(i, new
                        Image(path[i].getPath()));
    }
    System.out.println("[Thread " + this.ID + "]: " +
                       "finished");
    return true;
}
```

Figura 6: metodo compute*()*

3.3. Classe Worker

Questa è la classe che si occupa di prendere in carico tutto il lavoro e dividerlo nei vari task, seguendo il paradigma fork-join.

Attributi:

- *path(String)* percorso del filesystem da cui caricare le immagini
- *sharedBuffer(List<image>)* buffer condiviso tra tutti i Task
- *images(File[])* array contenente tutte le foto da caricare nel buffer
- *threadNumber(int)* numero di thread da utilizzare durante l'esecuzione del programma

Il metodo principale che applica il paradigma è il metodo *start().* Questo metodo, se necessario divide il set di immagini tra i vari thread, esegue le funzioni join e fork e calcola il tempo di esecuzione.

Possiamo dividere in due parti distinte il metodo: sezione sequenziale e sezione parallela. Vediamo ora in dettaglio entrambe le sezioni.

3.3.1 Sezione sequenziale

```
if (threadNumber == 1) {
    Task singleTask = new Task(0, images,
                sharedBuffer, 0, totalFile);
    long startTime =
                System.currentTimeMillis();
    singleTask.fork();
    singleTask.join();
    long endTime =
                System.currentTimeMillis();

    return;
}
```

Poiché l'implementazione è sequenziale si crea un singolo thread che riceverà e dovrà caricare l'intero set di immagini.

3.3.2 Sezione parallela

Nella sezione parallela si esegue una divisione del set di immagini si procede alla divisione dell'insieme delle immagini da caricare in N (numero di thread) sottoinsiemi disgiunti di immagini.

Nel caso in cui l'insieme non sia divisibile per il numero di thread, il thread N-1 riceverà più immagini da caricare.

Si eseguono in successione in due foreach distinti per le funzioni fork e join e si valutano i tempi di esecuzione.

Per facilitare la lettura il codice parallelo è presente nella sezione 6 appendice.

3.4. Classi Main e MainGUI

È la classe che si occupa di avviare e far terminare tutto il processo di caricamento.

Questa classe si occupa di creare il buffer condiviso e sincronizzato disponibile all'interno della libreria Collections di Java, in particolare si è fatto uso della lista *synchronizedList.* Successivamente crea un thread di tipo Worker, specificando la cartella da cui caricare le immagini, il buffer condiviso e il numero di core da utilizzare.

Worker crea tanti Task quanti ne ha richiesti l'utente, divide il lavoro come visto precedentemente ed esegue la fork. A questo punto tutte le immagini vengono caricate in modo non bloccante e l'utente le potrà visualizzare.

Il thread Worker esegue la join su ogni singolo Task, una volta che tutti i Task hanno concluso il lavoro, il main andrà a terminare il thread Worker.

La classe MainGUI, ha lo stesso comportamento della classe Main, ma in questa classe si genererà anche una finestra in cui l'utente potrà scegliere la directory da cui caricare le immagini e il numero di thread da utilizzare.

3.5. Classe Utils

Questa classe contiene metodi statici di utilità utilizzati per creare una lista di file presenti in una data cartella del file system, contare i file presenti in una determinata cartella, ottenere il numero di core della macchina e due metodi per la creazione di messaggi all'utente.

4. Versione con Buffer Privato

Per avere una metrica di paragone tra l'utilizzo di variabili condivise e private la prima implementazione è stata modificata in modo che ogni Task avesse il proprio buffer privato su cui caricare le immagini.

Una volta eseguito il join su tutti i task il Worker ha il compito di riunire tutti i singoli buffer privati in unico buffer.

I cambiamenti rispetto alla versione descritta in precedenza sono all'interno della classe Task e della classe Worker.

Le istanze della classe Task non utilizzeranno più un buffer di tipo *synchronizedList*, ma utilizzeranno un *ArrayList* privato, di conseguenza il metodo compute non restituirà un booleano, ma restituirà l'*ArrayList* riempito con le immagini appartenenti al proprio sottoinsieme.

La classe Worker lavora nello stesso modo della versione precedente, ma alla creazione di un Task non fornirà più il buffer sul quale andare a caricare le foto, ma sarà il task stesso a crearselo.

Un'altra modifica viene applicata dopo la conclusione di tutti i Task, all'interno di un foreach tutti i buffer privati vengono riuniti in un singolo *ArrayList*, come visibile nel codice qui sotto.

```
for (ForkJoinTask<ArrayList<Image>> task :
    tasks) {
    ArrayList<Image> img = task.invoke();
    imageToLoad.addAll(img);
}
```

Risultati e considerazioni

I test sono stati effettuati sulla seguente macchina:

- CPU -> AMD Ryzen 2600 (6 core / 12 thread)
- RAM -> 16 GB DDR4 3200 MHz
- SDD -> Sabrent SB-ROCKET-512

Le immagini utilizzati per i test sono disponibili al seguente link:
https://www.robots.ox.ac.uk/~vgg/data/oxbuildings/.

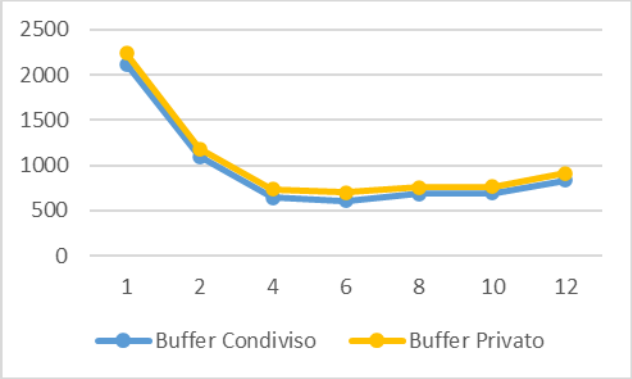
Per avere dei dati meno influenzati dal sistema operativo e dallo scheduler, ogni singolo test è stato eseguito 5 volte, il valore riportato nelle tabelle è la media di tali valori.

Nota: tutti i tempi sono espressi in millisecondi.

Il primo test è fatto su un insieme di 127 foto, i risultati sono i seguenti:

# Thread	Buffer Condiviso	Buffer Privato
1	2118	2243,4
2	1100,4	1185,2
4	643,6	733,2
6	611	700,4
8	689,4	759
10	693,6	765,4
12	834,6	913,4

Utilizzando i dati per costruire un grafico Tempo/Numero di thread si ottiene il seguente grafico:



Sia dalle tempistiche nella tabella sia dal grafico è possibile notare come il numero massimo di thread per avere la massima efficienza dell'algoritmo parallelo è 6.

In particolare, utilizzando più di 6 thread il miglioramento che si ottiene utilizzando un algoritmo parallelo viene perso per la gestione dei singoli thread a livello di sistema operativo.

Lo Speedup massimo ottenuto con 6 core è pari a 3.467 quando si utilizza un buffer condiviso, e 3.203 quando si utilizzano i buffer privati.

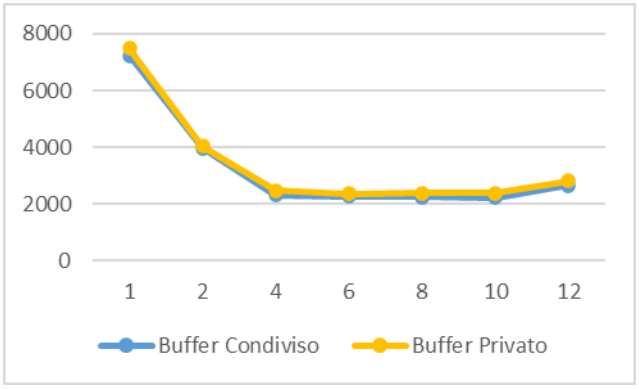
Per valutare se la dimensione del set di immagini poteva in qualche modo influenzare le tempistiche di esecuzione, il set di immagini utilizzato nel primo test è stato arricchito, portandolo a contenere 413 elementi.

I risultati ottenuti sono i seguenti:

# Thread	Buffer Condiviso	Buffer Privato
1	7237	7488,4
2	3963,2	4045,6
4	2305,2	2465,8
6	2258,2	2355
8	2239,6	2375,4
10	2229,2	2370,4
12	2639,6	2804

Lo Speedup massimo ottenuto con 10 core è pari a 3.246 quando si utilizza un buffer condiviso, e 3.159 quando si utilizzano i buffer privati.

Utilizzando i dati per costruire un grafico Tempo/Numero di thread si ottiene il seguente grafico:



Aumentando la dimensione del set di immagini da caricare, si può notare come è possibile aumentare il numero di thread da utilizzare per continuare ad avere un miglioramento nei tempi di esecuzione. In particolare, si può notare come con l'utilizzo di 8 e di 10 thread si ottengono i miglior risultati prima di un graduale aumento nei tempi di esecuzione.

In entrambi test l'uso di un buffer privato per ogni singolo thread al posto di un buffer condiviso e sincronizzato tra tutti i thread non ha portato dei vantaggi a livello di tempistiche.

L'implementazione con buffer privato è stata fatta solo per avere una metrica di paragone tra l'utilizzo di variabili condivise e private.

5. Appendice

Codice della sezione parallela

```
else {
    int chunkDimension = totalFile / threadNumber;
    int remainingImage = totalFile % threadNumber;
    int ID = 0;
    ArrayList<ForkJoinTask<Boolean>> tasks = new ArrayList<>(threadNumber);

    for (int i = 0; i < threadNumber; i++) {
        int startIndex = i * chunkDimension;
        int endIndex = (i + 1) * chunkDimension;

        if (i != threadNumber - 1) {
            tasks.add(new Task(ID, images, sharedBuffer, startIndex, endIndex));
            ID++;
        } else {
               tasks.add(new Task(ID, images, sharedBuffer, startIndex, endIndex +
                                  →remainingImage));
               ID++;
               }

   }

   long startTime = System.currentTimeMillis();
   for (ForkJoinTask<Boolean> task : tasks) {
       task.fork();
   }

   for (ForkJoinTask<Boolean> task : tasks) {
        task.join();
   }

   long endTime = System.currentTimeMillis();

}
```