PC 2020/21 Elaborato Mid-term

Alessandro Mini
E-mail address
alessandro.mini1@stud.unifi.it

Abstract

In questo progetto si implementerà l'elaborato mid-term "Image Reader" in modo prima sequenziale e poi parallelo. L'implementazione consiste nel programma Java con relativa interfaccia Swing, si misureranno quindi speed-up ed efficienza.

1. Introduzione

Image Reader è un progetto in cui viene richiesto di implementare un algoritmo sequenziale (e parallelo) per importare e visualizzare immagini in modo non bloccante. L'utente interagisce con il programma visualizzando le immagini senza attendere.

Nel caso in cui l'immagine scelta non sia ancora stata caricata, restituirà un errore temporaneo.

Ho scelto come linguaggio di programmazione il Java (JDK 14), i test di import delle immagini verranno eseguiti su un *dataset* principale con 211 immagini per un peso di 490mb con risoluzione 3264 x 2448.

Per eventuali replicazioni il *dataset* è disponibile all'indirizzo `http://lear.inrialpes.fr/people/jegou/data.php`

Il flusso di interazione con il programma può essere riassunto nel seguente modo:

1. L'utente specifica una folder contente immagini jpg.
2. Il programma inizia a caricare le immagini.
3. Mentre il programma carica le immagini l'utente può visualizzarle tramite doppio click sulla tabella, se un immagine non è ancora stata caricata, restituirà un errore e chiederà di riprovare (Figura 1,2,3).

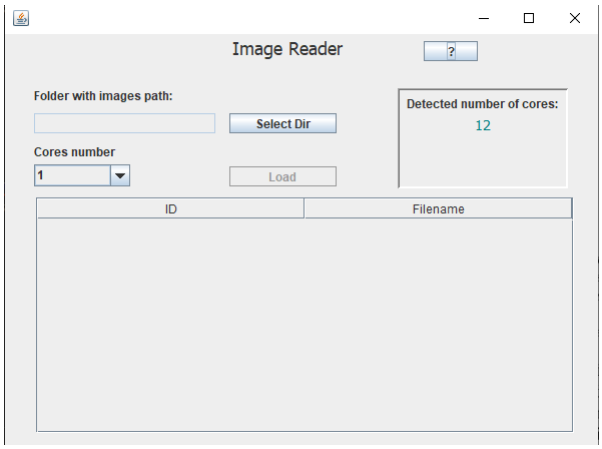

Figure 1. Interfaccia del programma, si può vedere la tabella in cui verranno importante le immagini

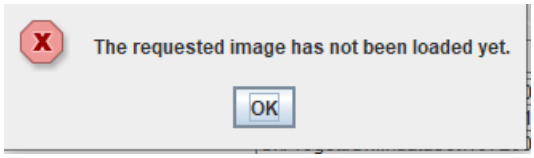

Figure 2. Tentativo di accesso ad immagine non ancora caricata dal worker in background

Alla fine del processo di caricamento, il programma visualizzerà un messaggio in cui mostra il tempo di caricamento totale. Come si può

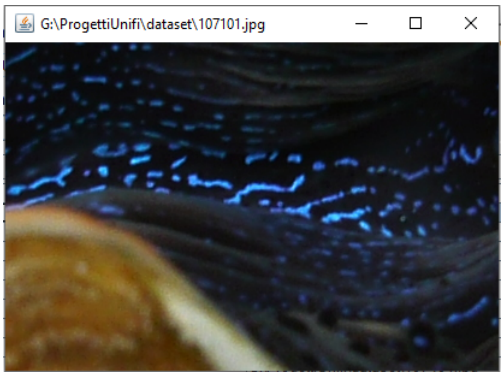

Figure 3. Immagine caricata con successo

notare nell'interfaccia è presente una *combobox* "Core" in cui si specifica il numero dei core da impiegare nell'esecuzione. Se core = 1 il programma eseguirà in modalità sequenziale.

1.1. Logica di funzionamento

Il nucleo fondamentale del programma è l'applicazione di un pattern di parallelismo di tipo **fork-join**.

Sulla base del numero di core scelto si crea un corrispondente numero di *tasks* che andranno a caricare *subsets* delle immagini in un buffer condiviso.

Ad ognuno di questi thread viene assegnato un task di tipo "loadTask" che viene inizializzato con la quadrupla:

(thID, img, bFolder, istart, iend)

dove:

- **thID:** ID del Thread (numero intero).
- **img:** Lista di tipo ArrayList di oggetti Image sincronizzata che funge da buffer comune ai vari task. Un oggetto di tipo Image è un oggetto che racchiude un .jpg importandolo in memoria e consentendone la visualizzazione
- **bFolder:** la cartella fisica nel file system dove risiedono le immagini.
- **istart:** Indice di partenza, il Task i-esimo assegnato ad un qualche processo importerà le immagini a partire dall'indice istart.
- **iend:** Indice di arrivo, il Task i-esimo assegnato ad un qualche processo importerà le immagini a partire dall'indice istart fino all'indice iend.

Il task *T* importerà nel buffer condiviso *img* le immagini presenti nella cartella di sistema *bFolder* a partire dall'indice *istart* fino a *iend*.

Attraverso l'applicazione di un opportuna politica di *divisione* del lavoro i thread ricopriranno uniformemente la cartella di partenza importando tutte le immagini.

2. Implementazione

Il programma è diviso in due packages, uno che contiene la classe "Main" con le primitive per l'interfaccia Swing ed uno che contiene più oggetti ed interfacce legate al "Worker" cioè alla porzione di codice che andrà effettivamente ad applicare il pattern *fork-join* per leggere le immagini.

Dal punto di vista di classi e packages il programma ha la seguente struttura:

```
GUI
├─Main (classe)
Worker (classe)
├─Drawable (interfaccia)
├─Image (classe)
├─loadTask (classe)
├─ParallelWorker (classe)
└─Utils (classe)
```

2.0.1 Descrizione delle classi

1. **Main.java :** Classe che contiene la GUI Swing costruita con Window Builder, questa classe inizializza il buffer condiviso:

   ```
   List<Image> immagini =
   Collections.synchronizedList(new
   ArrayList<Image>());
   ```

   Inoltre esegue (ed in seguito termina) su un thread separato un oggetto di tipo parallelWorker, che eseguirà il *fork-join* ed il lavoro di import delle immagini.

2. **Drawable.java :** Interfaccia che definisce i metodi di un oggetto "disegnabile".

3. **Image.java :** Oggetto immagine, le immagini vengono visualizzate attraverso swing.

4. **ParallelWorker.java :** Oggetto che divide i dati ed applica il pattern parallelo *fork-join*,crea i task per i vari thread li esegue da una pool (fork) ed aspetta la loro terminazione.

   Verrà quindi riportato il codice principale della classe ParallelWorker responsabile della divisione del lavoro e creazione dei task:

```
int totale = Utils.countFiles(dir);
int chunk = totale / numCore;
int resto = totale % numCore;

// tasks array.
ArrayList<ForkJoinTask<Boolean>> C = new
↪  ArrayList<ForkJoinTask<Boolean>>(numCore);

// Here we will divide the work.
int threadID = 0;
    for (int i = 0; i < numCore; i++) {
    // Per gli N-1 core: assegno un chunk.
    int start = i * chunk;
    int end = (i + 1) * chunk;
    if (i == numCore - 1) {
            C.add(new loadTask(threadID, immagini,
            ↪  images, start, end + resto));
            threadID++;
    }
            else {
            C.add(new loadTask(threadID, immagini,
            ↪  images, start,
            ↪  end));
            threadID++;

    }
}
```

   Questa procedura divide gli indici del buffer creato dal main e crea i vari task di tipo LoadTask, tanti quanti sono i core che si intende utilizzare.

5. **loadTask.java :** Questa classe rappresenta un singolo task, estende *RecursiveTask* di tipo Boolean, il metodo compute() restituirà true quando il lavoro di quel task sarà terminato.

```
@Override
protected Boolean compute() {
    for (int k = startIndex; k < endIndex; k++) {
            this.immagini.set(k,new
            ↪  Image(baseFolder[k].getAbsolutePath()));
    }
    System.out.println("Thread " + threadID + "
    ↪  Finito");
    return true;
}
```

   Il costrutture di *Image("path")* carica immediatamente in memoria l'immagine, ogni task carica un *subset* di immagini disgiunto che ricoprono l'intero *set* di immagini iniziali.

6. **Utils.java :** Classe che contiene alcuni metodi (tutti statici) per l'interazione con il sistema operativo.

Riassumendo schematicamente come si comporta il programma quando si avvia si possono identificare le seguenti fasi:

1. il Main crea un thread di tipo ParallelWorker specificando la cartella di destinazione con le immagini, il buffer condiviso e il numero di core da utilizzare (n).

2. ParallelWorker crea n thread, divide il lavoro con la procedura vista in sezione 1.2.1 e costruisce quindi n task che verranno assegnati ai thread.

3. ParallelWorker esegue la fork per ogni task, le immagini iniziano ad essere caricate in modo non bloccante, l'utente può aprirle dall'interfaccia grafica (doppio click sulla tabella), se l'immagine è stata caricata verrà aperta, altrimenti restituirà errore un errore temporaneo, chiedendo di riprovare in seguito.

4. ParallelWorker dopo un certo tempo finisce di caricare le immagini, eseguirà quindi la join degli n task.

5. il Main termina il thread ParallelWorker.

6. Il programma rimane attivo il thread della GUI in attesa di ulteriori istruzioni.

Se viene inserito $core = 1$ verrà creato un solo task che ricoprirà l'intera lista di immagini, caricandole in modo sequenziale, con la stessa strategia si implementa quindi sia una tecnica sequenziale che parallela, il numero di core "decide" quale strategia utilizzare.

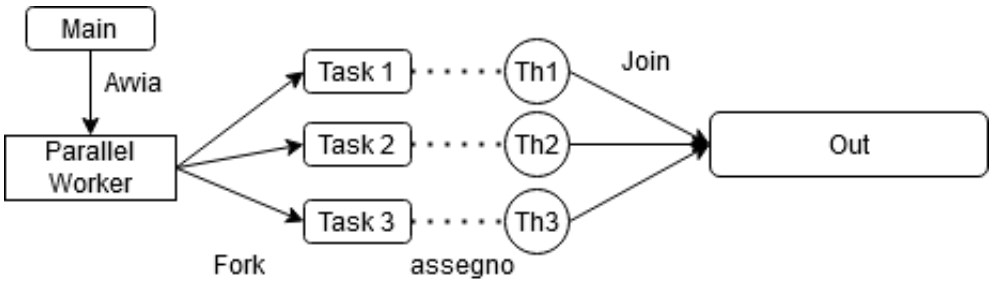

Figure 4. Rappresentazione della logica di funzionamento con 3 core e 3 tasks. Si nota il paradigma fork-join.

3. Calcoli e considerazioni

3.0.1 Assunzioni e proprietà

Il programma lavorerà sotto le seguenti assunzioni:

1. Le immagini JPG non sono corrotte e sono effettivamente accessibili dal programma.
2. La JVM allocherà al programma una quantità di *heap memory* tale da permettere il caricamento delle immagini
3. Il sistema operativo non interromperà il caricamento delle immagini (es. altro processo cerca di accederci/eliminarle).

Sotto queste assunzioni vale che:

1. Il programma prima o poi terminerà caricando le immagini.
2. L'unica sezione critica è l'accesso alla struttura dati condivisa che pertanto sarà una struttura sincronizzata.
3. Non ci sono condizioni di attesa circolare.

Il programma non presenta quindi eventuali condizioni di deadlock e di problemi dovuti all'accesso simultaneo ad una sezione critica.

3.0.2 Analisi dei costi

Il programma sequenziale caricherà le immagini in sequenza, assumendo che ogni immagine abbia un tempo di caricamento t_i otteniamo che il costo di caricamento totale può essere calcolato come :

$$(\sum_{i=1}^{N} t_i = t_1 + t_2 ...)$$

il quale ha complessità lineare, la versione parallela divide questo calcolo in C cores, ognuno dei quali eseguirà parte del lavoro, i core $c_0 ....c_{n-1}$ eseguiranno $\lfloor N/C \rfloor$ caricamenti mentre l'ultimo core c_n eseguirà la porzione rimanente.

Il tempo richiesto dal programma parallelo diverrà quindi il tempo richiesto dal thread che impiega più tempo, cioè l'ultimo.

L'ultimo thread impiegherà un tempo pari a :

$$\sum_{i=1}^{\lfloor \frac{N}{C} \rfloor + N \pmod C} t_i$$

Si procede quindi al calcolo dello *speedup* dapprima in termini analitici, in seguito sui test reali, vale che:

$$S_p = \frac{T_{Sequenziale}}{T_{Parallelo}} = \frac{\sum_{i=1}^{N} t_i}{\sum_{i=1}^{\lfloor \frac{N}{C} \rfloor + N \pmod C} t_i}$$

Posso approssimare il calcolo dello speedup maggiorando i tempi di caricamento delle singole immagini, si suppone che ogni immagine nel caso pessimo venga caricata con tempo t_m

$$t_m = \max_{t_i \in T}$$

Allora per l'intero caricamento del dataset di immagini ottengo i seguenti tempi approssimati:

- Costo sequenziale totale: $t_m \cdot N$
- Costo parallelo totale: $t_m \cdot (\lfloor \frac{N}{C} \rfloor + N \pmod C)$

Da cui si deriva uno speedup (Approssimato) per N grande:

$$S_{Pa} = \frac{T_{Sequenziale}}{T_{Parallelo}} = \frac{t_m \cdot N}{t_m \cdot \lfloor \frac{N}{C} \rfloor + N \pmod C} \approx C$$

All'atto pratico si è eseguito un test su un PC con il seguente hardware:

CPU	Ryzen 5 3600 (6 core)
RAM	16 GB DDR4
HDD	WD Blue 1TB
M2	Sabrent Q4

Per il programma sequenziale si ottiene un tempo di $18682ms$, per il programma parallelo si ottengono i seguenti tempi (in millisecondi) al variare dei cores:

Cores utilizzati:	Tempo HDD	Speedup	Efficienza
2	10737	1,74	0,87
4	8037	2,32	0,58
6	7331	2,55	0,42

Si può notare che per 6 core si ottiene il massimo speedup.

$$S_P = \frac{18682ms}{7331ms} \approx 2.55$$

Con un efficienza di:

$$\epsilon = \frac{1}{C} \cdot \frac{T_{sequenziale}}{T_{parallelo}} \approx 0.42$$

Per le caratteristiche intrinseche del programma i tempi di caricamento delle singole immagini (cioè $t_1 ... t_n$) possono variare molto a seconda della scelta del dataset in quanto diversi dataset impiegano un utilizzo del disco e della memoria diverso. Con 900 immagini con risoluzione 400 x 600 si ha un tempo sequenziale di $5686ms$ e parallelo di:

Cores:	Tempo HDD	Speedup	Efficienza
2	2982	1,91	0,95
4	1818	3,13	0,78
6	1672	3,40	0,57

Lo stesso test su M2:

Cores:	Tempo M2	Speedup	Efficienza
2	2979	1,88	0,94
4	1712	3,27	0,82
6	1685	3,32	0,55

4. Conclusioni

Si è costruita una versione parallela e sequenziale del programma, si ottiene sempre uno *speedup* sebbene questo sia fortemente influenzato dalla macchina e dal dataset *dataset*. Nel calcolo dello speedup reale va tenuto conto di un forte intervento del sistema operativo e dell' hardware per gestire i caricamenti delle immagini dalla memoria secondaria $\rightarrow$ memoria primaria.

Si ha infatti un forte utilizzo di CPU, memoria e Disco (in termine di accessi).