Non-Blocking JPEGs Reader in Java

Mattia Bacci
Matricola 7060637
mattia.bacci@stud.unifi.it

Abstract

Questo elaborato Mid-Term verte sull'implementazione di una versione non bloccante di un JPEGs Reader sia in versione sequenziale che in versione parallela. L'implementazione è stata fatta in Java e ne sono state analizzate le performance in un intel-i5 con 2-core e un intel-i5 con 4-core. Vengono successivamente riportati sia lo speedUp che l'efficienza in base al numero di processori disponibili.

1. Introduzione

1.1. Obiettivo

L'obiettivo di questo elaborato è la creazione di un applicativo Java che permetta la selezione di una cartella, all'interno della quale sono presenti immagini con formato JPEG, per caricarne il contenuto all'interno della memoria e rendere possibile la visualizzazione sia durante che al termine del caricamento. E' stata sviluppata un interfaccia grafica tramite Swing per realizzare l'intento, tale interfaccia si presenta, oltre per la visualizzazione delle immagini, come uno strumento di analisi, potendo variare i parametri del programma prima della sua esecuzione. Tale interfaccia presenta alla fine dell'esecuzione il tempo di calcolo necessario per il caricamento delle immagini in memoria.

Benché sia richiesto l'implementazione sia di codice sequenziale che parallelo, l'utilizzo dell'interfaccia grafica richiede che anche il software sequenziale sia in effetti MultiThread, per versione sequenziale si considera l'avvio di una solo Thread che esegue il caricamento delle immagini.

1.2. I Parametri

Prima dell'inizio della computazione posso essere scelti due parametri:

1. **Numero dei Thread:** Permette di scegliere su quanti Thread effettuare la nostra esecuzione
2. **Modo di esecuzione:** Permette di scegliere tra due metodi di esecuzione:
   - **Statico - Senza Coda:** Divide il numero di immagini in parti uguali sul numero di Thread.
   - **Dinamico - Con Coda:** Ai Thread viene fornita una coda condivisa di tipo *ConcurrentLinkedQueue* da cui prelevare le immagini, è Thread-Safe ed implementa algoritmi *wait-free*.
   - **Dinamico - Con Coda Sincronizzata:** Ai Thread viene fornita una coda condivisa di tipo *LinkedList* da cui prelevare le immagini, non è Thread-Safe, si devono sincronizzare gli accessi.

1.3. Nota

Come già evidenziato il software si presenta come strumento di analisi e didattico, non mira all'ottimizzazione della computazione in base hai dati forniti dal computer nel quale sta eseguendo.

2. Implementazione

Nell'implementazione non si è fatto distinzione tra la versione sequenziale e parallela, in quella sequenziale viene eseguito solo una volta il Thread identificato dalla classe *Reader*.

2.1. Classi

Sono state create 4 classi principali:

1. **UserInterface:** Viene inizializzata l'interfaccia grafica che provvede alla creazione di tutti i listener. Il Listener del pulsante di Start esegue la creazione e l'esecuzione dei Thread in base ai parametri impostati.
2. **ImageContainer:** gli oggetti che compongono la nostra lista di immagini, contengono il nome, il riferimento e il percorso dell'immagine.
3. **Reader:** gli oggetti Reader costituiscono i nostri Thread, la classe estende SwingWorker in modo da poter effettuare la computazione necessaria e effettuare l'aggiornamento sull'interfaccia grafica

4. **StateListener:** Quando l'esecuzione dei Reader termina lo stato degli SwingWorker(Reader) passa da STARTED a DONE, tale cambiamento lancia un evento catturato dall'istanza di questa classe, tale istanza è condivisa con tutti i Reader in esecuzione, un contatore tiene traccia dei Thread terminati, l'evento sollevato dall'ultimo Thread permette di interrompere un timer che segna il tempo di esecuzione dall'avvio al termine di tutti i Thread. Tale valore viene visualizzato sull'interfaccia grafica.

2.2. La sinronizzazione e l'EDT

Se prendiamo in analisi la versione Statica - Senza Coda dal codice si può notare come non sia presente alcun tipo di sincronizzazione; neanche nella lista di output, la quale è soggetta a scrittura da parte di tutti i Thread e lettura dall'interfaccia grafica.
I Reader attraverso il metodo *process(...)*(Overrided da SwingWorker) qui riportato effettuano l'aggiornamento della lista, i quali elementi verranno mostrati in tempo reale sull'interfaccia grafica.

```
@Override
   protected void process(List<ImageContainer> chunks) {
       for (ImageContainer imageContainer : chunks) {
           listModel.addElement(imageContainer);
       }
   }
```

Si può notare come l'aggiunta dell'elemento alla lista tramite il metodo *addElement(...)* non sia sincronizzato, ne sia stato acquisito nessun tipo di lock.
Anche il Listener di tale lista, il quale ha il compito di impostare correttamente il label per far visualizzare l'immagine, non è sincronizzato.

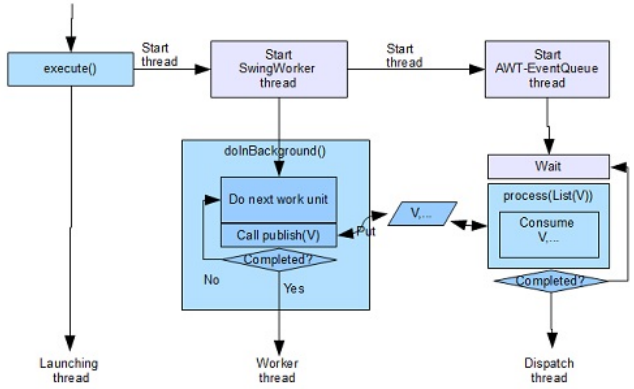
```
list.addListSelectionListener(e -> {
      if (!list.getValueIsAdjusting()) {
          Image image = null;
          try {
              image = list.getSelectedValue().getImage();
          } catch (NullPointerException ex) {
                    ex.printStackTrace();
          }

          if (image != null) {
              imageLabel.setIcon(new ImageIcon(image));
          } else {
              imageLabel.setIcon(null);
          }
      }
});
```

Allo stesso modo il codice dello StateListener non presenta atomicità nella variabile *i* che tiene traccia del numero dei Thread Terminati:

```
@Override
   public void propertyChange(PropertyChangeEvent evt) {
       String value = evt.getNewValue().toString();
       if (value.equals("DONE")) {
           System.out.println(i);
           if (++i == THREADS_NUM) {
               long end = System.currentTimeMillis();
               label.setText((end - start) + "ms");
           }
       }
   }
```

La motivazione di questa scelta risiede nella gestione degli eventi. Sia il *listener* che il metodo *process* non vengono eseguito all'interno del Thread chiamante, bensì vengono eseguito all'interno di un unico Thread, **l'EDT- Event Dispatch Thread**. Questo ci assicura che la nostra lista rimanga consistente in quanto ogni aggiornamento o lettura è effettuato da un unico Thread. Inoltre visto che le immagini sono rese visibili alla UserInterface solamente quando sono state caricate non sono presenti alcun tipo di Race Condition dovuto all'apertura di un immagini non ancora presenti.



2.3. Con Coda e Senza Coda

Dalla UserInterface c'è la possibilità di selezionare la modalità di esecuzione del programma, in particolare l'input dei Thread.

Nell'utilizzo senza coda il numero di immagini viene suddiviso in parti uguali a tutti i Thread, se se divisione della immagini per il numero di Thread non genera un numero intero, le immagini di resto vengono inserite nell'ultimo Thread. Lo sbilanciamento è minimo in quanto se indico con N il numero dei Thread ci saranno massimo N-1 immagini in più presenti nell'ultimo Thread.
Si riporta solo la parte di codice atta a effettuare la divisione:

```
int elementsNumber = files.length / THREADS_NUM;
for (int i = 0; i < THREADS_NUM - 1; i++) {
   readers[i] = new Reader(files, elementsNumber * i,
     elementsNumber * (i + 1), listModel);
   }
   readers[THREADS_NUM - 1] = new Reader(files,
     elementsNumber * (THREADS_NUM - 1), files.length,
     listModel);
```

Nell'utilizzo con coda di tipo *ConcurrentLinkedQueue* ai Reader viene passata una coda condivisa di immagini. I Thread prelevano da questa coda contemporaneamente, la sincronizzazione è garantita dalla natura Thread-Safe della classe.

Nell'utilizzo con coda di tipo *LinkedQueue* il funzionamento è lo stesso della precedente, data la natura Non

N. Immagini	Metodo Esecuzione	Tempo Sequenziale	Tempo MultiThread	Efficienza
900	Senza Coda	16100 ms	4500 ms	90%
900	Con coda	16100 ms	4700 ms	85%
900	Con coda sincronizzata	16200 ms	4900 ms	83%
1400	Senza Coda	26100 ms	7800 ms	84%
1400	Con coda	25900 ms	8200 ms	79%
1400	Con coda sincronizzata	26700 ms	8000 ms	83%

Figure 1. Intel Core i5 (I5-4570S) 4-Core

N. Immagini	Metodo Esecuzione	Tempo Sequenziale	Tempo MultiThread	Efficienza
900	Senza Coda	17900 ms	8300	100%
900	Con coda	18700 ms	8600	100%
900	Con coda sincronizzata	18200 ms	9100	100%
1400	Senza Coda	28500 ms	14600	100%
1400	Con coda	28900 ms	14900	100%
1400	Con coda sincronizzata	29100 ms	15100	100%

Figure 2. Intel Core i5 (I5-7267U) 2-Core

Thread-Safe della classe però gli acessi devono essere sincronizzati tramite l'acquisizione di lock sull'oggetto.

3. Analisi

Le analisi sono state condotte su due tipi di processori differenti e con un numero differente di core: l'Intel Core i5 (I5-4570S) con 4-Core e l'Intel Core i5 (I5-7267U) con 2-Core, i quali risultat sono riportati rispettivamente nella Fig.1 e Fig.2. I test sono stati effettuati con due dataset di immagini con risoluzione minore di 800x800, la prima composta da 900 immagini, la seconda da 1400.

Dall'osservazione dei dati possiamo condurre le seguenti conclusioni:

- Nel Processore da 4 Core si registra un speed-up comune compreso tra 3-3.5, l'efficenza valutata come numero di core fratto lo speed-up è, in qualsiasi caso, compresa tra 80-90%
- Nel Processore da 2 Core si registra un speed-up sempre uguale a 2, l'efficienza è sempre del 100%
- In entrambi i processori l'esecuzione di modalità diverse non ha effetto rilevante ne sullo speed-up e a conseguenza sull'efficienza. Questo risultato può essere ricondotto alle seguenti motivazioni:

    1. L'assegnazione statica assegna lo stesso carico di lavoro ai Thread che se mandati in esecuzione allo stesso tempo e non interrotti eseguono in modo ottimale.
    2. La *ConcurrentLinkedQueue* implementando algoritmi di tipo wait-free rende l'accesso da parte dei Thread immediato, permette inoltre di far fare lavoro maggior alcuni Thread se necessario.
    3. La *LinkedList* dovrebbe risultare la meno performante di tutte e tre le modalità, nella pratica essendo però sincronizzato solo l'accesso alla lista, il tempo di tale accesso è talmente irrisorio rispetto al tempo di caricamento delle immagini in memoria, che i Thread si trovano poche volte a dover aspettare il rilascio del lock e se devono attendere lo devono comunque fare per un tempo trascurabile.