Relazione Progetto PCD - Parte 2

Giacomo Manzoli 1049820

7 gennaio 2015

Indice

1 Introduzione

Questa parte del progetto consiste nel trasformare l'algoritmo ideato nella prima parte da algoritmo sequenziale a parallelo.
Questa trasformazione deve garantire che:

- Ci siano più thread che risolvano il puzzle e con un carico di lavoro uniforme;
- Non ci siano attese attive, interferenze tra thread e deadlock;

2 Algoritmo parallelo

Dato che la prima versione dell'algoritmo era stata ideata pensando che poi sarebbe stata eseguita su più thread non è stato necessario attuare modifiche riguardo la logica di risoluzione. Per quanto riguarda l'implementazione, invece, sono state necessarie varie modifiche.

2.1 Avvio della soluzione

Nella versione precedente la soluzione del puzzle avveniva con una chiamata del metodo `solve()` da parte del costruttore della classe, in questa versione invece il costruttore della classe si occupa solamente di creare e avviare i thread necessari alla soluzione del puzzle.

```
public Puzzle(Path inputPath){
    PuzzleElementList listaPezzi = new PuzzleElementList();
    LoaderThread loader = new LoaderThread(inputPath,listaPezzi, "Millenium Falcon");
    loader.start();
    SolverThread solver1 = new SolverThread(listaPezzi,"Rosso 1");
    SolverThread solver2 = new SolverThread(listaPezzi,"Rosso 2");
    solver1.start();
    solver2.start();
    try {
        solver1.join();
        synchronized (listaPezzi) {
            listaPezzi.notifyAll();
        }
        solution = listaPezzi.remove(0);
    }catch (InterruptedException e){
        System.err.println(e.getMessage());
        solution=null;
    }
}
```

Il thread che esegue il costruttore aspetta che termini il thread `solver1`, la scelta del thread aspettare la terminazione è indifferente perché i thread di tipo `SolverThread` terminano quando il puzzle è risolto, inoltre, non è possibile sapere quale thread finisce per primo a causa del comportamento non deterministico dello scheduler della JVM.
Se il thread che esegue il costruttore viene interrotto durante l'attesa della soluzione viene stampato il messaggio dell'eccezione e la soluzione viene posta a `null`.

2.2 Thread

Sono state create due classi che derivano la classe `Thread` di Java, la prima che si occupa di caricare i dati dal file e la seconda che implementa l'algoritmo risolutivo.
Tutte e due le classi thread sono contenute nel package `Puzzle` ed hanno visibilità `package`.

2.2.1 LoaderThread

Questa classe si occupa di caricare i dati dal file di input e di caricarli nella lista condivisa tra i vari thread.
Il funzionamento della lista condivisa e del perché questa sia thread safe viene discusso nella sezione 3.1.
Per creare in modo corretto i pezzi del puzzle, sono stati spostati in questa classe i metodi riguardo la verifica degli input che nella prima versione erano all'interno della classe `Puzzle`.
Di seguito viene riportato il codice della `run()` del thread.

```
public void run() {
    Charset charset = StandardCharsets.UTF_8;
    try (BufferedReader reader = Files.newBufferedReader(inputPath, charset)) {
        String line;
        while ((line = reader.readLine()) != null) {
            String[] parts = line.split("\t");
            if (parts.length == 6) {
                PuzzleElement p = LoaderThread.createElement(parts);
                pezzi.add(p);
            } else {
                Exception e = new Exception("Problema con il caricamento del file: " +
                    inputPath.toString());
                throw e;
            }
        }
    } catch (Exception e) {
        System.err.println(e.toString());
    }
}
```

L'unico cambiamento dalla prima versione riguarda la lista in cui vengono memorizzati i pezzi. Nel caso il puzzle sia diviso su più file e se un pezzo compare solamente in un file, ovvero la stringa che rappresenta un pezzo compare solo su un file, è possibile parallelizzare anche il caricamento.

2.2.2 SolverThread

Questa classe si occupa di implementare l'algoritmo risolutivo che precedentemente era nel metodo `solve()` della classe `Puzzle`.

```
public void run(){
    while (!pezzi.hasBeenCompleted()){
        PuzzleElement anItem=null, attachableItem;
        synchronized (pezzi) {
            if (pezzi.size() < 2) {
                if (pezzi.hasBeenCompleted()) {
                    return;
                }
                try {
                    pezzi.wait();
                    continue;
                } catch (InterruptedException e) {
                    System.err.println(e.getMessage());
                }
            } else {
                anItem = pezzi.remove(0);
            }
        }
```

```
            attachableItem = pezzi.removeAttachableTo(anItem);
            if (attachableItem != null){
                PuzzleElement newItem = anItem.attach(attachableItem);
                 pezzi.add(newItem);
            } else {
                pezzi.add(anItem);
            }
        }
}
```

Un thread di questo tipo continua la sua esecuzione fino a quando il puzzle non è stato completato, situazione che si può verificare sia a causa del thread corrente sia a causa di un thread concorrente.
s La zona critica comprende il test sul numero di pezzi e l'estrazione di un pezzo dalla lista, è stato necessario mettere tutto questo dentro un blocco `synchronized` anche se la lista offre solo metodo sincronizzati perché se fossero fuori dal blocco il thread corrente potrebbe essere sospeso e un altro thread concorrente potrebbe prendere l'unico pezzo presente nella lista creando una situazione inconsistente per il thread corrente.
Una volta preso il pezzo dalla lista, si cerca nella lista un pezzo che possa essere combinato e in caso positivo vengono combinati tra loro e il risultato viene rimesso in lista, altrimenti se `attachableItem` è `null` viene reinserito in lista il pezzo `anItem` senza che subisca modifiche, in modo da evitare la perdita di pezzi.
Queste operazioni non devono essere protette in un blocco `synchronized` perché i dati su cui lavora il thread non sono condivisi.

2.3 Costrutti utilizzati

- `synchronized`**:** viene usato nella `PuzzleElementList` per offrire un accesso ai dati sincronizzato lato server, in questo modo vengono evitate condizioni di data race potenzialmente pericolose. Questo costrutto viene usato anche nel *check-then-act* presente nel metodo `run()` della classe `SolverThread` e quanto è necessario utilizzare `wait()` e `notify()`;

- **wait/notify:** questi metodi vengono usati per evitare situazioni di attesa attiva quando non ci sono sufficienti pezzi disponibili per proseguire nella risoluzione del puzzle;

- **join:** viene usato per sincronizzare l'esecuzione del thread che richiede la soluzione del puzzle con l'esecuzione dei *solver*. All'inizio si era pensato di utilizzare `wait/notify` per sospendere il thread principale finché non il puzzle non fosse risolto, si è scelto di usare il `join` perché poteva verificarsi che il main thread andasse in `wait` dopo che la lista avesse eseguito il `notify`, creando una situazione di stallo.

2.4 Correttezza del parallelismo

Per assicurare la consistenza dei dati si è scelto di wrappare un `ArrayList` dentro una classe che implementi la sincronizzazione lato server.
Si è scelta questa strada perché questa lista viene usata da classi diverse e in questo modo si ha la sicurezza che l'accesso ai dati sia sempre eseguito in modo thread safe.
Non si possono verificare situazioni di deadlock in quanto è presente un solo oggetto su cui viene preso il lock e nel caso di chiamate a metodi della lista condivisa dentro ad un blocco `synchronized` si tratta sempre di un lock rientrante che viene rilasciato nel caso il thread che lo detiene si metta in `wait`.

2.4.1 Comportamento con input errato

Nel caso venga avviato il programma con dei dati errati (pezzi mancanti oppure input non valido) si crea una situazione di stallo con tutti i thread che si sospendo.
Non è stata gestita questa situazione dato le specifiche prevedono che il puzzle sia completo.

3 Cambiamenti rispetto la prima parte

L'algoritmo risolutivo è rimasto invariato, tuttavia sono state necessarie alcune modifiche extra all'implementazione in particolare legate alla lista dei pezzi e alle due classi `PuzzleRow` e `PuzzleBlock`.

3.1 PuzzleElementList

La classe `ArrayList` offerta dalle librerie Java non espone metodi `synchronized`, è stato quindi necessario definire una classe `PuzzleElementList` che si occupa di offrire metodi per l'accesso sincronizzato alla lista e metodi utili a capire se il puzzle che era memorizzato all'interno della lista è stato completato o meno.

3.2 PuzzleRow e PuzzleBlock

E' stato corretto un bug della classe `PuzzleBlock`, in particolare quando veniva chiamato il metodo `attach(anItem)` e come parametro si forniva una `PuzzleRow` completa si creava una situazione di inconsistenza.
Per correggere questo bug e per semplificare l'implementazione di alcuni metodi della classe `PuzzleElementList` è stata cambiata la creazione di una riga completa, in particolare, quando l'unione di due pezzi genera una riga completa, viene costruito subito un `PuzzleBlock` che la contiene. Nonostante i numerosi test effettuati sulla prima versione del programma questo bug è emerso solo ora perché le condizioni che lo provocano non si potevano mai verificare per come era implementato l'algoritmo sequenziale.

3.3 Classe Puzzle

La funzionalità principale di questa classe è quella di avviare i thread necessari alla risoluzione del puzzle.
Le modifiche attuate a questa classe sono state affrontate nella sezione 2.1

3.4 Gestione input errati

In questa versione, come specificato nella sezione 2.4.1, non vengono gestiti gli input errati e, nel caso di un input errato, il sistema va in stallo con tutti i `SolverThread` che si mettono in wait sulla lista condivisa.
Sono stati inoltre ridotti i controlli riguardo gli input validi, modificando il metodo `verify` che ora controlla solamente che i pezzi siano compsti da un solo carattere.