PuzzleSolver - Relazione Programmazione Concorrente e Distribuita, Parte 2

Tobia Tesan - #1051819

18 gennaio 2015

Sommario

La presente relazione dettaglia e motiva le scelte progettuali dell'allegato programma `PuzzleSolver`, che realizza le specifiche della parte 2 del progetto di Programmazione Concorrente e Distribuita per l'A.A. 2014/2015.

Indice

1 Vista d'insieme

Per realizzare le specifiche della parte 2 del progetto sono state necessarie modifiche minimali alla gerarchia delle classi. È stato infatti sufficiente introdurre una classe ConcurrentHashmapPuzzle che, come BFSHashmapPuzzle estende HashmapPuzzle.

L'organizzazione risultante è di conseguenza la seguente:

```
|-- core
|   |-- BasicPuzzlePiece.java
|   |-- BasicPuzzlePieceTest.java
|   |-- ConcurrentHashmapPuzzle.java
|   |-- ConcurrentHashmapPuzzleTest.java
|   |-- HashmapPuzzle.java
|   |-- IPuzzle.java
|   |-- IPuzzlePiece.java
|   |-- MissingPiecesException.java
```

```
|   `-- PuzzleNotSolvedException.java
|-- io
|   |-- IPuzzlePrinter.java
|   |-- MalformedFileException.java
|   |-- PlaintextPuzzlePrinter.java
|   `-- PuzzleFileParser.java
`-- PuzzleSolver.java
```

ConcurrentHashmapPuzzle include una classe annidata RowLinker che implementa Callable. ConcurrentHashmapPuzzle differisce nell'interfaccia esposta al pubblico nel fatto che prende un argomento di tipo ExecutorService, che viene usato per eseguire RowLinker. È stato anche necessario fare delle minime modifiche al metodo `main`, in modo che questo istanzi il medesimo ExecutorService.

2 ConcurrentHashmapPuzzle

2.1 Algoritmo

L'idea di base dell'algoritmo scelto consistente nell'iterare parallelamente su ciascuna riga del puzzle e aggiornare i riferimenti verso (ma non *da*) i vicini N, S, W, E.

2.2 Implementazione

L'iterazione per riga è implementata nel Callable RowLinker, di cui è riportata in Fig. 1 una versione semplificata.

L'esecuzione parallela ruota intorno a un ExecutorCompletionService che istanzia un RowLinker per ogni riga e attende, usando il metodo bloccante Future.get(), che siano completati tutti - in modo sostanzialmente analogo a istanziare manualmente un numero di thread e restare in attesa del loro completamento con una `join()`. Si è fatto riferimento a [GP06] (§6.3.6) per l'uso di ExecutorCompletionService.

Il metodo Future.`get` può sollevare InterruptedException, ExecutionException. ExecutionException è un wrapper intorno a eventuali eccezioni sollevate dal `call()`, in questo caso Linker.`call()`: l'unica eccezione *checked* da esso sollevata è MissingPiecesException, che viene estratta e sollevata, insieme alle eccezioni *unchecked*.

Viene inoltre gestita InterruptedException - corrispondente a una interruzione brusca della `call()` - interrompendo il thread principale.

2.3 Numero di thread

Il numero di istanze di RowLinker è, dopo l'ultimo `submit()`, $x : 0 \leq x \leq n$ se n è il numero di righe del puzzle. Il numero di thread *runnable* in un dato istante t è limitato superiormente dalla dimensione del ExecutorService ([GP06]).

```
1  public Void call() throws MissingPiecesException {
2
3          t = angolo NW
4
5          while (!ultima colonna est) {
6                  if (!prima riga nord) {
7                          // Preleva vicino nord da HashMap,
8                          // MissingPiecesException se non trovato;
9
10                         // Aggiungi riferimento a vicino nord
11
12                 }
13                 if (!ultima riga sud) {
14                         // Preleva vicino sud da HashMap,
15                         // MissingPiecesException se non trovato;
16
17                         // Aggiungi riferimento a vicino sud
18                 }
19                 if (!prima colonna ovest) {
20                         // ...
21                 }
22
23                 t = vicino est
24         }
25
26         if (!prima riga nord) {
27                 // ...
28         }
29         if (!ultima riga sud) {
30                 // ...
31         }
32         if (!prima colonna ovest) {
33                 // ...
34         }
35
36         return null;
37 }
```

Figura 1: Scheletro di RowLinker.call()

2.4 Uniformità

Poichè i puzzle sono rettangolari, ciascuna istanza di RowLinker itererà sullo stesso numero di pezzi.

È dunque realizzato il requisito di uniformità per puzzle di dimensioni $n \times m$ con $n, m \gg 1$.

Sono tuttavia casi degeneri quelli in cui il puzzle è formato da una sola riga o una sola colonna. Nel caso di una sola riga, l'algoritmo non presenterebbe, di fatto, parallelismo. Nel caso di una sola colonna di dimensione n, si avrebbero n thread che opererebbero su un solo pezzo, causando un grande overhead.

```
1  private void linkColSE () throws MissingPiecesException {
2          assert(NWCorner != null);
3
4          IPuzzlePiece it = NWCorner;
5          int rows = 0;
6          // JCIP
7          CompletionService<Void> completionService =
8                new ExecutorCompletionService<Void>(executor);
9          while (!it.isSRow()) {
10                 completionService.submit(new RowLinker(it, pieceHashMap));
11                 it = pieceHashMap.get(it.getSouthId());
12                 if (it == null) {
13                         throw new MissingPiecesException();
14                 }
15                 rows++;
16         }
17         completionService.submit(new RowLinker(it, pieceHashMap));
18         rows++;
19
20         try {
21                 for (int i = 0; i < rows; i++) {
22                         Future<Void> f = completionService.take();
23                         f.get();
24                 }
25                 // This completes without blocking or raising
26                 // execptions iff exactly n Callables have completed
27         } catch (InterruptedException ie) {
28                 Thread.currentThread().interrupt();
29         } catch (ExecutionException e) {
30                 Throwable t = e.getCause();
31                 // Unchecked
32                 if (t instanceof RuntimeException)
33                         throw (RuntimeException) t;
34                 else if (t instanceof Error)
35                         throw (Error) t;
36                 // Checked
37                 else if (t instanceof MissingPiecesException)
38                         throw (MissingPiecesException) t;
39         }
40 }
```

Figura 2: ConcurrentHashmapPuzzle.linkColSE()

2.5 Thread-safety, interferenze, deadlock

Non è possibile avere interferenze o deadlock all'interno dell'algoritmo di risoluzione.

Infatti, il metodo Hashmap.`get()` è thread-safe, a patto di non aggiungere elementi alla tavola hash durante la soluzione del puzzle, e ciascun PuzzlePiece viene modificato da uno e un solo thread (quello corrispondente alla riga a cui il pezzo appartiene).

Similmente, non è possibile ottenere deadlock perchè non si presentano le

condizioni di Coffman (è in particolare immediato che non ci può essere attesa circolare) [Tan01].

Thread-safety La classe risultante non è però (a dispetto del nome) thread-safe, e il suo uso in un programma terzo dovrebbe essere condizionato alla gestione manuale della concorrenza.

Si manifestano in particolare delle race condition sull'attributo `solved`: sarebbe possibile, ad esempio, avviare `solve` e aggiungere, frattanto, un pezzo al puzzle con il metodo `add`: se `solve` alzasse il flag `solved` *dopo* la terminazione di `add`, il puzzle sarebbe marcato come *risolto* trovandosi in realtà in uno stato imprevedibile.

3 Compilazione e utilizzo

```
$ cd parte2/
$ make
$ ./puzzlesolver.sh input.txt output.txt
```

oppure

```
$ cd parte2/
$ make
$ java -jar PuzzleSolver.jar input.txt output.txt
```

Opzionalmente:

```
$ make javadoc
```

Riferimenti bibliografici

[GP06] B. Goetz and T. Peierls. *Java concurrency in practice.* Addison-Wesley, 2006.

[Tan01] A.S. Tanenbaum. *Modern Operating Systems.* Modern Operating Systems. Prentice Hall, 2001.