Programmazione Concorrente e Distribuita 2014-2015. Seconda parte.

Andrea Baldan

August 28, 2015

Contents

1 Modifiche rispetto alla prima parte **1**

2 Algoritmo di ordinamento parallelizzato **1**

2.1 I thread 2

1 Modifiche rispetto alla prima parte

Il design pattern scelto ha permesso di mantenere inalterata l'organizzazione delle classi, permane quindi l'albero della directory formato da **models**, **views**, **controllers** e il package **puzzlesolver** con l'unica differenza che la classe SortAlg ora è stata estesa da 2 sottoclassi, SortAlgFromTop e SortAlgFromBottom, questo per permettere la parallelizzazione dell'algoritmo di ordinamento.
Le rimanente struttura del programma è rimasta pressochè invariata.

2 Algoritmo di ordinamento parallelizzato

Per consentire un approccio concorrente all'algoritmo ideato nella prima parte del progetto, la classe base astratta SortAlg.java questa volta è stata estesa in due sottoclassi che implementano l'interfaccia **Runnable**, in modo da potere essere incapsulate in un oggetto di tipo Thread ed aver così la possibilità di lanciare il metodo sort() in maniera concorrente.
SortAlgFromTop.java, la prima delle due sottoclassi, si occupa di ordinare la prima metà del puzzle utilizzando essenzialmente lo stesso sistema impiegato nella prima parte del progetto, ovvero localizzazione del primo pezzo (nord e ovest "VUOTO"), ordinamento della riga mediante metodo nextInRow(), localizzazione del pezzo a sud del primo precedentemente localizzato mediante metodo nextInCol() e cosi via fino al raggiungimento della soglia prevista. SortAlgFromBottom.java si occupa in maniera analoga di ordinare la metà inferiore del puzzle, ma questa volta l'algoritmo, per quanto si tratti delle stesse operazioni di SortAlgSeq.java, è stato implementato per ricostruire il puzzle al contrario. Il primo pezzo è quindi in questo caso l'ultimo del puzzle (sud ed est "VUOTO") e si procede in senso contrario a SortAlgSeq, cercando i pezzi successivi al corrente a sinistra invece che a destra, fino al raggiungimento del punto d'incontro con il Thread della parte superiore.

2.1 I thread

Il programma avvia in tutti i casi 3 thread distinti, il main thread e i due thread di ordinamento. Per mantenere un approccio il più thread-safe possibile, sull'oggetto condiviso vengono effettuate solamente operazioni di lettura, mentre per effettuare modifiche vengono utilizzate strutture ausiliarie Vector, che ad algoritmo completato vengono utilizzate nella sostituzione dell'oggetto puzzle inizialmente scomposto.
Ciò ha permesso di evitare l'utilizzo di costrutti `synchronized` che di fatto, stando alla struttura dell'algoritmo di ordinamento, avrebbero vanificato buona parte dei benefici della programmazione concorrente. Entrambi i thread rimangono attivi concorrentemente qualsiasi sia l'input (corretto) fornito, l'unico costrutto utilizzato è infatti il metodo `join()` della classe Thread, che permette di attendere il termine delle operazioni del thread in questione prima di terminarlo, richiamato all'interno di un blocco try/catch in modo da gestire un eventuale `InterruptedException`.
La scelta di utilizzare strutture di supporto e non una copia dell'oggetto condiviso, oltre a evitare il ricorso a blocchi `synchronized` e costrutti `wait/notify`, garantisce anche la totale assenza di situazioni di `deadlock/starvation`.