

# Programmazione Concorrente e Distribuita 2014-2015. Seconda parte.

Andrea Baldan

September 4, 2015

## Contents

---

<b>1</b>	<b>Modifiche rispetto alla prima parte</b>	<b>1</b>
<b>2</b>	<b>Algoritmo di ordinamento parallelizzato</b>	<b>1</b>
2.1	I thread . . . . .	2
2.2	SharedSortStat . . . . .	2
2.3	Ricapitolando: . . . . .	2
2.4	Note . . . . .	3
<b>3</b>	<b>Compilazione ed esecuzione</b>	<b>3</b>

## 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.

E' stata inoltre aggiunta una classe *SharedSortStat*, che rappresenta un oggetto condiviso allo scopo di verificare lo "stato" di ordinamento del puzzle da parte dei thread, e permettere una comunicazione tra i due. 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* questa volta è stata estesa in due sottoclassi che implementano l'interfaccia **Runnable**, in modo da poter essere incapsulate in un oggetto di tipo **Thread** ed aver così la possibilità di lanciare il metodo *sort()* in maniera concorrente; è stato inoltre aggiunto un campo dati di tipo *SharedSortStat*, cioè l'oggetto che verrà condiviso dalle due sottoclassi.

*SortAlgFromTop*, 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 chiamata al metodo privato *sortRow()*, localizzazione

del pezzo a sud del primo precedentemente localizzato mediante metodo `nextInCol()` e così via fino al raggiungimento della soglia prevista (il campo dati `size` della classe *SortAlg*, inizializzato alla metà della lunghezza totale del puzzle). *SortAlgFromBottom* si occupa in maniera analoga di ordinare la metà inferiore del puzzle, ma questa volta l'algoritmo, per quanto si tratti essenzialmente delle stesse operazioni di *SortAlgSeq*, 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 ovest invece che ad est, fino al raggiungimento del punto d'incontro con il Thread della parte superiore.

Terminato il metodo `sort()`, entra in gioco l'oggetto condiviso, che permette di far sapere al thread inferiore quando il thread superiore ha terminato l'ordinamento e scritto la sua "parte" sull'oggetto puzzle condiviso.

## 2.1 I thread

Il programma avvia in tutti i casi 3 thread distinti (oltre al garbage collector), il main thread e i due thread di ordinamento. Per mantenere un approccio il più thread-safe possibile, sull'oggetto condiviso di tipo *Puzzle* 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.

Unica eccezione a quanto detto si verifica alla fine dell'esecuzione del metodo di ordinamento, all'interno del metodo `run()` delle due classi *Runnable*, qui infatti, mediante costrutti *synchronized* e *wait/notify* opportunamente racchiusi in loop di sicurezza al fine di evitare risvegli involontari (es: chiamate `notifyAll()`), il thread inferiore saprà quando il thread superiore ha terminato le proprie operazioni di ordinamento e riscrittura dell'oggetto puzzle e potrà appendere la propria parte di puzzle ordinata all'oggetto condiviso; analogamente il thread superiore saprà quando attendere che la parte inferiore sia stata totalmente ordinata prima di procedere alla sovrascrittura del puzzle. Entrambi i thread rimangono attivi concorrentemente fino alla fine delle operazioni da eseguire qualsiasi sia l'input (corretto) fornito.

## 2.2 SharedSortStat

Si tratta dell'oggetto condiviso che permette la comunicazione tra i due thread sullo stato di ordinamento del puzzle, è una classe piuttosto banale formata da tre campi dati di tipo boolean che indicano la fine o meno dell'ordinamento da parte dei thread e la fine o meno della sovrascrittura del thread superiore sull'oggetto puzzle, con rispettivi getters e setters tutti marcati *synchronized* per evitare interferenze.

## 2.3 Ricapitolando:

Assumendo che il funzionamento del metodo `sort()` in *SortAlgFromTop* sia analogo a *SortAlgSeq* della prima parte, e che in *SortAlgFromBottom* applichi la stessa logica all'opposto, ossia partendo dalla fine del puzzle e ricostruendolo al contrario, il metodo `run()` procede nei due thread come segue:

- [TOP] mediante riferimento a oggetto condiviso *SharedSortStat* segnala che la parte superiore è stata ordinata e apre un blocco *synchronized* sull'oggetto puzzle
- [TOP] all'interno di un `while` con condizione di attesa sul termine ordinamento da parte del thread inferiore, chiama il metodo `wait()` sull'oggetto puzzle, all'interno del blocco *try/catch* su eventuale *InterruptedException*

- [BOT] mediante riferimento a oggetto condiviso *SharedSortStat* segnala che la parte inferiore è stata ordinata e apre in blocco *synchronized* sull'oggetto puzzle
- [BOT] all'interno di un while con condizione di attesa sul termine ordinamento e sovrascrittura dell'oggetto puzzle da parte del thread superiore, chiama il metodo *notify()* sull'oggetto puzzle e chiama poi il metodo *wait()* sullo stesso all'interno di un blocco *try/catch* per eventuale *InterruptedException*
- a questo punto [TOP] trovandosi "sbloccato" il while e risvegliato il thread sa che [BOT] ha concluso le proprie operazioni di ordinamento e può procedere
- [TOP] sovrascrive l'oggetto puzzle con la propria metà ordinata, segnala all'oggetto condiviso di tipo *SharedSortStat* che la sovrascrittura è terminata e chiama il metodo *notify()* sul puzzle.
- [BOT] trovandosi "sbloccato" il while e risvegliato il thread può procedere ad aggiungere la propria parte di puzzle ordinato a quella già inserita da [TOP]

Questo sistema fa sì che qualsiasi sia l'ordine di arrivo dei due thread, sia sempre il superiore a sovrascrivere per primo il puzzle, ovviamente solo se la parte inferiore è stata conclusa, questo per evitare di eliminare pezzi che potrebbero ancora servire al thread inferiore. Il tutto viene impostato all'interno del metodo *sort()* della classe *PuzzleController*:

1. Calcolo metà del puzzle
2. Creazione riferimento a oggetto *SharedSortStat*
3. Creazione riferimento a oggetto *SortAlgFromBottom*, con parametri il puzzle da ordinare, *size / 2* e riferimento a oggetto *SharedSortStat*
4. Creazione riferimento a oggetto *SortAlgFromTop*, con parametri il puzzle da ordinare, *size / 2* e riferimento a oggetto *SharedSortStat*
5. Creazione riferimenti e avvio thread top e bot, con parametri i due riferimenti dei punti 3 e 4
6. join dei due thread all'interno di blocco *try/catch* su eventuale *InterruptedException*

## 2.4 Note

Nelle classi derivate da *SortAlg* sono stati aggiunti altri due campi dati di utilità, il primo è un *Vector<IPiece>*, serve a contenere la parte di puzzle ordinata dalla classe, che verrà poi utilizzata per sovrascrivere (nel caso del thread superiore) o verrà appesa all'oggetto puzzle (nel caso di thread inferiore).

Il secondo campo dati è un array che conterrà i pezzi del puzzle mediante una conversione del *Vector<IPiece>* passato al costruttore, la sua utilità è esclusivamente orientata al lato performance dell'algoritmo, in quanto risultava più semplice e veloce scorrere un array piuttosto che un *Vector*.

## 3 Compilazione ed esecuzione

---

Come per la prima parte del progetto, *make* e *make clean* sono a disposizione, assieme allo script *puzzlesolver.sh* per che permette di automatizzare pulizia-compilazione-esecuzione richiedendo sempre input e output file come parametri.