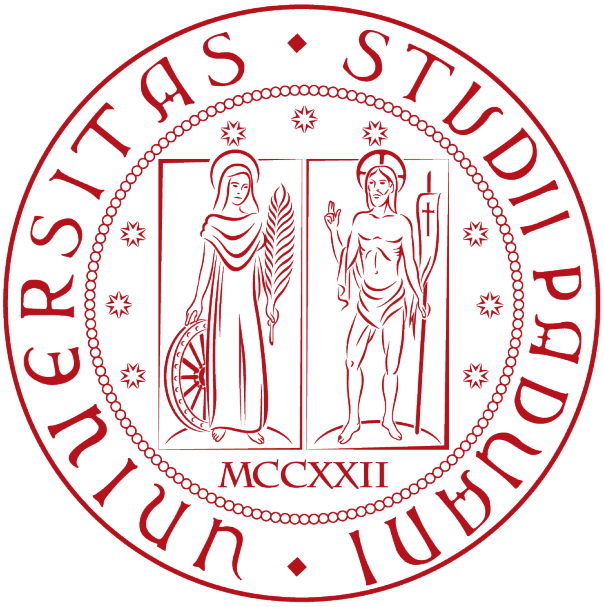

Risolutore di Puzzle

Parte 2

Stefano Munari mat. 1031243

Sommario:
Relazione del progetto di Programmazione concorrente e distribuita AA 2014-15

Indice

1 Note preliminari

Si è scelto di utilizzare alcuni costrutti presenti solamente dalla versione 7 di Java, pertanto per compilare il sorgente è necessario disporre di una versione >= Java7.
In particolare è stato usato il costrutto try-with-resources perchè piú conciso e semplice da utilizzare con gli Stream.
Comandi per compilazione ed esecuzione del progetto:

1. Per compilare il progetto lanciare il comando make da terminale.
2. Per eseguire il progetto scrivere la seguente riga da terminale dopo aver eseguito il punto precedente:
   sh puzzleSolver.bash nome_file_input nome_file_output

è necessario passare due parametri, come descritto al punto 2, per la corretta esecuzione del programma.

2 Logica dell'algoritmo parallelo

L'algoritmo concorrente si trova nel package algorithm in cui sono state inserite le classi che lo compongono (PuzzleSort.java e Column.java).
La logica dell'algoritmo è suddivisa in 3 macro-fasi:

1. *PuzzleSort():* Costruzione di un oggetto PuzzleSort, che rappresenta l'algoritmo, e inizializzazione dei campi dati ad esso relativi;
2. *sortColumns():* Ordinamento del puzzle per colonne;
   É a sua volta suddiviso in:
   (a) Creazione di N oggetti colonna (Column) ,
   con N= numero di colonne presenti nel puzzle, e costruzione della prima riga del puzzle (buildFirstRow);
   (b) Ordinamento della prima riga del puzzle (fetchFirstElement, rowSort);
   (c) Esecuzione di N threads paralleli, ogni thread risolve una colonna del puzzle:
   ogni thread è sincronizzato su un oggetto monitor condiviso con PuzzleSort, alla risoluzione di ogni colonna viene avvisato PuzzleSort(main thread) che, attraverso l'oggetto monitor, controlla che il numero di colonne ordinate corrisponda al numero totale di colonne del puzzle (vedi Figura 1);
   (d) Cancellazione di tutti gli oggetti appartenenti all'oggetto condiviso result.
3. *mergeColumns():* Unione delle colonne per righe inserendole all'interno della lista result.
   Si ottiene una lista di righe in ordine crescente (cioè dalla prima all'ultima).

PuzzleSort ora contiene il puzzle ordinato.

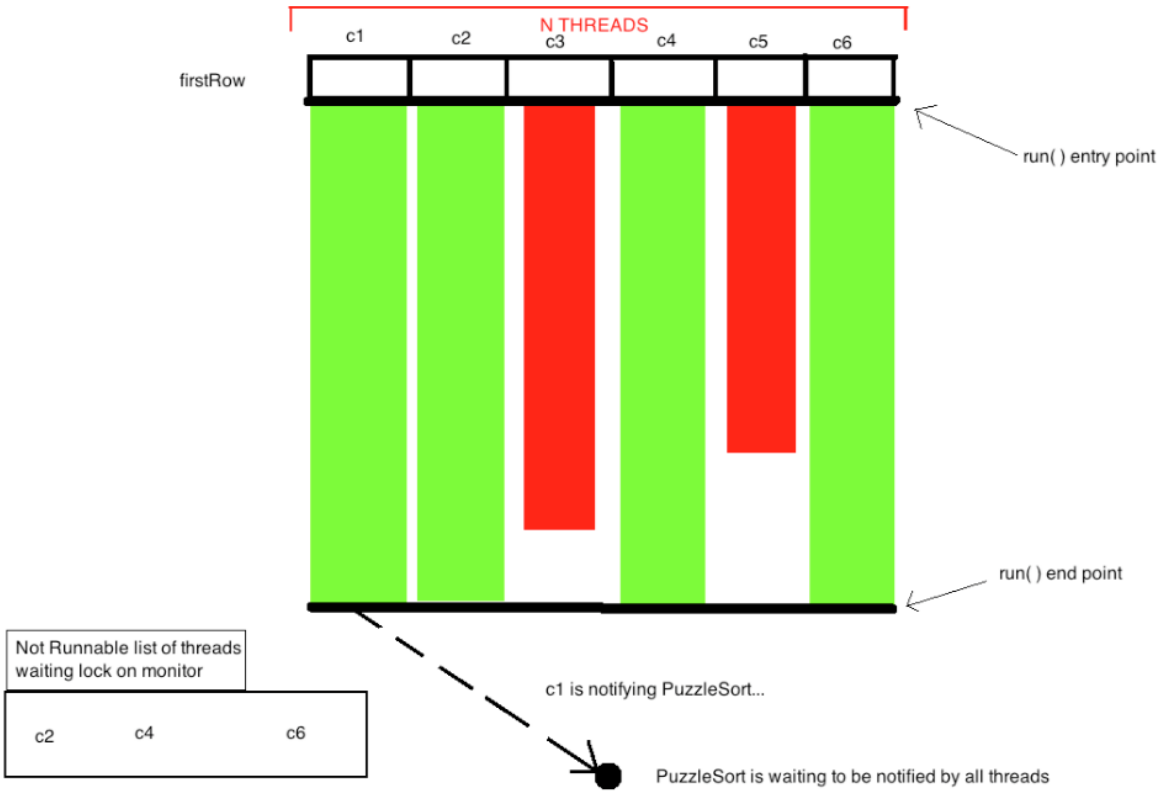

Figura 1: PuzzleSort::sortColumns() - parte concorrente dell'algoritmo

3 Threads

Quanti e quali thread vengono attivati :

Vengono avviati N thread con N che corrisponde esattamente al numero di colonne presenti nel puzzle da ordinare. Ogni thread avviato dal programma è di tipo Column, ciò significa che ogni thread ha le stesse caratteristiche e lo stesso comportamento.
Ritengo la scelta di attivare un numero di thread pari al numero di colonne una scelta sensata perchè ben si adatta alla risoluzione del problema, infatti in questo modo il carico di lavoro viene diviso equamente e indistintamente tra i thread.
In media piú colonne si hanno e piú grande sará il puzzle, di conseguenza si avranno piú thread attivi.

Numero di thread attivi concorrentemente :

Il numero di thread attivi concorrentemente è virtualmente N.
Nella pratica questo dipende fortemente dal tipo di CPU, dal numero di core fisici del processore, dal numero di core logici del processore, dalla politica di scheduling del sistema operativo, dal numero di processi in esecuzione.
Posso affermare che N sia il numero potenziale di thread che lavorano in parallelo in quanto **nessun thread attende per eseguire operazioni** presenti nel corpo della propria run(), ad eccezione dell'incremento dell'oggetto condiviso

monitor che avviene **sempre dopo** che il thread ha finito di ordinare la colonna che rappresenta.
L'ordinamento delle N colonne avviene dunque in parallelo, questo è possibile grazie al riferimento alla lista di pezzi del puzzle condiviso tra tutte le colonne e contenuto in PuzzleSort.
Quindi **si sta lavorando** in realtá solamente **su di una stessa lista condivisa** e non su N copie della lista non ordinata.
In corrispondenza della risoluzione di tutte le colonne avró un'occupazione della memoria pari a 2 liste (quella condivisa e quella risultante dalle copie eseguite dai thread column), lo stesso avviene anche quando si riuniscono le colonne ordinandole per righe. In entrambi i casi gli elementi della lista in piú vengono eliminati al termine delle operazioni dal main-thread.
L'incremento dell'oggetto condiviso (monitor) avviene in un blocco synchronized perchè, al fine di evitare race-condition, l'istruzione notifyAll() **deve avvenire in mutua esclusione** visto che serve per notificare il main-thread che a sua volta deve eseguire un controllo sullo stato dell'oggetto monitor.
Se un thread trova l'oggetto monitor occupato allora viene messo nella lista di attesa (thread not runnable) per quell'oggetto e quando ottiene il lock esegue l'incremento e notifica il main-thread.

4 Costrutti di concorrenza Java

I costrutti di programmazione concorrente del linguaggio Java che sono stati utilizzati nel progetto sono:

- *wait()*:
  Metodo della classe Object.
  Mette il thread nella lista di thread in attesa del lock sull'oggetto su cui wait() è invocato. Nel programma consegnato wait() viene usato per mettere il main-thread (attraverso PuzzleSort) in attesa di essere notificato dai vari thread Column;

- *notifyAll()*:
  Metodo della classe Object.
  Notifica tutti i thread nella lista di attesa del lock su cui notifyAll viene invocato. Nel programma consegnato viene utilizzato all'interno del metodo run() di ogni thread Column per notificare il main-thread;

- *synchronized*:
  la keyword synchronized permette la mutua esclusione di blocchi di codice o interi metodi nell'accesso all'oggetto che viene sincronizzato tra i vari thread. Ad esempio, nel programma consegnato, synchronized viene utilizzata sull'unico oggetto condiviso (monitor).

Sono stati inoltre utilizzati:

- *Thread*:
  classe appartenente alla libreria standard del linguaggio che permette di creare thread in un programma. Nel programma consegnato è stata specializzata dalla classe Column che rappresenta dei thread-colonna.

- *AtomicInteger*:
  classe appartenente alla libreria standard .concurrent che rappresenta un intero atomico. Il problema legato all'utilizzo di un semplice Integer (ricordando che possono essere condivisi solo oggetti) sta nel modo in cui quest'ultimo è implementato nel linguaggio. Ad esempio, un'operazione del tipo a++; crea un nuovo intero e quindi il riferimento condiviso sarebbe perduto. Vale invece il contrario per AtomicInteger. Nel programma consegnato AtomicInteger è il tipo dell'oggeto monitor.
  Usando AtomicInteger si è ritenuto piú sensato utilizzare una sincronizzazione lato client piuttosto che lato server per avere un codice piú semplice e comodo da leggere.

- *InterruptedException*:
  Eccezione sollevata quando un thread in attesa, sospeso o occupato viene interrotto.
  Nel caso specifico del progetto riguarda il metodo wait() di PuzzleSort::sortColumns(), quindi si riferisce al thread in attesa.

5 Correttezza della parte concorrente

Il programma consegnato è corretto per i seguenti motivi:

1. *Race-condition*:
   Non esiste alcun tipo di interferenza perchè gli oggetti che potrebbero crearla sono marcati come final e quindi non modificabili.
   Questa proprietá è assicurata anche dal passaggio di alcuni oggetti che avviene attraverso new(). In alcuni casi per evitare di creare interferenza l'oggetto diventa immutabile (final), in altri casi viene creata una copia (new()).
   Un caso particolare è l'oggetto monitor. Proprio per l'uso che ha nel programma l'oggetto è volutamente modificabile da tutti i thread ma in modo sincronizzato, quindi in mutua esclusione.

2. *Deadlock*:
   Non esiste alcun tipo di deadlock perchè non ci sono dipendenze circolari tra oggetti che sono sottotipi di thread.

3. *Attesa attiva*:
   Non esiste alcun tipo di attesa attiva perchè il main-thread viene messo nello stato not-runnable (e quindi rilascia la CPU e il lock su monitor) nel caso in cui la sua condizione non fosse verificata (attraverso wait()), mentre i thread di tipo Column sono messi nella lista di sincronizzazione (not runnable) sull'oggetto monitor qualora non riescano ad accedere al lock perchè occupato da altri thread-Column che al termine delle loro operazioni effettuano notifyAll() risvegliando i thread in attesa. Quindi ogni thread lavora solo quando ha qualcosa da eseguire.

4. *Bilanciamento del carico di lavoro tra thread*:
   La scelta di ordinare secondo colonne garantisce che il carico di lavoro sia bilanciato in egual misura per ogni thread.
   Questo perchè sapendo che in un puzzle valido/ordinabile ogni colonna ha la stessa lunghezza so anche che ogni thread del programma consegnato ha la stessa quantitá di oggetti Piece da ordinare. Quindi ogni thread avrá lo stesso carico di lavoro rispetto agli altri thread attivi, indipendentemente dalla grandezza del puzzle.

6 Modifiche rispetto alla versione precedente

Rispetto alla versione precedentemente consegnata (parte1) sono state apportate le seguenti modifiche:

1. Creazione della classe Column extends Thread:
   classe che rappresenta una colonna, il corpo del metodo run() corrisponde al corpo del vecchio metodo sortColumns() della parte1
   (eccetto la parte sincronizzata su monitor).
   Il vecchio metodo sortColumns() (vedi parte1), è stato dunque rimosso.
   Nella parte2 é stato modificato il nome del vecchio initializeColumns() (parte1) cambiandolo in sortColumns() (parte2) perchè piú adatto a descrivere il comportamento del metodo.

2. Il campo dati List<Piece> column era presente in parte1 internamente a PuzzleSort come List<List<Piece>> columns che in parte2 è invece List<Column> columns.

3. E' stato aggiunto il campo dati monitor per sincronizzare l'oggetto PuzzleSort con i vari thread della lista columns.

Tutte le modifiche apportate riguardano solamente il package algorithm, che contiene l'algoritmo d'ordinamento del puzzle.
Tutte le altre classi incapsulate negli altri package sono rimaste identiche alla parte1.