Mattia Sorgato, matricola 1004404

Relazione di Progetto di Programmazione Concorrente e Distribuita

Anno didattico 2014/2015

Seconda consegna - Parte Concorrente

Algoritmo di ordinamento Concorrente (metodo *sortPuzzle(List<Piece>)*)

L'ordinamento parallelo del puzzle nel mio progetto didattico è in qualche modo simile all'ordinamento sequenziale. Idealmente, nella sua realizzazione ho cercato di rendere parallelo l'ordinamento di ogni riga, cosicché il carico di ogni Thread fosse equo e ben distribuito.
Quindi, per ogni riga del puzzle, l'algoritmo crea un nuovo **SortLineThread** che si occupa di ordinare una singola riga, salvando il risultato dell'ordinamento su una lista di array condivisa con gli altri Thread, nella riga che gli compete.
Proprio per questa suddivisione in righe, la scrittura del risultato nell'oggetto condiviso non causa *race condition*, in quanto ogni Thread è responsabile di una singola riga e scrive solamente su quella.
Per ottenere questo risultato, però, è necessaria una prima passata iniziale sequenziale, per ricavare il primo membro di ogni riga e quindi il numero stesso di **SortLineThread** da creare. Questo risultato si ottiene invocando il metodo di classe *getLeftBorder(List<Piece>)*, il quale ritorna l'array contenente i primi pezzi di ogni riga.
Con questo array, si possono finalmente istanziare i **SortLineThread** necessari all'ordinamento.
Come si può intuire, nel processo di ordinamento, oltre al Main Thread sono attivi un numero **N** di Thread, con **N** il numero di righe del puzzle da ordinare.
Successivamente, la funzione di ordinamento aspetta che ogni Thread abbia finito, controllando il contatore *thread_ended* (di tipo **EndedMonitor**) fino a quando esso assume il valore desiderato, oppure quando uno dei Thread riscontra un problema nell'ordinamento. Questo lo si può notare dalla condizione del *while* di attesa posto nel blocco sincronizzato:

```
!(thread_ended.getEnded() == leftBorder.length) && allOk
```

In questa condizione si attende fino a che il numero di Thread conclusi non sia lo stesso dei Thread creati (quindi il numero di righe del puzzle), oppure fino a che la flag booleana *allOk* non assume valore **false**, ovvero quando un Thread riscontra dei problemi nell'ordinamento.
Avviene quindi un controllo dei risultati tramite il metodo *rowCheck()* e *compareSize(int, Piece[][])* e, se non si sono trovati errori, si prepara l'oggetto da ritornare con i valori corretti e successivamente viene ritornato al chiamante, non prima di aver reinizializzato i campi dati per un'operazione di ordinamento successiva della stessa istanza di **ConcurrentSort**.

Classe Interna SortLineThread

Questa classe, come si può intendere dal nome, estende la classe Thread ed esegue l'ordinamento di una singola riga.
Si può notare dal codice come ogni campo dato della classe interna sia marcato **final**, soprattutto per il campo ***private final List<Piece> puzzle***, che contiene il riferimento al puzzle disordinato ricevuto in input dall'utente. Questo garantisce quindi che avvengano solamente letture da questa lista, sebbene concorrenti, eliminando alla radice un possibile problema di *race condition* sul puzzle in entrata.

All'interno del metodo *run()* di ogni **SortLineThread**, viene creata una lista che conterrà i pezzi ordinati della riga **row** (l'intero che rappresenta la riga del **SortLineThread** corrente) del futuro puzzle ordinato.
Ora, visto che ogni Thread non conosce l'effettiva lunghezza della propria riga, per evitare che (per un **ID** malevolo) il Thread vada in esecuzione infinita, ho attuato il seguente stratagemma. Il numero massimo di iterazioni che un **SortLineThread** può compiere per trovare tutti i pezzi della sua riga è il caso in cui si abbia un puzzle 1xN e che i pezzi siano stati posti in ordine inverso. Quindi, il numero delle iterazioni dell'ordinamento diventa *puzzle.size() + (puzzle.size() - 1) + (puzzle.size() - 2) + ... + 1*. Il numero massimo di iterazioni si riconduce quindi al problema della somma dei primi N numeri naturali, la cui formula è Somma = (N * (N + 1)) / 2. Tenendo quindi questo numero come limite massimo delle iterazioni, ogni Thread scorre il puzzle ordinando la propria riga, aggiungendo alla lista locale ogni pezzo trovato, fino a giungere al pezzo con *id_est* uguale a VUOTO. Quando esce dal ciclo, viene controllato di essere effettivamente giunti al bordo est, se sì, può scrivere in *orderedPuzzle,* nella riga che gli compete, il risultato dell'ordinamento. Altrimenti scrive nella riga che gli compete un valore **null** e cambia la flag di **ConcurrentSort** *allOk* a **false**, per indicare che il processo di ordinamento ha riscontrato dei problemi. Fatto questo, incrementa il numero di Thread terminati.

Classe Interna EndedMonitor

Questa classe interna privata serve come contatore di monitor dei Thread terminati. Il metodo *sortPuzzle(List<Piece>)* attende che i Thread che ha creato abbiano finito la loro esecuzione (oppure che almeno uno di loro abbia riscontrato problemi), comparando il numero di Thread partiti con quello dei Thread conclusi. Per fare ciò, serve un contatore di monitor che sia acceduto concorrentemente dai vari Thread e dal Main stesso. Da qui la creazione di EndedMonitor, la cui istanza è presente in **ConcurrentSort** e il quale contiene metodi per l'incremento e per la lettura del contatore. Entrambi i metodi sono marcati **synchronized**, in modo da non causare modifiche contemporanee del contatore che porterebbero a stati inconsistenti e potenzialmente dannosi. L'invocazione di *incrementEnded()* porta inoltre all'invocazione del metodo *notify()*, il quale risveglia il Main Thread in attesa nel metodo *sortPuzzle(List<Piece>)*, notificandogli che un Thread ha terminato la sua esecuzione.

Costrutti di Concorrenza

All'interno del mio progetto, i campi dati ed i blocchi di sezioni critiche sono protetti da metodi di accesso sincronizzati e da opportuni lock. Questo è il caso del contatore dei Thread conclusi, il quale accesso in lettura (nel metodo *sortPuzzle(List<Piece>)*) e il suo incremento (nel corpo della *run()* dei vari **SortLineThread**) sono regolati da metodi sincronizzati, per evitare la modifica concorrente dannosa del contatore utilizzato.
Inoltre, lo stesso metodo *sortPuzzle(List<Piece>)* è marcato **synchronized**, in modo che due o più invocazioni dello stesso metodo con una stessa istanza di **ConcurrentSort** non creino conflitto nei contatori o nei campi dato dell'istanza.
L'utilizzo dei costrutti *wait()* e *notify()* si ritrova sempre nella gestione del contatore sopra menzionato. Ovvero, il Main Thread si mette in attesa fino a quando tutti i Thread sono terminati; viceversa, ogni Thread alla sua conclusione invoca il metodo *incrementEnded()* di **EndedMonitor**, il quale notifica il Main Thread di una nuova conclusione e lo fa risvegliare dalla sua attesa, per poi accertarsi dell'avvenuto cambiamento nella condizione del *while* della *wait()*.
Ogni *wait()* è ovviamente correlata di un'opportuna clausola **catch** che gestisce eventuali **InterruptedException** lanciate dall'interruzione del Thread mentre era in attesa.

Cambiamenti rispetto alla versione precedente

La nuova classe **ConcurrentSort** è un'ulteriore implementazione dell'interfaccia **ISort**, quindi il vincolo di ereditarietà permette di interscambiare la versione sequenziale **SequentialSort** con questa concorrente con molta facilità, visto che l'interfaccia parentale è la stessa. In particolare, è bastato istanziare un **ConcurrentSort** al posto del precedente **SequentialSort** nel main della classe **PuzzleSolver** per ottenere l'effetto desiderato.
Dalla versione precedente, però, ho deciso di arricchire suddetta interfaccia con metodi utili per entrambi gli algoritmi di ordinamento, come il metodo *getUpperLeft(List<Piece>)*, che ritorna il riferimento al pezzo in angolo in alto a sinistra, e il metodo *compareSize(int, Piece[][])*, utile ad eseguire il confronto tra la dimensione originaria del puzzle e quella della matrice ordinata. Ovviamente la classe **SequentialSort** è stata adattata a tali modifiche.
Inoltre, ora l'interfaccia dispone di una classe interna statica chiamata **DefaultSort** contente l'implementazione di default dei due metodi sopracitati, cosicché alle implementazioni dell'interfaccia basti solamente recuperare un'istanza della classe interna e realizzare il cosiddetto "forwarding" dei metodi a quell'oggetto per avere i metodi già implementati.
Sono stati inoltre inseriti altri due test nel pacchetto **puzzlesolver.tests** per dimostrare il corretto funzionamento del metodo *getLeftBorder(List<Piece>)* e dello stesso algoritmo di ordinamento concorrente. Queste due test prendono in esame diversi casi e vengono spiegati nei commenti e/o negli output del programma. Le classi dei test in questione sono **FirstColumnTest** e **ConcurrentSortTest**.

Bash

Nel root del mio progetto ho creato un file *bash* chiamato **puzzlesolver.sh.** Questo bash esegue prima di tutto una pulizia dei compilati e una successiva ricompilazione, chiamando il **make** incluso nella *root* del progetto. Successivamente, invoca il comando di esecuzione dell'eseguibile del *main* di **PuzzleSolver** con degli argomenti dati, come espresso in specifica. Il comando prende due argomenti, il percorso del file di input ed il percorso del file di output.

Nota: Se viene eseguito il comando *bash,* o anche tramite il **make***,* i compilati vengono posizionati ordinatamente nella cartella **bin.**

Se non si volesse utilizzare il *bash* per eseguire il progetto, una volta compilati i file necessari (utilizzando il compilatore di Java7), occorre posizionarsi nella cartella dei compilati (che nel caso del *bash* è **bin**) e chiamare il comando

java -cp . puzzlesolver.PuzzleSolver path_input path_output

Note

All'interno del metodo *run()* dei vari Thread sono state poste delle stampe sullo standard output che esprimono l'inizio e la terminazione di ogni Thread, identificati da un numero intero >= 1 che esprime la loro riga di appartenenza. Ho messo queste stampe per una questione di chiarezza e per rendere più facile il testing e la correzione.