Sebastiano Valle - 1050123

Risolutore di puzzle – Parte 2
Programmazione concorrente e distribuita
Progetto A.A. 2014/2015

1 Algoritmo parallelo di risoluzione

Dato un mucchio (possibilmente disordinato) contenente tutti i tasselli di un puzzle e una tabella con *rows* righe e *cols* colonne, l'algoritmo procede nel seguente modo:

1. Individuazione del lato *edge*, ovvero il lato più lungo tra quello superiore (north) ed il sinistro (west)
2. Riempimento del lato *edge* in modo sequenziale, partendo dall'angolo superiore sinistro
3. Riempimento in parallelo di ogni linea (colonna o riga), partendo dai tasselli definiti sul lato *edge* e procedendo, per ogni linea, dall'alto verso il basso o da sinistra a destra a seconda che *edge* corrisponda rispettivamente il lato superiore o quello sinistro

Il passo 2 dell'algoritmo è svolto grazie a delle istanze di *EdgeSorter* di *Puzzle*, create in quantità pari alla lunghezza del lato *edge*.

Il passo 3 dell'algoritmo non necessita che il passo 2 finisca per cominciare; per ogni linea è istanziato un oggetto della classe *LineSorter* di Puzzle; tuttavia, ogni linea necessita che i tasselli iniziali della rispettiva linea (colonna o riga) siano stati collocati per poter procedere col riempimento dell'intera riga.

2 Gestione dei thread

2.1 fill(path : Path) in Heap avvia Filler (Classe)

Chiamando n il numero di tasselli contenuti nel file di input, definisco r come √n arrotondato per eccesso.

Il metodo *fill(path : Path)* avvia r thread *Filler* che inseriscono a loro volta al massimo r tasselli ciascuno nell'oggetto che rappresenta una mucchio di tasselli.

2.2 sort() in Puzzle avvia EdgeSorter (Classe)

Dopo aver individuato il lato *edge* di lunghezza maggiore, l'istanza di *Puzzle* avvia un thread *EdgeSorter* che ricerca e colloca tutti i tasselli del lato *edge* del puzzle, estrandoli dal mucchio dei tasselli non ancora collocati; questo thread può essere attivo concorrentemente con i thread della classe *LineSorter*.

2.3 sort() in Puzzle avvia LineSorter (Classe)

Dopo aver avviato il thread della classe *EdgeSorter* che ha il compito di collocare i tasselli del lato *edge*, l'istanza di *Puzzle* avvia tanti thread *LineSorter* quanti sono i tasselli sul lato *edge* (questo numero viene chiamato *limit*).

Questi thread riempiono ciascuno una linea che parte da ogni tassello sul lato *edge* e che si estende fino al lato opposto di edge; per far questo hanno bisogno che il tassello sul lato *edge* della rispettiva linea sia inizializzato da un oggetto *EdgeSorter* e rimangono in attesa non attiva finché questa condizione non è verificata.

Quando il tassello iniziale è stato collocato, ognuno di questi thread ricerca e colloca tutti i tasselli della linea che devono riempire, estraendoli dal mucchio dei tasselli non ancora collocati; questo thread può essere attivo concorrentemente con un thread della classe *EdgeSorter* e con *limit*-1 thread della classe *LineSorter*.

3 Costrutti di concorrenza utilizzati

3.1 fill(path : Path) in Heap

Nel metodo *fill(path : Path)* vengono utilizzati due costrutti Java per la gestione di thread: con **start()**, si avviano i thread che aggiungono al mucchio una porzione della collezione di tasselli; con **join()** si verifica che il metodo di riempimento termini quando tutti i thread hanno popolato il mucchio con la sezione di collezione di loro competenza.

3.2 Filler (Classe) in Heap

Nella classe *Filler* la concorrenza viene gestita col costrutto **synchronized**(), in modo da serializzare le aggiunzioni al mucchio anziché permettere aggiunzioni concorrenti, possibilmente rischiose.

3.3 sort() in Puzzle

Nel metodo *sort()* viene avviato un thread *EdgeSorter* tramite il costrutto **start**(), così che con l'esecuzione del metodo *run*() vengano collocati tutti i tasselli sul lato *edge*.

Successivamente, sempre tramite il costrutto **start**(), vengono avviati tanti thread LineSorter quanti sono i tasselli del lato *edge*, in modo tale da ordinare tutte le linee del puzzle che partono da un tassello del lato *edge* e arrivano al lato opposto.

Infine il metodo *sort*() aspetta, tramite il costrutto **join**(), che tutti i thread della classe *LineSorter* terminino; quando ognuno di questi termina, allo stesso tempo una linea del puzzle è ordinata. Siccome erano stati avviati tante istanze di *LineSorter* quante erano le linee da ordinare, al termine di queste il puzzle è completamente ordinato.

3.4 setEdgePiece(position : int, piece : PuzzleItem) in EdgeSorter (Classe) in Puzzle

Nella classe *EdgeSorter* è definito un metodo privato *setEdgePiece(position : int, piece : PuzzleItem)*, in cui, tramite il costrutto **notifyAll**(), vengono risvegliati tutti i monitor che si sono posti in uno stato di attesa sulla matrice dei pezzi del puzzle. È utilizzato il costrutto

synchronized() così da prendere il lock sulla matrice e poter richiedere il risveglio di tutti gli altri monitor.

Viene utilizzato il costrutto **notifyAll**() anziché **notify**() siccome ci potrebbero essere più monitor di thread *LineSorter* in attesa che il primo elemento della loro linea venga collocato.

3.5 LineSorter (Classe) in Puzzle

I thread della classe *LineSorter* necessitano che il primo tassello della loro linea sia collocato da un thread *EdgeSorter*. Perciò, fin questo non succede, essi si pongono in attesa col costrutto **wait**() sulla matrice dei tasselli del puzzle, aspettando di essere risvegliati da un'istanza di EdgeSorter che sta riempiendo il lato *edge* dello stesso puzzle. Naturalmente, per richiedere un'attesa sulla matrice dei tasselli del puzzle, viene richiesto il lock su questa tramite il costrutto **synchronized**().

Per il resto della riga non viene utilizzato nessun costrutto per gestire la concorrenza perché ogni thread opera su porzioni distinte della matrice.

3.6 getPiece(id : String) in Heap

Questo metodo si occupa di estrarre un pezzo dal mucchio, dato il suo id. Siccome bisogna gestire situazioni del tipo check-then-act, viene richiesto l'accesso esclusivo all'oggetto che rappresenta il mucchio di tasselli.

Questo obiettivo è stato raggiunto aggiungendo **synchronized**() alla firma del metodo, in modo tale da richiedere il lock su **this** (ovvero l'oggetto che rappresenta il mucchio) ad ogni invocazione del metodo.

3.7 getEdgePiece(edge : Dir, refSide : Dir, ref : String) in Heap

Questo metodo si occupa di estrarre un pezzo dal mucchio, conoscendo il lato del puzzle su cui si trova e l'id di uno dei riferimenti su uno dei suoi lati, ma non il lato su cui si trova. Siccome bisogna gestire situazioni del tipo check-then-act, viene richiesto l'accesso esclusivo all'oggetto che rappresenta il mucchio di tasselli, analogamente al metodo descritto nel paragrafo 3.5.

Questo obiettivo è stato raggiunto aggiungendo **synchronized**() alla firma del metodo, in modo tale da richiedere il lock su **this** (ovvero l'oggetto che rappresenta il mucchio) ad ogni invocazione del metodo.

4 Correttezza della concorrenza

4.1 fill(path : Path) in Heap

4.1.1 Interferenze

Nel metodo *fill(path : Path)* non avvengono interferenze poiché non vi è alcun accesso concorrente ai dati; le possibili interferenze dovute ai thread *Filler* che vengono avviati verranno discusse nel paragrafo 4.2.1.

4.1.2 Deadlock

Nel metodo *fill(path : Path)* vi è un possibile rischio di deadlock a causa dei costrutti **join**(); tuttavia

questo rischio non è possibile che si presenti durante l'esecuzione poiché ogni thread *Filler* che viene avviato termina sicuramente, come si può vedere nel paragrafo 4.2.2; se l'esecuzione dovesse venire interrotta, l'eccezione viene catturata tramite un costrutto **try-catch** dove sono rinchiuse le **join**() dei thread.

4.1.3 Busy wait

Nel metodo *fill(path : Path)* non vi è attesa attiva, poiché l'unico momento in cui il metodo si pone in attesa è quando aspetta che i thread *Filler* terminino, ma dal momento che viene utilizzato il costrutto **join**(), il metodo pone il proprio flusso di controllo in attesa non attiva aspettando che I thread terminino.

4.2 Filler (Classe)

4.2.1 Interferenze

Nel metodo *run()* della classe *Filler* non avvengono interferenze poiché l'accesso alla struttura dati "*mucchio*" (una ArrayList) è rinchiuso in un blocco **synchronized**() che prende il lock sul mucchio, garantendo così l'aggiunta di tasselli senza interferenze.

4.2.2 Deadlock

Nel metodo *run()* della classe *Filler* non vi sono deadlock perché l'unica risorsa che tutti richiedono, ovvero il mucchio, è richiesta solamente da istanze di *Filler* che quando ne prendono il lock aggiungono un tassello e successivamente liberano subito la risorsa.

4.2.3 Busy wait

Nel metodo *run*() della classe *Filler* non vi è attesa attiva, poiché quando le istanze di *Filler* attendono che venga rilasciato il lock sulla risorsa mucchio, queste si pongono in attesa non attiva.

4.3 sort() in Puzzle

4.3.1 Interferenze

Nel metodo *sort()* della classe P*uzzle* non avvengono interferenze poiché non vi è alcun accesso concorrente ai dati; le possibili interferenze dovute ai thread *EdgeSorter e LineSorter* che vengono avviati verranno discusse nei paragrafi 4.4.1 e 4.5.1.

4.3.2 Deadlock

Nel metodo *sort()* della classe P*uzzle* vi è un possibile rischio di deadlock a causa dei costrutti **join**() che causano l'attesa del termine delle istanze di *LineSorter*; tuttavia questo rischio non è possibile che si presenti durante l'esecuzione poiché ogni thread *LineSorter* che viene avviato termina sicuramente, come si può vedere nel paragrafo 4.5.2; se l'esecuzione dovesse venire interrotta, l'eccezione viene catturata tramite un costrutto try-catch dove sono rinchiuse le **join**() dei thread. Si assume che tutti i tasselli siano forniti in modo corretto, ovvero che per ognuno di essi esista ogni altro tassello riferito sui quattro lati, se indicato diversamente da "VUOTO".

4.3.3 Busy wait

Nel metodo *sort()* della classe P*uzzle* non vi è attesa attiva, poiché l'unico momento in cui il metodo si pone in attesa è quando aspetta che i thread *LineSorter* terminino, ma dal momento che viene

utilizzato il costrutto **join**(), il metodo pone il proprio flusso di controllo in attesa non attiva aspettando che i thread terminino.

4.4 setEdgePiece(position : int, piece : PuzzleItem) in EdgeSorter (Classe) in Puzzle

4.4.1 Interferenze

Nel metodo *setEdgePiece(position : int, piece : PuzzleItem)* della classe *EdgeSorter* non avvengono interferenze poiché ogni volta che un pezzo viene collocato, nessun altro thread avrà mai come compito di collocare un tassello nella stessa posizione.

4.4.2 Deadlock

Nel metodo *setEdgePiece(position : int, piece : PuzzleItem)* della classe *EdgeSorter* non vi sono deadlock perché l'unica risorsa su cui tutti richiedono il lock, ovvero la matrice, è richiesta altrimenti soltanto da istanze di *LineSorter* che quando ne prendono il lock lo rilasciano immediatamente; questo perché se il tassello non è stato ancora collocato da *EdgeSorter*, lo attendono e rilasciano il lock, oppure se il tassello è stato collocato, non fanno alcuna operazione e rilasciano immediatamente il lock.

4.4.3 Busy wait

Nel metodo *setEdgePiece(position : int, piece : PuzzleItem)* della classe *EdgeSorter* non vi è attesa attiva, poiché quando le istanze di *EdgeSorter* attendono che venga rilasciato il lock sulla matrice, queste si pongono in attesa non attiva.

4.5 LineSorter (Classe) in Puzzle

4.5.1 Interferenze

Nel metodo run*()* della classe *LineSorter* non avvengono interferenze poiché dopo che *EdgeSorter* ha collocato il tassello iniziale della linea, ogni volta che un pezzo viene collocato in una linea nessun altra istanza di *LineSorter* sarà incaricata di collocare pezzi sulla stessa linea dell'istanza presa in considerazione.

4.5.2 Deadlock

Nel metodo run*()* della classe *LineSorter* non vi sono deadlock poiché l'istanza di *EdgeSorter* che inizializza il lato *edge* (dove sono situati i tasselli delle linee ordinate dalle istanze di *LineSorter*) procede collocando tutti i tasselli iniziali delle linee, perciò prima o poi, quando l'istanza di *LineSorter* presa in considerazione riuscirà a prendere il lock sulla matrice dei tasselli, troverà il proprio pezzo iniziale collocato e potrà procedere a riempire il resto della linea.

4.5.3 Busy wait

Nel metodo *run*() della classe *LineSorter* non vi è attesa attiva, poiché quando le istanze di *LineSorter* attendono che venga rilasciato il lock sulla matrice aspettando il tassello iniziale della linea, queste si pongono in attesa non attiva tramite il costrutto **wait**().

4.6 getPiece(id : String) in Heap

4.6.1 Interferenze

Nel metodo *getPiece(id : String)* della classe *Heap* non avvengono interferenze poiché il metodo contiene nella propria firma **synchronized**(): in tal modo solo una singola istanza alla volta ha il permesso di accedere ad un mucchio di tasselli per estrare un tassello.

4.6.2 Deadlock

Nel metodo *getPiece(id : String)* della classe *Heap* non vi sono deadlock poiché questo scorre linearmente senza richiedere il lock su alcuna risorsa. Allo stesso modo, questo metodo non è fonte di deadlock per nessuna classe che lo invoca poiché la sua terminazione avviene sempre, dopo aver scorso l'intero mucchio di tasselli (nel caso peggiore).

4.6.3 Busy wait

Nel metodo getPiece(id : String) della classe Heap non vi è attesa attiva, il metodo scorre linearmente senza dover richiedere il lock su alcuna risorsa.

4.7 getEdgePiece(edge : Dir, refSide : Dir, ref : String) in Heap

4.7.1 Interferenze

Nel metodo *getEdgePiece(edge : Dir, refSide : Dir, ref : String)* della classe *Heap* non avvengono interferenze poiché il metodo contiene nella propria firma **synchronized**(): in tal modo solo una singola istanza alla volta ha il permesso di accedere ad un mucchio di tasselli per estrare un tassello.

4.7.2 Deadlock

Nel metodo *getEdgePiece(edge : Dir, refSide : Dir, ref : String)* della classe *Heap* non vi sono deadlock poiché questo scorre linearmente senza richiedere il lock su alcuna risorsa. Allo stesso modo, questo metodo non è fonte di deadlock per nessuna classe che lo invoca poiché la sua terminazione avviene sempre, dopo aver scorso l'intero mucchio di tasselli (nel caso peggiore).

4.7.3 Busy wait

Nel metodo *getEdgePiece(edge : Dir, refSide : Dir, ref : String)* della classe *Heap* non vi è attesa attiva, il metodo scorre linearmente senza dover richiedere il lock su alcuna risorsa.

5 Variazioni

Le variazioni apportate al progetto sono le seguenti:

1. l'inserimento della concorrenza tramite la modifica di metodi o la creazione di classi interne;
2. la classe *Puzzle* presentava un errore (mancava un -1 in un ramo di un **if**-**else**) e l'errore è stato corretto;
3. lo script bash per l'avvio del programma è stato uniformato come da specifiche;
4. è stata aggiunto uno script bash *test.sh* nella cartella Testing tale che fornendo un file composto da una sola riga di caratteri, questo generi in automatico un puzzle e ne verifichi la correttezza, salvando gli esiti in un log presente nella stessa cartella.

5.1 PuzzleSolver (Classe)

In questa classe sono state rimosse le righe commentate che, se decommentate, permettevano di eseguire un test del programma.

5.2 Driver (Classe)

Come anticipato nell'introduzione, è stata creata una classe *Driver* , dotata di un metodo *main(args : String[])* così da poter essere facilmente avviata tramite un apposito script bash *test.sh* per verificare il corretto funzionamento di *PuzzleSolver*.

Questo test fa lo stesso test descritto nella sezione 4.2 della relazione sulla prima parte del progetto.

5.3 test.sh (Script bash)

Questo script avvia il test previsto per il programma nel metodo *main* della classe *Driver*.

5.4 Heap (Classe)

Questa classe è stata modificata in modo tale che il riempimento del mucchio di tasselli fosse reso parallelizzabile avviando dei thread appartenenti a *Filler*, classe interna, privata e non-statica. Inoltre è stata aggiunta la keyword **synchronized**() ai metodi *getPiece(id : String)* e *getEdgePiece(edge : Dir, refSide : Dir, ref : String)*.

Il metodo *fill(path : Path)* è stato modificato avviando istanze della classe *Filler* per riempire il mucchio di tasselli, come descritto nelle precedenti sezioni.

I metodi *isEmpty()* e *conta(side : Dir)* sono stati marcati **synchronized** per prevenire interferenze in possibili situazioni di race condition. La loro correttezza non è stata discussa poiché il programma *PuzzleSolver* non li impiega e poiché questa è triviale, poiché analoga a casi più complessi trattati nella sezione 4.

5.4.1 Filler (Classe)

Filler è una classe interna, non-statica e privata in quanto:

- Privata perché sono di utilità solamente per la classe *Heap*;
- Non-statica perché aggiungono tasselli ad una collezione rappresentata come campo dati nella classe *Heap*, ovvero l'**Outer.this**.

Questa classe concorre a riempire il mucchio di tasselli leggendo una parte dei tasselli letti dall'input, per poi inserirli nel mucchio evitando interferenze.

5.5 Puzzle (Classe)

Nella classe *Puzzle* è stato corretto un errore che comprometteva il funzionamento dell'intero programma; è stato inserito un “j-1” al posto di “j” in un subscript nella *run*() di *LineSorter*, classe che sostituisce il metodo *orderLine(top : Dir, i : int)* presente nella classe *Puzzle* nella prima parte del progetto, che presentava lo stesso errore in uno dei due rami condizionali.

Alcuni metodi, come *conta(side : Dir)*, *isEmpty()* e *getPiece(id : String)*, *getEdgePiece(edge : Dir, refSide : Dir, ref : String)* sono stati marcati **synchronized** così da prevenire interferenze o esiti

multipli in caso di loro utilizzo.

Sono state aggiunte due classi interne *EdgeSorter* e *LineSorter* private e non-statiche per rendere concorrente l'ordinamento del puzzle, e di conseguenza è stato modificato il metodo *sort()* di *Puzzle*. Questo metodo ora individua il lato più lungo, dopodichè avvia un thread della classe *EdgeSorter* che ordina il lato maggiore del puzzle; finché è possibilmente ancora in esecuzione questa istanza di *EdgeSorter*, il flusso di controllo procede nel metodo *sort()* e avvia tante istanze di *LineSorter* quanti sono i tasselli nel lato maggiore del puzzle. Il metodo *sort()* termina solamente quando tutte le linee processate dalle varie istanze di *LineSorter* sono state ordinate.

5.5.1 PuzzleItem (Classe)

Nella classe interna *PuzzleItem* sono stati marcati **final** i campi id ed i riferimenti, che rimanevano immutati dopo che i tasselli erano stati creati.

5.5.2 EdgeSorter (Classe)

EdgeSorter è una classe interna, non-statica e privata in quanto:

- Privata perché sono di utilità solamente per la classe *Puzzle*;
- Non-statica perché colloca tasselli sul lato maggiore del puzzle nella matrice della classe *Puzzle*, ovvero l'**Outer.this**.

Questa classe rappresenta un flusso di controllo che si occupa di collocare i tasselli del puzzle nel lato maggiore di questo; la sua esecuzione può avvenire parallelamente a quella dei flussi di controllo che riempiono le linee che iniziano dai tasselli del lato maggiore, con la condizione che queste aspettino che il primo tassello della linea venga collocato.

5.5.3 LineSorter (Classe)

LineSorter è una classe interna, non-statica e privata in quanto:

- Privata perché sono di utilità solamente per la classe *Puzzle*;
- Non-statica perché le sue istanze collocano tasselli nelle linee che partono dal lato maggiore del puzzle nella matrice della classe *Puzzle*, ovvero l'**Outer.this**.

Questa classe rappresenta un flusso di controllo che si occupa di collocare i tasselli del puzzle nelle linee che partono dal lato maggiore di questo; la sua esecuzione può avvenire parallelamente a quella dei flussi di controllo che riempiono le altre linee che iniziano dai tasselli del lato maggiore e concorrentemente con *EdgeSorter*, a patto che questa istanza di *LineSorter* aspetti che il primo tassello della linea che deve riempire venga collocato dall'istanza di *EdgeSorter*.

6 Note

Per la prima parte di progetto il codice è stato corretto, così che questo funzioni correttamente con tutti i puzzle che vengono forniti in ingresso.