Mattia Sorgato, matricola 1004404

# Relazione di Progetto di Programmazione Concorrente e Distribuita

## Anno didattico 2014/2015

## Seconda consegna - Parte Concorrente

### Algoritmo di ordinamento Concorrente

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 un oggetto condiviso 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.

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* fino a quando esso assume il valore desiderato, oppure quando uno dei Thread riscontra un problema nell'ordinamento.

Avviene quindi un controllo dei risultati e si restituisce in output il puzzle ordinato.

### 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>)* tiene 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 Thread in attesa del metodo *sortPuzzle(List<Piece>)*, notificandogli che un Thread ha terminato la sua esecuzione.