Homework di Programmazione Concorrente
22 dicembre 2017 — anno accademico 2017/2018

Organizzazione dell'homework

L'homework si compone di diverse parti di cui è obbligatoria solo la prima. Le altre, che pure contribuiscono alla valutazione, sono facoltative.

La prima parte prevede la risoluzione di un *problema di decomposizione parallela dinamica ed universale* utilizzando direttamente il *Lightweight Executable Framework* fornito dal package `java.util.concurrent`, e relativo testing di unità; la seconda parte prevede la risoluzione di una variante del medesimo problema con un aggiuntivo vincolo sull'occupazione di memoria; la terza parte prevede di scrivere una soluzione utilizzando il *Framework Fork/Join*; la quarta parte prevede la risoluzione utilizzando alcuni strumenti disponibili solo a partire da Java 8.

Oltre all'implementazione di un algoritmo facente utilizzo di specifiche librerie, ciascuna parte richiede una sperimentazione tesa a misurare lo speed-up ottenibile dalla soluzione prodotta. La misurazione dei tempi di esecuzione ed il calcolo dello *speed-up* devono essere eseguiti *automaticamente*. Il codice prodotto allo scopo deve ovviamente essere riutilizzato per tutte le soluzioni prodotte.

Modalità di consegna

L'homework va consegnato entro le ore 20:00 di domenica 7 gennaio inviando al docente una mail con subject "`PC: HWJ` *Nome Cognome Matricola*" ed allegando in formato `.tar.gz`, `.zip` oppure `.jar` tutti i sorgenti (codice e testo) prodotti e max cinque file `hwj1.jar`, `hwj2.jar`, `hwj3.jar`, `hwj4.jar`, per eseguire le soluzioni corrispondenti alle parti consegnate, più, in aggiunta, un file `HWJ.jar` che provvede ad eseguirli tutti (ed a stampare lo speed-up raggiunto da ciascuna soluzione: possibilmente mantenere il tempo di esecuzione complessivo sotto i 2 minuti).

Non inserire nell'archivio eseguibili ed in generale documenti che non siano sorgenti ad esclusione dei file `.jar` suddetti, oltre ad eventuali librerie

esterne. È possibile allegare un solo documento di testo (`doc.txt`)[1] contenente alcune note esplicative su ciascuna delle parti consegnate a supporto della comprensione (e non in sostituzione) del codice consegnato.

Riportare nel documento anche i risultati sperimentali ottenuti direttamente sulla piattaforma di sviluppo. I risultati degli eseguibili sviluppati per ciascuna delle parti di questo homework (e dei relativi test, se previsti) si devono poter ottenere eseguendo (all'interno della directory principale dell'archivio consegnato) unicamente `java -jar hwj<P>.jar` dove `<P>`$= 1, \ldots, 4$, per eseguire le soluzioni alle singole parti; oppure, ancora più semplicemente, con `java -jar HWJ.jar` per eseguirle tutte sequenzialmente.

Condizioni necessarie per la consegna

Per la consegna sono necessarie tutte le seguenti condizioni:

- Il codice principale, ed i relativi test, devono compilare, funzionare, ed essere perfettamente allineati; il documento di testo deve contenere una descrizione della soluzione fedelmente allineata al codice sviluppato.
- Per la scrittura dei test di unità in Java viene richiesto di utilizzare il framework di testing JUnit (versione 4 o successive) utilizzando le annotazioni (http://www.junit.org).
- È necessario sviluppare dei test-case che garantiscano copertura ragionevole (e quindi comunque *non* esaustiva) di alcuni degli scenari di test più rappresentativi.
- Per lo sviluppo della seconda parte è necessario utilizzare il *Lightweight Executable Framework* e *non* è possibile limitarsi ad utilizzare i meccanismi *nativi* per la scrittura di codice concorrente previsti dal linguaggio Java già prima della versione 5
- Per lo sviluppo della terza parte è necessario utilizzare almeno la versione 7 della piattaforma Java.
- Per lo sviluppo della quarta parte è necessario utilizzare almeno la versione 8 della piattaforma Java.
- Per effettuare le misurazioni sperimentali richieste, utilizzare una piattaforma hardware con almeno due processori fisici e prevedere la completa automatizzazione delle misurazioni e del calcolo dello speed-up tenendo conto del numero di processori fisici disponibili come ottenuto da `java.lang.Runtime.availableProcessors()`.

[1] Orientativamente non più di una pagina di testo complessivo.

Specifiche

Per migliorare le prestazioni di una libreria Java mono-thread per il calcolo numerico si è deciso di migrare ad una piattaforma multiprocessore dotata di N_{CPU} processori[2] e di riscrivere in versione multi-thread un metodo `BinaryTreeAdder.computeOnerousSum(Node root)` molto oneroso. Tale metodo permette di calcolare la somma dei valori ottenuti eseguendo un calcolo computazionalmente molto oneroso su valori interi nei nodi di un albero binario di enormi dimensioni.

Scrivere una classe che realizzi l'intervento implementando l'unico metodo di questa interfaccia:

```
public interface BinaryTreeAdder {
  public int computeOnerousSum(Node root);
}
```

dove `Node` è una interfaccia utilizzata dalla libreria per modellare i nodi di un albero binario a partire dalla radice che `computeOnerousSum()` riceve come parametro:

```
public interface Node {
  Node getSx();   //null se non esiste figlio sinistro
  Node getDx();   //null se non esiste figlio destro
  int getValue(); //restituisce un intero associato al nodo
}
```

I valori interi sui cui effettuare i calcoli onerosi sono restituiti dal metodo `Node.getValue()`, mentre il calcolo in questione è modellato dal metodo `onerousFunction()` della interfaccia `OnerousProcessor` di cui `FakeProcessor` fornisce una semplice simulazione:

```
public interface OnerousProcessor {
  public int onerousFunction(int value);
}

public class FakeProcessor implements OnerousProcessor {

  public final static int MIN_COUNT = 1000;

  private int max;
  private java.util.Random random;

  public FakeProcessor(int max) {
    this.max = max;
    this.random = new java.util.Random();
  }

  public int onerousFunction(int value) {
     int r = this.random.nextInt(this.max);
     int n = Math.max(MIN_COUNT, r); // non meno di MIN_COUNT
     // useless but onerous
     for(int counter=0; counter<n; counter++) {
        r = this.random.nextInt(this.max);     // maschera casuale
```

[2]Ovvero il numero di pipeline hardware di esecuzione disponibili, su alcune architetture coincide con il numero di *hyperthread*.

```
        r = r ^ this.random.nextInt(this.max); // inverti i bit
        r = r ^ this.random.nextInt(this.max); // due volte
      }
      return value; // n.b. value risulta invariato
    }
}
```

Strutturazione dei Test

I test-case devono verificare il corretto funzionamento del `BinaryTreeAdder` in diversi scenari. L'insieme di test-case e di scenari da considerare deve essere definito cercando di raggiungere un ottimale compromesso tra le esigenze contrapposte che naturalmente nascono in fase di test. In particolare bisogna verificare per primi gli scenari più semplici, in quanto utili per rilevare in un contesto semplificato i più gravi errori di progettazione, e solo dopo quelli più rappresentativi, in quanto utili a validare la soluzione prodotta in scenari più simili a quelli in produzione. Allo studente viene lasciato il compito di delineare degli scenari di test idonei a testare il comportamento del proprio codice principale.

Sulla base dell'esperienza dei precedenti homework, articolare i test-case in tre passi:

set-up si prepara uno scenario di utilizzo del codice principale (consultare il significato dell'annotazione `@Before` di JUnit)

sollecitazione si invoca il codice principale sollecitando uno degli aspetti di cui si vuol verificare il funzionamento (consultare l'annotazione `@Test` di JUnit)

verifica si verifica che l'esecuzione sia andata come atteso, ad esempio controllando che lo stato raggiunto in seguito alla sollecitazione ed il risultato conseguito siano quelli attesi (vedere la famiglia di metodi `assert`*`XXX`* di JUnit)

Vale la pena di precisare che alcuni test-case possono creare (in genere nella fase di set-up) un albero binario ed un certo numero di thread visitatori con le caratteristiche idonee a verificare l'aspetto che quel test-case si prefigge di testare.

Misurazione dello Speed-Up

Utilizzando una piattaforma con più CPU,[3] sviluppare una applicazione per rilevare sperimentalmente ed automaticamente lo speed-up ottenibile con ciascuna delle soluzioni sviluppate per le precedenti parti del presente

[3] N.B. Se non si ha la possibilità di accedere ad una piattaforma con queste caratteristiche, è possibile chiedere assistenza al docente.

howework. Il calcolo dello speed-up, e le relative misurazioni, devono essere ottenute automaticamente dal codice consegnato che deve saper tener conto del numero di processori fisici effettivamente disponibili sulla piattaforma di esecuzione.

Per effettuare le misurazioni, si consiglia di premettere alla rilevazione dei tempi di esecuzione alcune esecuzioni "a vuoto" di *warm-up*: in questo modo si è sicuri che alcune ottimizzazioni a tempo dinamico della JVM siano già effettive all'atto delle misurazioni vere e proprie. È inoltre necessario fare esecuzioni con input di dimensioni abbastanza elevate da rendere alcuni costi che non dipendono dalla dimensione dell'input trascurabili, e quindi le misurazioni adatte a rilevare lo speed-up effettivo. (N.B.: in generale, misurazioni plausibili dello speed-up s sono tali che $1 \leq s < N_{CPU}$).

Suggerimento: Per fornire valori dello speed-up più affidabili, è anche possibile studiare l'andamento asintotico dello stesso all'aumentare delle dimensioni del problema in ingresso.

1 Occupazione di Memoria Illimitata (*vale l'80%*)

Fornire una soluzione al problema creando un numero di thread *visitatori* dell'ordine del numero delle CPU disponibili; ciascun thread visitatore insiste su un unico buffer condiviso di dimensione illimitata. Il buffer inizialmente contiene il solo nodo radice, ed i thread visitatori vi accedono nel seguente modo: estraggono ripetutamente e concorrentemente un nodo dalla testa del buffer ed inseriscono in coda gli eventuali figli, elaborano il valore del nodo, e procedono sino al verificarsi di una opportuna condizione di terminazione la cui definizione è lasciata al lavoro dello studente.

Si richiede esplicitamente di evitare: (*i*) ogni forma di attesa attiva ed ogni forma di interferenza (*ii*) l'elaborazione multipla degli stessi nodi dell'albero da parte di thread diversi (*iii*) la proliferazione di thread che sopravvivano all'esecuzione del metodo `BinaryTreeAdder.computeOnerousSum(Node root)` pur essendo stati creati al suo interno.

Implementare una possibile soluzione al problema proposto utilizzando i java thread ed in particolare le classi ed interfacce, come ad esempio `Executor`, `Future`, `Executors` ecc. ecc., appartenenti al package `java.util.concurrent` e che formano il *Lightweight Executable Framework*.

Viene esplicitamente richiesto di utilizzare le segnature indicate di sopra e di fornire un'implementazione dell'interfaccia `Node` che si limiti a rappresentare l'intero associato al nodo ed i due riferimenti ai nodi figli, senza possibilità di aggiungere altre informazioni di supporto ai visitatori.[4]

[4] Ad esempio non è consentito aggiungere un flag *già_visitato*.

2 Occupazione di Memoria Limitata (*10%*)

In molti casi reali non è consigliabile utilizzare un buffer di dimensione illimitata. Si pensi al caso di un *web crawler* multi-thread: i nodi dell'albero visitato rappresentano pagine web e gli archi rappresentano i link uscenti da ciascuna pagina. Il fattore di diramazione di questo albero è sufficientemente elevato che qualsiasi algoritmo di visita in ampiezza, come quello precedentemente ipotizzato in questo homework, impegna buffer di dimensioni molto rilevanti in poche iterazioni. Il problema si può riassumere affermando che per alberi di enormi, la *frontiera* della visita può essere di dimensioni tali da non rendere possibile e/o conveniente la sua completa memorizzazione.

Se semplicisticamente si rimpiazzasse il buffer di dimensione illimitata con uno di dimensione limitata, purtroppo ben presto i thread *visitatori* finirebbero per passare la maggior parte del proprio tempo in attesa che si liberi dello spazio nel buffer condiviso, sino allo stallo dovuto all'impossibilità di aggiungere ulteriori nodi di frontiera in un buffer già saturo.

Progettare e testare una soluzione che, senza compromettere il livello di parallelismo raggiunto dalla soluzione precedente, superi l'idea dell'unico buffer illimitato ed utilizzi invece buffer di dimensione limitata. Tale soluzione deve richiedere di memorizzare un numero di nodi al massimo $\mathcal{O}(N \cdot D)$ dove N il numero di thread visitatori e D è la profondità massima dell'albero visitato (se l'albero è bilanciato D è $\mathcal{O}(\log M)$ dove M è il numero di nodi nell'albero). Si ribadisce che, come per la prima parte di questo homework, anche in questo caso l'implementazione dell'interfaccia `Node` non può contenere informazioni addizionali utili solo ai visitatori.

Suggerimento: Risultano molto utili le tecniche di *work-stealing* realizzabili utilizzando `java.util.concurrent.BlockingDeque` e derivate. I thread visitatori effettuano visite in profondità e non appena esauriscono il lavoro da svolgere, provvedono a rifornirsi di nuovo lavoro "rubandolo" da uno degli altri visitatori, con le opportune precauzioni necessarie per evitare di far visitare più volte gli stessi nodi da parte di thread visitatori diversi.

3 Framework Fork/Join (*10%*)

Produrre una soluzione utilizzando il Java Fork/Join Framework e specificatamente dedicato alla decomposizione parallela di algoritmi.

4 Parallel Stream (*+10%*)

Ideare una soluzione utilizzando il supporto al processamento parallelo degli `java.util.stream.Stream` (disponibile da Java 8) definendo una implementazione di `java.util.Spliterator` specializzata per il tipo di struttura dati nella specifica del problema.