Assignment #01

Filippo Venturini

filippo.venturini8@studio.unibo.it

Analisi del problema

Il problema sottoposto presenta la necessità di modellare vari aspetti utilizzando tecniche di concorrenza e ognuno dei suddetti aspetti può essere modellato con diverse strategie.

A seguito di un'analisi sono state individuate le attività principali che il sistema dovrà svolgere:

- **Ricerca dei file sorgente** .java a partire dalla directory indicata, che dovranno poi essere messi a disposizione delle entità attive del sistema che li dovranno processare.
- **Elaborazione dei file** con conseguente determinazione di un risultato per file (il numero di righe), anch'essi dovranno essere a disposizione delle entità attive responsabili del calcolo finale.
- **Costruzione dinamica della classifica** dei top N files con righe maggiori e calcolo della distribuzione, il risultato dovrà essere disponibile per le entità responsabili della visualizzazione.
- **Visualizzazione dell'output**, tramite console e/o GUI.

Strategia Risolutiva

Entità attive

Per quanto riguarda le entità attive del sistema si è deciso di utilizzare il pattern **Master-Worker**, suddividendo quindi il carico di lavoro e le responsabilità tra un Master Thread e più Worker Thread.

Master Thread:

- Viene creato dal Main Thread appena il sistema viene eseguito.
- Ha il compito di ricercare tutti i file sorgente e di inserire il loro path in un buffer condiviso (che può essere interpretato come una Bag of Task).
- Una trovati tutti i file esso ha il compito di creare i Worker Thread.
- Infine mentre i Worker Thread sono in esecuzione, si occupa di prelevare i risultati appena prodotti, elaborarli e notificare la View che un nuovo risultato è disponibile per essere visualizzato.

Worker Thread:

- Viene creato dal Master Thread
- Ha il compito di prelevare un path dal buffer condiviso, contare il suo numero di righe e inserire il risultato nel buffer condiviso dei risultati.
- Continua la sua esecuzione finché il buffer dei file non è vuoto oppure finché non viene interrotto.

Event Dispatcher Thread:

- Utilizzato solo per il secondo punto, processa gli eventi prodotti dall'utente che interagisce con la GUI, in questo caso il click dei bottoni Start e Stop.

Entità Passive

Le principali entità passive del sistema sono i buffer condivisi che vengono utilizzati per l'assegnamento dei task, la condivisione dei risultati parziali e la condivisione dei risultati finali da mostrare in output.

Synchronized Queue:

- Concepita come una coda che permette l'accesso in lettura e in scrittura solamente ad un Thread per volta. Viene implementata come **monitor** utilizzando al proprio interno una **mutex** per realizzare **lock** e **signal**.
- Il sistema possiede due Synchronized Queue condivise: in una vengono inseriti i path di tutti i file da processare (**Bag of Tasks**), nell'altra vengono inseriti tutti i risultati processati dai Worker Thread (**Bag of Results**)
- Al suo interno contiene due metodi simili: **remove** e **blockingRemove**. Questo perché nel caso della **Bag of Tasks**, il Master Thread cerca tutti i file e li inserisce nel buffer, solamente dopo vengono eseguiti i Worker Thread che dovranno rimuovere i file, che saranno quindi da processare finché il buffer non sarà vuoto (utilizzando una semplice remove). Nel caso invece della **Bag of Results** il Master Thread rimuove i risultati dal buffer dinamicamente mentre essi vengono prodotti; quindi, in caso il buffer sia vuoto la remove deve essere bloccante e aspettare che vengano prodotti nuovi risultati tramite una **condition variable**.

Results:

- Struttura dati concepita per essere condivisa tra il Master Thread e il Thread responsabile della visualizzazione dei risultati
- Implementata come **monitor** (realizzato con una **mutex**)
- Al suo interno contiene una lista dei **risultati** calcolati, ordinata per numero di righe decrescente e contiene inoltre una mappa che rappresenta la **distribuzione** dei file negli intervalli richiesti.
- In questo modo appena un Worker Thread produce un risultato, il Master Thread lo preleva dalla Bag of Results e lo inserisce in questa struttura dati, che automaticamente ad ogni inserimento mantiene la lista ordinata e ricalcola la distribuzione.

Flag:

- Implementato come **monitor** utilizzando **synchronized** viene utilizzato per permettere all'utente di terminare l'esecuzione tramite i pulsanti presenti nella GUI.

Pattern Architetturali

Come pattern architetturale si è deciso di utilizzare il pattern **MVC** a cui viene integrato il pattern **Observer** per permettere una visualizzazione reattiva degli output.

Model:

- Contiene al suo interno la struttura dati per i risultati finali, il flag per la terminazione dell'esecuzione e la lista di observer che devono essere notificati quando nuovi risultati sono disponibili.

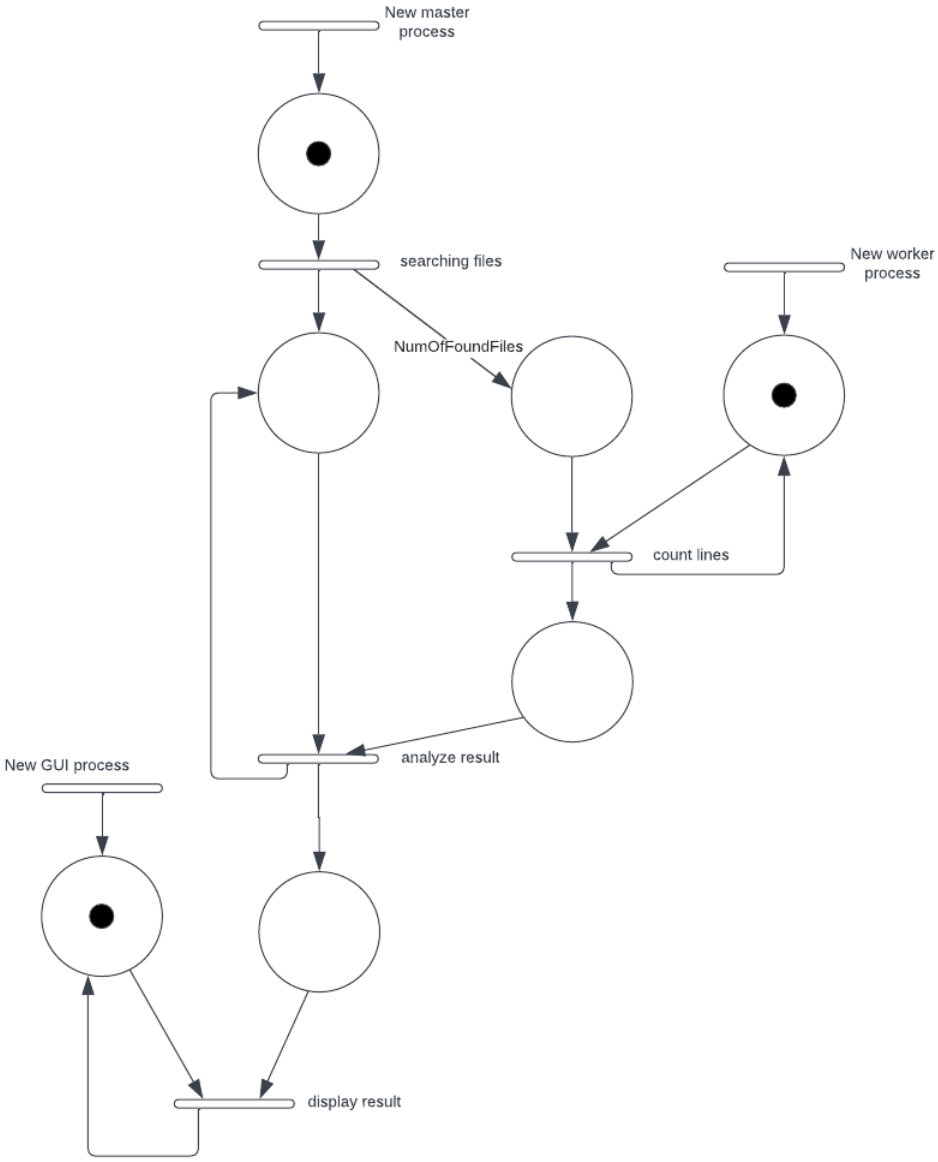
Controller:

- Starting point del sistema, crea il Master Thread e lo manda in esecuzione, fa da tramite per l'interazione tra Model e View (contiene quindi al suo interno tutti i metodi che le due parti invocano per interagire tra loro)

View:

- Si occupa della visualizzazione degli output, consiste quindi in un **ModelObserver** che ad ogni cambiamento nel model riceve una notifica e si occupa della visualizzazione. Possiede due implementazioni: una consiste in una **CLI** (primo punto dell'Assignment) mentre l'altra in una **GUI** (secondo punto dell'Assignment)
- Nel caso della **ConsoleView** essa reagisce solamente alla notifica della terminazione dell'esecuzione (visualizzando quindi solo l'output finale)
- Nel caso della **GUIView** essa reagisce alla notifica dell'aggiunta di ogni nuovo risultato, visualizzando dinamicamente gli aggiornamenti.

Descrizione del comportamento del sistema



Con la **Rete di Petri** sopra mostrata viene presentato il comportamento del sistema per quanto riguarda le dinamiche di concorrenza.

Sono presenti tre piazze con un token che rappresentano i tre processi principali: Master, Worker e GUI. Inizialmente solo il Master è abilitato ad eseguire la propria transizione che corrisponde alla ricerca dei files, appena li trova inserisce tanti token quanti sono i file nella piazza sottostante. In questo modo i worker presenti (per semplicità ne viene rappresentato solamene uno) sono abilitati a prelevare un token e a contare il numero di righe del file, generando un risultato parziale. A questo punto il Master può processare il risultato ottenuto permettendo alla GUI di visualizzare dinamicamente i nuovi aggiornamenti.

L'esecuzione termina quando terminano i NumOfFiles tokens, in questo modo il Worker non produce più risultati parziali, il Master di conseguenza non elabora risultati finali e la GUI non visualizza più nessun aggiornamento, questo corrisponde con lo stato finale della Rete di Petri.

Prove di performance

Sono state eseguite delle prove di performance rilevando le tempistiche di esecuzione del sistema sia variando il numero di Worker Thread che variando le dimensioni del progetto analizzato in input.

Durante questo processo è stato utilizzato un processore dotato di **4 core**.

Speedup e numero di Worker Thread

Inizialmente sono state testate le prestazioni del sistema con un differente numero di Worker Thread ed è stata validata l'assunzione per la quale il numero ottimale di Worker Thread è dato dal numero di core + 1 (Quindi nel nostro caso 5). Di seguito si riportano le misurazioni:

- **1 Worker:** 1746 ms
- **5 Worker:** 1260 ms
- **10 Worker:** 1492 ms

- **Speedup:** 1.386
- **Efficency:** 0.3465

Si noti infatti come le tempistiche minori si ottengono con esattamente 5 Worker Thread.

Si è inoltre calcolato il valore di Speedup ed Efficienza.

Scalabilità

Una volta individuato il numero ottimale di Worker Thread è stata verificata la scalabilità del sistema con tre differenti progetti, uno piccolo, uno medio e uno grande.

- **Progetto piccolo [100 sorgenti]:** 177 ms
- **Progetto medio [1569 sorgenti]:** 562 ms
- **Progetto piccolo [8017 sorgenti]:** 1260 ms

Le tempistiche di esecuzione hanno un andamento lineare rispetto all'aumentare dei file sorgenti da esaminare.

Identificazione di proprietà di correttezza e verifica

In generale sono state eseguite due attività di verifica.

Inizialmente è stato utilizzato TLA+, è stato realizzato un modello astratto che prevede due processi, Master e Worker, che agiscono su due strutture dati condivise: la coda dei file e la coda dei risultati. Tramite model checking è stato verificato che il sistema termina senza deadlock e mantiene verificate due invarianti che controllano la dimensione dei buffer condivisi.

Il sorgente relativo si trova nella sezione /src/tla+/

Infine, tramite JPF è stato collaudato il sistema in una versione semplificata, che simula la ricerca e l'elaborazione dei files, anch'esso termina senza rilevare errori o deadlock.

I sorgenti sono disponibili al seguente link:

https://github.com/FilippoVenturini8/JPFSourceTracker.git

Elaborato svolto con:

Nicolò Malucelli

nicolo.malucelli@studio.unibo.it