PCD-Assignment-01

Alessandro Mazzoli

alessandro.mazzoli9@studio.unibo.it

Luigi Borriello

luigi.borriello2@studio.unibo.it

Nicola Ferrarese

nicola.ferrarese@studio.unibo.it

13 aprile 2023

1 Analisi del Problema

1.1 Obiettivo

Nel progetto in oggetto si vuole realizzare un programma che, data una directory D all' interno del file system estragga metriche relative ai file con estensione `.java` all' interno del path considerato, incliso di sotto-cartelle.
In particolare, verranno computate le seguenti statistiche:

- Il numero di sorgenti `java` presenti all'interno della directory D.
- Gli N sorgenti con il numero maggiore di linee di codice.
- Un resoconto della distribuzione complessiva del numero di linee di codice dei sorgenti in esame, considerando un certo numero di intervalli NI e un numero massimo MAXL di linee di codice.

Il programma fornirà un'interaccia grafica per visualizzare queste statistiche, inoltre permetterà la configurazione di:

- Un Input frame per specificare i parametri (PATH, N, NI, MAXL)
- Dei pulsanti start/stop per avviare/fermare l'elaborazione
- Una sezione ove visualizzare interattivamente l'output aggiornato, mediante 2 frame:
  - Un frame relativo ai file con il maggior numero di linee di codice
  - Un frame relativo alla distribuzione

1.2 Concorrenza

Il focus dell'implementazione è stato rivolto in particolar modo sull'aspetto concorrente del programma, in particolare:

- Accesso ai file: al fine di poter essere analizzato, ogni file verrà aperto in sola lettura da un thread facente parte di un pool di ConsumerThread che concorrentemente analizzeranno i file sorgente;
  è pertanto necessario evitare conflitti e/o aperture multiple dello stesso file da parte di thread differenti.
- Strutture dati: si è deciso di utilizzare strutture dati condivise per tener traccia dell' avanzamento del programma, ondevitare conflitti l' accesso a tali strutture è governato da meccanismi di sincronizzazione che ne garantiscano il corretto utilizzo concorrente.
- Risorse di sistema: si è cercato di mantenere un bilanciamento che garantisse la corretta esecuzione del programma pur utilizzando in maniera efficiente le risorse del sistema su cui il programma viene eseguito.

2 Strategia Risolutiva

2.1 Architettura Concorrente

Per l' implementazione del progetto abbiamo scelto una variante dell' architettura Producer-Consumer, in particolare:

- 1 **Path Producer** il quale esplora la cartella scelta in input dall' utente, aggiungendo il path dei file aventi l'estensione `java` all'interno di una coda protetta da monitor, chiamata *PathQueue*.

- N **Path Consumer**, i quali accedono in maniera sincronizzata e concorrente alla *PathQueue*, estraendo un path alla volta.
  Per ogni percorso il relativo file verrà quindi aperto in sola lettura e verrà estratto il numero di linee di codice contenute nel file.
  A questo punto il **Path Consumer** assume il ruolo di **Statistic Producer**: Una volta estratto il numero di righe di codice viene creato un oggetto Statistic (tupla <Path, Integer>) e inserito nella coda *StatisticQueue*, sempre protetta da monitor.

- 1 **Statistic Consumer**, il quale estrae dalla *StatisticQueue* le informazioni per ogni singolo oggetto Statistic per poi aggiornare il Model che si occuperà di ordinare tutte le statistiche analizzate.

2.1.1 Gestione degli Agenti

Per la gestione e la coordinazione dei vari agenti descritti precedentemente, abbiamo realizzato una classe `AssignmentAlgorithm`. Questa classe, al suo interno incapsula:

- Lo stato e la configurazione dell'algoritmo

- La creazione e la terminazione dei vari agenti

- Una barriera per la coordinazione degli N Path Consumer

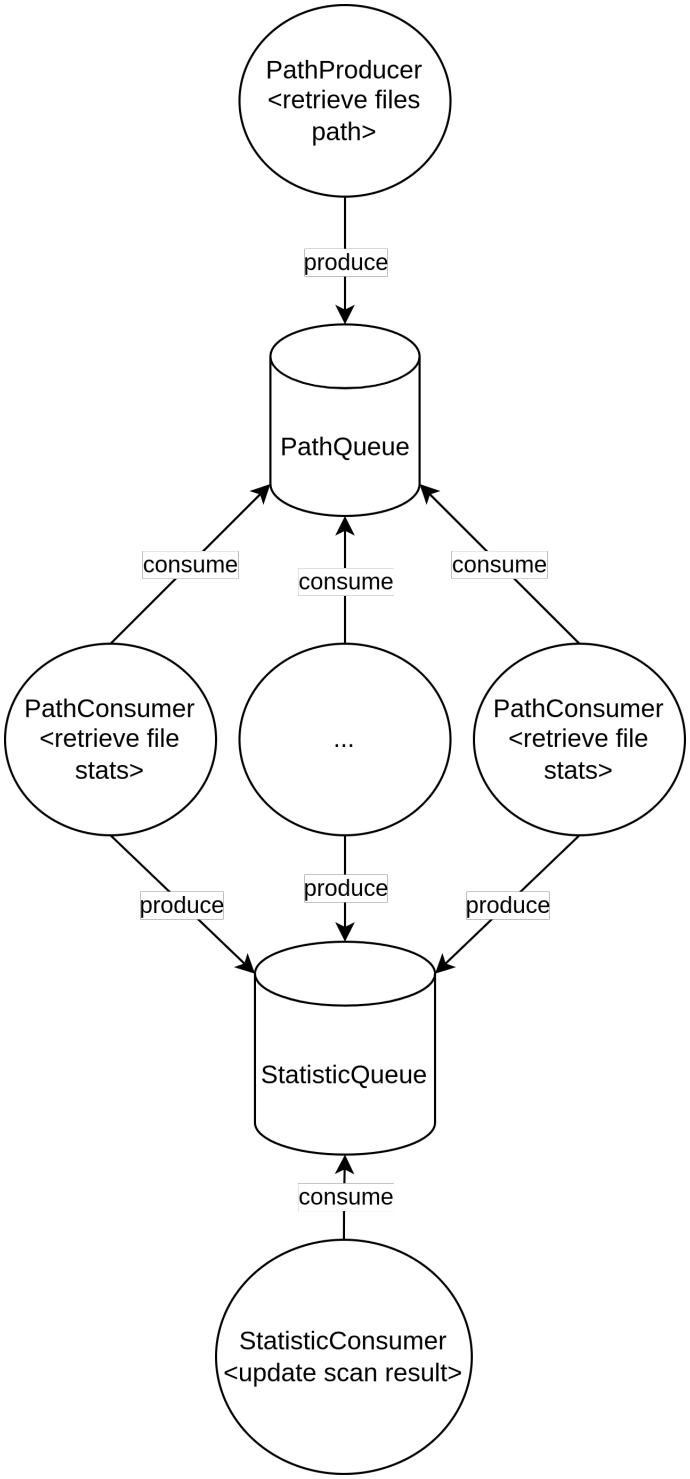

Figura 1: Architettura Producer-Consumer utilizzata

2.2 MVC

La gestione della GUI è stata implementata tramite il pattern MVC(Model-View-Controller) che permette alla View di ricevere in tempo reale gli aggiornamenti sullo stato della ricerca in modo da visualizzare i cambiamenti durante l'esecuzione ma rimanendo sempre reattiva agli input dell'utente.
Questo è stato ottenuto tramite il pattern Observer che ha reso il Model osservabile dal Controller, il quale invoca le funzioni di modifica sulla View.
Una volta lanciato il programma, l'utente potrà configurare i parametri tramite la GUI (`View`). Quest'ultima una volta ricevuto lo "start" input, inoltrerà i parametri al `Controller` che a sua volta, configurerà e creerà un istanza di `AssignmentAlgorithm`.

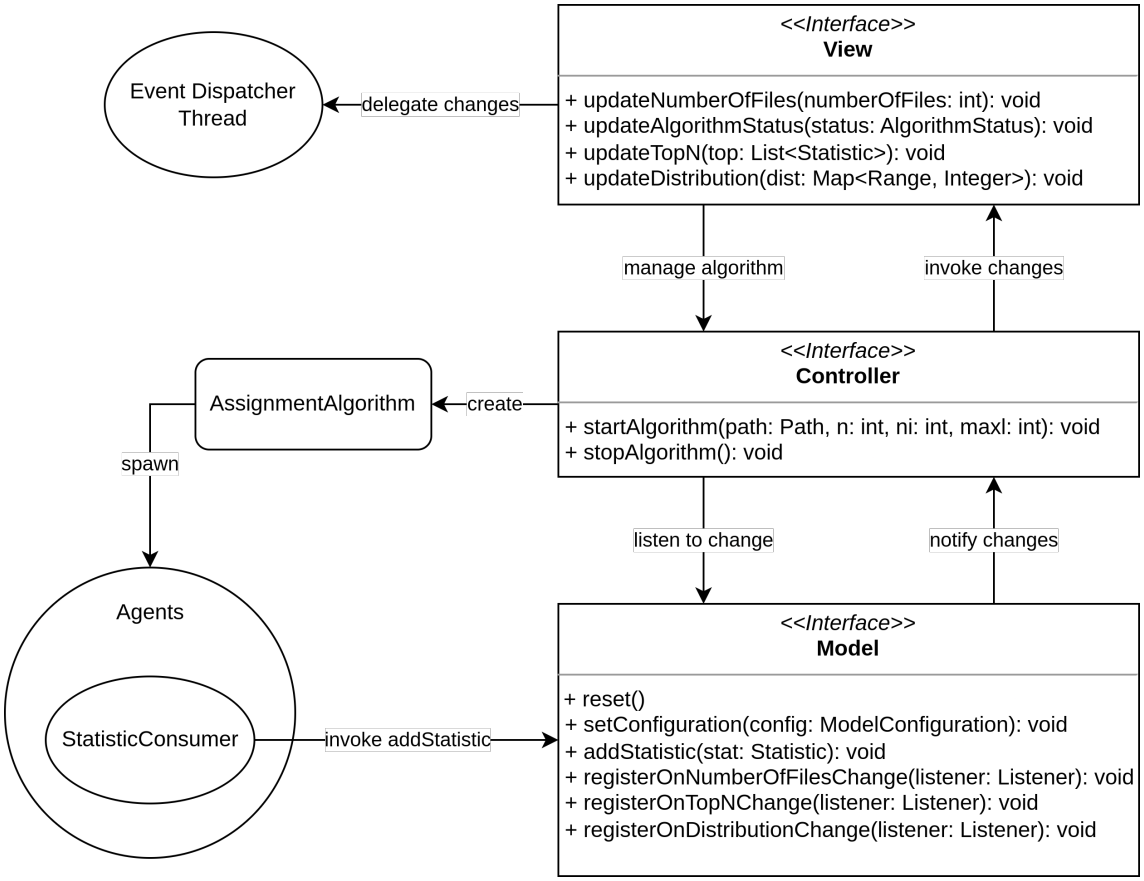

Figura 2: Gestione MVC

3 Comportamento del Sistema

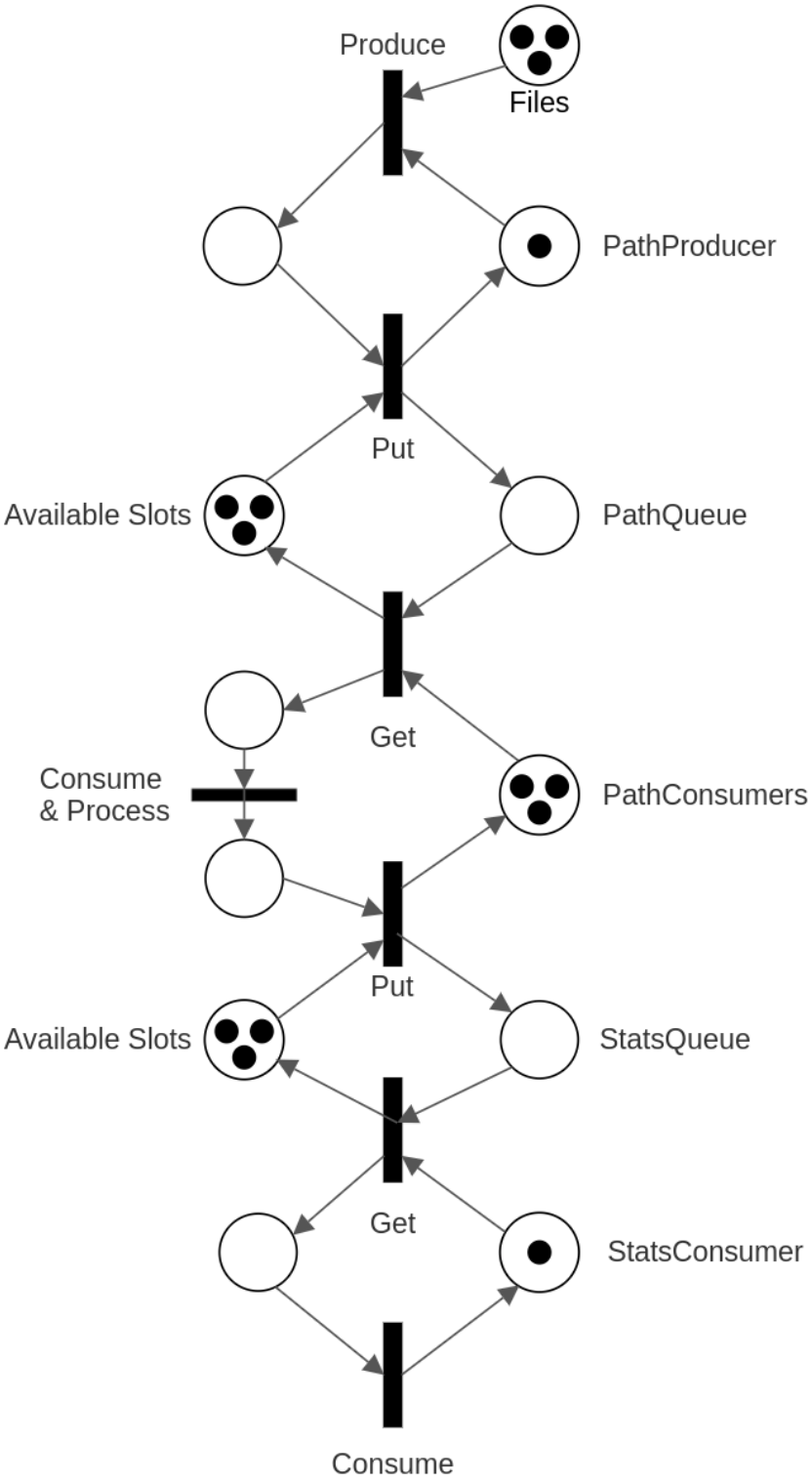

Figura 3: Petri-Net dell'architettura

4 Prove di Performance e considerazioni relative

5 Identificazione di proprietà di correttezza e verifica

5.1 JPF (Java Path Finder)

Per quanto riguarda il testing tramite JPF è stata implementata una versione ridotta dell'architettura che prevede:

- 1 Path Producer che produce semplicemente un numero limitato di elementi (gli elementi sono statici e non sono gli effettivi Path).
- 2 Path Consumer che una volta consumato l'elemento producono delle statistiche anch'esse statiche (senza andare a lavorare sul file system).
- 1 Stats Consumer che consuma semplicmente le statistiche ricevute.

5.2 Model Checking con TLA+

Considerate le seguenti variabili:

```
variables
    pathitems = <<"el1", "el2", "el3">>,
    pathqueue = <<>>,
    pathqueueclosed = FALSE,
    statsqueue = <<>>,
    statsqueueclosed = FALSE;
```

e i seguenti processi:

```
fair process pathproducer \in { "pathproducer" }
    ...
end process;

fair process pathconsumer \in { "pathconsumer1", "pathconsumer2" }
    ...
end process;

fair process statsconsumer \in { "statsconsumer" }
    ...
end process;
```

Sono state verificate le seguenti proprietà:

- Controlli sul Bound delle Queue:
    1. BoundedPathQueue: la coda dei path deve rispettare i vincoli di lunghezza.

    ```
    BoundedPathQueue ==
        Len(pathqueue) >= 0 /\ Len(pathqueue) <= MaxQueueSize
    ```

2. BoundedStatsQueue: la coda della statistiche deve rispettare i vincoli di lunghezza.

```
BoundedStatsQueue ==
    Len(statsqueue) >= 0 /\ Len(statsqueue) <=
   MaxQueueSize
```

- Controlli sulla terminazione degli agenti:
  1. PathProducerWillEnd: il path producer deve terminare.

```
PathProducerWillEnd == <>(pc["pathproducer"] = "Done")
```

  2. PathConsumersWillEnd: i path consumers devono terminare.

```
PathConsumersWillEnd ==
    <>(pc["pathconsumer1"] = "Done" /\ pc["pathconsumer2"]
    = "Done")
```

  3. StatsConsumerWillEnd: lo stats consumer deve terminare.

```
StatsConsumerWillEnd == <>(pc["statsconsumer"] = "Done")
```

  4. AllWillEnd: tutti gli agenti devono terminare.

```
AllWillEnd == <>(pc["pathproducer"] = "Done" /\
                    pc["pathconsumer1"] = "Done" /\
                    pc["pathconsumer2"] = "Done" /\
                    pc["statsconsumer"] = "Done")
```

- Controlli sulla chiusura delle code:
  1. PathQueueWillBeClosed: la coda dei path deve essere chiusa.

```
PathQueueWillBeClosed == <>pathqueueclosed
```

  2. StatsQueueWillBeClosed: la coda dei path deve essere chiusa.

```
StatsQueueWillBeClosed == <>statsqueueclosed
```

- Controlli sulle operazioni su code chiuse:
  1. NoPushOnPathQueueClosed: non ci devono essere operazioni di push sulla coda dei path una volta chiusa.

```
NoPushOnPathQueueClosed == [](pathqueueclosed ~> ~<>(pc["
   pathproducer"] = "put"))
```

  2. NoPushOnStatsQueueClosed: non ci devono essere operazioni di push sulla coda delle stats una volta chiusa.

```
NoPushOnStatsQueueClosed == [](statsqueueclosed ~> ~<>(pc[
   "pathconsumer1"] = "put" \/ pc["pathconsumer2"] = "put"
   ))
```

6 Prove di Performance e Considerazioni finali

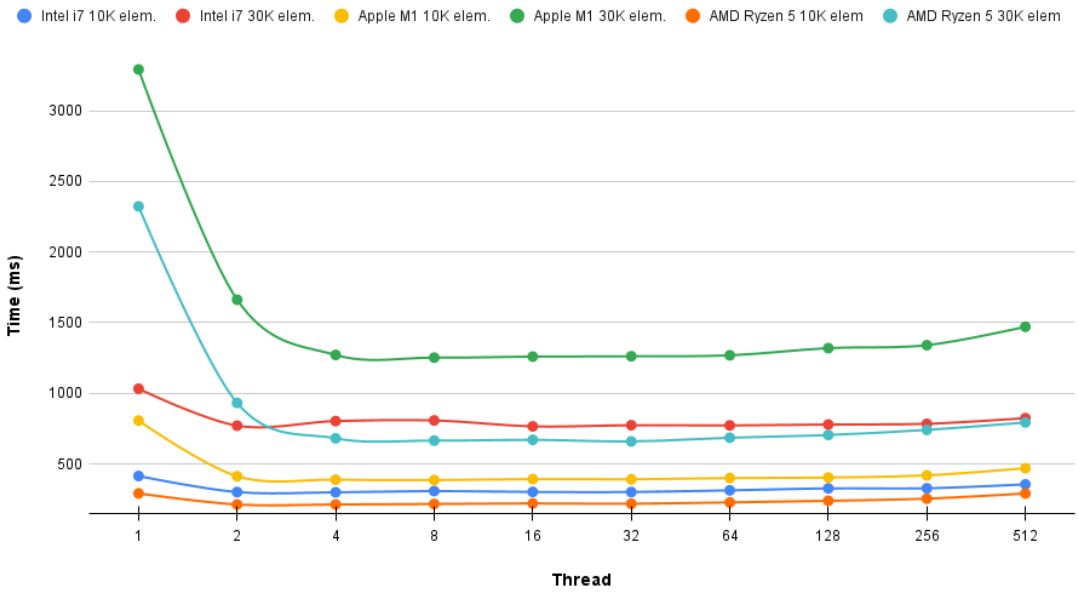

Figura 4: Benchmarking dell'algoritmo

Dall'analisi e test del progetto riteniamo di aver soddisfatto i requisiti richiesti e di aver prodotto un' architettura funzionale.
Tuttavia, siamo consapevoli del fatto che sia migliorabile, in quanto avremmo potuto parallelizzare ulteriormente anche le estremità del nostro algoritmo (abbiamo un solo `PathProducer` e un solo `StatsConsumer`).

Tali ottimizzazioni sarebbero state:

- L'implementazione di un `FolderProducer` avente il compito di scansionare la cartella di partenza ricercando solamente le sottocartelle e pushando i Path di tali folder in una `FolderQueue`. A questo punto più `FolderConsumers` avrebbero prelevato gradualmente i Path delle cartelle da questa coda e ricercato all'interno di esse i vari file .java pushandoli poi nella `PathQueue`.

- Parallelizzazione del calcolo delle statistiche in modo da avere più `StatsConsumers` in esecuzione, ognuno dei quali si calcola una propria porzione di statistiche basata solamente sui dati dei file ricevuti per poi effettuare un merge delle statistiche alla fine dell'esecuzione. Tuttavia questo non avrebbe permesso di ottenere i risultati se non a fine esecuzione.

Tuttavia queste modifiche avrebbero aumentato la complessità dell'architettura che sarebbe risultata più difficile da testare. Inoltre da veloci test effettuati non ci sembrava che tali modifiche avrebbero portato speed-up considerevoli in proporzione all'aumento della complessità dell'architettura.
Si è quindi optato per un architettura più semplice ma comunque efficace.