Programmazione Concorrente e Distribuita
Assignment 1

Matteo Ragazzini
Simone Romagnoli

A.A. 2020/2021

1 Introduzione

Nel presente documento si descrive la soluzione al problema del primo assignment del sopracitato corso realizzata dal nostro gruppo. Come linguaggio d'implementazione dell'assignment è stato utilizzato solamente *Java*, usufruendo della libreria `org.apache.pdfbox` fornitaci assieme alle specifiche; inoltre, sono stati utilizzati i tool *Java Path Finder* e *TLA+* per la verifica di correttezza delle soluzioni e per migliorare la progettazione del programma.

2 Analisi del problema

Il problema consiste nell'individuare le N parole più frequenti all'interno di un insieme D di documenti in formato pdf; va escluso dal conteggio un gruppo di parole F elencato in un file di testo. Il programma sfrutta la libreria `org.apache.pdfbox` per estrapolare testo grezzo dai documenti in formato pdf; dopodiché trasforma il testo in una lista di parole per riuscire a filtrarle (escludendo quelle da ignorare) e contarle meglio in seguito. Inoltre, l'intero procedimento deve essere implementato con architettura concorrente e deve poter essere bloccato in qualsiasi momento per poter poi riprendere in seguito. All'interno di questa procedura sono stati trovati quattro/cinque task principali:

- ***strip*** - corrisponde con l'azione di estrazione di testo grezzo dai file in formato pdf (così chiamata per via dell'oggetto `PDFTextStripper` che si occupa di questa procedura);
- ***split*** - la separazione delle parole del testo in base al carattere "spazio" (*ASCII* 32);
- ***filter*** - l'esclusione delle parole da ignorare ricevute come argomento del programma;
- ***count and order*** - il conteggio effettivo di ogni parola seguito dall'ordinamento decrescente delle frequenze.

Una volta identificati i task, si è passati all'analisi di eventuali dipendenze fra loro. In particolare vi è una dipendenza temporale precisa, che impone che i task vengano eseguiti nell'ordine sopra citato: i dati su cui ogni task lavora, corrispondono al risultato dell'elaborazione precedente, vedi fig. 1.

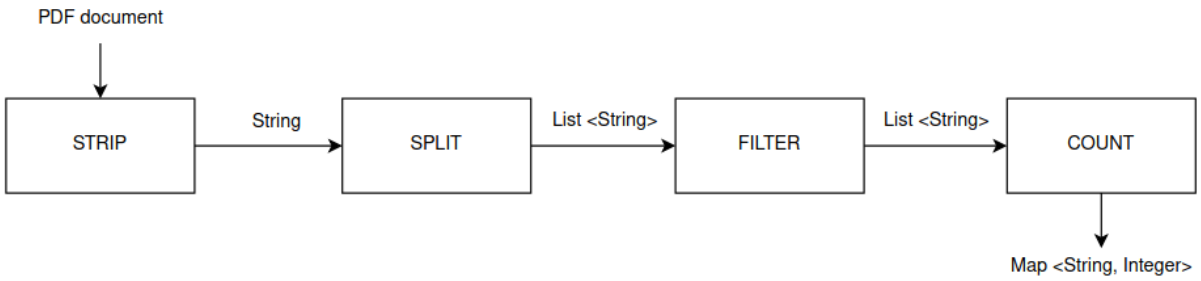

Figura 1: Dipendenze temporali e di risorsa dei task

Una volta individuati task e dipendenze, per comprendere al meglio le prestazioni da massimizzare, si è deciso di realizzare una versione sequenziale del programma.

Misurando i tempi dei singoli task, l'operazione più onerosa è risultata essere la prima (strip), ovvero quella che coinvolge più attività di input-output; in particolare, sono state effettuate più misurazioni al variare del numero di pdf, e ne è stata effettuata la media (in *ms*), come mostrato nella fig. 3.2.

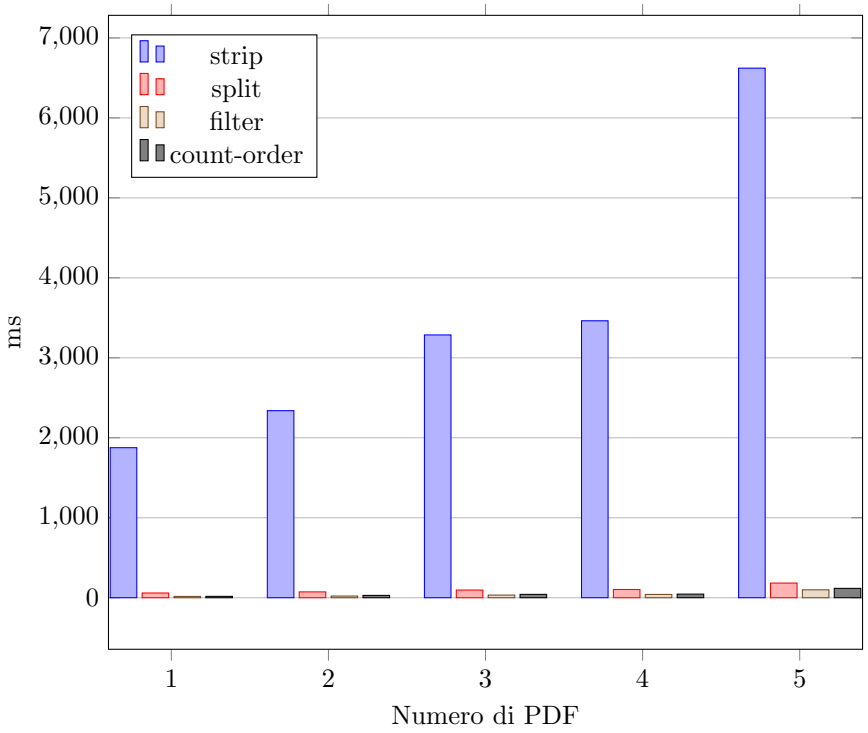

Figura 2: Misurazione dei tempi dei vari task nella versione sequenziale

3 Design della soluzione

Data la natura del problema, il tipo di parallelismo adatto si ritiene essere quello ad agenda. Questo tipo di parallelismo infatti prevede che ad ogni step vengano assegnati più worker: applicandolo al problema in analisi permette di suddividere il carico di lavoro tra i thread, i quali eseguiranno in maniera concorrente i task sopra citati. Essendo quindi un problema di *order-of-execution*, l'architettura concorrente più adatta risulta essere quella *master-worker*. L'idea è quella di un thread, detto *master*, che coordina un insieme di altri thread, detti *worker*, attraverso dei monitor.
Inizialmente, sono stati individuati due obiettivi principali per il design dell'applicazione:

- effettuare un buon bilanciamento del carico tra i worker;
- parallelizzare i task sopra citati, in particolare quello più oneroso (*strip*), per ottenere un buono speed up.

3.1 Design iniziale

Date le premesse, si è cercata una soluzione elegante che permettesse di distribuire equamente le pagine dei pdf tra i thread. La strategia di parallelizzazione adottata è stata il partizionamento a grana grossa: ogni thread si occupa di eseguire i task sopra citati per una partizione di pagine il più equa possibile; nel caso in cui un documento sia composto da 50 pagine, sfruttando 5 *worker thread*, il numero di pagine elaborate da un singolo *worker* è 10. In particolare, si è sfruttato un monitor chiamato `PdfMonitor` sul quale il *master* carica i documenti uno alla volta e li divide in modo da poter bilanciare il carico di lavoro dei *worker*; quest'ultimi ne estrapolano il testo prendendo il lock sul monitor per poi eseguire i restanti task sul testo grezzo in maniera concorrente. Inoltre, i *worker* usano un `OccurrencesMonitor` per aggiornare il numero di occorrenze di ogni parola ad ogni loro ciclo di esecuzione. La figura 3 rappresenta il flusso d'esecuzione appena descritto, nel caso in cui il numero di thread *worker* sia 5.

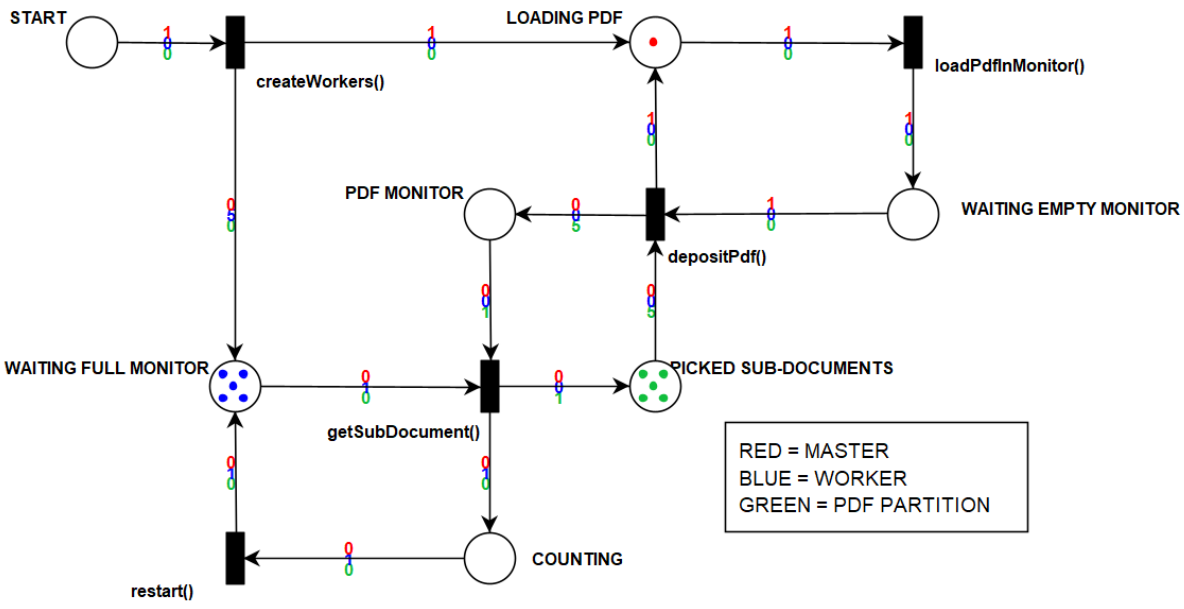

Figura 3: Rappresentazione del flusso di thread master e worker tramite una rete di Petri (design iniziale).

La soluzione ottenuta è risultata effettivamente un buon esempio di coordinazione e di distribuzione del carico di lavoro, tuttavia non ha raggiunto le prestazioni desiderate, mantenendo pressoché gli stessi tempi della versione sequenziale (come mostrato nel grafico: 3.2). Ciò è dovuto dal fatto che non è stato possibile parallelizzare lo *strip*, poiché più thread non possono operare sullo stesso file.

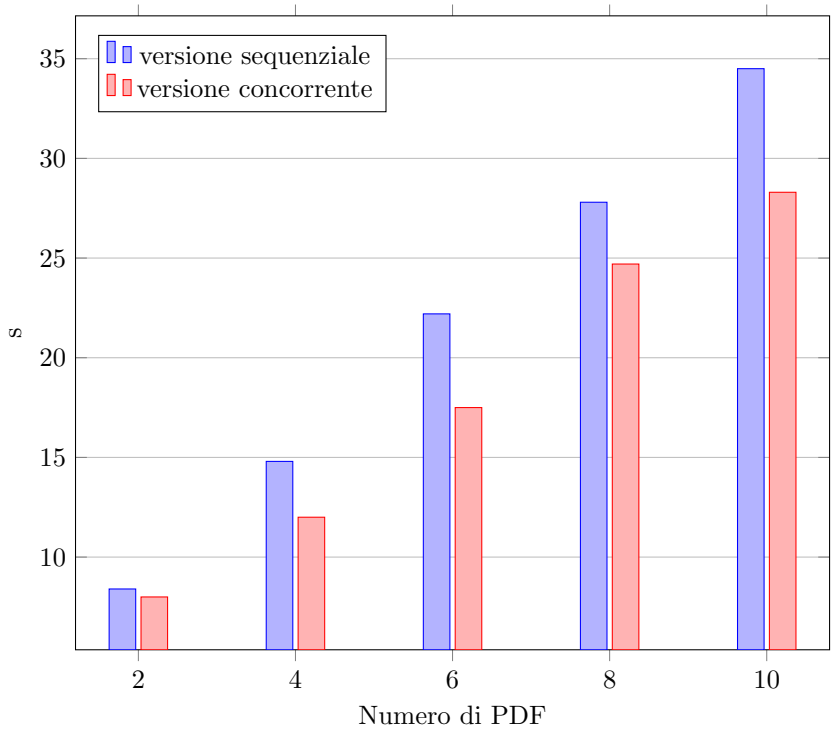

Figura 4: Confronto dei tempi della versione sequenziale con la prima versione parallela

Il difetto di tale versione del programma è che lo *strip* viene eseguito all'interno del `PdfMonitor` (come mostrato nel listato **??**): per poter migliorare questo aspetto è stato necessario spostare tale operazione. La libreria pdfBox dispone di una classe `Splitter` che permette di dividere un pdf in più sotto-documenti: una soluzione poteva essere la creazione di un nuovo file per ogni sotto-documento così da poter eseguire lo *strip* in modo concorrente, però si perderebbe molto tempo nella creazione dei file, non ottenendo poi il vantaggio cercato.

Coordinazione design iniziale

```
public class PdfMonitor {

   private boolean documentsFinished;
   private Queue<PDDocument> documents;

   public synchronized void setDocument(final PDDocument doc, final int workload, final boolean
       lastDocument) throws InterruptedException, IOException {
      while(!this.documents.isEmpty()) {
         wait();
      }
      this.documentsFinished = lastDocument;

      Splitter splitter = new Splitter();
      splitter.setSplitAtPage(workload);
      this.documents.addAll(splitter.split(doc));
      notifyAll();
   }

   public synchronized Optional<String> getText() throws InterruptedException, IOException {

      while(this.documents.isEmpty() && !documentsFinished) {
         wait();
      }

      if(!this.documents.isEmpty()) {
         PDDocument doc = documents.poll();
         PDFTextStripper stripper = new PDFTextStripper();
         ...
         if(this.documents.isEmpty()) {
            notifyAll();
         }
         return Optional.of(stripper.getText(doc));
      } else {
         return Optional.empty();
      }
   }
}
```

3.2 Design finale

Alla ricerca di migliori prestazioni, si è deciso di cambiare completamente il design del programma: piuttosto che dividere un documento tra i thread, si è pensato di assegnare l'esecuzione dei task per un intero pdf ad un singolo *worker*.
In questa architettura (che si ispira al paradigma *produttore-consumatori*), il `PdfMonitor` contiene una coda di documenti, che viene progressivamente riempita dal thread *master*, mentre i *worker* la consumano, come mostrato nella rete di Petri 5.

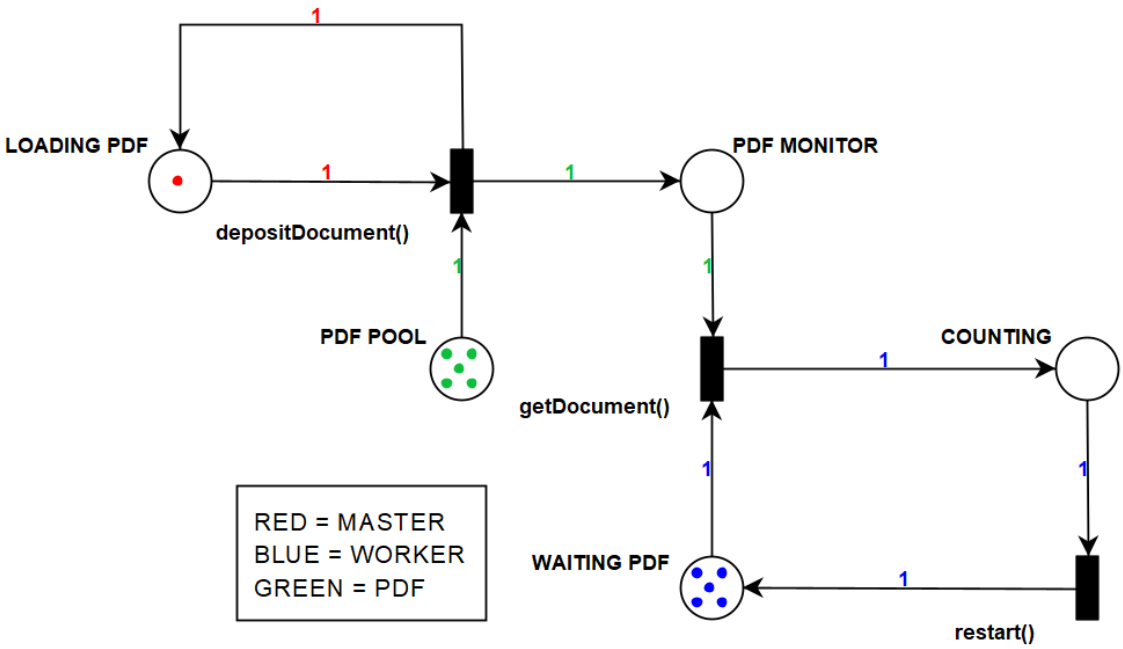

Figura 5: Rappresentazione del flusso di thread master e worker tramite una rete di Petri (design finale).

In questo modello, i *worker* riescono a prelevare un intero documento dal `PdfMonitor` per poi eseguire l'operazione di *strip* in seguito, parallelamente agli altri. Di fatti, si è ottenuto uno *speed-up* notevole rispetto alla prima versione del programma,

Coordinazione design finale

```
public class PdfMonitor {

   private static final int ZERO = 0;

   private Queue<PDDocument> documents;
   private Boolean documentsFinished = false;

   public synchronized void setDocuments(final PDDocument doc, Boolean documentsFinished) throws
        InterruptedException {
       documents.add(doc);
       this.documentsFinished = documentsFinished;
       notifyAll();
   }
```

```
    public synchronized Optional<PDDocument> getDocument() throws IOException,
        InterruptedException {

        while(this.documents.isEmpty() && !documentsFinished) {
            wait();
        }

        if(!documents.isEmpty()) {
            return Optional.of(documents.poll());
        } else {
            return Optional.empty();
        }
    }
}
}
```

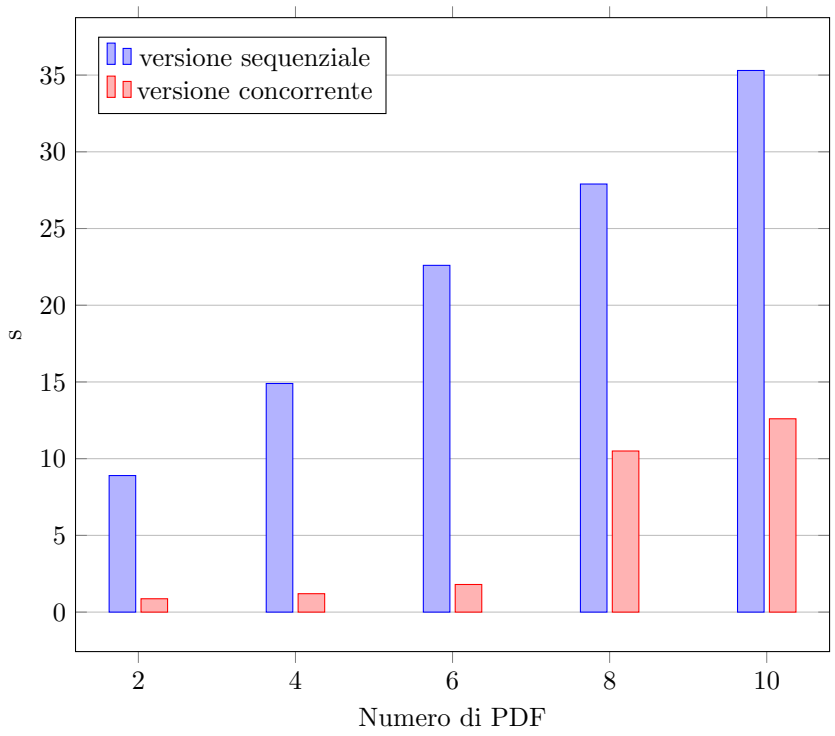

Figura 6: Confronto dei tempi della versione sequenziale con la seconda versione parallela

Sebbene tale implementazione guadagni in semplicità e prestazioni, non è perfetta dal punto di vista del bilanciamento del carico. Infatti, il caso pessimo per tale implementazione risulta essere quello in cui i documenti presentano una grande differenza nel numero delle pagine (nell'ordine delle centinaia), oppure quello in cui il numero di documenti da analizzare è minore del numero di thread *worker*; addirittura, nel caso limite di un solo documento non ci sarebbe alcun parallelismo.

4 Verifiche di correttezza

Verifiche con TLA+ e/o JPF.

5 Conclusioni

Nella realizzazione dell'elaborato sono state implementate più soluzioni, ognuna delle quali ha presentato vantaggi e svantaggi. Dopo aver effettuato diverse prove, si è concluso che il design che favorisce le prestazioni presenta maggiori vantaggi; in particolare, si ritiene che non esista una architettura che soddisfi ogni scenario d'esecuzione, ma che quella proposta sia una valida soluzione al problema. Il trade-off principale è stato tra il bilanciamento del carico e lo speed-up: solitamente, con una buona suddivisione del lavoro tra i thread si ottiene un miglioramento dei tempi d'esecuzione, assieme ad un miglior design dei modelli; tuttavia, questo elaborato ha dimostrato che non sempre tali aspetti siano collegati. Infatti, nel caso di attività onerose di input/output è difficile suddividerne l'esecuzione.