Progetto LSO: farm

Emiliano Sescu

Febbraio 2023

Contents

1 Architettura del programma

Il programma **farm** è composto da principalmente due processi:

- **MasterWorker**: processo multi-threaded composto da un thread *Master* e da n thread *Worker*. Il programma prende come argomenti una lista di file binari (trattando il suo contenento come una lista di numeri interi lunghi) ed un certo numero di argomenti opzionali (vedere la sezione legata agli argomenti).
  Il nome del generico file di input, dopo aver controllato che il file legato al nome sia un file regolare e che possieda l'estensione richiesta (preso .dat come riferimento, ma può essere cambiata), viene inviato ad uno dei thread *Worker* tramite una coda concorrente condivisa. Il generico thread Worker si occupa di leggere dal disco il contenuto dell'intero file il cui nome ha ricevuto in input, effettuare un calcolo sugli elementi, e inviare il risultato (unitamente al nome del file) al processo *Collector* tramite una connessione socket locale (vedere la sezione dedita a comunicazione tra processi).
  Il processo si dovrà anche occupare della gestione dei segnali (vedi sezione segnali).
- **Collector**: processo che attende il risultato dei vari calcoli dai *Worker* di *MasterWorker*, ed al termine stampa i valori ottenuti sullo standard output, ordinando la stampa sulla base del risultato in modo crescente. I due processi comunicano attraverso una connessione socket locale (vedere la sezione dedita a comunicazione tra processi).

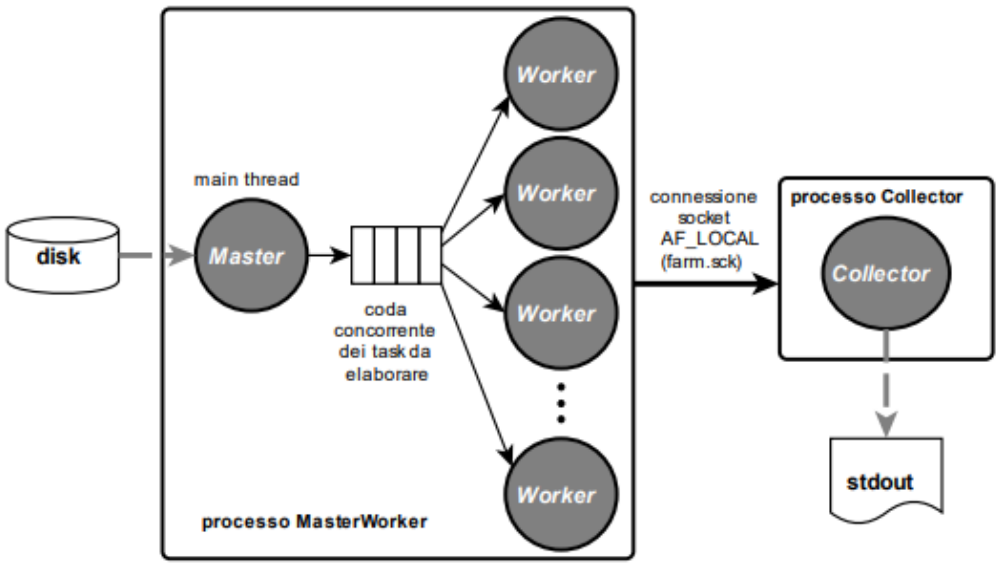

Figure 1: Rappresentazione grafica dell'architettura del sistema

2 Richieste implementative

In questa sezione tratterò più nel dettaglio quali sono state le richieste del progetto individuate dalla specifica.

2.1 Coda concorrente

Elemento che presenta concorrenza nel programma è la coda che mette in comunicazione *Master* e *Workers* all'interno del processo MasterWorker: essa conterrà i *tasks* che il *Master* inserirà nella coda, e che i *Workers* preleveranno per lavorarci sopra. Si può quindi notare il classico paradigma dove è presente un produttore e diversi consumatori, che viene poi implementato nel programma tramite variabili di condizione.

La coda può contenere al massimo qlen elementi.

2.2 Argomenti del programma

Gli argomenti che opzionalmente possono essere passati al processo MasterWorker sono i seguenti:

- **-n** <nthread>: specifica il numero di thread *Worker* del processo *MasterWorker* (default value: 4; max value: 256)
- **-q** <qlen>: lunghezza della coda concorrente tra il thread *Master* ed i thread *Worker* (default value: 8; max value: 512)
- **-d** <directory-name>: specifica una directory in cui sono contenuti file binari ed eventualmente altre directory contenente file binari; i file binari dovranno essere utilizzati come file di input per il calcolo
- **-t** <delay>: tempo in millisecondi che intercorre tra l'invio di due richieste successive ai thread Worker da parte del thread Master (valore di default 0; max value: 4096 ms)

2.3 Gestione segnali

Il processo *MasterWorker* deve gestire i segnali *SIGHUP*, *SIGINT*, *SIGQUIT*, *SIGTERM*, *SIGUSR1*. Alla ricezione del segnale *SIGUSR1* il processo MasterWorker notifica il processo *Collector* di stampare i risultati ricevuti sino a quel momento (sempre in modo ordinato), mentre alla ricezione degli altri segnali, il processo deve completare i task eventualmente presenti nella coda dei task da elaborare, non leggendo più eventuali altri file in input, e quindi terminare dopo aver atteso la terminazione del processo *Collector* ed effettuato la cancellazione del socket file.

Il processo *Collector* maschera tutti i segnali gestiti dal processo *MasterWorker*.

Il segnale *SIGPIPE* deve essere gestito opportunamente dai due processi.

2.4 Calcolo da effettuare

I files conterranno un numero N d'interi lunghi (*long*), dove per ogni files $result = \sum_{i=0}^{N} i * file[i]$.

Esempio: file "*mydir/fileX.dat*" con N=3; $result = \sum_{i=0}^{N} i * file[i] = (0 * 3 + 1 * 2 + 2 * 4) = 10$.
Collector stamperà:
10 *mydir/fileX.dat*

3 Scelte implementative

Le costanti (dimensione massima del path, nome del socket, ecc..) all'interno del programma si trovano nel file *farm.c*, e la loro modifica viene propagata per tutto il programma.

In questa sezione è presente la lista delle maggiori scelte implementative:

3.1 Comunicazione tra processi

I due processi comunicano attraverso una connessione socket *AF_LOCAL* (*AF_UNIX*), come da specifica. Il processo *Collector* fa da processo master per la connessione socket.

Vengono istaurate diverse connessioni (una per ogni *Worker*). Questo perché:

- Si vuole massimizzare l'utilizzo della select che si metterà in ascolto di molteplici connessioni da parte dei vari *Workers* (piuttosto che avere una sola connessione bidirezionale con il *Master*).
- Si vuole evitare di creare un collo di bottiglia nella pipe che metterebbe in comunicazione i Workers con il *Master*.

Queste connessioni saranno persistenti per tutto il periodo di vita dei *Workers*, dato che analizzando il caso pessimo di utilizzo del programma, si può notare che il *Worker* generico effettua più tasks durante tutto il periodo in cui è istanziato, quindi se fosse necessario aprire e chiudere la connessione verso il *Collector* ad ogni task si andrebbe a generare un overhead non indifferente .

Prima ancora dei *Workers*, sarà il *Master* ad effettuare la prima connessione con il Collector (in modo da mantenersi attivo un canale di comunicazione per quando dovrà notificare *Collector* in caso di segnale *SIGURS1*), permettendo anche una sincronizzazione tra la comunicazione dei due processi (cioè nel caso il *Master* provi a connettersi prima dell'avvio della select da parte del *Collector*, effettuerà un'attesa "attiva" (intervallata comunque da un timeout per evitare di saturare l'uso della CPU)).

Una volta che tutti i *Workers* si sono connessi, effettuano il loro ciclo di vita, e chiudono la connessione verso il *Collector*. Prima che il *Collector* proceda alla stampa dei risultato ordinati sullo standard output, dovrà attendere anche la chiusura della connessione col *Master*.

3.2 Gestione errori

Oltre ai classici controlli per errori legati a fallimenti di chiamate e parametri d'input, viene effettuato un controllo sul file da parte del Master prima di inserirlo nella coda concorrente:

se un file non è regolare e non ha estensione .dat (estensione .dat usata nei test cases, quindi presa come riferimento; è possibile senza troppe complicazioni andare ad aggiungere altre estensioni accettabili), esso viene scartato, avvisando tramite CUI l'evento.

Viene effettuato anche un semplice controllo di overflow quando vengono effettuati i calcoli.

3.3 Multithreading

Mentre il processo *MasterWorkers* è multi-threaded (un *Master* e tanti *Workers*), il processo *Collector* viene pensato come single-threaded, dato che l'accesso alla lista contenente i risultati avviene quasi esclusivamente in scrittura. Anche pensando ad una soluzione con tanti threads che si avviano quanti sono i *Workers* connessi, l'accesso alla lista avverrebbe solo in mutua esclusione, perdendo i vantaggi derivanti dalla concorrenza.

3.4 Argomenti del programma

La maggior parte delle informazioni legate agli argomenti del programma si può trovare nella sezione legata alle richieste implementative. Qui viene solo fornita un'indicazione per l'inserimento:

Gli argomenti del programma (ad eccezione dell'argomento directory identificato da -d) devono essere inseriti tra l'eseguibile e la lista di files [e/o directories]. Qualsiasi altro argomento (compresa ogni istanza di -d) seguente il primo elemento della lista di files [e/o directories] verrà trattato come nome di un file o, se presente -d, come nome di una directory.

3.5 Gestione segnali

Come richiesto da specifica, sarà il *Master* a gestire tutti i segnali in arrivo.

Mentre la maggior parte dei segnali vengono interamente gestiti dal *Master* (che terminerà il programma quando sente *SIGHUP*, *SIGINT*, *SIGQUIT* e *SIGTERM*), il segnale *SIGUSR1* viene comunque intercettato dal *Master*, ma viene anche comunicato al *Collector* (che andrà poi a stampare i risultati ottenuti fino ad ora) tramite il canale di comunicazione tra Master e Collector definito ad inizio programma.

Il segnale *SIGPIPE* verrà ignorato, ma gestito a livello di codice (non dal *Collector*, visto che non effettua *write()*) terminando il thread che si trova a scrivere in un canale privo di lettori, per evitare che il segnale termini l'intero processo.

3.6 Fine del programma

Una volta che il *Master* ha finito di iterare sui file di input (o sulla directory in input), si metterà in attesa della fine dei *Workers* (vedere come nella sezione legata alla coda concorrente). Prima di attendere la fine del processo Collector andrà a liberare la memoria allocata, per poi concludere il programma.

Tutti gli errori non distruttivi vengono gestiti con valore di ritorno piuttosto che tramite la funzione *exit()* (che viene comunque richiamata nel caso in cui un errore non rende più agibile l'intero programma (errori delle funzioni della famiglia *alloc* ed alcuni errori legati alla connessione tra processi)). Nei threads non viene chiamata la funzione *exit()* in modo da evitare l'arresto dell'intero programma nel caso in cui c'è stato un errore nel singolo thread (se passano *MAX_TIME_SECONDS* secondi e il *Master* è ancora in attesa sulla coda piena, allora chiude il suo metodo di riempimento coda, dato che ha intuito la chiusura per errori di tutti i threads).

3.7 Strutture dati

3.7.1 Coda concorrente

La coda concorrente viene utilizzata nel processo *MasterWorker*: essa è implementata come una coda circolare, e viene gestita tramite una *politica FIFO* (visto che non è una prerogativa quella di assegnare i task con fairness). La *politica FIFO* non solo garantisce un basso costo di overhead, ma viene in aiuto anche per quanto riguarda la notifica da parte del Master verso i suoi Worker di fine lavoro:

Una volta che il *Master* ha finito di iterare la lista di files, inserisce una stringa speciale nella lista (*"EOS"*: *End-Of-Stream*) per indicare la fine dei lavori. Quando l'i-esimo *Worker* si prepara ad estrarre il file, ma si accorge che si tratta appunto della stringa speciale, non la estrae e termina la sua vita. Il *Master* aspetterà la fine di tutti i *Workers* prima di andare avanti con la terminazione.

La coda concorrente conterrà quindi solamente files regolari e la stringa speciale.

3.7.2 Lista concatenata unidirezionale

La lista concatenata unidirezionale è presente nel processo *Collector*, e va a collezionare i calcoli fatti che riceve dai vari *Workers*. Essa è costantemente ordinata tramite *Insertion Sort* (utilizzando il risultato del calcolo come criterio d'ordinamento), rendendo eventuali richieste di stampa della lista più veloci.

La lista non prevede gestione della concorrenza, visto che è presente un solo writer alla volta: Collector (processo single-threaded) scriverà un elemento alla volta nella lista ogni volta che ne riceve uno. Nel caso si presentasse una richiesta di stampa improvvisa, il processo ferma momentaneamente le scritture nella lista e la stampa nella CUI.

3.7.3 Array dinamico

Array dinamico (la cui dimensione raddoppia ogni volta che si tenta di inserire un nuovo elemento quando l'array è pieno) in cui vengono inseriti tutti i nomi delle directories che vengono indicati tra i primi argomenti (*-d*, insieme a *-n*, *-q*, *-t* (vedi direttiva legata agli argomenti del programma).

Quando il programma effettua il parsing dei primi argomenti il *Master* non ha ancora avviato la coda concorrente, quindi è necessaria una struttura dati che ci permette di memorizzare il nome delle directories contenute nei primi argomenti. I primi files che verranno inseriti nella coda concorrente saranno quelli presenti nelle directories contenute all'interno dell'array dinamico.

4 Tests

[Compilare il progretto tramite il comando *make* prima di effettuare i tests tramite *make test*]

Oltre ai 5 tests, ne sono stati aggiunti 2 ulteriori per verificare le seguenti funzionalità:

- **test 6**: test che, dopo due secondi dall'avvio del programma, invia il segnale SIGUSR1. In questo test viene anche provata la funzione di aumento dimensione dell'array dinamico, visto che vengono passate 3 directory tra i primi argomenti del programma (solo una delle tre directory è effettivamente una directory, le altre restituiranno errore)).

- **test 7**: test che prova la chiusura automatica del Master in caso di questo particolare caso:

  In caso tutti i *Workers* escano a causa di un errore tramite *return* (come già detto in precedenza, nei *Workers* non vengono richiamati *exit()*), ma il *Master* nel suo ciclo di vita non è ancora arrivato nel momento in cui si mette in attesa dei *Workers* terminati, è possibile che il *Master* si ritrovi in attesa di una coda piena con nessun *Worker* che preleverà ulteriori stringhe. Per questo il *Master*, dopo essere rimasto in attesa sulla coda piena per *MAX_TIME_SECONDS* secondi, interrompe questa attesa, chiude la connessione col *Collector*, e si mette in attesa della fine del *Collector* (che dovrà comunque stampare ciò che ha ottenuto fino a quel momento).

  Per questo tipo di test viene utilizzato l'eseguibile *brokenfarm* (ottenibile tramite il comando *make brokenfarm*), dove a tempo di compilazione è stato aggiunto *RETURN_AFTER_ONE_TASK* tramite flag del preprocessore *-D*, in modo da interrompere l'esecuzione dei Workers dopo l'esecuzione di un singolo task (errore creato artificialmente per testare il comportamento del Master (riga 105 del file *src/worker.c*)).

Una volta terminato con i tests è possibile richiamare il comando *make cleantests* per andare ad eliminare tutti i files generati con *make test*, oppure è possibile chiamare il comando *make cleanall* per andare ad eliminare anche tutte le librerie, tutti i files oggetto, e tutti gli eseguibili ottenuti tramite *make*.