Wator

Un simulatore distribuito di un modello biologico

Leonardo Cariaggi
Corso A
Matricola 503140
A.A. 2014/2015

Contenuti

1. Scelte progettuali

1.1 Strutture dati principali:

- **struct neigh (Neighbor)**

  Struttura di supporto alla memorizzazione delle celle adiacenti a quella in cui è applicata una delle quattro regole che descrivono il comportamento degli squali e dei pesci. La struttura in questione tiene traccia delle coordinate di una cella all'interno della matrice e del contenuto della cella stessa (data la rotondità del pianeta, per stabilire le effettive coordinate di una cella è stata definita una funzione apposita che simula l'operazione di modulo dell'aritmetica). L'algoritmo utilizzato nell'applicazione delle regole consiste principalmente nella creazione di un array di Neighbor di dimensione 4, il quale sarà processato più volte per stabilire di quale tipo sono le celle adiacenti (utilizzando altri array di Neigbor come appoggio). Lo scopo è quello di distinguere i casi un cui non è possibile applicare una certa regola dai casi nei quali invece è permesso (e in tal caso la scelta della cella è casuale).

- **struct slice (Slice)**

  Struttura che rappresenta una porzione di pianeta. Essa è caratterizzata da un'origine (x,y) e dalle sue dimensioni (larghezza e altezza). Durante il processo di generazione delle porzioni, il pianeta è suddiviso in più parti, ognuna delle quali possiede delle dimensioni stabilite in maniera equa in base al numero di thread worker che dovranno processarle. Nei casi particolari in cui non si riesca a generare delle porzioni “complete”, le dimensioni saranno fissate soltanto in base

allo spazio rimanente (ci saranno eventualmente porzioni più piccole del normale).

- **struct slicelistitem (SliceListItem)**

  Struttura con la quale si identifica un elemento della lista di porzioni di pianeta condivisa tra gli worker. La lista è una classica linked-list monodirezionale, realizzata con la canonica struttura comprendente elemento e puntatore al prossimo elemento. Gli elementi della lista sono di tipo Slice.

1.2 Algortimi fondamentali:

- **Aggiornamento singolo per ogni pesce/squalo**

  Per garantire che in un singolo chronon le regole relative agli squali e ai pesci siano applicate non più di una volta per ogni abitante di wator si utilizza una matrice di appoggio. Essa è inizializzata *ogni volta* come una copia identica del pianeta al momento che precede l'applicazione delle regole. Per stabilire quindi se una cella non è già stata processata durante il chronon corrente, si controlla che essa sia uguale alla cella contenuta, nella stessa coordinata, nella “vecchia” matrice. La correttezza di questa scelta deriva dall'ordine e dal modo in cui viene scandita la matrice durante la simulazione: se una cella, per qualche motivo, dovesse essere visitata più di una volta, allora tale controllo non sarebbe più sufficiente a stabilire la condizione di aggiornamento singolo. É infatti possibile che dopo un'iterazione una cella abbia lo stesso contenuto di quello che aveva in precedenza, ma ciò non significa che tale cella non sia stata processata.

- **Accesso in mutua esclusione alla coda condivisa**

  Per l'accesso in mutua esclusione alla coda condivisa dal

thread *dispatcher* e dai thread *worker* si fa uso di una variabile di tipo *pthread_mutex_t,* che funge da lock. Al momento della generazione dei task, essa è acquisita da *dispatcher* e rilasciata dallo stesso al termine del compito. Quando invece è necessario prelevare un compito dalla coda (soltanto dopo che essa è stata riempita) da parte di un *worker*, esso acquisisce la lock, estrae il pezzo e immediatamente rilascia la lock.

- **Accesso in mutua esclusione al pianeta**

  Per quanto riguarda l'accesso al pianeta, è necessario distinguere due casi:

  1. *La cella a cui devo accedere si trova nella parte centrale della porzione di pianeta*
  2. *La cella a cui devo accedere si trova in prossimità di un bordo della porzione o a distanza 1 da uno dei bordi*

  Nel primo caso non c'è alcun tipo di problema in quanto non è possibile che una cella posizionata nella parte centrale di una porzione sia accedibile da più di un thread *worker*. Nel secondo caso invece, data la politica con cui si visitano le celle all'interno delle regole, può accadere che ci siano conflitti tra più thread *worker*. Per questo motivo si fa uso di una lock (borderMutex) per evitare tali situazioni. Quando a un *worker*, dunque, è affidato il compito di processare una cella con tali caratteristiche, esso acquisisce la suddetta lock prima dello svolgimento del lavoro e la rilascia subito dopo.

2. Strutturazione del codice

Il progetto è composto dai seguenti file:

- **wator.h** : Contiene la definizione di alcune strutture dati

utilizzate nel programma e la specifica di alcune funzioni usate per operare su di esse.

- **createWator.c** : Contiene l'implementazione delle funzioni new_wator e free_wator presenti in *wator.h* .

- **rulez.c** : Contiene l'implementazione delle funzioni che rappresentano le regole che devono essere applicate agli squali e ai pesci, di shark_count, fish_count e update_wator.

- **wator.c** : Contiene l'implementazione di tutte le altre funzioni specificate in *wator.h* .

- **myFunctions.h** : Contiene la definizione delle strutture dati Neighbor, Slice e SliceListItem e la specifica di funzioni aggiuntive utilizzate nel progetto.

- **myFunctions.c** : Contiene l'implementazione di tutte le funzioni presenti nell'header file *myFunctions.h* .

- **Wator.c** : Contiene il codice eseguito dal processo wator e dai tutti i thread da lui creati (*dispatcher*, *collector* e *workers*).

- **visualizer.c** : Contiene il codice eseguito dal processo visualizer (generato tramite fork dal processo wator).

- **watorscript** : script bash utilizzato per controllare la validità (anche in termini di strutturazione) di un file che rappresenta un pianeta.

2.1 Librerie

La libreria utilizzata nel progetto è libWator.a, che contiene i moduli oggetto nei quali sono implementate tutte le funzioni specificate in *wator.h* e in *myFunctions.h* .

3. Struttura dei programmi

Wator.c

Nella parte iniziale si registrano gli handler per la gestione dei segnali SIGUSR1, SIGINT e SIGTERM e poi si procede con l'analisi degli argomenti passati al programma in modo da inizializzare tutti i parametri della simulazione (per questo scopo è stata utilizzata la funzione **getopt** definita in getopt.h). Dopo aver deciso la giusta dimensione da assegnare alle porzioni di pianeta, avviene la creazione del processo visualizer tramite una **fork()**.
Fatto ciò si creano tutti i thread *worker*, il thread *dispatcher* e il thread *collector.*
Infine wator si mette in attesa dei segnali SIGUSR1, SIGINT e SIGTERM in maniera passiva tramite una *sigsuspend*.

visualizer.c

Il processo visualizer inizia la sua esecuzione creando la socket con la quale comunica con il thread *collector* e stabilendo l'output delle sue future stampe (file o **stdout**).
Il ciclo di vita di visualizer si riassume in una continua attesa di connessioni sulla suddetta socket (visual.sck) e, quando ciò avviene, esso procede alla stampa del pianeta nella destinazione prescelta.

watorscript

Watorscript è uno scipt bash di supporto che serve a verificare che un certo file che rappresenta un pianeta sia ben formattato.
Il controllo comincia con l'analisi delle opzioni passate da riga di comando: in particolare si controlla che sia presente l'unica opzione obbligatoria e si provvede a eseguire diverse azioni a seconda delle altre opzioni.
Il passo successivo consiste nel leggere il file da controllare carattere per carattere, ponendo particolare attenzione a quante celle sono presenti e a come sono distribuite.

4. Interazione fra processi

L'esecuzione del programma inizia innanzitutto con la creazione da parte di wator di tutti i thread necessari e la conseguente attesa passiva dei segnali.

La quasi totalità del lavoro è poi svolta dagli altri thread. Una normale iterazione è descritta dai seguenti passi:

1. Il thread *dispatcher* copia il contenuto del pianeta attuale nella matrice di appoggio utile a evitare l'applicazione multipla di regole su un abitante del pianeta wator. Fatto ciò procede con la creazione delle porzioni di pianeta riempiendo (in mutua esclusione) la lista condivisa con gli *workers* e al termine si sospende sulla variabile di condizione *readytoproduceCond* (non prima di aver chiamato una **broadcast** sulla variabile di condizione *emptyqueueCond* per risvegliare gli *workers* in attesa di lavoro).

2. Il thread *worker*, una volta che la lista è stata riempita, può prelevare (ancora in mutua esclusione) una porzione dalla lista e cominciare a lavorare su di essa (facendo attenzione ai casi particolari descritti tra gli algoritmi principali). Terminato il lavoro, esso incrementa in mutua esclusione una variabile

globale che indica il numero di porzioni finora processate da tutti i thread *worker* e cerca poi di prelevare un altro task (nel caso in cui non ve ne siano più, si sospende sulla CV *emptyqueueCond*). Proprio da quel contatore, il thread capisce se quella era l'ultima porzione che necessitava di essere processata, e in tal caso provvede ad avvertire il thread *collector* tramite una **signal** sulla variabile di condizione *readytosendCond*. Il comportamento è ciclico ed è lo stesso per ognuno degli *worker*.

3. *collector* quindi, una volta pronto per inviare il pianeta al processo visualizer, controlla se effettivamente è il momento di stampare (in questa sede si è deciso che visualizer stampi ogni volta che riceve i dati dalla socket) e in tale caso apre la connessione con visualizer, invia i dati tramite il protocollo descritto in seguito e richiude la connessione. Tale decisione è presa in base al fatto che siano trascorse o meno *visualcycle* iterazioni o al fatto che *collector* abbia ricevuto un segnale SIGUSR2 da wator (come conseguenza della ricezione di un SIGINT/SIGTERM da parte di wator stesso, che indica una terminazione forzata).
   Una volta che i dati sono stati inviati, collector avverte dispatcher che una nuova iterazione può cominciare tramite una signal sulla variabile di condizione *readytoproduceCond*. Come ultima cosa, *collector* si rimette in attesa di una signal da parte di un *worker* sulla CV *readytosendCond*.
   Il comportamento di collector è ciclico e termina quando si raggiunge la fine della simulazione (in tal caso *collector* invia un segnale SIGTERM a wator) o quando wator decide che è il momento di terminare (o meglio, quando a wator viene notificato un SIGTERM).

L'interazione tra i processi è descritta in maniera più concreta e sintetica dalla figura (4.1.1)

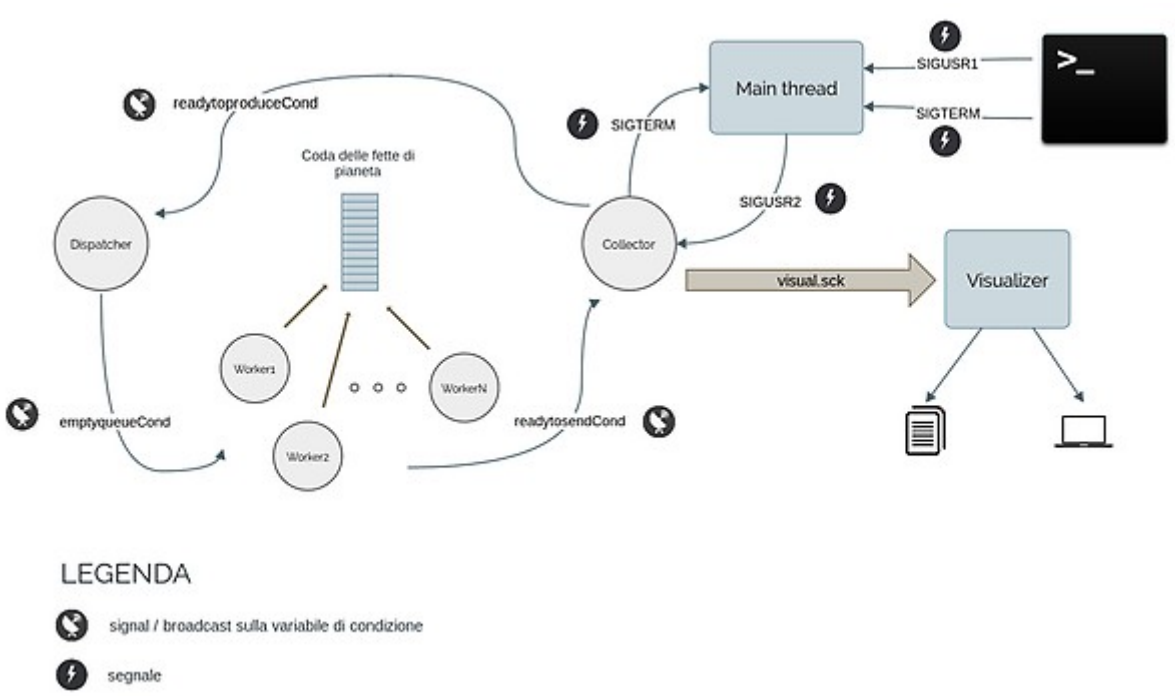

Figura (4.1.1)

Wator, infine, può anche ricevere dei segnali SIGUSR1, con i quali si intende effettuare un checkpoint del pianeta. In quel momento, infatti, wator provvede a stampare il pianeta sul file wator.check salvando così lo stato corrente della simulazione.
Il termine della simulazione invece è raggiunto al momento della ricezione di un segnale SIGTERM da parte di wator: in quell'occasione esso provvede ad inviare un segnale SIGUSR2 (le cui conseguenze sono descritte sopra) a collector e subito dopo attende che il processo visualizer termini, sospendendosi con una **waitpid**. L'intero processo può finalmente terminare.

5. Protocollo di comunicazione *collector*-visualizer

Negli istanti in cui è prevista la stampa del pianeta da parte di visualizer, *collector* comunica con quest'ultimo attraverso la socket *visual.sck,* provvedendo ad inviare i dati.

Il protcollo di comunicazione tra le due parti prevede l'invio da parte di *collector* dei due interi che rappresentano le dimensioni del pianeta (righe e colonne nell'ordine) sottoforma di **char*** e subito dopo l'invio di tutto il pianeta una riga alla volta. In totale dunque collector invia (*nrow* + 2) messaggi. Il primo dato ha anche la funzione di rendere consapevole visualizer di quanti messaggi esso si debba aspettare dopo i primi due.
Nel caso particolare in cui si debba avvertire quest'ultimo che i dati ricevuti sono quelli che pongono fine alla simulazione, la strategia adottata consiste nell'inserire il carattere speciale *'s'* nel messaggio che identifica il numero di righe da cui è composto il pianeta. In tale occasione, visualizer provvede comunque alla normale stampa ma invece di rimettersi in attesa di nuove connessioni sulla socket esce normalmente (il processo wator, ritornando dalla chiamata **waitpid** su cui si era sospeso a causa di un segnale SIGINT/SIGTERM, può dunque terminare).