Wator

Francesco Cariaggi

Matricola: 503146

Corso A

A.A. 2014-2015

Indice

1 Scelte progettuali

1.1 Strutture dati principali

Nella fase di analisi (che precede quella dell'applicazione delle regole) di una generica cella del pianeta, il contenuto delle celle adiacenti ad essa viene memorizzato all'interno di un array di dimensione fissa (4 posizioni). Ogni pesce o squalo infatti può, come da specifica, solamente "ispezionare" le celle che si trovano sopra, sotto, a sinistra o a destra di esso. La convenzione adottata per la memorizzazione del contenuto delle celle nell'array è la seguente:

- array[0] contiene il contenuto della cella che si trova sopra quella attualmente in esame.
- array[1] contiene il contenuto della cella che si trova sotto quella attualmente in esame.
- array[2] contiene il contenuto della cella che si trova a sinistra di quella attualmente in esame.
- array[3] contiene il contenuto della cella che si trova a destra di quella attualmente in esame.

Prima dell'applicazione delle regole su di una cella è necessario sincerarsi che questa non contenga pesci/squali appena nati (su di essi le regole verranno applicate solo al chronon successivo). La struttura dati di supporto che mi permette di rispettare tale regola è una matrice (delle stesse dimensioni di quella che rappresenta il pianeta) i cui elementi possono assumere i soli valori 0 o 1. Un valore 1 nella cella (i, j) della matrice simboleggia che nella stessa cella (i, j) della matrice rappresentante il pianeta è stato depositato un pesce/squalo appena nato, e che quindi nessuna regola dev'essere applicata se mi imbatto in essa. Un valore di 0 indica invece che sono libero di applicare le opportune regole sulla cella. Ovviamente tutte le celle della matrice di supporto vengono settate a zero quando scatta un nuovo chronon.
La struttura dati utilizzata dal thread **dispatcher** per commissionare i task ai thread **worker** è una semplice lista concatenata. Gli inserimenti avvengono in coda, mentre i prelievi (che comportano anche la rimozione dell'elemento dalla lista) vengono effettuati dalla testa. Gli elementi contenuti nella lista racchiudono tutte le informazioni necessarie ad un thread **worker** per poter svolgere il proprio lavoro. Tali informazioni sono organizzate in una struct e sono:

- Le coordinate della cella situata in alto a sinistra del rettangolo di competenza del **worker**.
- La larghezza del rettangolo
- L'altezza del rettangolo

1.2 Algoritmi principali

L'algoritmo di suddivisione del pianeta in rettangoli di dimensione KxN (usato dal thread **dispatcher** per inserire i task nella coda condivisa) opera in questo modo:

1. Vengono calcolate le dimensioni dei rettangoli in accordo alle seguenti formule:

$$K = \left\lceil \frac{nrow}{nwork} \right\rceil \qquad N = \left\lceil \frac{ncol}{nwork} \right\rceil$$

   Dove *nrow* e *ncol* sono rispettivamente il numero di righe e di colonne della matrice pianeta, mentre *nwork* è il numero di thread **worker**. (Scegliendo K e N in questo modo miro a parallelizzare il più possibile la computazione).

2. Ci si chiede quanti rettangoli di larghezza N si "incastrerebbero" bene nel senso della larghezza nella matrice. Nel migliore dei casi, la matrice verrà divisa in rettangoli di larghezza N senza che nessuna cella rimanga tagliata fuori. Se invece eccede un certo numero di celle (un numero minore di N), l'ultimo rettangolo avrà quella larghezza. Ad esempio, se la larghezza della matrice è uguale a $h \cdot N + r$ (con $h \geq 0$ e $0 \leq r < N$), allora h rettangoli di larghezza N si incastrano bene (nel senso della largezza) mentre l'ultimo avrà larghezza troncata ad r.

3. Ci si chiede quanti rettangoli di altezza K si "incastrerebbero" bene nel senso dell'altezza nella matrice. Nel migliore dei casi, la matrice verrà divisa in rettangoli di altezza K senza che nessuna cella rimanga tagliata fuori. Se invece eccede un certo numero di celle (un numero minore di K), l'ultimo rettangolo avrà quell'altezza. Ad esempio, se l'altezza della matrice è uguale a $h \cdot K + r$ (con $h \geq 0$ e $0 \leq r < K$), allora h rettangoli di altezza K si incastrano bene (nel senso del'altezza) mentre l'ultimo avrà altezza troncata ad r.

2 Strutturazione del codice

2.1 Divisione su file

L'intero progetto è suddiviso in più file secondo la seguente struttura:

- file **wator.h** (contiene la specifica di strutture dati, procedure e funzioni che vengono usate per svolgere la simulazione ed alcune macro.)
- file **cells.c** (contiene l'implementazione delle funzioni `cell_to_char` e `char_to_cell` definite in **wator.h**.)
- file **planet.c** (contiene l'implementazione delle funzioni `new_planet`, `print_planet`, `load_planet`, `fish_count`, `shark_count` e della procedura `free_planet` definite in **wator.h**.)
- file **rules.c** (contiene l'implementazione delle funzioni `shark_rule1`, `shark_rule2`, `fish_rule3` e `fish_rule4` definite in **wator.h**.)

- file **wator.c** (contiene l'implementazione delle funzioni `new_wator`, `update_wator` e della procedura `free_wator` definite in **wator.h**.)
- file **util.h** (contiene la specifica di funzioni e procedure di supporto definite da me ed alcune macro.)
- file **util.c** (contiene l'implementazione delle funzioni e delle procedure definite in **util.h**.)
- file **watorscript** (contiene lo script richiesto nella specifica del progetto.)
- file **wator_process.c** (è il cuore del progetto, e contiene il codice del processo **wator** e dei thread **dispatcher**, **collector** e **worker**.)
- file **visualizer.c** (contiene il codice del processo **visualizer**.)

2.2 Librerie

L'unica libreria di cui viene fatto uso nel codice dei processi/thread è la libreria statica **libWator.a**. La libreria consta di eseguibili che contengono complessivamente l'implementazione di tutte le funzioni/procedure definite nel file **wator.h**.

3 Struttura dei programmi

3.1 Struttura di wator_process.c

La prima parte del programma è volta a mettere in piedi l'intero sistema di simulazione. Per prima cosa, vengono definiti i gestori dei segnali che il processo si aspetta di ricevere. Successivamente, si passa all'analisi degli argomenti da riga di comando, dei quali viene verificata la correttezza prima della loro interpretazione. Segue la `fork` del processo **visualizer**, poi la generazione dei thread **dispatcher**, **collector** e di un opportuno numero di thread **worker**. La seconda parte del programma è quella in cui si attende passivamente l'arrivo di segnali. La struttura del codice del thread **dispatcher** è piuttosto semplice. Inizialmente, il thread maschera tutti i segnali di competenza altrui, quindi comincia un ciclo. All'inizio del ciclo, **dispatcher** verifica se è autorizzato a distribuire i task e, nel caso non lo sia, si sospende. Una volta arrivato il momento, vengono distribuiti i task seguendo l'algoritmo di suddivisione dei rettangoli illustrato nella sezione 1.2. A questo punto viene dato il via libera ai thread **worker**. Il codice del thread **collector** è invece strutturato come segue: Vengono mascherati i segnali di competenza altrui e definito il gestore per il segnale **SIGUSR2**; si inizia un ciclo in cui dapprima si verifica se la computazione relativa al chronon corrente è stata portata a compimento. Qualora non fosse così, **collector** si sospende. In caso contrario, **dispatcher** viene invitato a distribuire i nuovi task e **collector** soddisfa eventuali richieste di dump del pianeta interagendo con **visualizer** tramite la socket. La struttura del codice dei thread **worker** è tale per cui in un primo momento si mascherano i segnali destinati ad altri e si crea il file vuoto **wator_worker_wid**, come da specifica. Il thread entra poi in un ciclo in cui cerca di prelevare un task dalla coda (sospendendosi temporaneamente se questa è vuota), quindi applica le regole sulle

celle del rettangolo e infine avverte **collector** se si accorge che il task appena completato era l'ultimo della computazione.

3.2 Struttura di visualizer.c

Similmente, la prima azione intrapresa qui è la definizione dei gestori dei segnali. Una volta fatto questo, si attendono connessioni sulla socket dalla quale proverranno i dati del pianeta di cui si vuole effettuare il dump.

3.3 Struttura di watorscript

Nel file **watorscript** sono definite due procedure di supporto: `process_arg` e `scan_planetfile`. La prima delle due serve per raccogliere ed organizzare in un array gli argomenti dello script, verificare che siano corretti e che non siano ripetuti più di una volta. La seconda analizza carattere per carattere il file che rappresenta il pianeta per verificare che sia correttamente formattato. L'ultima parte dello script interpreta i parametri.

4 Interazioni tra i processi/thread

Le interazioni che avvengono tra i processi/thread (escludendo quelle richieste dalla specifica) sono le seguenti:

- Il processo **wator** invia un segnale **SIGTERM** al processo **visualizer** quando la simulazione deve terminare, indipendentemente dalla causa (che può essere la ricezione da parte di **wator** di un segnale **SIGINT** o **SIGTERM** o il semplice raggiungimento della durata totale della simulazione).

- Il processo **visualizer** invia un segnale **SIGUSR2** al thread **collector** quando è pronto ad accogliere nuove connessioni sulla socket. Se quest'ultimo si accorge che **visualizer** non è ancora pronto, attende passivamente la ricezione del segnale.

- Il thread **collector**, quando necessario, spedisce a **visualizer** il pianeta scrivendo sulla socket una riga alla volta. Il processo **visualizer** sa quante righe aspettarsi (e quanti elementi devono trovarsi all'interno di ogni riga) perché il processo **wator**, quando ne esegue la `fork`, lo fa passando come parametri il numero di righe e colonne del pianeta.

- Il thread **collector** esegue una `signal` su una variabile di condizione per avvertire **dispatcher** che la computazione relativa al chronon corrente è stata completata con successo e che quindi può distribuire i task relativi al chronon successivo.

- Il thread **worker** che porta a compimento l'ultimo task relativo al chronon corrente avverte **collector** tramite una `signal` su una variabile di condizione, il quale decide come procedere (continuare normalmente il ciclo, richiedere il dump o terminare).

- Il thread **dispatcher** esegue una `broadcast` su una variabile di condizione per risvegliare i thread **worker** che si erano sospesi incontrando la lista dei task vuota.

Una visione più chiara si può avere osservando la figura 4.1.

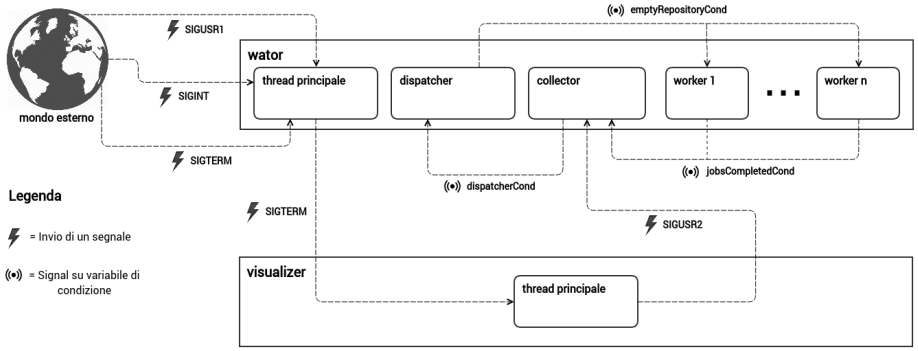

Figura 4.1: Le interazioni tra i processi/thread

5 Gestione della concorrenza tra i processi/thread

Problemi legati alla concorrenza possono verificarsi principalmente in due occasioni particolari, ovvero:

1. Quando più thread **worker** accedono alla coda per prelevare i task
2. Quando più thread **worker** accedono alla matrice che rappresenta il pianeta per applicare le regole

Nel primo caso è sufficiente una **lock**, che deve essere acquisita da ciascun thread **worker** prima di effettuare una qualunque operazione sulla coda. Nel secondo caso, bisogna essere sicuri che un thread **worker** non applichi regole sulla base del contenuto di celle che altri thread stanno osservando, e che quindi potrebbero modificare "nel momento sbagliato" (ad esempio subito dopo che l'altro thread ha appurato che essa contiene acqua ma prima che abbia intrapreso l'opportuna azione). La soluzione adottata consiste nell'imporre ai thread **worker** di acquisire una lock qualora si trovassero ad esaminare celle locate ai bordi del rettangolo o a distanza 1 da essi. È solo in quelle occasioni, infatti, che si è soggetti al problema di cui sopra. La figura 5.1 mostra le celle "sicure" e quelle "a rischio" in una matrice 23x37 su cui operano 4 thread **worker**.

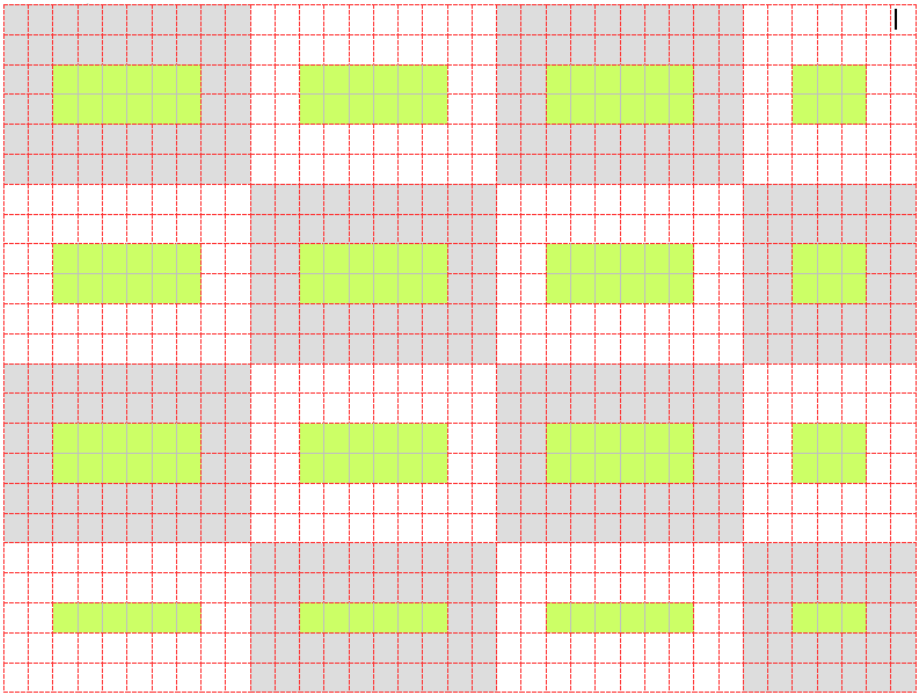

Figura 5.1 : Un esempio di suddivisione di un pianeta

I blocchi bianchi/grigi rappresentano i rettangoli in cui viene suddiviso il pianeta. Le celle contornate di rosso sono quelle “a rischio” (richiedono l’acquisizione della **lock**), mentre quelle colorate di verde sono quelle “sicure”.