Esercitazione 4

10 / 12 maggio 2023

Esercizio 1

Un noto rompicapo prevede di trovare la sequenza di operazioni elementari (somma, sottrazione, moltiplicazione e divisione) necessarie per ottenere 10, partendo da 5 numeri casuali da 0 a 9. I vincoli sono:

- le cinque cifre sono comprese tra 0 e 9 e possono ripetersi
- le cinque cifre devono essere utilizzate tutte, in qualsiasi ordine
- non ci sono limiti sulle operazioni (es. vanno bene anche quattro somme)
- non si considera la precedenza degli operatori, le operazioni vanno applicate da sinistra a destra secondo il loro ordine

Esempio: dato 2 7 2 2 1 una soluzione può essere 7 - 2 - 1 x 2 + 2 = 10

Scrivere un programma che, letta la sequenza di numeri da command line come argomento, trovi tutte le possibili soluzioni, se ve ne sono, le salvi in un vettore di stringhe (es: “7 - 2 - 1 x 2 + 2”) e lo stampi.
In un primo momento utilizzare un approccio *brute force*, ovvero elencare tutte le possibili permutazioni delle cinque cifre e, per ogni permutazione, tutte le possibili permutazioni delle quattro operazioni, provando a calcolare il risultato per ciascuna di esse. Se il risultato è 10 la permutazione viene salvata, altrimenti viene scartata.

Una volta verificato il corretto funzionamento, sfruttare dei thread per velocizzare la ricerca delle soluzioni in parallelo. Si divide la lista di tutte le possibili permutazioni in n blocchi uguali e per ciascuna si lancia un thread. Provare con n=2,3,4… ecc, misurare i tempi e trovare il numero di thread oltre il quale non vi sono vantaggi.
Cambia qualcosa se la divisione del lavoro fra thread anziché essere a blocchi è *interleaved*? Vale a dire con tre thread il primo prova le permutazioni con indice 0,3,6,... il secondo 1,4,7,... e il terzo 2,5,8,…

Esercizio 2

Adattare l’esercizio sul buffer circolare (esercizio 2 dell’esercitazione 2) facendo sì che:

- vi sia un solo processo
- producer e consumer vengono eseguiti in due thread differenti
- il buffer non sia più su file, ma in una struttura in memoria condivisa tra i due thread

Le altre logiche di funzionamento rimangono inalterate, quindi il producer scrive un valore al secondo, il consumer legge i valori ogni 10 secondi.

Attenzione: anche in questo caso l'accesso al buffer e agli indici di scrittura e lettura va sincronizzato, garantendo la mutua esclusione fra i due thread.

Bonus: modificare producer e consumer in modo che scrivano e leggano sul buffer senza pause (se non quando in buffer è pieno o vuoto) e misurare il throughput del buffer con dimensioni differenti (10, 1000, 10000 valori). Misurare anche il throughput aumentando il numero di producer e consumer: come varia?

Esercizio 3

Nell'esercizio sul file system (esercizio 3 dell'esercitazione 2) non si potevano restituire oggetti MatchResult contenenti directory, in quanto la soluzione richiedeva di utilizzare più di un riferimento mutabile alla stessa directory e, quindi, ci eravamo limitati a cercare dei file. Per aggiungere una directory nel risultato occorreva infatti salvare un suo riferimento mutabile, ma per poter proseguire la ricerca nei suoi figli occorreva mantenere sempre un riferimento mutabile alla directory stessa, cosa che il compilatore di Rust impedisce.

Per gestire situazioni analoghe, Rust mette a disposizione gli smart pointer Cell e RefCell che implementano il pattern di interior mutability, vale a dire che Cell<T> non è mutabile, ma è possibile provare a ottenere un riferimento mutabile a T a runtime. Solo nel momento in cui viene chiesto il riferimento viene fatto un check e la richiesta fallisce se non è possibile.

Modificare l'esercizio inserendo i nodi (file e directory) in Cell o RefCell e modificare la ricerca permettendo di restituire anche directory nei risultati.

Esercizio 4

Risolvere l'esercizio parallel letter frequency di exercism.
Eseguire anche i benchmark indicati e discuterne i risultati.

Quando la performance della soluzione in parallelo è inferiore alle attese come individuare i possibili motivi? Ecco alcuni suggerimenti

- l'input ha pochi dati? il costo di far partire e chiudere n thread potrebbe essere più alto del tempo guadagnato con una soluzione parallela
- attenzione alle copie dei dati! Ad esempio, quante volte dovete copiare l'input per passarlo ai thread? E' necessario?
- attenzione alla sincronizzazione: più sono i punti di sincronizzazione, più il programma è rallentato; ad esempio può convenire che i thread non accedano ad una struttura condivisa per i conteggi, ma lavorino in modo autonomo e uniscano I risultati alla fine (copiare poche volte molti dati può essere meglio di effettuare tante sincronizzazioni per copiare dati più semplici)

Esercizio 5

Implementare un modulo che offre le funzioni di una barriera ciclica.

La barriera è un costrutto di sincronizzazione che permette a n thread di attendere che tutti arrivino a un punto comune prima di andare avanti.
La barriera viene inizializzata con il numero di thread attesi (n) e, quando un thread chiama barrier.wait(), si blocca finché tutti i thread non chiamano wait().
Si dice ciclica una barriera che può essere riusata; questo implica che prima di permettere a nuovi thread di entrare occorre aspettare che siano stati sbloccati tutti.

Un esempio di come può essere usata:

```
fn main() {
  let abarrrier = Arc::new(cb::CyclicBarrier::new(3));
  let mut vt = Vec::new();
  for i in 0..3 {
    let cbarrier = abarrrier.clone();
    vt.push(std::thread::spawn(move || {
      for j in 0..10 {
        cbarrier.wait();
        println!("after barrier {} {}\n", i, j);
      }
    }));
  }
  for t in vt {
    t.join().unwrap();
  }
}
```

In questo esempio i thread avanzano con i rispettivi indici “j” sincronizzati, nessun thread avanza più velocemente degli altri. Provate a vedere la differenza commentando wait().
Attenzione! La barriera ha due fasi di funzionamento: quella di riempimento, in cui si attende che giungano tutti i thread attesi, e quella di svuotamento, in cui i thread vengono via via fatti uscire (come conseguenza dei meccanismi di sincronizzazione in uso). Può succedere che si presenti all'ingresso un thread che cerca di entrare mentre è in corso lo svuotamento: se le due fasi non vengono correttamente gestite, si potrebbe violare l'invariante della barriera (ovvero potrebbe succedere che un thread esca dalla barriera prima che siano giunti altri n-1 thread nell'ambito del ciclo riempimento/svuotamento corrente).

Esercizio 6

Un programma deve monitorare una macchina leggendo i valori da 10 sensori, che impiegano tempi diversi per fornire il risultato e sono letti da 10 thread, uno per sensore (simulare la lettura con una funzione read_value() che fa una sleep di lunghezza casuale e restituisce un numero casuale compreso tra 0 e 10).
Una volta raccolti i 10 valori, un altro thread raccoglie i risultati e ne esegue la somma. Se il risultato è maggiore di 50 rallenta la macchina, se inferiore l’accelera (da simulare con una funzione set_speed() che fa una sleep di lunghezza casuale).
È importante che lettura parametri (ciclo read) e impostazione macchina (ciclo write) non si sovrappongano, in quanto i valori potrebbero venire perturbati.

Il programma inoltre fa infiniti cicli read/write.

Provare a risolvere il problema utilizzando sia una versione modificata della barriera ciclica realizzata nell’esercizio 5 che i canali.