Esercitazione 6

14/16 maggio 2025

Esercizio 1: suddivisione lavoro fra thread

I thread sono spesso usati per suddividere il lavoro tra i core della cpu, per eseguire in parallelo delle porzioni lavoro e unire i risultati. Questo aspetto diventa interessante quando:

- l'algoritmo è parallelizzabile, ovvero ogni thread può risolvere in modo indipendente un pezzo di lavoro senza dover aspettare il risultato di alti thread
- le eventuali operazioni di sincronizzazione non sono troppo pesanti rispetto al lavoro da svolgere per risolvere il problema

In questo esercizio vediamo alcuni problemi che possono essere facilmente divisi in thread

Numeri primi

Nel file file prime.rs viene fornita una versione naive di **is_prime(n: us64)**, che verifica con brute force se n è un numero primo. Scrivere una funzione che cerchi tutti i numeri primi tra 2 e limit, suddividendo il lavoro tra n thread:

pub fn find_primes(limit: u64, n_threads: u64) -> Vec<u64> {}

Per suddividere il lavoro avete due possibilità, implementarle entrambe e verificare la più efficiente:

- condividere una variabile **counter** tra ogni thread, ogni thread a turno incrementa counter e verifica se quel numero è primo, se è primo lo memorizza altrimenti lo scarta; alla fine restituisce i numeri primi che ha trovato
- non condividere nulla, ogni thread conta a partire da 2,3,4,5,...n a limit modulo n e verifica quel numero; in questo modo ogni thread proverà dei numeri differenti (perché dividere in blocchi contigui sarebbe meno efficiente?)
  Es con tre thread uno rispettivamente verificherà:
  2 5 8 11 …
  3 6 9 12 …
  4 7 10 13 …

Provare la funzione cercando i numeri primi fino a un milione e passando da 1 a 16 thread: il tempo diminuisce linearmente con l'aumento dei thread? Perché?
Per misurare i tempi usare: **std::time::Instant::now()**

Verifica soluzioni

Quando si ha un set di dati da filtrare o di soluzioni da verificare, se ogni elemento del set è indipendente allora è facile suddividere il lavoro tra thread, ogni thread si occupa di un blocco di dati in parallelo con altri.
Un esempio è questo rompicapo: dati 5 numeri casuali scelti tra 0 e 9 occorre trovare una sequenza di operazioni aritmetiche che li utilizzi tutti, in qualsiasi ordine, e che dia come risultato 10.
Un soluzione è verificare brute force tutte le possibili combinazioni. Viene quindi fornito un

file game.rs con una funzione che prepara tutte le possibili permutazioni dei 5 numeri scelti (in s, es “74648”)

pub fn prepare(s: &str) -> Vec<String>

Il risultato è un vettore di stringhe con “esplose” tutte le possibili combinazioni date dalle permutazioni delle 5 cifre e unite alle 4 operazioni aritmetiche (5! le cifre e 4^4 e le operazioni, essendo ammessa ripetizione).
Si leggerò quindi qualcosa del tipo:["7+4+6+4+8", "7+4+6+4-8", "7+4+6+4*8", "7+4+6+4/8", ...]

Scrivere una funzione verify così definita

pub fn verify(v: &[String]) -> Vec<String> {}

che verifichi una per una le stringhe se danno come risultato 10 e restituisca quelle che hanno successo. verify() va provata su 1,2,3,..16 thread, suddividendo il vettore in opportune slice. Divisioni per 0 (zero) o frazionarie vanno scartate, tenere solo operazioni che diano come risultato interi.
Non è ammesso copiare completamente il vettore in ogni thread, il vettore è unico e i thread ricevono uno slice.

Suggerimenti:

- suddividendo il vettore in slice non sovrapposti non c’è bisogno di sincronizzazione, se non per raccogliere i risultati di verify()
- dovendo condividere uno slice di un vettore in diversi thread, quali problemi di lifetime ci possono essere? Come si può risolvere?

Esercizio 2: producer / consumer

Un pattern concorrente molto comune è il producer/consumer, dove un thread produce dei valori e li memorizza in una struttura condivisa (di solito una coda FIFO), e un secondo thread consuma i valori che estrae dalla struttura condivisa.

In questo setup molto semplice bisogna stare attenti ad alcune cose:

- la coda condivisa non può crescere all’infinito, ma se il producer è più veloce del consumer, il producer deve fermarsi finché il consumer non svuota almeno parzialmente la coda (questo meccanismo si chiama “*backpressure*” e permette di non saturare la memoria: casi molto comuni sono quando il producer legge da disco o riceve richieste da rete)
- quando la coda è vuota il consumer non dovrebbe usare cpu per controllare in continuazione se ci sono nuovi valori, ma dovrebbe essere svegliato solo quando c’è qualcosa da processare e farlo il più velocemente possibile
- occorre prestare molta attenzione alle condizioni di terminazione: quando il producer ha finto ed esce, il consumer deve prima processare tutti gli elementi nella coda prima di uscire anche lui; se esce prima il consumer il producer deve smettere di inserire nuovi valori e uscire pure lui.

Per gestire correttamente la terminazione possiamo creare una coda (VecDeque) che ha come elementi questa enum: **Item {Value(T), Stop}**.Quando il producer ha terminato può scrivere Stop nella coda ed esce, quando il consumer trova Stop esce pure lui.

Scrivere quindi una struct MyChannel che permetta di gestire più producer e più consumer rispettando i vincoli espressi sopra. La struttura suggerita è la seguente:

```
impl<T> MyChannel<T> {
        pub write(item: T) -> Result<(),_> {}
        pub read() -> Result<T, _> {}
        pub close() {}
}
```

MyChannel dentro deve avere un buffer FIFO di n elementi (specificato in costruzione).
Se il buffer è pieno, la write() rimane appesa fino a quando non si libera dello spazio per scrivere.
Se il buffer è vuoto è la read() che deve rimanere bloccata finché non c'è un valore da leggere.
Quando si chiama close() il canale viene chiuso, da quel momento in avanti ogni write() fallirà, mentre le read() continueranno finché il buffer non è svuotato, per poi dare errore anche loro.
Implementata la struttura scrivere due thread: uno che produca N valori ad intervalli casuali e poi chiami close() ed esca, uno che li consumi stampandoli. Verificare che vengano tutti ricevuti.

Suggerimenti: siccome sia read() che write() possono doversi bloccare e attendere che accada "qualcosa" nell'altro thread, considerare l'uso di una **condition variable** nella soluzione.
(l'alternativa sarebbe fare busy waiting, testando in loop la lunghezza della struttura per ogni read e write: potete provarla e confrontare la differenza di prestazioni e uso cpu)