Esercitazione 5

Esercizio 1

Risolvere l'esercizio parallel letter frequency di exercism

https://exercism.org/tracks/rust/exercises/parallel-letter-frequency

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 2

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 3

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 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).
E’ 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 2 che i canali.