Programmazione di sistema

Anno accademico 2017-2018

Esercitazione 4

Si implementi una soluzione concorrente al problema presentato nell'esercitazione 3 in grado di sfruttare la presenza di più core per esplorare l'intero spazio di soluzioni in un tempo inferiore a quello necessario all'implementazione sequenziale.

A tale scopo si operi come segue

1. Si determini il tempo di esecuzione per la soluzione sequenziale elaborata nel precedente laboratorio. Su macchine di derivazione Unix, questo può essere fatto utilizzando, da terminale, il comando
   ```
   time nome-eseguibile
   ```
   su macchine Windows, aprire una finestra Windows PowerShell ed eseguire il comando
   ```
   Measure-Command { nome-eseguibile }
   ```
   (le parentesi graffe sono obbligatorie). Eventualmente eseguire più volte la misura per determinare un valore medio più attendibile. Tale valore costituirà il riferimento per le attività successive.
2. Si faccia una copia del progetto precedente (così da non perderlo) e si cominci a modificare la struttura dati utilizzata per il caching delle soluzioni: tale struttura dovrà essere resa thread-safe, incorporando un opportuno meccanismo di sincronizzazione che consenta a due o più thread di accedere ai dati contenuti senza provocare interferenza. Per generalità, possiamo introdurre la seguente classe generica
   ```
   template<typename Problem, typename Solution>
   class cache_t {
     public:
       bool get(const Problem& p, Solution& s); //ritorna false in caso di MISS
       void put(const Problem& p, const Solution& s);
   };
   ```
   Tale classe dovrà essere corredata, nella propria parte privata, delle strutture dati necessarie a gestire in modo thread-safe il funzionamento della cache.
3. Si adatti il programma nella sua forma sequenziale corrente per utilizzare come cache una istanza della classe così descritta e si misuri il tempo di esecuzione, che dovrebbe risultare leggermente aumentato a seguito del costo delle operazioni di sincronizzazione introdotte e si controlli che produca esattamente lo stesso insieme di soluzioni della versione originale (con 7 numeri, ci sono 10851 soluzioni da {0,0,0,0,0,0,0} con punteggio 0 a {9,9,9,9,9,9,9} con punteggio 1656).
4. Poiché il problema trattato è essenzialmente CPU-bound (piuttosto che I/O-bound), il possibile beneficio introdotto dalla programmazione concorrente in teoria risulta massimo quando viene suddiviso tra un numero di thread pari al numero di core disponibili. Si modifichi pertanto la struttura del main() così da suddividere lo spazio di ricerca in N sottoinsiemi (N dovrà essere variato nei passi successivi, per cui si abbia cura di poterlo definire in un singolo posto), ciascuno costituito da 10.000.000/N possibili combinazioni iniziali (i problemi).
5. Utilizzando la funzione std::async(...), si affidi a N thread la soluzione concorrente dei singoli sottoinsiemi di problemi e si attenda la conclusione di queste attività attraverso il

metodo get() dell'oggetto std::future restituito da std::async(...). Si verifichi che il numero delle soluzioni ottenute continui ad essere coerente con quanto trovato in precedenza.
(Per stampare le soluzioni trovate, si abbia cura di definire un'apposita funzione responsabile della stampa di una singola soluzione, che possa essere chiamata dai diversi thread via via che ne individuano una lecita. Tale funzione potrà fare riferimento al proprio interno un oggetto di tipo std::mutex allo scopo di garantire che non ci siano sovrapposizioni sulle operazioni di I/O. Poiché la sequenza delle soluzioni generate risulterà inevitabilmente diversa da quella ottenuta con il programma originale, utilizzando i comandi offerto dallo shell di sistema, si provveda a salvare lo standard output prodotto dal programma su un file, ordinarlo e confrontarlo con il corrispettivo prodotto dal programma iniziale per essere certi che entrambi forniscano le stesse soluzioni).

6. Quando si ha la certezza che le due soluzioni sono funzionalmente identiche, si misurino e confrontino i tempi di esecuzione: la soluzione trovata è più veloce di quella originale? Perché? Si modifichi il valore di N per vedere cosa succede aumentando o riducendo il grado di parallelismo.
7. Si operi sulla classe cache_t per ridurre il livello di contesa legato all'accesso mutuamente esclusivo: se tale classe adotta un singolo mutex, qualsiasi thread che intende accedere (in scrittura o lettura) alla cache, deve attendere che nessun altro stia facendo un'operazione con la cache stessa. Poiché le singole coppie Problem/Solution sono indipendenti, è possibile partizionare la cache in M cache più piccole, ed assegnare una singola coppia ad una specifica sotto-cache in funzione di qualche parametro del problema. A tale scopo si introduca nella classe cache_t un metodo privato size_t hash(const Problem &p) che trasformi, con una ragionevole euristica, il problema in un corrispondente valore hash e si assegni tale problema alla sotto-cache di indice `hash(p) % M` così da separare lo spazio di contesa. Si verifichi la correttezza delle modifiche apportate (il programma deve continuare a generare le stesse soluzioni, al di là dell'ordinamento) e si misurino i tempi di esecuzione al variare di M (si provino valori di M pari a potenze del 2: 16,32,64,..., 1024, 2048,...). Come variano i tempi di esecuzione? Perché?
8. Si approfondisca il concetto di scalabilità dell'implementazione di una struttura dati concorrente studiando i seguenti articoli/libri
    a. https://github.com/huxia1124/ParallelContainers
    b. http://preshing.com/20160201/new-concurrent-hash-maps-for-cpp/
    c. https://manning-content.s3.amazonaws.com/download/a/9f3eb86-11ee-4825-9275-3818d63ffbe9/Williams4_CplusplusCiA_MEAP_ch1_v06.pdf
    d. https://manning-content.s3.amazonaws.com/download/3/75f3382-1e80-407a-894e-309e26860893/sample_ch02_CCiA.pdf (edizione precedente del libro di cui sopra)

Competenze da acquisire

- Programmazione concorrente
- Tecniche di sincronizzazione
- Comprensione dell'interazione con il sistema operativo
- Scalabilità e parallelismo