Corso di High Performance Computing

Esercitazione OpenMP del 16/10/2018

Moreno Marzolla

Ultimo aggiornamento: 2018-10-14

Per svolgere l'esercitazione è possibile collegarsi tramite ssh al server `disi-hpc.csr.unibo.it` oppure isi-`raptor03.csr.unibo.it`, usando come nome utente il proprio indirizzo mail istituzionale completo (es. `paolo.rossi@studio.unibo.it`), e come password la propria password istituzionale (quella usata per accedere alla casella di posta o ad AlmaEsami). Sulla macchina è installato il compilatore `gcc` e alcuni editor di testo per console: `vim`, `pico`, `joe`, `ne` e `emacs`. Per chi non è pratico suggerisco `pico`, che è semplice da usare e richiede poche risorse. Chi ha un portatile con il compilatore installato e configurato correttamente può lavorare in locale.

Per scaricare l'archivio con i sorgenti di questa esercitazione è possibile usare i comandi:

```
wget http://www.moreno.marzolla.name/teaching/HPC/ex1-openmp.zip
unzip ex1-openmp.zip
cd ex1-openmp/
```

0. Familiarizzare con l'ambiente di lavoro

Per chi non l'avesse ancora fatto, consiglio di prendere familiarità con l'ambiente di lavoro e con gli esempi visti a lezione:

1. Scaricare i sorgenti dei programmi illustrati a lezione:
   ```
   wget www.moreno.marzolla.name/teaching/HPC/HPC1819.zip
   unzip HPC1819.zip
   cd HPC1819/
   ```
2. Compilare gli esempi visti a lezione usando il comando `make openmp`; in alternativa è possibile compilare manualmente usando i flag `-fopenmp -Wall -Wpedantic -std=c99` di GCC per abilitare la maggior parte dei warning e forzare il rispetto dello standard C99 (molti programmi usano costrutti validi solo in C99).
3. Provare ad eseguire i programmi con varie dimensioni del pool di thread OpenMP, usando la variabile d'ambiente `OMP_NUM_THREADS`; ad esempio
   ```
   OMP_NUM_THREADS=4 ./omp-demo0
   ```

1. Prodotto scalare di due array

Il file `omp-dot.c` contiene una implementazione seriale di un programma che calcola il prodotto scalare di due array `v1[]` e `v2[]`. Il programma accetta come unico parametro a riga di comando la lunghezza n degli array, che vengono inizializzati in modo deterministico, in modo da conoscere il loro prodotto scalare senza doverlo calcolare. Ricordiamo che il prodotto scalare di due array `v1[]` e `v2[]` di lunghezza n è definito come:

$$\sum_{i=0}^{n-1} v1[i] \times v2[i]$$

Parallelizzare la versione seriale, usando inizialmente il costrutto `omp parallel` (non `omp parallel for`). Detto P il numero di thread OpenMP, il programma deve partizionare logicamente l'array in P blocchi di dimensione approssimativamente uniforme. Il thread p-esimo ($0 \le p < P$) calcola una porzione di prodotto scalare corrispondente alla somma dei prodotti degli elementi di `v1[]` e `v2[]` di indici `my_start`, …, `(my_end-1)`, cioè:

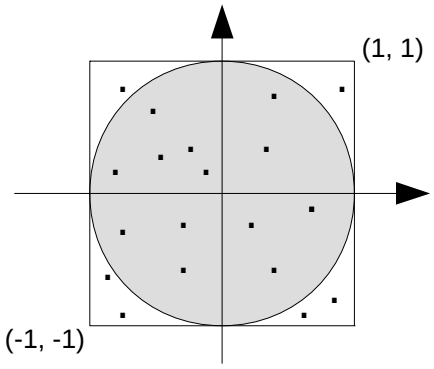
$$\text{my\_p} = \sum_{i=\text{my\_start}}^{\text{my\_end}-1} v1[i] \times v2[i]$$

Il risultato parziale calcolato dal thread *p*-esimo viene memorizzato nell'elemento `partial_p[p]`, dove `partial_p[]` è un array di lunghezza *P*; in questo modo ciascun thread gestisce un elemento diverso di `partial_p[]` e non si verificano *race condition*. Fatto questo, il master calcola la somma dei valori in `partial_p[]`, determinando così il risultato cercato. Si presti attenzione a gestire correttamente il caso in cui la lunghezza *n* degli array non sia un multiplo di *P*.

Includere nel codice il calcolo del tempo di esecuzione della porzione parallela (escludendo l'inizializzazione dell'array). Misurare il tempo di esecuzione del programma al variare del numero di thread usando la variabile d'ambiente `OMP_NUM_THREADS`, e calcolare lo *speedup* e la *weak* e *strong scaling efficiency*.

2. Calcolo del valore approssimato di π

Il file `omp-pi.c` contiene una implementazione seriale di un algoritmo di tipo Monte Carlo per il calcolo del valore approssimato di π. Negli algoritmi di tipo Monte Carlo si fa uso di sequenze di numeri casuali per il calcolo approssimato di valori numerici di interesse.



Il principio di funzionamento dell'algoritmo è molto semplice. Si generano *N* punti casuali uniformemente distribuiti all'interno del quadrato di vertici (-1, -1) e (1, 1). Definiamo con *x* il numero di punti che cadono all'interno del cerchio inscritto in esso. Il rapporto *x* / *N* approssima il rapporto tra l'area del cerchio e l'area del quadrato. Poiché sappiamo che l'area del cerchio inscritto è π mentre l'area del quadrato vale 4, possiamo stimare il valore di π come (4*x* / *N*).

Il file `omp-pi.c` contiene una versione seriale dell'algoritmo descritto sopra. Modificare il codice in modo da sfruttare il parallelismo con OpenMP. Iniziare con una implementazione che usi il costrutto `omp parallel`. Il programma dovrebbe operare come segue:

1. Sia *P* il numero di thread OpenMP
2. Il thread *p* invoca la funzione `generate_points()`, generando *N* / *P* punti e ponendo il risultato nell'elemento `inside[p]` di un array `inside[]` di lunghezza *P*. Tale array va dichiarato fuori dal blocco parallelo in modo da poter essere condiviso da tutti i thread OpenMP.
3. Al termine del blocco parallelo, il master somma i valori dell'array `inside[]`, determinando il numero totale di punti interni al cerchio e completando il calcolo di π

Suggerisco di iniziare assumendo che il numero di punti *N* sia un multiplo di *P*; una volta ottenuta una versione funzionante, modificare il programma in modo da funzionare con valori arbitrari di *N*.

3. Decifrare un messaggio cifrato

Il programma `omp-brute-force.c` contiene un messaggio cifrato memorizzato nell'array `enc[]` lungo 64 byte. Il messaggio è stato cifrato con una chiave lunga 8 byte usando l'algoritmo DES, usando funzioni di libreria che si trovano già installate sul server; la chiave di cifratura è una sequenza di 8 caratteri ASCII che rappresentano cifre numeriche, quindi è compresa tra “00000000” e “99999999”.

Nel programma è fornita una funzione `decrypt(enc, dec, n, key)` per decifrare un messaggio cifrato data la chiave:

- `enc` è un puntatore all'area di memoria contenente il messaggio cifrato;
- `n` è la lunghezza (in byte) del messaggio cifrato;
- `dec` è un puntatore ad un'area di memoria di *n* byte (la stessa lunghezza del messaggio cifrato), che deve essere preallocata dal chiamante, e che al termine della chiamata conterrà il messaggio decifrato;
- `key` è il puntatore alla chiave da usare per decifrare; la chiave è lunga 8 byte.

L'algoritmo di decifratura utilizzato (Data Encryption Standard, DES) produce sempre un messaggio "decifrato" data una chiave qualsiasi; se la chiave non è quella corretta, il messaggio decifrato conterrà caratteri “senza senso”. Nel nostro caso, il messaggio in chiaro è una stringa di testo (terminata con uno zero, quindi stampabile da `printf()`), i cui primi dieci caratteri sono “0123456789” (senza virgolette).

Scrivere un programma per realizzare un attacco di tipo “brute force” allo spazio delle chiavi, sfruttando il parallelismo OpenMP. Il programma deve tentare tutte le possibili chiavi da “00000000” a “99999999”, fino a quando ottiene un messaggio decifrato che inizia con la ssequenza “0123456789”; trovata la chiave, il programma deve stampare il messaggio decifrato, che è una citazione da un vecchio film.

Si noti che, a causa delle caratteristiche dell'algoritmo di cifratura utilizzato, ci sono chiavi diverse che decifrano correttamente il messaggio; questo è dovuto al fatto che l'algoritmo ignora l'ottavo bit di ciascun byte della chiave (la lunghezza effettiva della chiave è quindi di 56 bit, anziché di 64 bit). Ai fini dell'esercizio è sufficiente trovare una qualsiasi delle chiavi di decifratura corrette.

Si suggerisce di iniziare usando un costrutto `omp parallel` (non `omp parallel for`) inserendo all'interno del quale codice che assegna a ciascun thread un opportuno sottoinsieme dello spazio delle chiavi. Ricordare però che il costrutto `omp parallel` si applica a *structured blocks*, ossia a blocchi con un *unico* punto di ingresso e un *unico* punto di uscita. Quindi un thread non può uscire dal blocco con `return`, `break` o `goto` quando ha trovato la chiave corretta (l'uso di simili istruzioni in un blocco parallelo dovrebbe causare un errore in fase di compilazione); d'altra parte non vogliamo aspettare che tutti i thread abbiano esplorato l'intero spazio delle chiavi per terminare. È quindi necessario adottare un meccanismo appropriato per terminare la computazione in modo pulito non appena uno dei thread abbia abbia individuato la chiave. Non sono consentite soluzioni brutali come `exit()` o `abort()` per terminare il programma.

4. Frequenze dei caratteri

Il file `omp-letters.c` contiene una versione seriale di un programma che calcola il numero di ocorrenze e le frequenze delle 26 lettere alfabetiche che compaiono in una sequenza letta da standard input. Per testare il programma vengono forniti alcuni libri in formato ASCII resi disponibili da Project Gutenberg (https://www.gutenberg.org/). Si noti che per tutti e tre i libri le frequenze dei caratteri sono molto simili, così come accade per altri testi di una certa lunghezza in lingua inglese. Ogni lingua ha una propria frequenza caratteristica dei caratteri; chi è interessato può sperimentare con libri in altre lingue reperibili sul sito del Project Gutenberg.

Modificare la funzione `make_freq()` per sfruttare il parallelismo OpenMP mediante i costrutti necessari. Sulla base a quanto abbiamo visto in aula fino ad ora, la cosa più semplice da fare è di creare un array bidimensionale `local_hist[num_threads][26]`, dove `num_threads` è la dimensione del pool di thread OpenMP. Dopo averlo inizializzato a zero, ogni thread incrementa il valore opportuno della propria riga. Alla fine il numero di occorrenze di ciascun carattere si ottiene sommando la colonna di `local_hist` appropriate. Si presti attenzione che la funzione `make_hist()` deve restituire il numero di lettere dell'alfabeto processate.