Corso di High Performance Computing

Esercitazione OpenMP del 13/10/2020

Moreno Marzolla

Ultimo aggiornamento: 2020-10-11

Per svolgere l'esercitazione è possibile collegarsi tramite ssh al server `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 https://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 https://www.moreno.marzolla.name/teaching/HPC/HPC2021.zip
   unzip HPC2021.zip
   cd HPC2021/
   ```
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. 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 crittografico XOR, che consiste nell'applicare l'operatore logico "or esclusivo" (xor) tra il testo in chiaro e la chiave. Tale algoritmo non è sicuro, ma risulta adeguato per i nostri scopi.

La chiave di cifratura è una sequenza di 8 caratteri ASCII che rappresentano cifre numeriche; è quindi compresa tra "00000000" e "99999999". Nel programma è fornita una funzione `decrypt(enc, dec, n, key, keylen)` 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; viene interpretata come una sequenza "grezza" di byte, quindi non deve necessariamente essere terminata con 0 come per le stringhe di caratteri.
- keylen è la lunghezza della chiave.

L'algoritmo di cifratura 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 sequenza "0123456789"; trovata la chiave, il programma deve stampare il messaggio decifrato, che è una citazione da un vecchio film.

Si consiglia di usare un costrutto omp parallel 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 che ha trovato la chiave corretta non può uscire dal blocco con return, break o goto (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 realizzare un qualche meccanismo per terminare la computazione in modo pulito non appena uno dei thread abbia individuato la chiave. Non sono consentite soluzioni brutali come exit() o abort().

Nota: il tempo di esecuzione del programma parallelo potrebbe cambiare in modo molto irregolare al variare del numero P di thread OpenMP. Riuscite a dare una spiegazione di ciò?

2. 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 con le clausole che si ritengono appropriate. Si può procedere nel modo seguente: 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 ($0 \leq p < P$) 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:

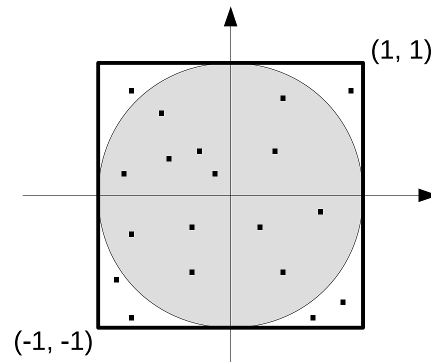
$$\text{my_p} = \sum_{i=\text{my_start}}^{\text{my_end}-1} v1[i] \times v2[i]$$

Per accumulare i risultati parziali è possibile procedere in diversi modi. Una possibilità consiste nel fare in modo che il valore calcolato dal thread p-esimo venga 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*. Il master calcola

poi 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. Una soluzione più comoda consiste nel fare uso della clausola `reduction()` che vedremo nelle prossime lezioni.

3. 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 è semplice. Si generano N punti casuali uniformemente distribuiti all'interno del quadrato di vertici (-1, -1) e (1, 1). Detto x il numero di punti che si trovano 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 l'area del cerchio inscritto è π e l'area del quadrato è 4, possiamo stimare il valore di π come ($4x / N$). Si può dimostrare che tale stima tende ad essere più accurata all'aumentare del numero N di punti generati.

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`. Detto P il numero di thread OpenMP, il programma potrebbe operare come segue:

1. Il thread p genera N / P punti invocando la funzione `generate_points()`, ponendo il risultato nell'elemento `inside[p]` di un array `inside[]` di lunghezza P. Tale array va dichiarato fuori dal blocco parallelo perché deve essere condiviso da tutti i thread.
2. 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 multiplo di P; una volta ottenuta una versione corretta, modificare il programma in modo da funzionare con valori arbitrari di N.

4. Frequenze dei caratteri

Il file `omp-letters.c` contiene una versione seriale di un programma che calcola il numero di occorrenze e le frequenze delle 26 lettere alfabetiche che compaiono in un file letto da standard input; le istruzioni per usare il programma sono descritte nei commenti iniziali. Vengono forniti tre libri di esempio in formato ASCII resi disponibili da Project Gutenberg (https://www.gutenberg.org/); è interessante osservare che le frequenze dei caratteri sono molto simili in tutte le opere, come lo sarebbero in qualunque testo di una certa lunghezza in lingua inglese. La distribuzione delle frequenze dei caratteri cambia da lingua a lingua; chi è interessato

può sperimentare con altri libri reperibili sul sito del Project Gutenberg.

Modificare la funzione `make_freq()` per sfruttare il parallelismo OpenMP. Suggerisco di creare un array bidimensionale `local_hist[num_threads][26]`, dove `num_threads` è la dimensione del pool di thread OpenMP. Dopo aver inizializzato l'array 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 ricordi che la funzione `make_hist()` deve restituire il numero complessivo di lettere dell'alfabeto trovate.

5. Il Crivello di Eratostene

Il *crivello di Eratostene* è un algoritmo che calcola i numeri primi appartenenti ad un dato intervallo che normalmente è l'insieme 2, … n}. Ricordiamo che un intero $p \geq 2$ è primo se e solo se gli unici suoi divisori sono 1 e p (2 è un numero primo).

Per illustrare il funzionamento del crivello di Eratostene usiamo un semplice esempio con $n = 20$. Iniziamo tabulando gli interi 2, … n:

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Il primo valore della tabella (2) è un numero primo; marchiamo tutti i suoi multipli, ottenendo la nuova tabella:

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Il successivo valore non marcato (3) è anch'esso un numero primo. Marchiamo tutti i suoi multipli, partendo da 3×3 (infatti 3×2 è già stato considerato tra i multipli di 2), ottenendo:

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Il successivo valore non marcato (5) è primo. Il più piccolo multiplo di 5 non ancora marcato è 5×5, dato che 5×2 è già stato considerato tra i multipli di 2, e 5×3 tra i multipli di 3. Poiché $5 \times 5 > 20$, il procedimento termina e tutti i valori non marcati sono primi:

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Il file `omp-sieve.c` contiene una implementazione seriale di un programma che conta quanti sono i numeri primi appartenenti all'insieme $\{2, \ldots n\}$ usando il crivello di Eratostene, con n parametro passato sulla riga di comando. L'implementazione seriale fornita potrebbe essere resa più efficiente, ma si è deciso di favorire la comprensibilità a scapito dell'efficienza. La tabella è rappresentata dall'array `isprime[]` di lunghezza $n + 1$; durante l'esecuzione, `isprime[k]` vale 0 se e solo se k è stato marcato, cioè k è un numero composto ($2 \leq k \leq n$); i primi due elementi dell'array, `isprime[0]` e `isprime[1]`, non sono utilizzati.

Viene fornita una funzione `int mark(char *isprime, int from, int to, int p)` che marca tutti i multipli di p appartenenti all'insieme $\{from, \ldots to - 1\}$. La funziona restituisce il numero di valori che sono stati marcati per la prima volta.

Il corpo principale del programma è costituito dalle istruzioni seguenti:

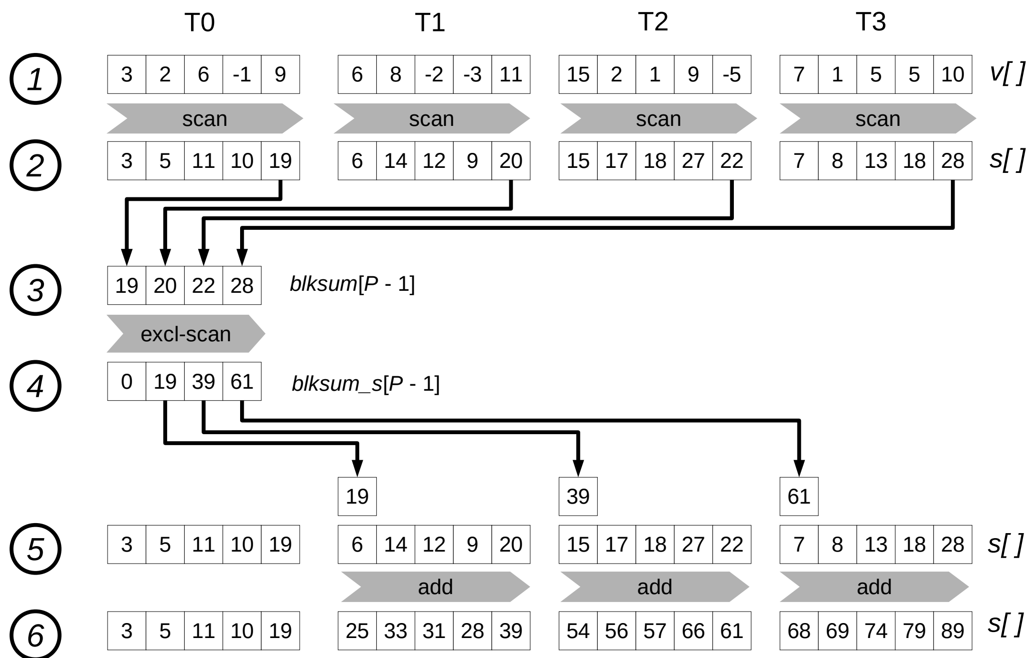
```
count = n - 1;
for (i=2; i*i <= n; i++) {
      if (isprime[i]) {
            count -= mark(isprime, i*i, n+1, i);
      }
}
```

Detto c il numero di primi appartenenti all'insieme $\{2, \dots n\}$ si pone inizialmente $c = n - 1$; ogni volta che si marca un valore per la prima volta, si decrementa c in modo da ottenere alla fine il risultato cercato.
Si noti che non è possibile parallelizzare il ciclo "for" del frammento di codice sopra, perché il contenuto dell'array `isprime[]` viene modificato dalla funzione `mark()`, e questo causa una *loop-carried dependency* non immediatamente visibile. Quello che si può fare è parallelizzare la fase di marcatura, cioè il corpo della funzione `mark()`, ad esempio dividendo l'intervallo $[i \times i, n]$ tra i thread in modo che ciascuno marchi solo i multipli di i nel proprio sottointervallo. A tale scopo si usi costrutto `omp parallel` determinando in modo opportuno gli estremi dei cicli eseguiti dai singoli thread (attenzione: è abbastanza complicato farlo nel modo corretto). Esistono anche altri modi un po' più semplici per distribuire la computazione tra i thread del pool. Vedremo nella prossima lezione un modo molto più semplice per delegare al compilatore la suddivisione delle iterazioni del ciclo tra i thread.

6. Scan inclusivo

Il file `omp-inclusive-scan.c` contiene una implementazione seriale dell'algoritmo per il calcolo dell'operatore *scan inclusivo* (somme prefisse) di un array `v[]` di n elementi; il risultato viene memorizzato in un secondo array `s[]`. Al termine dell'operazione si deve avere $s[i] = v[0] + v[1] + \dots + v[i]$ per ogni i.

Poiché nella programmazione OpenMP si assume generalmente che il numero P di core sia limitato ($P << n$), non è conveniente usare lo schema "ad albero" cui abbiamo accennato a lezione, che invece è adeguato per sistemi come le GPU in cui possiamo assumere di avere tanti core quanti elementi dell'array. Realizzeremo invece una strategia più semplice e più adeguata ad OpenMP; si faccia riferimento alla figura seguente, in cui assumiamo di avere $P = 4$ thread OpenMP.



1. Ogni thread opera su porzioni di `v[]` ed `s[]` i cui estremi devono essere determinati in modo opportuno.
2. Ciascun thread esegue la scan inclusiva della propria porzione di `v[]`, utilizzando un algoritmo sequenziale. il risultato viene memorizzato nella corrispondente porzione di `s[]`.
3. Il valore dell'ultimo elemento di ciascun sotto-vettore delle somme prefisse `s[]` viene copiato in un array temporaneo, chiamato ad esempio `blksum[]`, di dimensione *P*.
4. Il master applica l'operatore *scan esclusivo* a `blksum[]`, memorizzando il risultato su un nuovo array `blksum_s[]` (in realtà è anche possibile modificare direttamente il contenuto di `blksum[]`).
5. Ogni thread *p* somma il valore `blksum_[p]` a tutti gli elementi della propria porzione di somme prefisse `s[]`; il master (*p* = 0) dovrebbe sommare il valore zero, quindi in pratica non fa nulla.
6. Al termine della fase precedente, `s[]` contiene le somme prefisse di `v[]`