Corso di High Performance Computing

Esercitazione OpenMP del 24/3/2017

Moreno Marzolla

Ultimo aggiornamento: 2017/03/23

Per svolgere l'esercitazione è possibile collegarsi al server `disi-hpc.csr.unibo.it` tramite ssh, 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.

0. Familiarizzare con l'ambiente di lavoro

Per chi non l'ha ancora fatto, consiglio di prendere familiarità con l'ambiente di lavoro e con gli esempi visti a lezione (chi l'ha già fatto può passare direttamente all'esercizio successivo):

1. Scaricare i sorgenti dei programmi illustrati a lezione:
   ```
   wget www.moreno.marzolla.name/teaching/HPC/HPC1617.tar.gz
   tar xfz HPC1617.tar.gz
   cd HPC1617/
   ```
2. Compilare alcuni degli esempi OpenMP mostrati a lezione. E' fornito un Makefile per automatizzare la compilazione (il comando `make openmp` compila tutti i programmi OpenMP). Suggerisco di provare anche la compilazione manuale, usando i flag `-Wall -Wpedantic -std=c99` per abilitare i warning e forzare il rispetto dello standard C99 (chi vuole usare ANSI C deve usare `-ansi` al posto di `-std=c99`)
3. Eseguire i programmi con varie dimensioni del team di thread, settando la variabile d'ambiente `OMP_NUM_THREADS`:
   ```
   OMP_NUM_THREADS=3 ./omp-demo0
   ```

1. Somma degli elementi di un array [facile]

Scrivere un programma che calcola la somma dei valori contenuti in un array `a[]` di `double` (questo corrisponde all'operazione "sum reduce" di cui abbiamo parlato a lezione). La dimensione N dell'array può essere passata a riga di comando, oppure può essere definita *hardcoded* nel codice. Il programma deve sfruttare il parallelismo tramite il costrutto `omp parallel`, partizionando manualmente l'array tra i thread OpenMP. Procedere come segue:

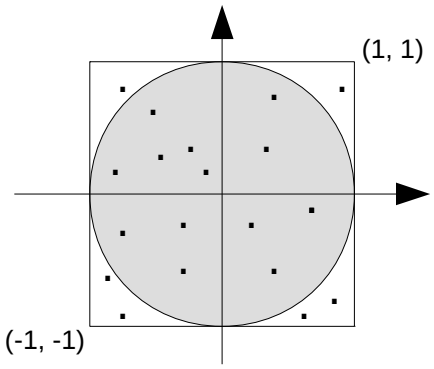
1. Iniziare realizzando una versione seriale del programma. Includere codice per inizializzare opportunamente l'array `a[]`, possibilmente *non* con valori casuali; conviene inizializzare l'array in modo deterministico per poter verificare la correttezza del risultato (es., con un valore costante, oppure con valori interi da 0 a N - 1 in modo che sia noto a priori quale deve essere il valore della somma dei valori);
2. Parallelizzare la versione seriale usando il costrutto `omp parallel` (inizialmente non usare `omp parallel for`). Detto P il numero di thread OpenMP, il programma deve partizionare l'array in P blocchi di dimensione approssimativamente uniforme. Ciascun thread determina gli estremi del sottovettore di cui è responsabile, usando il proprio ID e il valore di P, e calcola la somma della propria porzione di array. Le somme vengono memorizzate in un secondo array `s[]` di P elementi, in modo che ciascun thread scriva un elemento diverso senza causare race condition. Fatto questo, uno solo dei processi OpenMP (es., il master) calcola la somma dei valori in `s[]`, determinando così il risultato cercato.

3. Realizzare quindi una nuova versione della funzione di somma usando stavolta il costrutto `omp parallel for` e le direttive OpenMP per l'aggiornamento atomico della somma.

Si presti attenzione a gestire correttamente nel punto 2 il caso in cui *N* non sia un multiplo esatto di *P*; nel punto 3 ciò viene fatto automaticamente dal compilatore. Testare il programma anche nel caso in cui *N* sia minore del numero di thread *P*.

2. Calcolo del valore di π [facile]

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 inizia generando *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 di raggio unitario e centro nell'origine degli assi. Il rapporto *x* / *N* approssima il rapporto tra l'area del quadrato (che vale 4) e l'area del cerchio inscritto in esso. Poiché sappiamo che l'area di tale cerchio è π, 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 solo il costrutto `omp parallel` (non `omp parallel for`). L'implementazione deve operare come segue:

1. Sia *P* il numero di thread OpenMP utilizzati
2. Ciascun thread *p* invoca la funzione `generate_points()`, generando *N* / *P* punti e ponendo il risultato nella *p*-esima posizione di un array `inside[P]`; tale array va dichiarato al di fuori del blocco parallelo (tutte le variabili visibili all'inizio di un costrutto `omp parallel` sono condivise tra tutti i thread OpenMP all'interno del costrutto stesso).
3. Un solo processo (es, il master) somma i valori dell'array `inside[]`, determinando il numero totale di punti interni al cerchio e completando il calcolo di π

Prestare attenzione al fatto che *N* potrebbe non essere divisibile per *P*. Suggerisco di ignorare il problema nella prima stesura del programma, per affrontarlo solo in seguito.

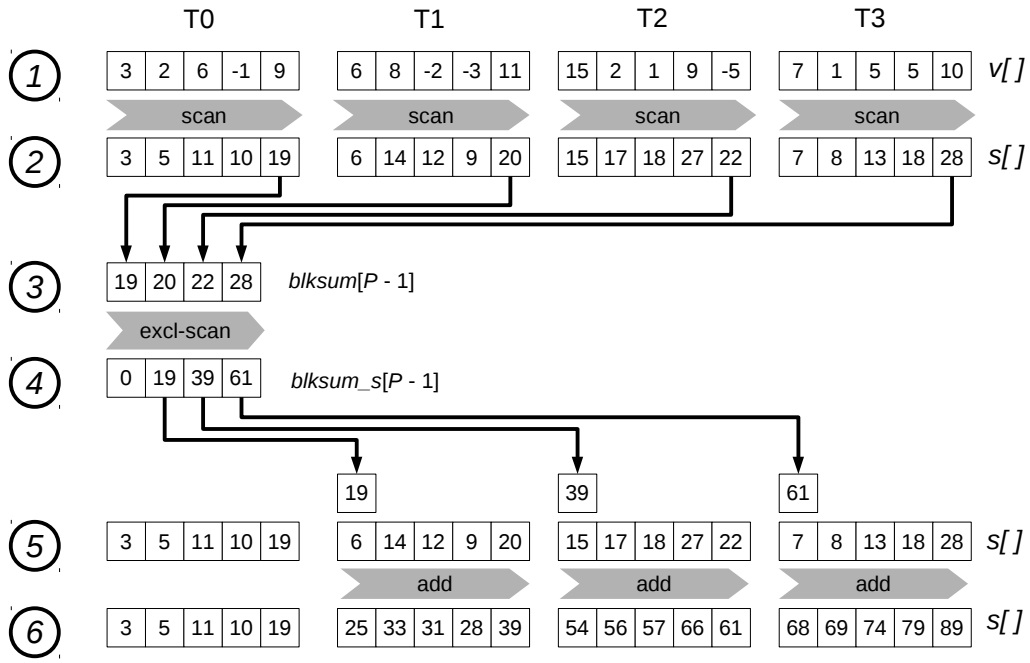
Creare poi una seconda versione del programma sfruttando tutti i costrutti OpenMP visti a lezione fino ad ora che si ritiene necessari.

3. Scan [medio]

Il file `omp-inclusive-scan.c` contiene una implementazione seriale di un algoritmo per il calcolo della *scan inclusiva* (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 $s[i] = v[0] + v[1] + \ldots + v[i]$ 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 $P = 4$.



1. L'array `v[]` è logicamente distribuito tra i thread OpenMP; 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 di *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 t somma il valore `blksum_[`t`]` a tutti gli elementi della propria porzione di somme prefisse `s[]`; il master ($t = 0$) dovrebbe sommare il valore zero, quindi non deve fare nulla.
6. Al termine della fase precedente, `s[]` contiene le somme prefisse di `v[]`

4. Decrittare un messaggio cifrato [medio]

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, buf, 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;
- `buf` è un puntatore ad un'area di memoria ampia *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 esattamente 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 sarà però illeggibile. Nel nostro caso, il messaggio in chiaro è una stringa di testo (stampabile con `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 offerto da OpenMP. In altri termini, il programma deve tentare tutte le possibili chiavi da “00000000” a “99999999”, fino a quando ottiene un messaggio decifrato che inizia con “0123456789”; trovata la chiave, il programma deve terminare in modo “pulito” stampando il messaggio decifrato, che è una citazione tratta da un vecchio film.

La prima soluzione che viene in mente consiste nel far uso di un costrutto `omp parallel`, all'interno del quale inserire il codice che esplora un opportuno sottoinsieme dello spazio delle chiavi per ciascun thread. Questo è concettualmente corretto, ma occorre ricordare che a lezione abbiamo detto 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; d'altra parte non vogliamo aspettare che tutti i thread abbiano esplorato l'intero spazio delle chiavi per terminare. E' 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.

5. Scan con computazione ad albero [medio]

Quando abbiamo parlato di pattern per la programmazione parallela abbiamo visto come sia possibile realizzare l'operatore *scan esclusivo* utilizzando una computazione strutturata "ad albero", con una fase ascendente (*up-sweep*) e una discendente (*down-sweep*). Sebbene a lezione non siano state fornite spiegazioni, nei lucidi a pagina 88 e 89 del documento http://www.moreno.marzolla.name/teaching/high-performance-computing/2016-2017/L03-patterns.pdf viene fornito il codice C per realizzare tali fasi; ricordare che quel codice funziona solo se la lunghezza dell'array di cui fare la scan è una potenza di due.

Realizzare una implementazione parallela dell'algoritmo sfruttando OpenMP; parallelizzare quanto più possibile di ciascuna fase usando i costrutti OpenMP che si ritengono più adeguati. Confrontare i tempi di esecuzione di questa implementazione con quelli del programma dell'esercizio 3.