Corso di High Performance Computing

Esercitazione OpenMP del 23/10/2018

Moreno Marzolla

Ultimo aggiornamento: 2018-10-19

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/ex2-openmp.zip
unzip ex2-openmp.zip
cd ex2-openmp/
```

1. Simulare la clausola "schedule(dynamic)" di OpenMP

A lezione abbiamo visto come sia possibile utilizzare la clausola `schedule(dynamic)` del costrutto `omp parallel for` per assegnare dinamicamente ogni iterazione di un ciclo “for” al primo thread OpenMP disponibile. Lo scopo di questo esercizio è quello di simulare la clausola `schedule(dynamic)` usando solo il costrutto `omp parallel` (non `omp parallel for`).

Il file `omp-dynamic.c` contiene una implementazione seriale di un programma che effettua le seguenti computazioni. Il programma crea e inizializza un array `vin[]` di n interi (il valore n può essere passato da riga di comando; se non viene specificato nulla viene usato un valore di default). Il programma crea un secondo array `vout[]`, sempre di lunghezza n, e ne definisce il contenuto in modo tale che si abbia `vout[i] = Fib(vin[i]))`, essendo Fib(k) il k-esimo numero della successione di Fibonacci (Fib(0) = Fib(1) = 1; Fib(k) = Fib(k - 1) + Fib(k - 2) se $k \geq 2$). Il calcolo dei numeri di Fibonacci viene fatto in modo volutamente inefficiente usando un algoritmo ricorsivo; questo serve a far sì che il tempo di calcolo dei `vout[i]` vari significativamente al variare di i.

Iniziare parallelizzando il ciclo “for” indicato nel codice con il commento “[TODO]” mediante il costrutto `omp parallel for`. Mantenendo fissa la lunghezza n degli array, osservare i tempi di esecuzione nei casi seguenti:

1. Parallelizzando il ciclo con la direttiva `#pragma omp parallel for`, in cui si usa lo schedule di default (che nel caso di GCC è lo schedule statico con dimensione del blocco n / numero_thread_OpenMP);
2. Parallelizzando il ciclo con uno schedule statico ma con un blocco di dimensione inferiore, ad es. 64; si può usare la direttiva `#pragma omp parallel for schedule(static,64)`;
3. Parallelizzando il ciclo con uno schedule dinamico, usando la direttiva `#pragma omp parallel for schedule(dynamic)`; ricordiamo che in questo caso la dimensione di default del blocco è 1.

Fatto ciò, si chiede di realizzare lo stesso comportamento del punto 3 (schedule dinamico con blocco di dimensione 1) utilizzando il costrutto `omp parallel` (non `omp parallel for`). Consiglio di procedere come segue: si crea un pool di thread OpenMP, e si utilizza una variabile

condivisa per indicare quale è l'indice del prossimo elemento di `vin[]` che deve ancora essere processato. Ogni thread acquisisce in modo atomico (usando le direttive OpenMP adatte) il prossimo elemento di `vin[]`, se presente, e lo elabora in parallelo.

2. Visita di LinkedList con task OpenMP

Il file `omp-linked-list.c` crea una lista concatenata in cui ciascun nodo p contiene due attributi interi n e *fibn*. I valori di n sono inizialmente definiti in modo pseudocasuale; successivamente, nella funzione `main()` è presente un ciclo *while* che visita tutti i nodi della lista, ponendo il valore *fibn* di ciascun nodo all'n-esimo numero di Fibonacci. Parallelizzare il ciclo di cui sopra sfruttando i task OpenMP.

3. Il Crivello di Eratostene

Il *crivello di Eratostene* è un algoritmo utilizzato per individuare i numeri primi appartenenti ad un dato intervallo, che normalmente è l'insieme $\{2, \ldots n\}$. 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 $2, \ldots n$:

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 della tabella (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 $\{2, \ldots n\}$ 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” con un banale costrutto `omp parallel for` , dato che il contenuto dell'array `isprime[]` viene prima letto e poi modificato dalla funzione `mark()` ad ogni iterazione del ciclo. Quello che invece si può fare è parallelizzare la fase di marcatura, cioè il corpo della funzione `mark()`, dividendo l'intervallo $[i \times i, n]$ tra i thread in modo che ciascuno marchi solo i multipli di i nel proprio sottointervallo.
Suggerisco di iniziare con la soluzione più semplice, che consiste nell'applicare il costrutto `omp parallel for` al ciclo contenuto nella funzione `mark()`; invito a specificare sempre in modo esplicito la visibilità delle variabili (usare la clausola `default(none)` per assicurarsi di non averne dimenticata alcuna).
Fatto ciò, si provi a parallelizzare il ciclo presente nel corpo della funzione `mark()` usando il costrutto `omp parallel` e determinando in modo opportuno gli estremi dei cicli eseguiti dai singoli thread (attenzione: è abbastanza complicato farlo nel modo corretto).

4. Area dell'insieme di Mandelbrot

Come visto a lezione, l'insieme di Mandelbrot corrisponde alla parte nera nella Figura 1.

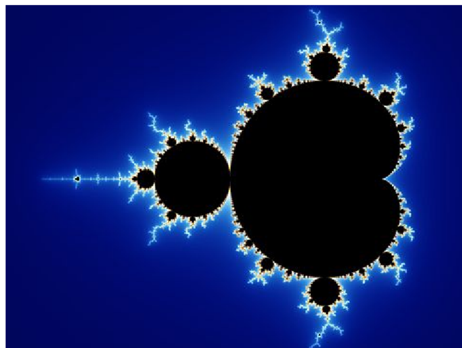

Figura 1: Insieme di Mandelbrot

Nel file `omp-mandelbrot-area.c` viene fornita la versione seriale di un programma che stima l'area dell'insieme di Mandelbrot. Il sorgente include le istruzioni per la compilazione e l'esecuzione. È interessante osservare che il valore esatto dell'area dell'insieme di Mandelbrot non è noto, ma sono note delle approssimazioni.

Compilare ed eseguire il programma per verificarne il funzionamento. Modificare quindi il programma per sfruttare il parallelismo OpenMP sfruttando le direttive e le clausole che si ritengono necessarie.
Se avanza tempo, studiare sperimentalmente se e come la scelta del tipo di scheduling (*static* o *dynamic*) e della dimensione dei blocchi (*chunksize*) influisca sul tempo di esecuzione del programma. Non è ovviamente possibile esaminare tutti i valori di *chunksize*, quindi si consiglia di usare un insieme limitato di valori per avere una idea grossolana del comportamento del programma.

5. Prodotto matrice-matrice

Il file `omp-matmul.c` contiene l'implementazione seriale dell'algoritmo *cache-efficient* per calcolare il prodotto matrice-matrice che abbiamo visto in una delle precedenti lezioni.

1. Analizzare la funzione `matmul_transpose` e parallelizzarla ove indicato usando le direttive OpenMP appropriate.
2. Valutare sperimentalmente lo speedup della versione parallela, fissata una dimensione delle matrici che consenta di osservare tempi significativi.