Corso di High Performance Computing

Esercitazione OpenMP del 20/10/2020

Moreno Marzolla

Ultimo aggiornamento: 2021-05-05

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

Alcuni dei programmi discussi negli esercizi producono immagini in formato PPM (*Portable Pixmap*) o PGM (*Portable Graymap*), che le macchine Windows dei laboratori potrebbero non essere in grado di visualizzare. Per convertire tali immagini in un formato diverso (ad esempio, JPEG) è possibile lanciare sul server il comando

```
convert image.ppm image.jpeg
```

e copiare il file risultante sul proprio PC usando Winscp o simili.

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. Parallelizzazione di loop

Il file `omp-loop.c` contiene delle funzioni con cicli che iterano su array o matrici. Per ciascuna funzione viene presentata una versione seriale. Applicare le tecniche di parallelizzazione dei loop viste a lezione per realizzare la versione parallela corrispondente; nel sorgente sono forniti dei suggerimenti su come procedere caso per caso.
La funzione `main()` controlla la correttezza dei risultati confrontando l'output della versione seriale di ciascuna funzione con la corrispondente versione parallela. Si presti attenzione al fatto che il controllo non è esaustivo, per cui potrebbero esserci errori non rilevati dai test.

3. MergeSort

Il file `omp-mergesort.c` contiene una implementazione ricorsiva dell'algoritmo MergeSort; si faccia riferimento ai commenti nel sorgente per i dettagli di funzionamento, che comunque non dovrebbero destare sorprese dato che è già stato visto nel corso di Algoritmi e Strutture Dati.
Parallelizzare il programma sfruttando i task OpenMP, facendo in modo che ogni chiamata ricorsiva venga delegata ad un task OpenMP. Verificare se la versione parallela è più o meno veloce di quella seriale, variando il numero di thread OpenMP. Utilizzare una lunghezza appropriata per l'array, in modo da ottenere tempi di esecuzione non troppo brevi, che sarebbero soggetti ad una eccessiva variabilità.

4. Ray tracing

Il file `c-ray.c`. contiene l'implementazione di un semplice programma di ray-tracing scritto da John Tsiombikas e rilasciato con licenza GPLv2. Le istruzioni per la compilazione e l'uso sono incluse nei commenti nel codice; il programma può anche essere compilato tramite il Makefile fornito. Sono inclusi due file di input: `sphfract.small.in` e `sphfract.big.in`. Entrambi descrivono un oggetto frattale: nel primo file ci sono meno dettagli, e quindi il tempo di elaborazione è inferiore.
Per produrre l'immagine rappresentata da `sphfract.small.in` si usa il comando:

```
./c-ray < sphfract.small.in > img.ppm
```

Viene prodotta una immagine `img.ppm` con risoluzione 800×600 che è possibile modificare da riga di comando (si vedano i commenti nel file `c-ray.c` per i dettagli). Per visualizzare l'immagine su Windows è utile convertirla in formato JPEG con il comando:

```
convert img.ppm img.jpeg
```

prima di trasferirla sul proprio PC per la visualizzazione. I file di input producono entrambi una immagine simile alla Figura 1.

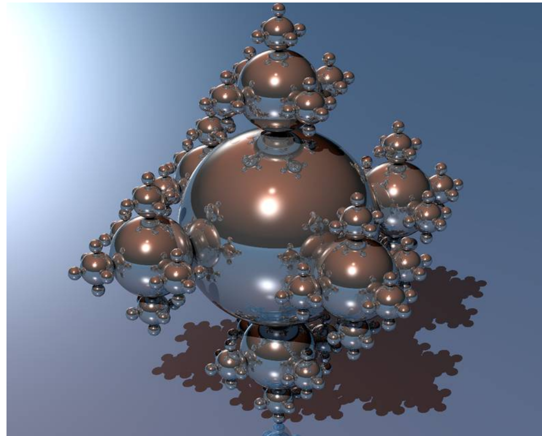

Figura 1: Immagine generata dal programma c-ray

Parallelizzare la funzione `render()` usando le direttive OpenMP appropriate nei punti del codice che si ritengono opportuni. Si tenga presente che il programma seriale è ben strutturato: in particolare, le funzioni non modificano variabili globali, quindi non ci sono dipendenze "nascoste" da gestire. Suggerisco di usare l'immagine `sphfract.small.in` per fare le prove, visto che richiede meno tempo per essere calcolata (l'altra immagine richiede circa 90 secondi usando 12 thread OpenMP!).

Se avanza tempo, misurare lo *speedup* e l'efficienza del programma ottenuto.

5. La mappa del gatto

La *mappa del gatto* di Arnold è una funzione caotica proposta negli anni '60 dal matematico russo Vladimir Igorevich Arnold (1937–2010), che ne ha illustrato il funzionamento tramite l'immagine stilizzata di un gatto (da cui il nome). Nella sua versione discreta, la funzione trasforma una immagine P di dimensione $N \times N$ in una nuova immagine P' delle stesse dimensioni. Per ogni $0 \leq x < N$, $0 \leq y < N$, il pixel di coordinate (x, y) in P viene collocato nella posizione (x', y') di P' dove:

$$x' = (2x + y) \bmod N, \qquad y' = (x + y) \bmod N$$

(mod è l'operatore modulo, corrispondente all'operatore `%` del linguaggio C). Si può assumere che le coordinate (0, 0) indichino il pixel in alto a sinistra e le coordinate $(N - 1, N - 1)$ quello in basso a destra, in modo da poter indicizzare l'immagine come se fosse una matrice in linguaggio C.

La mappa del gatto ha proprietà sorprendenti. Applicata ad una immagine ne produce una versione molto distorta. Applicata nuovamente a quest'ultima immagine, ne produce una ancora più distorta, e così via. Tuttavia, dopo un certo numero di iterazioni (il cui valore dipende da N, ma che in ogni caso è sempre minore o uguale a $3N$) ricompare l'immagine di partenza! (si veda la Figura 2).

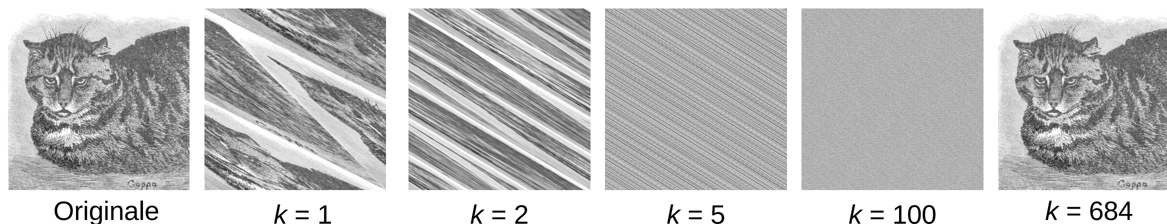

Figura 2: Alcune immagini ottenute iterando la mappa del gatto k volte

Il tempo minimo di ricorrenza per l'immagine `cat1368.pgm` di dimensione 1368×1368 fornita come esempio è 36; iterando k volte della mappa del gatto si otterrà l'immagine originale se e solo se k è multiplo di 36 (684 lo è). Non è nota alcuna espressione analitica che leghi il tempo minimo di ricorrenza alla dimensione N dell'immagine.
Viene fornita una implementazione seriale di un programma che calcola la k-esima iterata della mappa del gatto. Il programma viene invocato specificando sulla riga di comando il numero di iterazioni k. Il programma legge da standard input una immagine in formato PGM (Portable GrayMap), e produce su standard output una nuova immagine ottenuta applicando k volte la mappa del gatto. Ad esempio:

```
./omp-cat-map 100 < cat1368.pgm > cat1368-100.pgm
```

genera un file `cat1368-100.pgm` contenente l'immagine che si ottiene iterando $k = 100$ volte la mappa del gatto sull'immagine `cat1368.pgm`. Per visualizzare il risultato sotto Windows può essere necessario convertire le immagini PGM in un altro formato, ad esempio JPEG:

```
convert cat1368-100.pgm cat1368-100.jpeg
```

Si richiede di parallelizzare il corpo della funzione `cat_map()` mediante OpenMP

Estensione

La funzione `cat_map()` fornita si basa sullo schema seguente:

```
for (i=0; i<k; i++) {
      for (y=0; y<N; y++) {
            for (x=0; x<N; x++) {
                  "applica la mappa al punto di coord. (x, y)"
            }
      }
}
```

In questo caso si può applicare la tecnica di *loop interchange* vista a lezione per riscrivere l'iterazione nel modo seguente:

```
for (y=0; y<N; y++) {
      for (x=0; x<N; x++) {
            for (i=0; i<k; i++) {
                  "applica la mappa al punto di coord. (x, y)"
            }
      }
}
```

In questa seconda versione è possibile parallelizzare uno dei due cicli più esterni (o entrambi, usando la clausola `collapse(2)`). Parallelizzare questa seconda versione, e confrontarne le prestazioni con la precedente.

Per chi vuole approfondire

Qual è il tempo minimo di ricorrenza dell'immagine `cat1024.pgm` di dimensioni 1024×1024? Per rispondere è possibile procedere per tentativi iterando la mappa del gatto 1, 2, 3, … volte, controllando di volta in volta se si ottiene l'immagine di partenza. Esiste però un metodo migliore che non richiede di conoscere l'immagine di input ma solo la sua dimensione. L'idea è la seguente: supponiamo che un pixel dell'immagine abbia tempo minimo di ricorrenza 15, e un altro pixel abbia tempo minimo di ricorrenza 21. Il tempo minimo necessario affinché *entrambi* i punti ritornino nelle rispettive posizioni iniziale è il minimo comune multiplo di 21 e 15 (cioè 105). Estendendo il ragionamento, il tempo minimo di ricorrenza dell'immagine è il minimo comune multiplo di tutti i tempi di ricorrenza dei punti.

Il file `omp-cat-map-rectime.c` contiene lo scheletro di un programma per il calcolo del tempo minimo di ricorrenza di una immagine di dimensioni $N \times N$, con N parametro di input. Viene fornita la funzione per calcolare il minimo comune multiplo di due interi. Completare il programma e parallelizzarlo usando le direttive OpenMP che si ritengono appropriate.

La tabella seguente mostra i tempi minimi di ricorrenza per alcuni valori della dimensione N dell'immagine.

N	Tempo minimo di ricorrenza
64	48
128	96
256	192
512	384
1368	36

La Figura 3 mostra il tempo minimo di ricorrenza per dimensioni dell'immagine fino a 1024×1024; nonostante l'andamento irregolare, si può osservare come i valori tendano ad allinearsi lungo alcune rette.

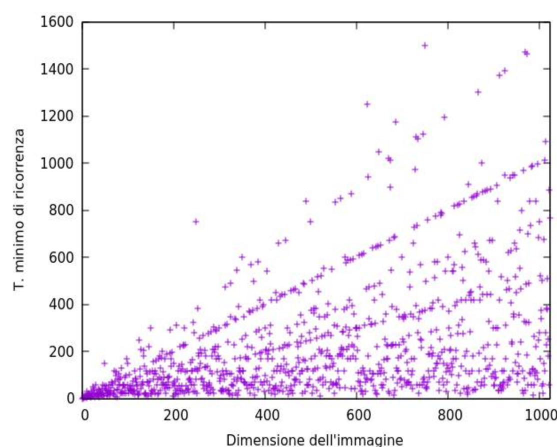

Figura 3: Tempo minimo di ricorrenza in funzione della dimensione dell'immagine

6. Area dell'insieme di Mandelbrot

Come visto a lezione, l'insieme di Mandelbrot corrisponde alla parte nera nella Figura 4.

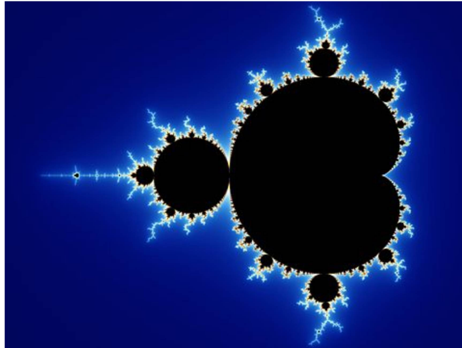

Figura 4: Insieme di Mandelbrot

Nel file `omp-mandelbrot-area.c` viene fornita la versione seriale di un programma che stima l'area dell'insieme di Mandelbrot. Il programma genera implicitamente una rappresentazione bitmap dell'insieme di Mandelbrot, e calcola il rapporto tra il numero di pixel dell'insieme e il numero di pixel totali dell'immagine. L'area A dell'insieme di Mandelbrot è quindi:

$$A \approx \frac{\text{n. pixel neri}}{\text{n. pixel totali}} \times \text{area rettangolo}$$

dove "area rettangolo" è l'area della regione rettangolare rappresentata dalla bitmap, calcolata in modo esatto sulla base delle coordinate degli intervalli orizzontali e verticali della regione mostrata. È interessante osservare che il valore esatto dell'area dell'insieme di Mandelbrot non è noto, ma sono note delle approssimazioni.

Il sorgente include le istruzioni per la compilazione e l'esecuzione. Eseguire il programma per verificarne il funzionamento. Modificarlo quindi 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.