Corso di High Performance Computing

Esercitazione OpenMP del 31/3/2017

Moreno Marzolla

Ultimo aggiornamento: 2017/03/30

Per svolgere l'esercitazione è possibile collegarsi al server `disi-hpc.csr.unibo.it` tramite ssh, usando come *username* il proprio indirizzo mail istituzionale completo, e come password la propria password istituzionale (cioè quella usare 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 Linux può lavorare localmente, dopo aver installato il compilatore.

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

```
wget http://www.moreno.marzolla.name/teaching/HPC/ex3-openmp.zip
unzip ex3-openmp.zip
cd ex3-openmp/
```

Alcuni dei programmi discussi negli esercizi producono immagini in formato PPM (*Portable Pixmap*) o PGM (*Portable Graymap*), che le macchine Windows dei laboratori non sono in grado di visualizzare. È necessario convertire tali immagini in un formato diverso (ad esempio, PNG) dando sul server un comando come:

```
convert image.ppm image.png
```

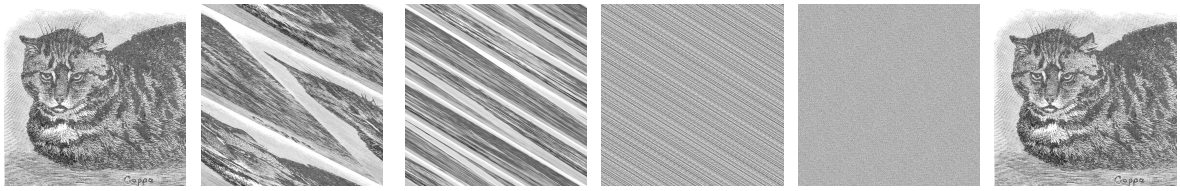
per poi copiare il file risultante sul proprio PC usando il programma Winscp (già installato).

1. Mappa del gatto

La *mappa del gatto di Arnold* è una funzione caotica proposta negli anni '60 dal matematico russo Vladimir Igorevich Arnold; essa prende il nome dal fatto che Arnold ne ha illustrato il funzionamento tramite l'immagine stilizzata di un gatto.
La mappa del gatto trasforma una immagine quadrata P di dimensione $N \times N$ in una nuova immagine P' delle stesse dimensioni: il pixel di coordinate (x, y) in P, $0 \le x < N$, $0 \le y < N$, viene collocato nella posizione (x', y') di P' dove:

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

(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 rappresentare l'immagine come una matrice.
La mappa del gatto ha proprietà sorprendenti. Applicata ad una immagine, se ne ottiene una versione molto distorta. Applicando nuovamente la mappa a quest'ultima immagine, se ne ottiene un'altra ancora più distorta, e così via. Tuttavia, dopo un certo numero di iterazioni (il cui valore dipende da N ma in ogni caso risulta sempre minore o uguale a $3N$) ricompare l'immagine di partenza!



Originale $K = 1$ $K = 2$ $K = 5$ $K = 100$ $K = 684$

Viene fornito lo scheletro di un programma che calcola la *k*-esima iterata della mappa del gatto; nel programma manca l'implementazione della funzione `cat_map(img, k)` che modifica l'immagine puntata da `img` per renderla uguale a quella che si otterrebbe applicando per *k* volte la mappa del gatto; se $k = 0$ l'immagine non deve essere modificata.
Il programma viene invocato specificando sulla riga di comando il numero di iterazioni *k*. Il programma legge una immagine in formato PGM da standard input, e produce una nuova immagine su standard output ottenuta applicando *k* volte la mappa del gatto. Occorre ricordarsi di redirezionare lo standard output su un file, come indicato nelle istruzioni nel sorgente.

1. Completare il programma implementando la funzione `cat_map()`, iniziando con una implementazione seriale. La funzione è già parzialmente realizzata; occorre solamente definire ulteriori variabili se necessarie, e scrivere il codiceche copia i pixel (Byte) dell'immagine corrente nella nuova posizione dell'immagine successiva.
2. Provare il programma seriale usando il file `cat.pgm` come input. Calcolare l'immagine ottenuta dopo 1, 10, 100, e 684 iterazioni; se l'implementazione è corretta, dopo 684 iterazioni si ottiene l'immagine di partenza.
3. Modificare la versione seriale della funzione `cat_map()` sviluppata al punto 1 per sfruttare il parallelismo tramite OpenMP.
4. Calcolare lo speedup e l'efficienza della versione parallela, usando `cat.pgm` come input e mantenendo il numero di iterazioni costante a 684. Calcolare i tempi di esecuzione al variare del numero di thread OpenMP, ripetendo le misure un certo numero di volte per poi calcolare il tempo medio come spiegato a lezione.

2. 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.

1. Parallelizzare il programma sfruttando i task OpenMP.
2. Verificare se la versione parallela è più o meno veloce di quella seriale. A tale scopo può essere utile effettuare i test con array di dimensione maggiore; è possibile specificare la lunghezza dell'array da ordinare sulla riga di comando.

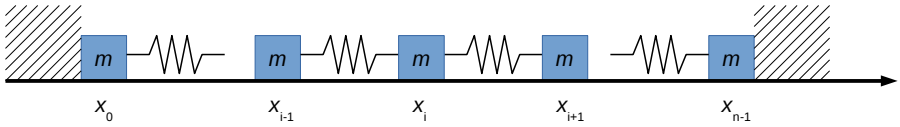
3. Visita LinkedList

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.

1. Parallelizzare il ciclo di cui sopra sfruttando i task OpenMP.
2. Verificare se la versione parallela è più o meno veloce di quella seriale.

4. Oscillatori accoppiati

Si considerino n punti di massa m disposti lungo una retta alle coordinate x_0, x_1, … x_{n-1}. Masse adiacenti sono collegate da una molla di costante elastica k e lunghezza a riposo L. Il primo e l'ultimo punto (quelli in posizione x_0 e x_{n-1}) occupano una posizione fissa e non possono muoversi.

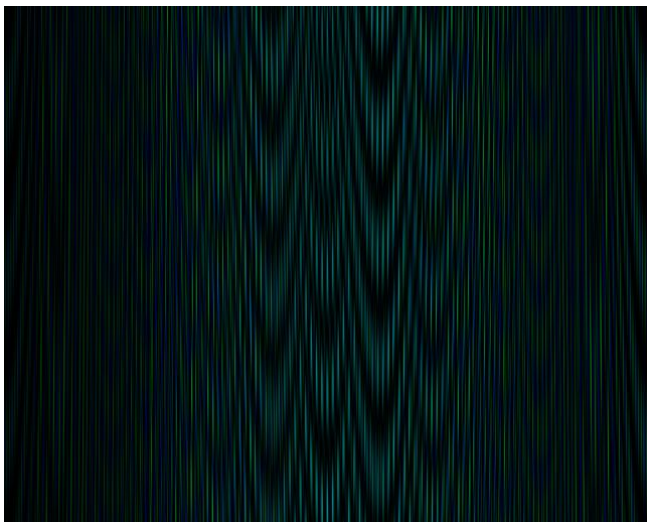


Se all'istante iniziale una delle molle non è a riposo, si innescheranno delle oscillazioni che proseguiranno indefinitamente se le masse scivolano senza attrito. Tralasciando i dettagli di fisica (che comunque dovrebbero essere alla vostra portata dato che le uniche cose che servono sono la seconda legge della dinamica di Newton $F = ma$, e il principio di Hooke che afferma che una molla di costante k compressa di una quantità Δx esercita una forza pari a $k\Delta x$), vogliamo sviluppare un programma che, date le posizioni e le velocità iniziali delle masse, calcoli posizioni e velocità al tempo $t > 0$. Il programma si basa su un algoritmo iterativo che partendo dalle posizioni e velocità delle masse al tempo t, determina le nuove posizioni e le nuove velocità al tempo $t + \Delta t$. In particolare, la funzione `step(double *x, double *v, double *xnext, double *vnext, int n)` calcola posizione `xnext[i]` e velocità `vnext[i]` della massa i-esima al tempo $t + \Delta t$, $0 \le i < n$, date le posizioni `x[i]` e velocità `v[i]` al tempo t.

1. Per ogni $i = 1, \ldots n - 2$ si calcola la forza F_i che agisce sulla massa i-esima come $F_i := k(x_{i-1} - 2x_i + x_{i+1})$. Le masse 0 e $n - 1$ rimangono in posizione fissa, quindi le forze che agiscono su di esse sono ininfluenti e non vengono calcolate.
2. Per ogni $i = 1, \ldots n - 2$ si calcola la nuova velocità v'_i della massa i-esima al tempo $t + \Delta t$ come $v'_i := v_i + (F_i/m)\Delta t$. Le masse 0 e $n - 1$ restano fisse, quindi la loro velocità sarà sempre zero.
3. Per ogni $i = 1, \ldots n - 2$ si calcola la nuova posizione x'_i della massa i-esima al tempo $t + \Delta t$ come $x'_i := x_i + v'_i \Delta t$. Le masse 0 e $n - 1$ restano ferme quindi la loro posizione al tempo $t + \Delta t$ sarà uguale a quella al tempo t: $x'_0 = x_0$. $x'_{n-1} = x_{n-1}$.

Il file `omp-coupled-oscillators.c` contiene lo scheletro di un programma che invoca ripetutamente la funzione `step()` per calcolare l'evoluzione di un insieme di oscillatori accoppiati. Il programma produce una immagine bidimensionale in cui ogni riga mostra le energie delle $n - 1$ molle in ogni istante di tempo.

Scopo di questo esercizio è implementare la funzione `step()` in base ai passi descritti in precedenza, iniziando con una versione seriale; le costanti da usare (k, m, Δt) sono già definite nelle costanti `k`, `m` e `dt` all'inizio del sorgente. Una volta realizzata una prima versione seriale, verificandone il funzionamento, è richiesto di applicare i costrutti OpenMP appropriati per crearne una versione parallela. Se il programma funziona correttamente dovrebbe produrre un file `coupled-oscillators.ppm` contenente una immagine simile alla seguente:



5. Scan usando le primitive di sincronizzazione

L'esercizio 3 della prima esercitazione OpenMP chiedeva di realizzare l'operazione *scan inclusiva* in tre fasi: nella prima fase ciascun processo OpenMP effettua una scan di una porzione di array; nella seconda fase il master effettua una *scan esclusiva* degli ultimi valori calcolati nel passo precedente; nell'ultima fase i processi OpenMP sommano alla propria porzione di array il valore appropriato risultante dalla scan esclusiva fatta dal master.

Nel file `omp-inclusive-scan.c` è presente una possibile soluzione basata sulle conoscenze che avevamo la scorsa settimana. Modificare il codice inserendo le tre fasi in una unica sezione parallela (`#pragma omp parallel`), e sfruttando i costrutti per la sincronizzazione visti questa settimana per fare in modo che la seconda fase sia fatta solo dal master, forzando la sincronizzazione tra i processi nei punti appropriati.