Nome e Cognome _____ Matricola _____

Corso di *High Performance Computing*
Ingegneria e Scienze Informatiche—Università di Bologna

Prova Scritta, 29/6/2017

- La prova dura 60 minuti
- Durante la prova non è consentito consultare libri, appunti o altro materiale.
- Non è consentito l'uso di dispositivi elettronici (ad esempio, cellulari, tablet...), né interagire in alcun modo con gli altri studenti pena l'esclusione dalla prova, che potrà avvenire anche dopo il termine della stessa.
- Le risposte devono essere scritte **a penna** su questi fogli, in modo **leggibile**. Le parti illeggibili o scritte a matita saranno ignorate.
- Eventuali altri fogli possono essere utilizzati per la brutta copia ma non devono essere consegnati e non verranno valutati.
- I voti saranno pubblicati su AlmaEsami e ne verrà data comunicazione all'indirizzo mail di Ateneo (@studio.unibo.it).
- I voti restano validi fino alla sessione d'esame di gennaio/febbraio 2018 inclusa. Dopo tale data tutti i voti in sospeso saranno persi.

NON SCRIVERE NELLA TABELLA SOTTOSTANTE

D. 1	D. 2	D. 3	D. 4
/ 8	/ 8	/ 8	/ 8

Domanda 1. Descrivere la differenza tra *task parallelism* e *data parallelism*.

Osservazione. Occorre prestare attenzione a non confondere la programmazione SIMD con la programmazione data-parallel. Nella programmazione SIMD la stessa *operazione elementare* (es., somma, prodotto) si applica a dati diversi. Nella programmazione data-parallel la stessa *computazione* si applica a dati diversi, ma non è detto che la computazione esegua le stesse identiche operazioni su tutti i dati. L'esempio classico è l'insieme di Mandelbrot: nell'approccio data-parallel ciascun processore calcola una porzione dell'insieme applicando la stessa funzione, ma il comportamento della funzione dipende dai dati di input (in particolare, dipende da quanto tempo ci mette l'iterata di ciascun punto ad "uscire" dal cerchio di raggio due). Un altro esempio è la decifratura di un messaggio cifrato usando una tecnica brute-force: in questo caso i dati (lo spazio delle possibili chiavi) viene distribuito tra i processi che eseguono la stessa computazione ma non necessariamente la stessa identica sequenza di istruzioni, dato che ciascun tentativo richiederà una sequenza di operazioni elementari diversa in base alla chiave usata.

Domanda 2. Si consideri il seguente frammento di codice in linguaggio C:

```
b[0] = f(0);
c[0] = g(42);
for (i = 1; i<n; i++) {
   b[i] = f( i );
   c[i] = g( b[i-1] );
}
```

Assumendo che:

- gli array `b[]` e `c[]` abbiano *n* elementi ciascuno;
- tutte le variabili e le funzioni siano definite in modo opportuno;
- le funzioni `f()` e `g()` restituiscano valori che dipendono solo dai rispettivi parametri;

ristrutturare il ciclo in modo che produca lo stesso risultato ma non siano presenti *loop-carried dependencies*.

Risposta. Una possibile soluzione (non è l'unica) è la seguente; le parti modificate sono evidenziate.

```
b[0] = f(0);
c[0] = g(42);
c[1] = f(b[0]);
for (i = 1; i<n-1; i++) {
   b[i] = f( i );
   c[i+1] = g( b[i] );
}
b[n-1] = f(n-1);
```

Domanda 3. Vogliamo effettuare il rendering di un filmato composto da circa 50'000 frame aventi risoluzione 2048×1024 pixel ciascuno. Viene impiegato un algoritmo di *ray-tracing*: ogni frame è descritto in termini di unioni e intersezioni di primitive geometriche elementari (sfere, piani, coniche...). L'algoritmo di ray tracing proietta un raggio dall'osservatore verso ciascun pixel dell'immagine, e determina se il raggio interseca una delle primitive geometriche. Se il raggio colpisce una superficie riflettente e/o traslucida, vengono generati raggi secondari che devono essere elaborati in modo simile.

Supponiamo di disporre di due sistemi hardware:

[MC] Un singolo server dotato di un processore "potente" (es., un Intel Xeon di ultima generazione) con 16 core fisici e 64 GB di RAM;
[RPI] Un cluster di 128 Raspberry PI modello 3 connessi mediante Fast Ethernet (100 Mbit/s); ciascun Raspberry PI è equipaggiato con un processore ARM quad core con 1 GB di memoria RAM. Assumere che il processore di un Raspberry PI sia molto meno potente di quello del server di cui al punto precedente.

Descrivere le potenzialità e i limiti di ciascuno dei due sistemi hardware se impiegati a risolvere il problema precedente. È possibile fare ulteriori assunzioni oltre a quelle indicate sopra, purché ragionevoli e indicate nella risposta.

Risposta. Prima di rispondere alla domanda è importante inquadrare bene l'applicazione considerata. Abbiamo avuto a che fare con il problema del rendering 3D in una delle esercitazioni su OpenMP: scopo dell'esercitazione era la parallelizzazione di un semplice programma di ray-tracing che supportava la sfera come unica primitiva geometrica. La struttura di quel semplice programma era molto simile a quella dei software più sofisticati; in particolare, la natura della computazione è essenzialmente di tipo embarassingly parallel e CPU-bound. È embarassingly parallel perché ogni frame può essere calcolato indipendentemente dagli altri, e anche ogni pixel di ciascun frame può essere calcolato indipendentemente dagli altri pixel dello stesso frame. Non sono necessarie comunicazioni tra i processi; le uniche operazioni di I/O sono quelle relative alla lettura delle descrizioni delle scene e il salvataggio delle singole immagini risultanti. L'applicazione è CPU-bound e non memory-bound perché gli unici dati che vengono letti sono le descrizioni delle primitive geometriche di cui è composta la scena; non è quindi molto verosimile supporre che l'accesso alla memoria sia un fattore limitante in questo genere di computazioni.
Dal testo dell'esercizio si può intuire come la natura ricorsiva della computazione possa produrre problemi di sbilanciamento del carico in presenza di zone con un numero elevato di elementi trasparenti o traslucidi che causano la generazione di ulteriori raggi riflessi/rifratti. I problemi di sbilanciamento del carico possono essere affrontati con le strategie viste a lezione, ossia riducendo la granularità della computazione ad esempio assegnando a ciascun processore una porzione più piccola dell'immagine, o adottando un paradigma di tipo master/worker. Qualcuno ha fatto l'ipotesi che ogni raggio coincida con un thread di esecuzione: questa soluzione, per quanto non impossibile, è comunque poco verosimile.
Sulla base delle considerazioni precedenti, entrambe le architetture si prestano in vario modo alla risoluzione del problema. Nel caso MC appare naturale distribuire il calcolo dei colori dei pixel ai 16 core, in modo che ogni core calcoli una porzione di un singolo frame, oppure calcoli un frame separato (l'ampia quantità di memoria a bordo rende plausibile questa seconda possibilità).
Nel caso RPI è possibile adottare una strategia di tipo *scatter-gather* oppure *master-worker*. Nella strategia *scatter-gather*, ciascun Raspberry PI calcola una porzione di immagine pari a circa 1/128 dell'immagine totale, e un nodo coordinatore (il master) assembla le porzioni per salvare i frame completi su qualche dispositivo esterno di memorizzazione. Nella strategia *master-worker*, il processo master invia la descrizione di un intero frame al primo nodo disponibile, che calcola l'intera immagine risultante. La dimensione di una immagine, valutata grossolanamente come

$2000 \times 1000 \times 3$ Byte/pixel = circa 6 MB, è ampiamente entro i limiti della capacità di memoria di un Raspberry, anche tenuto conto della porzione occupata dal Sistema Operativo, dal software di rendering e dalla descrizione del frame. Si noti che in ogni caso, sia che un Raspberry debba calcolare una porzione di 1/128 di immagine, sia che debba calcolare una immagine intera, è possibile una ulteriore decomposizione della computazione tra i quattro core di ciascun processore.
Possiamo schematizzare gli aspetti positivi e negativi di ciascuna architettura nella tabella seguente:

	Pro	**Contro**
MC	• Se si dispone di una versione seriale del codice di rendering, svilupparne una versione parallela a memoria condivisa potrebbe risultare semplice • Una applicazione di tipo CPU-bound come questa trae beneficio da un processore potente	• Costo potenzialmente elevato • L'efficienza dal energetica di questa soluzione (misurata, ad esempio, in Joule/frame) potrebbe risultare largamente inferiore a quella del sistema RPI, ma servono maggiori dettagli per valutare questo aspetto
RPI	• Costo potenzialmente contenuto • Soluzione che può, almeno in linea di principio, scalare con la semplice aggiunta di nuovi nodi • Potenzialmente efficiente dal punto di vista energetico, ma servono maggiori dettagli per valutare questo aspetto	• Realizzare una versione parallela a memoria distribuita dell'applicazione di rendering potrebbe essere laborioso • L'impatto della rete di interconnessione va valutato con attenzione. L'applicazione è di tipo embarassingly parallel, per cui è ragionevole aspettarsi che la rete venga usata solo per comunicare ai nodi la descrizione dei frame e per recuperare l'immagine finale. In base al tempo richiesto per calcolare un frame si può stimare la banda di rete richiesta e confrontarla con quella fornita.

Domanda 4. Lo *speedup* $S(p)$ di una applicazione parallela è dato dal rapporto tra il tempo di esecuzione della versione seriale e il tempo di esecuzione della versione parallela con p processori Normalmente si ha $S(p) \leq p$; in quali casi si può avere $S(p) > p$?

Risposta. È raro, ma non impossibile, osservare uno speedup superlineare. Quando ciò si verifica, le cause possono essere almeno due:

1. All'aumentare di p, la dimensione del dominio locale a ciascun processore potrebbe diminuire a tal punto da consentire una migliore località dei dati che sfrutta meglio le cache;
2. Alcune architetture offrono un parallelismo di tipo eterogeneo: ad esempio, il processore Cell usato dalla vecchia Playstation 3 contiene una CPU affiancata da un certo numero di coprocessori vettoriali specializzati. In questi casi, l'applicazione seriale che fa uso della CPU può risultare sensibilmente più lenta di una versione parallela che deleghi ai coprocessori la maggior parte del lavoro, usando la CPU unicamente come coordinatore.