Nome e Cognome _____ Matricola _____

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

Prova Scritta, 14/1/2020

- 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 verranno valutati.
- Si risponda in modo sintetico ma il più possibile esaustivo.
- 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 **settembre 2020** 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.
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 eseguita usando p processori. Normalmente si ha $S(p) \leq p$; in quali casi si può avere $S(p) > p$? Discutere e motivare la risposta.

Risposta: Si potevano discutere i seguenti punti

- Hardware eterogeneo
- Riduzione della dimensione del dominio che consente di sfruttare meglio le cache
- Nel caso delle architetture SIMD, riduzione del numero di istruzioni eseguite

Domanda 2.
Si consideri il seguente frammento di codice, che realizza una computazione iterativa di tipo *stencil*. Ad ogni iterazione del ciclo *do-while* si usano i valori di un dominio di dimensioni $N \times N$ memorizzati in `phi[cur][][]` per calcolare nuovi valori `phi[next][][]`. I valori sul bordo dei domini non cambiano, e si assuma che siano stati inizializzati in modo opportuno; i domini non si assumono ciclici. Il ciclo *do-while* termina quando le differenze in valore assoluto tra i vecchi e i nuovi valori risultano tutte minori o uguali ad una certa costante `epsilon`.

```
   #define N 10000
   double phi[2][N][N], maxdelta;
   const double epsilon = 1.0e-10;
   int cur = 0, next = 1;

   /* ...inizializzazioni varie non mostrate... */

1. do {
2.    maxdelta = 0.0;
3.    for (int i=1; i<N-1; i++) {
4.       for (int j=1; j<N-1; j++) {
5.          phi[next][i][j] = (phi[cur][i+1][j] + phi[cur][i-1][j] +
                               phi[cur][i][j+1] + phi[cur][i][j-1]) / 4;
6.          const double delta = fabs(phi[next][i][j] - phi[cur][i][j]);
7.          if (delta > maxdelta) {
8.             maxdelta = delta;
9.          }
10.      }
11.   }
      /* scambia cur e next */
12.   const int tmp = cur; cur = next; next = tmp;
13. } while (maxdelta > epsilon);
```

Discutere in che modo si possa usare OpenMP per parallelizzare il codice precedente (non è indispensabile indicare la sintassi corretta delle direttive OpenMP da usare). **Prestare attenzione ai dettagli!** Fare riferimento se necessario ai numeri di riga e/o richiamando il codice; evitare risposte confuse o generiche che verranno penalizzate.

Risposta: La soluzione verosimilmente più efficiente è la seguente (nota: la clausola collapse(2) non è indispensabile, e non è nemmeno garantita migliorare le prestazioni).

```
do {
   maxdelta = 0.0;
#pragma omp parallel for collapse(2) reduction(max:maxdelta)
   for (int i=1; i<N-1; i++) {
      for (int j=1; j<N-1; j++) {
         phi[next][i][j] = (phi[cur][i+1][j] + phi[cur][i-1][j] +
                            phi[cur][i][j+1] + phi[cur][i][j-1]) / 4;
         const double delta = fabs(phi[next][i][j] - phi[cur][i][j]);
         if (delta > maxdelta) {
            maxdelta = delta;
         }
      }
   }
   /* scambia cur e next */
   const int tmp = cur; cur = next; next = tmp;
```

```
} while (maxdelta > epsilon);
```

Diverse risposte suggerivano l'uso della clausola `#pragma omp critical` per proteggere l'"if" delle righe 7–9; sebbene tecnicamente corretto, ciò sarebbe risultato estremamente inefficiente, dato che equivale essenzialmente a serializzare l'esecuzione delle iterazioni dei cicli "for".

Domanda 3.
Abbiamo visto a lezione come le primitive `MPI_Send` e `MPI_Recv` di MPI possano causare deadlock se non usate correttamente.

1. Si illustri il problema
2. Si spieghi in che modo si può evitare il verificarsi di deadlock.

Risposta: vedi slide. Ho notato un po' di confusione sulla semantica della funzione `MPI_Send`. Non è vero che `MPI_Send` rimane bloccata fino a quando il destinatario esegue `MPI_Recv`!! `MPI_Send` rimane bloccata fino a quando i dati sono stati "consegnati" al nodo destinatario, il che vuol dire che `MPI_Send` potrebbe terminare non appena i dati sono stati copiati in un buffer lato destinatario, senza che quest'ultimo abbia ancora eseguito una corrispondente `MPI_Recv`.

Domanda 4.

1. Spiegare che cosa è e a cosa serve la *shared memory* di una GPU.
2. Si illustri un esempio pratico in cui l'uso della shared memory può essere utile; si può ricorrere ad uno degli esempi visti a lezione o in laboratorio, oppure proporne uno nuovo.

Risposta: vedi slide. Era importante sottolineare come la *shared memory* risulti utile nel caso in cui la stessa informazione possa essere letta più volte, da thread diversi. Relativamente al punto 2, l'esempio della trasposizione di una matrice, citato in una risposta, non era in realtà particolarmente rilevante. Infatti l'uso della *shared memory* nella trasposizione della matrice serve perché consente a thread diversi di accedere a dati contigui in memoria globale, che è molto più efficiente rispetto ad accessi “strided”.