# High Performance Computing
# Prova scritta – 12 settembre 2019 – 1h30

## PARTE 1 – RISPOSTA SINGOLA - Ogni domanda ha una sola risposta VERA.

- Una risposta esatta fa acquisire il punteggio positivo riportato a fianco della domanda
- Una risposta errata fa perdere il punteggio negativo riportato a fianco della domanda
- Una risposta lasciata in bianco viene valutata 0

1. **(2, -.5)** Nel paradigma del *message passing*
   a) La comunicazione tra task/processi è esplicita, la sincronizzazione implicita
   b) La comunicazione tra task/processi è implicita, la sincronizzazione esplicita
   c) La comunicazione tra task/processi è esplicita, come anche la sincronizzazione
   d) La comunicazione tra task/processi è implicita, come anche la sincronizzazione

2. **(2, -.5)** Il protocollo di *cache coherence* MSI
   a) È più efficiente del protocollo MESI, in quanto tratta le read private come quelle condivise
   b) È meno efficiente del protocollo MESI, in quanto tratta le read private come quelle condivise
   c) È ugualmente efficiente al protocollo MESI, ma richiede meno area per essere realizzato
   d) Nessuna delle precedenti risposte è vera

3. **(2, -.5)** Lo *Utilization Wall*
   a) Stabilisce che il numero di transistor in un circuito integrato CMOS raddoppia all'incirca ogni 18 mesi
   b) Stabilisce che con ogni nuova generazione di processo produttivo la percentuale di un chip che può operare a massima frequenza decade esponenzialmente per vincoli di potenza
   c) Stabilisce che il miglioramento che si può ottenere su una certa parte del sistema è limitato dalla frazione di tempo in cui tale attività ha luogo
   d) Stabilisce che due task T1 e T2 sono paralleli se l'input di T1 (T2) non è parte dell'output di T2 (T1) e se gli output di T1 e T2 non si sovrappongono

4. **(2, -.5)** Il partizionamento del carico di lavoro in CUDA
   e) È trasparente al programmatore e derivato automaticamente da costrutti di alto livello
   f) È interamente gestito dal programmatore, sia a livello di singolo *thread* che di intero *offload*
   g) È bilanciato in hardware, il programmatore non ha modo di influenzare lo *schedule*
   h) Nessuna delle precedenti risposte è vera

## PARTE 2 – (POSSIBILI) RISPOSTE MULTIPLE -
## Ogni domanda può avere da una a quattro risposte CORRETTE.

- Ogni risposta esatta viene calcolata: +1
- Ogni risposta errata viene calcolata: -0.5
- Una risposta lasciata in bianco viene calcolata: 0

4. Supportare OpenMP su una determinata architettura
   a) Richiede necessariamente che quella architettura supporti i POSIX threads (o pthreads)
   b) Richiede che il compilatore C/C++ per quella architettura sia in grado di riconoscere le #pragma OpenMP e trasformarle in codice parallelo
   c) Richiede una libreria di supporto a runtime
   d) Nessuna delle precedenti

5. Si consideri il seguente stralcio di codice, eseguito su due processori

```
int s = 0;
#pragma omp parallel
{
   int local_s = 0;
   #pragma omp for
   for (int i = 0; i < n; i++)
      local_s += sqr(A[i]);
   s += local_s;
}
```

   a) Il programma esegue correttamente, dividendo il carico di lavoro equamente tra i due processori
   b) Il programma contiene una race condition
   c) I due processori potrebbero non eseguire lo stesso carico di lavoro
   d) Nessuna delle precedenti

6. Nelle architetture NVIDIA un *warp*
   a) Contiene *threads* che eseguono in maniera SIMD
   b) Contiene *threads* che possono eseguire istruzioni distinte
   c) È un'unità di scheduling dentro un SM
   d) Nessuna delle precedenti

7. Le ottimizzazioni di un compilatore moderno
   a) Avvengono principalmente nel blocco *front-end*
   b) Avvengono principalmente nel blocco *middle-end*
   c) Non possono avvenire a *link-time*
   d) Possono avvenire nel blocco *back-end*

# PARTE 3 – DOMANDE APERTE

- Una risposta esatta fa acquisire il punteggio positivo riportato a fianco della domanda
- Una risposta errata può eventualmente causare una penalità che dipende dalla gravità dell'errore
- Una risposta lasciata in bianco viene calcolata: 0

8. **(6 pt)** Si descrivano le principali differenze tra la metodologia di progetto HDL e HLS per FPGA.

   HDL: Best quality of results, but HW design expertise needed , low abstraction level , much slower than SW design on processors
   - Level of abstraction is RTL (register transfer level), where building blocks are adders, multipliers, flip-flops, etc.
   - Need to handle explicitly sequential logic signals (registers, flip-flops, control signals (e.g., reset), clock)

   HLS: Behavioral description of the HW via a procedural, high level language. Automatic generation of an (optimized) register transfer level (RTL) description suitable for hardware Implementation. Main benefits:
   • Productivity: lower design complexity and faster simulation speed
   • Portability: single source → multiple implementations
   • Permutability: rapid design space exploration → higher quality of result

9. **(9 pt)** Si consideri il seguente stralcio di codice di un programma OpenMP.

```
int main() {
    int a[12];

    Work1(a); // 1 Second

   #pragma omp parallel for shared(a) schedule(static)
    for(int i=0; i < 12; ++i)
        a[i] = Work2(); // 0.5 Seconds

   return 0;
}
```

a) Si calcoli lo speedup atteso per l'esecuzione su una macchina a 4 processori (rispetto all'esecuzione sequenziale). Si assuma un tempo di esecuzione pari a 1 secondo per la funzione Work1 e di 0,5 secondi per la funzione Work2. Si assuma che non esistano overheads. **(2 pt)**

b) Si calcoli lo speedup massimo ottenibile e il minimo numero di processori richiesto per ottenerlo. **(3pt)**

c) Supponendo di usare le clausole "**`num_threads(5) schedule(static,4)`**", si scriva il set di indici del loop assegnati a ciascun thread (nella forma lower bound (LB) e upper bound (UB)). Si calcoli lo speedup atteso rispetto al codice sequenziale, assumendo che la macchina abbia più di 4 processori. **(4pt)**

**SOLUTION**

**a)** Sequential execution time:

$$T_{seq} = T_{work1} + T_{work2} * T_{iterations} = 1 + 0.5 * 12 = 7\ s$$

The parallel execution time includes two components (sequential and parallel part of the code):

$$T_{seq\_par} = T_{work1}$$

$$T_{par\_part} = T_{work2} * \left\lceil \frac{iterations}{N} \right\rceil, \qquad N \equiv number\ of\ cores$$

$$T_{par} = T_{seq\_par} + T_{par\_part} = 1 + 0.5 * \left\lceil \frac{12}{4} \right\rceil = 2.5\ s$$

Speedup:

$$Speedup = \frac{T_{seq}}{T_{par}} = 2.8$$

**b)** The maximum speedup is achieved when the time required to execute the parallel part is minimized. Considering the structure of the loop, this means:

$$T_{par\_part} = 0.5 * \min_{0<N<\infty} \left\lceil \frac{12}{N} \right\rceil = 0.5 * 1 = 0.5, \qquad \forall\ N \geq 12$$

Then we can compute the value of $T_{par}$ and the resulting speedup:

$$T_{par} = T_{seq\_part} + T_{par\_part} = 1 + 0.5 = 1.5\ s$$

$$Speedup = \frac{T_{seq}}{T_{par}} \approx 4.67$$

The minimum number of processors required to achieve this speedup is **N = 12**.

**c)** The indexes executed by each thread expressed as LB and UB are:

- Thread 0: **LB = 0, UB = 3**
- Thread 1: **LB = 4, UB = 7**
- Thread 2: **LB = 8, UB = 11**
- Thread 3: ***no work***
- Thread 4: ***no work***

To compute the speedup compared to the sequential code, we need to find the new value of $T_{par}$:

$$T_{par} = T_{seq\_par} + T_{par\_part} = 1 + 0.5 * \left\lceil \frac{12}{3} \right\rceil = 3\ s$$

$$Speedup = \frac{T_{seq}}{T_{par}} \approx 2.33$$