Tecniche di programmazione portabili per la simulazione di sistemi fisici su GPU e Manycore

Paolo Leoni

Relatore: Prof. Roberto Alfieri

Correlatore: Prof. Roberto De Pietri

Università degli Studi di Parma - Dipartimento di Fisica e Scienze delle Terra

17 Luglio 2014

Outline

Outline

Quali problemi vogliamo risolvere?

Un tipico problema fisico

$$\frac{\partial u}{\partial t} = f(\vec{x}, t, u, \frac{\partial u}{\partial x_1}, \frac{\partial u}{\partial x_2}, ...)$$

Equazione del calore

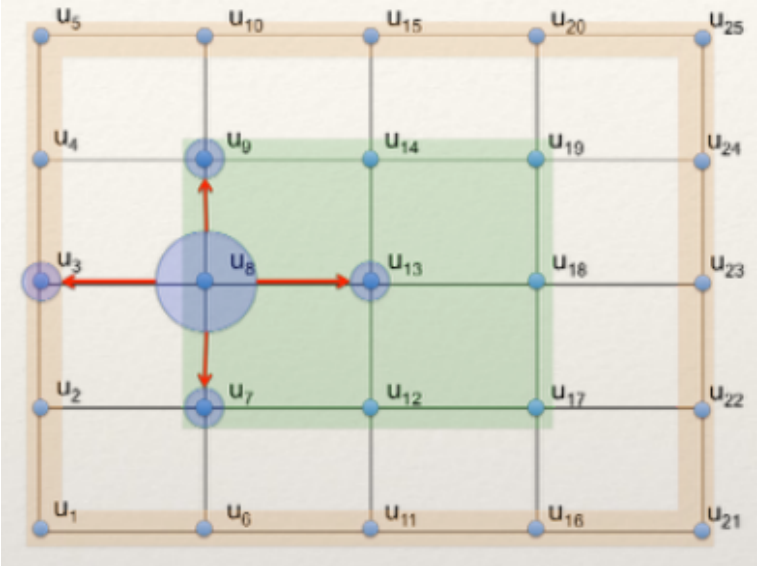
$$\frac{\partial u}{\partial t} = \triangle u$$

Equazione delle onde

$$\frac{\partial^2 u}{\partial t^2} = c^2 \triangle u$$

Quali problemi vogliamo risolvere?

- deve essere un problema altamente parallelizzabile
- una buona parallelizzazione deve poter agire a più livelli (cluster/core/vector)

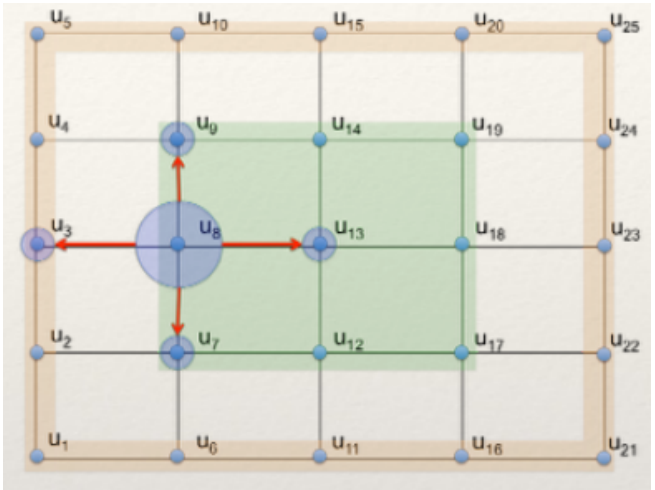


- discretizzazione del dominio spaziale
- discretizzazione del dominio temporale (modello a steps)
- il dominio spaziale deve essere «separabile»

Un esempio

Algoritmo per l'eq. del calore

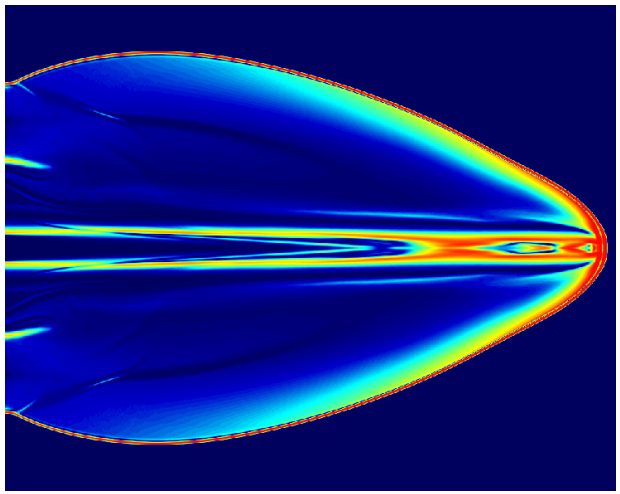
```
[...]
for(int i=1; i< NX-1; i++) {
     for(int j=1; j< NY-1; j++) {
         Tnew[i][j] = 0.25 * (T_right + T_left
+ T_up + T_down);
     }
}
[...]
```

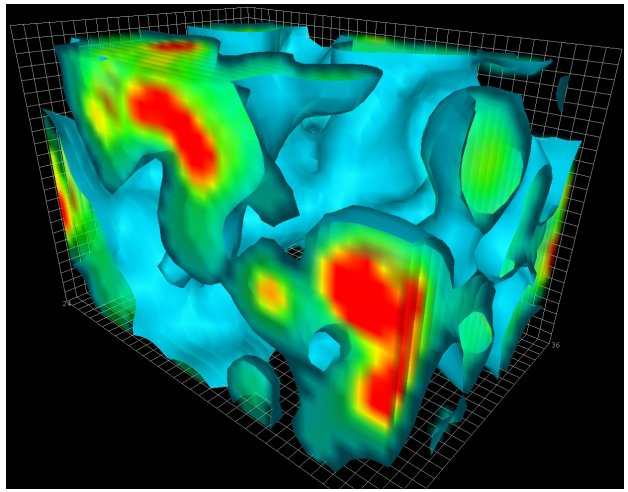


Quali problemi vogliamo risolvere?

Nella realtà, vorremmo poter accelerare il calcolo (senza riscrivere interamente il codice) nella risoluzione di problemi come:

- fluidodinamica relativistica
- problemi di turbolenza
- Hybrid Montecarlo Lattice QCD





Quali problemi vogliamo risolvere?

Ci serve un modello di programma da poter studiare, con alcune particolari caratteristiche:

- deve essere semplice da sviluppare, in modo da mantenere un controllo maggiore sul suo comportamento
- al tempo stesso deve avere le caratteristiche tipiche di un software scientifico, in modo da poter capire quali problemi dovremo affrontare in un porting reale

Quali problemi vogliamo risolvere?

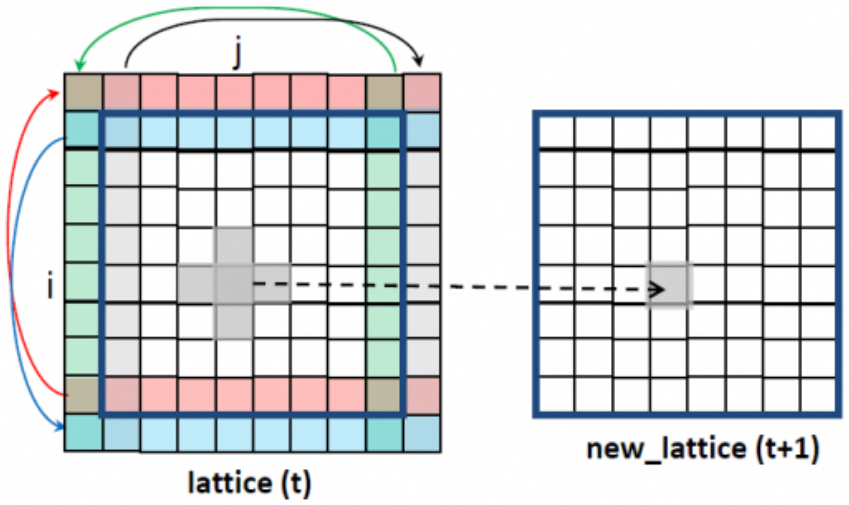
Ci serve un modello di programma da poter studiare, con alcune particolari caratteristiche:

- deve essere semplice da sviluppare, in modo da mantenere un controllo maggiore sul suo comportamento
- al tempo stesso deve avere le caratteristiche tipiche di un software scientifico, in modo da poter capire quali problemi dovremo affrontare in un porting reale

Outline

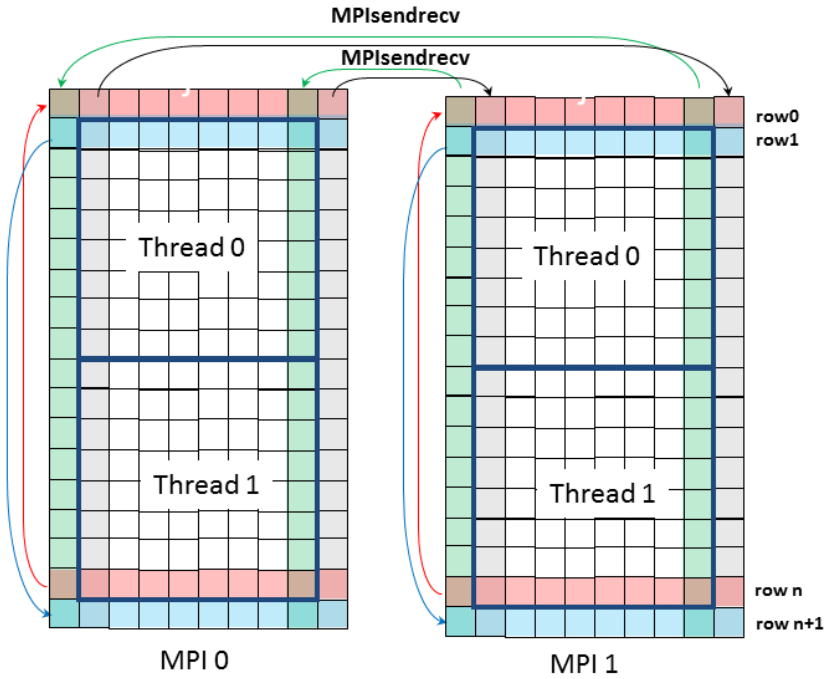
L'algoritmo Game of Life

- è stato ideato da John Conway nel 1970.
- è un gioco senza giocatori, la sua evoluzione dipende dalle condizioni iniziali.
- semplice regola di aggiornamento: la cella i-esima «vive» se tra le celle vicine ce ne sono 3 o 2 accese, altrimenti «muore».



L'algoritmo Game of Life

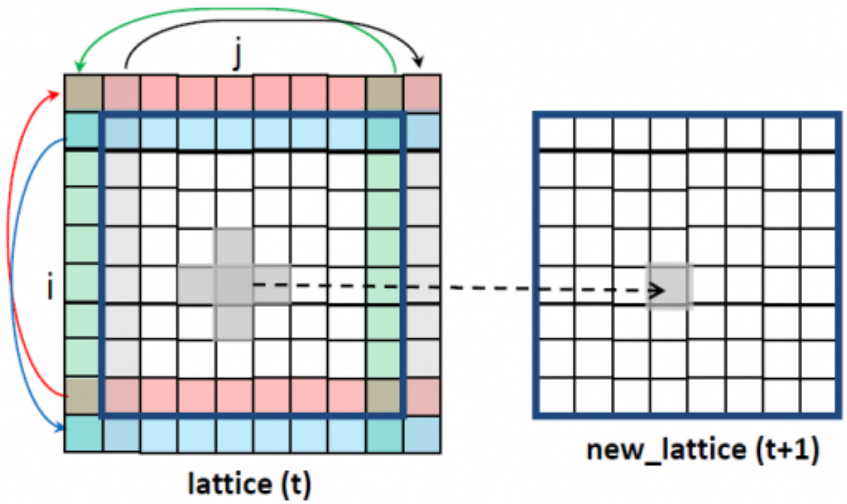
- è un algoritmo con un elevato potenziale di calcolo parallelo
- possibilità di separare il calcolo dei bordi da quello degli interni
- questa struttura permette inoltre lo scambio dei bordi tra processi MPI
- per studiare un ulteriore livello di parallelismo, abbiamo aggiunto un semplice calcolo vettoriale:
  `sum += A[ncomp] + B [ncomp]`



L'algoritmo Game of Life

Quante operazioni svolge il programma?
(griglia 17000x17000 - 10 steps)

- senza calcolo vettoriale: *29 Gflop*.
- con il calcolo vettoriale (Ncomp=1000): *5800 Gflop*.



Outline

Il mondo dell'HPC sta cambiando...

- da sistemi tradizionali seriali collegati in parallelo...
- ...a sistemi (sempre collegati in parallelo) dotati di accelerazione *vettoriale*, attraverso *coprocessori di calcolo*
- si pone quindi il problema di *capire* quale delle tecnologie di accelerazione disponibili si adatta meglio alle esigenze del calcolo scientifico.

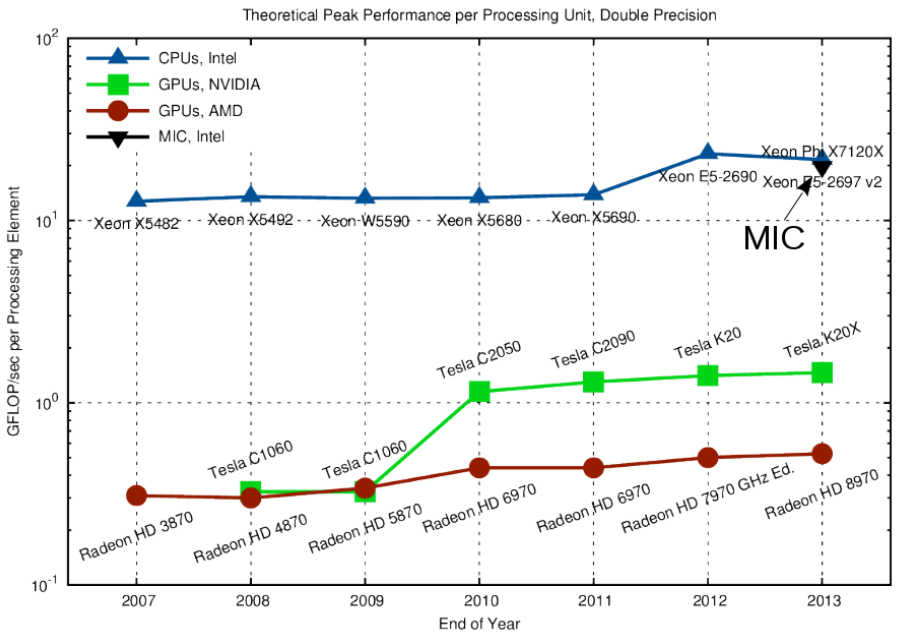
Il mondo dell'HPC sta cambiando...

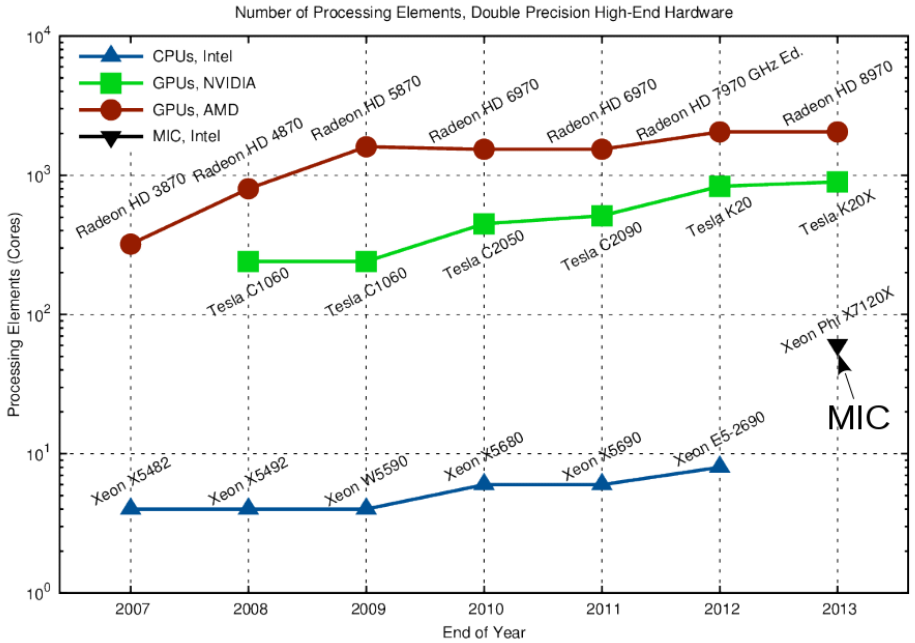
- da sistemi tradizionali seriali collegati in parallelo...
- ...a sistemi (sempre collegati in parallelo) dotati di accelerazione *vettoriale*, attraverso *coprocessori di calcolo*
- si pone quindi il problema di *capire* quale delle tecnologie di accelerazione disponibili si adatta meglio alle esigenze del calcolo scientifico.

Il mondo dell'HPC sta cambiando...

- da sistemi tradizionali seriali collegati in parallelo...
- ...a sistemi (sempre collegati in parallelo) dotati di accelerazione *vettoriale*, attraverso *coprocessori di calcolo*
- si pone quindi il problema di *capire* quale delle tecnologie di accelerazione disponibili si adatta meglio alle esigenze del calcolo scientifico.

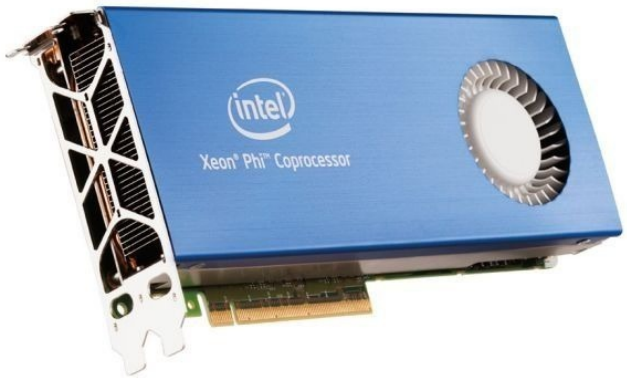
Il mondo dell'HPC sta cambiando...

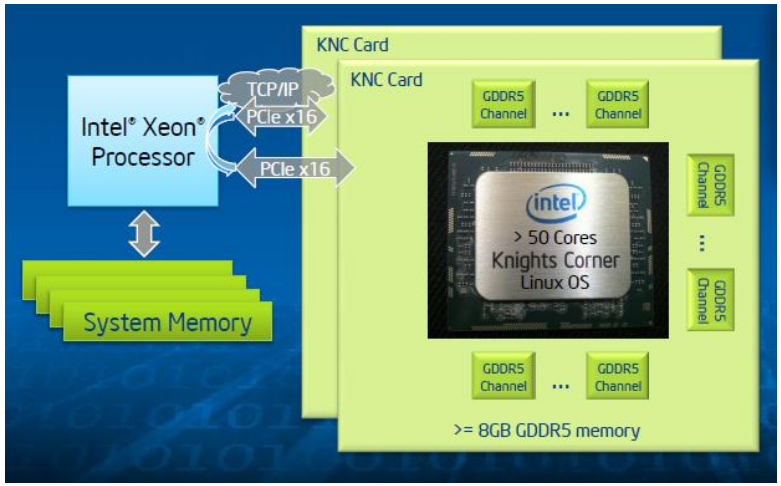




Intel Xeon PHI MIC

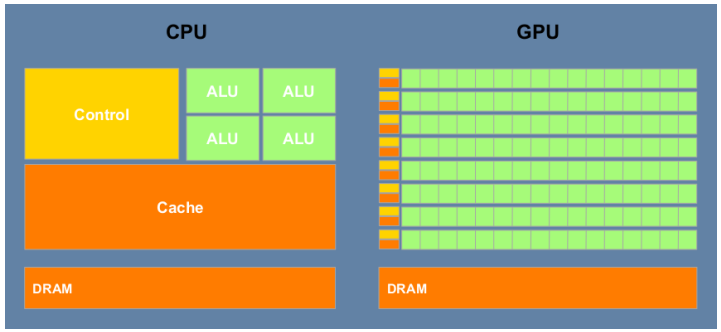
- 60 core da 1 Ghz (4 thread / core)
- 8 Gb di memoria
- possibilità di utilizzo sia in *modalità nativa* (utilizzata da noi) che *offload*
- unità SIMD da 512 bit
- performance di picco pari *1011 Gflops*





Nvidia Tesla Kepler 20

- 2496 core da 600 Mhz, suddivisi in 15 macro unità di elaborazione (SMX)
- 5 Gb di memoria
- si può utilizzare solo in modalità *offload*
- performance di picco pari a *1170 Gflops*
- il bus di trasferimento dati tra GPU e CPU è piuttosto lento



Un prototipo di studio: il cluster Eurora

- 64 nodi di calcolo
- ogni nodo contiene 2 CPU Sandy Bridge da 8 core (*268 Gflops*)
- a 32 nodi sono accoppiati 2 acceleratori Intel Xeon PHI MIC
- ai rimanenti 32 sono accoppiati 2 acceleratori Nvidia Tesla Kepler 20
- i nodi sono collegati tra loro da una rete Qlogic Infiniband.

Outline

Obiettivi del software

Per poter seguire il cambiamento tecnologico, abbiamo bisogno di strumenti software che ci consentano di:

- ridurre al minimo la necessità di conoscere i dettagli fisici dell'architettura di accelerazione in cui avviene l'esecuzione del codice.
- ricavare informazioni sulle prestazioni massime raggiungibili senza un'ottimizzazione specifica del singolo hardware su codice generico.

Obiettivi del software

Per poter seguire il cambiamento tecnologico, abbiamo bisogno di strumenti software che ci consentano di:

- ridurre al minimo la necessità di conoscere i dettagli fisici dell'architettura di accelerazione in cui avviene l'esecuzione del codice.
- ricavare informazioni sulle prestazioni massime raggiungibili senza un'ottimizzazione specifica del singolo hardware su codice generico.

La nostra scelta: i linguaggi «directive-based»

Vantaggi:

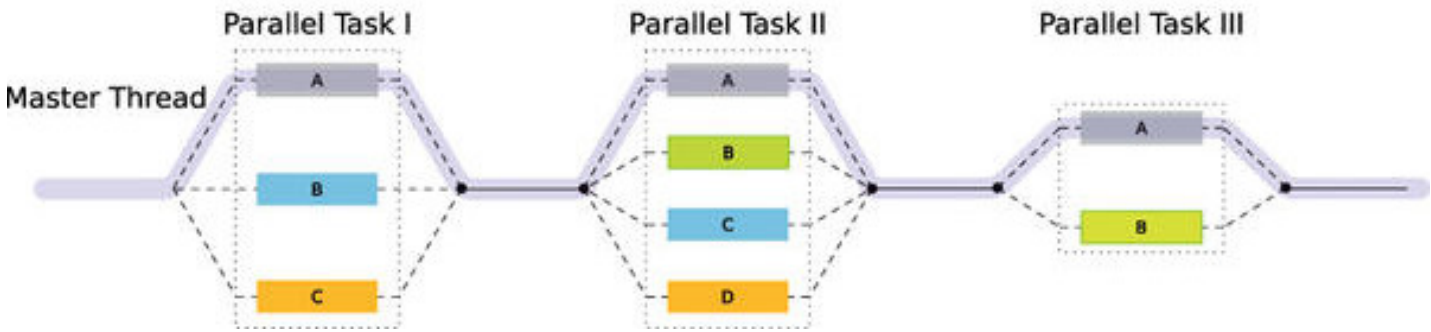
- forniscono un approccio più intuitivo e quindi più semplice
- ci allontanano dalla complessità dell'architettura
- se l'implementazione è buona, è possibile ottenere una buona performance

Svantaggi:

- minore controllo su ciò che avviene
- il profiling del programma può essere più difficoltoso
- il codice è meno ottimizzato (prestazioni più basse)

Calcolo su CPU: OpenMP

- permette di distribuire il calcolo tra le unità fisiche del singolo nodo, diminuendo il tempo di esecuzione (strong-scaling)
- si basa sull'inserimento delle direttive «pragma omp...».
- consente di ottenere buone prestazioni senza conoscere il linguaggio di basso livello (e.g. POSIX pthreads)
- è stato integrato sia per il porting su CPU SandyBridge che su acceleratore Intel Xeon PHI MIC.

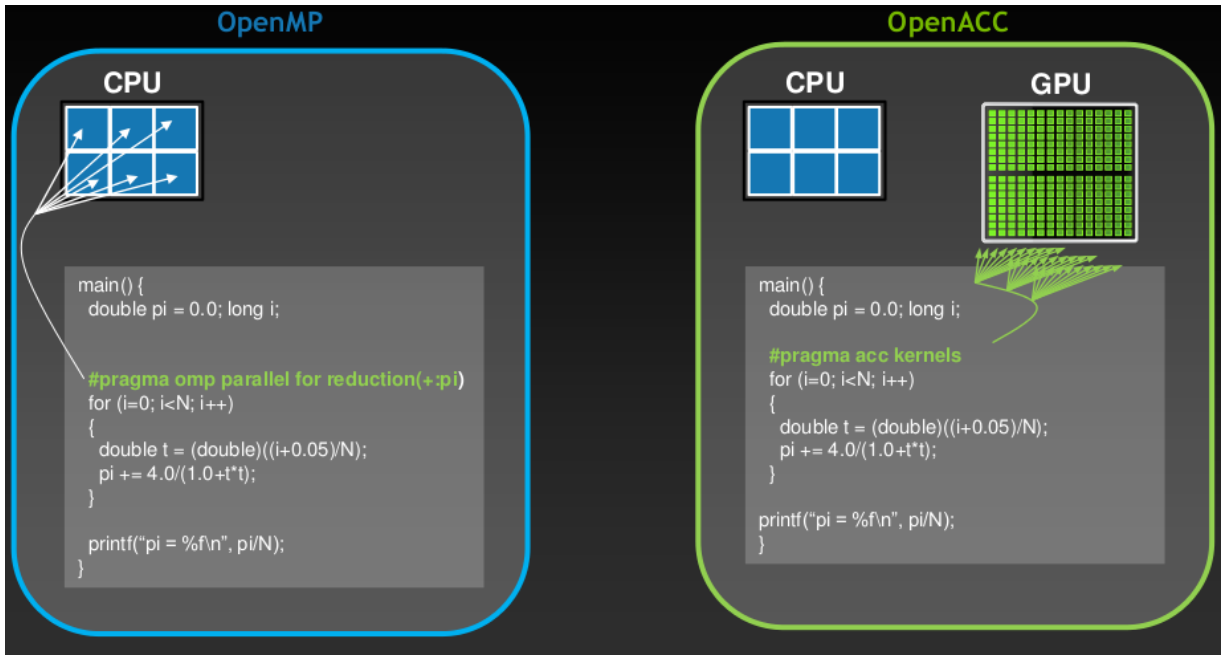


Calcolo su GPU

- la programmazione su GPGPU è generalmente svolta tramite le estensioni di linguaggio.
- per le GPU Nvidia un codice C o Fortran può essere adattato con l'estensione CUDA-C o CUDA-Fortran.
- per tutte le altre GPU esiste un'estensione dal funzionamento simile, chiamata OpenCL.
- **Vantaggi:** software ottimizzato, controllo elevato sull'architettura
- **Svantaggi:** sviluppo del codice più lento e difficile, bassa portabilità

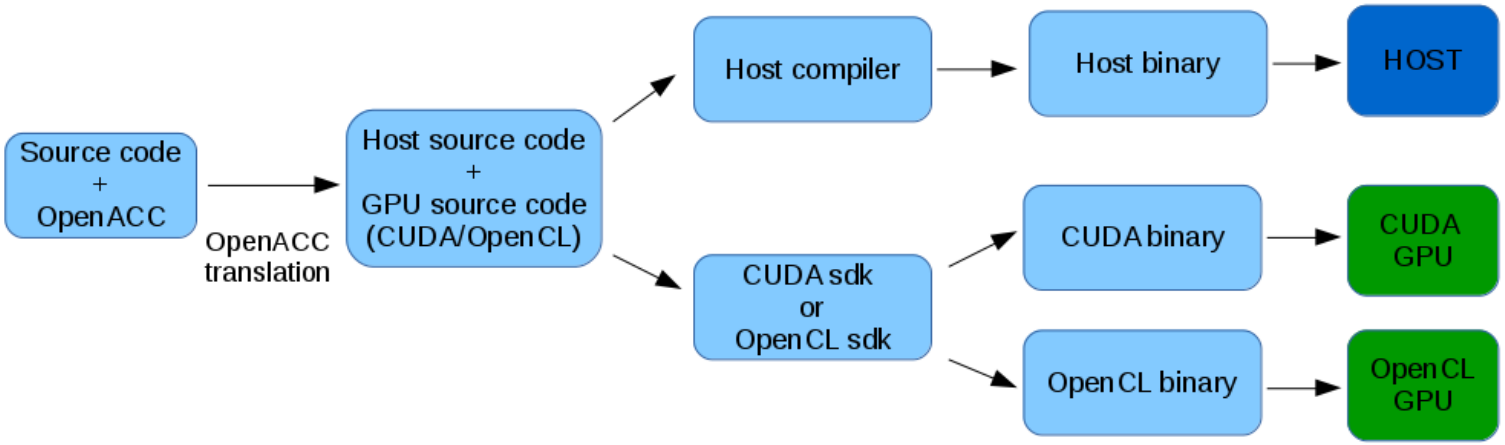
Calcolo su GPU: OpenACC

- approccio molto simile ad OpenMP, ma orientato all'utilizzo su acceleratori
- consente l'utilizzo delle risorse senza conoscere le estensioni di basso livello
- mantiene una buona portabilità e facilita l'utilizzo degli acceleratori



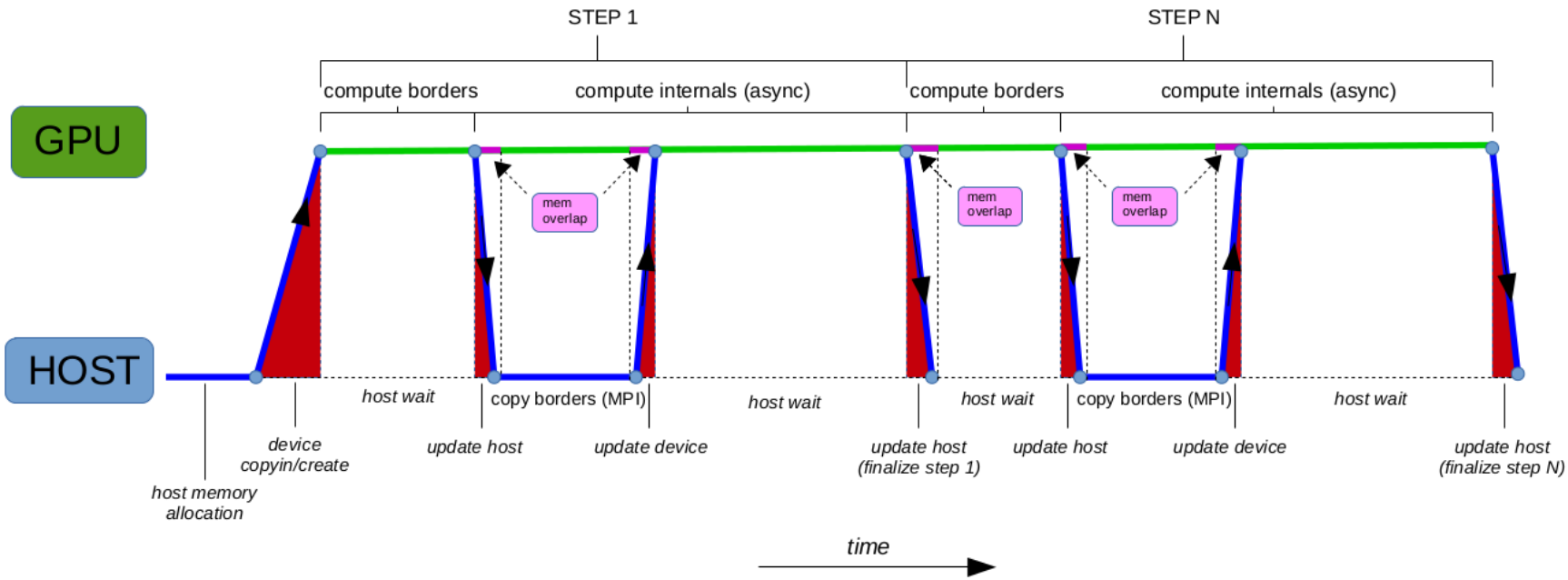
Calcolo su GPU: OpenACC

- permette solo l'utilizzo della modalità offload.
- rispetto ad OpenMP, si ha un controllo minore delle risorse disponibili.
- l'ottimizzazione dipende molto dal compilatore utilizzato.



Calcolo su GPU: OpenACC

- gestione del momento e del modo con cui vengono trasferiti i dati sul device
- possibilità di rendere alcune parti del programma asincrone con la clausola `async(k)`
- possibilità di fare overlap tra calcolo e trasferimento dati



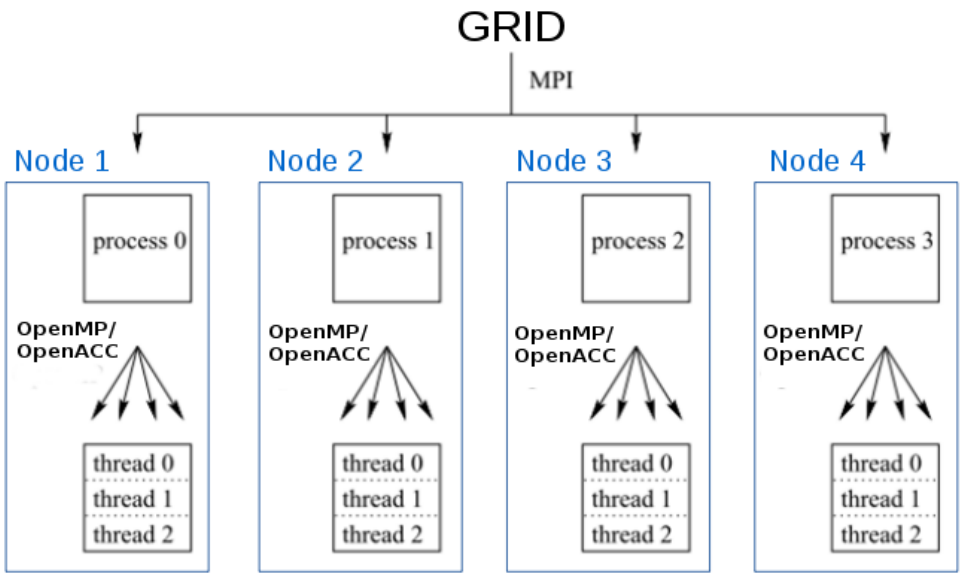
Life OpenACC/OpenMP

```
void compute_Internals(double ** grid, double ** next_grid) {
  int k,l,j,i;
  double neighbors=0.0;

  #pragma acc kernels present(grid[0:nrows+2][0:ncols+2],next_grid[0:nrows+2]    \
                              [0:ncols+2],sum,A[0:ncomp],B[0:ncomp]) async(2)
  {
    #pragma omp for private(i,j,k) reduction(+:sum) schedule(static)
    #pragma acc loop independent reduction(+:sum)
     for (i=rmin_int; i<=rmax_int; i++) {
     #pragma acc loop independent reduction(+:sum)
       for (j=cmin_int; j<=cmax_int; j++) {
        #pragma ivdep
        #pragma vector aligned
       #pragma acc loop vector(32) independent reduction(+:sum)
         for (k=0; k < ncomp; k++)  sum += A[k] + B[k];
     } }
     #pragma omp for private(i,j,k,neighbors) schedule(static)
     #pragma acc loop independent collapse(2)
     for (i=rmin_int; i<=rmax_int; i++) {
       for (j=cmin_int; j<=cmax_int; j++) {
         neighbors = grid[i+1][j+1] + grid[i+1][j] + grid[i+1][j-1] + grid[i][j+1] + grid[i][j-1] \
                                    + grid[i-1][j+1]+grid[i-1][j]+grid[i-1][j-1];
         if ( ( neighbors > 3.0 ) || ( neighbors < 2.0 ) )
           next_grid[i][j] = 0.0;
         else if ( neighbors == 3.0 )
           next_grid[i][j] = 1.0;
         else
           next_grid[i][j] =  grid[i][j];
       }
     }
```

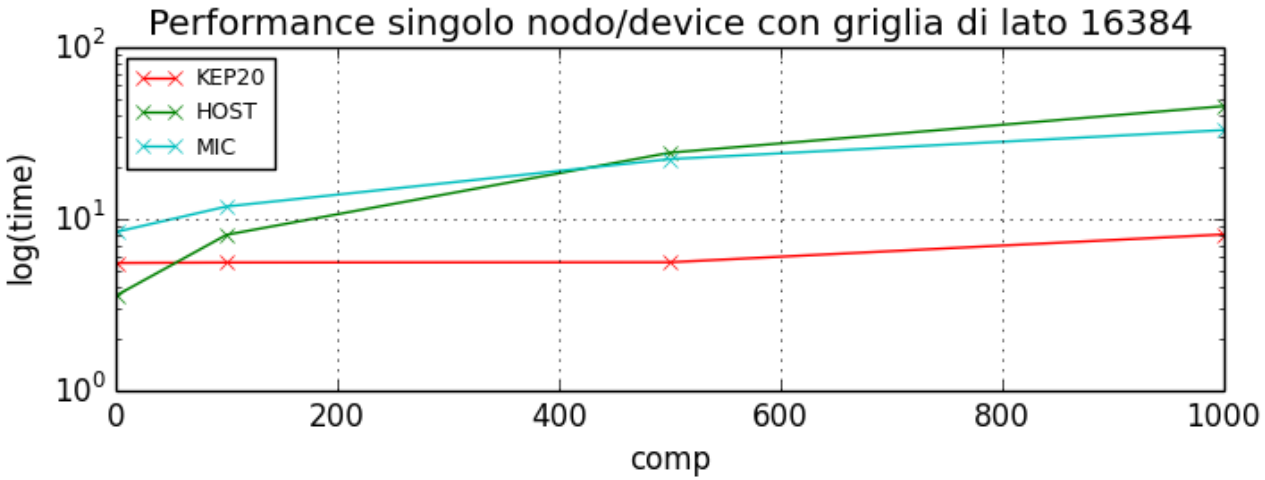
OpenMPI

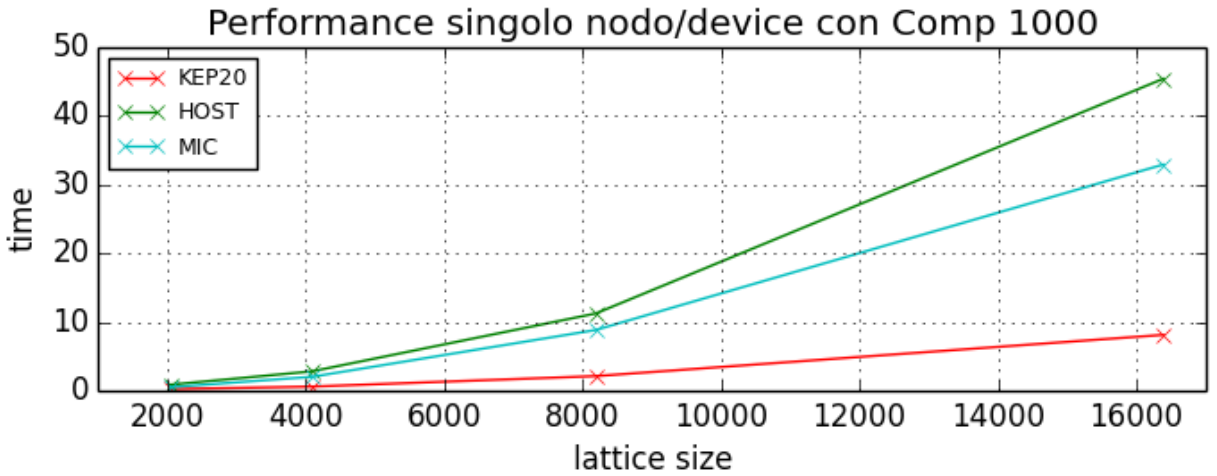
- distribuisce il calcolo su più nodi, la dimensione del problema aumenta senza aumentare il tempo di esecuzione (weak-scaling).
- si mantiene un buon controllo dell'interazione tra i processi.
- nel nostro modello ogni processo MPI avvia sul nodo assegnato una serie di processi OpenMP o OpenACC.



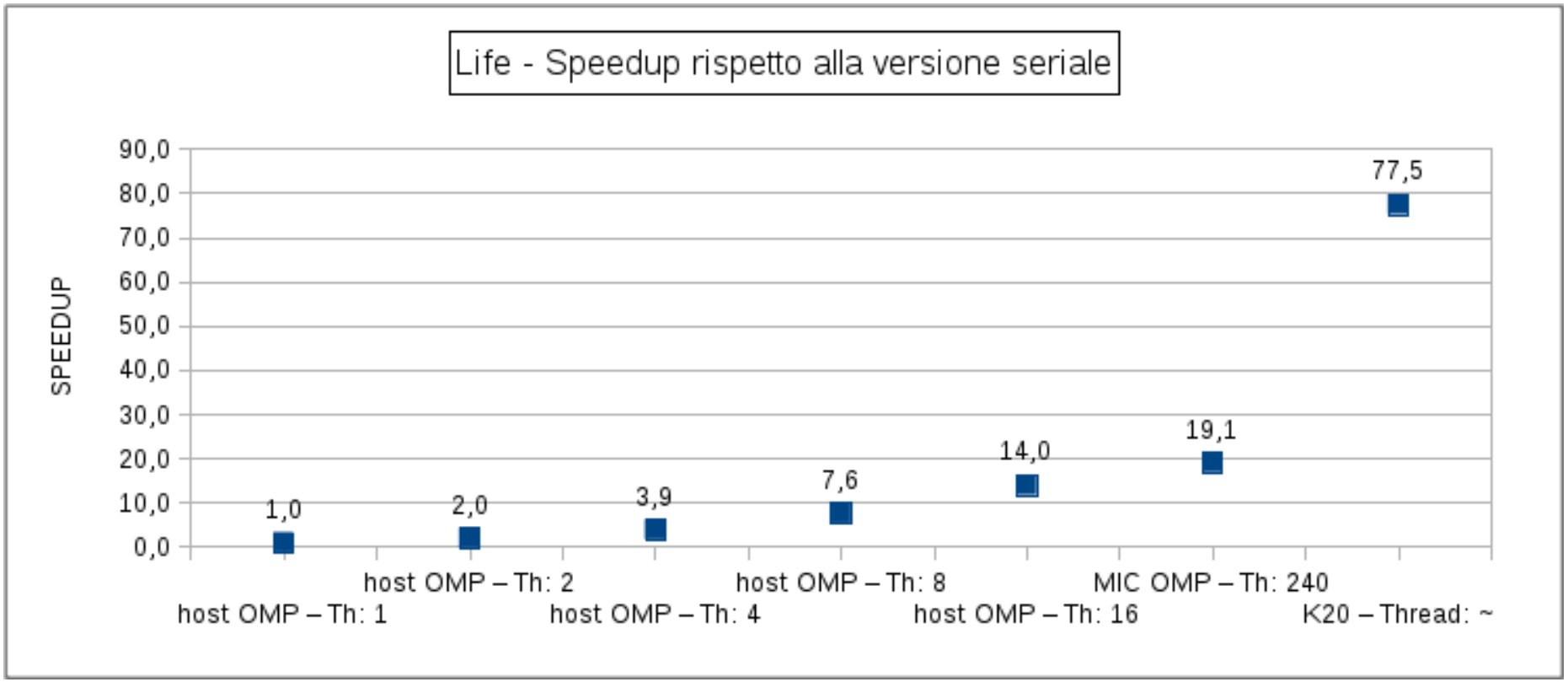
Outline

Confronto tra i device su singolo nodo



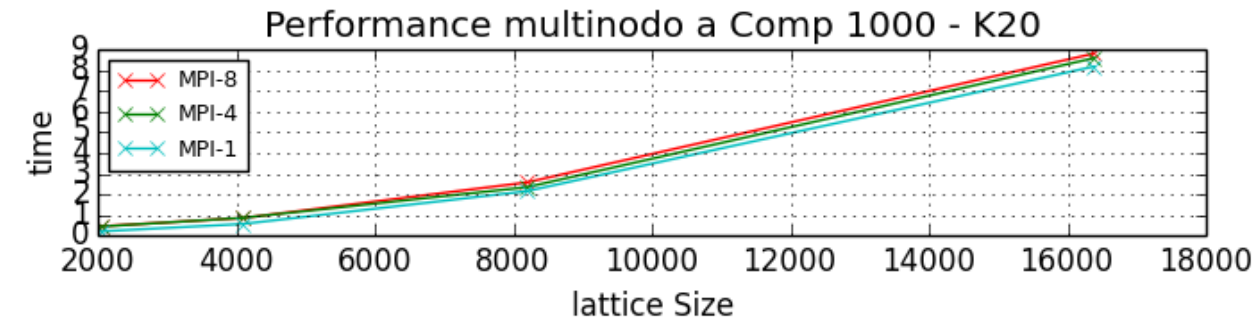


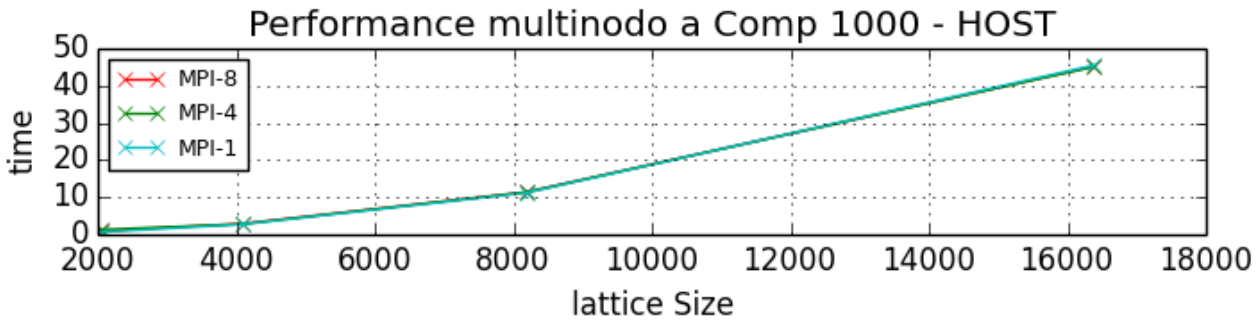
Confronto tra i device su singolo nodo

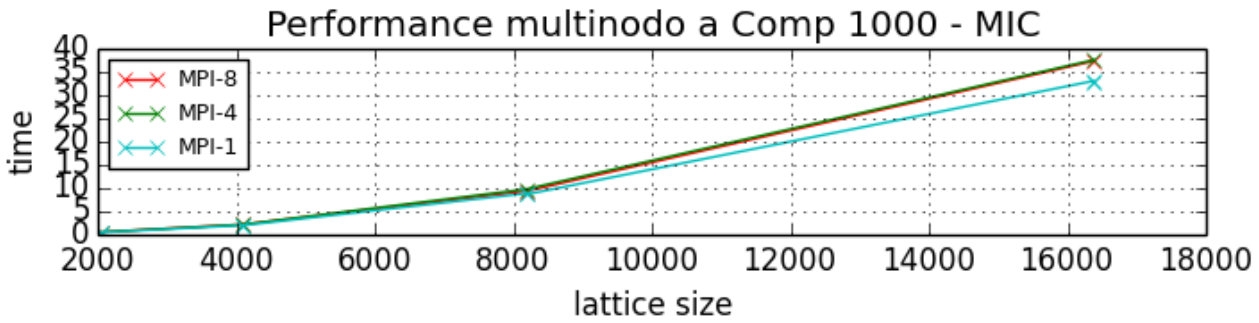


Outline

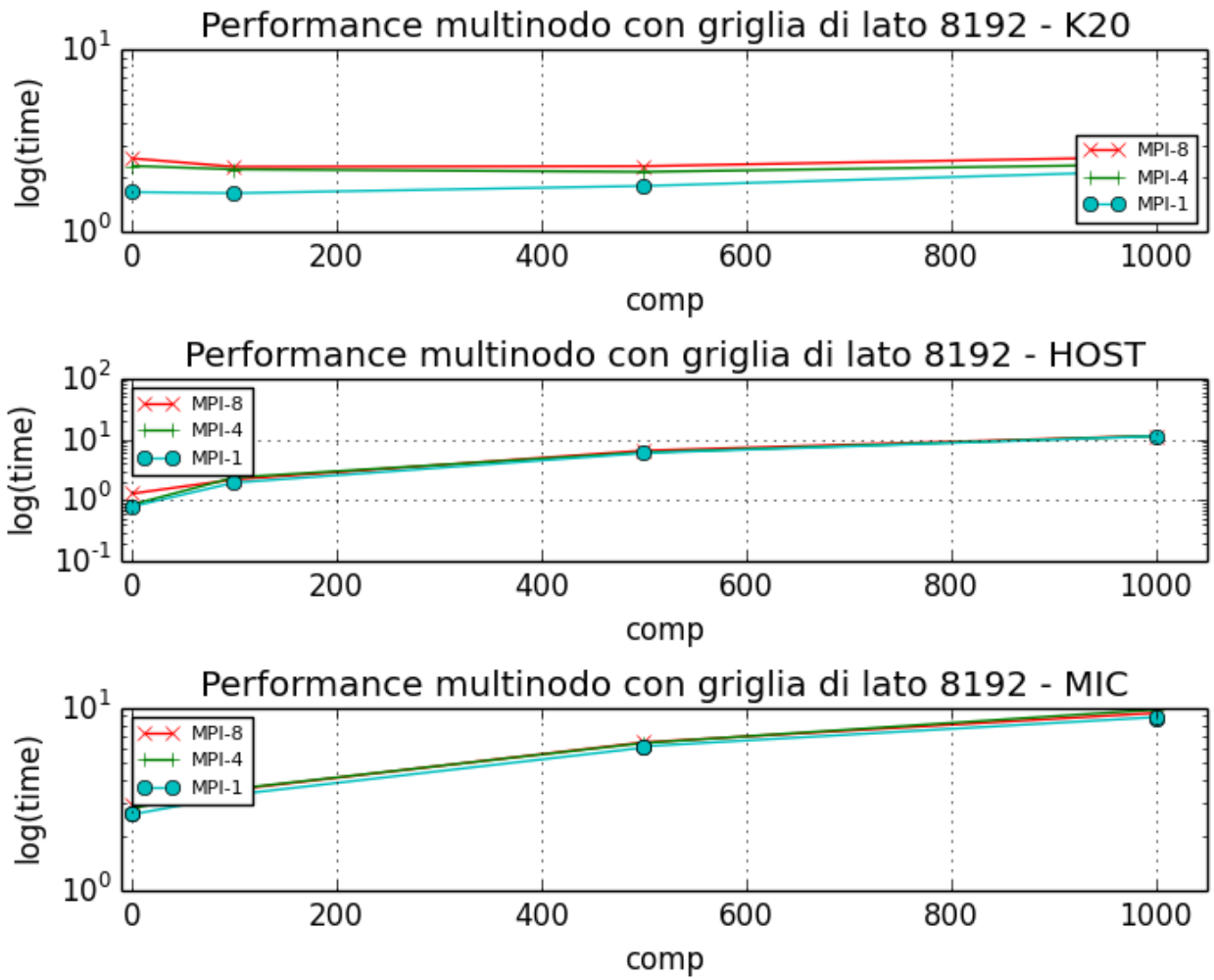
Confronto tra i device multinodo





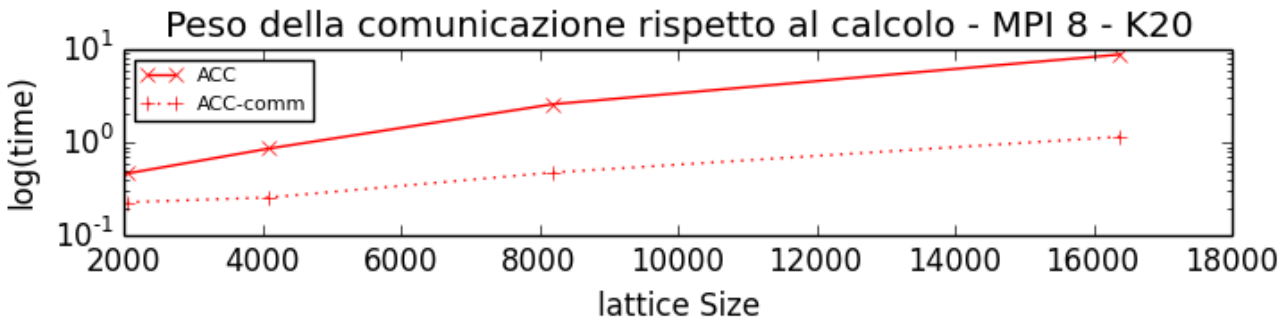


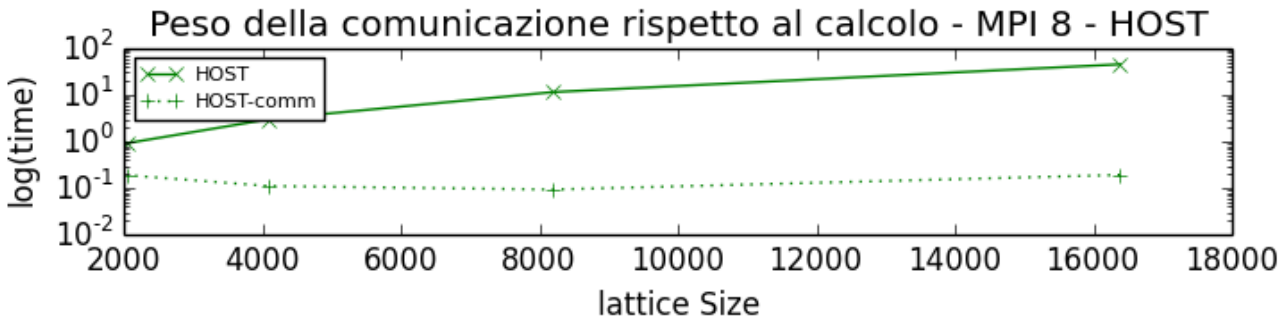
Confronto tra i device multinodo

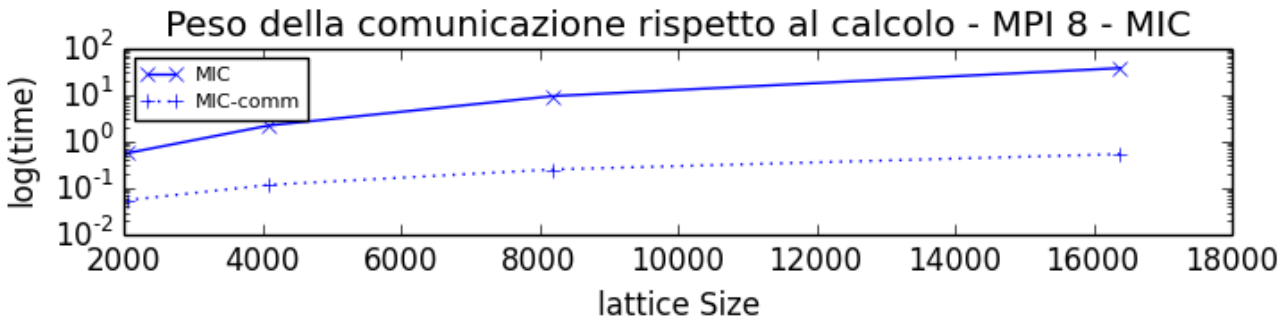


Outline

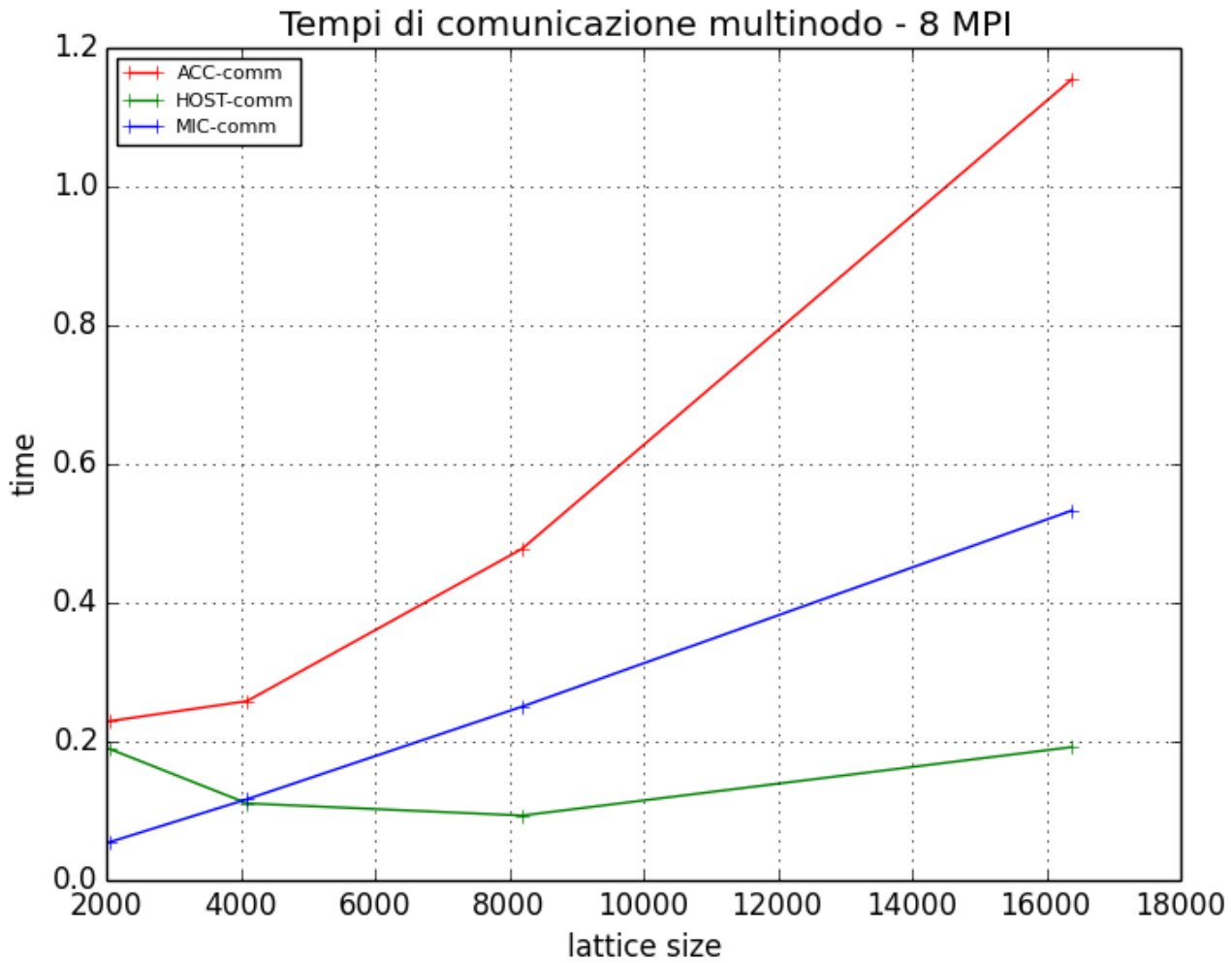
Tempi di esecuzione/comunicazione (Comp = 1000)







Tempi di comunicazione



Conclusioni

- Il porting su Intel Xeon PHI MIC si è dimostrato più semplice, ma non ha prodotto performance soddisfacenti.
- Il porting su Nvidia Tesla K20 tramite OpenACC si è rivelato molto promettente, ma è ancora necessaria una minima conoscenza del codice in cui vengono tradotti i kernel (nel nostro caso, CUDA).
- L'implementazione del protocollo MPI si è dimostrata efficace nel distribuire il problema su più nodi di calcolo.

- Prospettive future
  - Estensione su griglia 3D con bordi di dimensione variabile.
  - Studio di altre API per gli acceleratori (e.g. OpenMP 4.0)

Conclusioni

- Il porting su Intel Xeon PHI MIC si è dimostrato più semplice, ma non ha prodotto performance soddisfacenti.
- Il porting su Nvidia Tesla K20 tramite OpenACC si è rivelato molto promettente, ma è ancora necessaria una minima conoscenza del codice in cui vengono tradotti i kernel (nel nostro caso, CUDA).
- L'implementazione del protocollo MPI si è dimostrata efficace nel distribuire il problema su più nodi di calcolo.
- Prospettive future
  - Estensione su griglia 3D con bordi di dimensione variabile.
  - Studio di altre API per gli acceleratori (e.g. OpenMP 4.0)

Conclusioni

- Il porting su Intel Xeon PHI MIC si è dimostrato più semplice, ma non ha prodotto performance soddisfacenti.
- Il porting su Nvidia Tesla K20 tramite OpenACC si è rivelato molto promettente, ma è ancora necessaria una minima conoscenza del codice in cui vengono tradotti i kernel (nel nostro caso, CUDA).
- L'implementazione del protocollo MPI si è dimostrata efficace nel distribuire il problema su più nodi di calcolo.

- Prospettive future
  - Estensione su griglia 3D con bordi di dimensione variabile.
  - Studio di altre API per gli acceleratori (e.g. OpenMP 4.0)

Conclusioni

- Il porting su Intel Xeon PHI MIC si è dimostrato più semplice, ma non ha prodotto performance soddisfacenti.
- Il porting su Nvidia Tesla K20 tramite OpenACC si è rivelato molto promettente, ma è ancora necessaria una minima conoscenza del codice in cui vengono tradotti i kernel (nel nostro caso, CUDA).
- L'implementazione del protocollo MPI si è dimostrata efficace nel distribuire il problema su più nodi di calcolo.

- Prospettive future
  - Estensione su griglia 3D con bordi di dimensione variabile.
  - Studio di altre API per gli acceleratori (e.g. OpenMP 4.0)

Conclusioni

GRAZIE!