Progetto di High Performance Computing 2018/2019

Nicolas Farabegoli, matr. 788928
20 giugno 2019

Università di Bologna
nicolas.farabegoli@studio.unibo.it

1 Introduzione

L'obiettivo principale del progetto è ottimizzare, nel modo migliore possibile, l'algoritmo per la simulazione di terremoti proposto da Burridge-Knopoff [1]. Per ogni versione del progetto, si è proceduto per raffinamenti successivi, quindi inizialmente si è seguita la linea della semplicità; a seguito di verifiche, sono state effettuate migliorie per raggiungere un livello ragionevole di ottimizzazione. I risultati ottenuti sono quantomeno significativi sia in termini di scalabilità che di efficienza.

2 Versione OpenMP

L'implementazione a memoria condivisa con OpenMP è stata implementata senza troppe difficoltà. In un primo momento ci si è concentrati sul corretto funzionamento del programma nella sua versione parallela; solo in un secondo momento, a seguito di dettagliate sessioni di misurazione delle prestazioni, si è proceduto a ottimizzare il simulatore.

Inizialmente si sono individuati i cicli `for` rilevanti ai fini della computazione e su di essi è stata applicata la clausola `#pragma omp parallel for`. I risultati ottenuti a seguito di questa operazione sono stati piuttosto soddisfacenti, in quanto si sono osservati speedup rilevanti.

Le funzioni `average_energy` e `count_cells` dopo un'attenta analisi, è emerso che prevedono un' operazione di riduzione con l'operatore somma, di conseguenza è stata utilizzata la direttiva `reduction(+:sum)` per eseguire efficientemente tale operazione.
Sono stati introdotti alcuni miglioramenti, tra cui l'implementazione di ghost cells (halo) per rendere più efficiente la computazione. L'introduzione dell'halo è così implementato: si riportano celle fittizie intorno al dominio effettivo, assegnando a queste celle il valore 0, che rappresenta l'energia posseduta da queste celle. Inoltre l'halo non è oggetto di computazione da parte dell'algoritmo, preservando di conseguenza sempre il valore di riferimento 0. Può essere utile comparare il funzionamento dell'algoritmo con e senza halo:

In figura 1 viene mostrato il funzionamento al contorno dell'algoritmo senza l'utilizzo di ghost cells, come si osserva dalla figura, è necessario che le celle di bordo non accedano fuori dal dominio; prendendo come caso la cella in alto a sinistra, quest'ultima non deve accedere alla cella superiore ne a quella di sinistra. Per far ciò è necessario introdurre dei controlli che verifichino quale cella si sta elaborando e di conseguenza escludere l'accesso a celle non consentite, rendendo più complesso l'algoritmo. In figura 2 si illustra il funzionamento dell'algoritmo con l'utilizzo dell'halo. Questa versione ci consente di ottimizzare e rendere leggermente più efficiente l'algoritmo in quanto il problema precedentemente illustrato non si pone; dal momento che vengono elaborate e valutate solo le celle bianche della figura, possiamo accedere in ogni caso a tutte le celle adiacenti senza preoccuparci del fatto che si potrebbero fare accessi *out-of-bound*, poiché l'halo ci garantisce che sia sempre presente una cella sul bordo. Infine, avendo ogni cella come valore 0, il controllo che viene effettuato per verificare che la cella che si sta controllando abbia superato la soglia di $EMAX$ non si verificherà mai e quindi non si compromette in alcun modo la correttezza dell'algoritmo.

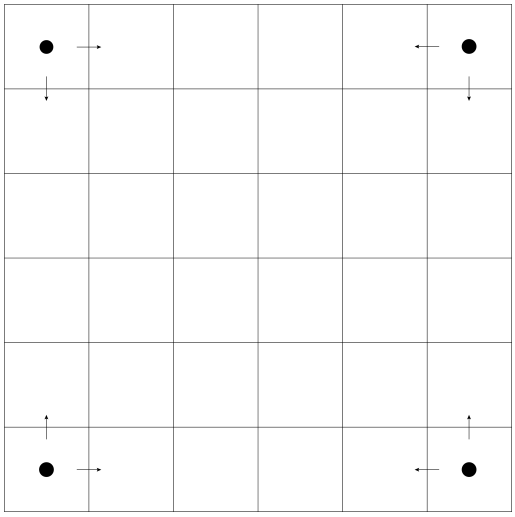

Figura 1. Schema funzionamento algoritmo senza halo. Si mostrano le condizioni al contorno.

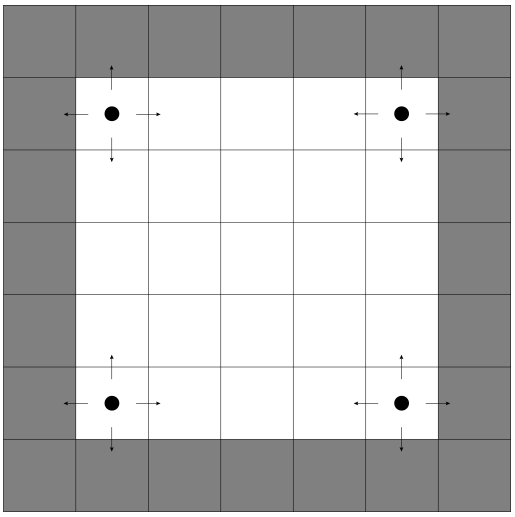

Figura 2. Schema funzionamento algoritmo con halo. Si mostrano le condizioni al contorno.

Sono state pensate altre ottimizzazione come l'utilizzo della clausola `collpse(2)`, nelle funzioni in cui sono presenti due cicli `for` annidati. L'utilizzo della clausola si è rivelata inefficiente in maniera significativa sui tempi di esecuzione del simulatore. I tempi di esecuzione si sono alzati a tal punto che si è preferito parallelizzare solamente il ciclo `for` più esterno, raggiungendo prestazioni migliori. La ragione per la quale la clausola `collapse(2)` risulti in questo caso inefficiente, sarà oggetto di future indagini.

Altre migliorie che possono essere apportate sono per esempio l'utilizzo e la manipolazione dei dati. In questo caso particolare, dovendo manipolare e agire su matrici, è opportuno che l'accesso alle celle sia efficiente in termini di cache. Come dimostrato in vari esempi (cache [2]) l'accesso per righe ad una matrice può aumentare le prestazioni; si è quindi verificato che l'accesso sia di tipo *row-wise* e conseguentemente sfruttare al massimo le *cache-line*, riducendo quindi i tempi di lettura dalla memoria centrale.

```
Performance counter stats for './omp-earthquake 100000 256':

  830,384,198  cache-references:u              ( +-  0.65% )
       36,065  cache-misses:u     # 0.004%     ( +-  8.63% )

       1.5413 +- 0.0243 seconds time elapsed   ( +-  1.58% )
```

Come osserviamo dal report, i *cache-miss* (che rappresentano il degrado delle prestazioni), sono lo 0.004% e quindi una percentuale non significativa in termini di prestazioni.

Infine si è preferito non definire esplicitamente un tipo di scheduling (`static:dynamic`) in quanto a seguito di vari test, lo scheduling `dynamic` ha prodotto un degrado delle prestazioni, mentre lo scheduling `static` non ha segnato particolari differenze rispetto al non utilizzo esplicito. In linea puramente teorica, per il tipo di applicazione che stiamo implementando, risulterebbe migliore uno scheduling di tipo `statico` poiché il tipo di computazione per ogni cella è pressoché identico, quindi l'utilizzo di una politica `dynamic` introduce un overhead a tempo di esecuzione che non produce nessun vantaggio nei tempi di esecuzione, al contrario invece degrada notevolmente le prestazioni.

In figura 3 sono mostrati i tempi di esecuzione, tutti i test sono stati svolti impostando 100000 come numero di steps, mentre le dimensioni del dominio sono: 256, 512 e 1024 (lato dominio).
Si precisa che i test effettuati per OpenMP, sono stati eseguiti su un processore
Intel ® Core ™ i7-6700 e i sorgenti compilati con GCC 8.2.1.

Come si può osservare dal grafico, il risultato migliore lo si ottiene con l'utilizzo di 8 cores, dal momento che il processore prevede 4 cores fisici più tecnologia Intel ® Hyper-Threading. L'overhead introdotto dallo scheduling di più di 8 threads degrada in maniera significativa i tempi di esecuzione. Diversamente da come ci si aspetterebbe, il miglioramento di prestazioni tra l'uso di 4 threads e 8 threads, non è pari al doppio poiché la tecnologia Hyper-Threading promette un incremento di prestazioni fino ad un 30% circa; questo viene dimostrato in maniera abbastanza approssimativa calcolando il divario percentuale tra il tempo impiegato con 4 threads (7.8s) e quello con 8 thread (5.9s) che è del 24.3%.

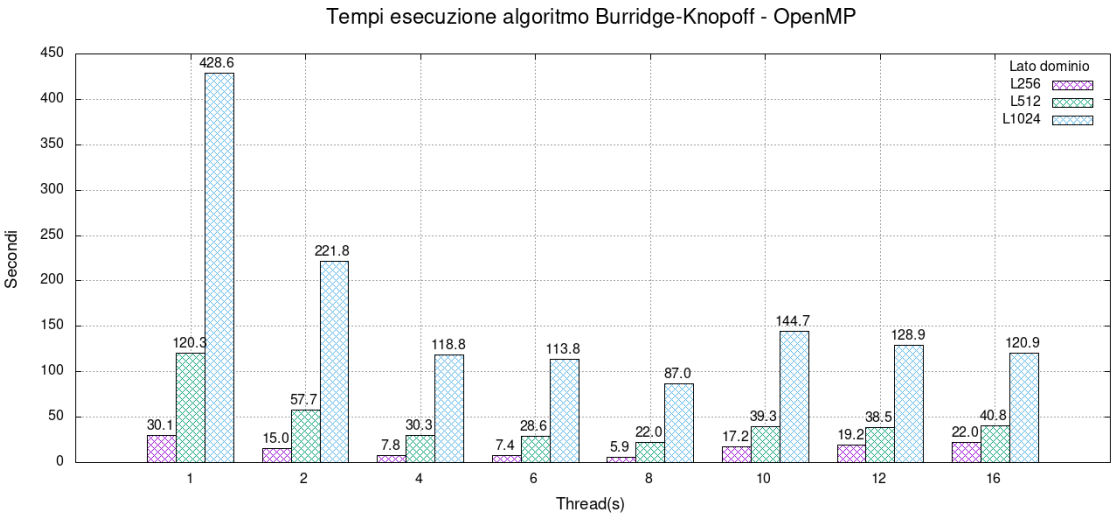

Figura 3. Tempi di esecuzione con OpenMP.

Le considerazioni precedentemente effettuate sono confermate dal grafico di figura 4 che mostra gli speedup con dimensioni del dominio: 256, 512, 1024. Dal grafico osserviamo che il picco massimo lo raggiungiamo in tutti i casi con 8 threads.

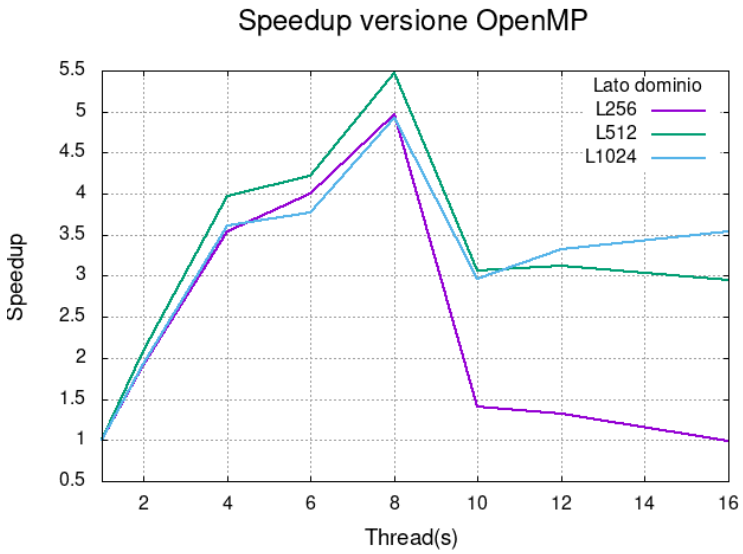

Figura 4. Speedup

In figura 5 viene mostrato il grafico della *Strong Scaling Efficiency*, mentre in figura 6 è mostrato il grafico della *Weak Scaling Efficiency*.

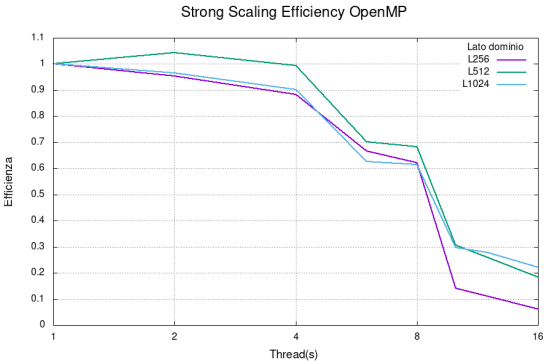

Figura 5. Grafico rappresentante Strong Scaling Efficiency.

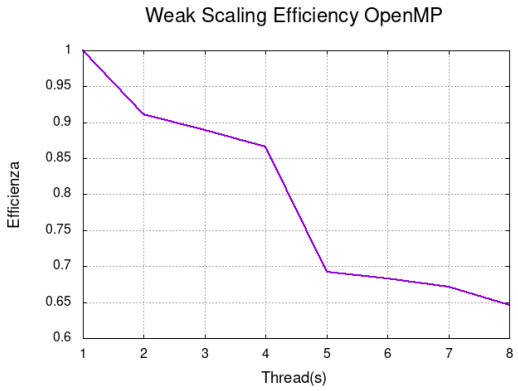

Figura 6. Grafico rappresentante Weak Scaling Efficiency.

3 Versione CUDA

La versione CUDA ha rappresentato una sfida nell'implementazione dei vari Kernel. Come nella precedente versione il primo approccio al problema lo si è svolto perseguendo la linea della semplicità senza soffermars in un primo momento a cercare le ottimizzazioni.
Le prime implementazioni non hanno prodotto speedup significativi, rispetto a quanto ci si possa aspettare con l'utilizzo di GPU; quindi si è proceduto a cercare eventuali inefficienze e colli di bottiglia.
Si è rivelato di fondamentale importanza lo strumento *NVIDIA Visual Profiler* fornito con il compilatore `nvcc`, in quanto è stato possibile analizzare con una cura e precisione sorprendente tutti i kernel. Inoltre il profiler consente di analizzare l'efficienza e tempi di trasferimento della memoria `host/device` e `device/host`, consentendomi di identificare un evidente collo di bottiglia.

Il collo di bottiglia evidenziato è il seguente: nella versione seriale del programma, all'interno del cilco for per l'esecuzione degli steps è presente una `printf()` che consente di stampare le celle con valore superiore a *EMAX* e la media totale di energia presente sulla superficie; inizialmente quindi si copiavano i dati del conteggio celle e della media dal device all'host per ogni steps. Mediante il profiler è stato possibile identificare questa inefficienza e pensare a una soluzione migliore.
La soluzione pensata, anche in accordo con le linee guida NVIDIA (che consigliano di lavorare per tutto il ciclo di esecuzione del software sui dati presenti nel device e solo a termine della computazione copiare i risultati sull'host), prevede l'allocazione di due array (uno per il conteggio celle, l'altro per la media) di dimensione pari al numero di steps, e i kernel `average_energy` e `count_cells` scrivono i risultati appena calcolati su questi due array. Così facendo non chiamiamo mai `cudaMemcpy()` all'interno del ciclo `for` che esegue gli steps, ma viene chiamata una sola volta per la copia dei due array e solo successivamente vengono stampati i valori su `stdout`.
Questa piccola accortezza ha consentito di ridurre di circa qualche secondo i tempi di esecuzione.

Listing 1.1. Copia valori per ogni steps

```
for (unsigned int s = 0; s < nsteps; s++) {
    ...
    count_cells<<<grid1, block1>>>(d_cur, ext_n, d_c);
    ...
    average_energy<<<c_grid, c_block>>>(d_next, d_emean, ext_elem);

    cudaMemcpy(c, d_c, sizeof(int), cudaMemcpyDeviceToHost);
    cudaMemcpy(emean, d_emean, sizeof(float), cudaMemcpyDeviceToHost);
```

```
    printf("%d %f\n", c, emean);
    ...
}
```

Listing 1.2. Copia valori al termine della computazione

```
for (unsigned int s = 0; s < nsteps; s++) {
    ...
    count_cells<<<grid1, block1>>>(d_cur, ext_n, d_c, s);
    ...
    average_energy<BLKSIZE/2>
        <<<c_grid, c_block, sum_buff_size>>>(d_next, d_emean, ext_elem, s);
    ...
}
cudaMemcpy(c, d_c, sizeof(int)*nsteps, cudaMemcpyDeviceToHost);
cudaMemcpy(emean, d_emean, sizeof(float)*nsteps, cudaMemcpyDeviceToHost);
for (unsigned int s = 0; s < nsteps; s++) {
    printf("%d %f\n", c[s], emean[s]/(n*n));
}
```

Un'altra ottimizzazione che è stata effettuata riguarda l'operazione di riduzione nel kernel `average_energy`; questa implementazione è stata realizzata prendendo esempio da *Optimizing Parallel Reduction in CUDA - Nvidia* [3]. Questa implementazione fa uso della `Shared-Memory` allocata dinamicamente, in questo modo possiamo, a tempo di esecuzione, allocare dinamicamente il vettore condiviso andando a migliorare le prestazioni sia in termini di efficienza che di spazio allocato.
L'implementazione di questa versione ha portato ad un incremento di prestazione notevole, riducendo in maniera significativa i tempi di esecuzione, inoltre il profiler non ha rilevato alcuna anomalia (come branch-divergence, pattern di accesso alla memoria, ecc.) nella realizzazione ed esecuzione di questo kernel.

Infine, sempre mediante l'uso del profiler, si è osservato che a livello computazionale il kernel `propagate_energy` è risultato quello con il carico di lavoro maggiore e quindi quello che segna i tempi di esecuzione maggiori. Su questo kernel il profiler segnala due criticità in particolare: *branch-divergence* e *pattern di accesso alla memoria inefficiente*.

Come trattato in diversi articoli inerenti la programmazione CUDA, gli `if-else` causano quello che si definisce *branch-divergence* ovvero thread dello stesso Warp si "aspettano" in quanto il corpo dell'if e quello dell'else vengono eseguiti in maniera sequenziale e non parallela come ci si potrebbe aspettare. Questo comporta un degrado delle prestazioni abbastanza importante. La soluzione che è stata pensata per sopperire a questo problema è quello di trasformare il costrutto `if-else` in una espressione:

Listing 1.3. Costrutto if presente nella versione seriale

```
if (*IDX(cur, i, j-1, ext_n) > EMAX) {
    F += FDELTA;
}
```

Listing 1.4. Trasformazione dell cotrutto if in una espressione

```
F += (float)(*IDX(cur, i, j-1, ext_n)>EMAX) * FDELTA;
```

Con la trasformazione in espressione trasformiamo in un `float` la condizione `IDX(cur, i, j-1, ext_n)>EMAX` che assumerà valore 1.0 nel caso in cui sia vera, 0.0 nel caso in cui sia falsa; dal momento che vogliamo che ad `F` venga sommato `FDELTA`, se la condizione è vera viene moltiplicato 1.0 per `FDELTA` che banalmente produce sempre `FDELTA`; se la condizione è falsa verrà moltiplicato 0.0 per `FDELTA` che produce 0.0 come risultato e quindi non altera in alcun modo il risultato di `F`.

In questo modo si è eliminata la problematica del *branch-divergence*, ciò nonostante i miglioramenti in termini di esecuzione sono trascurabili.

Per quanto riguarda la problematica del pattern di accesso alla memoria, il problema su cui stiamo lavorando non ci consente di accede a dati adiacenti, poiché le condizioni controllano le

celle laterali e quelle superiori e inferiori, il problema si cela proprio sulle ultime due in quanto non sono adiacenti in memoria e questo causa un pessimo accesso ai dati, causando inefficienze in fase di lettura dei dati. Per questo motivo il kernel `propagate_energy` è stato implementato sia nella versione senza *shared-memory* che nella versione con *shared-memory*; quest'ultima versione, inaspettatamente, non produce un miglioramento nell'accesso ai dati e quindi nemmeno un miglioramento delle prestazioni.

Altra inefficienza presente è dovuta alle innumerevoli chiamate dei vari kernel, dal momento che il numero degli steps è molto elevato, richiamare i kernel per ogni step introduce un overhead drammatico: le chiamate a kernel sono operazioni molto costose poiché impiegano da qualche microsecondo a diverse decine di microsecondi (in base al kernel che si deve invocare); come accennato in precedenza le chiamate sono molte e questo provoca un aumento abbastanza significativo dei tempi di esecuzione.

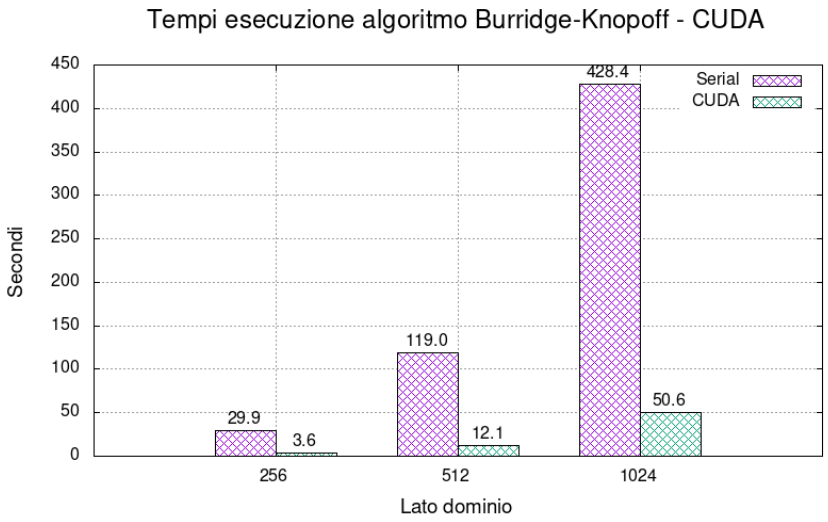

Figura 7. Tempi di esecuzione con CUDA

In figura 7 vengono rappresentati i tempi della versione seriale, comparati con la versione CUDA (per tutti i casi il numero di steps è fissato a 100000), i risultati ottenuti sono soddisfacenti, ottenendo come speedup massimo un valore pari a **9.83** con lato del dominio pari a 512.
Tutti i test eseguiti con CUDA sono stati effettuati su una GPU *NVIDIA GeForce GTX 1070*.

4 Conclusioni

I risultati ottenuti sono discreti, ulteriori modifiche sono sicuramente possibili, ma per le conoscenze e competenze acquisite i risultati sono emersi. Nonostante le difficoltà incontrate nell'implementazione il risultato è più che soddisfacente.

Riferimenti bibliografici

1. R. Burridge, L. Knopoff; Model and theoretical seismicity. Bulletin of the Seismological Society of America; 57 (3): 341–371 (1967)
2. https://www.moreno.marzolla.name/teaching/HPC/L01-parallel-architectures.pdf
3. https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf