Progetto di High Performance Computing

Fabrizio Margotta (fabrizio.margotta@studio.unibo.it)
Matricola 789072

Alma Mater Studiorum-Università di Bologna

30/09/2019

Sommario

Lo scopo del progetto è di implementare versioni parallele di un semplice modello matematico di propagazione dei terremoti. Il modello che consideriamo è una estensione in due dimensioni dell'automa cellulare Burridge-Knopoff (BK).[2]

1 Introduzione

La realizzazione dell'intero progetto è avvenuta per fasi: inizialmente è stata implementata la versione OpenMP, solo in seguito la versione CUDA. Per ogni fase si è proceduti allo sviluppo della soluzione proposta per raffinamenti successivi (ciclo di design APOD[3]) e, contestualmente, alla scrittura della relazione.

L'intero progetto è stato gestito seguendo una versione semplificata della metodologia Kanban[5], utilizzando opportuni strumenti digitali per l'organizzazione delle singole attività di lavoro.

1.1 Analisi

L'analisi sul modello da implementare è stata particolarmente semplificata dalle indicazioni fornite dal docente.

Vengono tuttavia di seguito presentate alcune osservazioni sul modello dei dati: si noti ad esempio che nei cicli non è presente alcuna specifica dipendenza sui dati.

Ciò permette quindi di procedere con la parallelizzazione della versione seriale fornita dal docente senza particolari problemi.

2 Sviluppo

2.1 Ghost area

Entrambe le soluzioni proposte si avvalgono dell'utilizzo di una ghost area inizializzata a zero (e mai più modificata).

Ciò permette di ridurre notevolmente l'impatto sulle prestazioni dell'operazione di propagazione, poiché, avendo una nuova regione di celle intorno al dominio, non sarà più necessario controllare che le celle adiacenti a quella presa in esame siano ai bordi del dominio stesso. Il valore delle ghost cell (zero), inoltre, non influirà sui risultati delle operazioni dell'algoritmo.

In figura 1 [4] è possibile osservare il comportamento appena descritto.

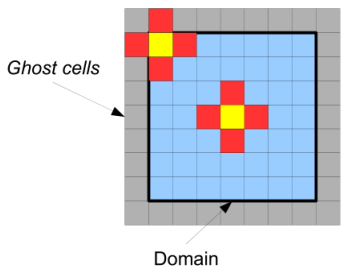


Figura 1: Rappresentazione della ghost area (celle grigie).

2.2 Accesso in memoria al dominio

È stato inizialmente ipotizzato l'accesso alla matrice considerandola come un array, in accordo con la rappresentazione in memoria delle matrici fornita dal linguaggio C. Ciò avrebbe permesso di scorrere in modo sequenziale la matrice stessa, risparmiando le chiamate alla funzione `IDX`.

Ad esempio, l'operazione di incremento di energia sarebbe stata implementata come segue:

```
for (int i = 0; i < n; i++) {
    grid[i] += delta;
}
```

Tuttavia, l'introduzione della *ghost area* ha reso preferibile (in termini di gestione dell'accesso alla memoria) l'utilizzo della funzione `IDX`, che contribuisce inoltre ad una buona comprensione e leggibilità del codice sorgente.

2.3 Versione OpenMP

In figura 2 è possibile osservare il risultato della simulazione espresso graficamente come l'andamento dell'energia media delle celle e del numero di celle che hanno superato il valore soglia.

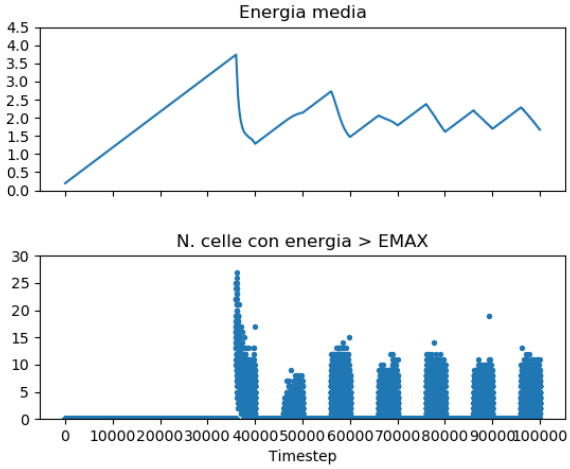


Figura 2: Energia media e numero di celle con energia maggiore di `EMAX` in un modello BK, n=256, 10^5 timestep

2.3.1 Modifiche alla versione seriale

setup L'inizializzazione della ghost area a zero non è stata parallelizzata poiché i test effettuati e mostrati in tabella 1 hanno evidenziato un degradamento, seppur minimo, delle prestazioni.

	Lato dominio	Numero passi	T_{setup} (s)
seriale	512	1000	0.00337177
parallelizzato			0.00454419

Tabella 1: Tempi di esecuzione della funzione `setup` nella versione seriale e nella versione parallelizzata.

Tale comportamento può essere dovuto ad un ulteriore carico di lavoro assegnato allo scheduler per un'attività che mal si presta (soprattutto per l'accesso ai lati sinistro e destro del dominio a causa dell'accesso in memoria) ad essere parallelizzata.

altre operazioni Tutte le principali operazioni (incremento, conteggio, propagazione e calcolo energia media) contengono dei cicli `for` che sono stati parallelizzati utilizzando la direttiva OpenMP `omp parallel for`. In tale modo l'intero carico di lavoro del ciclo a cui è stato applicato il costrutto viene suddiviso tra i thread che concorrono all'esecuzione del programma, realizzando la vera e propria parallelizzazione.

Nelle operazioni di conteggio e di calcolo dell'energia media è stata inoltre introdotta la clausola di riduzione con l'operatore somma, per poter parallelizzare anche le operazioni di somma all'interno delle direttive OpenMP.

2.3.2 Ulteriori ottimizzazioni

collapse È stato valutato l'utilizzo della clausola `collapse(2)` nei cicli `for` innestati per collassare due iterazioni in una, allargando lo spazio di iterazione come da specifiche[1].

Tale clausola ha tuttavia causato una degradazione delle prestazioni e pertanto non è stata implementata nella soluzione proposta.

	Lato dominio	Numero passi	$\overline{T}_{\text{3-run}}(s)$	cache-references (M/sec)	cache-misses (%)[1]
senza `collapse(2)`	256	100000	13.33183	0.770	0.390
con `collapse(2)`			19.18336	0.560	0.402

Tabella 2: Dati raccolti sull'esecuzione della simulazione con e senza la clausola `collapse(2)`.

Dai dati presenti in tabella 2 è possibile osservare che, al contrario di come ci si aspetterebbe, nonostante la clausola `collapse` modifichi i *chunk* di memoria a cui si accede, ciò non comporta un degradamento significativo delle prestazioni nell'accesso alla memoria cache.

Di conseguenza si ipotizza che l'incremento delle tempistiche di esecuzione possa essere legato ad una cattiva gestione da parte dello scheduler nell'organizzare il lavoro dei thread.

schedule È stata inoltre valutata l'adozione della clausola `schedule`, utilizzando il parametro `type` nelle sue varianti `static`, `dynamic` e `guided`.

Le prove effettuate hanno mostrato un significativo aumento delle prestazioni impostando uno scheduling statico con blocchi di dimensione 8.

La tabella 3 mostra la differenza di tempistiche medie (su 3 esecuzioni) tra la versione in cui la clausola `schedule` non è specificata e quella in cui viene usata con parametri `static, 8`.

Lato dominio	Numero passi	$\overline{T}_{\text{3-run}}(s)$	$\overline{T}_{\text{3-run}}(s)$ con `schedule(static,8)`
256	100000	12.9581	10.3148
512		50.4409	40.5048
1024		250.4084	177.2195

Tabella 3: Comparazione delle tempistiche di esecuzione applicando uno scheduling statico.

Tale clausola ha quindi permesso una importante riduzione delle tempistiche, compresa tra il 20 e il 30% in meno rispetto alla precedente versione già parallelizzata.

Altri tipi di scheduling o diverse dimensioni dei *chunk* hanno pareggiato o addirittura peggiorato le precedenti tempistiche.

2.4 Versione CUDA

L'approccio all'implementazione della versione CUDA è stato più articolato, poiché ha coinvolto l'inizializzazione e l'interazione con il dispositivo. Di seguito si riporta il flusso di lavoro adottato nella programmazione della versione CUDA della soluzione proposta:

1. modifica delle operazioni dell'algoritmo: ogni funzione esprime il lavoro svolto dal singolo thread
2. allocazione delle porzioni di memoria necessarie a contenere il dominio;
3. partizionamento del dominio ed indicizzazione dei thread in accordo con il modello di suddivisione del carico di lavoro del dispositivo;
4. inizializzazione del dominio e copia delle variabili nella memoria del dispositivo;
5. esecuzione delle operazioni dell'algoritmo, se necesario sincronizzando i thread del dispositivo tra un'operazione e l'altra e scambiando i dati relativi all'esecuzione di una singola computazione
6. deallocazione della memoria sia sull'*host* che sul *device*

[1] of all cache refs

2.4.1 Partizionamento del dominio

Il dominio è stato partizionato, come da suggerimenti, in blocchi 2D organizzati a loro volta in griglie 2D.

Per le operazioni effettuate nelle funzioni `count_cells` e `average_energy` il dominio è stato invece partizionato in blocchi monodimensionali, in accordo con la rappresentazione matriciale nel linguaggio di programmazione C.

2.4.2 Riduzione

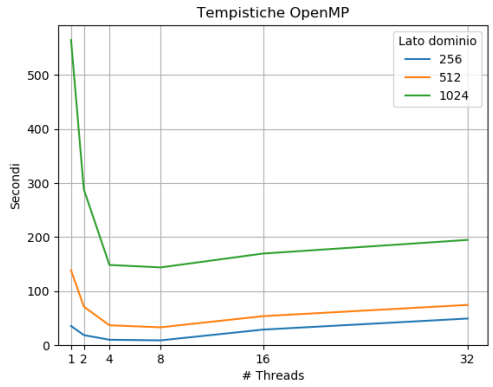
È stato inoltre adottato il pattern di riduzione con operazione somma mediante l'API `atomicAdd`, di fatto interpretando le operazioni di conteggio e di calcolo dell'energia media come somma degli elementi di un array.

3 Valutazione prestazioni

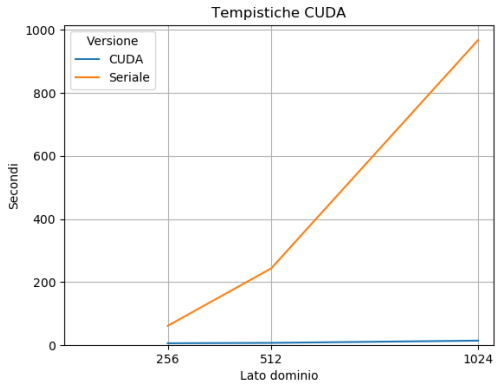
I test della versione OpenMP sono stati effettuati su una macchina equipaggiata con una CPU Intel® Core™ i7–4710HQ quad core (8 thread) @ 2.50GHz ed una NVIDIA GeForce GTX 860M.

La valutazione delle prestazioni per la versione CUDA è stata invece effettuata su una macchina equipaggiata con due processori Intel® Xeon® E5–2603 v4 hexa core @ 1.70GHz e tre GPU NVIDIA GeForce GTX 1070.

Le tempistiche indicate sono state tutte ottenute mediante le funzioni di libreria messe a disposizione dal docente. Per ulteriori informazioni si consulti il codice sorgente.



(a) Comparazione tempistiche di esecuzione in OpenMP.



(b) Comparazione tempistiche di esecuzione in CUDA.

Figura 3: Comparazione delle tempistiche di esecuzione dell'algoritmo.

In figura 3a è possibile notare che nell'esecuzione dell'algoritmo implementato con parallelismo OpenMP il maggior guadagno si registra nell'esecuzione in cui il dominio ha dimensione 1024 e che nelle esecuzioni che usano più di 8 thread si ha un incremento di tempistiche. Ciò è ancora una volta riconducibile ad un eccessivo carico di lavoro che lo scheduler deve affrontare durante l'esecuzione del programma.

In figura 3b, invece, è possibile osservare come la versione CUDA vada ad abbattere drasticamente i tempi di esecuzione dell'algoritmo, denotando un ingente guadagno prestazionale.

3.1 Speedup

Il calcolo dello speedup viene eseguito mediante la seguente formula:

$$S(p) = \frac{T_{serial}}{T_{parallel}(p)}$$

in cui:

p	:	# processori/core
T_{serial}	:	tempo di esecuzione dell'algoritmo in versione seriale (T_{serial}=$T_{parallel}(1)$)
$T_{parallel}(p)$	:	tempo di esecuzione della versione parallela con p processori/core

Poiché l'implementazione della soluzione contiene porzioni di codice non parallelizzabili (si veda la funzione `setup`), dovremmo considerare $T_{parallel}(p)$ nel seguente modo:

$$T_{parallel}(p) = \alpha \cdot T_{serial} + \frac{(1-\alpha) \cdot T_{serial}}{p}$$

in cui α è il fattore relativo alla porzione di codice non parallelizzabile, calcolato come segue:

$$\alpha = \frac{T_{serial}}{T_{serial} + T_{parallel}}.$$

Tuttavia, come mostrato in tabella 4, la porzione di codice seriale (non parallelizzabile) impiega tempo trascurabile rispetto alla porzione di codice parallelizzata, pertanto anche α assume valore trascurabile.

Lato dominio	Numero passi	$T_{seriale}$ (s)	$T_{parallelo}$ (s)	α
256	100000	0.00140063	14.2519	0.00009827
1024		0.02356447	150.2037	0.00015686

Tabella 4: Comparazione delle tempistiche per le porzioni seriale e parallele dell'algoritmo.

Per le siffatte osservazioni, considereremo $T_{parallel}(p)$ in tal modo:

$$T_{parallel}(p) = \frac{T_{serial}}{p}.$$

In figura 4 è possibile osservare che lo speedup cresca fino a 8 thread, mantenendo di fatto la tendenza registrata nella misurazione delle tempistiche.

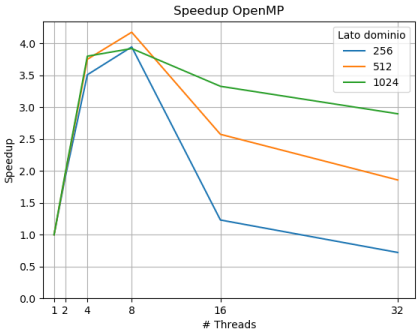


Figura 4: Grafico dello speedup della versione OpenMP.

3.2 Scalabilità

3.2.1 Scalabilità forte

La valutazione della scalabilità forte (*strong scaling*), nonché i risultati mostrati in figura 5a sono stati calcolati mediante la seguente formula:

$$E(p) = \frac{S(p)}{p} = \frac{T_{serial}}{p \cdot T_{parallel}(p)}.$$

3.2.2 Scalabilità debole

La valutazione della scalabilità debole (*weak scaling*), nonché i risultati mostrati in figura 5b sono stati calcolati mediante la seguente formula:

$$W(p) = \frac{T_1}{T_p}$$

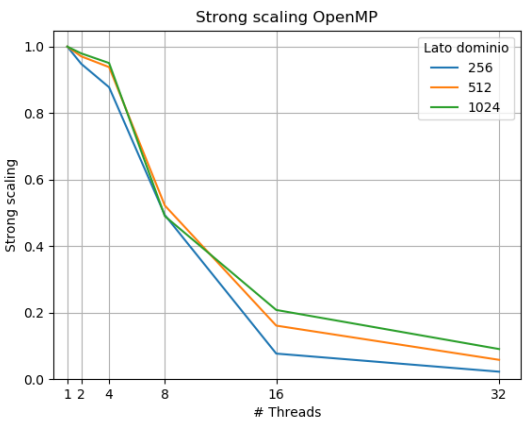
in cui:

p : # processori/core
T_1 : tempo di esecuzione di una unità di lavoro con un processore/core
T_p : tempo di esecuzione di p unità di lavoro con p processori/core

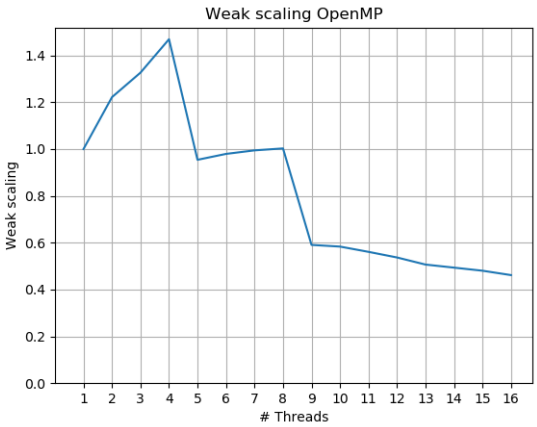
Poiché nella valutazione della scalabilità debole la dimensione del problema è proporzionale al numero di processori/core utilizzati, il lato della matrice utilizzata nell'algoritmo è stato calcolato come segue:

$$side(p) = 256 \cdot \sqrt[3]{p}$$

in cui 256 è la dimensione presa come lato di base.



(a) Grafico della scalabilità forte.



(b) Grafico della scalabilità debole.

Figura 5: Grafici che rappresentano la scalabilità della versione OpenMP.

4 Conclusioni

Tale relazione di progetto vuole portare all'attenzione tutte le fasi di realizzazione del progetto stesso: l'analisi del modello, l'implementazione delle versioni parallele e la valutazione delle prestazioni seguendo un approccio quanto più metodico possibile.

Ogni sezione della relazione è inoltre accompagnata da considerazioni svolte durante l'implementazione stessa, come, ad esempio, l'utilizzo di pattern, clausole o l'adozione di una variante di una formula piuttosto che un'altra in base a determinate condizioni. Tali scelte sono state inoltre motivate con dati raccolti durante le varie verifiche in corso d'opera.

4.1 Note di sviluppo

Sono stati realizzati diversi script che facilitano ed automatizzano l'esecuzione su server remoti e la valutazione delle prestazioni.

Possono risultare di interesse i meccanismi ideati per trasferire, compilare, eseguire l'algoritmo e scaricare i risultati da/verso server remoti utilizzando il comando `scp` e l'opzione `ProxyJump` di SSH.

In ultimo sono stati realizzati degli script per l'automatizzazione del calcolo dei parametri prestazionali quali tempistiche, speedup e scalabilità.

Si consultino i sorgenti per maggiori informazioni.

Riferimenti bibliografici

[1] OpenMP Architecture Review Board. *OpenMP Application Programming Interface*, 5.0 edition, November 2018.

[2] Robert Burridge and Leon Knopoff. Model and theoretical seismicity. *Bulletin of the seismological society of america*, 57(3):341–371, 1967.

[3] NVIDIA Corporation. Cuda toolkit v10.1.243, best practices guide, Assess, Parallelize, Optimize, Deploy. [Online; accessed 29-September-2019].

[4] Moreno Marzolla. Lucidi delle lezioni per il corso di High Performance Computing: Parallel programming patterns, 38, September 2018.

[5] Wikipedia contributors. Kanban (development) — Wikipedia, the free encyclopedia, 2019. [Online; accessed 25-September-2019].