Studio Definitivo Progetto LAR Splitting 2D – CPD22

Gruppo: 5.a – Castagnacci Giulia 581749, Giordano Elisabetta 536265

June 3, 2022

Link Repository GitHub: https://github.com/GiuliaCastagnacci/LARSplitting2D.git

Link relazioni precedenti:

- al seguente link è presente una descrizione accurata delle funzioni presenti nella classe *refactoring.jl*: https://github.com/GiuliaCastagnacci/LARSplitting2D/blob/main/docs/relazio
- al seguente link è presente una prima analisi delle funzioni prese in esame: https://github.com/GiuliaCastagnacci/LARSplitting2D/blob/main/docs/relazioni/studioEsecutivo.md

Contents

Introduzione

Lo scopo principale del progetto è stato quello di migliorare le prestazioni delle funzioni prese in esame, attraverso l'utilizzo di metodi di parallelizzazione. Inoltre, per garantire una migliore prestazione delle funzioni è stato effettuato un refactoring di alcune funzioni, scomponendo il codice in funzioni elementari.

Inizialmente è stata svolta un'analisi preliminare delle funzioni con l'obiettivo di misurarne le prestazioni e individuare eventuali porzioni di codice che potessero rallentare i tempi di esecuzione, facendo riferimento a quanto discusso in aula durante le lezioni e ai capitoli del libro consigliato **Julia High Performace**.

In questa fase, sono state utilizzate le macro:

- *@btime*: calcola e stampa il tempo trascorso durante l'esecuzione della funzione;
- *@code_warntype*: consente di visualizzare i tipi dedotti dal compilatore, individuando così ogni tipo di instabilità nel nostro codice;
- *@benchmark*: valuta i parametri della funzione separatamente, quindi chiama la funzione più volte per creare un campione dei tempi di esecuzione.

Per ottimizzare e parallelizzare le funzioni sono state principalmente usate le seguenti macro:

- *@threads*: questa macro è apposta davanti a un for per indicare a Julia che il loop è una regione multi-thread;
- *@spawn*: è uno degli strumenti che Julia mette a disposizione per assegnare compiti ai lavoratori;
- *@inbounds*: può essere applicata prima di una funzione o di una definizione di ciclo. Il vantaggio in termini di prestazioni di questa operazione è piccolo in termini assoluti, ma può essere complessivamente significativo per i circuiti interni

Macro utilizzate per la definizione dei task:

- *@async*: racchiude l'espressione in un Task ed inizierà con l'esecuzione di questa attività procedendo con qualsiasi altra cosa venga dopo nello script, senza aspettare che il Task termini;
- *@sync*: contrariamente al precedente, questa macro aspetta che tutti i Task creati dalla parallelizzazione siano completati prima di proseguire;

- *Thread.@spawn*: crea un Task e schedula l'esecuzione su un qualsiasi thread disponibile. Il Task viene assegnato ad un Thread quando diventa disponibile.

Abbiamo quindi definito i task che ci consentono di sospendere e riprendere i calcoli per l'I/O, la gestione degli eventi e modelli simili. I task possono sincronizzarsi attraverso operazioni come wait e fetch e comunicare tramite canali. Pur non essendo di per sé un calcolo parallelo, Julia consente di programmare i task su più thread.

Nel paragrafo successivo sono riportate le funzioni da noi analizzate. In particolare viene riportato come abbiamo ottimizzato e parallelizzato le funzioni, dove è stato effettuato un refactoring del codice e i risultati ottenuti con l'utilizzo della CPU e della GPU del superserver **Nvidia DGX-1**.

Analisi funzioni ottimizzate e metodologia di parallelizzazione

boundingbox

La versione iniziale era type unstable; questo era dovuto alla funzione mapslices (funzione di base). Per ovviare al problema, si è effettuato un refactoring della funzione, riscrivendola di fatto da zero. In questo modo si è ottenuta la stabilità di tipo e un notevole miglioramento delle prestazioni da 45.000 μs a 7.180 μs.

Risultati ottenuti con la macchina Nvidia DGX-1: da 10.670 μs a 4.621 μs

boxcovering

Abbiamo parallelizzato la funzione boxcovering utilizzando la macro @thread prima del ciclo for, e aggiungendo delle funzioni di supporto:

- *createIntervalTree*: dato un dizionario ordinato crea un intervalTree, ovvero una struttura dati che contiene intervalli e che consente di trovare in modo efficiente tutti gli intervalli che si sovrappongono a un determinato intervallo o punto;
- *addIntersection*: aggiunge gli elementi di iterator nell'i-esimo array di covers.

Risultati ottenuti con la macchina Nvidia DGX-1: da 3.468 μs a 6.708 μs

congruence

Abbiamo paralellizzato la funzione attraverso le macro @inbounds e @threads. Si nota un certo miglioramento nelle performance se al posto di usare la list comprehension per ottenere i dati di EV validi si fa un filter. Abbiamo quindi

convertito alcune list comprehension in cicli del tipo *for i=1:n,* in modo da poter utilizzare la macro @inbounds per disabilitare il boundchecking del compilatore.

Risultati ottenuti con la macchina Nvidia DGX-1: da 14.768 μs a 26.165 μs

linefragments

Abbiamo paralellizzato la funzione attraverso la macro @threads, notando un lieve miglioramento delle performance.

Risultati ottenuti con la macchina Nvidia DGX-1: da 25.588 μs a 50.046 μs

pointInPolygonClassification

Nella versione parallelizzata abbiamo creato una nuova funzione per ogni edge, sostituendo i relativi if else a catena di pointInPolygonClassification con le rispettive chiamate alle nuove funzioni. Inoltre, nella funzione principale abbiamo inserito la macro @async prima del ciclo for. Le prestazioni della funzione parallelizzata migliorano nettamente rispetto a quella iniziale: da 261.128 ns a 86 ns.

Risultati ottenuti con la macchina Nvidia DGX-1: da 94.987 μs a 94.714 μs

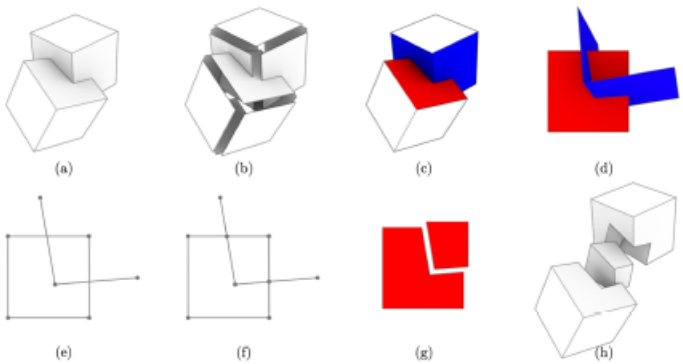

Figure 1: Esempio splitting

Lo splitting avviene mediante la riduzione della faccia di interesse e tutte quelle di possibile intersezione con lei ai loro spigoli di intersezione con il piano z=0. Nella figura, la faccia rossa viene ridotta ai suoi spigoli di bordo, così come le altre incidenti vengono ridotte agli spigoli di bordo (f); generato il grafo avente come nodi tutte le possibili intersezioni questo viene ridotto alla sua componente di lunghezza massimale e su questa vengono estratte le due celle. Nella figura sottostante (punto (d)) è rappresentato il risultato dello splitting: per ciascuna delle celle originali si producono le celle della sua suddivisione ma anche tutti gli spigoli suddivisi e tutti i vertici indotti dalla suddivisione.

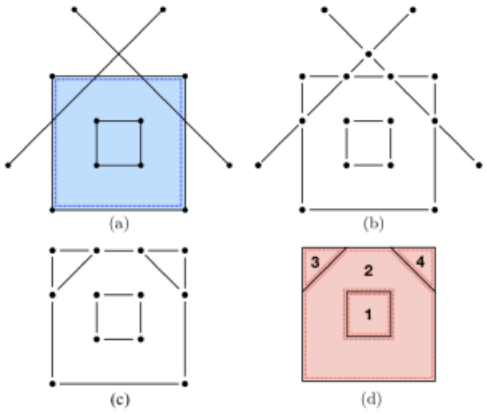

Figure 2: Risultato splitting

spaceindex

Per la parallalizzazione abbiamo usato la macro @spawn e creato delle funzioni aggiuntive di supporto, createIntervalTree (già citata in boxcovering) e removeIntersection, funzione che elimina le intersezioni di ogni bounding box con loro stessi. Grazie a questo, le prestazioni hanno subito un notevole miglioramento, passando da 123 μs a 60 μs.

Risultati ottenuti con la macchina Nvidia DGX-1: da 82.399 μs a 56.714 μs

fragmentlines

Per la funzione fragmentlines abbiamo migliorato le performance con i seguenti passi:

1. Abbiamo convertito la list comprension della creazione dei vettori “params” in un ciclo for in modo da poter utilizzare la macro @threads (in quanto task parallelizzabile)
2. Abbiamo allocato line1 e line2 fuori dal for in modo tale che venissero distrutti e riallocati ad ogni iterazione
3. Abbiamo eliminato la variabile “fragparams” presente nell’ultima porzione di codice, in quanto riallocava la stessa informazione contenuta in ’params’ eliminando solo i doppioni. Inoltre, abbiamo aggiunto la macro threads a tutte le task parallelizzabili.
4. Abbiamo convertito l’ultimo for each in un for i=1:n

Risultati ottenuti con la macchina Nvidia DGX-1: da 187.045 μs a 184.982 μs

coordintervals

Per la funzione coordintervals essendo molto semplice, non sono serviti grandi interventi. Si è provato a convertire il ciclo nel tipo i=1:n per poter usare @inbounds ma cambiamento non ha provocato grandi miglioramenti.

Test

Una volta parallelizzato e ottimizzato le funzioni siamo passate alla fase di testing per verificare il corretto funzionamento del codice.

Inizialmente sono stati eseguiti i test pre-esistenti e, dopo aver verificato il successo di questi, si è proceduto alla realizzazione di nuovi test:

- @testset "createIntervalTree test": creato un OrderedDict e un intervaltrees vogliamo testare che i dati siano stati disposti nel giusto ordine nella struttura dati. Per farlo estraiamo i singoli valori e li confrontiamo con i valori che ci aspettiamo di trovare nelle singole locazioni.
- @testset "removeIntersection test": avendo isolato il task della funzione spaceindex che rimuove le intersezioni dei singoli boundingbox con se stesso, vogliamo assicurarci che funzioni nel modo corretto. Per farlo creiamo un array covers di test e controlliamo che la funzione modifichi la struttura dati nel modo corretto per ogni valore.
- @testset "addIntersection test": avendo isolato il task della funzione boxcovering che aggiunge in 'covers' in i-esima posizione tutti i bounding box che intersecano l'i-esimo bounding box, vogliamo assicurarci che funzioni nel modo corretto. Per farlo creiamo un boundingbox di test e un OrderedDict con cui creare un intervalTree. A questo punto diamo queste variabili come input alla nostra funzione e confrontiamo il risultato ottenuto con quello atteso.
- test "spaceindex": Per provare l'effettivo funzionamento di spaceindex bisogna controllare se il boundingbox i-esimo interseca tutti i boundingbox contenuti nella variabile covers[i], e se non interseca quelli che non sono contenuti in covers[i]. Di ogni boundingbox conosciamo una coppia di punti (N1, N2) che sono i punti opposti del parallelepipedo rispettivamente più vicini e più lontani dall'origine degli assi. Un boundingbox A interseca un boundingbox B se A contiene B (o viceversa), oppure se almeno uno dei suoi punti si trova all'interno di B, cioè se, per ogni coordinata x, y e z, vale che N1b <= Na <= N2b

Esempio

Un esempio di visualizzazione del lavoro svolto nello splitting è il seguente :

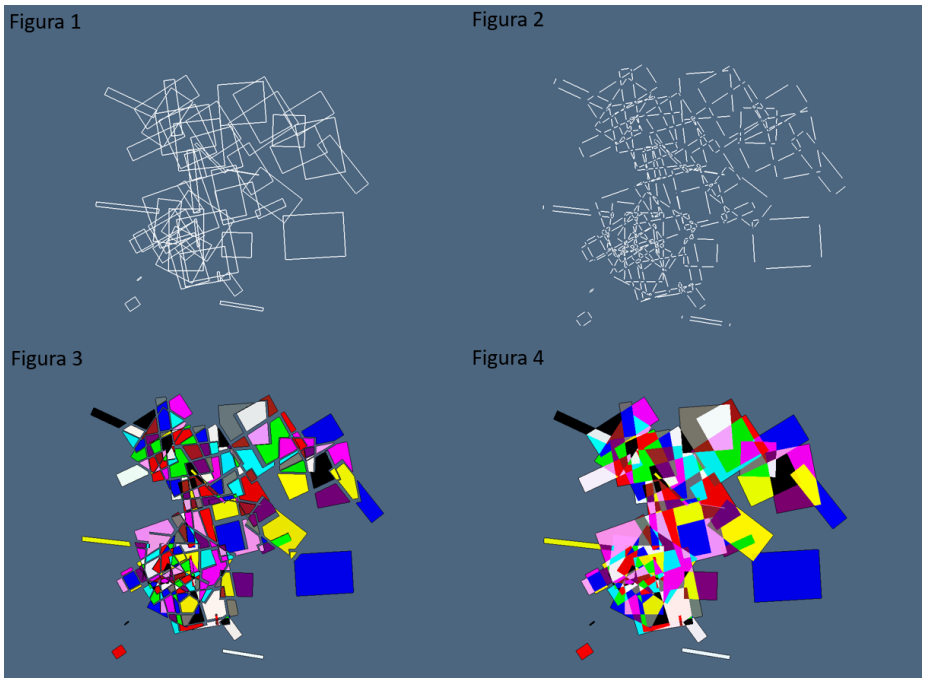

Figure 3: Esempio di visualizzazione

Si vede come l'elemento i-esimo rappresenta quali intersezioni ha il bounding box i-esimo con gli altri bounding box. Successivamente viene eseguita l'intersezione di coppie di segmenti di linea: in particolare i nuovi punti generati dall'intersezione tra spigoli.

Conclusioni

In generale, abbiamo notato che utilizzando le macro @inbounds e @thread non sempre le prestazioni migliorano. Inoltre, in alcuni casi abbiamo provato ad utilizzare la macro @async, notando un notevole peggioramento delle prestazioni; quindi, abbiamo deciso di non inserirlo nel codice già ottimizzato.

Grafo delle dipendenze

la funzione edge_code1_15 rappresenta le 15 funzioni create per l'ottimizzazione di pointInPolygon.

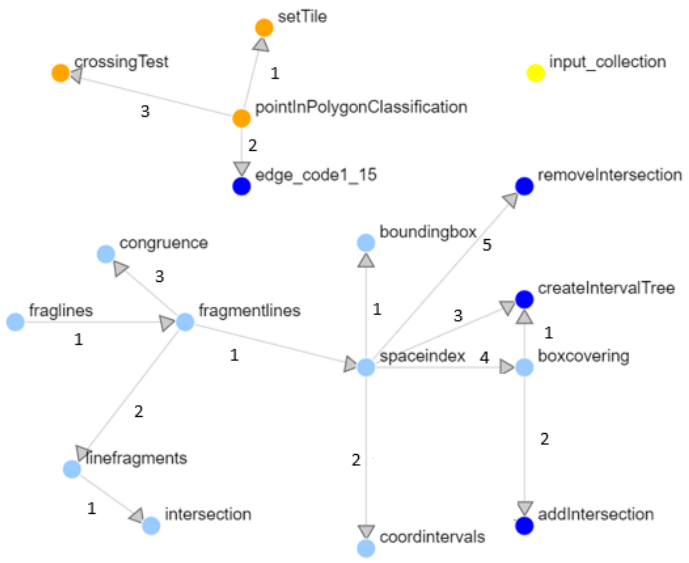

Figure 4: Grafo delle dipendenze