Studio Definitivo Progetto LAR Splitting 2D – CPD22 % Gruppo: 5.b – Caponi Marco 508773, Ceneda Gianluca 488257 % June 13, 2022

Prime analisi, test e possibili ottimizzazioni sul progetto LAR SPLITTING 2D con l'utilizzo della seguente repository:

- Main repository: https://github.com/MarcoCap13/LAR-SPLITTING-2D-5.b-
  - https://github.com/cvdlab/LinearAlgebraicRepresentation.jl/blob/master/src/refactoring.jl

Contents

Obiettivi:

- Studio del progetto **LAR SPLITTING 2D** e di tutte le funzioni e strutture dati utilizzate.
- Descrizione di ogni task individuata, tipo e significato di ogni parametro ed eventuale valore di ritorno.
- Suddivisione delle tipologie di funzioni e creazione di grafi delle dipendenze.
- Individuare eventuali problemi riscontrati durante lo studio preliminare del codice.

RELAZIONE DEL PROGETTO

Analisi introduttiva

Lo scopo del nostro progetto è stato quello di studiare e dove possibile migliorare attraverso metodi di parallelizzazione le prestazioni dei metodi e delle funzioni riguardante quest'ultimo. Per far ciò ci siamo mossi creando nuove funzioni per alleggerire il codice, effettuando refactoring e applicando delle macro per poter gestire il tutto.

Per vedere le differenze di prestazioni tra una versione e un'altra abbiamo usato due computer con caratteristiche hardware diverse e per alcune funzioni abbiamo utilizzato la workstation **DGX-1** messa a disposizione dal dipartimento di *matematica e fisica di roma tre*. le differenze notate tra il computer meno performante e quello più performante sono state notevoli. Per quanto riguarda la parte precedente del codice, è presente una descrizione accurata dei vari dati acquisiti attraverso i nostri calcolatori e descritti nella relazione precedente, visitabile all'indirizzo qui di seguito:
https://github.com/MarcoCap13/LAR-SPLITTING-2D-5.b-/blob/main/relazioni/relazione02.md

Metodi di parallelizzione usati

Abbiamo continuato il nostro studio sulle *macro* per poter parallelizzare e migliorare la velocità computazionale delle varie funzioni. Nello specifico ci siamo soffermati questa volta sullo studio delle seguenti macro:

- **@views**: con views si possono creare delle viste degli array che ci permettono di accedere ai valori di quest'ultimo senza effettuare *nessuna copia*
- **@btime**: questa macro svolge lo stesso lavoro di _@benchmark_ ma restituisce un output meno complesso e più intuitivo, stampando a schermo le velocità di calcolo delle funzioni
- **@benchmark**: Ci permette di valutare i parametri della funzione in maniera separata; Richiama la funzione più volte per creare un campione dei tempi di esecuzioni restituendo i tempi minimi, massimi e medi.
- **@code_warntype**: ci consente di visualizzare i tipi dedotti dal compilatore, identificando così tutte le instabilità di tipo nel codice preso in esame.

Per quanto riguarda l'ottimizzazione e la parallelizzazione delle funzioni, sono state impiegate le seguenti macro:

- **@threads**: l'utilizzo di questa macro è fondamentale per indicare a *Julia* la presenza di **loop** che identificano *regioni multi-thread*.
- **@spawn**: identifica uno degli stumenti cardini di *Julia* per l'assegnazioni dei vari compiti per le task.

- **@async**: questa macro crea e pianifica le attività per tutto il codice all'inteno della sua attività. E' similare alla macro _@spawn_ con la differenza che runna le task solo a livello locale senza aspettare che il task termini.
- **@sync** il suon funzionamento è l'opposto del precedente; Aspetta che tutti i task convolti nella parallelizzazione siano completati prima di poter proseguire a livello computazionale.

Studio delle funzioni ottimizzate

Per vedere nel dettaglio i nuovi dati ed i benchmark riporto il link diretto:

- https://github.com/MarcoCap13/LAR-SPLITTING-2D-5.b-/tree/main/docs/benchmark

Lo studio preliminare del progetto è iniziato dalla comprensione del codice per capire come funzionasse lo *splitting 2D* per poi essere in grado di manipolare le strutture ad esso associate. Dopo di che si è passati allo studio delle funzioni più importanti come *spaceindex* e *pointInPolygonClassification* attraverso varie simulazioni delle stesse evidenziando così un'instabilità di tipo su alcune sue variabili grazie all'uso della macro _@code_warntype_ citata poco fa, oltre ad una velocita di esecuzione non proprio ottimale. Per risolvere questi problemi, si sono dovute studiare tutte le singole sotto-funzioni in particolare quelle che sollevavano l'**instabilità** sul tipo:

1) **spaceIndex**: attraverso lo strumento _@code_warntype_, è emersa un'instabilità in alcune variabili e non dell'intero metodo. Nel particolare sono *type unstable*: bboxes, xboxdict, yboxdict, zboxdict, xcovers, ycovers, zcovers ed infine covers. Parallelizzando il codice e creando un funzione di supporto denominata *removeIntersection* per poter alleggerire il codice stesso, abbiamo raggiunto i seguenti risultati con un notevole miglioramento.
   - Tipo: instabile
   - Velocità di calcolo:
     - iniziale: 116 μs
     - modificata (con workstation DGX-1): 74.8 μs
2) **boundingBox**: attraverso l'utilizzo della funzione denominata *@code_warntype_, è risultata un'instabilità in questo metodo. L'instabilità è dovuta unicamente alla funzione* mapslices*. Per ovviare a tale problematica abbiamo richiamato la funzione* hcat *che concatena due array lungo due dimensioni rendendo boundingbox* type stable *aumentando notevolmente le prestazioni. (per verificarlo abbiamo richiamato* @benchmark_ e comparato i risultati)
   - Tipo: instabile
   - Velocità di calcolo:
     - iniziale: 9.38 μs
     - modificata (con workstation DGX-1): 8.21 μs

Altre sotto-funzioni **type stable** invece sono state studiate per comprendere il funzionamento del codice e analizzare i tempi di esecuzione:

3) **boxcovering**: boxcovering è type stable ma la variabile covers è un array di Any. Si procede tipizzando covers e dividendo la funzione in microtask. Per parallelizzare questa funzione abbiamo utilizzato la funzione _@Thread_ e aggiunto due funzioni di supporto che illustreremo in seguito: *createIntervalTree* e *addIntersection*.

   - Tipo: stabile
   - Velocità di calcolo:
     - iniziale: 8.936 μs
     - modificata (con workstation DGX-1): 4.46 μs

4) **coordintervals**:Attraverso la macro *@code_warntype_ è stata individuata la stabilità di quest'ultima. La funzione è risultata molto semplice e qualsiasi intervento svolto, non ha portato a grossi miglioramenti. Abbiamo utilizzato la macro* @inbounds_ ma non ha portato a notevoli stravolgimenti.

   - Tipo: stabile
   - Velocità di calcolo:
     - iniziale: 958.143 ns
     - modificata: 1.029 μs

5) **fragmentlines**: Abbiamo 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. L'inserimento esplicito della macro simd non ha comportato alcun beneficio, infatti come si apprende dal sito ufficiale Julia:* "Note that in many cases, Julia can automatically vectorize code without the @simd macro". *Per quanto riguarda la macro* @inbounds_,invece, ha ridotto leggermente il numero di allocazioni in memoria. Nel complesso non sono stati rilevati miglioramenti riguardo le prestazioni della versione iniziale e modificata. Utilizzando la workstation *DGX-1* non abbiamo riscontrato migliorameti importanti.

   - Tipo: stabile
   - Velocità di calcolo:
     - iniziale: 196.813 μs
     - modificata: 197.939 μs

6) **linefragment**: Per quanto riguarda quest'ultima funzione, sono stati riscontrati notevoli miglioramenti a livello di prestazioni utilizzando la macro _@threads_

   - Tipo: stabile
   - Velocità di calcolo:
     - iniziale: 66.414 μs
     - modificata: 33.001 μs

7) **congruence**:funzione che prende in ingresso un modello di Lar, restituendo

una funzione di base denominata hcat che concatena due array lungo due dimensioni. Le macro utilizzate sono *@threads_ e* @inbounds_. Si nota un certo miglioramento se utilizziamo dei *filter* per i dati di EV

- Tipo: stabile
- Velocità di calcolo:
  - iniziale: 36.8 μs
  - modificata (workstation DGX-1): 19.6 μs

8) **pointInPolygonClassification**: funzione di notevole importanza nel nostro progetto. In questo caso abbiamo scomposto i vari else/if in tante *mono-task* per poter alleggerire il codice. Attraverso l'utilizzo della macro _@async_ abbiamo riscontrato un leggero miglioramento rispetto alla funzione iniziale.

- Tipo: stabile
  - Velocità di calcolo:
    - iniziale: 82.6 μs
    - modificata: 81.1 μs

Funzionamento dello splitting

Nella figura sottostante vedremo come lavora *pointInPolygon* e il funzionamento dello splitting, denotando tutti quei segmenti che intersecano le facce del poligono preso in esame nel piano z=0. Nello specifico nel punto (a) vediamo i singoli segmenti (o linee) che intersecano il poligono; Nel sezione (b) vengono illustrati tutti quei punti che sono situati esternamente, internamente o sul bordo della faccia del poligono e nel punto (c) vengono cancellati tutti quei segmenti che vanno verso l'esterno della faccia del poligono mentre per finire vediamo nel punto (d) il risultato finale dello *splitting*.

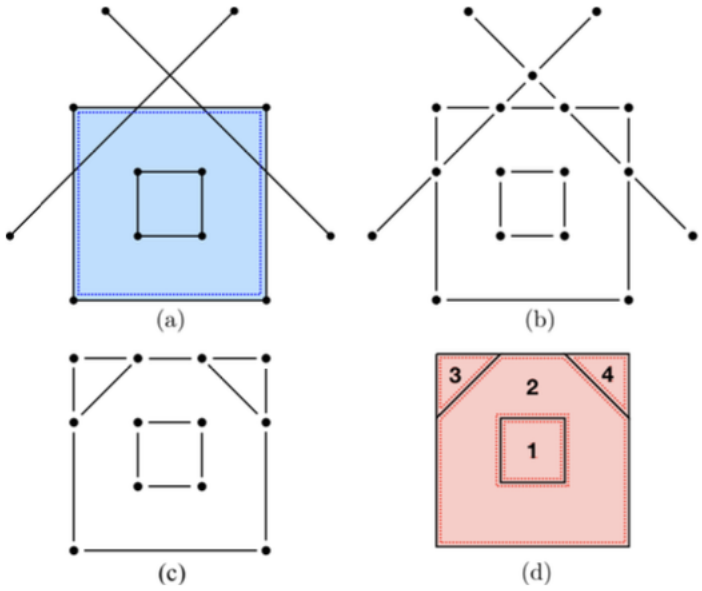
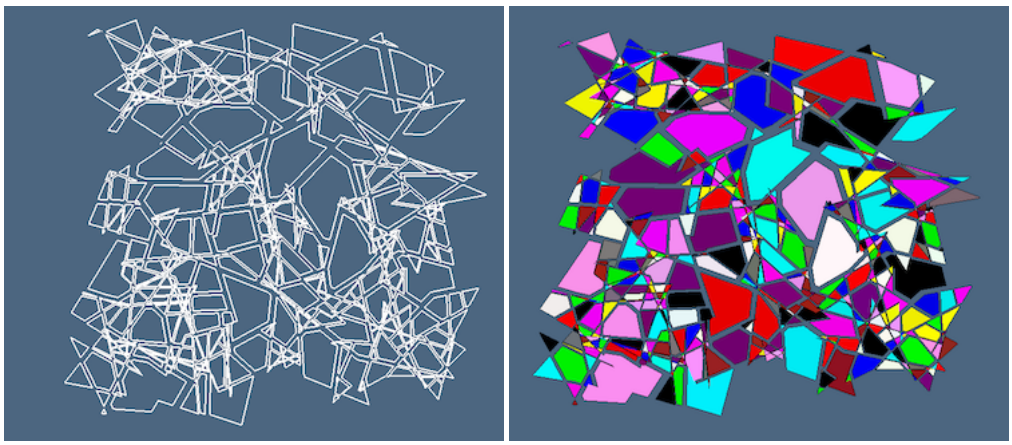


Figure 1: Lavoro di pointInPolygonClassification

Esempio dello splitting

Riportiamo un esempio di splitting effettuato durante lo studio definitivo del progetto attraverso due screenshot fondamentali:



Nella **prima figura** (di sinistra) vediamo le intersezioni del bounding-box i-esimo con i restanti boundingbox e nella **seconda figura** vediamo la generazione dei punti dell'intersezioni tra le varie parti.

Funzioni aggiuntive create

In questa sezione verranno illustrate tutte le funzioni secondarie da noi utilizzate create per migliorare, alleggerire e semplificare gran parte del codice.

- **addIntersection**(covers::Array{Array{Int64,1},1}, i::Int64, iterator) aggiunge gli elementi di iterator nell'i-esimo array di covers.
- **createIntervalTree**(boxdict::AbstractDict{Array{Float64,1},Array{Int64,1}}) dato un insieme ordinato, crea un intervalTree; Nel particolare parliamo di una struttura dati che contiene intervalli e che ci consente di cercare e trovare in maniera efficiente tutti gli intervalli che si sovrappongono ad un determinato intervallo o punto.
- **removeIntersection**(covers::Array{Array{Int64,1},1}): siamo riusciti a rendere più stabile il tutto diminuendo in linea generale i tempi di calcolo della funzione stessa. Quest'ultima elimina le intersezioni di ogni boundingbox con loro stessi.

Test delle funzioni principali e aggiuntive

inizialmente si sono eseguiti i test pre-esistenti per verificare il corretto funzionamento delle funzioni principali anche dopo aver effettuato lo studio di parallelizzazione con le macro dei singoli task. Dopo aver verificato il successo di questi, si è proceduto alla realizzazione di nuovi test:

1) **@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.
2) **@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.
3) **@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.

Per quanto rigurdano i test delle funzioni principali da noi studiate, abbiamo svolto con successo i test sulle funzioni iniziali ricevendo i risultati aspettati. Solo successivamente (con un po' di difficoltà) abbiamo svolto i test sulle funzioni da noi modificate arrivando alla completa correttezza di quest'ultimi. nello specifico si possono revisionare i vari test nei vari *notebook* aggiornati,

seguendo il link qui riportato:
https://github.com/MarcoCap13/LAR-SPLITTING-2D-5.b-/tree/main/notebook

Considerazioni finali sulla parallelizzazione

Durante lo studio preliminare ed esecutivo, abbiamo cercato di ottimizzare il nostro codice sia a livello di CPU che GPU. Considerato questo, abbiamo deciso di concentrarci maggiormente sulla parallelizzazione su CPU poichè uno dei pc utilizzati per il progetto era sprovvisto di una GPU dedicata. Per la parallelizzazione su CPU abbiamo utilizzato maggiormente le macro sopra elencate: *@threads_ e* @spawn_. La prima viene usata su un *ciclo for* per dividere lo spazio di iterazione su più thread secondo una certa politica di scheduling, mentre la seconda permette di eseguire una funzione su un thread libero nel momento dell'esecuzione. **@threads** viene usata nella funzione removeIntersection(), boxcovering() e infine in addIntersection(), mentre **@spawn** viene usata principalmente nella funzione spaceindex(). Per eseguire questo tipo di parallelizzazione bisogna tenere conto del numero di core presenti sulla macchina e proprio per questo motivo abbiamo provato a lavorare con la **workstation DGX-1** di *nvidia* per poter raggiungere prestazioni migliori. Anche con ciò, si è notato che utilizzare un numero di thread maggiore di quelli disponibili non porta ad un aumento delle prestazioni. Il numero di thread da assegnare ai vari processi julia va stabilito prima dell'avvio e può essere controllato e settato tramite la funzione *nthreads()*. Nel grafico sottostante abbiamo testato le prestazioni di *spaceindex()*, una delle funzioni più importanti per lo *splitting*. Nello specifico abbiamo visto cosa accadeva al variare del numero di thread, in particolare, quando si hanno a disposizione uno, quattro o otto thread (il massimo ottenibile dalla worksation DGX). Analizzando i tempi, si evince che il numero di thread deve essere scelto in base alla complessità del modello preso in esame. Infatti, utilizzando modelli semplici, un numero elevato di thread porta ad un peggioramento delle prestazioni, mentre all'aumentare della complessità si ha un miglioramento.

Grafo delle dipendenze aggiornato

In sintesi, questo **grafo** rappresenta il lavoro svolto sino ad ora con tutte le nuove funzioni create, aggiornate ed aggiunte. I nodi color celeste sono le funzioni di supporto, i nodi colorati di rosso sono le funzioni principali della classe e gli ultimi colorati di blu sono funzioni secondarie equamente importanti alla fine del progetto stesso. Nello specifico il nodo *Utility_function_PointInPolygon_1-15* racchiude tutte le 15 funzioni create per il supporto a PointInPolygonclassification.

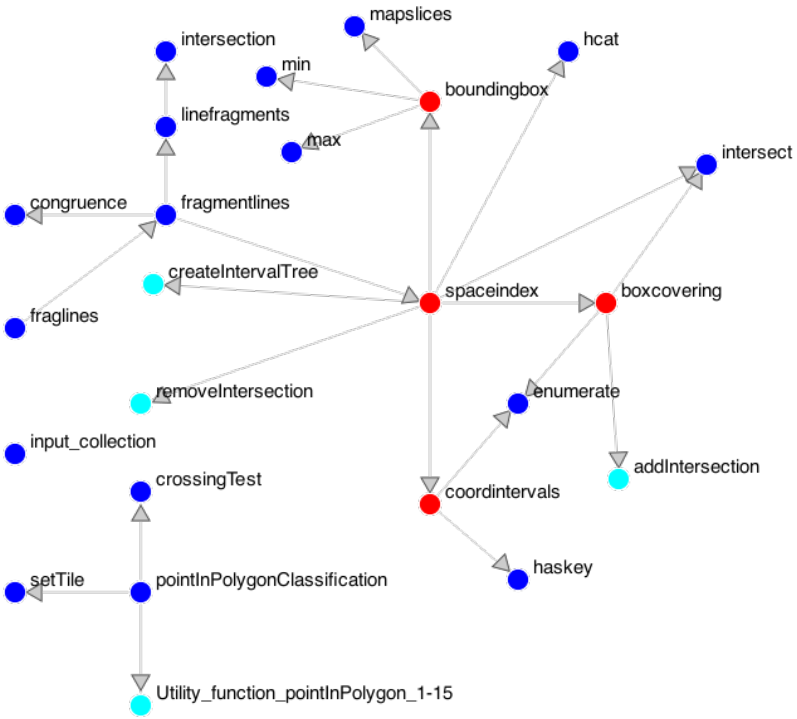


Figure 2: Grafo delle dipendenze della classe Refactoring (Aggiornato)