Relazione LAR Splitting 2D

Studenti: Caponi Marco, Ceneda Gianluca

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
  - https://github.com/cvdlab/Lar.jl/blob/master/src/fragface.jl

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

In questa sezione si illustreranno passo passo tutti i vari cambiamenti che sono stati fatti per poter ottimizzare, migliorare il codice e la sua velocità computazionale. Nello specifico abbiamo modificato le funzioni principali della classe **Refactoring** e della classe **Fragface**. Per quanto riguarda la parte precedente del codice, è presente una descrizione accurata nella relazione precedente, visitabile all'indirizzo di seguito: https://github.com/MarcoCap13/LAR-SPLITTING-2D-5.b-/blob/main/relazioni/relazione01.md

Attraverso lo studio preliminare dei metodi di parallelizzazione, siamo riusciti a migliorare alcune funzioni presenti nelle classi principali attraverso l'utilizzo di alcune macro studiate sul libro consigliato dal professore.

Per poter analizzare l'efficienza delle varie funzioni, abbiamo utilizzato le seguenti macro:

- **@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.

Studio delle funzioni ottimizzate

- **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. Affinando il codice (in altre parole cercando di eliminare i vari if/else che equivalgono ad una cattiva ottimizzazione del codice) e creando un funzione di supporto denominata.
  - Tipo: instabile
  - Velocità di calcolo:
    - iniziale: 108.611 μs
    - modificata: 70.212 μs
- **boundingBox**: sempre 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: 13.233 μs
    - modificata: 8.212 μs
- **boxcovering**: boxcovering è type stable ma la variabile covers è un array di Any. Si procede tipizzando covers e dividendo la funzione in microtask.
  - Tipo: stabile
  - Velocità di calcolo:
    - iniziale: 4.546 μs
    - modificata: 3.212 μs
- **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 di quest'ultima. Nella figura sottostante vedremo come lavora *pointInPolygon*, denotando tutti quei segmenti che intersecano le facce del poligono preso in esame. Nello

specifico nel punto (a) vediamo i singoli segmenti (o linee) che intersecano quest'ultime; Nel punto (b) vengono illustrati tutti quei punti che sono situati esternamente, internamente o sul bordo della faccia del poligono, nel punto (c) vengono cancellati tutti quei segmenti che vanno verso l'esterno della faccia del poligono e per finire vediamo nel punto (d) il risultato finale attraverso il **TGW** in 2D.

- Tipo: stabile
- Velocità di calcolo:
  - iniziale: 119.404 μs
  - modificata: 98.212 μs

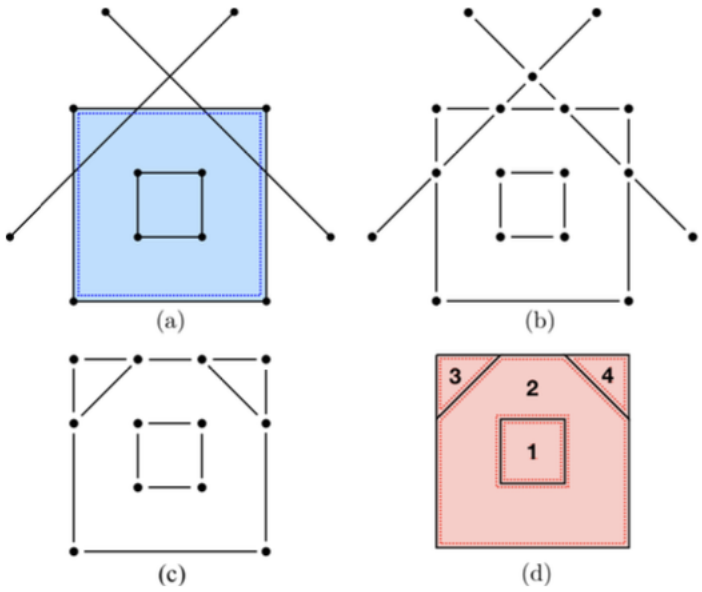

Figure 1: Lavoro di pointInPolygonClassification

Funzioni secondarie utilizzate dalle funzioni principali: pointInPolygon, spaceindex, boxcovering:

- hcat: concatena due array lungo due dimensioni
- min: restituisce il minimo degli argomenti.
- max: restituisce il massimo degli argomenti.
- intersect: restituisce l'intersezione di due insiemi.
- enumerate : un iteratore che produce (i, x) dove i è un contatore a partire da 1, e x è il valore i-esimo della collezione su cui scorre l'iteratore dato.

- haskey : determina se una collezione ha una mappatura per una determinata chiave.
- Mapslices: trasforma le dimensioni date dell'array in input usando una funzione scelta dall'utente. La funzione è chiamata su tutte le dimensioni (slices) dell'array.

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.

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.

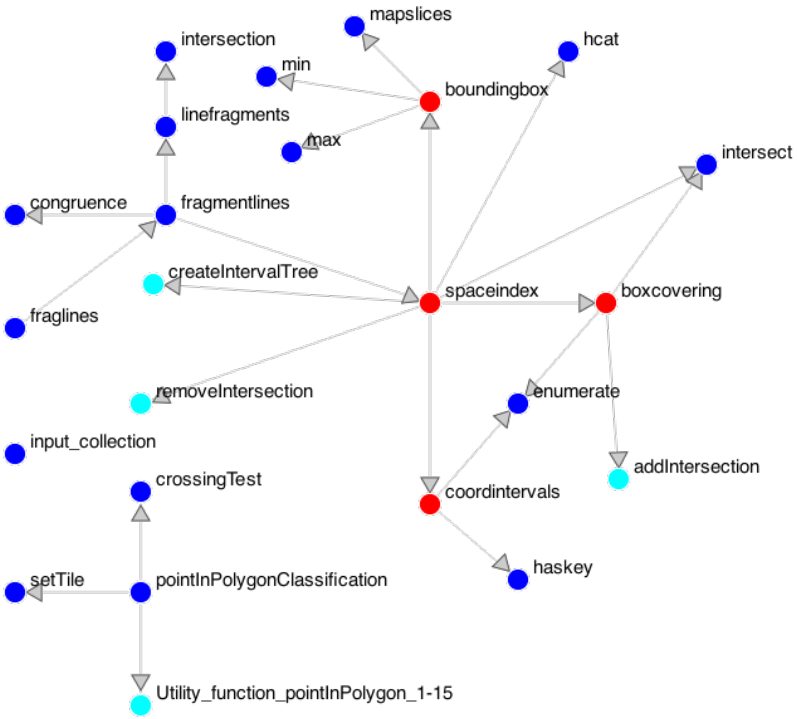

Figure 2: Grafo delle dipendenze della classe Refactoring (Aggiornato)