Studio Esecutivo Progetto LAR Splitting 2D – CPD22

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

20 Mag 2022

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

Contents

Analisi, ottimizzazione e parallelizzazione

In questa sezione si illustreranno i cambiamenti effettuati per poter ottimizzare e parallelizzare le funzioni. Per quanto riguarda l'analisi delle funzioni, è presente una descrizione accurata nella relazione precedente, visitabile al seguente indirizzo: https://github.com/GiuliaCastagnacci/LARSplitting2D/blob/main/relazioni/studioPreliminare.md

Premessa

Per analizzare le prestazioni delle funzioni, abbiamo utilizzato le seguenti macro:

1. *@btime*:
    a. Calcola e stampa il tempo trascorso durante l'esecuzione della funzione.

   b. Misura e stampa la quantità di memoria allocata durante l'esecuzione del codice.

2. *@code_warntype*:
   a. Consente di visualizzare i tipi dedotti dal compilatore, individuando così ogni tipo di instabilità nel nostro codice.
   b. L'output è un abstract syntax tree (AST), che ci avviserà di qualsiasi problema di inferenza di tipo nel codice, evidenziandolo in rosso. Altrimenti, se la funzione è di tipo stabile verrà evidenziato l'output in blu.
3. *@benchmark*:
   a. Valuta i parametri della funzione separatamente, quindi chiama la funzione più volte per creare un campione dei tempi di esecuzione.
   b. L'output mostrerà il tempo medio impiegato per eseguire il codice.

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

1. *@threads:*
   a. Questa macro è apposta davanti a un *for* per indicare a Julia che il loop è una regione multi-thread.
   b. Lo spazio di iterazione viene suddiviso tra i thread, dopodiché ogni thread scrive il proprio ID thread nelle posizioni assegnate.
2. *@spawn*: è uno degli strumenti che Julia mette a disposizione per assegnare compiti ai lavoratori.

Funzioni analizzate

pointInPolygonClassification

- Versione iniziale della funzione
  - Tempo di esecuzione ottenuto: 163.289 ns
  - Tipo: stabile
  - Benchmark: Figura 1
- Versione modificata della funzione:
  - Tempo di esecuzione ottenuto dopo le modifiche: 146.026 ns
  - Benchmark: Figura 2

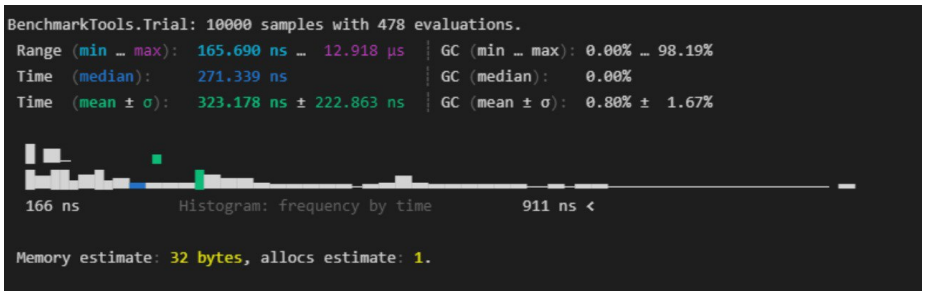

Figure 1: Benchmark pointInPolygonClassification

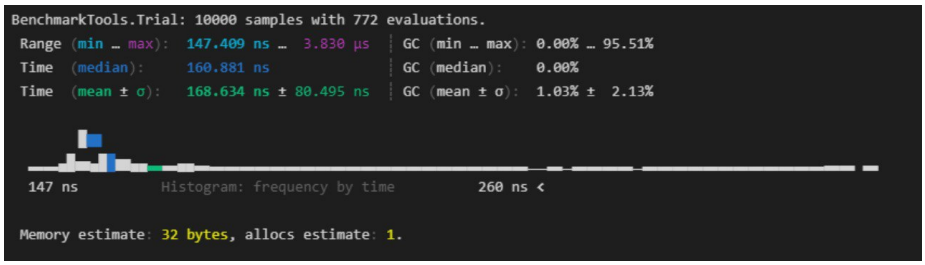

Figure 2: Benchmark pointInPolygonClassification modificato

Nella versione modificata abbiamo creato una nuova funzione per ogni edge, sostituendo i relativi if else a catena di pointInPolygonClassification con le rispettive chiamate alle nuove funzioni.

Lo splitting avviene mediante la riduzione della faccia di interesse e tutte quelle di possibile intersezione con lei. La faccia viene ridotta ai suoi spigoli di bordo, così come le altre incidenti vengono ridotte agli spigoli di bordo; 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.

spaceindex

- Versione iniziale della funzione
  - Tempo di esecuzione ottenuto: 132.000 μs
  - Tipo: Del code_warntype di spaceindex emerge la instabilità di tipo di alcune variabili e non del metodo. in particolare, sono type unstable bboxes, xboxdict, yboxdict, zboxdict, xcovers, ycovers, zcovers e covers.
  - Benchmark: Figura 3
- Versione modificata della funzione:
  - Tempo di esecuzione ottenuto dopo le modifiche: 60.400 μs
  - Benchmark: Figura 4

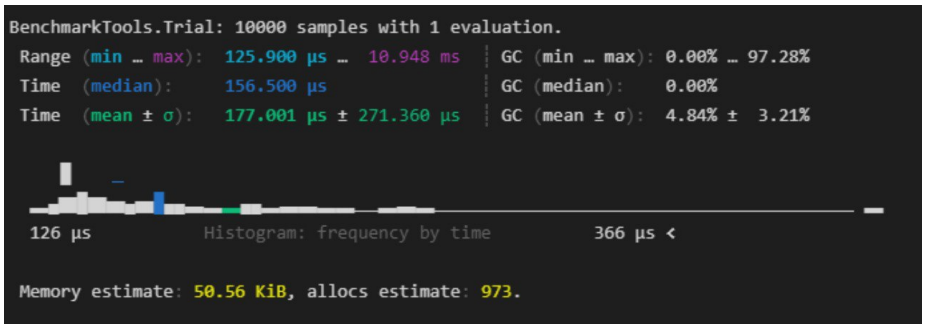

Figure 3: Benchmark spaceindex

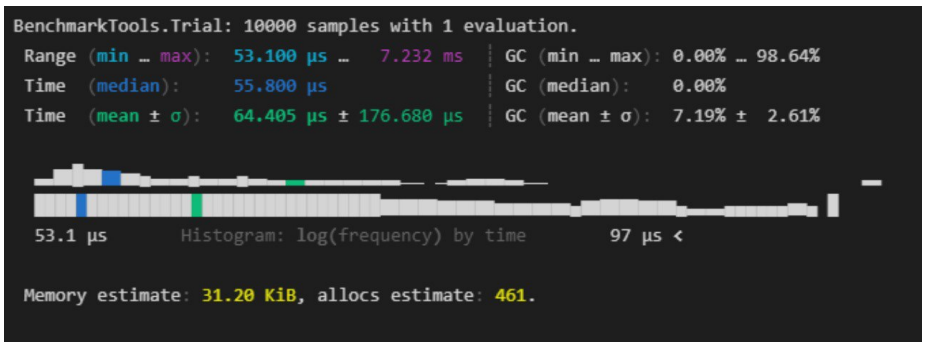

Figure 4: Benchmark spaceindex modificato

boundingbox

- Versione iniziale della funzione
  - Tempo di esecuzione ottenuto: 17.200 μs
  - Tipo: instabile
  - Benchmark:

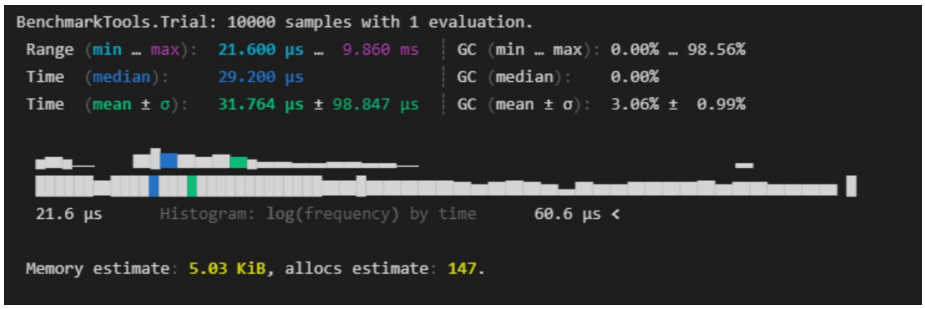

Figure 5: Benchmark boundingbox

- Versione modificata della funzione:
  - Tempo di esecuzione ottenuto dopo le modifiche: 6.920 μs
  - Benchmark:

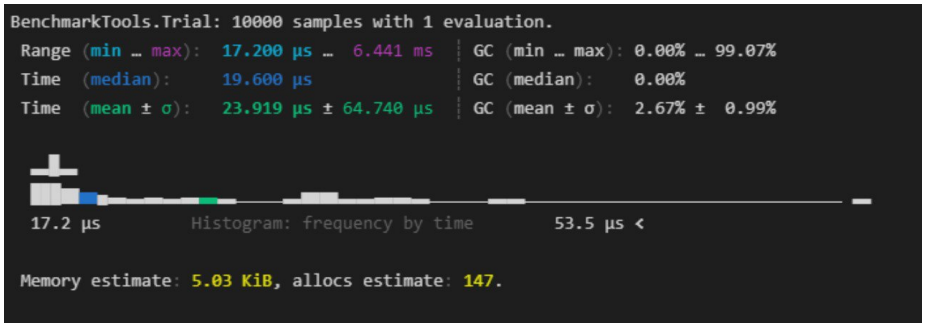

Figure 6: Benchmark boundingbox modificato

boxcovering

- Versione iniziale della funzione
  - Tempo di esecuzione ottenuto: 5.750 μs
  - Tipo: stabile

– Benchmark:

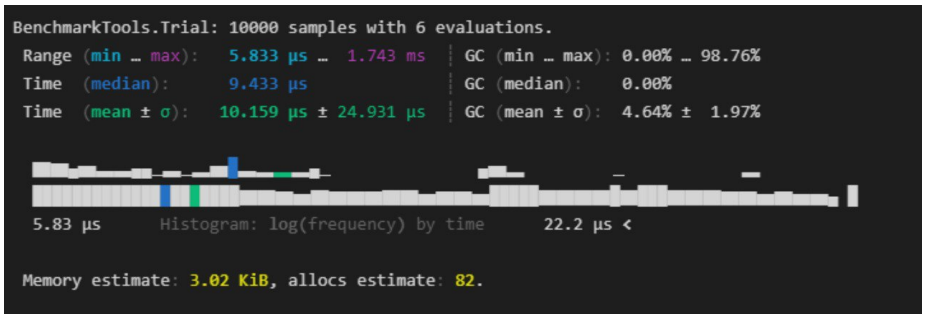

Figure 7: Benchmark boxcovering

- Versione modificata della funzione:
  - Tempo di esecuzione ottenuto dopo le modifiche: 8.767 μs
  - Benchmark:

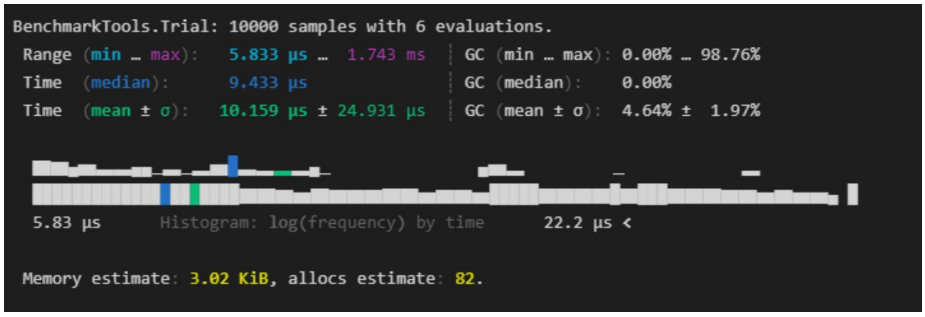

Figure 8: Benchmark boxcovering modificato

Funzioni aggiunte

- addIntersection(covers::Array{Array{Int64,1},1}, i::Int, iterator) aggiunge gli elementi di iterator nell'i-esimo array di covers. Utilizzata in: boxcovering
- createIntervalTree(boxdict::AbstractDict{Array{Float64,1},Array{Int64,1}}) 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. Utilizzata in: spaceindex, boxcovering

- removeIntersection(covers::Array{Array{Int64,1},1}) elimina le intersezioni di ogni bounding box con loro stessi. Utilizzata in: spaceindex

Sulle funzione aggiuntive sono stati eseguiti i test per verificare il corretto funzionamento.

Funzioni di libreria utilizzate dalle funzioni principali

- **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.
- **size**: restituisce una tupla contenente le dimensioni dell'input. E' possibile specificare una dimensione per ottenere solo la lunghezza di tale dimensione.
- **IntervalMap{K, V}**: alias di tipo per IntervalTree{K, IntervalValue{K, V}} per semplificare l'associazione di dati di tipo V con intervalli.
- **intersect**: restituisce l'intersezione di due insiemi.
- **append!(c, c2)**: per un contenitore ordinato c, aggiunge gli elementi di c2 a c.
- **hcat**: concatena due array lungo due dimensioni.
- **minimum**: restituisce il risultato più piccolo di una funzione che viene chiamata su ogni elemento dell'array passato come parametro.
- **maximum**: restituisce il risultato più grande di una funzione che viene chiamata su ogni elemento dell'array passato come parametro.
- **setdiff**: date due collezioni, restituisce gli elementi che sono presenti nel primo insieme ma non nel secondo, se un elemento è presente in entrambi gli insiemi viene rimosso.
- **zip**: dati due array della stessa lunghezza, crea coppie di elementi.
- **length**: ritorna il numero di elementi dell'array in ingresso.
- **tuple**: costruisce una tupla con i parametri dati in input.
- **haskey**: determina se una collezione ha una mappatura per una determinata chiave.
- **push!**: inserisce uno o più item nella collezione. Se la collezione è ordinata, gli item vengono inseriti alla fine (nell'ordine specificato).
- **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.
- **min**: restituisce il minimo degli argomenti.
- **max**: restituisce il massimo degli argomenti.

Grafo delle dipendenze refactoring

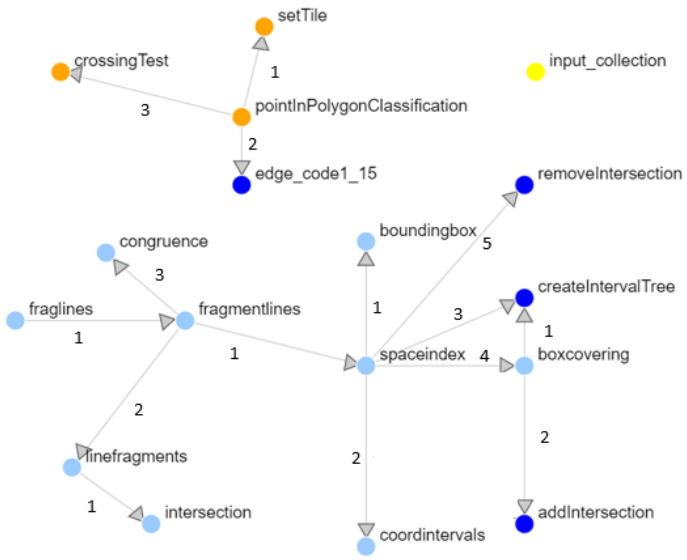

Figure 9: Grafo delle dipendenze refactoring