Calcolo Parallelo e Distribuito - Relazione progetto “COLLECTION”

URL Repository Git : *https://github.com/alegio98/Collection*

Fase Preliminare , descrizione introduttiva dello svolgimento del progetto :

dopo aver svolto il paragrafo “0.1 Primi Passi” , ho disegnato a mano il grafo delle dipendenze e poi riportato in bella copia tramite il tool diagrams.net , per due volte cercando di spiegare al meglio in ogni sezione (sotto la prima foto) il tipo e significato di ogni parametro e valore di ritorno.
All’interno della repository in src -> progetto.jl vi sarà il codice del progetto con le nuove parti commentate sopra ogni funzione che descrivono i task , le funzioni sono state modificate e migliorate (per quanto possibile) , facendo di esse un’analisi dettagliata soprattutto nei notebook Julia .

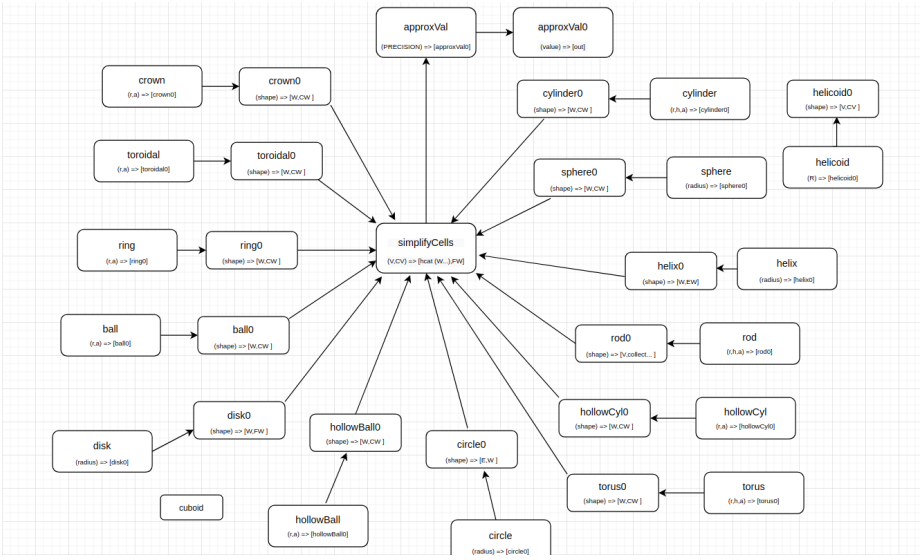


Figure 1: initial dependencies graph

- **approxVal**: PRECISION = numero di cifre a cui approssimare il valore “value” parametro della funzione annidata approxVal0 , il valore di ritorno è proprio approxVal0 che rappresenta il valore approssimato ad una determinata PRECISION (=digits)
- **simplifyCells** : V, CV = vengono presi in input questi 2 array che all’interno della funzione vengono elaborati(semplificati) e successivamente restituiti sotto la nomenclatura W, FW

- **circle** : presi come parametri (radius) raggio e (angle) angolo poi elaborati (in generale anche attraverso simplifyCells) restituito “circle0” che è la funzione annidata di circle dove vengono generati i due vettori contenenti i cerchi .
- **helix** : presi come parametri (radius) raggio , (pitch) inclinazione e numero di giri (nturns) poi elaborati e restituito “helix0” funzione annidata di helix che lo genera .
- **disk** : presi come parametri (radius) raggio e (angle) angolo poi elaborati e restituito “disk0” funzione annidata di disk che lo genera .
- **helicoid** : presi come parametri (radiusB) raggio grande , (radiusS) raggio piccolo , (pitch) inclinazione e numero di giri (nturns) poi elaborati e restituito “helicoid0” funzione annidata di helicoid che lo genera .
- **ring** : presi come parametri (radiusS) raggio piccolo , (radiusB) raggio grande e (angle) angolo poi elaborati e restituito “ring0” funzione annidata di ring che lo genera .
- **cylinder** : presi come parametri (radius) raggio , (height) altezza e (angle) angolo poi elaborati e restituito “cylinder0” funzione annidata di cylinder che lo genera .
- **sphere** : presi come parametri (radius) raggio , (angle1) primo angolo , (angle2) secondo angolo e (surface) superficie poi elaborati e restituito “sphere0” funzione annidata di sphere che lo genera .
- **toroidal** : presi come parametri (radiusS) raggio piccolo, (radiusB) raggio grande , (angle1) primo angolo , (angle2) secondo angolo poi elaborati e restituito “toroidal0” funzione annidata di toroidal che lo genera .
- **crown** : presi come parametri (radiusS) raggio piccolo , (radiusB) raggio grande , (angle) angolo poi elaborati e restituito “crown0” funzione annidata di crown che lo genera .
- **cuboid** : presi come parametri (array1) un array dove è presente il punto massimo (maxpoint) del parallelepipedo, (flag) un valore booleano, (array2) un secondo array dove è presente il punto minimo (minpoint) del parallelepipedo , restituisce un cubo di dimensione “d”, dove “d” è la lunghezza comune degli array “minpoint” e “maxpoint”.
- **ball** : presi come parametri (radius) raggio , (angle1) primo angolo , (angle2) secondo angolo poi elaborati e restituito “ball0” funzione annidata di ball che lo genera .
- **rod** : presi come parametri (radius) raggio , (height) altezza e (angle) angolo poi elaborati e restituito “rod0” funzione annidata di rod che lo genera .
- **hollowCyl** : presi come parametri (radiusS) raggio piccolo , (radiusB) raggio grande , (height) altezza , (angle) angolo poi elaborati e restituito

"hollowCyl0" funzione annidata di hollowCyl che lo genera .

- **hollowBall** : presi come parametri (radiusS) raggio piccolo , (radiusB) raggio grande , (angle1) primo angolo , (angle2) secondo angolo poi elaborati e restituito "hollowBall0" funzione annidata di torus che lo genera .
- **torus** : presi come parametri (radiusS) raggio piccolo , (radiusB) raggio grande , (height) altezza, (angle1) primo angolo , (angle2) secondo angolo poi elaborati e restituito "torus0" funzione annidata di torus che lo genera

La maggior parte delle funzioni crea oggetti , come cerchi , cubi , elicodali ecc , le funzioni sono già ben ripartite in sottofunzioni , ho cercato di fare degli esempi pratici descritti nei notebook , ovviamente lo stesso metodo di suddivisione può essere applicato a tutte quante le funzioni che creano oggetti . SimplifyCells è la funzione principale (fondamentale) a cui è stato fatto refactoring ed è stata testata , tutto ovviamente è possibile trovarlo all'intero dei notebook .

Progetto Definitivo :

Il corso di Calcolo Parallelo e Distribuito è stato interamente incentrato sulla libreria Linear Algebraic Representation (LAR) sviluppata dal professor Alberto Paoluzzi. LAR, si pone l'obiettivo di ottenere una caratterizzazione minima della geometria e della tipologia di un complesso cellulare ricostruendo la catena principale (anche detta pipeline 3D) di generazione dei vertici, spigoli, facce e celle di un qualsiasi tipo solido , nel mio caso in particolare ho potuto vedere lo svilupparsi di 15 diverse tipologie . La rappresentazione scelta per descrivere la matematica in LAR è un insieme di matrici sparse di tipo intero; esse sono la principale struttura dati utilizzata in libreria.
Le matrici sparse vengono molto spesso sfruttate nel calcolo scientifico soprattutto per la loro proprietà di memorizzare solo gli elementi non nulli, presenti nella matrice, in modo da risparmiare sia in termini di tempo (nella computazione) e sia in termini di spazio (nella memorizzazione). Durante il corso si è posta l'attenzione sulla valutazione e sul miglioramento delle performance della libreria LAR, sviluppata nel linguaggio Julia.

Ottimizzazione del codice :

Il codice del progetto Collection è incentrato sullo sviluppo di figure solide (e non solo) come , cubi , quadrati , sfere , elicodali ecc ..
L'obiettivo principale è stato quello di cercare di ottimizzare il codice rendendolo più stabile e meno ambiguo .Il progetto è suddiviso in molte parti(funzioni) ognuna di esse svolge un unico compito .. citando un esempio iniziale nel codice : la funzione *circle* svolge gia' solo un compito ovvero "crea un cerchio". Se avesse creato un cerchio centrato su un punto P avrebbe invece svolto due compiti logicamente ben separati: "creare un cerchio" e "centrarlo(traslarlo) sul punto P". Quindi in questo caso era utile e non deleterio scrivere funzioni mono-task rendendo il codice più leggibile e mantenibile .
Le due funzioni iniziali all'interno del codice tornano molto utili per non sovraccaricare il calcolo e la computazione nell'esecuzione del codice poichè , la prima –> approxVal() : approssima i valori passati in input spesso di tipo float , mentre la seconda –> SimplifyCells() : trova e rimuove i vertici duplicati e le celle errate. Entrambe aumentano di molto le performance delle successive funzioni , quest'ultima è la funzione richiamata da tutte le sue successive , lo studio è stato incentrato fondamentalmente su di essa poichè migliorandone le sue prestazioni di conseguenza si migliorano le prestazioni di tutte le altre funzioni , nel notebook che la riguarda ci sono tutte le spiegazioni dettagliate sul suo refactoring e sul miglioramento delle prestazioni in termini di tempo grazie alle macro di Julia , approfondirò il discorso relativo alle macro a breve.

Refactoring notebook Julia :

Inizialmente il codice non girava per via dei problemi legati ai package Triangle e Polyhedral , quindi il primo obiettivo è stato quello di sistemare le librerie presenti in LAR nel migliore dei modi . Il package Triangle presenta diversi problemi e non è più aggiornato quindi ho eliminato le dipendenze da ogni file (linearalgebraicrepresentation.jl) , così facendo le funzioni di Triangle non vengono richiamate . Mentre Polyhedral andava in conflitto con il package ViewerGL (utile per la visualizzazione del progetto) quindi anche in questo caso Polyhedral è stato rimosso . Una volta essere sicuro che il codice non aveva più problemi legati ai package sono passato al suo refactoring, ho indentanto nel modo più chiaro possibile il codice per facilitarne la leggibilità e ho sistemato dei piccoli errori di sintassi (esempio : function non chiuse o ; mancanti). Successivamente ho inizializzato un notebook Julia relativo ad una function –> esempio : "circle" , dato che , tutte le funzioni che hanno il compito di creare oggetti sono strutturate nello stesso modo .
La maggior parte delle implementazioni sono state oggetto di benchmark , ho stimato comportamenti con le macro @code .. , e per valutare i tempi di esecuzione è stato fatto uso delle macro @btime , @time , presenti nel package BenchmarkTools.jl , usando in particolare @code_warntype , essa definisce una rappresentazione del codice che può essere utile per trovare espressioni che generano incertezza sul tipo. Infatti richiamando la macro nella function "circle" si può notare che il tipo di valore restituito è evidenziato con il colore rosso ciò sta a significare che alcuni suoi calcoli intermedi sono instabili di tipo .

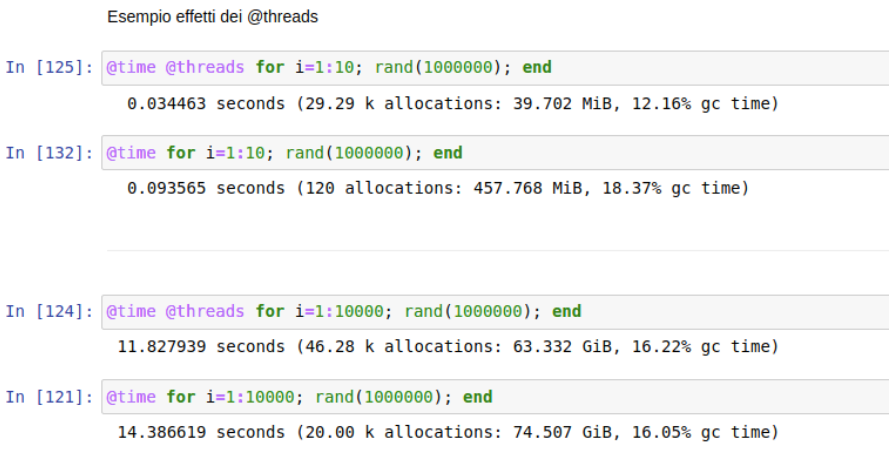
L'uso delle macro :

Ho usato il pacchetto BenchmarkTools che semplifica il monitoraggio delle prestazioni del codice Julia fornendo un framework per la scrittura e l'esecuzione di gruppi di benchmark e il confronto dei risultati dei benchmark , è la base della stima delle prestazioni di funzioni . Elencherò e spiegherò in alcuni casi anche con esempi tutte le macro utilizzate all'interno del progetto .

- **@btime**: Utile per stampare il tempo necessario per l'esecuzione, il numero di allocazioni e il numero totale di byte dovuti all'allocazione, prima di restituire il valore dell'espressione.
- **@benchmark**: Il concetto è lo stesso del precedente ma in questo caso viene anche generato una sorta di piccolo grafico che mostra tutti i dati in % delle prestazioni .
- **@code_warntype**: Serve a diagnosticare problemi relativi al tipo , con

essa riusciamo a visualizzare tutti i tipi della funzione (variabili , return ecc)

- **@inbounds**: Come molti linguaggi di programmazione moderni, Julia utilizza il controllo dei limiti per garantire la sicurezza del programma durante l'accesso agli array. In stretti cicli interni o in altre situazioni critiche per le prestazioni, si potrebbe voler saltare questi controlli dei limiti per migliorare le prestazioni di runtime. Quindi inbounds dice al compilatore di saltare controlli dei limiti degli array all'interno del blocco dato.
- **@threads**: Macro utilizzata per velocizzare i tempi di esecuzione , scopo: parallelizzare un ciclo for da eseguire con più thread. Divide lo spazio di iterazione tra più attività ed esegue tali attività sui thread in base a un criterio di pianificazione. Alla fine del ciclo viene posizionata una barriera che attende il completamento dell'esecuzione di tutte le attività. Per fare in modo che i threads funzionino però bisogna eseguire dei passaggi fondamentali che variano in base alla macchina su cui gira il codice... Ho settato il numero di threads pari a 2 e per forza ho dovuto eseguire (come scritto nella guida di julia) i seguenti passaggi : export JULIA_NUM_THREADS=2 , set JULIA_NUM_THREADS=2 , il numero di threads va settato in base alla macchina detenuta , se non lo si fa l'uso dei threads è nullo . Con l'applicazione di essi , ci sono stati dei leggeri miglioramenti visualizzabili all'interno di ogni notebook con le relative stime dei tempi .



Il risultato ottenuto evidenzia un miglioramento in termini di prestazioni ,(applicazioni simili nei notebook con le funzioni del progetto!)

Test e risultati :

Una delle parti più importanti del progetto riguarda i test . Sono stati aggiunti nel direttorio "test" piccoli e mirarti test basati su una funzione iniziale chiamata BoxCalculation() , essa prende una geometria in ingresso se essa é 2D calcola l'area del rettangolo che la circoscrive, mentre se é 3D ne calcola il volume del parallelepipedo che la circoscrive. Risulta essere molto utile per verificare size , length e disposizioni di vertici in tutte le tipologie di forme . Nel direttorio progetto.jl/examples/ è stata inserita una classe relativa a test piu grandi dove vi sono 4 funzioni più complesse che testano il corretto svilupparsi dei solidi (caso particolare : cubo) sia in forma 2D(quadrato) che 3D , e una che testa esclusivamete la classe simplifyCells , i dati utilizzati non sono in quantità elevate poiche non ho avuto intenzione di testarne la sua velocità (fatto nei notebook) , ma la sua funzionalità , tutto funziona. Per generare l'input è stata usata la funzione Lar.cuboidGrid che genera una griglia di cubi proporzionata all'input immesso.
Nonostante i vari cambiamenti e modifiche del codice , implementazione e rimozione di diversi package , *tutti i test risultano validi* .
L'obiettivo principale è stato proprio quello di ottimizzare il codice favorendone la sua leggibilità , sottoponendolo a nuovi test .

`Alessandro Giovannini (matricola : 520310)`