POLITECNICO DI TORINO

Corso di Laurea
in Matematica per l'Ingegneria

Progetto di

Programmazione e calcolo scientifico

Partecipanti	**Matricole**
Emanuele Martin	296214
Andrea Terenziani	284817
Federico Pessina	293945

Anno Accademico 2023-2024

Introduzione

Il progetto assegnato fornisce in input un DFN (Discrete Fracture Network, ovvero un sistema di poligoni planari, detti fratture, intersecati tra di loro) e richiede due consegne differenti:

innanzitutto di individuare e catalogare le intersezioni tra i poligoni, dette tracce, in passanti e interne, e secondariamente di tagliare i poligoni seguendo le tracce stesse e utilizzando come gerarchia di taglio prima le tracce passanti (ordinate per lunghezza decrescente) e successivamente quelle interne (sempre in ordine di lunghezza decrescente).

Per progettare e sviluppare il codice risolutivo, ci siamo soffermati su 3 fasi principali, che hanno reso possibile in primis il raggiungimento degli output richiesti dalla consegna, e in secondo luogo hanno reso il programma più facilmente leggibile e computazionalmente efficace (e di conseguenza dunque, ottimale in ottica di tempistiche risolutive):

-organizzazione del codice: ovvero la suddivisione su più file e varie classi, per rendere il lavoro più leggibile, ordinato e lineare da programmare.

-scelte logiche e procedimento: ovvero la fase di decisione su come approcciarsi al problema stesso e quali meccanismi risolutivi fossero necessari in ciascuna funzione utilizzata.

-ottimizzazione algoritmo e strutture dati: ovvero una fase più tecnica, in cui vengono presi numerosi accorgimenti per rendere il codice performante, rapido e computazionalmente vantaggioso.

Indice

Capitolo 1

Il codice

1.1 Struttura del codice

Il programma utilizza 3 classi principali:

-Fracture: E' la classe di una singola frattura, contenente tutte le sue caratteristiche e i rispettivi metodi. Oltre alle ovvie informazioni quali id, matrice dei vertici ecc... in essa sono presenti: internal_traces e passant_traces : contenente gli id delle tracce di tale frattura (rispettivamenete interne nel primo caso e passanti nel secondo);
partition_id : ovvero la partizione dello spazio a cui appartiene la frattura.
Tale informazione risulta fondamentale per ridurre i tempi di calcolo (e verrà spiegata più nel dettaglio nella terza sezione della relazione).

-TracesMesh: E' l'elenco di tutte le tracce Le informazioni contenute sono ad esempio gli Id della traccia, la lunghezza, le fratture che l'hanno generata, le coordinate dei vertici estremi, ecc...

-PolygonalMesh: E' la mesh di ogni singola frattura.
Risulta fondamentale per per il secondo punto del progetto, in quanto in essa vengono salvati tutti i nuovi poligoni generati dai tagli. Come di consuetudine, è stata divisa in celle 0Ds ovvero i vertici, 1Ds ovvero i lati e 2Ds ovvero i poligoni. Il codice (e di conseguenza quindi anche le funzioni) è invece stato diviso su più file:

-main.cpp: Il suo scopo principale è quello di chiamare le funzioni (attivando così una cascata di chiamate ricorsive) e di calcolare i tempi impiegati dal codice (fondamentale per confermare che le scelte di ottimizzazione prese, fossero effettivamente vantaggiose).

-Algorithms.cpp: In esso sono contenute tutte le funzioni che operano sulle fratture e sulle tracce, in grado di fornire i risultati concreti ricercati e necessari per l'avanzamento dell'algoritmo, a partire dalle strutture dati già munite degli input necessari.

-Utils.cpp: contiene le funzioni necessarie per il funzionamento del programma, più slegate tutta via dal procedimento logico puramente inerente allo studio delle fratture e delle tracce. Ne sono un esempio le funzioni di stampa, di lettura e salvataggio dei dati, di comparazione...

-Test.hpp: in questo file viene testata la correttezza di tutte le principali funzioni utilizzate. E' di fondamentale importanza in quanto permette di concentrarsi esclusivamente sulla progettazione logica del problema, escludendo i casi in cui il procedimento è giusto ma l'output ottenuto è sbagliato.

1.2 Scelte logiche e procedimento

Consegna numero 1:

Per quanto riguarda la risoluzione della prima richiesta del progetto, la porzione di codice che ha fornito il maggiore contributo, richiedendo così numerose scelte sia a livello procedurale che implementativo, è collocata all'interno di Fracture.cpp e più precisamente è il metodo *GenerateTrace*.
Tale metodo infatti viene applicato su una frattura, e consiste nel confrontarla con un'altra frattura appartenente alla sua stessa partizione e con il baricentro sufficientemente vicino, calcolare la traccia generata da esse e infine distinguerne la natura (passante o interna).
Per fare ciò sono necessari più passaggi e la divisione su più casistiche: Cercare l'equazioni dei piani su cui giacciono le fratture, e calcolarne l'intersezione che sarà dunque una retta (a meno di fratture parallele).

Cercare l'intersezione tra tale retta e le due fratture, e trovare quanti di questi punti siano DISTINTI. In base al numero di punti ottenuti, siamo in grado di catalogare il tipo di traccia:

2 : i punti di intersezione coincidono a coppie e la traccia è passante per entrambe le fratture.

3 : un punto di intersezione è in comune per entrambe le fratture. Trovato quel punto, il segmento di lunghezza minore sarà la traccia, e risulterà passante per la frattura che ha entrambi i vertici della traccia sul bordo, e interna per l'altra.

4 : Innanzitutto cerco il punto estremo, e i due punti più vicino ad esso, e poi catalogo per tutti i casi che possono capitare:
il segmento di intersezione è esterno ad entrambi i poligoni: non c'è traccia se i due punti vicini (e non l'estremo) appartengono alla stessa frattura, la traccia sarà il segmento di unione tra questi due punti, e sarà passante per quest'ultima e interna per l'altra.
Se i due punti vicini (e non l'estremo) NON appartengono alla stessa frattura, la traccia sarà il segmento di unione tra questi due punti, e sarà interna per entrambe le fratture.

Consegna numero 2:
Per quanto riguarda la risoluzione della seconda richiesta del progetto invece, il metodo che contiene il procedimento risolutivo, è sempre contenuto in Fracture.cpp e si chiama *CutPolygonalMesh*.
Tale metodo permette di, data una frattura, tagliarla progressivamente in più fratture seguendo le tracce (rigorosamente effettuando i tagli prima lungo le tracce passanti e poi quelle interne, ordinate entrambe per lunghezza decrescente) e aggiornare la mesh con i dati delle fratture ottenute.

Per fare ciò, a partire da una mesh contenente delle fratture (attive e non), si riconduce ricorsivamente a casistiche sempre più facili da studiare attraverso alle chiamate del metodo *CutMeshBySegment* e della funzione *Algorithms::CutPolygonBySegment*.

Analizzandola più nel dettaglio, le casistiche da gestire e gli accorgimenti da adottare per potersi ricondurre al caso elementare di un singolo poligono tagliato da una traccia passante, sono raggruppabili in tale modo:

1. Mesh costituita da una sola frattura attiva (primo step) o da più fratture, alcune attive e altre disattivate (step successivi)

2. Operazione di taglio lanciata su una traccia passante (caso banale) o traccia interna (caso più complicato).
   Nel caso di una traccia interna, sarà necessario ricondursi ad una traccia passante:

   - Viene prolungata la traccia.
   - Si cerca l'intersezione tra la retta su cui giace la traccia e ogni lato del poligono.
   - Si verifica quali coefficienti trovati dalla soluzione di tale sistema siano compresi tra 0 e 1 (ciò garantisce che la combinazione sia convessa e l'intersezione avvenga all'interno del lato e non sul suo prolungamento).
   - Si verifica che venga tagliato solo uno dei poligoni attivi.

Infine, una volta ricondotti al caso elementare, *CutPolygonBySegments* effettua il taglio e la creazione di due nuove fratture attive, disattivando la frattura padre precedente, e garantisce un'uniformità nell'ordinamento dei vertici (vecchi e nuovi) di ciascuna frattura generata.

Capitolo 2

Scelte implementative

2.1 Scelte computazionali per le principali funzioni

Per quanto riguarda l'efficienza in termini di spazio l'utilizzo delle referenze per passare gli argomenti delle funzioni è fondamentale per evitare di generare, inutilmente, delle copie ad ogni chiamata di funzione.
Invece i *const* prima di passare gli argomenti sono stati utilizzati quando il dato passato non doveva venire modificato in quella funzione.
E' stato necessario ordinare le fratture in ordine decrescente per lunghezza, e per tale scopo abbiamo incluso gli algoritmi di ordinamento *Mergesort* e *Bubblesort*.
Nonostante il costo computazionale del *Bubblesort* sia $\mathcal{O}(n^2)$ che risulta essere maggiore del costo per implementare il *Mergesort* che invece è $\mathcal{O}(n \log n)$, per i dataset forniti impiega meno tempo. Questo a causa del basso costo fisso del *Bubblesort*, e di come funzionano i due algoritmi: il *Mergesort* (a differenza del *Bubblesort*) è basato sulle chiamate di funzioni ricorsive, ciascuna delle quali ha un impatto sulla velocità dell' algoritmo.

2.2 Strutture dati

I due principali tipi di contenitori impegiati sono stati i vettori della libreria standard, utili quando le loro dimensioni non erano note a priori, ed i vettori/matrici della libreria *Eigen*, che hanno il vantaggio di avere implementati nativamente i metodi per svolgere le operazioni tra vettori e matrici, come ad esempio prodotti vettoriali e risoluzioni di problemi lineari.
Abbiamo utilizzato in alcune funzioni la struttura map (ad esempio per associare ad una frattura tutte le tracce coinvolte).

2.3 Scelta della tolleranza e altri dettagli

Poichè è stato necessario effettuare diversi calcoli durante il progetto, spesso ci si è posto davanti il problema di dover stabilire quando due numeri fossero uguali, e poichè i conti sul calcolatore non sono esatti, abbiamo dovuto tenere in considerazione che le uguaglianze in senso stretto non hanno senso, e dunque abbiamo introdotto le uguaglianze a meno di una tolleranza. Come valore per la tolleranza abbiamo scelto il valore $500^*\epsilon$ con ϵ = epsilon di macchina.
Il valore è stato scelto euristicamente grande a sufficienza per evitare di classificare due punti diversi come lo stesso, e piccolo abbastanza per sfruttare il più possibile la precisione fornita dai dati iniziali. Inoltre è stato inserito come variabile universale quindi facilmente modificabile

Altri dettagli:

- Abbiamo inoltre dovuto stabilire se alcuni segmenti fossero degeneri (distanza tra gli estremi troppo piccola) e per farlo abbiamo considerato anzichè la norma dei vettori la norma al quadrato evitando così di chiamare la funzione *norm* che include la radice quadrata, che è una funzione computazionalmente costosa.

- In alcune funzioni avremmo potuto utilizzare il comando inline per velocizzare il codice: *addPoint* e *addEdge* sono due metodi per i quali, il comando inline, sarebbe stato opportuno in quanto entrambi sono brevi e vengono richiamati di sovente.
  Inline evita che tali metodi vengano richiamati ogni volta, alleggerendo così lo stack frame.

- Nella seconda parte del progetto inizialmente avevamo individuato un altro algoritmo per risolvere il problema dei tagli delle fratture, ma procedendo nel suo sviluppo ci siamo resi conto che era molto dispendioso, in quanto si basava sul calcolo di molti prodotti vettoriali e di molti sistemi lineari, e lo abbiamo dunque accantonato per prendere in considerazione un'altra strada (che è risultata essere piuttosto efficiente).

Capitolo 3

Ottimizzazione

3.1 Euristica dell'algoritmo di ricerca delle tracce

Un aspetto fondamentale per quanto riguarda l'ottimizzazione del codice è l'algoritmo euristico utilizzato per ottimizzare il numero di confronti tra fratture per la ricerca delle tracce.
La scelta più ingenua è lanciare l'algoritmo di ricerca delle tracce su tutte le possibili combinazioni di fratture presenti: per quanto l'algoritmo sia capace di distinguere ed eliminare i casi di mancata intersezione, il costo computazione di una tale scelta diventa rapidamente ingestibile, in quanto ogni comparazione richiede la risoluzione di sistemi lineari.
Una prima ottimizzazione implementabile, seppur approssimativa, è stato calcolare per ogni frattura la sfera circoscritta ad essa al momento di creazione dell'istanza associata (con conseguente memorizzazione dei dati di centro e raggio all'interno dell'istanza stessa).
Il costo di ciò è $O(N)$, ma quest'informazione permette di ridurre in modo sostanziale i calcoli richiesti in seguito, poiché se le sfere associate a due fratture non sono intersecanti è possibile a priori escludere la possibilità di presenza di una traccia. Questa condizione è facilmente verificabile verificando se la distanza tra i due centri è minore della somma dei raggi delle due sfere.
Appare evidente che la sola implementazione di questo controllo garantisce una notevole riduzione del numero di operazioni da effettuare: in molti casi basta verificare la condizione di intersezione delle sfere senza dover passare dalla ricerca (vana) di punti di intersezione (ricordiamo che questo comporta la necessità di eseguire prodotti vettoriali e di risolvere svariati sistemi lineari) per essere sicuri che sia possibile escludere una coppia di fratture dall'elenco delle candidate possibili generatrici di tracce.

3.2 Partizionamento del dominio

Per quanto l'algoritmo introdotto in precedenza sia efficace, esso richiede comunque l'esecuzione di $O(N^2)$ confronti contenuti nelle relative guardie. Per quanto ciò sia comunque meglio dell'esecuzione alla cieca di sistemi lineari, essa contiene comunque un margine di miglioramento in termini di prestazioni e costi computazionali.
L'idea che abbiamo avuto è stata quella di introdurre un **partizionamento del dominio**, al fine di inserire ogni frattura nella sua regione di appartenenza e poter così limitare i controlli incrociati agli elementi di quella partizione.
L'implementazione da noi scelta consiste nell'analisi delle dimensioni effettive del dominio al fine di trovare il più piccolo parallelepipedo contenente tutte le fratture. Esso viene poi partizionato lungo ogni asse in k regioni, finendo così per generare un ricoprimento di k^3 partizioni di uguali dimensioni, indicizzate con un numero intero $i \in [0, k^3]$. Il criterio per assegnare a ogni frattura una partizione è il seguente: con un semplice calcolo andiamo a calcolare la partizione di appartenenza di ogni vertice della frattura, e nel caso essa dovesse coincidere per tutti i vertici andiamo ad assegnarla alla frattura.
Nel caso vertici diversi appartenessero a partizioni diverse? D'ora in avanti definiremo **degeneri** le fratture di questo tipo, e la nostra scelta è stata quella di inserirle in una partizione con id 0: esse andranno confrontate con tutte le fratture esistenti (è stata una scelta di compromesso per evitare che l'algoritmo di assegnamento alle partizioni diventasse estremamente complicato).
Come prima cosa bisogna osservare che il costo di assegnazione alle partizioni è $O(N)$, e a fronte di un un numero di confronti successivi dell'ordine di $O(N^2)$ esso non costituisce un peso eccessivo neanche nel caso peggiore, ossia quello in cui tutte le fratture ricadono nella partizione 0.
Andiamo ora ad analizzare, introducendo via via delle ipotesi sui dataset, le possibili situazioni che si possono verificare e come si comporta il partizionamento in questi casi. Nell'ipotesi di un dataset *clusterizzato* il numero di fratture degeneri è trascurabile, e con esso anche i confronti annessi (il cui numero non dipende dalla presenza delle partizioni per definizione), l'impatto sul costo computazionale della presenza delle partizioni è più o meno esteso in base alla distribuzione delle fratture tra le partizioni.
Nell'ipotesi aggiuntiva che le fratture siano distribuite più o meno equamente tra le partizioni, il costo computazionale passa da essere $f(N) \in \theta(N^2)$ all'incirca:

$$g(N) = \frac{f(N)}{a}$$

dove a rappresenta il numero totale di partizioni e la parte lineare è (formalmente) introdotta per tenere conto dei calcoli necessari per attribuire le partizioni. Un'analisi molto rapida fatta al computer mostra che se il numero di partizioni

a è nell'ordine di $log(N)$ (ipotesi molto ragionevole), il guadagno in termini di confronti richiesti si aggira su insiemi di fratture ragionevolmente grandi (migliaia) si attesta attorno al 80%/90% circa.
Nell'ipotesi ulteriore in cui la distribuzione delle fratture non sia uniforme, allora il costo computazione è drasticamente ammortizzato se non addirittura invariato (nel caso estremamente ottimista in cui una frattura è inserita in una partizione vuota).
Tutte queste considerazioni sono però da rapportare alle esigenze di ogni singolo caso: la nostra implementazione è tutt'ora estremamente rudimentale e serve solo da punto di partenza (si pensi alla possibilità di introdurre un partizionamento adattivo o alla possibilità di specificare manualmente la geometria del partizionamento).

3.3 Multithread

La creazione di un sistema di partizioni suggerisce la possibilità di implementare un'altra forma di ottimizzazione, ossia la possibilità di inserire una forma di calcolo parallelo nel programma. Nel nostro caso abbiamo osservato che era possibile introdurre, con uno sforzo minimo, una forma rudimentale di calcolo parallelo nella creazione delle mesh poligonali, dal momento che il problema di creazione delle stesse è un problema locale a ogni frattura che non richiede alcuna forma di sincronizzazione o comunicazione tra threads distinti (gli unici dati condivisi sono l'istanza della classe che gestisce l'elenco di tracce e il vettore di output).
L'utilizzo delle partizioni si presta anch'esso al calcolo in parallelo. La presenza di questo partizionamento del dominio ci permette di scomporre un problema globale come la ricerca delle tracce in tanti problemi locali indipendenti tra di loro. E' quindi ragionevole pensare che un buon passo in avanti possa essere una forma di implementazione di calcolo distsribuito, in quanto tutti gli algoritmi da noi implementati possono essere facilmente riscritti per agire su strutture dati distinte nella memoria, una per ogni istanza (ricomporre poi la soluzione globale consisterebbe solo nel concatenamento di vari vettori o nell'unione di insiemi di tracce).