UNIVERSITÀ DEGLI STUDI ROMA TRE
Sezione di Informatica e Automazione

Dipartimento di Ingegneria
Corso di Laurea in Ingegneria Informatica

Tesi Di Laurea

Implementazione di una libreria per metodo Lattice Boltzmann

Relatore
Prof. Franco Milicchio

Candidato
Luca De Silvestris
Matricola 486652

Anno Accademico 2016/2017

Ai miei genitori

Indice

Elenco delle figure

Elenco delle tabelle

Introduzione

Ringraziamenti

Non è facile andare a ritroso negli anni e rievocare ricordi, emozioni e sensazioni per arrivare ad avere una lista di coloro che mi hanno reso ciò che sono e che spero contribuiranno ad essere ciò che sarò. Desidero ringraziare tutte quelle persone che, con suggerimenti, critiche e osservazioni, hanno fornito un importante aiuto nell'esperienza di questi tre anni appena trascorsi, con la promessa di migliorarmi sempre. Ringrazio mia madre, mio padre e mia sorella a cui ho dedicato questi miei tre anni di studio, per aver creduto in me fin dall'inizio di questo percorso. Una dedica, in particolare, va a mia sorella Diandra: nella speranza che tu possa sempre gioire dei miei traguardi come io dei tuoi e con la consapevolezza che ti appoggerò sempre, anche se con una visione critica e cinica, in qualunque tua scelta futura. Colgo l'occasione per ringraziare coloro con cui ho condiviso lo stesso lungo tragitto di questi tre anni, tutta la rappresentanza studentesca, i miei colleghi e i miei amici. Un ringraziamento speciale va: ad Alessandro, con il quale ho cominciato questo percorso; ad Alex, confidente e grande compagna di avventure; ad Elena e Lorenzo, con i quali ci siamo rialzati; a Francesca, amica di vita. Ultima, ma non per importanza, ringrazio Rea Silvia, il mio lume della ragione, il mio sorriso.

Luca De Silvestris

Premessa

Il lavoro che verrà descritto in questa tesi rappresenta la relazione finale di un progetto svolto dal candidato nell'ambito di un tirocinio formativo della durata di tre mesi circa. Il progetto ha riguardato lo sviluppo di una libreria per l'utilizzo del metodo Lattice Boltzmann con l'obiettivo di essere di facile uso e soprattutto efficente a livello di risorse computazionali e di velocità.

Le macro-aree di interesse sono quindi la programmazione C++, la programmazione parallela e concorrente, la programmazione generica e la computer grafica applicata alla fluidodinamica. Tutti i concetti enunciati nella descrizione sintetica qui riportata verranno approfonditi, illustrati, documentati e motivati nel corso della lettura. Alla base dello studio vi è la necessità di mostrare l'importanza di questo metodo ormai così ampiamente utilizzato.
La tesi si concentra su questi sei punti chiave:

1. Stato dell' arte: breve panoramica sui concetti chiave del lavoro;

2. Strumenti e metodologie: un focus sugli strumenti e sulle tecnologie usate per risolvere i problemi e sulle metodologie;

3. Analisi dei requisiti e problemi: la definizione degli obiettivi del tirocinio e raccolta dei requisiti progettuali e problemi da risolvere individuati durante le fasi di analisi;

4. Progettazione e implementazione: la descrizione dell' architettura del progetto finale, con approfondimento delle scelte progettuali;

5. Esempi di utilizzo e sviluppi futuri: considerazioni sulle possibili implementa-

zioni future e esempi di applicazione;

6. Conclusioni: considerazioni sui risultati ottenuti.

Capitolo 1

Stato dell'arte

1.1 Fluido dimanica e computer grafica

In computer grafica è di grande interesse lo studio della fluidodinamica sia per studi scientifici sia per applicazioni pratiche tra cui simulazioni, elementi videoludici ed animati. Studiando un fluido da un punto di vista macroscopico, il suo comportamento é determinato dalle **equazioni di Navier-Stokes** .

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} = -\frac{1}{\rho}\nabla p + \gamma \nabla^2 \mathbf{u} + \frac{1}{\rho}\mathbf{F}$$

In fluidodinamica le equazioni di Navier-Stokes nel caso generale coinvolgono 5 equazioni scalari differenziali alle derivate parziali e 20 variabili descriventi il comportamento di un fluido la cui ipotesi di base è che questo possa essere modellato come un continuo deformabile. Esse devono il loro nome a Claude-Louis Navier e a George Gabriel Stokes che le formalizzarono e la loro soluzione analitica generale rappresenta attualmente uno dei problemi irrisolti della matematica moderna per il quale vale il premio Clay. Soluzioni analitiche particolari si hanno infatti

solo in caso di utilizzo di estreme semplificazioni.

Un approccio alternativo a queste simulazioni fluidodinamiche computazionali fu inventato alla fine degli anni '80 con i metodi reticolati dei gas. Questi metodi permettono alle particelle di muoversi in un reticolo discreto e che la massa e il momento siano conservati nelle collisioni locali. La grande diffusione di questi metodi è da accreditare alla notevole facilità di implementazione rispetto all'utilizzo delle equazioni di Navier-Stokes.

1.2 Metodo Lattice Boltzmann

Il metodo Lattice Boltzmann, spesso abbreviato con la sigla **LBM**, dal termine inglese Lattice Boltzmann method, è un insieme di tecniche (di CFD) usate per la simulazione dei fluidi. Invece di risolvere le equazioni di **Navier-Stokes**, l'equazione di Boltzmann viene risolta per simulare il flusso di un fluido newtoniano mediante modelli di collisione. Simulando l'interazione di un limitato numero di particelle, il comportamento del flusso viscoso emerge automaticamente dal movimento intrinseco delle particelle stesse e dai processi di collisione che ne conseguono [Ber17]. Nel tempo si è riscontrato lo straordinario successo applicativo di tali metodologie in ambito vario, riuscendo anche ad eseguire simulazioni sull' equazione di Schrödinger.

Il metodo Lattice Boltzmann si basa su tre pilastri fonzamentali: l'equazione di Boltzmann, lo schema a reticolo discreto detto Lattice e la distribuzione delle velocità. Tutti i punti verranno spiegati di seguito per dare una idea generale delle equazioni, dell'ambiente e dell'algoritmo utilizzati nel metodo Lattice Boltzmann.

1.2.1 Equazione di bolzmann

L'equazione di Boltzmann, conosciuta anche come equazione di Boltzmann per il trasporto, deve il suo nome proprio al celebre fisico austriaco Ludwig Eduard Boltzmann ed è data da

$$\partial_t f + v\partial_x f + F\partial_v f = \Omega$$

dove al primo membro c'è la derivata totale della funzione di distribuzione $f = f(x, c, t)$ mentre al secondo membro l'operatore di collisione Ω.

Il problema pricipale nella risoluzione dell'equazione di Boltzmann è dato dalla natura complicata delle collisioni. Non è quindi sorprendente che ne siano state proposte versioni alternative e più semplici, note come "modelli d'urto", a cui si è arrivati mediante approssimazioni. Quella di maggior utilizzo risulta essere di Bhatnagar, Gross e Krook (anche definita approssimazione di **BGK**), proposta nello stesso periodo anche da Walander, che linearizza l'operatore di collisione ottenendo l'equazione Boltzmann-BGK data da:

$$\frac{\partial f}{\partial t} + \vec{c}\nabla f = -\frac{1}{\tau}(f - f^(eq))$$

dove τ è il tempo dimensionale di rilassamento legato direttamente alla viscosità ed f^eq è la funzione di distribuzione di equilibrio.

Visto che le collisioni dipendono da quanto le distribuzioni si discostano da quelle di equilibrio, l'operatore di collisione assume un significato fisico che governa il rilassamento più o meno repentino verso una condizione di equilibrio, che dipende dalla viscosità del fluido in esame. I metodi reticolati di Boltzmann sono utilizzati dunque come tecnica di simulazione per sistemi **fluidodinamici** complessi alla cui base c'è la fisica computazionale. Nei metodi tradizionali si risolvono numericamente le equazioni di conservazione di proprietà macroscopiche

come massa, quantità di moto ed energia. Nei modelli di Boltzmann invece il fluido è costituito da particelle fittizie, e queste operano consecutivamente processi di **propagazione e collisione**, spostandosi su una **griglia reticolare discreta** (Figura 1.1).

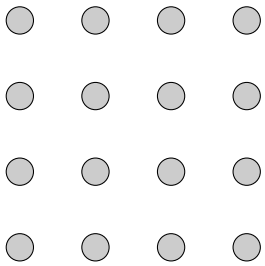

Figura 1.1: Esempio Griglia reticolare a due dimenioni

Data dunque una griglia generica si può decidere il sottospazio da utilizzare, avendo quindi libertà di stabilire su quante dimensioni implementare il modello e su quante direzioni si vuol simulare il movimento della particella.

1.2.2 Tipi di reticoli e classificazione DnQm

La potenza della discretizzazione dell'equazione di Boltzmann consiste nell'adozione di una griglia detta lattice. Questa griglia consta di nodi ugualmente spaziati tra loro e disposti secondo uno schema preciso, che dipende dal tipo di formulazione. Ogni schema definisce un set di direzioni predefinite con cui ogni nodo comunica con quelli adiacenti. Il riscontro fisico di tali direzioni si ritrova nelle velocità molecolari c ed è, a seconda della scelta del loro numero, vincolato a precise leggi di simmetria indispensabili per ricavare ad esempio le equazioni di flusso. Questi modelli vengono anche definiti **DnQm**, con n dimensioni e m velocità. Due dei modelli più utilizzati solo il **D2Q9** e il **D3Q27** spiegati man mano nella trattazione dell'articolo. In particolare in uno spazio a 2 dimensioni vari studi numerici hanno dimostrato che il reticolo quadrato a 9 velocità, oltre a

rispettare le leggi di simmetria, fornisce risultati più accurati di quelli basati sull' esagonale; inoltre è più semplice implementare differenti condizioni al contorno. In **Figura 1.2** é riportato un esempio a due dimensioni (D2) senza specifica delle velocitá.

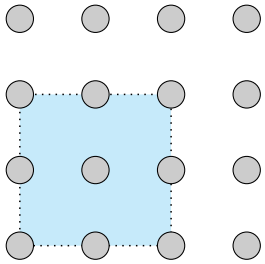

Figura 1.2: Esempio modello a due dimensioni

I modelli lattice Boltzmann possono operare su diversi tipi di reticoli, sia cubici che triangolari, con o senza particelle rimanenti nella funzione di distribuzione discreta.

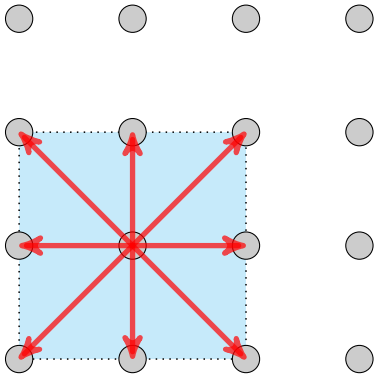

Figura 1.3: Cella LBM di un modello D2Q9

Decidendo quindi di associare al nostro modello di esempio a 2 dimensioni mostrato in Figura 1.2, nove diversi gradi di libertà o direzioni (anche chiamate

velocità) si avrà quindi un modello *D2Q9* in cui ogni particella possiede nove velocità diverse (8 velocità reticolari più la velocità nulla) come mostrato in **Figura 1.3**. Esprimendo quindi la discretizzazione come $c = \frac{\Delta x}{\Delta t}$ è possibile scrivere le velocità reticolari del modello *D2Q9* come:

$$
\begin{aligned}
c_0 &= c(0,0) \\
c_{1,3} &= c(\pm 1, 0) \\
c_{2,4} &= c(0, \pm 1) \\
c_{5,7} &= c(\pm 1, \pm 1) \\
c_{6,8} &= c(\mp 1, \pm 1)
\end{aligned}
$$

Le velocità descritte qui sono riportare in **Figura 1.4**.

1.2.3 Stream and collision

La principale caratteristica dell'algoritmo LBM, oltre alla locale conservazione della massa e del momento, è l'alternanza delle fasi di **streaming** e **collision** ovvero processi di propagazione e di collisione all'interno della griglia reticolare discreta (*Lattice*). Queste fasi sono descritte anch'esse nell'equazione cinetica di Boltzmann.

In un primo momento le particelle si propagano tra le varie celle seguendo la propria direzione reticolare e in un secondo, dopo un intervallo di tempo, ogni particella si sposterà sul nodo vicino, a seconda della propria direzione. Qualora sullo stesso nodo arrivassero particelle da più direzioni e con velocità diverse, queste collliderebbero e cambierebbero le loro direzioni in base ad un insieme di regole di collisione. Regole di collisione appropriate dovrebbero conservare

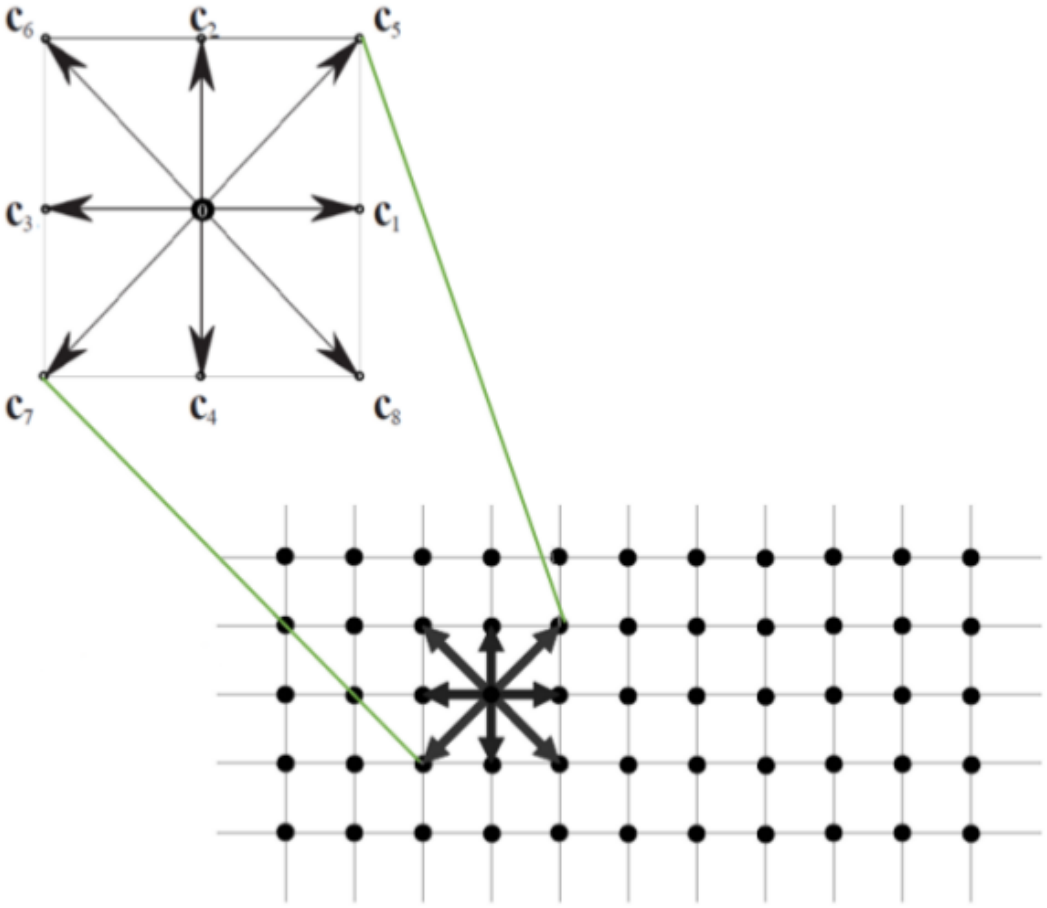

Figura 1.4: zoom sulle veocità di un reticolo D2Q9

il numero di particelle, il momento e l'energia prima e dopo la collisione. Nel capitolo successivo verranno descritti gli strumenti utilizzati per la realizzazione per passare poi alle tecnologie e alle metodologie utilizzate con relativi esempi di impiego e motivazioni.

Capitolo 2

Tecnologie e metodologie

Di seguito sono riportate tutte le tecnologie utilizzate, ognuna accompagnata da una descrizione, che a volte comprenderà anche un esempio di utilizzo e la motivazione che ha spinto ad utilizzare tale tecnologia. Oltre questo sono anche riportate tutte le metodologie applicate per poter realizzare il progetto. Tutto ció che verrà descritto è stato utilizzato su una macchina con sistema operativo **Linux**. Per la precisione è stata scelta la distribuzione *Ubuntu*(nella sua versione 16.10). Tutti i processi sono stati eseguiti su un laptop Lenovo modello G50-80 dotato di 8Gb di RAM.

2.1 Linguaggio e ambiente di sviluppo

2.1.1 C++

Nel corso del lavoro svolto sono state adottate varie tecnologie a partire dal linguaggio di programmazione utilizzato fino alle scelte metodologiche. Per quanto riguarda la programmazione la scelta è ricaduta sul linguaggio **C++**, linguaggio di programmazione orientato agli oggetti con tipizzazione statica sviluppato da

Bjarne Stroustrup ai Bell Labs nel 1983 come un miglioramento del linguaggio C. Standardizzato nel 1998 fu poi successivamente sostituito dal C++11 e dopo una revisione minore avvenuta nel 2014(C++14) si è arrivati poi all'ultima versione dello standard pubblicata nel 2017 con il nome di **C++17**, utilizzato nella realizzazione del progetto. Già lo standard del 1998 consisteva di due parti: il nucleo del linguaggio e la libreria standard che include gran parte della **Standard Template Library** che sarà di particolare importanza ma é anche possibile, usando il linking esterno, utilizzare librerie esterne. Il motivo di tale scelta risiede nella velocità computazionale di questo particolare linguaggio orientato agli oggetti. La velocità computazionale è infatti alla base del lavoro svolto. Nel progetto originario il linguaggio utilizzato era il **Fortran** ma si è optato per un linguaggio più moderno ma che mantenesse lo stesso principio cardine. Fortran e C++ hanno infatti una velocità computazionale molto simile a parità di operazioni eseguite.Il Fortran, sviluppato a partire dal 1954 da un gruppo di lavoro guidato da John Backus, è stato il primo linguaggio di programmazione di alto livello della storia nonchè il primo ad utilizzare un compilatore.

2.1.2 IDE e tools

Si è poi passati alla scelta dell'ambiente di lavoro, trovando in **Clion** la soluzione migliore. Clion, sviluppato da **JetBrains**®, un'azienda di sviluppo software che offre una vasta gamma di prodotti per svariati linguaggi, è un IDE multipiattaforma per C e C++. Un IDE (acronimo di Integrated Development Environment) è uno strumento software che, in fase di programmazione, aiuta i programmatori nello sviluppo del codice sorgente di un programma e consiste di più componenti, da cui appunto il nome integrato. Tra le principali componenti si trovano: editor di codice sorgente, compilatore, tool di building automatico, a cui poi spesso si

aggiunge un debugger. Si è optato per l'utilizzo di un tool modulare da affiancare allo sviluppo del progetto in modo da automatizzarlo ed è stato dunque introdotto il tool **Cmake**. CMake, nome derivato da una abbreviazione di "cross platform make", è uno strumento open source e multipiattaforma progettato per creare, testare e pacchettizzare software. CMake viene utilizzato per controllare il processo di compilazione del software utilizzando semplici file di configurazione indipendenti dalla piattaforma e dal compilatore e generare **makefile** automatizzando l'operazione *make*. Cmake dispone di una sintassi che comprende moltissime macro da utilizzare in uno specifico file chiamato **cmakeLists.txt**, da cui poi si genererà il Make file e successivamente si compila il progetto. Si passa di seguito a mostrare un esempio di codice e delle potenzialità del tool descritte anche nel [KM13]. In **Figura 2.1** è mostrato un esempio molto semplificato di come Cmake riesca in poche righe a gestire un piccolo progetto C++.

```
cmake_minimum_required (VERSION 3.6)  # versione minima di Cmake
project (Tutorial)  # nome del progetto


# The version number.
set (Tutorial_VERSION_MAJOR 1)
set (Tutorial_VERSION_MINOR 0)
# specifica la versione di c++ che verrà utilizzata
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

# add the executable
add_executable (Tutorial main.cxx)
```

Figura 2.1: Esempio base di creazione file CmakeList.txt

Vediamo adesso come si presenta il file CmakeList della libreria. All' esempio di base sono stati aggiunti e utilizzati diversi comandi tra i quali ***find package*** e

```
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

#SET variables
list(APPEND CMAKE_MODULE_PATH "~/tbb-2017_U7/cmake")

#use file for gob all the header/source file
file(GLOB headers
"boltzmann/TBB_Utility/*.hpp"
"mph/*.hpp"
"fmt/*.h"
"unittest/*.hpp"
)
#use file for gob all the header/source file
file(GLOB sources
        "fmt/*.cc"
        "unittest/*.cpp"
)

#include tbb
find_package(TBB COMPONENTS tbbmalloc tbbmalloc_proxy)

if (TBB_FOUND)
        INCLUDE_DIRECTORIES("${TBB_INCLUDE_DIR}")
        link_directories("${TBB_LINK_DIR}")
endif (TBB_FOUND)

add_executable(Boltzmann ${headers} ${sources})

#link the executable
target_link_libraries(Boltzmann libtbb.so)
```

Figura 2.2: File CmakeList del progetto

il comando ***file*** i quali impieghi saranno spiegati di seguito. **find_package** viene utilizzato per poter trovare e rendere quindi utilizzabile la libreria esterna di cui si è fatto uso (e che verrà introdotta tra poco). **file** è stato invece introdotto nel CmakeList per poter unire più file in modo da avere quindi un codice più chiaro e facilmente modificabile. Il file CmakeLists.txt come mostrato in **Figura 2.2** si conclude poi con i due comandi che hanno l'impiego rispettiamente di creazione dell'eseguibile, a cui vengono assegnati nome e file compresi, e il linkaggio di librerie esterne, a cui sono i richiesti come parametri il nome dell'eseguibile e lo

"shared object" ovvero il file di libreria compilato che nel sistema operativo linux ha estensione .so, simile ai file DLL di Windows. Si evince dunque l'importanza della scelta di questo particolare tool in grado di offrire una gestione del progetto ottimale con un linguaggio semplice ed ad alto livello.

2.2 Programmazione concorrente e parallela

2.2.1 Introduzione

Sempre tenendo presente la politica di un software leggero e computazionalmente veloce è stata introdotta la programmazione multithreading mediante l'utilizzo della libreria Intel®**Threading Building Blocks** [rei07] (da qui in avanti chiamata IntelTBB). La programmazione parallela è l'esecuzione di uno o più programmi, su più microprocessori o più core dello stesso processore, allo scopo di aumentare le prestazioni di calcolo del sistema di elaborazione ma introducendo il rischio di stallo. Un noto esempio che illustra egregiamente i problemi del controllo della concorrenza e della sincronizzazione fra processi paralleli è il problema dei filosofi a cena o problema dei **cinque filosofi** che verrà descritto di seguito a puro titolo di esempio per introdurre il concetto di concorrenza in informatica.

2.2.2 Problema dei cinque filosofi

Il problema dei cinque filosofi, descritto da Edsger Dijktra nel 1965 per esporre un problema riguardante appunto la sincronizzazione, li descrive seduti su una tavola rotonda. Ognuno di loro ha un piatto davanti, una forchetta sulla destra e una sulla sinistra appoggiate sul tavolo per un totale quindi di cinque forchette e cinque piatti. Immaginando la vita di un filosofo fatta da periodi alterni di

pensare e mangiare e che ognuno di loro abbia bisogno di due forchette per poter mangiare ma che esse debbano essere prese una per volta, si richiede di arrivare ad una soluzione in grado di far mangiare i cinque filosofi. In altre parole è richiesto di sviluppare un algoritmo che impedisca lo stallo, in informatica un **deadlock**, situazione in cui due o più processi o azioni si bloccano a vicenda aspettando che uno esegua una certa azione, che può verificarsi quanto tutti e cinque i filosofi possiedono una sola forchetta, e la morte d'inedia o starvation, ovvero l'impossibilità perpetua, da parte di un processo pronto all'esecuzione, di ottenere le risorse sia hardware sia software di cui necessita per essere eseguito, che può verificarsi nel caso in cui uno dei due filosofi non riesce a prendere mai entrambe le forchette.

2.2.3 Tipologie di parallelismo

Si possono utilizzare più tipologie di parallelismo classificate in due grandi sotto-categorie: interazione dei processi e decomposizione dei problemi. Nella categoria della decomposizione dei problemi si può avere il parallelismo sui dati e sulle attività.

Il parallelismo a livello di dati, anche noto come **Data parallelism** è una forma di parallelismo focalizzata sulla suddivisione di un dato in più nodi in modo da operare su più parti in parallelo. Può infatti esser applicata alle regolari tipologie di strutture dati quali array, vettori e matrici.

Il parallelismo a livello di attività invece, anche noto come **Task parallelism** è focalizzato sulla distribuzione delle attività su più processi da svolgere in parallelo. Un tipo comune e molto utilizzato di parallelismo è definito **Pipelining** e consiste nello spostare un singolo set di dati attraverso una serie di attività risultando quasi una combinazione tra i due tipi di parallelismo. Alla base di questo tipo di

programmazione ormai ampiamente utilizzata oltre ai concetti di dati e attivitá si deve prender nota della definizione di thread. Un **thread di esecuzione** è una suddivisione di un processo in più filoni o sottoprocessi che vengono eseguiti concorrentemente da sistemi monoprocessori, multiprocessori e multicore.

2.2.4 Prestazioni

È riportato ora a puro titolo di esempio il frammento di codice in **Figura 2.3** con relativo output mostrato in **Tabella 2.1** in cui è mostrato il tempo di esecuzione di un semplice programma che calcola la norma di un vettore.

Numero valori inseriti	Thread utilizzati	tempi di esecuzione
2000	1	7,75* 10^-5 s
2000	4	7,75* 10^-5 s
4000	1	0.00013 s
4000	4	0.0001 s
6000	1	0.00026 s
6000	4	0.00021 s
10000	1	0.00040 s
10000	4	0.00033 s
20000	1	0.00061 s
20000	4	0.00052 s

Tabella 2.1: output dei tempi per calcolo norma di un vettore

Parlando di prestazioni però si devono introdurre il concetto di scalabilità e la legge di Amdahl. La scalabilità di un programma in ambito di programmazione parallela è la misura dello Speedup in rapporto alla quantitá di core e thread nel processo. Speedup è il rapporto tra il tempo necessario ad eseguire un programma senza parallelismo rispetto al tempo in cui viene eseguito in parallelo. A puro titolo di esempio uno Speedup 2X indica che il programma parallelo verrà eseguito impiegando la metà del tempo dello stesso processo lanciato in sequenziale. La **legge di Amdahl** invece, ideata nel 1967 da Gene Amdahl, progettista noto per

```
class SommaVettore
{
    vector<int>& vettore;
public:
    float sum;
    void operator() (const blocked_range<size_t>& r )
    {
        for( size_t i=r.begin(); i!=r.end( ); ++i )
        {
            vettore[i] = pow(vettore[i], 2);
            sum += vettore[i];
        }
    }
    SommaVettore( SommaVettore& x, split ) : vettore(x.vettore), sum(0) {}

    void join( const SommaVettore& y ) {sum+=y.sum;}

    SommaVettore(vector<int>& vettore ) : vettore(vettore), sum(0) {}
};


int main(int argc, char** argv)
{
    // Construct task scheduler with p threads
    int n = task_scheduler_init::default_num_threads();
    task_scheduler_init init(n);

    //execute parallel algorithm using task here
    vector<int> vettore;
    for (int i=1;i<argc+1;i++)
        vettore[i]=(strtod(argv[i], NULL));

    //start time
    tick_count t0 = tick_count::now();

    SommaVettore somma(vettore);
    parallel_reduce(blocked_range<size_t>(0, argc),somma);

    //end time
    tick_count t1 = tick_count::now();
    double t = (t1 - t0).seconds();

    //Final time print.
    cout << "time = " << t << " with " << n << " threads\n";

    return 0;
}
```

Figura 2.3: Codice di esempio di programma che calcola la norma di un vettore

le sue osservazioni sul miglioramento massimo di un sistema informatico, afferma che il miglioramento che si può ottenere su una certa parte del sistema è limitato dalla frazione di tempo in cui tale attività ha luogo. Questo denota anche

che se accelerassimo tutto in un programma di 2X, potremmo aspettarci che il programma risultante funzioni due volte più velocemente. Tuttavia, migliorando le prestazioni di solo metà del programma di 2X, il sistema complessivo migliora solo di 1,33X. In formule la legge di Amdahl si può esprimere come:

$$\frac{1}{(1-P)+\frac{P}{S}}$$

con S fattore di quanto si riduce il tempo di calcolo dopo l'aggiunta del miglioramento(ad esempio se il miglioramento raddoppia la velocità della porzione modificata allora S varrà 2) e con P proporzione tra la percentuale a cui é applicato il miglioramento e l'intero programma. La legge denota anche che al crescere del numero di processori che lavorano in parallelo l'incremento di prestazioni diventa sempre minore dato che la componente non parallelizzabile dei programmi diventa sempre più significativa nei tempi di calcolo totali.

2.3 Programmazione generica

Si è scelto di applicare al progetto una metodologia di programmazione detta **programmazione generica** (Generics programming) in quanto ritenuta la più consona per assicurare efficienza e flessibilità al progetto. La programmazione generica riguarda la generalizzazione dei componenti software in modo che possano essere facilmente riutilizzati in un' ampia varietà di situazioni. In particolare la programmazione generica è stata introdotta mediante l'utilizzo dei **templates** e delle nozioni raccolte nel libro "Modern C++ Design: Generic Programming and Design Patterns Applied"[Ale07] I templates di classi e metodi sono meccanismi particolarmente efficaci per la programmazione generica perchè rendono possibile la generalizzazione senza sacrificare l'efficienza. La programmazione generica

mediante l'utilizzo dei template è stata inserita poichè, nonostante C++ sia un linguaggio fortemente tipizzato in cui quindi tutte le variabili devono avere un tipo specifico, le funzioni sono spesso identiche indipendentemente dal tipo di dati. Si passa di seguito a mostrare un esempio di utilizzo della programmazione generica e di come essa sia implementata nel linguaggio C++ attraverso la (**Figura 2.4**) e la (**Figura 2.5**) in cui sono mostrati rispettivamente le definizioni di base di una classe per una struttura *Coda* e di un metodo per l'inserimento di un elemento della coda (templetizzati). Inoltre l'introduzione della programmazione generica evita anche di avere la ripetizione di codice e quindi aumenta il riutilizzo e l'affidabilità dello stesso,favorendo insieme all'ereditarietà l'introduzione del **polimorfismo** che sia esso per esclusione (tramite l'ereditarietà) o parametrico (tramite la programmazione generica)

```
//classe Coda templetizzata
template <class T> class Coda
{
public:
        Coda(); //costruttore
        ~Coda(); // distruttore

        //metodi base per una coda
        void inserisci( const T& );
        T& rimuovi();
        bool vuota();
private:
        //metodi privati...
};
```

Figura 2.4: esempio di una classe template

```
//metodo inserisci della classe Coda, templetizzato
template<class T> void Coda<T>::inserisci(const T& val)
{
        CodaItem<T> *pt = new CodaItem<T>(val);
        if(vuota())
                testa = coda = pt;
        else
                {
                coda->succ = pt;
                coda = pt;
                }
}
```

Figura 2.5: esempio di un metodo template

Anche nel lavoro svolto è stato fatto un largo utilizzo della programmazione generica soprattutto per poter implementare le celle di cui è composto il reticolo discreto di interesse. Come si vede nella (**Figura 2.6**) nella classe cell_traits oltre a renderla templetizzata sono state aggiunte piccole accortezze per un utilizzo più

chiaro delle variabili *value_ type* e *storage_ type* oltre ai metodi base che ritornano la dimensione e le velocità della cella, le quali ovviamente una volta stabilite dall' utilizzatore sono fisse.

```
/// Cell traits class
template<class T> class cell_traits
{
public:

    /// Value type stored in one cell
    typedef typename T::value_type value_type;

    /// Storage type of all probability distributions
    typedef typename T::storage_type storage_type;

    /// Return the dimension of the cell (e.g., 2D, 3D)
    static constexpr std::size_t get_dimension()
    {
            return T::dimension;
    }

    /// Return the number of probability distributions of the cell (e.g., for D2Q9 is 9)
    static constexpr std::size_t get_distributions_number()
    {
            return T::distributions;
    }

    /// Returns the displacements of all probabilities (e.g., with D2Q9, South-West is { -1, -1 })
    static const constexpr storage_type& get_distribution_displacements(){ return T::storage }

};
```

Figura 2.6: Codice della classe cell_traits

2.4 Scelta della struttura dati

Anche nel lavoro svolto la scelta delle strutture dati da utilizzare è ricaduta sugli *vector<>* della C++ Standard Library. Come espresso anche nel libro "C++ Linguaggio, libreria standard, principi di programmazione" [Str00] gli elementi di un vettore sono memorizzati in modo contiguo e con un utilizzo limitato della memoria. A differenza di altri contenitori STL, come deques e liste, i vettori consentono all'utente anche di indicare una capacità iniziale per il contenitore

e,inoltre, permettono l'accesso casuale. Importante è la gestione e l'utilizzo della memoria stack e heap della macchina. I vettori vengono creati nella memoria stack ma gli oggetti di cui è composto vengono inseriti nell'heap. Al contrario, strutture come ad esempio gli Array inseriscono gli elementi direttamente nello stack.

Come espresso anche in "Operating Systems: Internals and Design Principles"[Sta14] uno **stack** è un'area di memoria del computer con un'origine fissa e una dimensione variabile. Inizialmente la dimensione della pila è zero. Un puntatore dello stack, solitamente sotto forma di un registro hardware, punta alla posizione di riferimento più recente nello stack; quando lo stack ha una dimensione pari a zero, il puntatore dello stack punta all'origine dello stack. Si fa presente inoltre che nel caso di piú thread ognuno di essi possiederà un suo stack personale.

L'**heap** è la memoria riservata per l'allocazione dinamica. A differenza dello stack, non esiste un modello forzato per l'allocazione e la deallocazione dei blocchi dall'heap. Ciò rende molto più complesso tenere traccia di quali parti dell'heap siano allocate o libere in un dato momento; esistono molti allocatori di heap personalizzati disponibili per ottimizzare le prestazioni dell'heap per diversi modelli di utilizzo. Infine al contrario dello stack, nel caso di utilizzo di pi‘u thread di esecuzione questi condivideranno comunque lo stesso heap.

Capitolo 3

Analisi e problemi

In questo capitolo, dopo aver preso nota delle tecnologie e metodologie utilizzate e descritte nel *Capitolo 2* ,si vedrà più in dettaglio lo studio dei requisiti e dei problemi risolti nel lavoro svolto,prima di passare alla descrizione della progettazione dell'applicativo.

3.1 Obiettivi formativi

Gli obiettivi formativi inizialmente pensati per essere realizzati durante lo svolgimento del tirocinio,erano quelli di poter realizzare una libreria in grado di poter utilizzare due tipi di algoritmi: algoritmo con fase di streaming e fase di collisione e un algoritmo in cui si sarebbe eliminata la fase di streaming e di vedere le differenze a livello di tempo di esecuzione tra i due. Date le tempistiche ridotte del tirocinio questa idea è stata però abbandonata per dare maggior attenzione allo studio delle tecniche per aumentare le prestazioni computazionali arrivando a due obiettivi da raggiungere: lo sviluppo di un software veloce che potesse essere di facile impiego per l'utente utilizzatore, il tutto completato dall'implementazione di classi automatizzate di test che accompagnano lo sviluppo. LâĂŹattività di

definizione dei requisiti a breve termine è stata, piú che un punto di partenza,una fase ricorrente del lavoro alternata allo sviluppo vero e proprio della libreria. Il paragrafo che segue raccoglie tutti i requisiti che sono stati individuati e le problematiche da risolvere prese in considerazione durante il lavoro.

3.2 Analisi dei requisiti e scelte progettuali

Gran parte dello studio rivolto all'analisi dei requisiti è stato svolto, come si addice ad un software sviluppato con **metodologia agile**, nelle prime iterazioni e riguarda tutte quelle esigenze descritte nel *Capitolo 1* e che poi hanno trovato un buon riscontro nelle tecnologie analizzate nel *Capitolo 2*. Di seguito in **Figura 3.1** è descritto in maniera semplificata il diagramma degli oggetti di dominio, risultato dall' analisi del progetto nella prima iterazione.

Come si nota la classe templetizzata descritta nel *paragrafo 2.3* è stata fin da subito pensata come l'oggetto base da cui partire e su cui basare le iterazioni. Un altro fattore su cui si è scelto di lavorare per facilitare l'utilizzo come si conviene ad una buona libreria. Per rendere la suddetta accessibile e utilizzabile in pieno si è scelto di dare poche funzionalità all'utente mostrando dunque solo le operazioni di interesse lasciando a lui la possibilità di inserire alcune condizioni di utilizzo che saranno mostrate nel capitolo successivo.I problemi che sono stati affrontati con maggior interesse riguardano due importanti fattori: la facilità di impiego e la velocità computazionale.

Per quanto riguarda la velocità computazionale la soluzione ottima o comunque quella che si è visto essere la migliore è stata l'introduzione della programmazione parallela e concorrente e la templetizzazione delle classi. La programmazione generica in C++ infatti, nella creazione degli oggetti, dà un grande aiuto non

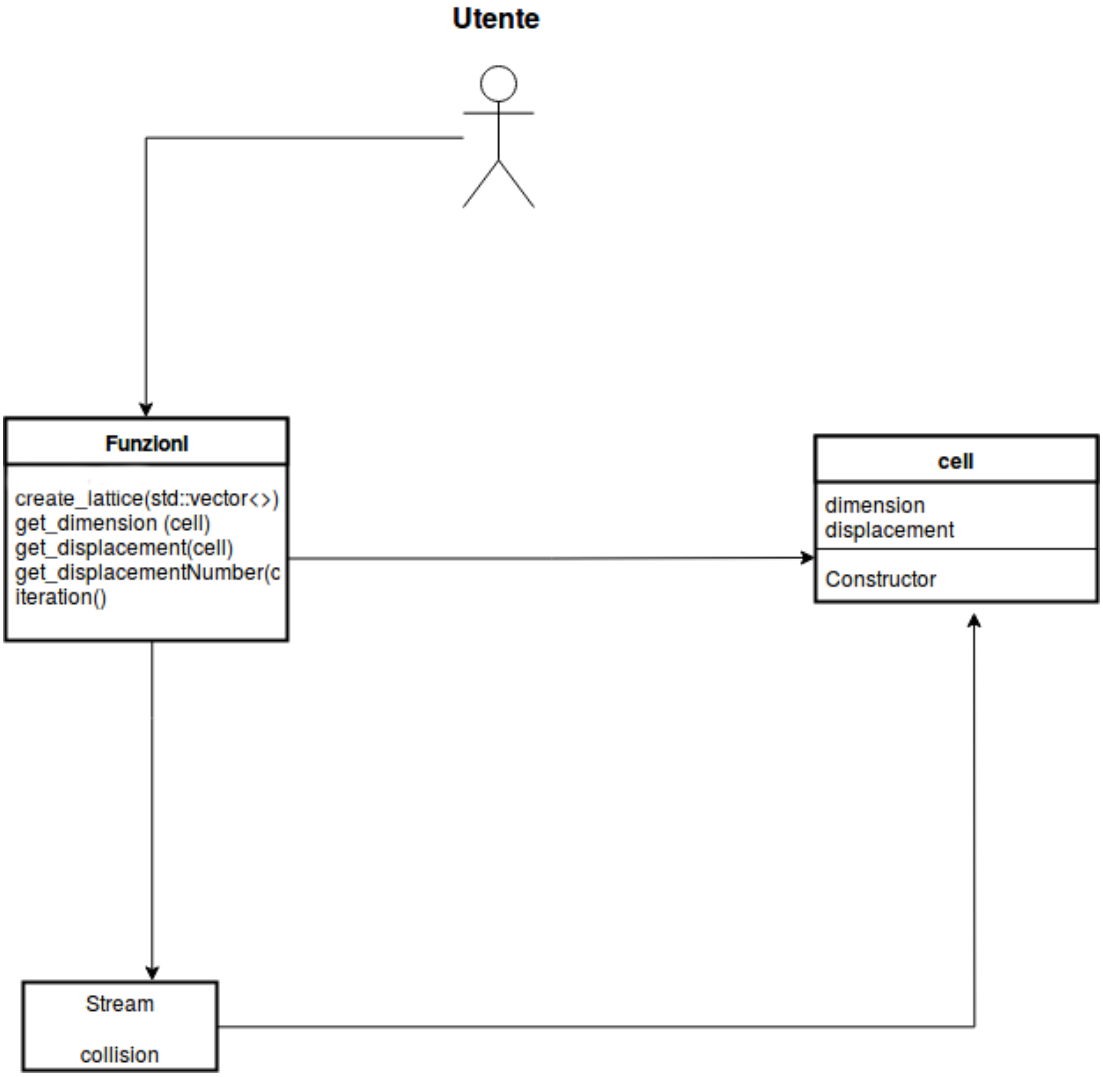

Figura 3.1: diagramma degli oggetti ideato nella prima iterazione

solo per quanto riguarda la flessibilità del progetto ma anche alla velocità computazionale, in quanto i template vengono elaborati e creati a tempo di compilazione lasciando quindi meno lavoro da svolgere a tempo di esecuzione dell'applicativo. Per poter essere di sostegno a qualunque tipologia di progetto e impiego si è preferito lasciare all'utente la possibilità di inserire le condizioni di bordo che egli

ritiene più utili per il proprio scopo. Per **condizioni di bordo** si intendono le condizioni che rispondono alla domanda di come si comporta una particella con una certa velocità quando "incontra" il bordo fittizio del reticolo scelto.

Durante una successiva analisi si è poi optato per alcune modifiche progettuali che sono mostrate nel diagramma ad oggetti di seguito in Figura 3.2 seguendo i principi espressi dal libro "UML distilled. Guida rapida al linguaggio di modellazione standard"[Fow02] ed utilizzando **draw.io**, software di diagrammi online gratuito per la creazione di diagrammi di flusso, diagrammi di processo, organigrammi, ER e diagrammi di rete.

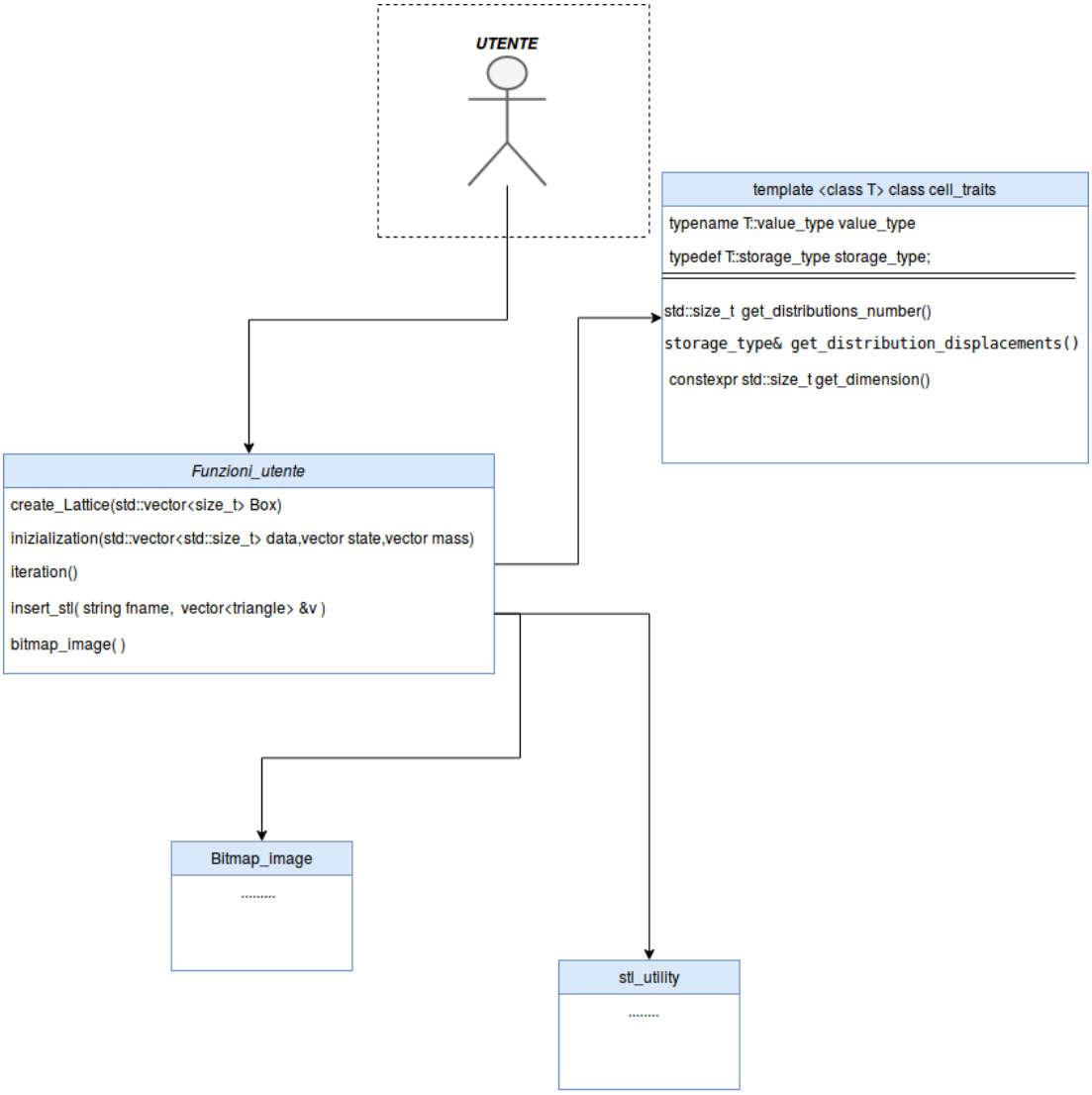

Figura 3.2: diagramma degli oggetti ideato nella seconda iterazione

Nell'ingegneria del software, UML, acronimo di unified modeling language ovvero di "linguaggio di modellizzazione unificato" è un linguaggio di modellazione e specifica basato sul paradigma orientato agli oggetti. Il nucleo del linguaggio fu definito nel 1996 da Grady Booch, Jim Rumbaugh e Ivar Jacobson. Lo stantard è tuttora gestito dall'Object Management Group. UML svolge un'im-

portantissima funzione di "lingua franca" nella comunità della progettazione e programmazione ad oggetti utilizzato anche dalla gran parte della letteratura del settore informatico per descrivere soluzioni analitiche e progettuali in modo sintetico e comprensibile ad un vasto pubblico.

Tutto ciò che é stato descritto fino ad ora in questo capitolo rientra in tutta quella branca dell'informatica definita ingegneria del software che divide la creazione del progetto in tre parti distinte ma sviluppate quasi in parallelo: analisi, progettazione ed implementazione. La progettazione è infatti una fase del ciclo di vita del software. Sulla base della specifica dei requisiti prodotta dall'analisi, la progettazione ne definirà i modi in cui tali requisiti saranno soddisfatti, entrando nel merito della struttura che dovrà essere data al sistema software che deve essere implementato. In particolare durante il lavoro svolto è stata utilizzata una metodologia definita come **Agile**.

Nell'ingegneria del software, l'espressione "metodologia agile", o sviluppo agile del software, si riferisce a un insieme di metodi di sviluppo del software emersi a partire dai primi anni 2000. Di grande spunto è stato anche il libro "Agile Software Development: Principles, Patterns, and Practices"[Mar02] I metodi agili si contrappongono al modello a cascata e altri processi software tradizionali, proponendo un approccio meno strutturato e focalizzato sull'obiettivo di consegnare al cliente, in tempi brevi e frequentemente, software funzionante e di qualità. Questi principi sono definiti nel "Manifesto per lo sviluppo agile del software" [KB01], pubblicato nel 2001 da Kent Beck, Robert C. Martin e Martin Fowler in cui si specificano le seguenti considerazioni:

Gli individui e le interazioni pù che i processi e gli strumenti

Il software funzionante più che la documentazione esaustiva

La collaborazione col cliente più che la negoziazione dei contratti

Rispondere al cambiamento più che seguire un piano

Fra le pratiche promosse dai metodi agili si trovano: la formazione di team di sviluppo piccoli, poli-funzionali e auto-organizzati, lo sviluppo iterativo e incrementale, la pianificazione adattiva, e il coinvolgimento diretto e continuo del cliente nel processo di sviluppo. Si passa ora ad analizzare la libreria con particolare interesse all' inizializzazione e alla fase di spostamento delle particelle (detto *Stream*).

Capitolo 4

Progettazione e implementazione

In questo capitolo saraà illustrato il risultato del lavoro di progettazione e implementazione svolto fino al termine del tirocinio; i problemi individuati nel capitolo 3 sono stati qui risolti utilizzando le tecnologie e le metodologie illustrate nel capitolo 2 partendo dalla costruzione del reticolo discreto (Lattice). Si procederà a descrivere la libreria partendo dalle prime operazioni svolte dall'utente.

4.1 Implementazione del reticolo discreto

Al fine di realizzare un software utilizzabile in più situazioni l'utente utilizzatore ha la possibilità di indicare il numero di dimensioni su cui si andrà a lavorare, ovvero se utilizzare la libreria su uno, due oppure su tre assi e di specificare il numero di direzioni o velocità da assegnare ad ogni particella. Questa creazione del reticolo è possibile utilizzando la classe delle celle templetizzata
template <std::size_t dim, std::size_t dis, class Index, Index const (*v)[dis][dim], class T = double> **class lb_cell**
in cui *dim* specifica la dimensione, *dis* il numero delle velocità e *const (*v)[dis][dim]* specifica la loro distribuzione nello spazio così come visto nel paragrafo 1.2.2 in

cui si sono trattate le tipologie di reticoli discreti più utilizzati e la classificazione $DnQm$.

Considerato poi il grande utilizzo delle classificazioni D2Q9 e D3Q27 si è provveduto ad implementare questi due specifici reticoli discreti distribuendo le velocità nel caso del D2Q9 nel modo illustrato di seguito:

static const int **D2Q9_lattice[9][2]**{
{ +0, +0 }, // center d0
{ +0, +1 }, // north d1
{ +1, +1 }, // north-east d2
{ +1, +0 }, // east d3
{ +1, -1 }, // south-east d4
{ 0, -1 }, // south d5
{ -1, -1 }, // south-west d6
{ -1, 0 }, // west d7
{ -1, +1 } // south-east d8

Distribuzione delle velocità per reticolo discreto D2Q9

Utilizzando la funzione **Using** del linguaggio, che consente la dichiarazione di un tipo e l'utilizzo di un alias che prende le specifiche indicate:

using **D2Q9** = lb_cell<2, 9, int, **&D2Q9_lattice**, double>;

si è poi facilitata l'assegnazione dei valori della classe generica nel caso delle classificazioni sopra citate per una assegnazione quanto più immediata possibile. Da notare infatti che utilizzando l'alias D2Q9 si costruisce il reticolo utilizzando la distribuzione mostrata sopra. Si ricorda inoltre che nel linguaggio di pro-

grammazione C++ la & restituisce l' indirizzo di memoria dell' operando cui è applicato.

4.2 Creazione dello spazio discreto

Una volta deciso il tipo di reticolo discreto da utilizzare si passa alla costruzione e alla creazione e all'inizializzazione dei vari contenitori o *storage*(nome utilizzato nella libreria).

La creazione avviene mediante un costruttore a cui vengono passati in input la dimensione degli assi su cui verrà creato il reticolo. A puro titolo di esempio se si stesse utilizzando un sistema a due dimensioni, inserendo in input i valori {3,3} in cui il primo valore rappresenta la dimensione desiderata dell'asse X e il secondo valore l'asse delle Y il reticolo discreto verrebbe costruito come in **Figura 4.1**

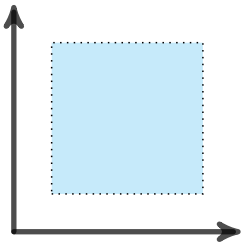

Figura 4.1: spazio discreto

È stato inoltre implementato un sistema che attua la strategia del controllo incrociato, riportando un messaggio di errore nel momento in cui si prova ad inserire valori per più o meno assi cartesiani di quelli su cui si è scelto di lavorare, in modo tale da evitare un utilizzo non desiderato. Utilizzato il costruttore il ri-

sultato sarà la creazione dello "spazio di lavoro" che risulterà però vuoto; l'utente può ora passare alla fase di inizializzazione.

4.3 Inizializzazione degli Storage

L'inizializzazione avviene invocando l'apposito metodo in cui si ha la possibilità di inserire:le **direzioni**, ovvero le velocità e che costituiranno le nostre particelle; lo **stato** della cella, in cui si specifica lo stato fisico della materia di cui è composta la cella; la massa dei punti.

Per l'implementazione come si è detto nel **Capitolo 2**, in cui si sono trattati gli strumenti e le metodologie applicate, si è optato per l'utilizzo degli *STD::VECTOR<>* generalizzando il tipo di dati. Utilizzando la funzione **typedef** del linguaggio i vari vettori sono definiti come

typedef std::vector<value_type> **storage_type**

avendo dunque la generalizzazione sul tipo di dato utilizzato di modo da poter avere una sola definizione di "storage" e render il codice più pulito. Nella libreria i vari vector sopracitati sono denominati rispettivamente: *Storage*, *state_points* e *mass_of_points*, tutti del tipo *storage_type* visto prima. La struttura più

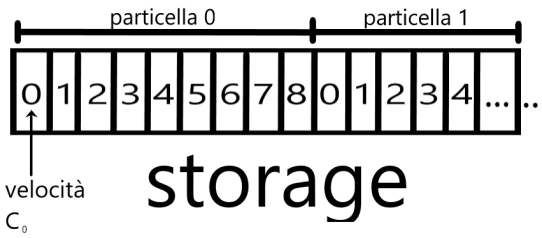

Figura 4.2: vettore Storage per modello D2Q9

"complessa" è quella dello **storage** in cui sono inserite le particelle e le rispettive velocità in un unico *vector* secondo lo schema riportato in **Figura 4.2**

Come si vede in figura è stata utilizzata una rappresentazione del D2Q9 e si nota che ogni particella può ed è espressa in relazione alle sue velocità e dalla posizione nel vettore. Questo ci ha permesso di ottimizzare lo spazio di indirizzamento della distribuzione cosa che non sarebbe potuta accadere se si fosse optato per l'utilizzo degli Array in quanto quella scelta particolare sarebbe andato a incidere sull'eccessivo utilizzo dello stack.

Per poter inizializzare lo spazio di lavoro creato bisogna dunque specificare i valori della distribuzione inserendo in input: un vector con le velocità che verrà inizializzato come detto sopra, un vector per specificare le celle di stato solido e uno per le masse delle varie parcelle. Il vettore con le informazioni sullo stato è fondamentale in quanto il metodo Lattice Boltzmann e tutta fluidodinamica si comporta diversamente al variare dello stato fisico della cella. Incontrando una cella solida ad esempio si potrebbe infatti avere un urto che porterebbe ad una variazione del verso della velocità. Dal vettore degli stati passato in input durante l'inizializzazione, si costruisce anche il vettore

$$storage\_type\ \boldsymbol{boundaries\_points}$$

che, mediante l'utilizzo di valori booleani, dà conoscenza delle celle di bordo del nostro spazio di riferimento etichettando con valore uno (*true*) la cella che fa parte del bordo. In **Figura 4.3** è mostrato un esempio in cui, sempre per un caso a due dimensioni, vengono inizializzate anche le celle di bordo. Si nota come le particelle siano inserite nello Storage a partire da quella con Y maggiore e X minore fino ad arrivare a quella con Y minore e X maggiore.

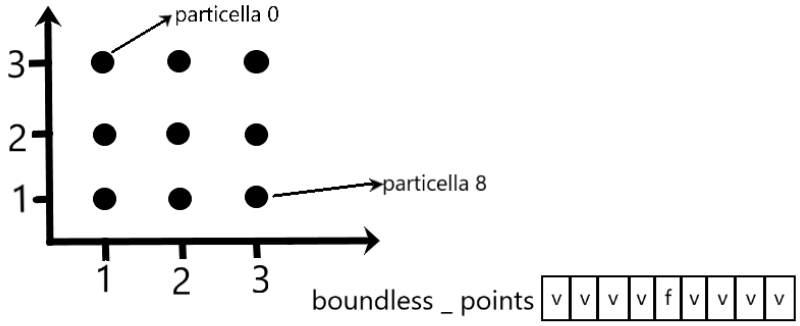

Figura 4.3: vettore Storage per modello D2Q9

Questa inizializzazione del vettore boundaries_points è operata mediante l'utilizzo di due metodi: *check_solid_bound* e *check_boundaries_point*.

check_solid_bound prendendo in input un intero, ovvero l'indice dello storage da analizzare, restituisce in output un booleano true nel caso in cui la cella con indice in ingresso è bordo di una cella solida.

check_boundaries_point utilizzato per restituire il bordo perimetrale del reticolo discreto.

Di seguito, in Figura 4.4 è riportato un esempio concreto di inizializzazione di un reticolo discreto in cui sono evidenziate: le celle di stato fisico solido, le celle di fluido inserito e quelle di bordo. Infine viene inizializzato anche un vettore *storage_type axes* di interi, di supporto per l'utilizzo di metodi che da un indice ritornano le coordinate del punto e viceversa, che ha dimensione pari al numero di assi su cui si estende il reticolo discreto e i cui valori ne specificano la grandezza degli stessi.

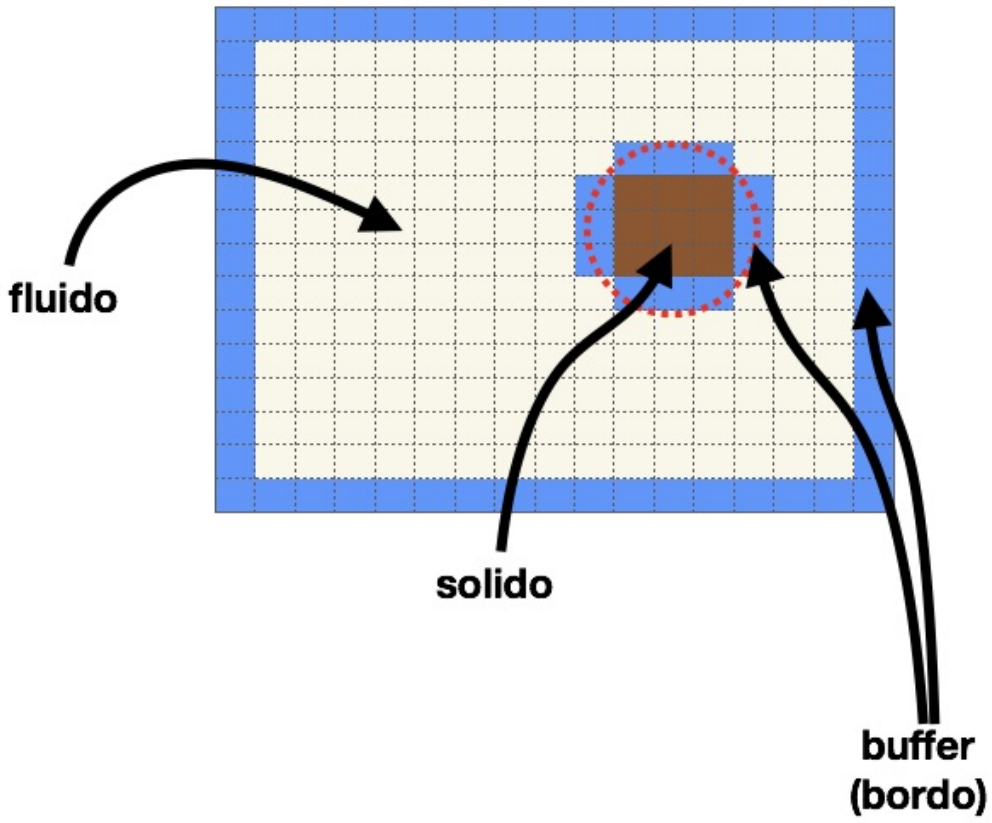

Figura 4.4: esempio completo di reticolo discreto

4.4 Iterazione

Una volta inizializzato il reticolo discreto l'utente può eseguire le iterazioni. Queste come visto sono strutturate in due step, o fasi compiute in intervalli di tempo separati: Stream e Collision, che adesso saranno mostrate più in dettaglio.

4.4.1 Stream

$$f_i(\vec{x} + \vec{e}_i\delta_t, t + \delta_t) = f_i^t(\vec{x}, t + \delta_t)$$

Nella prima fase si avrà lo **Stream** in cui ci sarà la propagazione delle particelle, che si muoveranno nelle varie celle del reticolo discreto secondo le distribuzioni di velocità definite in precedenza, come mostrato in Figura 4.5 per una distribuzione D2Q9

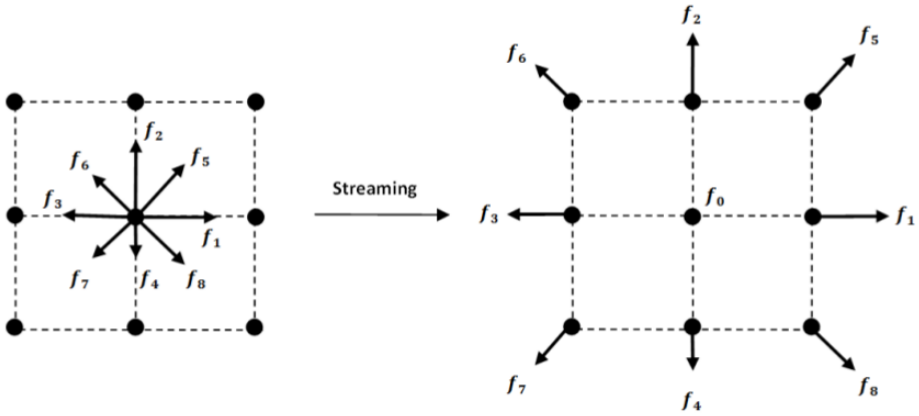

Figura 4.5: Illustrazione fase di stream in un modello D2Q9

Ricordando la disposizione delle velocità e delle particelle del vettore Storage, per poter realizzare l'effetto mostrato in figura, si passa quindi a una scansionare ogni particella e per ognuna di esse le proprie velocità, vedendo quali di esse si muoveranno ed in quale direzione. Dato che le particelle saranno anche collegate alla loro massa ed avendo visto che utilizzando LBM si conserva la quantità di moto, ad ogni iterazione la massa totale sarà conservata ma la distribuzione risulterà variata.

4.4.2 Condizioni di bordo e di angolo

Come solo accennato in precedenza parlando di scelte e metodologie applicate al progetto svolto, nel Capitolo 2, si sono lasciate le condizioni di bordo a scelta dell'utente. Sono state infatti implementate semplici condizioni di default, come necessità per poter passare alla seconda fase, che simulassero **l'uscita della particella** da reticolo andando quindi a sostituire quella particolare velocità con un valore nullo come mostrato in Figura 4.6. L'utilizzatore ha la possibilità di sostituire questa condizione riscrivendo il metodo *Boundaries_ conditions* e implementando quella più consona al proprio modello. Esistono vari tipi di condizioni di bordo tra le quali spiccano:inflow, outflow, no-slip, free-slip e la bounce-back. È riportato un esempio di condizioni al contorno testate in precedenza in cui il reticolo possiede un bordo fisso e solido che a contatto le particelle mosse nella fase di Stream provocherà loro un urto e un conseguente cambio della direzione della velocità (**bounce-back**)come mostrato in Figura 4.7. Inoltre, come con

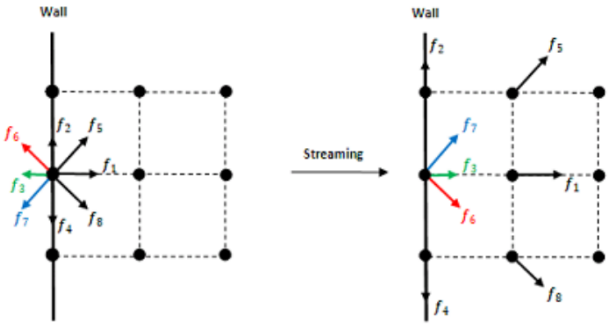

Figura 4.6: Illustrazione delle condizioni di bordo di default

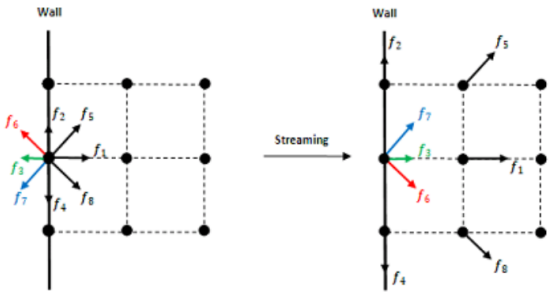

Figura 4.7: Illustrazione delle condizioni di bordo solido

le condizioni da attuare ai bordi del reticolo discreto, si è lasciata all'utente la possibilità di avere delle condizioni sugli angoli della griglia diverse da quelle di bordo, creando un metodo *"corners_ condition"* in cui sono state inserite come

condizioni di default le stesse implementate per quelle di bordo.

4.4.3 Collisione

$$f_i^t(\vec{x}, t + \delta_t) = f_i(\vec{x}, t) + \frac{1}{\tau_f}(f_i^{eq} - f_i)$$

Analizzata la fase di Stream e visti alcuni esempi di condizioni al contorno si passa ora al secondo step dell'iterazione andando quindi ad analizzare la fase di **Collisione** in cui, nello stesso intervallo di tempo, ogni particella si sposterà sul nodo vicino, a seconda della propria direzione andando a concludere l'iterazione con il ricalcolo delle distribuzioni. Durante questa fase viene inoltre ricalcolata la funzione di distribuzione aggiornata in modo tale che l'algoritmo possa procedere con un'altra iterazione e quindi con una nuova alternanza delle due fasi, conservando localmente massa e momento. Si è notato che in questo step le collisioni dipendono da quanto le distribuzioni si discostano da quelle di equilibrio, indicando un rapporto stretto tra le due fasi. Considerata dunque la vicinanza tra le due fasi, sia a livello temporale che di relazione si è dunque utilizzata la fase di collisione quasi in concomitanza con la fase di Stream.

Di seguito, in Figura 4.8 è mostrato un esempio di iterazione completa, con condizioni al contorno che comprendono la risposta al bordo solido denominato "wall". L'iterazione di esempio si compone di quattro fasi: pre-stream; post-stream; collision and reversing; bounce-back.

4.5 Caricare solidi

Per poter essere completamente aperta a qualunque tipo di utilizzo l'utente utilizzatore ha la possibilità di inserire, all'interno del reticolo discreto, forme e

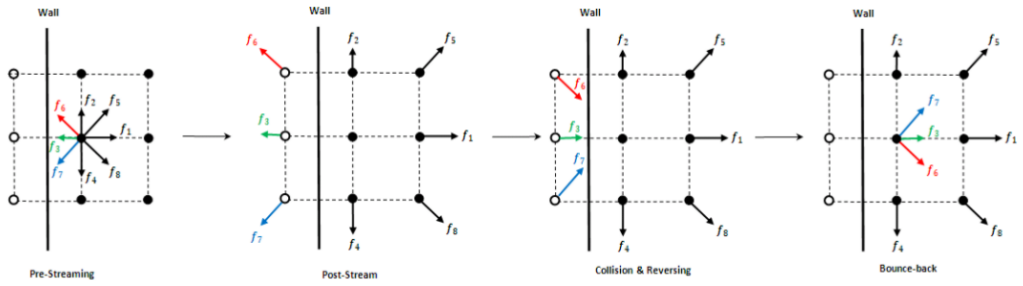

Figura 4.8: Illustrazione iterazione completa

figure solide. Per dare questa possibilità è stata implementata la funzione *void* ***insert_ stl****(string fname, vector<triangle> &v* con la quale è possibile leggere file con estensione .stl.

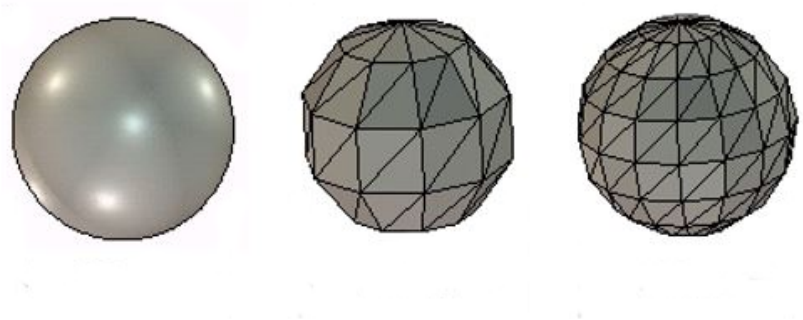

Figura 4.9: Illustrazione solido costruito con file .stl

STL (Acronimo di "Standard Triangulation Language") è un formato di file, binario o ASCII, nato per i software di stereolitografia CAD. È utilizzato nella prototipazione rapida (rapid prototyping) attraverso software CAD. Un file .stl rappresenta un solido la cui superficie è stata discretizzata in triangoli. Esso consiste delle coordinate X, Y e Z ripetute per ciascuno dei tre vertici di ciascun

triangolo, con un vettore per descrivere l'orientazione della normale alla superficie. In Figura 4.9 è mostrato un esempio di un solido sferico costruito come file con estenzione .stl. Data la definizione di file con estenzione .stl per poter utilizzare la funzione *insert_stl* è stata creata una classe triangle a cui in input vengono assegnati tre valori, uno per ogni asse. Questi tre valori sono di tipo V3, classe con un costruttore sovraccarico che può avere in input un puntatore ad un array, con i valori sugli assi, oppure direttamente le coordinate X, Y, Z che specificheranno dove creare l'oggetto. Una volta caricato il file l'utente ha il compito di settare il vettore *state_points*, che durante l'inizializzazione è necessario anche per costruire il vettore **check_boundaries_point**.

4.6 Bitmap

È stata implementata la possibilità di creare immagini e salvarle direttamente nella memoria interna del calcolatore utilizzato. In particolare si è optato per l'utilizzo di immagini con estensione .bmp. Un file BMP ÃÍ un file bitmap, cioè un file di immagine grafica che immagazzina i pixel sotto forma di tabella di punti e che gestisce i colori sia in true color che attraverso una paletta indicizzata. La struttura di un file bitmap è la seguente: intestazione del file (in inglese file header); intestazione del bitmap (in inglese bitmap information header); paletta (opzionale); corpo dell'immagine. Le immagini bitmap, anche dette immagini **raster**, si contrappongono alle immagini vettoriali ed a tutta la grafica vettoriale, nella quale gli elementi vengono geometricamente ubicati nell'immagine mediante l'indicazione delle coordinate dei punti di applicazione, anzichè descrivendoli utilizzando una griglia di pixel. Un esempio che riporta la differenza tra le due tipologie di immagini è riportata in figura Figura 4.10.

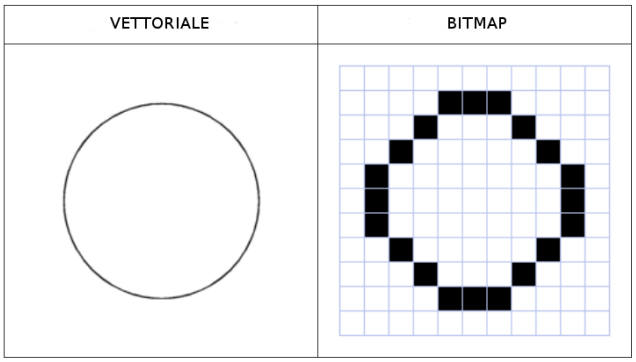

Figura 4.10: Illustrazione iterazione completa

Il formato file BMP è in grado di memorizzare immagini digitali bidimensionali sia monocromatiche che a colori, in varie profondità di colore e, opzionalmente, con: compressione dati, canali alfa e profili colore. Nel codice è stata importata una libreria esterna *bitmap_image* con cui è stato implementato il metodo *void bitmap_image()* il quale: crea un oggetto bitmap la cui grandezza tiene conto delle dimensioni del reticolo; ne disegna le distribuzioni delle velocità con una palette di colori che passa da quelli meno accesi per una densità di particelle e di velocità minori, fino ad arrivare a colori più accesi dove è presente un numero elevato di particelle e velocità; salva l'immagine creata e disegnata in memoria una volta terminato il programma.

L'utente utilizzatore ha anche la possibilità di analizzare, facendo molteplici iterazioni, il cambiamento nel pre e post per ogni iterazione attraverso un numero di immagini BMP quante ne sono state stampate durante l'intera esecuzione del processo.

4.7 Testing

Come detto nel paragrafo 3.1 lo sviluppo è stato accompagnato e guidato da test. In particolare è stato inserito il file *Catch.hpp* distribuito sotto la licenza software Boost.

Boost è un insieme di librerie per il linguaggio di programmazione C++ che fornisce supporto per attività e strutture quali algebra lineare, generazione di numeri pseudocasuali, multithreading, elaborazione di immagini, espressioni regolari e test di unità. Molte di queste librerie sono rilasciate sotto Boost Software License, licenza progettata per consentire a Boost di essere utilizzato con progetti software gratuiti e proprietari.

Lo sviluppo guidato dal testing dell'applicativo è una parte del ciclo di vita del software di fondamentale importanza, non solo per individuare le carenze di correttezza, completezza e affidabilità delle componenti software in corso di sviluppo ma anche per valutare se il comportamento del software, prima della distribuzione, rispetta i requisiti. Inoltre si parla di **sviluppo guidato** in quanto i test devono accompagnare tutte le fasi dello sviluppo i modo da vedere se dopo un determinato cambiamento sia venuta meno la stabilit‘a del codice. L'obiettivo dell'impiego di test è quello di mantenere malfunzionamenti del codice e bug, che hanno costi di debugging proporzionali al tempo di "creazione" e alla quantità di rige di codice, in zone a spiccata località di modo da essere facilmente identificabili ed eliminare la **regressione**.

Capitolo 5

Esempi di utilizzo e possibili sviluppi

Di seguito saranno viste le varie possibilitá prese in considerazione e non ancora implementate. L'idea di base è quindi quella di estendere non le funzionalità ma le tipologie di implementazioni, tramite l'utilizzo di strutture dati diverse, in modo da avere come obiettivo finale quello di poter dare all'utente utilizzatore la possibilità di scegliere l'implementazione desiderata.

5.1 Quadtree e Octree

5.1.1 Definizione delle strutture

Un **Quadtree**, anche detto albero quadramentale, è una struttura dati ad albero in cui ogni nodo interno ha esattamente quattro figli, come mostrato in Figura 5.1.Questa implementazione in particolare è molto utilizzata in computer grafica poichè in uno spazio a due dimensioni, presa una determinata area, un nodo ne rappresenta un riquadro di delimitazione che copre una parte dello

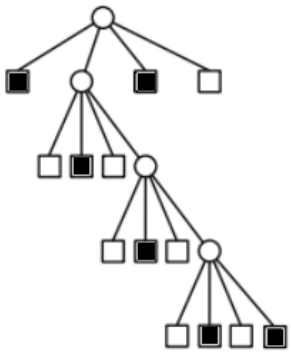

Figura 5.1: Rappresentazione struttura Quadtree

spazio che viene indicizzato. La radice dell'albero ovvero quel nodo privo di arco entrante può esser visto come la rappresentazione dell'intera area. Identificando uno spazio a due dimensioni con l'albero quadramentale di esempio mostrato in figura Figura 5.1 si otterrà la suddivisione mostrata in Figura 5.2. Il fondamento alla base dell'assegnazione dell'intera area alla radice dell'albero trova spiegazione nel fatto che un albero senza radice non può essere acceduto risultando quindi inutilizzabile. Quando si lavora invece su spazi a tre dimensioni vengono utilizzati gli **Octree**, anche chiamati alberi ottali, su cui valgono gli stessi principi detti prima per gli alberi quadramentali con la differenza che ogni nodo ha esattamente otto figli, dividendo così lo spazio sui tre assi (x, y, z). Implementando anche questa struttura dati l'utilizzatore può dunque creare il reticolo discreto intero come la radice dell'albero (che sia Quadtree oppure Octree) andando poi a vedere ogni cella come una foglia dell'albero.

5.1.2 Teoremi stipulati sulle strutture

Sarà ora descritta la funzione $O(f(x))$ utilizzata per i successivi due teoremi. La notazione matematica O-grande è utilizzata per descrivere il comportamento

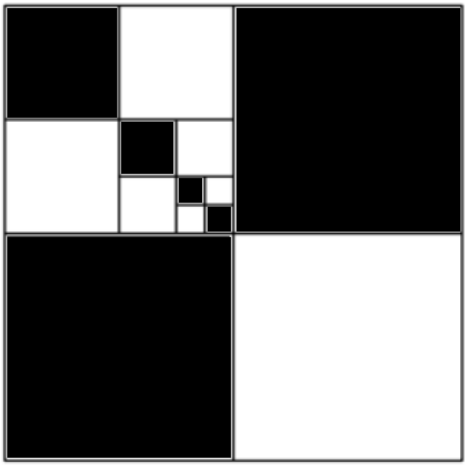

Figura 5.2: Divisione dello spazio mediante Quadtree

asintotico delle funzioni. Il suo obiettivo è quello di caratterizzare il comportamento di una funzione per argomenti elevati in modo semplice ma rigoroso, al fine di poter confrontare il comportamento di più funzioni fra loro. Più precisamente, è usata per descrivere un limite asintotico superiore per la magnitudine di una funzione rispetto ad un'altra, che solitamente ha una forma più semplice. Ha due aree principali di applicazione: in matematica, dove è solitamente usata per caratterizzare il resto del troncamento di una serie infinita, in particolare di una serie asintotica ed in informatica dove risulta utile all'analisi della complessità degli algoritmi. Importanti i teoremi stipulati sugli alberi quadramentali secondo cui: la profondità di un Quadtree per un set P di punti nel piano è al massimo uguale a

$$log(s/c) + 3/2$$

dove c è la distanza minima tra due punti qualsiasi in P e s è la lunghezza laterale del quadrato iniziale che contiene P. Inoltre un quadtree di profondità d che

memorizza un insieme di n punti ha un numero di nodi e un tempo di costruzione entrambi pari a

$$O((d+1)n)$$

Data poi la definizione di albero bilanciato come albero in cui due nodi vicini differiscono al massimo di uno in profondità questa può essere applicata per bilanciare l'albero dell'esempio in Figura 5.1 così come mostrato in Figura 5.3

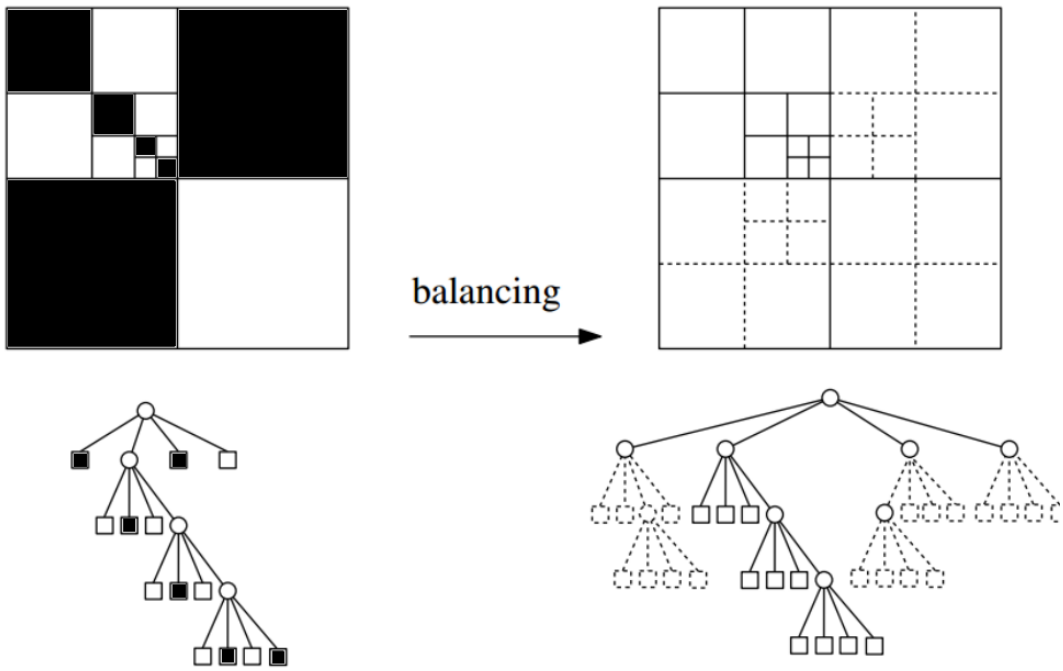

Figura 5.3: Bilanciamento di un Quadtree

5.1.3 Inserimento particelle

Dopo la creazione si passa all'inzializzazione e in questa fase si nota il punto di forza di questa struttura dati in quanto si possono assegnare con precisione le particelle nel punto dello spazio desiderato. Il rilevamento delle collisioni è una parte essenziale della computer grafica applicata alla fluidodinamica e rappresenta una delle operazioni più dispendiose. Per evitare queste operazioni, all'aumentare degli oggetti, per non farli collidere in un unico nodo, questo si dividerà per assegnare ad ogni particella la propria foglia come mostrato in **Figura 5.4**. Da questo si notano anche che le collisioni possibili sono molto ridotte in quanto una particella in uno specifico nodo non entrerà in collisione con un'altra appartenente ad un nodo di un'altro semipiano di profondià minore.

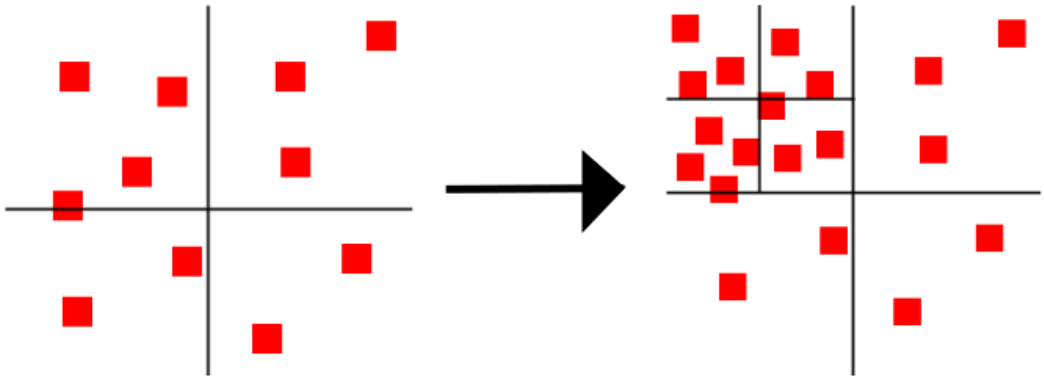

Figura 5.4: Divisione in relazione all'aumento dei punti

5.2 Griglie a infittimento e autoadattive

Il metodo Lattice Boltzmann utilizza come visto griglie cartesiane che nell'attuale lavoro svolto si potrebbero definire statiche. Implementando il reticolo con la struttura dati a quadtree della sezione precedente si risolverebbe questo problema potendo infatti **infittire** le zone di interesse per poter analizzare la posizione delle particelle o del solido inserito in maniera più accurata. Volendo infittire invece la griglia cartesiana di riferimento si andrebbe però ad influire negativamente sulle prestazioni ma una griglia più veloce a livello computazionale sarebbè per'o inaccurata. La soluzione ottima si ha nell'introduzione di una griglia a infittimento che aumenta l'accuratezza solo ed esclusivamente nelle zone di interessa come mostrato in Figura 5.5.

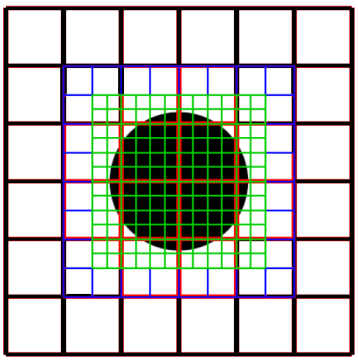

Figura 5.5: reticolo discreto con griglia a infittimento

Esistono due tipologie di griglie a infittimento: a priori ed autoadattive. Nelle seconde si nota che, oltre ad avere un infittimento nelle aree del reticolo discreto più soggette alla presenza di particelle, adattano automaticamente l'infittimento nelle zone più soggette ad avere densità maggiore.

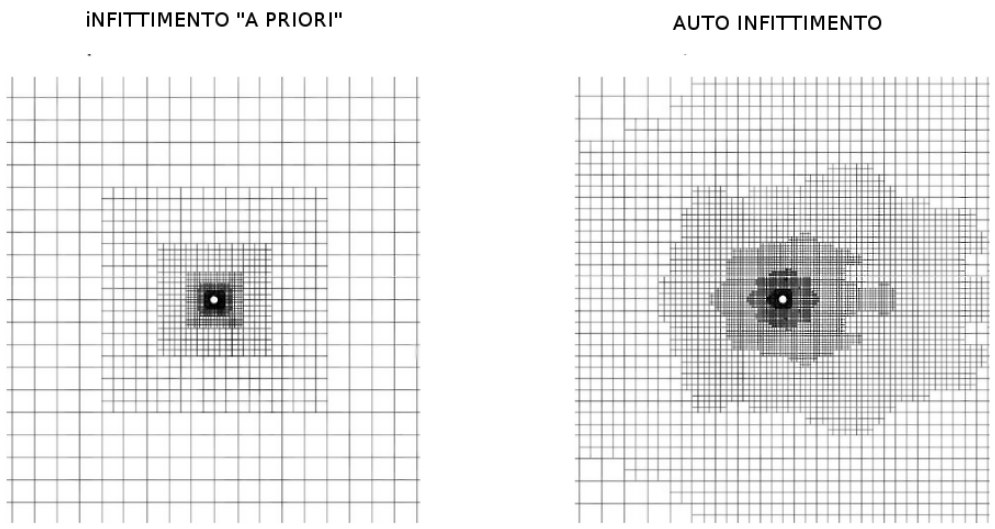

Figura 5.6: illustrazione griglia a infittimento a priori e autoadattive

5.3 Esempi di utilizzo

In questo paragrafo vedremo un esempio di utilizzo incentrato sul vettore di *storage*, sul suo stato pre iterazione e la sua variazione post iterazione. Come già detto infatti utilizzando l'algoritmo LBM si conserva, così come la quantità di moto, anche la massa ma, dopo l'alternanza delle fasi di stream e collision, risulterà cambiata la sua distribuzione e quella delle velocità. Per l'esempio che mostreremo è stato creato un reticolo discreto a due dimensioni in cui gran parte delle particelle si trovano nella parte in alto a sinistra della griglia. Per la rappresentazione verrà utilizzata la funzione descritta nel paragrafo 4.6 per salvare un BMP dello stato pre e post varie iterazioni. In **Figura 5.7** sono mostrate le im-

mabini bitmap memorizzate in memoria che identificano le situazioni antecedenti e successive all'iterazione.

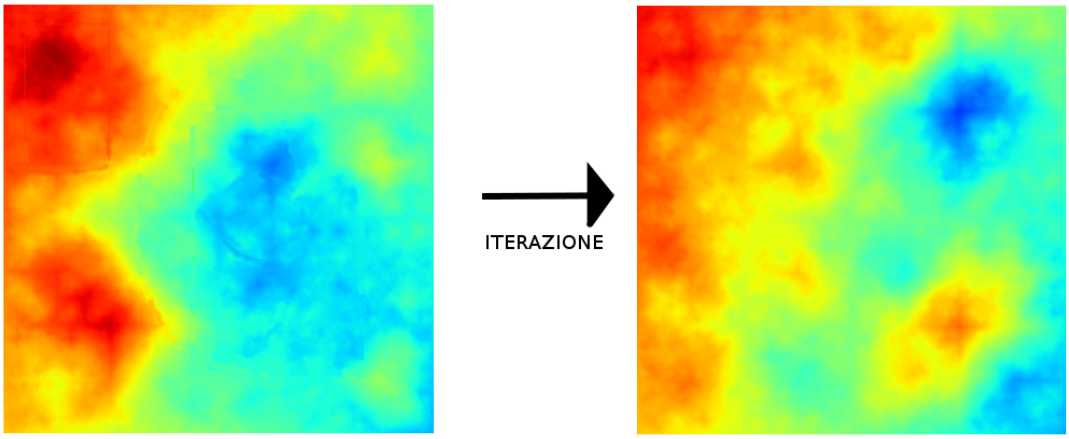

Figura 5.7: illustrazione immagini bmp pre e post iterazione

Conclusioni

Il metodo *Lattice Boltzmann* come visto è tuttora utilizzato in tutti gli ambiti inerenti la fluidodinamica proprio per l'algoritmo che, a contrario delle equazioni di Navier-Stokes viste nel capitolo 1, è facilmente realizzabile a livello di computazione e ciò ha spinto l'attività di tirocinio formativo a trovare ed analizzare altri aspetti, che potessero affiancare l'utilizzo di questo metodo, da implementare e realizzare.

All'inizio dell'attività di tirocinio una volta preso nota l'importanza nell'ambito della computer grafica e delle simulazioni dell'utilizzo del metodo Lattice Bolzmann sono stati quindi individuati due obiettivi chiave su cui porre un'attenzione particolare. Primo dei quali, non per importanza, è l'aumento della velocità di esecuzione che potesse comunque adattarsi ed essere utilizzato su qualunque tipologia di macchina. Il secondo si concentra invece sulla facilità di utilizzo da parte dell'utente finale, obiettivo messo in primo piano da molti sviluppatori di librerie.

Al termine del tirocinio formativo e con il risultato della progettazione della libreria sono stati raggiunti entrambi gli obiettivi che furono posti tre mesi prima. Il primo, la realizzazione di un software veloce è stato raggiunto mediante l'introduzione di due paradigmi di programmazione che puntano proprio alla velocità di esecuzione come visto nei paragrafi 2.2 e 2.3: la programmazione concorrente

e parallela, che fa uso di più thread di esecuzione per suddividere un problema in più sottoproblemi da svolgere in parallelo introducendo quindi il concetto di multi-tasking e la programmazione generica, che mediante l'utilizzo delle classi e dei metodi templetizzati o "modelli" risolti a tempo di compilazione anzichè a tempo di esecuzione migliora anche il riutilizzo del codice e il refactoring.

I risultati ottenuti in questa tesi rappresentano un primo passo esplorativo nella direzione dello studio della velocità computazionale e del confronto di essa a parità di prestazioni della macchina. La velocità computazione infatti tiene conto: delle strutture dati utilizzate, per gli storage e per il reticolo discreto e della quantità di celle, particelle e distribuzioni di velocità di cui è composta la griglia, ovvero dalla tipologia di reticolo scelto dall'utente.

Bibliografia

[Ale07] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied.* Addison-Wesley, 2007.

[Ber17] S. Berrino. *Il modello numerico Lattice "Boltzmann" per le equazioni Shallow Water.* PhD thesis, Università degli studi di Roma Tre, 2017.

[Fow02] Martin Fowler. *UML distilled. Guida rapida al linguaggio di modellazione standard.* Pearson; 4 edizione, 2002.

[KB01] Robert C. Martin e Martin Fowler Kent Beck. Manifesto for agile software development. 2001.

[KM13] Bill Hoffmann Ken Martin. *Intel Threading Building Blocks.* Kitware, 2013.

[Mar02] Rober C. Martin. *Agile Software Development: Principles, Patterns, and Practices.* Pearson; 1st edition, 2002.

[rei07] James reinders. *Intel Threading Building Blocks.* O'Reilly Media, 2007.

[Sta14] William Stallings. *Operating Systems: Internals and Design Principles.* Pearson College Div; 8 edizione, 2014.

[Str00] Bjarne Stroustrup. *C++ Linguaggio, libreria standard, principi di programmazione*. Pearson Addison-Wesley, 2000.