Alma Mater Studiorum · Università di Bologna

DIPARTIMENTO DI INFORMATICA — SCIENZA E INGEGNERIA
Corso di Laurea in Ingegneria e Scienze Informatiche

IMPLEMENTAZIONE PARALLELA DI UN ALGORITMO DI PROIEZIONE TOMOGRAFICA

Elaborato in:
High Performance Computing

Relatore:
Chiar.mo Prof.
Moreno Marzolla

Correlatori:
Chiar.ma Prof.ssa
Elena Loli Piccolomini

Presentata da:
Lorenzo Colletta

Sessione 03/10/2024
Anno Accademico 2023/2024

Indice

1 Introduzione **1**
- 1.1 Il modello matematico . . . 2
- 1.2 Risoluzione del modello matematico . . . 4
  - 1.2.1 La matrice M . . . 4
  - 1.2.2 Il vettore f . . . 4
- 1.3 La geometria . . . 5
  - 1.3.1 Coordinate cartesiane di un piano . . . 5
  - 1.3.2 Retta rappresentante un raggio . . . 6
  - 1.3.3 La sorgente . . . 7
  - 1.3.4 Il rilevatore . . . 8

2 Descrizione dell'implementazione **11**
- 2.1 Struttura del codice . . . 11
- 2.2 Parametri di inizializzazione dell'ambiente simulativo . . . 12
- 2.3 Flusso di esecuzione del proiettore . . . 13
- 2.4 Strutture dati . . . 15
- 2.5 Funzioni per il calcolo della proiezione . . . 16
  - 2.5.1 Calcolo coordinate cartesiane della sorgente . . . 16
  - 2.5.2 Calcolo coordinate di una unità del rilevatore . . . 16
  - 2.5.3 Calcolo dell'intersezione tra un piano e un raggio . . . 17
  - 2.5.4 Calcolo dell'assorbimento di un raggio . . . 19
  - 2.5.5 Calcolo della proiezione . . . 20
- 2.6 Funzioni per la generazione dell'input . . . 24
- 2.7 Descrizione dell'output . . . 27

3 Strategie di parallelizzazione **29**
- 3.1 OpenMP . . . 29
- 3.2 Costo computazionale e profiling . . . 30
- 3.3 Versione parallela . . . 31
  - 3.3.1 Una prima strategia di parallelizzazione . . . 31

3.3.2 La strategia di parallelizzazione applicata 32

4 Valutazione delle prestazioni della soluzione parallela 35
4.1 Strong-scaling efficency 36
4.2 Weak-scaling efficency 37

5 Conclusioni 39

Sommario

La Tomografia Computerizzata (TC) è una tecnica diagnostica che sfrutta le radiazioni ionizzanti (o raggi X) per ottenere immagini dettagliate di sezioni trasversali di un corpo studiato. Questo metodo si basa sulla raccolta di dati proiettivi da diverse angolazioni, che vengono poi elaborati per ricostruire immagini bidimensionali o tridimensionali degli oggetti studiati. La tomografia computerizzata trova applicazione in molti settori, tra cui la medicina, l'archeologia, la paleontologia e numerosi altri campi. Un problema tipico della tomografia computerizzata consiste nella ricostruzione delle immagini a partire dai dati raccolti dal dispositivo di scansione. Dal punto di vista matematico, questo rappresenta un problema inverso, cioè un problema in cui si cerca di determinare le cause (l'immagine originale) dagli effetti osservati (le proiezioni raccolte).

In questa sede viene invece affrontato il problema diretto: conoscendo le proprietà del corpo studiato si ottengono le immagini di proiezione. Scopo di questa tesi è:

- Implementare in linguaggio C un algoritmo che risolva il problema diretto della tomografia computerizzata.
- Implementare una versione parallela dell'algoritmo utilizzando OpenMP, un'interfaccia di programmazione per il parallelismo su piattaforma a memoria condivisa.
- Studiare e analizzare le prestazioni della versione parallela dell'algoritmo al fine di valutarne la scalabilità e l'efficienza nel migliorare le prestazioni complessive del codice seriale.

La struttura del documento è organizzata come segue:

- Per prima cosa vengono introdotte le nozioni matematiche alla base della soluzione del problema diretto.
- Viene discussa l'implementazione seriale con annesso codice.
- Saranno esplorate le diverse strategie di parallelismo considerate con le relative problematiche, con conseguente descrizione della strategia effettivamente implementata.
- Verranno analizzate le prestazioni della versione parallela dell'algoritmo.

Capitolo 1

Introduzione

Una immagine di proiezione rappresenta visivamente i dati di attenuazione dei raggi X che attraversano un corpo oggetto di studio. I colori rappresentano graficamente i vari livelli di attenuazione: un raggio che attraversa una regione più densa del corpo subisce una maggiore attenuazione rispetto a un raggio che attraversa una regione meno densa.

Durante una scansione tomografica, il dispositivo acquisisce numerose immagini di proiezione, ciascuna a una diversa angolazione lungo una traiettoria circolare intorno al corpo. La quantità totale di radiazioni a cui viene esposto il corpo dipende dal numero di immagini di proiezioni acquisite. Per ridurre l'esposizione complessiva alle radiazioni, è possibile adottare diverse strategie. Una di queste consiste nella riduzione del numero di scansioni effettuate lungo la traiettoria circolare. In alternativa, è possibile limitare la scansione a una porzione dell'intera traiettoria circolare. In entrambi i casi diminuire il numero di scansioni implica ottenere dati tomografici incompleti.

Nei paragrafi seguenti, verrà innanzitutto presentato il modello matematico alla base del calcolo delle proiezioni, seguito da una descrizione del problema da un punto di vista geometrico.

1.1 Il modello matematico

La **legge di attenuazione di Lambert-Beer** descrive la riduzione dell'intensità di un raggio in funzione del coefficiente di attenuazione. In particolare, il coefficiente di attenuazione è una funzione continua definita all'interno del dominio spaziale di riferimento. Applicando la legge di Lambert-Beer, si può derivare la seguente relazione per il calcolo della proiezione del coefficiente di attenuazione lungo un segmento di lunghezza W [1]:

$$P_W\mu = -\ln(\frac{m}{m_0}) = \int_0^W \mu(w)\,dw \tag{1.1}$$

Dove:

- **m** è l'intensità del raggio dopo aver attraversato il materiale.
- $\mathbf{m_0}$ è l'intensità iniziale del raggio.
- $\boldsymbol{\mu(w)}$ è una funzione continua che descrive il **coefficiente di attenuazione** a un dato punto w lungo il percorso attraversato dal raggio.
- **W** è la lunghezza totale del percorso attraverso il materiale.

La **trasformata di Radon** in un modello continuo, di una porzione di oggetto descritta da una funzione $\mu(t)$, è data dall'insieme delle proiezioni acquisite lungo l'intera traiettoria circolare [1].

Nel caso discreto, il corpo è suddiviso in porzioni, ciascuna rappresentante un'area (nel caso bidimensionale) o un volume (nel caso tridimensionale, in seguito riferito anche con il nome di "**voxel**") in cui il corpo è omogeneo: nei punti interni di un singolo voxel il coefficiente di attenuazione assume un valore costante.

In figura (1.1) si mostra un'esempio di discretizzazione di un oggetto in un numero di voxel pari a $N = Nx \times Ny \times Nz = 2 \times 2 \times 2$, dove **Nx**, **Ny** e **Nz** sono il numero di voxel lungo gli assi x, y e z. In generale i voxel condividono tra loro le stesse dimensioni e presentano la forma di un parallelepipedo.

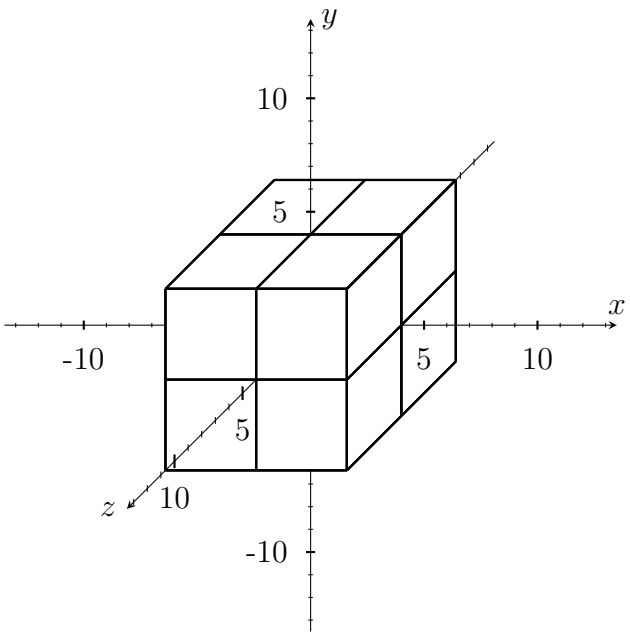

Figura 1.1: Esempio di discretizzazione di un oggetto con un numero di voxel di $N = Nx \times Ny \times Nz = 2 \times 2 \times 2$.

La forumla **(1.1)** nel caso discreto diventa:

$$g_i = -\ln(\frac{m}{m_0}) = \sum_{j=0}^{N} M_{i,j} f_j \quad \forall i \in 1, \ldots, n_p \tag{1.2}$$

Dove:

- $\boldsymbol{n_p}$ è il numero di raggi usati per il calcolo della proiezione.
- $\boldsymbol{N}$ è il numero di voxel che costituiscono il corpo.
- $\boldsymbol{g_i}$ è l'attenuazione subita dall'i-esimo raggio.
- $\boldsymbol{M}$ è una matrice che per ogni raggio definisce la lunghezza della porzione sottesa al volume di ciascun voxel: $|M| = N \times n_p$.
- $\boldsymbol{f}$ è una funzione discreta che esprime il coefficiente di attenuazione: f_j è il valore del coefficiente di attenuazione assunto nel volume interno al j-esimo voxel.

Applicando al caso discreto la trasformazione di Radon, si ottengono un numero $\mathbf{N_\theta}$ di proiezioni.

1.2 Risoluzione del modello matematico

Il modello matematico si limita a descrivere la composizione delle immagini di proiezione. Rimane il problema di trovare la matrice M e il vettore f coinvolti nel calcolo di ciascuna proiezione come definito nell'equazione (1.2).

1.2.1 La matrice M

Esistono numerosi algoritmi per calcolare efficientemente i valori della matrice M. In questa sede si implementa l'algoritmo di **Siddon** [2]. I punti chiave dell'algoritmo possono essere così riassunti:

- L'algoritmo risolve il problema del calcolo dell'equazione (1.2):

$$d = \sum_j M_j f_j$$

- I voxel sono considerati come volumi sottesi alle intersezioni di insiemi ortogonali di piani paralleli invece che come unità indipendenti.
- Si calcolano i punti di intersezione tra la retta (che rappresenta il raggio) e i piani che definiscono i voxel.
- Si calcola la lunghezza dei segmenti sottesi ai voxel come differenza tra punti di intersezione consecutivi.
- Per ciascun segmento trovato, si determina l'indice del voxel cui il segmento è sotteso. Ciascun elemento della matrice M indica la lunghezza del segmento sotteso dall'area del voxel corrispondente all'indice dell'elemento.

La matrice M presenta, per ogni raggio usato per il calcolo della proiezione, un numero di elementi pari al numero di voxel di cui è composta la discretizzazione del corpo.

1.2.2 Il vettore f

Il vettore f rappresenta il coefficiente di attenuazione assunto nella regione interna al volume sotteso da ciascun voxel. Presenta dunque un numero di elementi pari al numero di voxel in cui è suddiviso l'oggetto.

Il coefficiente di attenuazione è una proprietà del corpo soggetto di studio cui occorre conoscere al fine di ottenere le proiezioni. In questa sede si implementano tre diverse configurazioni del vettore dei coefficienti di attenuazione in modo che rappresentino rispettivamente:

- Un **cubo**: stabilita la lunghezza del lato del cubo che si intende rappresentare, gli elementi del vettore corrispondenti ai voxel che risiedono all'interno di una regione cubica sono posti rispettivamente a 1, altrimenti 0.
- Una **semi-sfera**: stabilito il raggio r della semisfera, un asse i verso cui è rivolto il lato troncato della sfera, i coefficienti dei voxel, posti ad una distanza inferiore a r dal centro e con componente lungo l'asse $i > 0$ nel caso venga stabilito il verso positivo, < 0 nel caso del verso negativo, sono posti a 1, 0 altrimenti.
- Un **Cubo con una cavità sferica**: stabilito il lato del cubo, i valori dei coefficienti sono determinati come nel caso del cubo, ad eccezione dei voxel, la cui distanza da un punto stabilito all'interno dell'oggetto sia inferiore ad un raggio r stabilito, i cui coefficienti sono posti a 0.

1.3 La geometria

Questa sezione approfondisce i calcoli geometrici necessari per l'implementazione dell'algoritmo di Siddon e per ottenere le proiezioni secondo il modello matematico definito. Verranno trattati i concetti chiave relativi alle coordinate cartesiane di un piano, alla rappresentazione di un raggio, e alla determinazione delle posizioni della sorgente e del rilevatore.

1.3.1 Coordinate cartesiane di un piano

L'equazione cartesiana di un piano nello spazio tridimensionale è un'equazione di primo grado in tre incognite della forma:

$$ax + by + cz + d = 0$$

dove a,b,c e d sono numeri reali.

Nell'algoritmo di Siddon, i piani considerati sono quelli che delimitano i volumi sottesi dai voxel, che derivano dalla discretizzazione del volume dell'oggetto di studio. Questi voxel, che hanno dimensioni uniformi e forma di parallelepipedi, rappresentano una suddivisione regolare del volume totale. Di conseguenza, i piani coinvolti sono paralleli ai piani tracciati dagli assi cartesiani, le loro equazioni sono dunque della forma:

$$ax + d = 0 \ \textit{se parallelo al piano } yz$$
$$by + d = 0 \ \textit{se parallelo al piano } xz$$
$$cz + d = 0 \ \textit{se parallelo al piano } xy$$

Dato che le distanze tra i piani paralleli sono regolari, data l'equazione di un piano è possibile determinare l'equazione degli altri traslandolo più volte. A partire dal piano con coordinata cartesiana minore, si determina l'equazione di ciascun piano nel seguente modo:

$$\begin{aligned} X_{plane}(i) &= X_{plane}(0) + i \times d_x \quad \forall i \in 1, \ldots, N_x \\ Y_{plane}(j) &= Y_{plane}(0) + j \times d_y \quad \forall j \in 1, \ldots, N_y \\ Z_{plane}(k) &= Z_{plane}(0) + k \times d_z \quad \forall k \in 1, \ldots, N_z \end{aligned} \tag{1.3}$$

dove $\mathbf{N_x}, \mathbf{N_y}, \mathbf{N_z}$ sono il numero di voxel lungo gli assi x,y,z, mentre $\mathbf{d_x}$,$\mathbf{d_y}$ e $\mathbf{d_z}$ sono a loro volta le dimensioni dei voxel lungo i tre assi cartesiani.

1.3.2 Retta rappresentante un raggio

Un raggio può essere rappresentato mediante l'equazione parametrica di una retta passante per due punti (X_1, Y_1, Z_1) e (X_2, Y_2, Z_2):

$$\begin{aligned} X(a) &= X_1 + a(X_2 - X_1) \\ Y(a) &= Y_1 + a(Y_2 - Y_1) \\ Z(a) &= Z_1 + a(Z_2 - Z_1) \end{aligned}$$

In questo modo:

- per definire un raggio basta conoscere il punto "sorgente" ed il punto sul "rilevatore" dove avverrebbe la misurazione.
- è possibile rappresentare un punto appartenente alla retta del raggio attraverso un unico parametro a, in particolare: $a = 0$ coincide con il punto X_1, $a = 1$ coincide con il punto X_2, per valori di a compresi tra [0,1] si hanno i punti appartenenti al segmento di cui i due punti sono gli estremi.

Il punto di intersezione di un raggio ed un piano lo si ottiene risolvendo in a il sistema lineare dato dall'equazione del piano e una componente della retta come di seguito:

$$\begin{cases} X = X_1 + a(X_2 - X_1) \\ X = X_{planes}(0) + i \times d_x \end{cases}$$

$$\begin{cases} Y = Y_1 + a(Y_2 - Y_1) \\ Y = Y_{planes}(0) + i \times d_y \end{cases} \tag{1.4}$$

$$\begin{cases} Z = Z_1 + a(Z_2 - Z_1) \\ Z = Z_{planes}(0) + i \times d_z \end{cases}$$

dove sono stati indicati i sistemi utilizzati nel caso in cui il piano sia parallelo al piano yz (primo sistema), al piano xz (secondo sistema) o al piano xy (terzo sistema).

1.3.3 La sorgente

La sorgente è un punto che indica la posizione a partire dalla quale sono diretti i raggi. Come già affrontato, le posizioni sono distribuite lungo una traiettoria circolare (o una sezione di essa) intorno all'oggetto e da ciascuna posizione sono diretti un determinato numero di raggi. Rimane il problema di determinare ciascuna posizione.

Conoscendo l'estensione angolare θ della sezione della traiettoria considerata e la distanza angolare α tra ciascuna posizione, rimane semplice l'uso delle coordinate polari: una posizione è individuata dalla distanza dall'origine e dalla distanza angolare dall'asse polare. In questa sede viene considerato un sistema di coordinate polari il cui polo coincide con il centro dell'oggetto di studio, mentre si adotta l'asse y come asse polare. La distanza angolare è positiva in senso orario.

In questo sistema di riferimento, la posizione della sorgente può essere individuata nel seguente modo:

$$(d_s, \frac{-\theta}{2} + \alpha i) \quad \forall i = 0, \dots, N_\theta - 1$$

dove $\boldsymbol{d_s}$ è la distanza dall'origine, $\boldsymbol{i}$ rappresenta l'indice della posizione, con 0 indice dell'angolo iniziale pari a $\frac{-\theta}{2}$, $\boldsymbol{N_\theta}$ è il numero di posizioni dato dal rapporto:

$$N_\theta = \frac{\theta}{\alpha} + 1$$

Le coordinate cartesiane si ottengono a partire dalle coordinate polari nel seguente modo:

$$\begin{cases} X_i = \cos(\frac{\pi}{2} + (\frac{\theta}{2} - \delta i))d_s = \sin(\frac{-\theta}{2} + \delta i)d_s \\ Y_i = \sin(\frac{\pi}{2} + (\frac{\theta}{2} - \delta i))d_s = \cos(\frac{-\theta}{2} + \delta i)d_s \qquad per \quad i = 0, \dots, N_\theta - 1 \\ Z_i = 0 \end{cases}$$

dove sono state usate le formule degli archi associati per seno e coseno per ricondursi dal sistema tradizionale per la misurazione di un angolo (senso antiorario a partire dal semi asse positivo delle x) al sistema precedentemente descritto (senso orario a partire dal semi asse positivo delle y). Z_i è posta a 0 in quanto, per costruzione, la traiettoria giace sul piano xy.

In figura **1.2** si riporta un esempio grafico dello schema appena descritto.

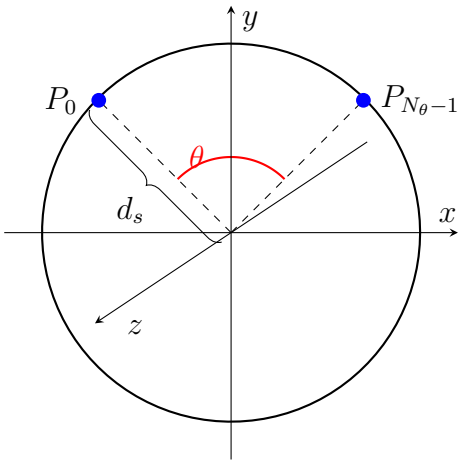

Figura 1.2: Rappresentazione grafica della traiettoria della sorgente. In questo caso si ha $N_\theta = 2, \theta = 90°$ e $\alpha = 90°$. La posizione P_1 è individuata da $\delta_1 = -45°$, la posizione di $P_{N_\theta-1}$ è individuata dall'angolo $\delta_{N_\theta-1} = 45°$.

1.3.4 Il rilevatore

Nel caso tridimensionale, il rilevatore può essere considerato una matrice di unità di misurazione posta ad una distanza d_r dal centro del corpo di studio. La figura (1.3) riporta una rappresentazione grafica del rilevatore.

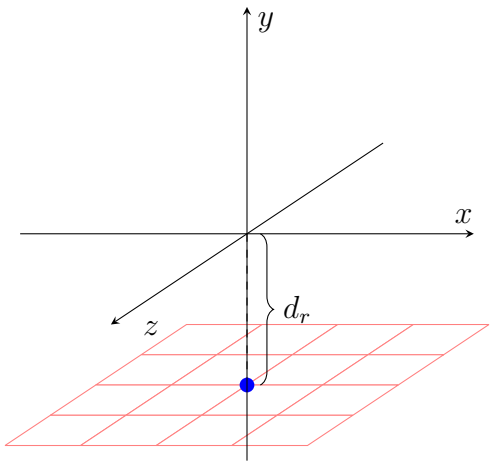

Figura 1.3: Rilevatore con 16 unità, ortogonale all'asse y, posto ad una distanza d_r dall'origine.

L'approccio usato per definire la matrice M (1.2.1) viene detto **ray-driven**: per ciascuna unità di misurazione del rilevatore si considera l'esistenza di un unico rag-

gio passante, diretto dalla sorgente al centro dell'unità. Secondo questo approccio, il numero $\mathbf{n_p}$ di raggi usati per il calcolo della proiezione coincide con il numero di unità di misurazione del rilevatore.

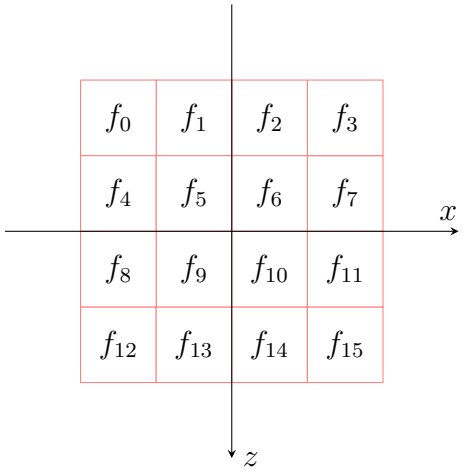

Figura 1.4: Vista ortogonale del rilevatore. Le unità di misurazione sono numerate in ordine crescente all'aumentare della componente x e all'aumentare della componente z del centro delle unità.

Il rilevatore è sempre ortogonale all'asse che congiunge la sorgente e il centro del corpo. Le coordinate cartesiane del centro di una unità j del rilevatore sono derivate nel seguente modo:

- ciascuna posizione può essere considerata come una trasformazione della posizione (X, Y) ricoperta nel caso in cui il rilevatore sia ortogonale all'asse y (come in figura **(1.3)**);
- siano (r, δ) le coordinate polari del punto, considerando il semiasse negativo di y come asse polare, le coordinate a seguito della trasformazione saranno:

  $$(r, \delta - \lambda)$$

  dove λ è l'angolo cui il punto ruota.
- le coordinate della posizione trasformata sono date da:

  $$\begin{cases} X' = \cos(\frac{3\pi}{2} - \delta + \lambda)r = \sin(\delta - \lambda)r \\ Y' = \sin(\frac{3\pi}{2} - \delta + \lambda)r = \sin(\delta - \lambda)r \end{cases}$$

utilizzando le formule di somma degli angoli per seno e coseno si ottiene:

$$\begin{cases} X' = \sin(\delta)\cos(\lambda)r - \cos(\delta)\sin(\lambda)r \\ Y' = \cos(\delta)\cos(\lambda)r + \sin(\delta)\sin(\lambda)r \end{cases}$$

dato che:

$$r = \frac{X}{\sin(\delta)} \quad r = \frac{Y}{\cos(\delta)}$$

allora sostituendo r e sfruttando le formule degli archi associati per seno e coseno si ottiene:

$$\begin{cases} X' = \cos(-\lambda)X + \sin(-\lambda)Y \\ Y' = \cos(-\lambda)Y - \sin(-\lambda)X \end{cases} \tag{1.5}$$

- analogamente a quanto avviene con la sorgente, la posizione di una unità del rilevatore varia, ad intervalli regolari, di un angolo α su di una sezione di ampiezza angolare pari a θ. Le posizioni di una unità j del rilevatore possono dunque essere ottenute:

$$\begin{cases} X_j^i = \cos(-(\theta/2 - \delta i))X + \sin(-(\theta/2 - \delta i))Y \\ Y_j^i = \cos(-(\theta/2 - \delta i))Y - \sin(-(\theta/2 - \delta i))X \end{cases} \tag{1.6}$$

  $\forall i = 0, \ldots, N_\theta - 1$

Capitolo 2

Descrizione dell'implementazione

La soluzione proposta si compone di due principali programmi: un programma che data da una configurazione dei coefficienti di attenuazione di un oggetto ne produce la proiezione, e un secondo programma che genera la configurazione dei coefficienti di attenuazione di un oggetto.

2.1 Struttura del codice

In una fase iniziale, era prevista una soluzione basata su un unico programma, in cui la generazione dei coefficienti di attenuazione e il calcolo della proiezione venivano eseguiti sequenzialmente, con l'input per la proiezione generato in tempo reale. Tra i principali vantaggi di questo approccio vi erano la riduzione del tempo necessario per l'acquisizione dell'input e l'assenza della necessità di creare un nuovo formato di file per la memorizzazione dell'input. Tuttavia, per migliorare la chiarezza del codice e favorire il riuso, l'applicativo è stato successivamente suddiviso in due programmi distinti, ciascuno organizzato in moduli:

- **common.h**: Definisce le strutture dati e i tipi di dato di base che rappresentano i concetti chiave della proiezione e dichiara variabili globali per i parametri di configurazione, che devono essere definite e inizializzate nei programmi che utilizzano i moduli successivi.
- **voxel.h**: contiene le dichiarazioni delle funzioni per la generazione di una matrice di voxel;
- **projection.h**: contiene le dichiarazioni delle funzioni per il calcolo della proiezione;
- **voxel.c**: implementa le funzioni dichiarate in voxel.h;

- **projection.c**: implementa le funzioni dichiarate in projection.h;
- **inputGeneration.c**: è un programma dedicato alla generazione e alla scrittura in un file di una configurazione del vettore di coefficienti di attenuazione;
- **projector.c**: è un programma che calcola la proiezione a partire da una configurazione di coefficienti di attenuazione;

2.2 Parametri di inizializzazione dell'ambiente simulativo

Nel capitolo precedente, è stato esaminato in dettaglio il calcolo della proiezione da una prospettiva rigorosamente matematica. Sono state introdotte diverse formule e notazioni essenziali per comprendere e implementare la soluzione proposta. In questa fase, è opportuno fare un riepilogo delle notazioni utilizzate finora e aggiungere alcune ulteriori specifiche necessarie per l'implementazione proposta.

Le notazioni già introdotte sono:

- $\mathbf{d_s}$: distanza della sorgente dal centro del corpo oggetto di studio. Coincide con il raggio della traiettoria.
- $\mathbf{d_r}$: distanza del rilevatore dal centro del corpo oggetto di studio.
- $\boldsymbol{\theta}$: ampiezza angolare della sezione di traiettoria lungo la quale la sorgente è posizionata a intervalli regolari.
- $\boldsymbol{\alpha}$: distanza angolare tra posizioni successive della sorgente. Per costruzione, essa equivale anche alla distanza angolare tra posizioni successive del rilevatore.

In aggiunta a queste notazioni, alcune ulteriori specifiche fondamentali sono:

- **lato del rilevatore**: è il lato della superficie quadrata del rilevatore;
- **lato di un'unità del rilevatore**: è il lato della superficie quadrata di una singola unità del rilevatore;
- **lato dell'oggetto**: è il lato del volume cubico che forma l'oggetto;
- **dimensioni dei voxel lungo ciascun asse**: i voxel, di cui è campionato l'oggetto, hanno la forma di un parallelepipedo di cui occorre conoscere le dimensioni lungo ciascun asse;

- **numero di voxel lungo ciascun asse**: questa è una specifica derivata. Essa è data, per ciascun asse, dal rapporto tra la lunghezza del lato dell'oggetto e la lunghezza del voxel lungo una dimensione;
- **numero di piani lungo ciascun asse**: questa specifica, anch'essa derivata, indica il numero di piani per ogni insieme di piani paralleli;

La soluzione proposta basa il calcolo della proiezione sui valori dei parametri appena elencati.

2.3 Flusso di esecuzione del proiettore

Il flusso di esecuzione dell'algoritmo di proiezione segue fedelmente il modello matematico descritto nel primo capitolo:

- il vettore f dei coefficienti di attenuazione costituisce l'input del programma;
- in primo luogo, viene calcolata la matrice M delle lunghezze dei segmenti del raggio che attraversano i voxel:
  - vengono determinati i due punti che definiscono la retta che rappresenta il raggio, ovvero il punto sorgente e il centro di una unità del rilevatore;
  - si verifica l'intersezione del raggio con l'oggetto calcolando il primo e l'ultimo punto di intersezione con i piani che delimitano l'oggetto, ottenendo due punti lungo la retta che definiscono l'intervallo in cui si trovano le intersezioni con ogni voxel;
  - a partire dall'intervallo dei valori parametrici, si determina per ciascun insieme di piani paralleli l'intervallo degli indici dei soli piani coinvolti nell'intersezione tra il raggio e l'oggetto;
  - si calcolano i valori parametrici dell'intersezione tra il raggio e i piani selezionati;
  - si misura la lunghezza dei segmenti compresi tra ciascuna coppia di intersezioni;
  - per ogni segmento si calcola l'indice del voxel che lo sottende;
- si calcola l'assorbimento del raggio selezionato;
- si ripete per ogni raggio condotto da ciascuna posizione della sorgente a ciascuna unità del rilevatore;

In figura (2.1) viene riportato il diagramma di flusso dell'algoritmo del calcolo della proiezione.

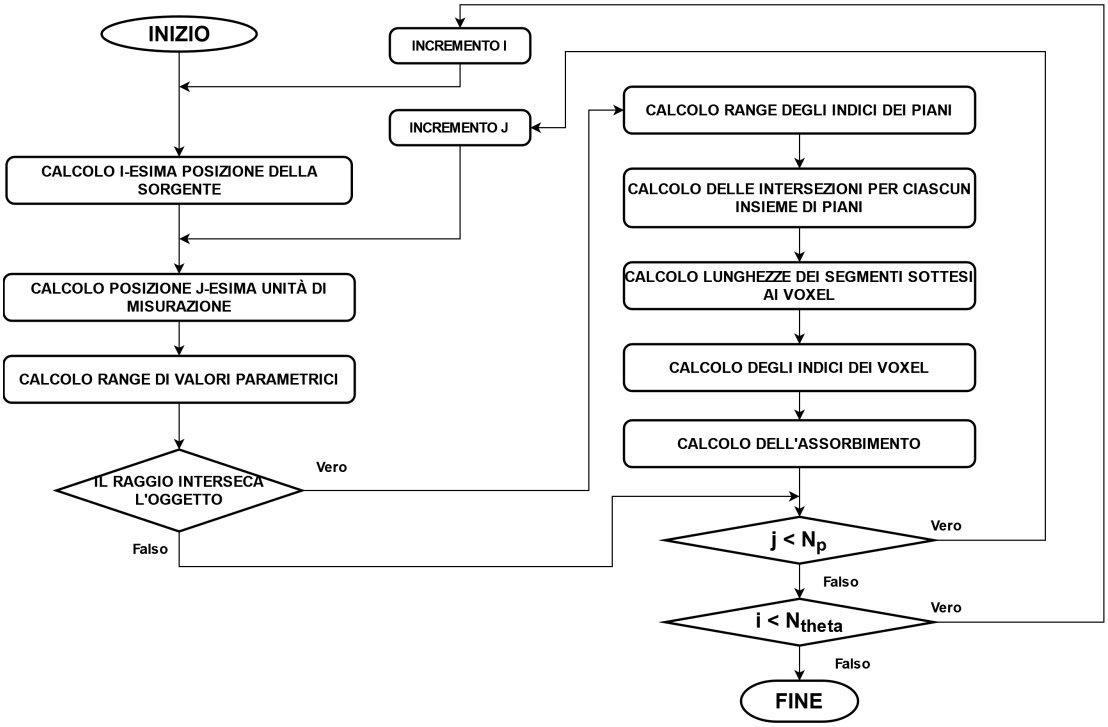

Figura 2.1: diagramma di flusso dell'algoritmo del calcolo delle proiezioni.

2.4 Strutture dati

Nel contesto dell'algoritmo di proiezione, sono state utilizzate tre principali strutture dati:

- **Axis**: questa enumerazione definisce gli assi di un sistema cartesiano tridimensionale. È utilizzata per identificare e distinguere le direzioni lungo gli assi nel sistema di coordinate, facilitando la manipolazione e l'accesso ai dati lungo ciascun asse.

```
typedef enum{
    X,
    Y,
    Z
} Axis;
```

Listato 2.1: Enumerazione Axis.

- **Point**: Questa struttura modella un punto nello spazio cartesiano. La struttura è costituita di tre membri, che rappresentano le coordinate del punto lungo gli assi X, Y e Z. La struttura Point è utilizzata per memorizzare e gestire le posizioni dei punti di interesse, come il punto sorgente e il centro delle unità del rilevatore.

```
typedef struct{
    double x;
    double y;
    double z;
} Point;
```

Listato 2.2: Struttura dati Point.

- **Ranges**: Questa struttura rappresenta un intervallo di indici dei piani da considerare per il calcolo delle intersezioni. È composta da due membri interi, `minIndx` e `maxIndx`, che definiscono rispettivamente l'indice minimo e massimo dei piani coinvolti nell'intersezione con il raggio.

```
typedef struct{
    int minIndx;
    int maxIndx;
} Ranges;
```

Listato 2.3: Struttura dati Ranges.

2.5 Funzioni per il calcolo della proiezione

2.5.1 Calcolo coordinate cartesiane della sorgente

La trasposizione dei calcoli matematici descritti nel paragrafo (**1.3.3**) è diretta: considerati due array, denominati rispettivamente `sineTable` e `cosineTable`, che contengono il seno e il coseno dell'angolo corrispondente all'indice `index`, e tenendo presente che la variabile `gl_distanceObjectSource` rappresenta la distanza della sorgente dall'oggetto, la funzione `getSource` restituisce le coordinate cartesiane della posizione della sorgente, come illustrato nel listato (**2.4**).

```
1  Point getSource(int index)
2  {
3      Point source;
4
5      source.z = 0;
6      source.x = sineTable[index] * gl_distanceObjectSource;
7      source.y = cosineTable[index] * gl_distanceObjectSource;
8
9      return source;
10 }
```

Listato 2.4: Funzione del calcolo della posizione della sorgente.

2.5.2 Calcolo coordinate di una unità del rilevatore

Anche per la funzione `getPixel`, mostrata nel listato (**2.5**), i calcoli effettuati si basano sulle formule descritte nel paragrafo (**1.3.4**). Le variabili `r` e `c` rappresentano rispettivamente la riga e la colonna dell'unità del rilevatore nella matrice. Le variabili globali `gl_distanceObjectDetector` e `gl_pixelDim` indicano rispettivamente la distanza del rilevatore dall'oggetto e la lunghezza del lato di un pixel. Anche in questo caso `sineTable` e `cosineTable` sono due array che contengono il seno e il coseno dell'angolo corrispondente all'indice `index`.

```
Point getPixel(int r, int c, int index)
{
    Point pixel;
    const double sinAngle = sineTable[index];
    const double cosAngle = cosineTable[index];
    const double elementOffset =  gl_detectorSideLength / 2 -
       gl_pixelDim / 2;

    pixel.x = (-gl_distanceObjectDetector * sinAngle) + cosAngle *
        (-elementOffset + gl_pixelDim * c);
    pixel.y = (-gl_distanceObjectDetector) * cosAngle - sinAngle *
        (-elementOffset + gl_pixelDim * c);
    pixel.z = -elementOffset + gl_pixelDim * r;

    return pixel;
}
```

Listato 2.5: Funzione che calcola la posizione di una unità del rilevatore.

2.5.3 Calcolo dell'intersezione tra un piano e un raggio

Per il calcolo dell'intersezione tra un piano e un raggio ci sono due principali funzioni:

- La funzione `getIntersection` (listato **(2.6)**) calcola le intersezioni tra un raggio e un insieme di piani. Essa risolve un sistema di equazioni della forma indicata nelle formule **(1.4)**. I parametri `a` e `b` rappresentano rispettivamente la componente della sorgente e la componente del rilevatore. Il parametro `plane` è un vettore che contiene le coordinate cartesiane di ciascun piano con cui viene effettuato il calcolo dell'intersezione. I valori parametrici risultanti sono restituiti nel vettore `inters`. La funzione restituisce 0 se il raggio è parallelo ai piani, e 1 in caso contrario.

```
int getIntersection(double a, double b, double *plane, int
    gl_nPlanes, double *inters)
{
    if(a - b != 0){
        for(int i = 0; i < gl_nPlanes; i++){
            inters[i] = (plane[i] - a) / (b - a);
        }
        return 1;
    }
    return 0;
}
```

Listato 2.6: Funzione che calcola l'intersezione tra un raggio e un insieme di piani.

- La funzione `getAllIntersections` (listato **(2.7)**) seleziona le componenti delle coordinate cartesiane della sorgente e del rilevatore e determina le coordinate cartesiane dei piani per il calcolo delle intersezioni. I parametri `source` e `pixel` rappresentano rispettivamente le posizioni della sorgente e dell'unità del rilevatore, mentre `planeIndexRange` specifica l'intervallo degli indici dei piani. L'asse `ax` definisce le coordinate dei piani e viene utilizzato anche per selezionare le componenti della sorgente e del rilevatore. La funzione invoca `getIntersection` per effettuare il calcolo delle intersezioni. Le intersezioni calcolate vengono poi restituite tramite il vettore `a`.

```
1  void getAllIntersections(const double source, const double
       pixel, const Ranges planeIndexRange, double *a, Axis ax)
2  {
3      int start = 0, end = 0;
4      double d;
5
6      start = planeIndexRange.minIndx;
7      end = planeIndexRange.maxIndx;
8      double plane[end - start];
9      if(ax == X){
10         plane[0] = getXPlane(start);
11         d = gl_voxelXDim;
12         if(pixel - source < 0){
13             plane[0] = getXPlane(end);
14             d = -gl_voxelXDim;
15         }
16     } else if(ax == Y){
17         plane[0] = getYPlane(start);
18         d = gl_voxelYDim;
19         if(pixel - source < 0){
20             plane[0] = getYPlane(end);
21             d = -gl_voxelYDim;
22         }
23     } else if(ax == Z){
24         plane[0] = getZPlane(start);
25         d = gl_voxelZDim;
26         if(pixel - source < 0){
27             plane[0] = getZPlane(end);
28             d = -gl_voxelZDim;
29         }
30     } else assert(0);
31     for (int i = 1; i < end - start; i++){
32         plane[i] = plane[i-1] + d;
33     }
34     getIntersection(source, pixel, plane, end - start, a);
35 }
```

Listato 2.7: Funzione che calcola le coordinate dei piani e l'intersezione di un raggio.

2.5.4 Calcolo dell'assorbimento di un raggio

La funzione `computeAbsorbption` calcola la lunghezza dei segmenti sottesi ai voxel a partire dai punti di intersezione del raggio con ciascun voxel. Procede poi con il calcolo dell'indice del voxel per ogni segmento in modo da poter selezionare il coefficiente di attenuazione associato. Applica infine la formula **(1.2)** per calcolare l'attenuazione del raggio. Tra i parametri:

- `source` e `pixel` rappresentano rispettivamente la posizione della sorgente e dell'unità del rilevatore;
- `angle` rappresenta l'indice della posizione della sorgente rispetto a tutte le posizioni ordinate;
- `a` è un vettore che contiene i punti di intersezione espressi come parametri dell'equazione della retta, `lenA` è la lunghezza del vettore;
- `slice` è un numero che indica a partire da quali voxel lungo l'asse y si sta calcolando la proeizione;
- `f` è il vettore dei coefficienti di attenuazione.

Nel listato **(2.8)** si mostra il codice della funzione.

```
1  double computeAbsorption(Point source, Point pixel, int angle,
       double *a, int lenA, int slice, double *f)
2  {
3      const double d12 = sqrt(pow(pixel.x - source.x, 2) + pow(pixel
           .y - source.y, 2) + pow(pixel.z - source.z, 2));
4      double absorbment = 0.0;
5      for (int i = 0; i < lenA - 1; i++) {
6          const double segments = d12 * (a[i + 1] - a[i]);
7          const double aMid = (a[i + 1] + a[i]) / 2;
8          const int xRow = min((int)((source.x + aMid * (pixel.x -
               source.x) - getXPlane(0)) / gl_voxelXDim), gl_nVoxel[X]
                - 1);
9          const int yRow = min3((int)((source.y + aMid * (pixel.y -
               source.y) - getYPlane(slice)) / gl_voxelYDim),
               gl_nVoxel[Y] - 1, OBJ_BUFFER - 1);
10         const int zRow = min((int)((source.z + aMid * (pixel.z -
               source.z) - getZPlane(0)) / gl_voxelZDim), gl_nVoxel[Z]
                - 1);
11         absorbment += f[(yRow) * gl_nVoxel[X] * gl_nVoxel[Z] +
               zRow * gl_nVoxel[Z] + xRow] * segments;
12     }
13     return absorbment;
14 }
```

Listato 2.8: Funzione del calcolo dell'attenuazione di un raggio.

2.5.5 Calcolo della proiezione

L'intero calcolo della proiezione avviene nella funzione `computeProjections`. Essa, a partire da un vettore dei coefficienti `f`, restituisce le immagini di proiezione sotto forma lineare nel vettore `absorbment`. La funzione restituisce inoltre il minimo e il massimo assorbimento ottenuto tra i raggi di tutte le immagini. La funzione prende un ulteriore parametro, il parametro `slice`. Questo indica il numero di voxel lungo l'asse y a partire dal quale bisogna calcolare la proiezione. In questo modo è possibile partizionare l'oggetto verticalmente e calcolare la proiezione in più fasi, effettuando chiamate ripetute alla funzione. Questo permette di limitare la quantità di memoria utilizzata per ogni proiezione. Il listato (2.9) mostra la definizione di `computeProjections`.

```
1 void computeProjections(int slice, double *f, double *absorbment,
     double *absMax, double *absMin){
  ...
```

Listato 2.9: Definizione della funzione `computeProjections`.

La funzione presenta la struttura già discussa in (2.3) e riassunta in figura (2.1):

- L'algoritmo itera su ogni posizione della sorgente e successivamente su ogni unità del rilevatore, in modo da selezionare i due punti che definiscono il raggio ed ottenerne le coordinate cartesiane. `nTheta` e `nSidePixels` rappresentano rispettivamente il numero di posizione della sorgente e il numero di pixel per lato del rilevatore. Segue la porzione di codice dedicata.

```
    ...
15  const int nTheta = (int)(gl_angularTrajectory /
        gl_positionsAngularDistance);
16  const int nSidePixels = gl_detectorSideLength /
        gl_pixelDim;
17  //iterates over each source position
18  for(int positionIndex = 0; positionIndex <= nTheta;
        positionIndex++){
19      const Point source = getSource(positionIndex);
20
21      //iterates over each detector's pixel
22      for (int r = 0; r < nSidePixels; r++) {
23          for (int c = 0; c < nSidePixels; c++) {
24              Point pixel;
25
26              pixel = getPixel(r,c,positionIndex);
27  ...
```

Listato 2.10: Selezione dei punti che definiscono un raggio.

- Successivamente calcola l'intersezione con i piani che delimitano ciascun insieme di piani paralleli. Uno dei casi particolari che si possono verificare è che il raggio sia parallelo ad un insieme di piani, in tal caso non può verificarsi alcuna intersezione con essi. La variabile `isParallel` indica a quale insieme di piani il raggio è parallelo (`X` per i piani paralleli al piano yz, `Y` per i piani paralleli al piano xz, `Z` per i piani paralleli al piano xy), in modo da poterli escludere nei calcoli successivi.

```
    ...
30              int isParallel = -1;
31              getSidesXPlanes(sidesPlanes);
32              if (!getIntersection(source.x, pixel.x,
                    sidesPlanes, 2, &temp[X][0])) {
33                  isParallel = X;
34              }
35              getSidesYPlanes(sidesPlanes, slice);
36              if (!getIntersection(source.y, pixel.y,
                    sidesPlanes, 2, &temp[Y][0])) {
37                  isParallel = Y;
38              }
39              getSidesZPlanes(sidesPlanes);
40              if (!getIntersection(source.z, pixel.z,
                    sidesPlanes, 2, &temp[Z][0])) {
41                  isParallel = Z;
42              }
43      ...
```

Listato 2.11: Calcolo delle intersezioni con i lati dell'oggetto.

- Procede poi al calcolo dell'intervallo dei parametri di intersezione. Il modo in cui vengono calcolati i due estremi permette di verificare se il raggio interseca o meno l'oggetto: nel caso in cui il risultato del calcolo dell'estremo sinistro sia in realtà superiore a quello dell'estremo destro allora il raggio non interseca l'oggetto. Nel caso non intersechi l'oggetto, il blocco termina e si passa all'iterazione successiva. Il listato **(2.12)** che mostra la sezione di codice.

```
    ...
44              aMin = getAMin(temp, isParallel);
45              aMax = getAMax(temp, isParallel);
46
47              if(aMin < aMax){
48      ...
```

Listato 2.12: Calcolo dell'intervallo dei parametri di intersezione.

- Nel caso in cui il raggio intersechi l'oggetto si prosegue prima con il calcolo degli intervalli di indice di ciascun insieme di piani paralleli. Questo avviene al fine di escludere i voxel con cui non avviene l'intersezione. Nel listato (**2.13**) avviene la chiamata alla funzione `getRangeOfIndex` per ciascun insieme di piani paralleli.

```
50      ...
                indeces[X] = getRangeOfIndex(source.x, pixel.
                    x, isParallel, aMin, aMax, X);
51              indeces[Y] = getRangeOfIndex(source.y, pixel.
                    y, isParallel, aMin, aMax, Y);
52              indeces[Z] = getRangeOfIndex(source.z, pixel.
                    z, isParallel, aMin, aMax, Z);
53      ...
```

Listato 2.13: Calcolo dell'intervallo di indici dei piani.

- I valori delle variabili utilizzati per il calcolo degli indici sono di tipo a virgola mobile finita (double), tuttavia gli indici calcolati sono di tipo intero. Nell'approssimare i valori calcolati agli interi più vicini si può verificare che l'estremo destro dell'intervallo risulti inferiore all'estremo sinistro. Nel listato (**2.14**) si rende nulla la lunghezza degli intervalli per cui si verifica tale situazione.

```
55      ...
                    if(lenX < 0){
56                      lenX = 0;
57                  }
58                  if(lenY < 0){
59                      lenY = 0;
60                  }
61                  if(lenZ < 0){
62                      lenZ = 0;
63                  }
64                  const int lenA = lenX + lenY + lenZ;
65      ...
```

Listato 2.14: Controllo sull'approssimazione dei valori degli intervalli di indice dei paini.

- Nel listato (**2.15**) si riporta il proseguimento del codice dove vengono calcolate le intersezione tra il raggio e tutti i piani selezionati per ogni insieme di piani paralleli. Per ciascun insieme, le intersezioni vengono calcolate in ordine crescente del parametro di intersezione. Successivamente si uniscono i tre insiemi di valori ottenuti.

```
    ...
68                  getAllIntersections(source.x, pixel.x,
                        indeces[X], aX, X);
69                  getAllIntersections(source.y, pixel.y,
                        indeces[Y], aY, Y);
70                  getAllIntersections(source.z, pixel.z,
                        indeces[Z], aZ, Z);
71
72                  merge3(aX, aY, aZ, lenX, lenY, lenZ,
                        aMerged);
73      ...
```

Listato 2.15: Calcolo delle intersezioni tra il raggio e tutti i piani selezionati.

- A questo punto si hanno i dati necessari a calcolare le lunghezze dei segmenti e gli indici dei voxel coinvolti. Nel listato (**2.16**), la chiamata a `computeAbsorbption` si occupa di questi ultimi passaggi e calcola l'attenuazione del raggio come in (**1.2**). Il risultato viene restituito come valore del pixel dell'immagine corrispondente, nel vettore `absorbment`. Il risultato viene inoltre confrontato con il massimo e il minimo valore di attenuazione calcolato fin quel momento e vengono aggiornati nel caso sia rispettivamente maggiore del primo o minore del secondo.

```
    ...
75                  const int pixelIndex = positionIndex *
                        nSidePixels * nSidePixels + r *
                        nSidePixels + c;
76                  absorbment[pixelIndex] += computeAbsorption
                        (source, pixel, positionIndex, aMerged,
                        lenA, slice, f);
77                  amax = fmax(amax, absorbment[pixelIndex]);
78                  amin = fmin(amin, absorbment[pixelIndex]);
79
80              }
81          }
82      }
83  }
84  *absMax = amax;
85  *absMin = amin;
86  }
```

Listato 2.16: Calcolo dell'attenuazione del raggio.

2.6 Funzioni per la generazione dell'input

Le funzioni di generazione dell'input sviluppate sono tre, una per ogni configurazione trattata nel paragrafo (**1.2.2**). Ciascuna funzione è progettata per generare solo una parte dell'intera configurazione. Ogniuna di esse presenta i parametri: `offset` che specifica il numero di voxel lungo l'asse y a partire dal quale iniziare la generazione della porzione dell'oggetto e `nOfSlices`, che determina il numero di voxel lungo l'asse y fino al quale si intende estendere la generazione. Il vettore `f` è utilizzato per restituire i valori generati. Le funzioni sono:

- La funzione `generateCubeSlice` crea una configurazione dei coefficienti di attenuazione che rappresenta un cubo con `sideLength` voxel per lato. I voxel che fanno parte del cubo presentano un coefficiente di attenuazione pari a 1, mentre tutti i voxel restanti hanno coefficiente con valore 0. La funzione è mostrata nel codice che segue.

```
1  void generateCubeSlice(double *f, int nOfSlices, int offset,
       int sideLength)
2  {
3      const int innerToOuterDiff = gl_nVoxel[X] / 2 -
           sideLength / 2;
4      const int rightSide = innerToOuterDiff + sideLength;
5
6      for(int n = 0 ; n < nOfSlices; n++){
7          for(int i = 0; i < gl_nVoxel[Z]; i++){
8              for(int j = 0; j < gl_nVoxel[X]; j++){
9                  f[(gl_nVoxel[Z]) * i + j + n * gl_nVoxel[X] *
                       gl_nVoxel[Z]] = 0;
10                 if((i >= innerToOuterDiff) &&
11                     (i <= rightSide) &&
12                     (j >= innerToOuterDiff) &&
13                     (j <= rightSide) &&
14                     (n + offset >= innerToOuterDiff) &&
15                     (n + offset <=  gl_nVoxel[Y] -
                           innerToOuterDiff)){
16                     f[(gl_nVoxel[Z]) * i + j + n * gl_nVoxel[
                           X] * gl_nVoxel[Z]] = 1.0;
17                 } else {
18                     f[(gl_nVoxel[Z]) * i + j + n * gl_nVoxel[
                           X] * gl_nVoxel[Z]] = 0.0;
19                 }
20             }
21         }
22     }
23 }
```

Listato 2.17: Generazione di una configurazione di coefficienti di attenuazione per un oggetto dalla forma di un cubo.

- La funzione `generateSphereSlice` crea una configurazione dei coefficienti di attenuazione che rappresenta una semisfera di raggio pari a `radius`. Il lato troncato della sfera è rivolto lungo il semi-asse positivo delle x. I voxel che fanno parte della semisfera presentano un coefficiente di attenuazione pari a 1, mentre tutti i voxel restanti hanno coefficiente con valore 0. Segue il listato che mostra il codice della funzione.

```
1  void generateSphereSlice(double *f, int nOfSlices, int offset
       , int radius)
2  {
3      for (int n = 0; n < nOfSlices; n++) {
4          for (int r = 0; r < gl_nVoxel[Z]; r++) {
5              for (int c = 0; c < gl_nVoxel[X]; c++) {
6                  Point temp;
7                  temp.y = -(gl_objectSideLenght / 2) + (
                       gl_voxelYDim / 2) + (n + offset) *
                       gl_voxelYDim;
8                  temp.x = -(gl_objectSideLenght / 2) + (
                       gl_voxelXDim / 2) + (c) * gl_voxelXDim;
9                  temp.z = -(gl_objectSideLenght / 2) + (
                       gl_voxelZDim / 2) + (r) * gl_voxelZDim;
10                 const double distance = sqrt(pow(temp.x, 2) +
                        pow(temp.y, 2) + pow(temp.z, 2));
11                 if(distance <= radius && c < gl_nVoxel[Z] /
                       2){
12                     f[(gl_nVoxel[Z]) * r + c + n * gl_nVoxel[
                           X] * gl_nVoxel[Z]] = 1;
13                 } else {
14                     f[(gl_nVoxel[Z]) * r + c + n * gl_nVoxel[
                           X] * gl_nVoxel[Z]] = 0.0;
15                 }
16             }
17         }
18     }
19 }
```

Listato 2.18: Generazione di una configurazione di coefficienti di attenuazione per un oggetto dalla forma di una semisfera.

- In `generateCubeWithSphereSlice` (listato **(2.19)**) la configurazione creata rappresenta un cubo con `sideLength` voxel per lato con una cavità di forma sferica all'interno. I voxel che fanno parte del cubo presentano un coefficiente di attenuazione pari a 1, mentre i voxel al di fuori e quelli che rientrano nella cavità sferica hanno coefficiente con valore 0.

```
void generateCubeSlice(double *f, int nOfSlices, int offset,
    int sideLength)
{
    const int innerToOuterDiff = gl_nVoxel[X] / 2 -
        sideLength / 2;
    const int rightSide = innerToOuterDiff + sideLength;

    for(int n = 0 ; n < nOfSlices; n++){
        for(int i = 0; i < gl_nVoxel[Z]; i++){
            for(int j = 0; j < gl_nVoxel[X]; j++){
                f[(gl_nVoxel[Z]) * i + j + n * gl_nVoxel[X] *
                    gl_nVoxel[Z]] = 0;
                if((i >= innerToOuterDiff) &&
                    (i <= rightSide) &&
                    (j >= innerToOuterDiff) &&
                    (j <= rightSide) &&
                    (n + offset >= innerToOuterDiff) &&
                    (n + offset <=  gl_nVoxel[Y] -
                        innerToOuterDiff)
                ){
                    f[(gl_nVoxel[Z]) * i + j + n * gl_nVoxel[
                        X] * gl_nVoxel[Z]] = 1.0;
                } else {
                    f[(gl_nVoxel[Z]) * i + j + n * gl_nVoxel[
                        X] * gl_nVoxel[Z]] = 0.0;
                }
            }
        }
    }
}
```

Listato 2.19: Generazione di una configurazione di coefficienti di attenuazione per un oggeto dalla forma di un cubo con una cavità interna di forma sferica.

2.7 Descrizione dell'output

Il risultato dell'algoritmo di proiezione è una matrice di valori in virgola mobile. Attraverso l'applicazione di un processo di quantizzazione uniforme, tali valori possono essere mappati su una scala di grigi, facilitando la conversione in un formato di immagine. Sebbene questo processo comporti una perdita di informazioni riguardanti il risultato preciso della proiezione, le immagini generate forniscono comunque una rappresentazione chiara dell'oggetto scansionato. Nelle figure **(2.2)**, **(2.3)** e **(2.4)** sono illustrate le immagini risultanti dalla scansione dei tre modelli di voxel implementati.

Ogni immagine mostra, da sinistra verso destra, le proiezioni ottenute rispettivamente agli angoli di -45°, -15°, 15° e 45° rispetto all'asse y (verticale).Considerando gli angoli di visualizzazione utilizzati per ottenere le immagini, ci si aspetta che le proiezioni siano simmetriche a coppie. Questo effetto è chiaramente osservabile nella proiezione del modello cubico di voxel. Tuttavia, risulta più complesso da intuire nella proiezione della semisfera.

Per una corretta interpretazione delle immagini, è fondamentale comprendere il principio di funzionamento della proiezione stessa: le forme rappresentate nelle immagini riflettono la densità dell'oggetto scansionato. Pertanto, le immagini ottenute non devono essere interpretate come fotografie tradizionali dell'oggetto, ma piuttosto come una rappresentazione della distribuzione di densità all'interno dell'oggetto scansionato. È per questo motivo che il lato troncato della semisfera è visibile in ogni immagine, poiché la proiezione evidenzia la densità dell'oggetto in relazione alla sua geometria e orientamento specifici.

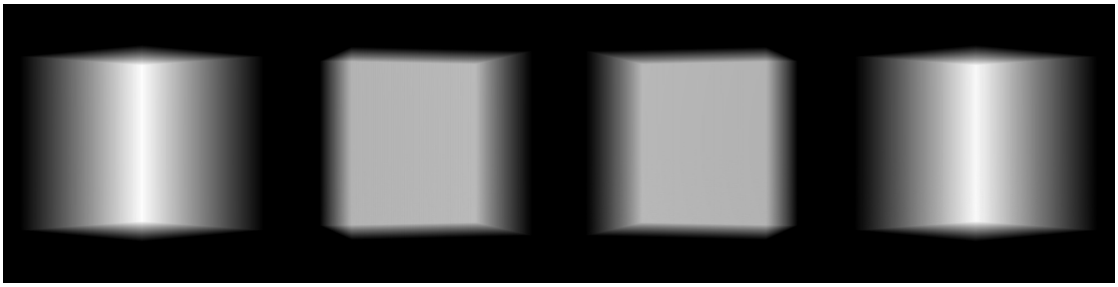

Figura 2.2: Immagine risultante della proiezione di un oggetto di forma cubica.

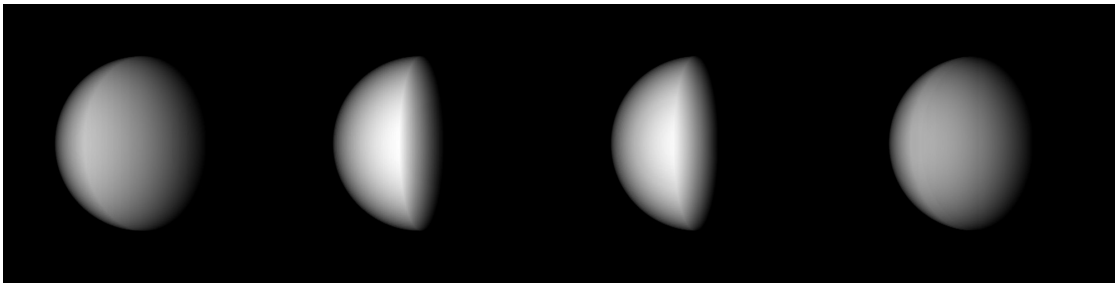

Figura 2.3: Immagine risultato della proiezione di un oggetto di forma semi-sferica.

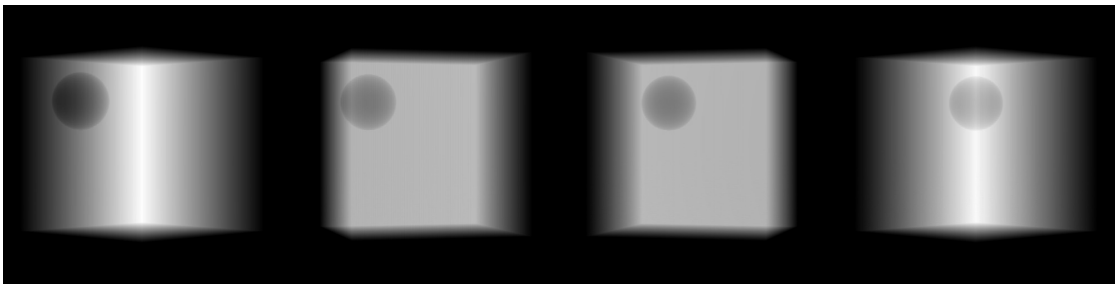

Figura 2.4: Immagine risultato della proiezione di un oggetto di forma cubica con una cavità sferica.

Capitolo 3

Strategie di parallelizzazione

Il presente capitolo è dedicato all'analisi delle strategie di parallelizzazione adottate per l'ottimizzazione delle prestazioni computazionali. L'obiettivo principale è fornire una panoramica dettagliata delle tecniche di parallelizzazione, con particolare attenzione agli strumenti e ai metodi impiegati per migliorare l'efficienza del codice.

3.1 OpenMP

OpenMP è un'interfaccia di programmazione dell'applicazione (API) che offre un modello scalabile e portabile per lo sviluppo di applicazioni parallele su architetture a memoria condivisa. OpenMP ottiene il parallelismo attraverso l'uso di thread, l'unità di elaborazione più piccola gestibile dal sistema operativo. I thread esistono all'interno delle risorse di un singolo processo e la loro quantità può corrispondere al numero di processori o core della macchina, sebbene l'uso effettivo dei thread dipenda dall'applicazione. Il paradigma di programmazione su cui si basa è "single program multiple data": tutti i thread eseguono lo stesso codice, ma ciascuno di essi può accedere e modificare dati differenti e seguire percorsi di esecuzione distinti. Tutti i thread hanno accesso alla memoria condivisa, ma possiedono anche copie temporanee delle variabili che possono essere modificate in modo indipendente dalle variabili in memoria.

Il modello di parallelizzazione di OpenMP segue il paradigma "fork and join". All'inizio, il programma è eseguito in modalità seriale da un singolo thread. Le regioni parallele sono definite tramite direttive #pragma, creando un pool di thread che eseguono il medesimo codice. Al termine della regione parallela, i thread vengono terminati e l'esecuzione prosegue in modalità seriale. L'overhead associa-

to alla creazione e alla terminazione dei thread dipende dall'implementazione di OpenMP.

La maggior parte del parallelismo è ottenuto attraverso le direttive (o pragmas). La direttiva #pragma omp parallel avvia un pool di thread, i quali eseguono tutti le stesse istruzioni. È possibile specificare il numero di thread con la clausola num_threads(), e gestire il parallelismo innestato tramite la variabile di ambiente OMP_NESTED=TRUE. È possibile condizionare l'uso della parallelizzazione attraverso la clausola if. OpenMP fornisce inoltre clausole come reduction, for, atomic e critical per gestire operazioni di riduzione, cicli paralleli, aggiornamenti atomici e regioni critiche. Sono inoltre previste delle clausole per definire la visibilità e il comportamento delle variabili come: le variabili specificate come 'shared' sono condivise tra tutti i thread; private, in questo caso i thread dispongono ciascuno di una copia non inizializzata delle variabili; 'firstprivate', ciascun thread dispone di una copia di tali variabili, ma sono inizializzate al valore contenuto prima della regione parallela. OpenMP offre meccanismi di sincronizzazione, come barriere e regioni critiche, per evitare race conditions. Tuttavia, non esegue automaticamente la parallelizzazione, né verifica la presenza di race conditions o dipendenze nei loop, e non garantisce che la versione parallela sia più veloce di quella seriale.

3.2 Costo computazionale e profiling

Per analizzare il tasso di crescita del tempo di esecuzione occorre stabilire una relazione con la dimensione dell'input e identificare i dati che influiscono maggiormente sul tempo di esecuzione. Nella sezione (2.2) sono stati esaminati i parametri che definiscono le caratteristiche della proiezione, tuttavia, non tutti influiscono direttamente sul carico di lavoro svolto dal programma. In particolare i parametri principali che incidono sulla dimensione dell'input sono:

- **Numero di unità del rilevatore**: è un valore derivato dal **lato del rilevatore** e dal **lato di un'unità del rilevatore**. Esso coincide con il numero di raggi usati per il calcolo della proiezione;
- **Numero di voxel lungo ciascun asse**: determina la risoluzione della campionatura dell'oggetto;
- Rapporto tra gli angoli $\boldsymbol{\theta}$ e $\boldsymbol{\alpha}$: determina il numero di proiezioni da effettuare;

Considerando il diagramma di flusso del programma (2.3) e l'analisi della funzione `computeAbsorbption` (sezione (2.5.5), si può effettuare un'analisi dettagliata di ciascuna fase dell'algoritmo.

Nel listato **(2.10)** è evidente che l'intero codice è organizzato in tre cicli annidati.

Il primo ciclo esegue $\mathbf{N_\theta}$ iterazioni. All'interno di questo ciclo, la funzione `getSource` viene invocata e richiede un tempo di esecuzione costante.

All'interno del primo ciclo for, si trovano due cicli annidati, ciascuno dei quali esegue un numero di iterazioni pari al numero di unità di misurazione per lato del rilevatore.

Nel ciclo annidato più interno, vengono effettuate le chiamate alle seguenti funzioni:

- Le funzioni `getPixel`, `getSidesXplanes`, `getIntersection`, `getAMax`, `getAMin` e `getRangeOfIndex` hanno ciascuna un tempo di esecuzione costante.
- Le funzioni `getAllIntersections`, `merge` e `computeAbsorption` eseguono un numero di istruzioni che dipende dal numero di piani con cui il raggio interseca l'oggetto. Il caso peggiore si verifica quando il raggio interseca tutti i piani che compongono l'oggetto.

Pertanto, il limite superiore del tempo di esecuzione dell'algoritmo è dato da:

$$O(N_\theta \times N_{pixel}^2 \times (nVoxel_X + nVoxel_Y + nVoxel_Z)) \tag{3.1}$$

dove $\mathbf{N_\theta}$ è il numero di proiezioni, $\mathbf{N_{pixel}}$ è il numero di unità di misurazione per lato del rilevatore, mentre $\mathbf{nVoxel_X}$, $\mathbf{nVoxel_Y}$ e $\mathbf{nVoxel_Y}$ sono rispettivamente il numero di voxel lungo l'asse X, Y, Z.

3.3 Versione parallela

3.3.1 Una prima strategia di parallelizzazione

La tabella **(3.1)** riporta i risultati del profiling del programma di calcolo della proiezione, effettuato utilizzando un input di riferimento. I dati nella tabella indicano il numero di invocazioni e la percentuale di tempo impiegata da ciascuna funzione rispetto al tempo totale di esecuzione.

Dall'analisi emerge che le funzioni `computeAbsorption`, `merge` e `min` sono quelle che consumano la maggior parte del tempo di esecuzione, con `computeAbsorption` che contribuisce in misura significativamente maggiore rispetto alle altre. Seguono le funzioni `getAllIntersections`, `getIntersection` e `min3`, che mostrano un tempo di esecuzione simile tra loro.

Le funzioni `getZPlane`, `getYPlane` e `getXPlane` eseguono calcoli sostanzialmente identici. Sebbene la loro percentuale di tempo cumulata sia del 15% del totale, tale valore è attribuibile al numero elevato di invocazioni, piuttosto che alla complessità computazionale di ciascuna funzione, che è costante e relativamente bassa. Infine, le restanti funzioni impiegano un tempo uguale o inferiore all'1% del totale.

Questi risultati suggeriscono una strategia di parallelizzazione che si concentri sulle funzioni che richiedono la maggior parte del tempo di esecuzione. Tuttavia, l'elevato numero di invocazioni delle funzioni principali può comportare un significativo overhead dovuto alla gestione del pool di thread.

3.3.2 La strategia di parallelizzazione applicata

L'analisi approfondita del codice della funzione `computeProjections` rivela che le variabili con visibilità oltre il ciclo annidato più interno includono: un vettore `temp`, con un numero di elementi costante e di ridotte dimensioni; i vettori `aMerged`, `aX`, `aY` e `aZ`, tutti destinati a contenere i parametri dei punti di intersezione dei quali si conosce il numero massimo; il vettore `absorbment`, destinato a contenere i risultati delle attenuazioni dei raggi; `amax` e `amin`, due variabili usate per mantenere i valori massimo e minimo di attenuazione temporanei.

Le uniche che comportano una dipendenza tra iterazioni successive (loop carried dependencies) sono `amin` e `amax`: l'indice dell'elemento del vettore `absorbment` su cui avviene l'accesso dipende univocamente dagli indici dei cicli `for` ed è unico per ogni terna di valori assunti dai tre indici; i valori memorizzati in `temp`, `aMerged`, `aX`, `aY` e `aZ` sono calcolati ad ogni iterazione del ciclo più interno, i vettori sono quindi di appoggio temporaneo. Le restanti variabili sono o locali al ciclo annidato più interno, o gli unici accessi effettuati sono in lettura. Le variabili `amin` e `amax`, invece, sono utilizzate per operazioni binarie associative.

La soluzione proposta applica il pattern di `reduction` per parallelizzare l'applicazione di operatori binari associativi su sequenze di elementi, come, in questo caso, gli operatori di minimo e massimo. OpenMP supporta la parallelizzazione delle riduzioni tramite la clausola appropriata nella direttiva `#pragma omp parallel`. Questa clausola opera in modo da creare una copia privata per ciascun thread della variabile soggetta alla riduzione inizializzandola al valore neutro dell'operazione. Ogni thread esegue quindi la regione parallela, al termine della quale l'operatore di riduzione è applicato tra tutti i valori ottenuti da ciascun thread e il valore contenuto dalla variabile prima della regione parallela.

Percentuale tempo	Invocazioni	Nome
33.28 %	140194890	computeAbsorption
15.48 %	280389780	merge
10.01 %	11423958160	min
7.42 %	420584670	getAllIntersections
6.99 %	1582284510	getIntersection
6.23 %	2759181220	min3
5.55 %	4024285475	getZPlane
5.49 %	4024323119	getXPlane
4.91 %	4094427340	getYPlane
1.13 %	387233280	getAMax
0.93 %	387233280	getAMin
0.92 %	10	computeProjections
0.60 %	420584670	getRangeOfIndex
0.39 %	387233280	getPixel
0.25 %	140194890	merge3
0.13 %	387233280	getSidesYPlanes
0.13 %	387233280	getSidesZPlanes
0.10 %	387233280	getSidesXPlanes
0.01 %	70	getSource

Tabella 3.1: Profiling del programma di calcolo della proiezione. L'analisi è stata eseguita utilizzando un input costituito da un oggetto con una griglia di $1000 \times 1000 \times 1000$ voxel e un rilevatore composto di 2352 unità di misurazione per lato. L'output dell'elaborazione ha ottenuto 7 immagini.

La direttiva applicata (visibile nel listato (**3.1**)) è strutturata come segue:

- `#pragma omp parallel`: definisce una regione parallela creando un pool di thread, che eseguono il medesimo codice.

- `for`: distribuisce le iterazioni di un ciclo `for` tra i thread.
- `collapse(2)`: collassa due cicli annidati in una sequenza di iterazioni unificata, migliorando la distribuzione del carico.
- `schedule(dynamic)`: assegna dinamicamente le iterazioni ai thread durante l'esecuzione.
- `default(none)`: richiede che venga specificata la visibilità per ogni variabile utilizzata nella regione parallela.
- `shared()`: indica le variabili con visibilità condivisa tra i thread, come in questo caso `absorbment`. Essendo questa un puntatore, una visibilità diversa da quella condivisa non permetterebbe di creare una copia privata dell'intero vettore per ogni thread, bensì si otterrebbe una copia del solo puntatore; tutti i thread devono avervi accesso, non si verificano race condition in quanto ogni thread accede ad un elemento distinto.
- `private()`: definisce le variabili con visibilità privata per ciascun thread, come `temp`, `aMerged`, `aX`, `aY`, e `aZ`. Pe rqueste variabili tale visibilità è essenziale per evitare race condition.
- `reduction(min:amin)` e `reduction(max:amax)`: applicano la riduzione degli operatori di minimo e massimo sulle variabili `amin` e `amax`.

`computeProjections` contiene le invocazioni ad ogni altra funzione, con la direttiva mostrata il numero totale di invocazioni di ciascuna funzione viene distribuito tra i thread. È importante inoltre notare che all'interno della regione parallela è presente un blocco condizionale che può influire sul tempo di esecuzione. Per ottimizzare la distribuzione del carico di lavoro, si è scelto di utilizzare la clausola schedule(dynamic).

```
1  void computeProjections(int slice, double *f, double *absorbment,
       double *absMax, double *absMin)
2  {
       ...
18
19 #pragma omp parallel for collapse(2) schedule(dynamic) default(
       none) shared(nSidePixels, positionIndex, source, slice, f,
       absorbment, nTheta, gl_nVoxel) private(temp, aX, aY, aZ,
       aMerged) reduction(min:amin) reduction(max:amax)
20         for (int r = 0; r < nSidePixels; r++) {
21             for (int c = 0; c < nSidePixels; c++) {
22                 Point pixel;
```

Listato 3.1: Applicazione della direttiva OpenMP per parallelizzare il calcolo della proiezione.

Capitolo 4

Valutazione delle prestazioni della soluzione parallela

Il programma di calcolo della proiezione è progettato per gestire configurazioni personalizzabili dei parametri. Tuttavia, la configurazione di parametri descritta di seguito costituiva uno dei requisiti che il programma doveva soddisfare:

- **Dimensione dell'oggetto**: lato di 100.000 μm;
- **Dimensione del voxel**: cubo con lato di 100 μm;
- **Dimensione del rilevatore**: lato di 200.000 μm;
- **Dimensione di un'unità del rilevatore**: lato di 85 μm;
- **Distanza tra la sorgente e l'oggetto**: 150.000 μm;
- **Distanza tra il rilevatore e l'oggetto**: 600.000 μm;
- **Ampiezza angolare della traiettoria**: 90°;
- **Distanza angolare tra le posizioni della sorgente**: 15°.

I suddetti valori sono stati utilizzati come riferimento per testare il programma e per l'analisi delle sue prestazioni. Si precisa che il tempo considerato riguarda esclusivamente il calcolo della proiezione, escludendo le fasi di generazione dell'input, lettura dell'input e scrittura dell'output.

I risultati presentati sono stati ottenuti su una macchina equipaggiata con un processore Intel Core i5-7600, dotato di 4 core fisici e 4 core logici (senza supporto all'hyperthreading).

L'analisi delle prestazioni ha l'obiettivo di rispondere alle seguenti domande:

- In che misura è possibile aumentare la velocità di esecuzione del programma incrementando il numero di core disponibili, mantenendo costante l'input?
- Di quanto è possibile ampliare l'input in modo che il programma venga eseguito nello stesso intervallo di tempo, al variare del numero di core disponibili?

4.1 Strong-scaling efficency

La **strong scaling efficiency** (S.S.E) è un indicatore utile per rispondere alla prima domanda dell'analisi delle prestazioni. La S.S.E. rappresenta il rapporto tra lo speedup e il numero di core utilizzati, fornendo una misura dell'efficienza con cui le risorse vengono impiegate all'aumentare del numero di processori. Lo speedup, a sua volta, indica il rapporto tra il tempo necessario per eseguire un problema con una singola unità di esecuzione e il tempo richiesto quando si utilizzano p unità di esecuzione.

La tabella (4.1) riporta, per ciascun numero di core, i valori del tempo medio di esecuzione, dello speedup e della S.S.E.

N. Core	T. medio	Speed up	S.S.E.
1	802.51	1.00	1.00
2	426.05	1.88	0.94
3	296.52	2.71	0.90
4	232.73	3.45	0.86

Tabella 4.1: In ordine: numero di core, tempo medio impiegato nell'esecuzione, speed-up e strong scaling efficiency.

In uno scenario ideale, la versione parallela dovrebbe richiedere un tempo pari a $\frac{1}{p}$ del tempo necessario alla versione seriale, dove p rappresenta il numero di processori. Tuttavia, questo scenario ideale è difficile da raggiungere nella pratica. Come indicato dalla tabella, con l'aumento del numero di core, lo speedup mostra un incremento decrescente. Questo fenomeno può essere attribuito a un crescente sbilanciamento del carico di lavoro tra i thread.

Seguono il grafico dello speedup e della strong scaling efficiency.

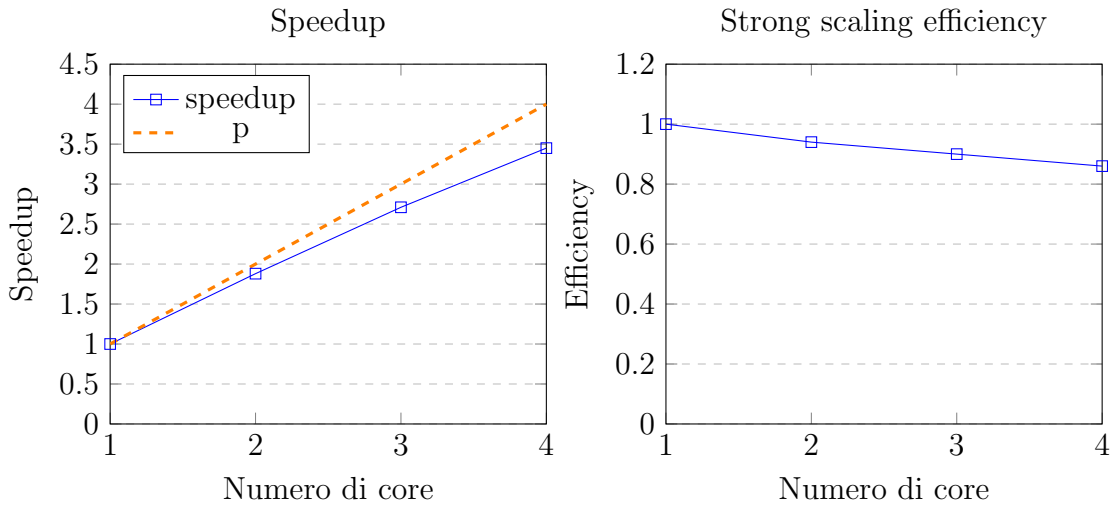

Figura 4.1: A sinistra, il grafico mostra il tasso di crescita dello speedup in funzione del numero di core, confrontato con una crescita lineare. A destra, il grafico illustra la strong scaling efficiency (S.S.E.).

4.2 Weak-scaling efficency

La **weak scaling efficiency** (W.S.E.) rappresenta il rapporto tra il tempo necessario alla versione seriale per elaborare un input e il tempo richiesto alla versione parallela per elaborare un input p volte più grande utilizzando p unità di lavoro.

L'input di riferimento è quello descritto all'inizio del capitolo. Per ottenere un input p volte maggiore, sono stati scalati i valori dei parametri mantenendo le stesse proporzioni tra di loro. Sono state considerate diverse modalità di scaling dell'input:

- *Aumento del numero di proiezioni, risultando in un numero di immagini p volte maggiore.* Questo metodo è stato escluso poiché il carico di lavoro può variare significativamente a seconda dell'angolazione della proiezione. In altre parole, due immagini ottenute da prospettive diverse, anche se riguardano la stessa configurazione dell'oggetto, non corrispondono necessariamente alla stessa quantità di lavoro.
- *Aumento della risoluzione del campionamento dell'oggetto, riducendo la dimensione dei voxel.* Questo approccio mantiene costante il numero di immagini e le prospettive, ma fissa il numero di raggi mentre aumenta il numero di intersezioni per ogni raggio, variando tra i raggi stessi. Questo potrebbe

incrementare le differenze nel tempo di calcolo tra i raggi e, di conseguenza, provocare uno sbilanciamento del carico di lavoro.

- *Aumento della dimensione dell'oggetto e del rilevatore.* In questo caso, il numero di immagini rimane costante, ma aumenta il numero di intersezioni per ciascun raggio e anche il numero di raggi utilizzati per la proiezione.

Tra le tre opzioni considerate è stata scelta l'ultima. La tabella (**4.2**) mostra all'aumentare del numero di core usati, il tempo impiegato e la W.S.E..

N. Core	T. medio	W.S.E
1	801.11	1.00
2	835.75	0.96
3	863.39	0.93
4	877.19	0.91

Tabella 4.2: In ordine: numero di core, tempo medio impiegato nell'esecuzione e weak scaling efficiency.

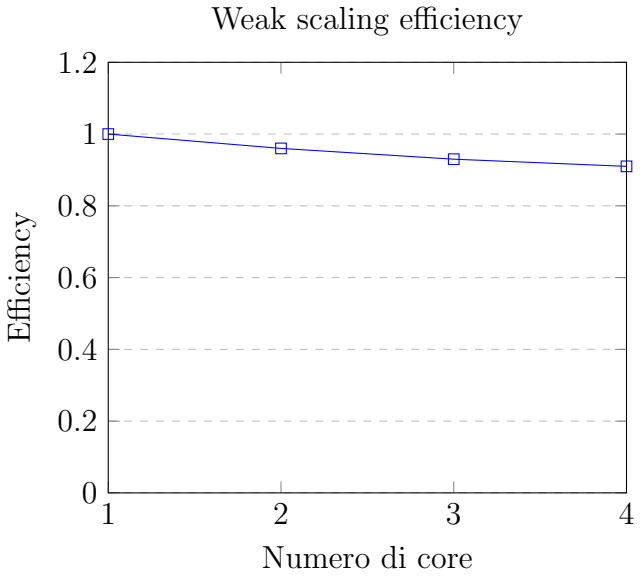

Figura 4.2: mostra l'andamento della weak scaling efficiency.

Capitolo 5

Conclusioni

Il presente lavoro ha affrontato il problema diretto della Tomografia Computerizzata (TC), con l'obiettivo di sviluppare un algoritmo in linguaggio C per la generazione di immagini di proiezione basate su dati noti delle proprietà del corpo studiato. Il lavoro è stato articolato in diverse fasi, ciascuna delle quali ha contribuito in modo significativo al raggiungimento degli obiettivi prefissati.

Nel Capitolo 1, sono state dettagliate le basi matematiche e geometriche necessarie per la risoluzione del problema diretto. È stato fornito un quadro chiaro della matrice M e del vettore f, fondamentali per il calcolo delle proiezioni. Inoltre, sono state esplorate le coordinate cartesiane e le relazioni geometriche tra raggi, sorgente e rilevatore, permettendo una modellazione accurata del sistema.

Il Capitolo 2 ha descritto l'implementazione pratica dell'algoritmo. È stata analizzata la struttura del codice, i parametri di inizializzazione, e il flusso di esecuzione del proiettore. Sono state dettagliate le strutture dati e le funzioni implementate, comprese quelle per il calcolo delle coordinate e l'assorbimento dei raggi, offrendo una visione complessiva della simulazione.

Nel Capitolo 3, è stata esplorata la parallelizzazione dell'algoritmo mediante l'uso di OpenMP. È stata condotta un'analisi approfondita del costo computazionale e delle tecniche di profiling per identificare le aree di miglioramento. Sono state discusse diverse strategie di parallelizzazione, con una descrizione dettagliata della strategia adottata e la sua implementazione effettiva.

Il Capitolo 4 ha fornito una valutazione delle prestazioni della versione parallela dell'algoritmo, analizzandone l'efficienza. I risultati ottenuti hanno dimostrato un miglioramento significativo delle performance rispetto alla versione sequenziale, confermando l'efficacia delle tecniche di parallelizzazione applicate.

In conclusione, il lavoro ha fornito una soluzione efficace e scalabile per il problema diretto della TC, integrando con successo tecniche matematiche, geometriche e computazionali. I risultati ottenuti offrono una base per ulteriori sviluppi e ottimizzazioni, suggerendo che future ricerche potrebbero includere l'esplorazione di ulteriori tecniche di parallelizzazione, con particolare riferimento all'uso delle GPU e la risoluzione del problema inverso della TC.
Il codice sviluppato è reperibile al link: https://github.com/LorenzoColletta/Parallel-implementation-of-a-tomographic-projection-algorithm

Bibliografia

[1] Elena Morotti and Elena Loli Piccolomini. *Sparse Regularized CT Reconstruction: An Optimization Perspective*, pages 1–34. Springer International Publishing, Cham, 2021.

[2] Robert L. Siddon. Fast calculation of the exact radiological path for a three-dimensional ct array. *Medical Physics*, 12(2):252–255, 1985.

[3] Blaise Barney. Openmp. https://hpc-tutorials.llnl.gov/openmp/.