Alma Mater Studiorum · Università di Bologna
Campus di Cesena

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

Parallelizzazione di un algoritmo Ray-Driven per la Ricostruzione Tomografica con OpenMP in C

Tesi di Laurea in
HIGH PERFORMANCE COMPUTING

Relatore
Prof. Moreno Marzolla
Co-relatore
Prof.ssa Elena Loli Piccolomini

Presentata da
Emanuele Borghini

III Sessione di Laurea
Anno Accademico 2023-2024

Sommario

Questa tesi triennale affronta l'applicazione di tecniche di High Performance Computing alla ricostruzione tomografica.
La tomografia computerizzata è una tecnologia che utilizza raggi X per acquisire sezioni trasversali del corpo umano o di oggetti, che vengono poi rielaborate per ottenere immagini tridimensionali. Tuttavia, questo processo è estremamente dispendioso in termini computazionali, e la sua efficienza influisce direttamente sulla qualità e velocità delle immagini ricostruite.

L'obiettivo principale di questo lavoro è ottimizzare il processo di retroproiezione, utilizzando l'algoritmo ray-driven di Robert L. Siddon, implementandone una versione parallela per migliorare ulteriormente le prestazioni.

Prima di partire con l'implementazione, è fornita al lettore una panoramica teorica della tomografia computerizzata, necessaria per comprendere il funzionamento dell'algoritmo usato.

La tesi descrive in dettaglio l'algoritmo di Siddon, includendo le sue formule matematiche e una sua implementazione in linguaggio C.

In seguito, verrà fornita un'introduzione a OpenMP, un'API di programmazione parallela; utilizzata per parallelizzare il programma implementato. Verranno illustrate tre strategie di parallelizzazione applicate all'algoritmo di Siddon, spiegando vantaggi e svantaggi di ciascuna.

Infine, i risultati ottenuti sono stati analizzati in modo approfondito, automatizzando il processo di raccolta dati e confrontando le diverse strategie di parallelizzazione.

L'analisi delle prestazioni è stata effettuata utilizzando metriche standard apprese durante il corso di HPC, dimostrando come l'obiettivo stabilito sia stato raggiunto con successo.

La tesi si conclude con una valutazione del lavoro svolto, proponendo possibili sviluppi futuri; con la speranza che questo progetto possa essere un punto di partenza per ulteriori studi e applicazioni reali.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

1.1 High Performance Computing in ambito medico

La tomografia computerizzata, sviluppata negli anni '70 da Godfrey Hounsfield e Allan Cormack, ha rivoluzionato il campo medico, fornendo immagini dettagliate delle strutture interne del corpo umano senza la necessità di interventi chirurgici invasivi. Utilizzando i raggi X, la TAC acquisisce sezioni trasversali ricostruibili in immagini tridimensionali, permettendo di individuare con precisione anomalie e patologie di vario genere. [1]

Tuttavia, la qualità delle immagini dipende fortemente dall'efficienza dei metodi di ricostruzione impiegati. La ricostruzione di immagini tridimensionali, a partire da proiezioni bidimensionali prodotte da un tomografo, richiede un notevole costo computazionale. Maggiore è la qualità dell'immagine, maggiore è il tempo di calcolo necessario per l'elaborazione, diventando spesso il fattore limitante.

È qui che entra in gioco l'informatica per ottimizzare il processo, partendo da algoritmi di ricostruzione più efficienti e poi velocizzando ulteriormente con tecnologie di *High Performance Computing*. L'ottimizzazione algoritmica e la parallelizzazione consentono di migliorare la velocità delle ricostruzioni, permettendo di aumentare la qualità delle immagini e di ridurre i tempi di attesa per i pazienti, aprendo la strada a nuove possibilità in campo medico.

Negli ultimi 40 anni, la tecnologia per la ricostruzione tomografica ha compiuto enormi progressi [2]. Questo progetto si inserisce in questo percorso di innovazione, contribuendo con un'implementazione efficiente e parallelizzata di un algoritmo per la ricostruzione tomografica.

1.2 Obiettivo della tesi

L'obiettivo di questa tesi è la parallelizzazione dell'algoritmo di retroproiezione proposto da Robert L. Siddon, usando *OpenMP*, mirando a migliorarne significativamente le prestazioni. Questo significa saper riconoscere la tipologia del problema e saperlo parallelizzare in modo efficiente, applicando le nozioni di *High Performance Computing* apprese durante il corso di laurea. Inoltre, bisogna saper valutare i risultati ottenuti e presentarli in modo chiaro e comprensibile.

1.3 Struttura della tesi

La tesi è stata scritta con l'obiettivo di essere accessibile anche a lettori senza una conoscenza approfondita di informatica o radiologia. Il documento è strutturato in modo da fornire inizialmente le basi teoriche necessarie per comprendere la tomografia assiale computerizzata e l'algoritmo di Siddon. Successivamente, viene illustrata l'implementazione in C dell'algoritmo, seguita da una descrizione dettagliata del processo di parallelizzazione del codice tramite *OpenMP*. I risultati ottenuti sono poi presentati e discussi, con una valutazione oggettiva del lavoro svolto e dei possibili sviluppi futuri.

Capitolo 2

Fondamenti di Tomografia Computerizzata

Nella tesi tratteremo un aspetto specifico della radiografia, la tomografia computerizzata, includendo anche il calcolo del percorso radiologico attraverso un oggetto tridimensionale. È necessario comprendere almeno in parte cos'è la tomografia, il funzionamento di un tomografo e il processo di ricostruzione delle immagini.

La tomografia è una tecnica di diagnostica per immagini (o più semplicemente, *imaging*) utilizzata in vari settori come quello industriale, scientifico e medico, con lo scopo di ottenere immagini tridimensionali, o una sequenza di sezioni bidimensionali, di oggetti partendo da una serie di proiezioni acquisite lungo angolazioni diverse.

In breve, la tomografia si basa sull'acquisizione di dati provenienti tipicamente da un fascio di raggi X, che attraversano l'oggetto preso in esame e interagiscono con esso tramite specifiche proprietà fisiche. I raggi attraversando l'oggetto vengono attenuati in base alla densità e alla composizione del materiale che incontrano lungo il loro percorso. Una volta attraversato l'oggetto, i raggi raggiungono una serie di rivelatori posti di fronte alla sorgente. Questi rivelatori misurano l'intensità residua dei raggi, che risulta attenuata rispetto all'emissione iniziale. In base al valore di attenuazione, viene calcolato il coefficiente di assorbimento del materiale attraversato per ciascun raggio.

La tesi approfondirà due concetti inerenti alla tomografia: il problema del **calcolo del percorso radiologico** e la **ricostruzione tomografica** in un contesto di **High Performance Computing**.
Questo capitolo fornisce solo una panoramica generale sulla tomografia computerizzata strettamente necessaria per comprendere il problema che verrà affrontato, in caso si voglia approfondire l'argomento lascio al lettore la possibilità di consultare *Principles of Computerized Tomographic Imaging*[3].

2.1 Struttura di un Tomografo Moderno

Il tomografo computerizzato è una macchina generalmente costituita da:

- **Sorgente**: uno o più punti da dove il dispositivo emette i raggi X (o altre forme di radiazione). Nei tomografi moderni, questa sorgente è montata su un supporto rotante, che consente di irradiare i raggi da diverse angolazioni tutto attorno all'oggetto;
- **Oggetto scansionato**: non fa parte del dispositivo stesso, ma è il volume dell'oggetto che si vuole analizzare. Può essere un paziente, un oggetto industriale o qualsiasi altro oggetto che si desidera esaminare;
- **Il rivelatore**: una serie di sensori posizionati di fronte alla sorgente, sul lato opposto dell'oggetto, che ha il compito di misurare l'intensità dei raggi dopo che questi lo hanno attraversato, essenzialmente fornendo dati per la successiva ricostruzione dell'immagine.

Il programma creato simulerà un **tomografo di terza generazione**[4], dove la sorgente di raggi e i rivelatori ruotano in modo sincrono attorno all'oggetto in esame come mostrato in figura 2.1. Ulteriori dettagli sul processo di acquisizione delle proiezioni verranno trattati nei capitoli successivi.

Comprendere questa configurazione molto basilare è cruciale per i capitoli successivi, poiché influenzerà sia il modello geometrico impiegato per calcolare il percorso dei raggi, sia l'implementazione e l'ottimizzazione delle tecniche di parallelizzazione necessarie per gestire il grande volume di dati da elaborare.

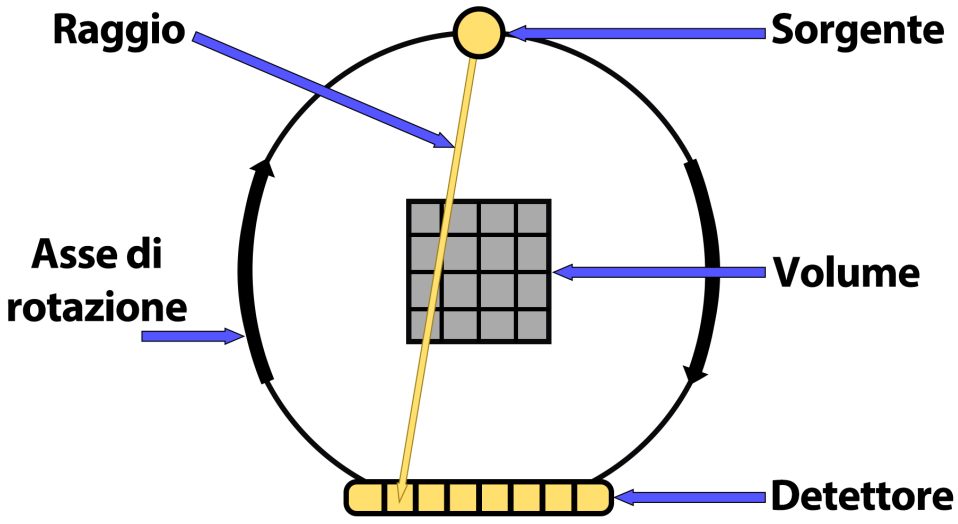


Figura 2.1: Diagramma di un tomografo di terza generazione.

2.2 Ricostruzione Tomografica

Il processo di ricostruzione tomografica è un problema inverso, in cui si cerca di ritornare a una rappresentazione tridimensionale di un oggetto a partire da una sequenza di proiezioni bidimensionali acquisite da un tomografo. Queste proiezioni forniscono informazioni sull'assorbimento e/o attenuazione della radiazione lungo ciascun percorso, ma non descrivono direttamente la distribuzione interna dell'oggetto.

L'obiettivo della ricostruzione è, quindi, determinare i valori dei *voxel* (abbreviazione di *volumetric pixel*), che rappresentano piccole unità volumetriche all'interno dell'oggetto in esame, a partire dai dati delle proiezioni. Per fare ciò, è necessario risolvere il problema del percorso radiologico, che determina come e dove i raggi interagiscono con l'oggetto lungo il loro tragitto.

Il metodo più elementare di ricostruzione è la retroproiezione (*Backprojection*), che "proietta all'indietro" i dati di ogni proiezione attraverso lo spazio tridimensionale, distribuendo l'informazione lungo i raggi percorsi, e accumulando i valori di attenuazione in ciascun voxel attraversato.

Esistono altri metodi, in particolare quelli iterativi che si basano su una ripetuta stima della soluzione e successiva correzione; essi cercano di minimizzare gli errori e garantire una ricostruzione più accurata rispetto ai metodi analitici. Questi approcci più sofisticati, sebbene più costosi dal punto di vista computazionale, sono in grado di fornire risultati migliori.

Nel contesto di questa tesi, verrà considerato solo il caso più semplice di retroproiezione, in quanto rappresenta un metodo di ricostruzione adatto e sufficiente per applicare diverse nozioni di High Performance Computing.

2.3 Problema del calcolo del percorso radiologico

La sfida principale della ricostruzione tomografica è il calcolo del percorso esatto che un raggio segue attraverso l'oggetto in questione. Questo processo, noto come **ray tracing** o **calcolo del percorso radiologico**, è essenziale per misurare con precisione l'interazione tra il fascio di raggi e la sezione dell'oggetto attraversata.

Il calcolo del percorso radiologico diventa un problema computazionale complesso, soprattutto in un contesto tridimensionale. In particolare, la tecnica usata, nota come **ray-driven**, traccia ogni raggio dalla sorgente attraverso una griglia tridimensionale di voxel, calcolando le intersezioni del raggio con ciascun voxel attraversato. Questo approccio richiede un'elevata capacità di calcolo, poiché bisogna determinare la lunghezza dell'intersezione all'interno di ogni voxel attraversato.

Per comprendere meglio, possiamo immaginare un fascio di raggi X che parte dalla sorgente e attraversa un oggetto tridimensionale. L'oggetto è suddiviso in una griglia di piccoli cubi chiamati voxel. Ogni volta che il raggio entra in un nuovo voxel, è necessario calcolare la distanza che percorre al suo interno. Questo è importante perché la quantità di attenuazione del raggio dipende dalla lunghezza del percorso all'interno di ciascun voxel e dalle proprietà del materiale di cui è composto.

Il calcolo del percorso radiologico è quindi un problema di geometria computazionale, dove si devono determinare tutte le intersezioni tra il raggio e i voxel, e calcolare le lunghezze di questi segmenti. Questo processo deve essere ripetuto per ogni raggio emesso dalla sorgente, il che può risultare in un numero molto elevato di calcoli, specialmente con una risoluzione del rivelatore elevata.

Nella tesi, l'algoritmo *Fast calculation of the exact radiological path for a three-dimensional CT array*[5] di Robert L. Siddon sarà utilizzato per risolvere questo problema in modo efficiente. Approfondito nel capitolo successivo.

Capitolo 3

Algoritmo di Siddon

In questo capitolo verrà presentato l'algoritmo *Fast calculation of the exact radiological path for a three-dimensional CT array*[5] sviluppato da Robert L. Siddon. Pubblicato negli anni '80, questo algoritmo ha rappresentato un'importante innovazione nel campo della tomografia computerizzata, poiché ha reso possibile il calcolo del percorso radiologico in tre dimensioni in tempi altrimenti proibitivi.

L'algoritmo di Siddon risolve il problema computazionalmente complesso del calcolo delle intersezioni tra un raggio e i voxel di un volume tridimensionale, riducendo significativamente la complessità rispetto ai metodi tradizionali.

Nel corso di questo capitolo, verranno esplorati i dettagli dell'algoritmo, inclusa la sua complessità computazionale e spaziale, la matematica e la geometria sottostante, il design e l'implementazione in linguaggio C. Inoltre, verranno discusse le strutture dati necessarie per l'implementazione e le limitazioni del programma, fornendo una visione completa di quello che sarà il punto di partenza per la parallelizzazione.

3.1 Matematica e geometria

Prima di affrontare l'algoritmo, è importante dare le basi matematiche e geometriche necessarie per comprenderne il funzionamento. Questa sezione offre una panoramica sintetica di questi concetti matematici per garantire una visione più chiara di ciò che si sta calcolando e il perché.

3.1.1 Spazio tridimensionale

Nel programma sviluppato per la tesi, definiamo lo spazio tridimensionale con origine nel centro del volume dell'oggetto in esame. L'asse verticale è l'asse Z, con valori negativi

verso l'alto (in direzione della sorgente) e positivi verso il basso (in direzione del rivelatore). L'asse X e Y compongono le sezioni orizzontali del volume, con X che va da sinistra a destra e Y da fronte a retro.

La sorgente e rivelatore si trovano entrambi su due archi circolari di raggio rispettivamente DOS (Distance of Source) e DOD (Distance of Detector) attorno al volume e paralleli tra loro.

Possiamo esprimere le coordinate spaziali per i punti sorgente (X_s, Y_s, Z_s) e di destinazione (X_d, Y_d, Z_d) dei raggi sfruttando la trigonometria. Per la sorgente, è sufficiente considerare l'angolo di rotazione θ della sorgente rispetto all'asse Z, e calcolare le coordinate (X_s, Y_s, Z_s) come:

$$\begin{aligned} X_s &= -\sin(\theta) \times DOS, \\ Y_s &= \cos(\theta) \times DOS, \\ Z_s &= 0, \end{aligned}$$

Per la destinazione, invece, dobbiamo anche tener conto della posizione del pixel rispetto al rivelatore. Quindi, oltre all'angolo di rotazione θ bisogna considerare lo spostamento sul piano orizzontale del rivelatore in questo modo:

$$\begin{aligned} X_d &= \cos(\theta) \times \Delta_X, \\ Y_d &= -\sin(\theta) \times \Delta_X, \\ Z_d &= \Delta_Y. \end{aligned}$$

dove Δ_X e Δ_Y sono rispettivamente lo spostamento orizzontale e verticale del pixel rispetto all'angolo in alto a sinistra del rivelatore.

3.1.2 Piani paralleli

Lo spazio tridimensionale è suddiviso in una griglia formata da piani paralleli agli assi coordinati. Questi piani sono definiti dalle coordinate X_i, Y_j e Z_k con i, j e k che vanno da 1 a N_x, N_y e N_z rispettivamente, dove N_x, N_y e N_z sono il numero di voxel lungo gli assi X, Y e Z.

Ogni piano è distante d_x, d_y e d_z rispettivamente lungo gli assi X, Y e Z, le distanze tra i piani rappresentano la dimensione dei voxel. Usando i piani come delimitatori bisogna fare qualche calcolo per risalire alla posizione in base agli indici di uno specifico voxel. Vedremo come calcolare gli indici nella sezione 3.2.

3.1.3 Retta parametrica in tre dimensioni

Un raggio in tre dimensioni può essere rappresentato da una retta parametrica, definita da due punti $P_1 = (X_1, Y_1, Z_1)$ e $P_2 = (X_2, Y_2, Z_2)$ e un parametro α che varia da 0 a 1.

La retta è definita come:

$$X(\alpha) = X_1 + \alpha(X_2 - X_1),$$
$$Y(\alpha) = Y_1 + \alpha(Y_2 - Y_1),$$
$$Z(\alpha) = Z_1 + \alpha(Z_2 - Z_1).$$

d'ora in poi considereremo il primo punto P_1 come il punto di partenza di un raggio dalla sorgente e il secondo punto P_2 come il centro di un pixel del rivelatore del tomografo.

3.1.4 Calcoli con Numeri Finiti

Anche se lo scopo di questa tesi non è quello della simulazione fisica esatta, è importante tenere a mente che i calcoli effettuati dal programma sono limitati dalla precisione finita dei numeri in virgola mobile dei calcolatori. Questo significa che, in alcuni casi, potrebbero verificarsi errori di approssimazione che potrebbero influire sulla accuratezza dei risultati.

Il programma non effettua attivamente nessuna mitigazione degli errori numerici, l'unica precauzione presa è l'utilizzo di `double` per rappresentare i numeri in virgola mobile. Questo permette di avere una precisione fino a circa 15-16 cifre decimali. [1]

Nonostante il tipo `double` offra una precisione di circa 16 cifre decimali, non si può garantire che i risultati siano sempre completamente accurati. In particolare, se l'algoritmo o il problema trattato è mal condizionato, gli errori numerici possono propagarsi e influenzare l'accuratezza del risultato finale. Questo fenomeno può verificarsi anche con la doppia precisione.

3.2 Metodologia dell'algoritmo

L'algoritmo di Siddon si basa su un principio fondamentalmente semplice per convertire uno spazio tridimensionale in tre set di piani bidimensionali, facilitando il calcolo delle intersezioni tra un raggio e i voxel di una griglia tridimensionale. In questo modo, l'intero volume tridimensionale viene suddiviso in una griglia di voxel, dove ognuno è delimitato da piani paralleli agli assi coordinati.

Per ogni asse X, Y e Z, vengono definiti i piani bidimensionali che delimitano i voxel. Ad esempio, per l'asse X, i piani sono definiti dalle coordinate X_i con i che va da 1 a N_x (numero di voxel lungo l'asse X). Analogamente, per gli assi Y e Z, i piani sono definiti rispettivamente dalle coordinate Y_j e Z_k.

[1] La precisione dei numeri in virgola mobile rappresentati con il tipo `double` è generalmente di 15-16 cifre decimali su processori che seguono lo standard IEEE 754, come la maggior parte delle CPU moderne, inclusi quelli di Intel e AMD.

Un raggio invece viene rappresentato da una linea composta da tre parametri:

$$\begin{aligned} X(\alpha) &= X_1 + \alpha(X_2 - X_1), \\ Y(\alpha) &= Y_1 + \alpha(Y_2 - Y_1), \\ Z(\alpha) &= Z_1 + \alpha(Z_2 - Z_1) \end{aligned} \tag{1}$$

dove (X_1, Y_1, Z_1) rappresenta il punto di partenza del raggio e (X_2, Y_2, Z_2) il punto di arrivo. Il parametro α varia da 0 a 1, indicando la posizione del raggio lungo la sua traiettoria.

Per comprendere meglio, ho realizzato una rappresentazione grafica 3.1 di un raggio che attraversa un volume tridimensionale in linea retta parallela all'asse X.

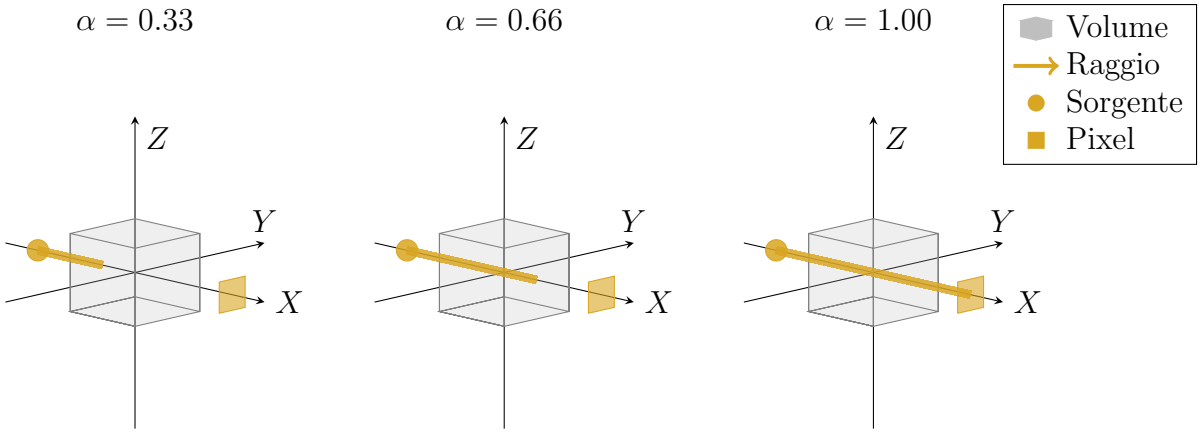


Figura 3.1: Rappresentazione parametrica di un raggio che attraversa un volume tridimensionale.

Questa rappresentazione si può facilmente estendere in più dimensioni, considerando che un raggio può attraversare il volume in qualsiasi direzione.

Gli insiemi dei piani bidimensionali X_{piano}, Y_{piano} e Z_{piano} dei tre assi vengono anch'essi rappresentati da equazioni parametriche del tipo:

$$\begin{aligned} X_{piano} &= X_{piano}(1) + (i-1)d_x \quad \text{con} \quad i = 1, 2, ..., N_x, \\ Y_{piano} &= Y_{piano}(1) + (j-1)d_y \quad \text{con} \quad j = 1, 2, ..., N_y, \\ Z_{piano} &= Z_{piano}(1) + (k-1)d_z \quad \text{con} \quad k = 1, 2, ..., N_z, \end{aligned} \tag{2}$$

dove i piani di indice (1) e (N_x), (N_y), (N_z) rappresentano rispettivamente il piano più vicino e più lontano rispetto alla sorgente. d_x, d_y e d_z rappresentano la distanza tra due piani adiacenti lungo gli assi X, Y e Z rispettivamente.

Con queste equazioni abbiamo definito lo spazio su cui svolgeremo il calcolo del percorso radiologico, ora passiamo a descrivere come l'algoritmo di Siddon calcola le intersezioni tra un raggio e i voxel del volume.

L'algoritmo inizia con il calcolo delle coordinate delle intersezioni α tra il raggio e i lati dei piani per ciascun asse. Quindi, partendo con l'asse X, troviamo le intersezione per i due piani di indice 1 e N_x, toccati dal raggio:

$$\begin{aligned} \alpha_x(1) &= \frac{X_{piano}(1) - X_1}{X_2 - X_1}, \\ \alpha_x(N_x) &= \frac{X_{piano}(N_x) - X_1}{X_2 - X_1}. \end{aligned} \tag{3}$$

Notare che se $X_1 = X_2$ allora il raggio è parallelo all'asse X e non interseca nessun piano, quindi consideriamo $\alpha_x(1)$ e $\alpha_x(N_x)$ come indefiniti.

Ripetiamo lo stesso procedimento per gli assi Y e Z, calcolando $\alpha_y(1)$, $\alpha_y(N_y)$, $\alpha_z(1)$ e $\alpha_z(N_z)$.

Con le intersezioni dei lati del volume calcolate, possiamo determinare i punti di ingresso e uscita del raggio dal volume. Il punto di ingresso è definito come il punto di intersezione tra il raggio e il piano più vicino alla sorgente, mentre il punto di uscita è quello con il piano più lontano.
In termini matematici, il punto di ingresso α_{min} e il punto di uscita α_{max} sono calcolati come:

$$\begin{aligned} \alpha_{min} = \max\{&\min(\alpha_x(1), \alpha_x(N_x)), \\ &\min(\alpha_y(1), \alpha_y(N_y)),\ \min(\alpha_z(1), \alpha_z(N_z)),\ 0\}, \\ \alpha_{max} = \min\{&\min(\alpha_x(1), \alpha_x(N_x)), \\ &\max(\alpha_y(1), \alpha_y(N_y)),\ \max(\alpha_z(1), \alpha_z(N_z)),\ 1\}. \end{aligned} \tag{4}$$

Con i punti di ingresso e uscita possiamo filtrare i piani che il raggio attraversa, riducendo il numero di calcoli necessari per determinare le intersezioni con i voxel.

Calcoliamo gli indici minimi e massimi dei piani attraversati lungo ciascun asse:

$$\begin{aligned} i_{min} &= N_x - (X_{piano}(N_x) - \alpha_{min} \times (X_2 - X_1) - X_1)/d_x, \\ i_{max} &= 1 + (X_1 + \alpha_{max} \times (X_2 - X_1) - X_{piano}(1))/d_x. \end{aligned} \tag{5}$$

Nel caso in cui $X_2 < X_1$, scambiamo i valori di α_{min} e α_{max}.

Ripetiamo lo stesso procedimento per gli assi Y e Z, ottenendo j_{min}, j_{max}, k_{min} e k_{max}.

Per ogni indice i, j, k calcoliamo le intersezioni tra il raggio e i piani attraversati.

Definiamo tre insiemi di valori parametrici α_x, α_y e α_z, partendo dall'asse X:

Se $(X_2 - X_1) > 0$ allora:

$$\{\alpha_x\} = \{\alpha_x(i_{min}), ..., \alpha_x(i_{max})\}, \tag{6}$$

altrimenti:

$$\{\alpha_x\} = \{\alpha_x(i_{max}), ..., \alpha_x(i_{min})\}.$$

dove $\alpha_x(i)$ è il valore di α per il piano X_i, è ottenuto come:

$$\alpha_x(i) = \frac{X_{piano}(i) - X_1}{X_2 - X_1}.$$

oppure calcolato ricorsivamente:

$$\alpha_x(i) = \alpha_x(i-1) + \frac{d_x}{X_2 - X_1}.$$

Gli insiemi generati dall'equazione (6) risultano ordinati in modo crescente.

Unendo i valori di $\{\alpha_x\}$, $\{\alpha_y\}$ e $\{\alpha_z\}$ in un unico insieme $\{\alpha\}$ e ordinandoli in modo crescente otteniamo tutti i punti di intersezione tra il raggio e i voxel attraversati.

Due punti adiacenti nell'insieme corrispondono a un'intersezione tra il raggio e un voxel.

Per concludere, l'algoritmo di Siddon calcola anche la lunghezza di ogni segmento di raggio all'interno di ciascun voxel.

Trovare la lunghezza dell'intersezione definita da due elementi m e $m-1$ appartenenti ad $\{\alpha\}$ significare fare:

$$l(m) = d_{12} \times (\alpha(m) - \alpha(m-1)), \tag{7}$$

dove d_{12} è la distanza euclidea tra i punti di intersezione:

$$d_{12} = \sqrt{(X_2 - X_1)^2 + (Y_2 - Y_1)^2 + (Z_2 - Z_1)^2}. \tag{8}$$

Per trovare gli indici i, j, k del voxel attraversato dall'intersezione m e $m-1$ troviamo prima il punto medio:

$$\alpha_{mid} = \frac{\alpha(m) + \alpha(m-1)}{2}, \tag{9}$$

e poi calcoliamo gli indici:

$$\begin{aligned} i(m) &= 1 + (X_1 + \alpha_{mid} \times (X_2 - X_1) - X_{piano}(1))/d_x, \\ j(m) &= 1 + (Y_1 + \alpha_{mid} \times (Y_2 - Y_1) - Y_{piano}(1))/d_y, \\ k(m) &= 1 + (Z_1 + \alpha_{mid} \times (Z_2 - Z_1) - Z_{piano}(1))/d_z. \end{aligned} \tag{10}$$

Con questi calcoli, l'algoritmo è terminato e abbiamo tutti i dati per proseguire con la ricostruzione tomografica.

3.2.1 Complessità computazionale e spaziale

In questa sezione, discutiamo la complessità computazionale e spaziale dell'algoritmo utilizzato. Do per scontato che il lettore abbia familiarità con questi concetti, in caso contrario consiglio di consultare il terzo capitolo di *Introduction to Algorithms*[6].

Per risolvere il problema del calcolo del percorso radiologico, in un approccio ingenuo, si potrebbe pensare di iterare attraverso ogni voxel del volume e verificare se il raggio interseca ciascuno di essi. Questo comporta una complessità computazionale di $O(n^3)$, dove n è il numero di voxel per lato del volume. Risultando chiaramente proibitivo per volumi di grandi dimensioni.

L'algoritmo proposto da Siddon riduce significativamente la complessità computazionale del problema. Invece di iterare attraverso ogni singolo voxel, l'algoritmo calcola direttamente i punti di intersezione del raggio con i bordi dei voxel, tracciando il raggio lungo il suo percorso e determinando esattamente quali voxel vengono attraversati e per quanto tempo. Questo approccio invece di scalare con il numero di voxel, dipende dal numero di piani che compongono la griglia di voxel, riducendo la complessità a $O(n)$.

Tuttavia, nonostante il miglioramento nella complessità computazionale, la complessità spaziale del problema rimane invariata. Indipendentemente dal metodo utilizzato per calcolare le intersezioni dei raggi, è necessario memorizzare i valori di attenuazione o assorbimento per ciascun voxel del volume. Questo significa che la memoria richiesta è ancora di $O(n^3)$, poiché si devono immagazzinare i dati relativi a ciascun voxel nel volume tridimensionale.

La gestione della memoria è un aspetto critico, specialmente in scenari di grandi dimensioni di input. Di conseguenza, diventa essenziale adottare tecniche di gestione della memoria efficienti o considerare strategie di compressione dei dati per ridurre l'impatto della memoria spaziale necessaria.

3.3 Design e implementazione in C

3.3.1 Strutture dati

Per semplificare lo sviluppo del programma sono state definite alcune strutture dati basilari per rappresentare lo spazio tridimensionale, i raggi, i voxel e i dati di proiezione. Di seguito vengono elencate insieme al loro utilizzo nell'algoritmo.

```
typedef enum axis {
    NONE = -1,
    X = 0,
    Y = 1,
    Z = 2
} axis;
```

L'enumerazione `axis` è utilizzata per rappresentare i tre assi cartesiani (X, Y, Z) o come indice per accedere alle componenti di un punto nello spazio tridimensionale. Viene sfruttata anche per comparare gli assi durante i calcoli delle intersezioni.

```
typedef union point3D {
    const double coordsArray[3];
    struct coords {
        const double x;
        const double y;
        const double z;
    } coords;
} point3D;
```

La struttura `point3D` rappresenta un punto nello spazio tridimensionale. È definita come un'unione tra un array di coordinate e una struttura con coordinate esplicite per x, y, e z. Viene usata per rappresentare la sorgente e il pixel (punto finale) del raggio.

```
typedef struct ray {
    const point3D source;
    const point3D pixel;
} ray;
```

La struttura `ray` rappresenta un raggio nello spazio tridimensionale, non è altro che l'unione dei due punti di partenza e arrivo del raggio. Semplifica la gestione dei dati relativi al raggio tra le funzioni del programma incapsulando i dati in un'unica struttura.

```c
typedef struct range {
    int min;
    int max;
} range;
```

La struttura `range` definisce un intervallo con un valore minimo e massimo, ed è impiegata nell'algoritmo per delimitare i piani da considerare lungo ogni asse, riducendo il numero di controlli necessari nel calcolo delle intersezioni.

```c
typedef struct projection {
    int index;
    double angle;
    double minVal;
    double maxVal;
    int nSidePixels;
    double* pixels;
} projection;
```

La struttura `projection` rappresenta una singola proiezione tomografica. Include l'indice della proiezione, l'angolo di acquisizione, i valori di assorbimento minimo e massimo e una matrice di pixel che contiene i dati della proiezione. Questo tipo di struttura è utilizzata per memorizzare tutti i dati provenienti dal rivelatore e successivamente necessari per la ricostruzione del volume.

```c
typedef struct volume {
    const int nVoxelsX;
    const int nVoxelsY;
    const int nVoxelsZ;
    const double voxelSizeX;
    const double voxelSizeY;
    const double voxelSizeZ;
    double* coefficients;
} volume;
```

La struttura `volume` rappresenta il volume tridimensionale di voxel, con informazioni sul numero di voxel lungo ciascun asse ($nVoxelsX$, $nVoxelsY$, $nVoxelsZ$) e sulle dimensioni di ciascun voxel. La memoria per i coefficienti di assorbimento è allocata come un array tridimensionale. Questa struttura immagazzina i dati relativi ai voxel attraversati dal raggio e per determinare gli assorbimenti lungo il percorso radiologico. Alla fine dell'esecuzione del programma, i valori contenuti in `volume.coefficients` rappresentano il volume ricostruito.

3.3.2 Struttura generale del programma

Il programma è strutturato in modo modulare, parametrico, ordinato e con funzioni distinte per ciascuna fase dell'algoritmo di Siddon. Particolare attenzione è stata posta alla leggibilità e alla manutenibilità del codice.

Il programma può essere suddiviso in tre macro sezioni:

- **Input e inizializzazione**: in questa fase vengono letti i dati in input, inizializzate le strutture dati e allocata la memoria necessaria per i calcoli successivi;
- **Calcolo del percorso radiologico**: in questa fase viene eseguito l'algoritmo di Siddon per calcolare il percorso radiologico per ciascun raggio e determinare gli assorbimenti nei voxel attraversati;
- **Output**: in questa fase vengono scritti i risultati su file per poi essere visualizzati tramite software di visualizzazione medica.

La figura 3.2 mostra il diagramma di flusso del programma, illustrando le principali fasi in colori differenti.

Input e inizializzazione

Prima dell'esecuzione, il programma richiede all'utente di specificare i parametri necessari per la ricostruzione:

- le dimensioni dei voxel lungo gli assi X, Y e Z;
- le dimensioni di un pixel del rivelatore;
- la distanza angolare percorsa dalla sorgente durante l'acquisizione delle proiezioni
- la distanza angolare tra le proiezioni;
- la lunghezza del lato del volume;
- la distanza tra la sorgente e il volume;
- la distanza tra il volume e il rivelatore.

Questi parametri possono essere impostati direttamente nel codice sorgente, mentre per i file delle proiezioni in input e volume in output sono richiesti i percorsi ad ogni esecuzione del programma.

Dopo aver validato la correttezza dei parametri, il programma procede con l'inizializzazione delle strutture dati per le proiezioni e volume. Alloca la memoria per i coefficienti di assorbimento dei voxel e dei pixel delle proiezioni, azzera i valori dei voxel e legge i dati delle proiezioni dai file in input con la funzione `readProjection`.

Oltre a queste operazioni, il programma inizializza degli array di supporto per memorizzare dei valori intermedi per accelerare il calcolo, come le posizioni geometriche dei piani iniziali e finali per ogni asse e tabelle di lookup per le funzioni trigonometriche.

Calcolo del percorso radiologico

È la fase che occupa la maggior parte del tempo di esecuzione del programma. In questa fase, una funzione del programma `computeBackProjection` raccoglie tutti gli step dell'algoritmo di Siddon e calcola il percorso radiologico per ciascun pixel nella proiezione.

All'interno della funzione summenzionata, vengono richiamate le seguenti funzioni per calcolare il percorso radiologico, ognuna ricopre un'equazione dell'algoritmo di Siddon:

- `getSidesIntersections`: implementa l'equazione (3), restituisce una coppia di valori per gli assi X, Y e Z calcolando le intersezioni tra il raggio e il primo e l'ultimo piano attraversato di ciascun asse;
- `getAMin` e `getAMax`: implementa l'equazione (4), calcola i valori di α_{min} e α_{max} con i valori di `getSidesIntersections`;
- `getPlanesRanges`: implementa l'equazione (5), calcola gli indici minimi e massimi dei piani attraversati lungo ciascun asse e li salva in una tripla di `range`;
- `getAllIntersections`: implementa l'equazione (6), calcola gli insiemi di valori di $\{\alpha_x\}$, $\{\alpha_y\}$ e $\{\alpha_z\}$ per ciascun asse;
- `mergeIntersections`: unisce i valori di $\{\alpha_x\}$, $\{\alpha_y\}$ e $\{\alpha_z\}$ in un unico insieme $\{\alpha\}$ efficientemente tramite un algoritmo di merge sort;
- `computeAbsorption`: implementa le equazioni (7), (8), (9) e (10), per ogni intersezione trovata, calcola la lunghezza del segmento di raggio all'interno del voxel e determina gli indici i, j, k del voxel attraversato. Infine, aggiorna il valore del voxel sommando il prodotto tra la lunghezza e il valore del pixel colpito.

Output

Quando il programma ha terminato i calcoli su tutte le proiezioni, esegue la funzione `writeVolume` che ha come unico compito scrivere i dati della struttura `volume` in un file. La funzione si adatta al tipo di file di output specificato dall'utente, attualmente supporta i formati `.nrrd` e `.raw`. Questi file possono poi essere visualizzati tramite software di visualizzazione medica e verranno discussi ulteriormente nel capitolo 6.1.

Per maggiori dettagli implementativi, documentazione e codice sorgente completo consultare le appendici (A.1) alla fine della tesi.

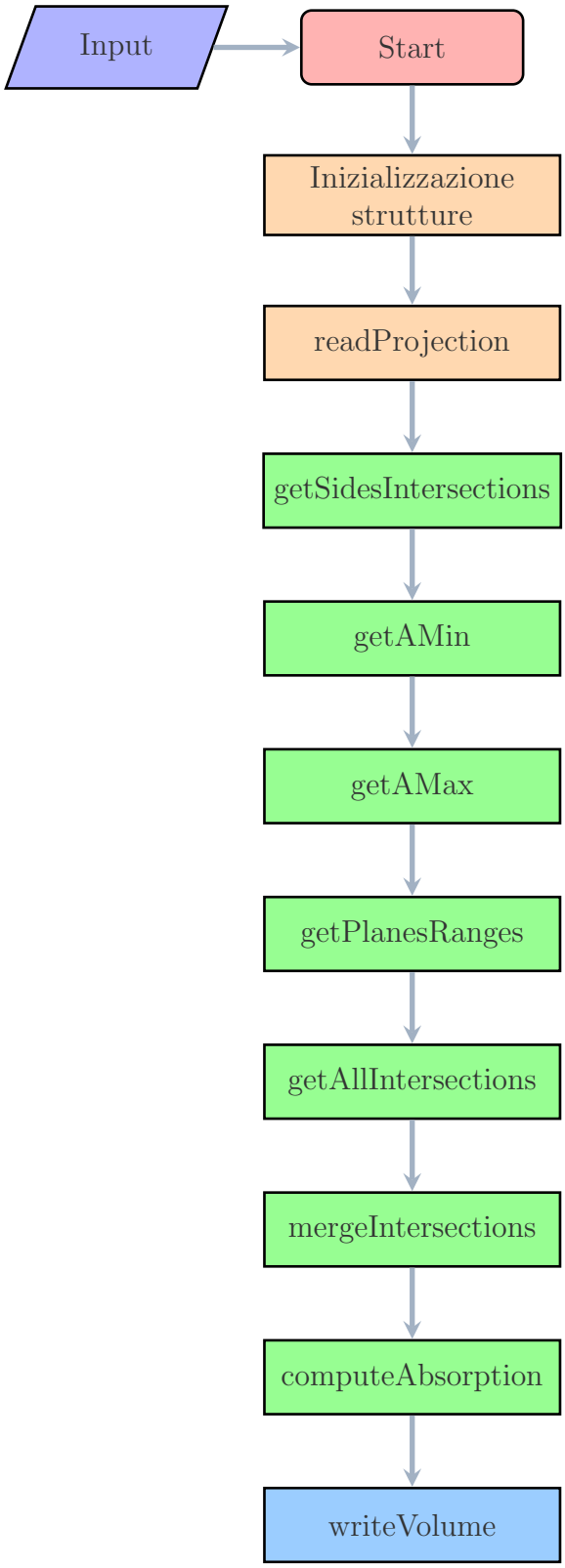


Figura 3.2: Diagramma di flusso del programma.

Capitolo 4

Parallelizzazione dell'algoritmo con OpenMP

Questo capitolo è dedicato alla parallelizzazione dell'algoritmo di Siddon, uno degli obiettivi principali di questa tesi. Partiamo con spiegare cos'è OpenMP e successivamente l'utilizzo della libreria per il linguaggio C e la sua applicazione all'algoritmo, discutendo le scelte implementative e le prestazioni attese.

4.1 Introduzione a OpenMP

OpenMP è un'API, ovvero un'interfaccia per librerie software di programmazione parallela, ideata per facilitare lo sviluppo di software ad alte prestazioni su architetture a memoria condivisa. È compatibile con linguaggi come C, C++ e Fortran, e consente l'esecuzione di codice *multi-thread* senza richiedere una gestione complessa dei *thread* manuale. È un ottimo strumento per risolvere comuni pattern di programmazione parallela.

OpenMP dipende dal supporto del compilatore e utilizza direttive del preprocessore per indicare quali sezioni del programma parallelizzare, integrandosi direttamente nel codice sorgente senza la necessità di riscritture significative. Questo approccio semplifica notevolmente l'ottimizzazione delle prestazioni per le CPU multi-core in programmi già esistenti.

4.2 Architettura a memoria condivisa

Un'architettura a memoria condivisa è un tipo di configurazione hardware in cui più processori o *core* condividono un'unica memoria centrale.

Un esempio comune è una CPU multi-core in cui ciascun *core* rappresenta un'unità di elaborazione separata, ma con accesso diretto alla RAM. In contesti di programmazione parallela, come in un'applicazione OpenMP, ogni *core* può eseguire un *thread*, che rappresenta un flusso di istruzioni parallelo indipendente. Tutti i *thread* creati possono interagire con le stesse variabili e strutture dati nella memoria riservata dal programma.

L'architettura a memoria condivisa è una delle due principali utilizzate per la programmazione parallela, insieme all'architettura a memoria distribuita. La differenza primaria tra le due è la modalità di accesso alla memoria. In un'architettura a memoria distribuita, i *thread* devono comunicare attraverso messaggi per accedere o scambiare dati degli altri processori.

4.3 Direttive di OpenMP

Le direttive di OpenMP sono istruzioni interpretate dal compilatore per generare codice eseguibile in parallelo. Queste direttive vengono inserite nel codice sorgente e sono riconosciute dal compilatore solo se il supporto per OpenMP è abilitato.

Questa sezione coprirà principalmente le direttive usate nel programma con una breve spiegazione di ciascuna. L'intero manuale con tutte le regole è disponibile nella documentazione ufficiale[7].

4.3.1 Parallelizzazione di cicli

La parallelizzazione dei cicli è una tecnica comune in OpenMP. Fu una delle prime funzioni introdotte nella libreria[8].

OpenMP fornisce una direttiva specifica, `parallel for`, che permette di distribuire le iterazioni di un ciclo tra i *thread* disponibili in modo automatico. Per particolari esigenze, è possibile specificare manualmente la grandezza dei blocchi di iterazioni assegnate a ciascun *thread* con il parametro `chunksize`.

Ad esempio, il seguente codice esegue un ciclo in parallelo:

```
1 #pragma omp parallel for
2 for (int i = 0; i < N; i++) {
3     // Codice eseguito in parallelo
4 }
```

Esiste anche un parametro opzionale: `collapse`, che serve a parallelizzare più cicli nidificati. Quando si utilizza la clausola `collapse`, OpenMP tratta i cicli innestati come un unico ciclo di iterazioni. Ad esempio, consideriamo il seguente codice:

```
#pragma omp parallel for collapse(2)
for (int i = 0; i < N; i++) {
    for (int j = 0; j < M; j++) {
        // Operazioni da parallelizzare
    }
}
```

In questo esempio, la clausola `collapse(2)` indica a OpenMP di trattare i due cicli `for` come un unico ciclo con `N * M` iterazioni. OpenMP distribuisce poi queste iterazioni tra i *thread* disponibili.

4.3.2 Variabili condivise e private

In OpenMP, la gestione dello *scope* delle variabili all'interno delle sezioni parallele differisce dalla normale programmazione seriale. In questo contesto, lo *scope* di una variabile definisce come questa viene condivisa o meno tra i vari *thread* durante l'esecuzione parallela. OpenMP permette di definire lo *scope* delle variabili attraverso specifiche direttive che garantiscono un controllo preciso sull'accesso alla memoria condivisa e privata.

Le principali categorie di *scope* delle variabili in OpenMP sono:

- **Shared**: le variabili *shared* sono condivise tra tutti i *thread*. Questo significa che tutti i *thread* accedono alla stessa istanza della variabile, il che può portare a problemi di concorrenza se non gestita correttamente. Per impostazione predefinita, le variabili definite esternamente a un blocco `parallel` sono considerate *shared*;
- **Private**: una variabile *private* è locale a ciascun *thread*, ovvero ogni *thread* ha la propria copia indipendente e non inizializzata. Questo assicura che le modifiche apportate da un *thread* non influenzino gli altri *thread*, ma comporta anche la necessità di inizializzare manualmente le variabili se necessario;
- **Firstprivate**: simile a *private*, ma con la differenza che le variabili sono inizializzate per ogni *thread* con il valore che la variabile aveva prima dell'inizio del blocco parallelo. Questo è utile quando si desidera che ogni *thread* lavori con una copia locale della variabile, ma inizializzata a un valore iniziale uguale.

Oltre a queste direttive, OpenMP offre la possibilità di controllare ulteriormente lo *scope* delle variabili attraverso la clausola `default`, che permette di definire un comportamento predefinito per tutte le variabili all'interno di una sezione parallela.

I possibili valori sono:

- **Shared**: imposta tutte le variabili come condivise, a meno che non siano dichiarate diversamente. È l'opzione predefinita, ma può portare a comportamenti indesiderati se non gestita con attenzione;

- **None**: impedisce a OpenMP di dedurre automaticamente lo *scope* delle variabili, richiedendo che tutte vengano dichiarate esplicitamente come *shared* o *private*. Questo approccio è spesso consigliato, poiché riduce il rischio di errori di accesso concorrente, migliorando la chiarezza del codice e la gestione delle variabili. Studi come "An Evaluation of Auto-Scoping in OpenMP" [9] suggeriscono che l'uso esplicito dello *scoping* riduce il rischio di bug dovuti a errori di deduzione automatica da parte del compilatore e migliora le prestazioni.

4.3.3 Sincronizzazione

Quando si lavora con più *thread* e variabili `shared`, a seconda del contesto, potrebbe essere necessario gestire l'ordine di accesso all'area di memoria condivisa per evitare una *race condition*, ossia una situazione in cui più *thread* tentano di accedere e modificare una variabile condivisa contemporaneamente, generando risultati imprevedibili. OpenMP offre direttive come `critical`, `atomic`, `barrier`, `master` e `single`. Nel programma, la sincronizzazione è utilizzata per gestire correttamente gli aggiornamenti dei voxel, come vedremo più avanti.

Le uniche direttive di sincronizzazione utilizzate nel programma sono:

- **Critical**: garantisce che un'intera sezione di codice venga eseguita da un solo *thread* alla volta. Quando un *thread* entra in una sezione critica, gli altri *thread* devono attendere che il *thread* in esecuzione termini prima di poter accedere alla sezione critica. Le sezioni possono contenere un numero arbitrario di istruzioni, ma è consigliabile mantenerle il più brevi possibile per evitare di ridurre le prestazioni del programma;

- **Atomic**: garantisce che l'accesso a una variabile sia atomico, ovvero che non venga interrotto da altri *thread*. Al contrario della direttiva `critical`, `atomic` è usata per singole operazioni, come incrementi o somme. È possibile specificare anche il tipo di operazione da eseguire in modo atomico, nel nostro caso è `atomic update`.

4.4 Parallelizzazione dell'algoritmo di Siddon

4.4.1 Classificazione del problema di High Performance Computing

Partiamo analizzando il problema di calcolo del percorso radiologico, valutando un singolo raggio che attraversa il volume. Considerando le funzioni descritte nella sezione 3.3.2, quella che occupa la maggior parte del tempo di esecuzione è `computeAbsorption`, fortunatamente, vedremo che è anche la più facile da parallelizzare.

Secondo le classificazioni di High Performance Computing, il calcolo del percorso radiologico è un problema che si presta bene al parallelismo, definito come *embarrassingly parallel*. Nella terminologia HPC, un problema *embarrassingly parallel* è un problema che può essere suddiviso in compiti completamente indipendenti, dove ogni sezione opera senza dipendenze con le altre. Questo rende il problema altamente scalabile e ideale per l'esecuzione in parallelo. Nel caso delle intersezioni di raggi, dove ognuna viene associata ad un solo voxel, possiamo considerare ogni intersezione come indipendente.

Tuttavia, quando si utilizza l'algoritmo di Siddon per la retroproiezione, la complessità logica aumenta e il problema può spostarsi dalla semplice parallelizzazione delle intersezioni di un singolo raggio, al calcolo di tutti i raggi in parallelo. A prima vista potrebbe sembrare altrettanto semplice da parallelizzare, ma con questo approccio viene introdotto un nuovo problema, ovvero l'aggiornamento dei valori dei voxel. Dobbiamo chiederci, cosa succederebbe se due *thread* tentassero di aggiornare lo stesso voxel contemporaneamente?

4.4.2 Problema di race condition

Mettiamoci nella situazione in cui due raggi, gestiti da due *thread* diversi, abbiano un punto di intersezione in comune. Questo particolare caso è illustrato in figura 4.1 e dà vita a una possibile *race condition*. Qui entra in gioco la sincronizzazione descritta precedentemente. Nel programma parallelo, viene marcata l'operazione di aggiornamento dei pixel come `critical`, garantendo che la modifica di un voxel sia eseguita in modo atomico da un solo *thread* alla volta.

L'uso della sincronizzazione comporta un impatto sulle prestazioni del programma. Se ci fossero molte collisioni tra i raggi, l'efficienza del programma potrebbe diminuire significativamente. Questo perché i *thread* devono attendere il loro turno per accedere alla sezione critica, causando un aumento del tempo di esecuzione complessivo. In scenari con un alto numero di collisioni, il vantaggio del parallelismo può essere ridotto a causa del costo introdotto dalla sincronizzazione.

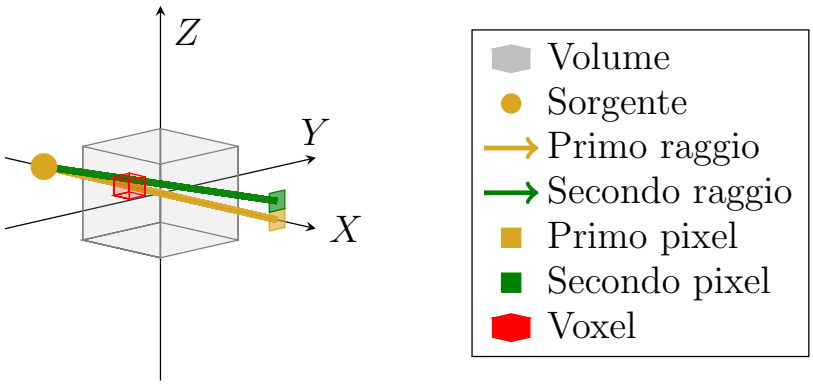


Figura 4.1: Esempio di collisione tra raggi nello stesso voxel.

4.5 Strategie di parallelizzazione

In questa sezione faremo uno studio più profondo di quali tecniche alternative possiamo applicare per ottimizzare il problema ulteriormente, valutando i pro e i contro, le loro limitazioni e scalabilità.

Per ognuna verrà mostrato un esempio di pseudo implementazione, con le direttive OpenMP utilizzate. Si noti che i dettagli implementativi sono stati omessi per semplicità, ma le direttive sono utilizzate in modo appropriato per parallelizzare i cicli e sincronizzare le operazioni critiche.

In fondo a questa sezione verrà data anche una rappresentazione grafica (figura 4.2) delle strategie di parallelizzazione. L'istogramma mostra la percentuale di righe di codice parallela e seriale per ciascuna strategia.

4.5.1 Parallelizzazione del calcolo delle intersezioni di un raggio

Questo è l'approccio più semplice già accennato che consiste nel parallelizzare il ciclo che itera le intersezioni di ciascun raggio con i voxel e calcola gli indici e la lunghezza delle intersezioni. Un grosso vantaggio è che questa strategia non necessita di sincronizzazione, poiché ogni *thread* si occupa di un'unica intersezione alla volta, i calcoli di un voxel sono completamente indipendenti dagli altri. Tuttavia, a seconda della grandezza del volume e del numero di intersezioni, potrebbe non essere la strategia più efficiente, poiché il carico di lavoro potrebbe essere sbilanciato. Questo vuol dire che in caso un raggio attraversi un piccolo numero di pixel, o addirittura nessuno, la sezione parallela del codice non sarebbe sfruttata appieno. Il numero di intersezioni è influenzato dalla struttura del volume, dalla posizione della sorgente e del rivelatore e il numero di voxel totale. Questo approccio

è particolarmente inadatto per volumi sparsi, con numero di voxel piccolo e sorgente e rivelatore molto vicini.

È stata implementata in questo modo, con la direttiva `parallel for`:

```
void computeAbsorption(const ray ray, const double a[], const int lenA,
                       const volume* volume, const projection* projection,
                       const int pixelIndex) {
    #pragma omp parallel for
    for(int i = 1; i < lenA; i ++){
        [...] // equazioni (7), (8), (9) e (10)
    }
}
```

4.5.2 Parallelizzazione di pixel

In questa strategia, la sezione parallela del programma si espande dal singolo raggio a una proiezione intera. Ogni *thread* esegue il calcolo del percorso radiologico per un singolo pixel del rivelatore allo stesso tempo. Anche in questo caso si parallelizza un ciclo, ma a differenza del caso precedente, è necessario applicare la sincronizzazione per evitare le *race condition* discusse in precedenza. Dal lato positivo, questa strategia non ha lo stesso grado di problema di sbilanciamento del carico, poiché se un raggio non ha intersezioni, il *thread* termina rapidamente il calcolo e passa al pixel successivo. L'ambiente in cui questa strategia eccelle è l'opposto della precedente, delineando due casi d'uso distinti.

È stata implementata in questo modo, con le direttive `parallel for` e `atomic`:

```
void computeBackProjection(const projection* projection, volume* volume) {
    [...]
    #pragma omp parallel for collapse(2)
    for (int row = 0; row < projection->nSidePixels; row++) {
        for (int col = 0; col < projection->nSidePixels; col++) {
            [...]
            #pragma omp atomic update
            volume->coefficients[voxelIndex] += voxelAbsorptionValue;
            [...]
        }
    }
}
```

4.5.3 Parallelizzazione di proiezioni

Quest'ultimo approccio si applica a un contesto ancora più ampio, parallelizzando l'iterazione su tutte le diverse proiezioni in input. In questo caso, ogni *thread* si occupa di una proiezione diversa.

Questo approccio migliora la coerenza dei dati, riducendo la quantità di risorse condivise tra i *thread* e aumentando la probabilità che i dati necessari siano già presenti nella *cache*. La *cache* è un livello di memoria più piccolo e più veloce situato vicino alla CPU, in cui vengono memorizzati temporaneamente valori a cui si accede di frequente. Nella programmazione parallela, l'uso efficace della *cache* è fondamentale per migliorare le prestazioni riducendo il tempo di accesso ai dati rispetto alla memoria principale.

Anche in questo caso è necessaria la sincronizzazione per i valori dei voxel. Inoltre, la sezione parallela include ora anche la lettura da file. La soluzione implementata nel programma usa barriere e indici per far si che la lettura da file avvenga in modo seriale e sequenziale, mentre tutti i calcoli all'interno della proiezione sono paralleli. A livello di prestazioni vedremo nei capitoli successivi che la lettura da file seriale sarà ininfluente rispetto al tempo totale di esecuzione. Per questo motivo non sono stati implementati metodi più complessi per parallelizzarla.

In questo caso viene anche introdotta una nuova direttiva di OpenMP, `schedule`, che permette di controllare il carico di lavoro tra i *thread*. Questa direttiva definisce come vengono assegnate le iterazioni del ciclo parallelo ai *thread*, scegliendo tra `static` e `dynamic`. Con `static`, le iterazioni vengono divise in blocchi di dimensione fissa assegnati ai *thread*, mentre con `dynamic`, ogni *thread* riceve un blocco di iterazioni alla volta, riducendo il rischio di sbilanciamento del carico di lavoro per questo caso particolare. Dato che le diverse angolazioni delle proiezioni attorno al volume possono avere un numero di intersezioni variabile, la scelta di `dynamic` è stata usata per garantire che i *thread* non rimangano inattivi mentre altri sono ancora occupati.

Un beneficio unico a quest'ultime due soluzioni è che lasciano intatto il codice seriale iniziale. È quindi possibile esternare l'implementazione dell'algoritmo per il percorso radiologico e retroproiezione in una libreria separata, parallelizzando solo il codice di gestione delle proiezioni.

Quest'ultima strategia è stata implementata in questo modo:

```
1 int main(int argc, char* argv[]) {
2     [...] // Lettura dei parametri
3
4     #pragma omp parallel for schedule(dynamic)
5     for (int i = 0; i < N_THETA; i++) {
6         // La lettura da file deve essere seriale
```

```
7           #pragma omp critical
8           readProjectionDAT(inputFile, &projection,
9                             &width, &height, &minVal, &maxVal);
10
11          // Mentre il resto del codice è parallelo
12          computeBackProjection(&projection, &volume);
13      }
14
15      [...] // Scrittura del volume ricostruito
16  }
```

4.5.4 Parallelizzazione condizionale

Infine, consideriamo una versione ibrida che combina le tecniche descritte in precedenza, sfruttando la direttiva di OpenMP `if`, per selezionare dinamicamente la strategia adatta in base alle caratteristiche del problema e alla configurazione hardware.

Come abbiamo visto, la scelta della strategia dipende da diversi fattori. Si può partire dalla strategia delle proiezioni, considerando che, su una CPU commerciale, il numero di *core* è limitato e spesso inferiore a quello delle proiezioni richieste [10][11]. Quindi sfruttare la *cache* dovrebbe essere più vantaggioso rispetto alle altre strategie. Successivamente, a patto di aver ancora *core* disponibili, consideriamo il numero di intersezioni e il numero di pixel. Se il numero di intersezioni di un raggio è basso, evitiamo semplicemente di parallelizzare il ciclo e dedichiamo i *thread* ai pixel per un'efficienza maggiore. Infine, per architetture avanzate con molti *core*, è possibile aggiungere la parallelizzazione sia di intersezioni e pixel in aggiunta alle proiezioni.

Questa soluzione ibrida dovrebbe risultare in una scalabilità superiore rispetto a qualsiasi singola strategia, adattandosi in modo più flessibile ad input diversi e migliorando le prestazioni del sistema.

4.5.5 Primo confronto delle strategie

In figura 4.2 è mostrato un istogramma che confronta le percentuali di righe di codice parallelo e seriale per le tre strategie di parallelizzazione. Ci può dare un'idea delle prestazioni attese per ciascuna strategia; in teoria più la sezione parallela è grande, più il programma dovrebbe essere veloce. Nel prossimo capitolo vedremo effettivamente i risultati pratici ottenuti.

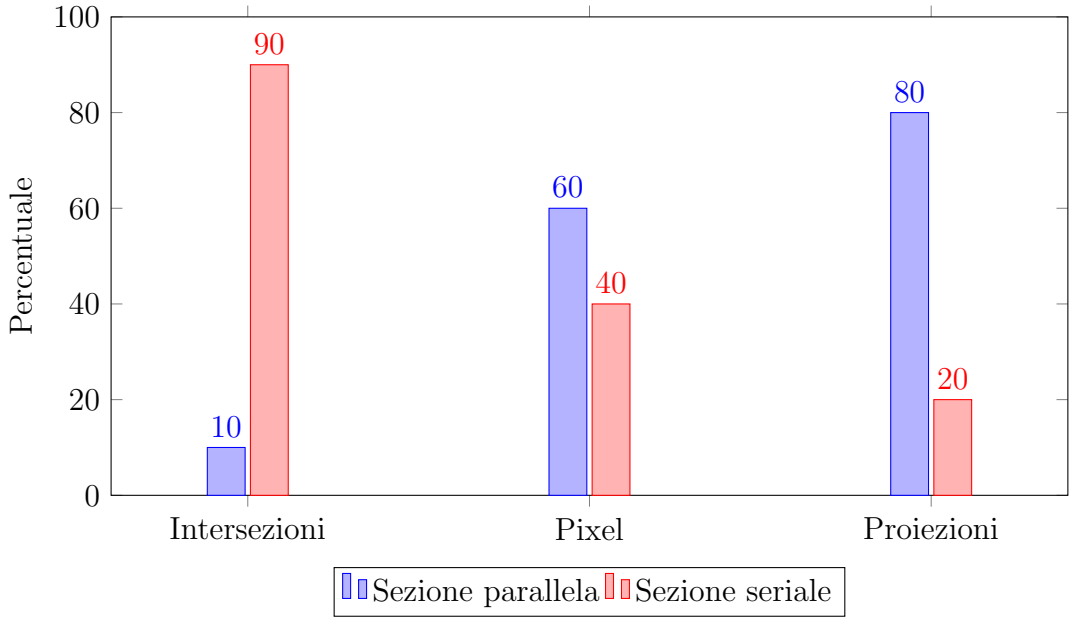


Figura 4.2: Percentuale di righe di codice parallelo e seriale per le diverse strategie di parallelizzazione.

Capitolo 5

Analisi delle prestazioni

In questo capitolo analizziamo le prestazioni del programma sviluppato. In particolare, l'implementate delle diverse strategie di parallelizzazione discusse precedentemente. L'obiettivo principale è valutare l'efficienza e la scalabilità del programma, utilizzando alcune metriche standard.

La sezione iniziale descrive le fondamenta alla base di questo studio delle prestazioni, descrivendo l'ambiente di lavoro utilizzato. Segue una breve descrizione degli strumenti utilizzati per misurare le prestazioni del programma. Poi verranno spiegate le formule matematiche utilizzate per calcolare le metriche di valutazione delle prestazioni. Infine, verranno presentati tabelle e grafici che illustrano i tempi di esecuzione e le metriche di valutazione, per ciascuna strategia di parallelizzazione implementata.

5.1 Ambiente software e hardware

Questa sezione descrive l'ambiente in cui sono stati effettuati i test delle prestazioni del programma. È importante che tutti i dettagli riguardo l'hardware e il software siano riportati in modo da poter riprodurre i test che verranno descritti in seguito. Consentendo di verificare la validità dei risultati ottenuti e confrontarli con quelli di altri autori.

5.1.1 Sistema operativo, compilatore e librerie

Il programma è scritto in linguaggio C, più precisamente seguendo lo standard ISO/IEC 9899:1999 [12], comunemente noto come C99. Il compilatore utilizzato è GCC (GNU Compiler Collection) versione 11.4.0. GCC è stato scelto perché, a partire dalla versione 6.0 [13], supporta le estensioni di OpenMP per il linguaggio C. Inoltre GCC costituisce

lo standard de facto per la compilazione di programmi scritti in C e C++. La versione di OpenMP supportata da GCC 11.4.0 è la 4.5 [14].

Il programma è sviluppato per essere eseguito su sistemi operativi basati su Linux. Non essendoci particolari dipendenze richieste dal sistema operativo, può essere compilato ed eseguito su qualsiasi sistema operativo che supporti le versioni di GCC e OpenMP sopra citate.

I test sono stati effettuati su un'installazione pulita del sistema operativo Ubuntu 20.04 LTS. I test sono successivamente verificati anche su Windows 10, per assicurare la portabilità del programma e controllare la presenza di eventuali *outlier*[1].

Tutti i test sono stati effettuati sul programma compilato con il seguente parametro di GCC: `-O0`. Questo parametro disabilita l'ottimizzazione del compilatore, permettendo di analizzare le prestazioni del programma senza che l'implementazione di GCC influenzi i risultati. Ulteriori parametri e istruzioni per la compilazione sono riportati nel file `README.md` del progetto (A.1).

5.1.2 Architettura hardware

I test delle prestazioni sono stati effettuati su due macchine differenti. La prima è un computer desktop con processore *Intel Core i7-9700K* a 8 *core* e 8 *thread*, con 32 GB di RAM. La seconda è un server con processore *Intel Xeon E5-2603 v4* a 6 *core* e 6 *thread*, con tecnologia Intel Hyper-Threading[2] e 62 GB di RAM. Nella tesi vengono riportati i risultati ottenuti esclusivamente sul computer desktop, in quanto i risultati ottenuti sul server sono simili e non portano a conclusioni significativamente diverse.

5.2 Profilazione del codice

Durante lo sviluppo del programma ho utilizzato alcuni strumenti per profilare il codice e individuare le parti che richiedevano ottimizzazione.

Inizialmente ho utilizzato il comando `time` per misurare il tempo d'esecuzione totale del programma. Questo comando è utile per avere un'idea generale delle prestazioni. Offre un riscontro immediato sull'efficacia delle ottimizzazioni apportate, anche se non permette di individuare eventuali "colli di bottiglia".

[1] In statistica, un *outlier* è un valore che si discosta significativamente dagli altri valori di un insieme di dati. Nel contesto delle prestazioni di un programma, può essere un tempo d'esecuzione molto più alto o molto più basso rispetto agli altri.

[2] Intel Hyper-Threading Technology (HTT) è una tecnologia sviluppata da Intel che permette a un singolo processore di eseguire più thread contemporaneamente.

Per questo motivo ho utilizzato il comando `gprof`, un *profiler* per programmi scritti in C e C++. `gprof` permette di misurare il tempo d'esecuzione di ogni funzione del programma e di individuare quelle che ne occupano di più. Questo strumento serve esattamente per individuare le funzioni da ottimizzare.

Infine ho sfruttato alcuni programmi open source per estrapolare una rappresentazione grafica, come in figura 5.1, dai dati prodotti da `gprof`. Questi programmi sono `gprof2dot` [15] e `dot` [16], che permettono di generare un grafo delle chiamate delle funzioni del programma. Questo grafo evidenzia eventuali problemi di prestazioni, come funzioni che vengono chiamate troppe volte o che richiedono troppo tempo per essere eseguite.

Per misurare i tempi di esecuzione delle sezioni parallele del codice ho utilizzato la funzione `omp_get_wtime()` di OpenMP. Questa funzione restituisce il tempo d'esecuzione in secondi ed è stata inserita prima dell'inizio e dopo la fine di ogni sezione parallela del codice. I tempi di esecuzione sono stati poi stampati a video e salvati su file per essere analizzati successivamente.

Questo è un esempio di codice minimo per misurare il tempo d'esecuzione di una sezione parallela utilizzando `omp_get_wtime()`:

```
1  #include <omp.h>
2  #include <stdio.h>
3
4  int main() {
5      double start_time = omp_get_wtime();
6      #pragma omp parallel {
7          // Sezione parallela
8      }
9      double end_time = omp_get_wtime();
10     printf("Tempo d'esecuzione: %f secondi\n", end_time - start_time);
11     return 0;
12 }
```

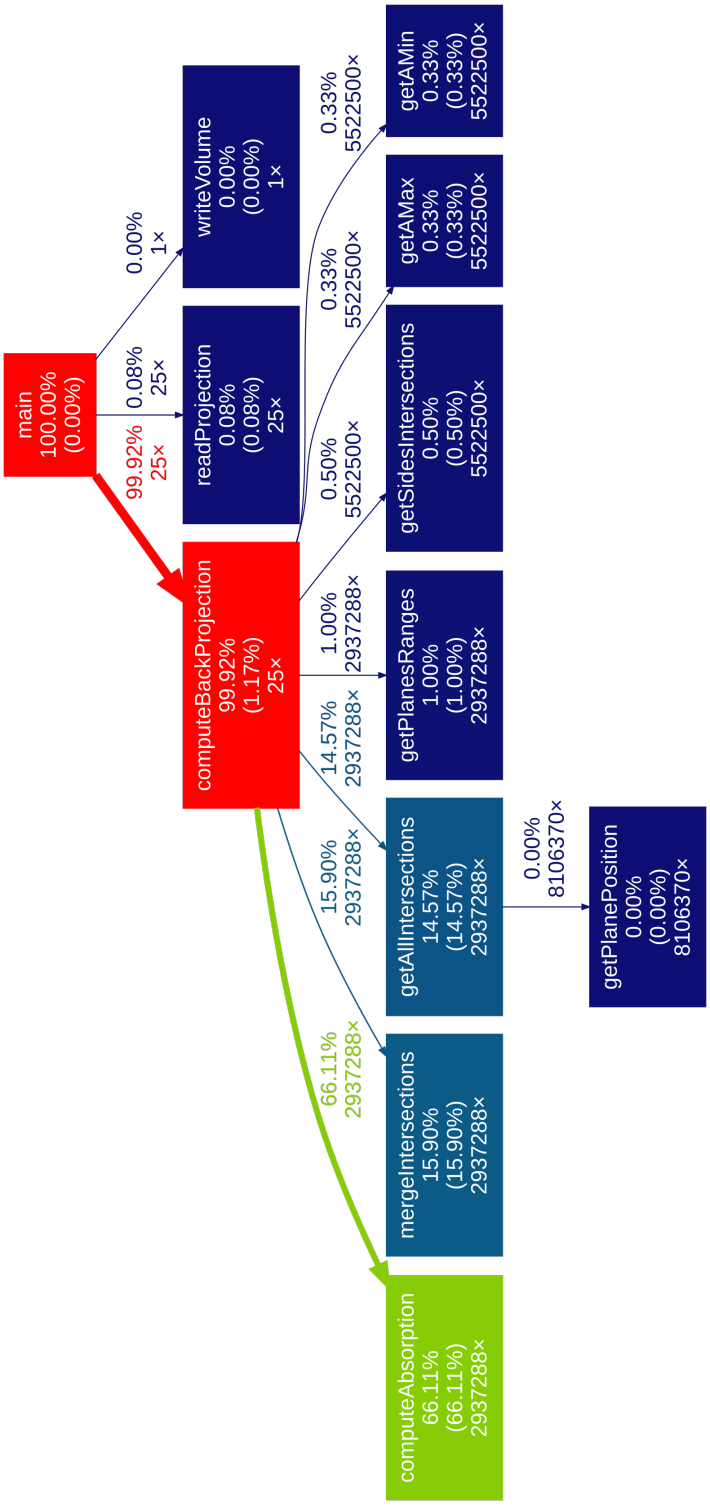


Figura 5.1: Grafo delle chiamate delle funzioni del programma generato da `dot`.

5.3 Metodologia di valutazione delle prestazioni

Per valutare le prestazioni del programma ho utilizzato tre metriche principali: *speedup*, *strong scaling* e *weak scaling*. Queste metriche permettono di valutare l'efficienza del programma in relazione al numero di *thread* utilizzati e alla dimensione dell'input.

In particolare, definiamo i seguenti parametri:

- p: numero di *thread* utilizzati.
- n: dimensione del problema, ovvero il numero di unità di lavoro da svolgere.
- T_{seq}: tempo d'esecuzione del programma sequenziale;
- $T_{par}(p)$: tempo d'esecuzione del programma parallelo usando p *thread*;

Con questi parametri possiamo definire le formule per calcolare lo *speedup*, lo *strong scaling* e il *weak scaling*:

- *Speedup*: misura dell'efficienza di un programma parallelo rispetto a un programma sequenziale. È definito come il rapporto tra il tempo d'esecuzione del programma sequenziale e il tempo d'esecuzione del programma parallelo.

  La formula per calcolare lo speedup è la seguente:

  $$S(p) = \frac{T_{seq}}{T_{par}(p)} \approx \frac{T_{par}(1)}{T_{par}(p)} \tag{5.1}$$

  Notare che a causa del costo computazionale della creazione dei *thread* o della sincronizzazione, è più corretto considerare il tempo d'esecuzione del programma sequenziale come tempo del programma parallelo con un solo thread.

- *Strong scaling*: misura dell'efficienza di un programma parallelo rispetto al numero di thread usati, mantenendo la dimensione del problema costante. È definito come il rapporto tra lo speedup e il numero di processori utilizzati.

  La formula per calcolare lo strong scaling è la seguente:

  $$E(p) = \frac{S(p)}{p} \tag{5.2}$$

- *Weak scaling*: misura dell'efficienza di un programma parallelo all'aumento proporzionale di thread e dimensione del problema.

  La formula per calcolare il weak scaling è la seguente:

  $$W(p) = \frac{T_1(n)}{T_p(n \times p)} \tag{5.3}$$

  Dove:

- $T_1(n)$ è il tempo d'esecuzione del programma parallelo con un solo processore e dimensione del problema n.
- $T_p(n \times p)$ è il tempo d'esecuzione del programma parallelo con p processori e dimensione del problema $n \times p$.

In questo calcolo è importante che la dimensione del problema cresca proporzionalmente al numero di processori utilizzati. In modo che il lavoro svolto da ogni processore rimanga costante.

Nel problema del calcolo del percorso radiologico, la dimensione del problema corrisponde alla grandezza del volume tridimensionale. Come abbiamo visto nei capitoli precedenti, grazie all'algoritmo di Siddon, la complessità computazionale cresce linearmente con il numero di voxel n per lato del volume cubico.

5.4 Risultati sperimentali

Per raccogliere i dati, il programma viene eseguito più volte con l'aiuto di uno *script*, creato appositamente per automatizzare i test delle prestazioni e raccogliere i tempi di esecuzione.

Lo *script* esegue il programma modificando questi parametri:

- il numero di processori utilizzati, da 1 a 8;
- la dimensione del problema, da 100 a 800 voxel (per lato del volume); 100 voxel corrispondono a 235 unità di lavoro.

Questo intervallo di valori è scelto in modo da coprire un'ampia gamma di casi e facilitare il calcolo delle prestazioni del programma. I limiti superiori sono stati scelti in modo da permettere di eseguire i test in un tempo ragionevole.

Per ogni parametro modificato vengono eseguite tre ripetizioni e i risultati vengono mediati per ridurre l'incertezza dei dati. I dati mostrati sono le medie delle tre ripetizioni, arrotondati a tre cifre decimali. Tutti i dati grezzi sono invece disponibili nella repository del progetto (A.1). I tempi di esecuzione sono espressi in secondi.

Una volta raccolti i tempi di esecuzione, lo *script* calcola *speedup*, *strong scaling* e *weak scaling* utilizzando le formule matematiche descritte in precedenza (sezione 5.3). Il risultato dei calcoli è poi salvato su file per ulteriore elaborazione e per la creazione dei grafici. Mostrato nella sezione successiva.

Questa procedura si ripete per ogni strategia di parallelizzazione implementata nel programma. I risultati sono riportati in tabelle e grafici per facilitarne la lettura e comprensione.

Tempi di esecuzione: Intersezioni

Partiamo dalla strategia di parallelizzazione per intersezioni. I tempi d'esecuzione raccolti sono riportati nella tabella 5.1 qui sotto.

Thread / Unità di lavoro	1	2	4	6	8
235	10.471	11.309	10.726	10.737	11.573
470	17.371	16.154	14.481	14.168	15.156
940	33.708	26.995	22.787	21.805	22.813
1410	55.156	40.699	32.277	30.539	31.357
1880	83.413	59.945	46.498	43.418	44.005

Tabella 5.1: Tempi di esecuzione raccolti dai test effettuati con la strategia di parallelizzazione per intersezioni.

Tempi di esecuzione: Pixel

A seguire, i dati raccolti usando la strategia di parallelizzazione per pixel, riportati nella tabella 5.2.

Thread / Unità di lavoro	1	2	4	6	8
235	8.561	4.290	2.240	1.514	1.151
470	17.442	8.774	4.569	3.088	2.351
940	37.634	19.319	10.379	7.194	5.543
1410	63.651	33.263	18.035	12.535	9.684
1880	92.561	48.130	26.587	18.583	14.508

Tabella 5.2: Tempi di esecuzione raccolti dai test effettuati con la strategia di parallelizzazione per pixel.

Tempi di esecuzione: Proiezioni

Infine, i dati raccolti usando la strategia di parallelizzazione per proiezioni, riportati nella tabella 5.3.

Unità di lavoro \ Thread	1	2	4	6	8
235	8.448	4.235	2.210	1.415	1.139
470	17.112	8.696	4.594	2.999	2.432
940	36.811	19.046	10.379	6.687	5.718
1410	64.367	33.541	18.030	11.566	9.910
1880	91.379	47.789	26.102	16.831	14.193

Tabella 5.3: Tempi di esecuzione raccolti dai test effettuati con la strategia di parallelizzazione per proiezioni.

Ora che abbiamo visto tutti i dati raccolti, passiamo alle metriche di valutazione delle prestazioni. Nelle tre sezioni successive, verranno presentati i risultati dei calcoli di *speedup*, *strong scaling* e *weak scaling* per ogni strategia di parallelizzazione implementata nel programma e per ogni unità di lavoro nel set di dati.

Un ulteriore *script* è stato utilizzato per calcolare queste metriche e generare i grafici dei risultati. I grafici sono stati creati utilizzando il software open source `matplotlib` [17] per Python[3].

[3]Python è un linguaggio di programmazione interpretato, orientato alla programmazione ad oggetti, di alto livello. Le sue strutture dati di alto livello, combinate alla tipizzazione dinamica, lo rendono molto interessante per lo sviluppo rapido di applicazioni.

5.4.1 Speedup

Speedup: Intersezioni

Seguendo lo stesso ordine di prima, partiamo con il calcolo dello *speedup* per la strategia di parallelizzazione per intersezioni. I risultati sono riportati in tabella 5.4 e grafico 5.2.

Unità di lavoro \ Thread	1	2	4	6	8
235	1.000	0.926	0.976	0.975	0.905
470	1.000	1.075	1.200	1.226	1.146
940	1.000	1.249	1.479	1.546	1.478
1410	1.000	1.355	1.709	1.806	1.759
1880	1.000	1.391	1.794	1.921	1.896

Tabella 5.4: Risultati dei calcoli dello *speedup* per parallelizzazione per intersezioni.

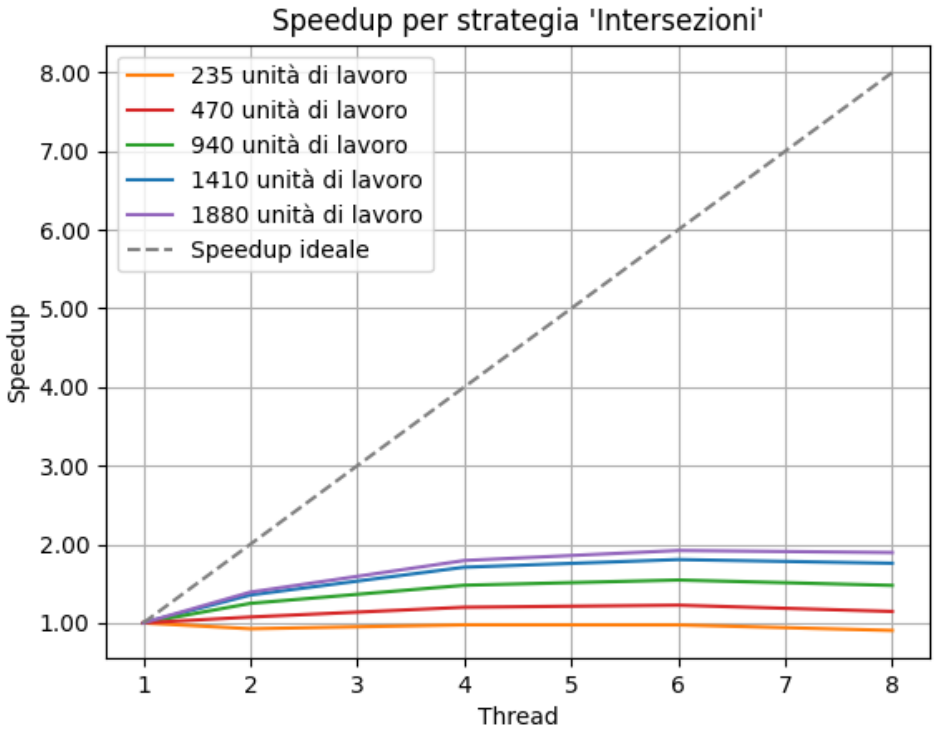


Figura 5.2: Grafico dello *speedup* della strategia di parallelizzazione per intersezioni.

Speedup: Pixel

Ora calcoliamo lo *speedup* per la strategia di parallelizzazione per pixel. I risultati sono riportati in tabella 5.5 e grafico 5.3.

Unità di lavoro \ Thread	1	2	4	6	8
235	1.000	1.995	3.822	5.656	7.436
470	1.000	1.988	3.817	5.647	7.418
940	1.000	1.948	3.626	5.231	6.790
1410	1.000	1.914	3.529	5.078	6.573
1880	1.000	1.923	3.481	4.981	6.380

Tabella 5.5: Risultati dei calcoli dello *speedup* per parallelizzazione per pixel.

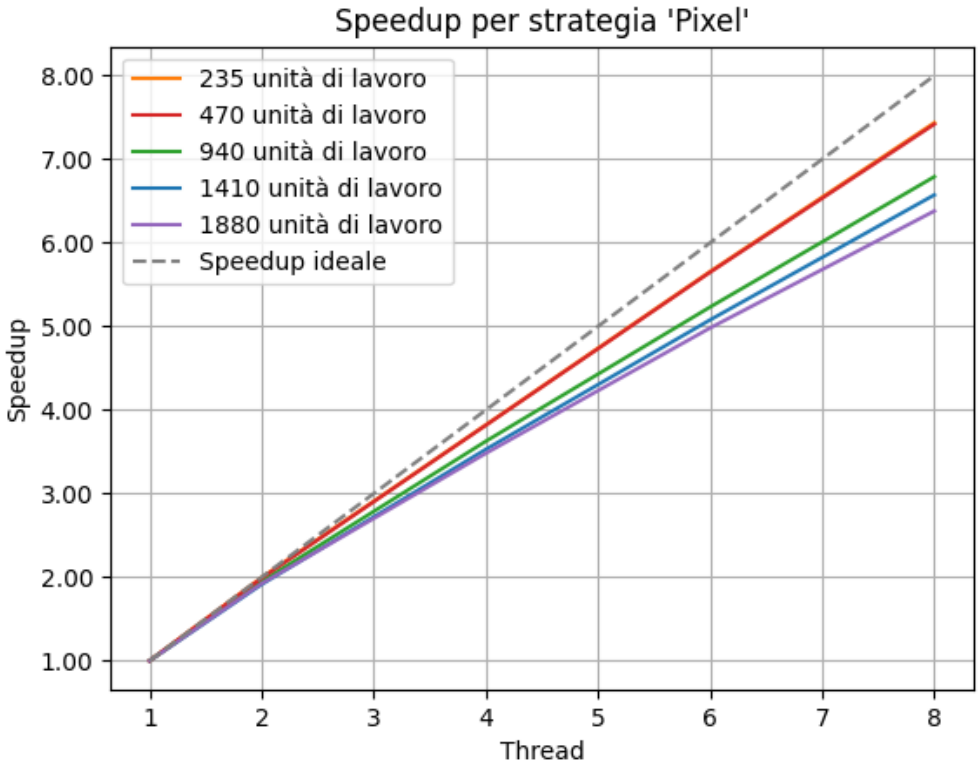


Figura 5.3: Grafico dello *speedup* della strategia di parallelizzazione per pixel.

Speedup: Proiezioni

Infine, calcoliamo lo *speedup* per la strategia di parallelizzazione per proiezioni. I risultati sono riportati in tabella 5.6 e grafico 5.4.

Thread / Unità di lavoro	1	2	4	6	8
235	1.000	1.995	3.822	5.969	7.414
470	1.000	1.968	3.725	5.707	7.035
940	1.000	1.933	3.547	5.504	6.438
1410	1.000	1.919	3.570	5.565	6.495
1880	1.000	1.912	3.501	5.429	6.439

Tabella 5.6: Risultati dei calcoli dello *speedup* per parallelizzazione per proiezioni.

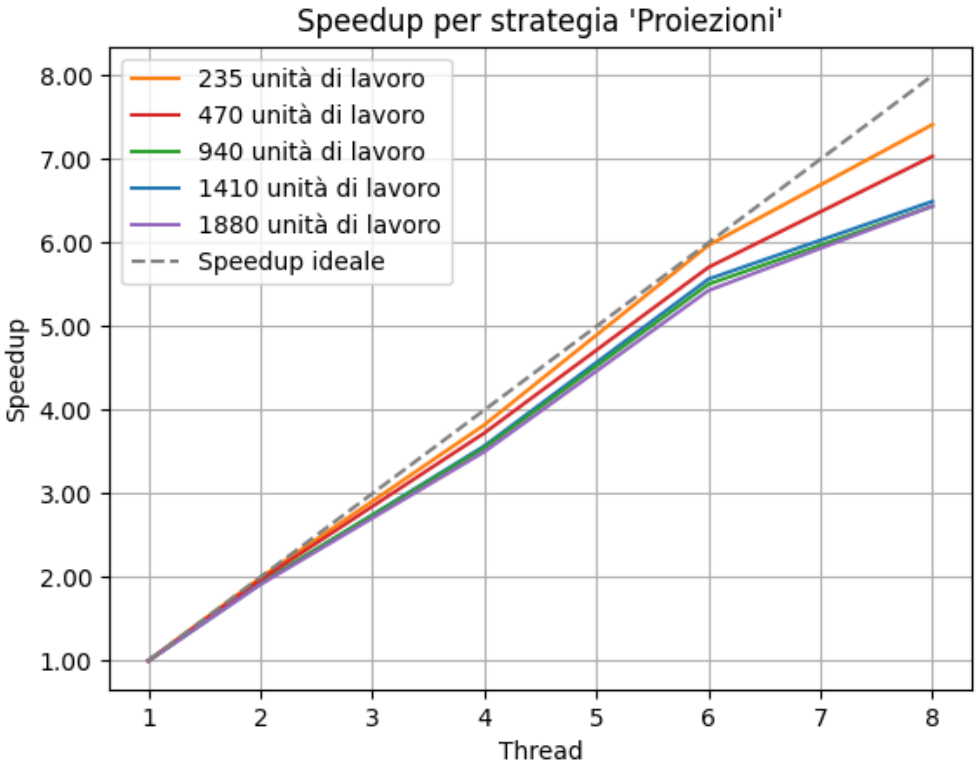


Figura 5.4: Grafico dello *speedup* della strategia di parallelizzazione per proiezioni.

5.4.2 Strong scaling

Strong scaling: Intersezioni

Passiamo ora dallo *speedup* allo *strong scaling*. Iniziamo con la strategia di parallelizzazione per intersezioni. I risultati sono riportati in tabella 5.7 e grafico 5.5.

Unità di lavoro \ Thread	1	2	4	6	8
235	1.000	0.463	0.244	0.163	0.113
470	1.000	0.538	0.300	0.204	0.143
940	1.000	0.624	0.370	0.258	0.185
1410	1.000	0.678	0.427	0.301	0.220
1880	1.000	0.696	0.448	0.320	0.237

Tabella 5.7: Risultati dei calcoli dello *strong scaling* per parallelizzazione per intersezioni.

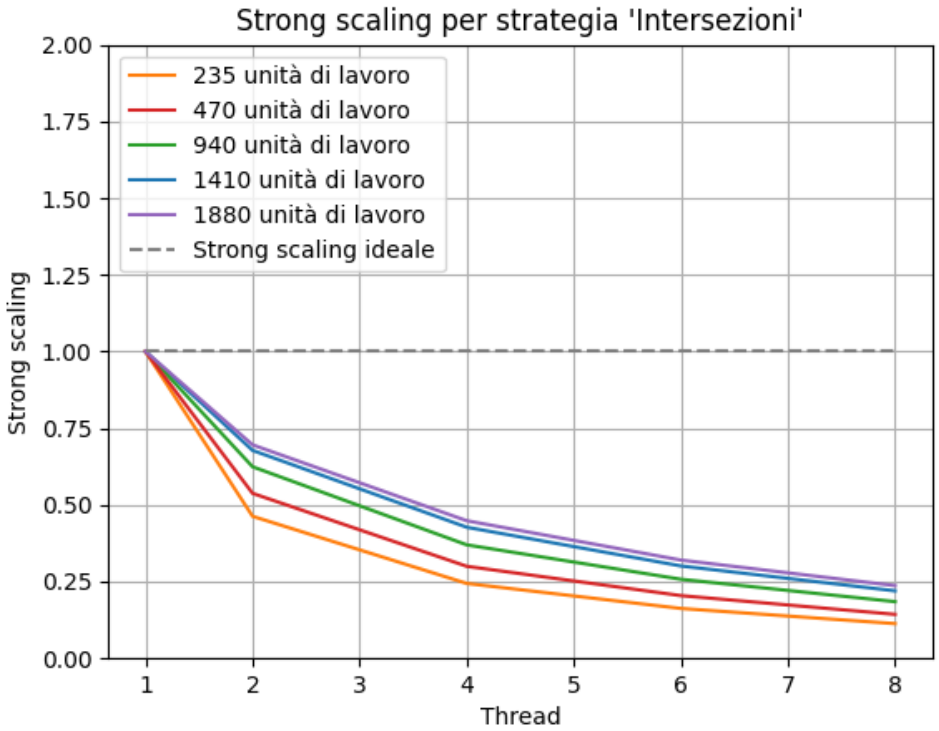


Figura 5.5: Grafico dello *strong scaling* della strategia di parallelizzazione per intersezioni.

Strong scaling: Pixel

Consideriamo di nuovo la strategia di parallelizzazione per pixel. I risultati sono riportati in tabella 5.8 e grafico 5.6.

Thread / Unità di lavoro	1	2	4	6	8
235	1.000	0.998	0.956	0.943	0.930
470	1.000	0.994	0.954	0.941	0.927
940	1.000	0.974	0.906	0.872	0.849
1410	1.000	0.957	0.882	0.846	0.822
1880	1.000	0.962	0.870	0.830	0.797

Tabella 5.8: Risultati dei calcoli dello *strong scaling* per parallelizzazione per pixel.

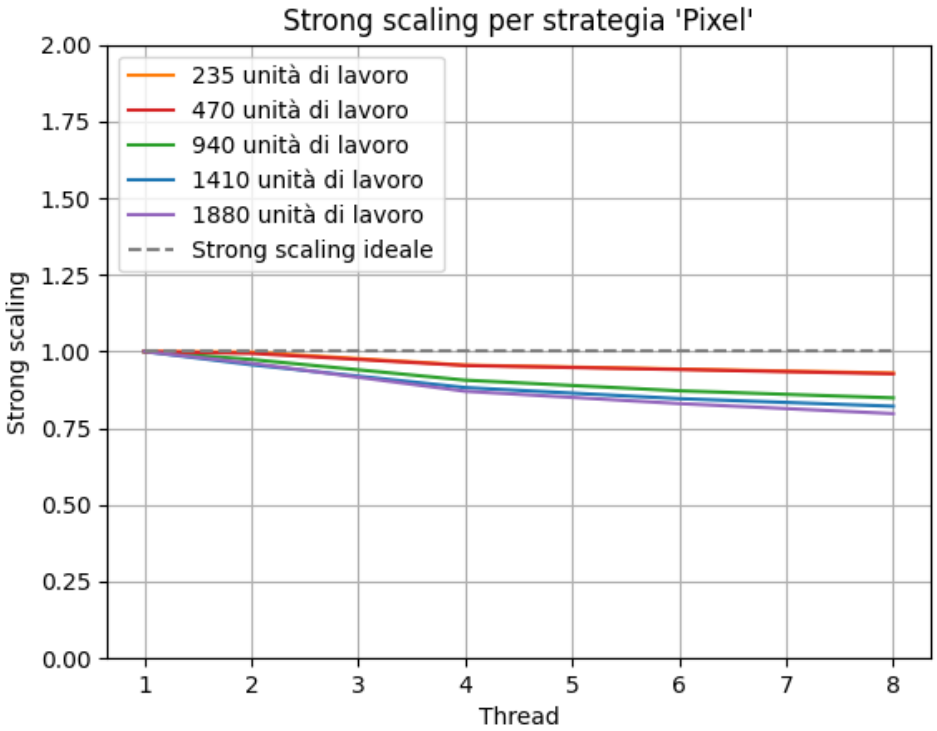


Figura 5.6: Grafico dello *strong scaling* della strategia di parallelizzazione per pixel.

Strong scaling: Proiezioni

Ed infine, la strategia di parallelizzazione per proiezioni. I risultati sono riportati in tabella 5.9 e grafico 5.7.

Unità di lavoro \ Thread	1	2	4	6	8
235	1.000	0.997	0.956	0.995	0.927
470	1.000	0.984	0.931	0.951	0.879
940	1.000	0.966	0.887	0.917	0.805
1410	1.000	0.960	0.892	0.928	0.812
1880	1.000	0.956	0.875	0.905	0.805

Tabella 5.9: Risultati dei calcoli dello *strong scaling* per parallelizzazione per proiezioni.

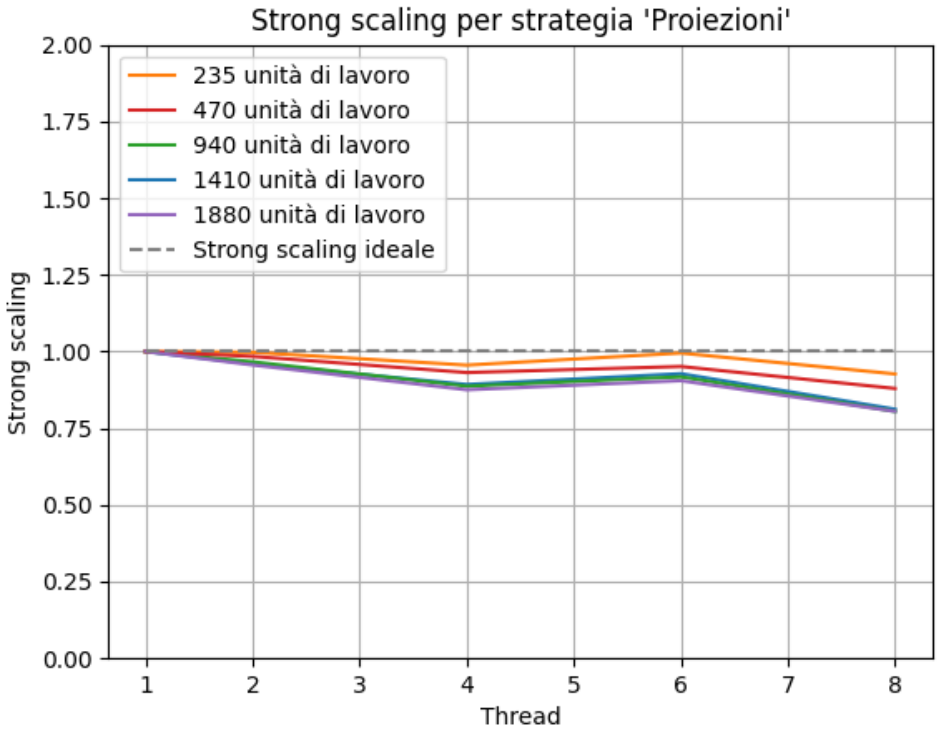


Figura 5.7: Grafico dello *strong scaling* della strategia di parallelizzazione per proiezioni.

5.4.3 Weak scaling

Weak scaling: Intersezioni

Come ultima metrica, calcoliamo il *weak scaling*. Iniziamo con la strategia di parallelizzazione per intersezioni. I risultati sono riportati in tabella 5.10 e grafico 5.8.

Thread / Unità di lavoro	1	2	4	6	8
235	1.000	/	/	/	/
470	/	0.648	/	/	/
940	/	/	0.460	/	/
1410	/	/	/	0.343	/
1880	/	/	/	/	0.238

Tabella 5.10: Risultati dei calcoli del *weak scaling* per parallelizzazione per intersezioni.

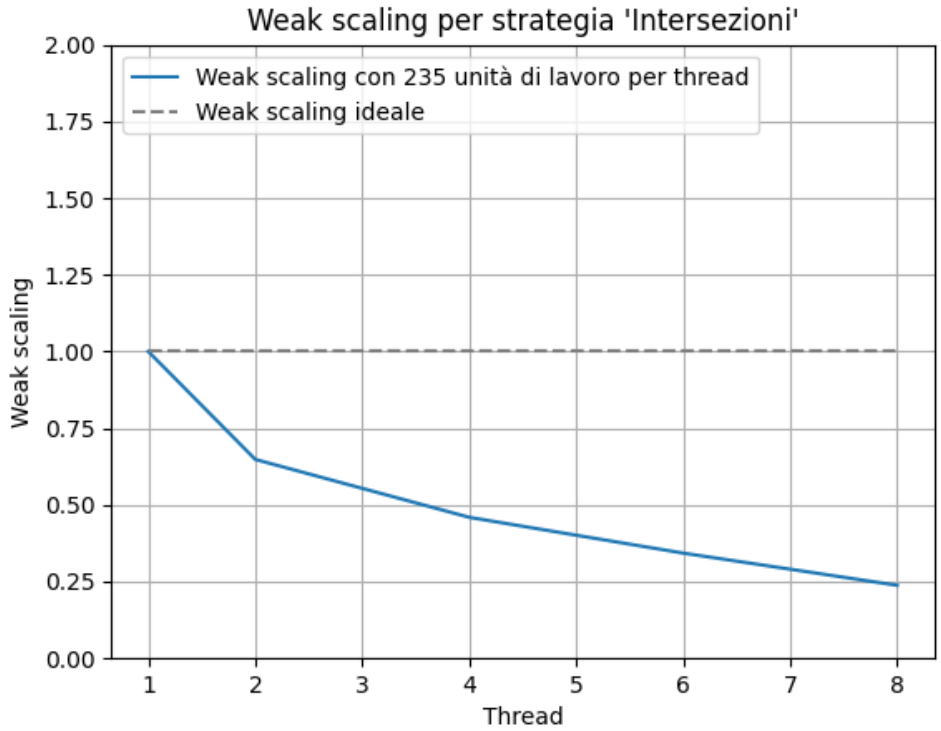


Figura 5.8: Grafico del *weak scaling* della strategia di parallelizzazione per intersezioni.

Weak scaling: Pixel

Passiamo alla strategia di parallelizzazione per pixel. I risultati sono riportati in tabella 5.11 e grafico 5.9.

Unità di lavoro \ Thread	1	2	4	6	8
235	1.000	/	/	/	/
470	/	0.976	/	/	/
940	/	/	0.825	/	/
1410	/	/	/	0.683	/
1880	/	/	/	/	0.590

Tabella 5.11: Risultati dei calcoli del *weak scaling* per parallelizzazione per pixel.

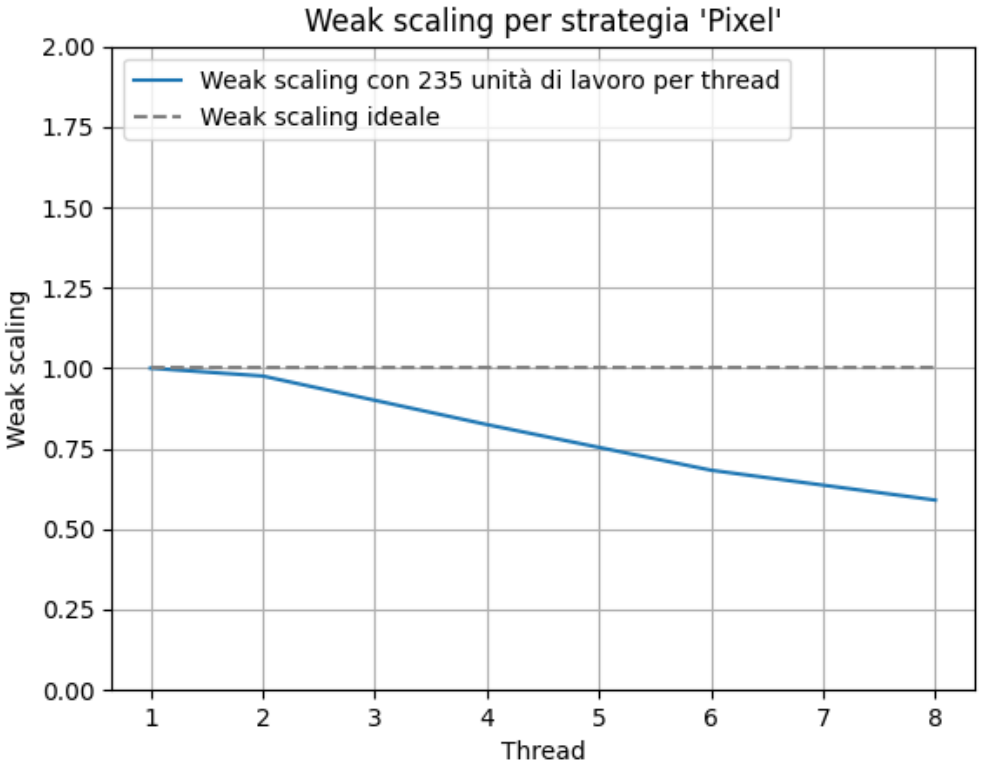


Figura 5.9: Grafico del *weak scaling* della strategia di parallelizzazione per pixel.

Weak scaling: Proiezioni

Per concludere il capitolo manca solo il calcolo del *weak scaling* per la strategia di parallelizzazione per proiezioni. I risultati sono riportati in tabella 5.12 e grafico 5.10.

Thread / Unità di lavoro	1	2	4	6	8
235	1.000	/	/	/	/
470	/	0.971	/	/	/
940	/	/	0.814	/	/
1410	/	/	/	0.730	/
1880	/	/	/	/	0.596

Tabella 5.12: Risultati dei calcoli del *weak scaling* per parallelizzazione per proiezioni.

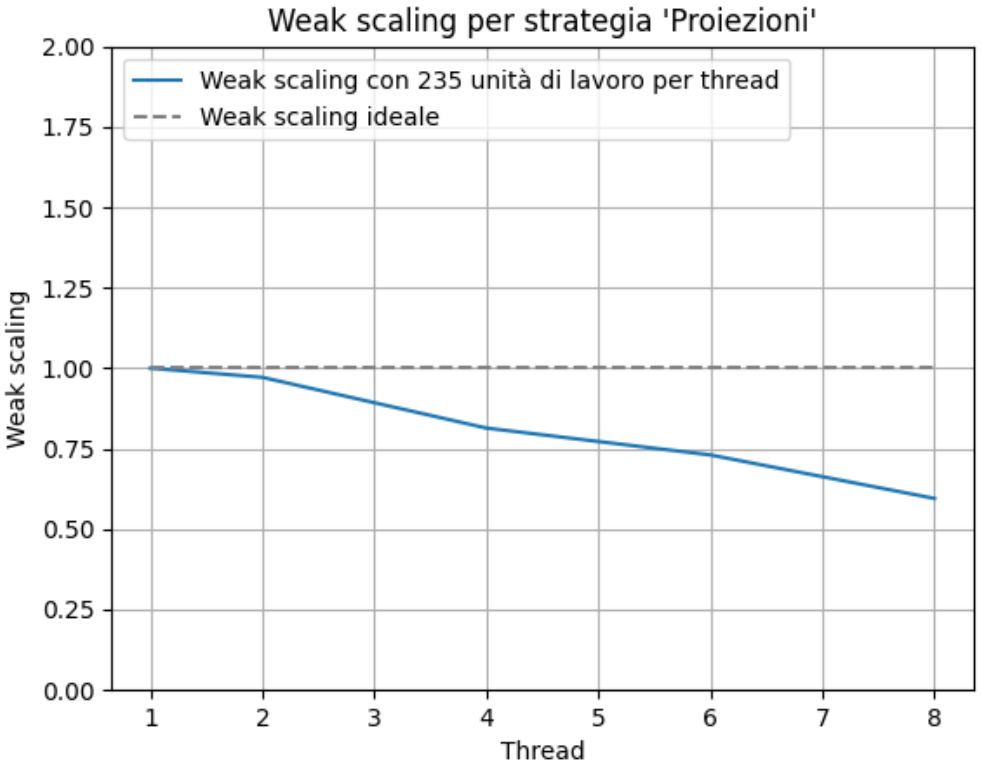


Figura 5.10: Grafico del *weak scaling* della strategia di parallelizzazione per proiezioni.

5.5 Prestazioni del programma

In questa sezione, vengono presentati i risultati della precedente analisi delle prestazioni del programma nelle diverse strategie di parallelizzazione implementate.

5.5.1 Prima strategia

La prima tecnica di parallelizzazione, basata sulla distribuzione delle intersezioni dei raggi, non mostra risultati particolarmente efficaci. Dal grafico dello *speedup* (fig. 5.2), si può osservare che la curva si mantiene piatta e incrementa molto lentamente, fino a stabilizzarsi intorno a un valore massimo di due. Nonostante il risultato deludente, possiamo notare che l'efficacia rispetta le predizioni fatte. A contrario delle altre tecniche, questa tende ad essere più efficiente all'aumentare delle unità di lavoro. Tuttavia, anche lo *strong scaling* e il *weak scaling* risultano insoddisfacenti (fig. 5.5 e 5.8); con prestazioni che, in alcuni test, regrediscono rispetto alla versione sequenziale.

5.5.2 Seconda e terza strategia

Le altre due tecniche, basate sulla parallelizzazione dei pixel e delle proiezioni, hanno mostrato invece un comportamento migliore, con risultati vicini alle aspettative. Entrambe le strategie offrono uno *speedup* quasi lineare, con un leggero calo delle prestazioni nel momento in cui si arriva a utilizzare otto processi. Il risultato dello *strong scaling* è anch'esso positivo e decisamente superiore rispetto alla tecnica precedente.

Un aspetto non positivo, presente in tutte e tre le tecniche di parallelizzazione, riguarda il *weak scaling*: in tutti i casi, questa metrica tende a diminuire significativamente con l'aumentare del numero di processi.

Una soluzione per migliorare questo aspetto potrebbe essere l'introduzione di una strategia ibrida, come menzionato nel capitolo precedente, che combini le migliori caratteristiche delle tecniche di parallelizzazione per pixel e per proiezioni, mirando a migliorare anche il *weak scaling* finale.

5.5.3 Considerazioni generali

Nel complesso, il programma mostra buoni risultati in termini di prestazioni. Anche con un aumento considerevole del carico di lavoro, ben oltre quello usato per i test, il programma riesce a completare i calcoli in tempi ragionevoli, un obiettivo irraggiungibile nella versione seriale. Questo dimostra l'efficacia delle tecniche di parallelizzazione applicate, seppur con qualche margine di miglioramento.

Capitolo 6

Analisi dei risultati ottenuti

6.1 Output del programma

Dopo aver completato il processo di retroproiezione, il programma è in grado di salvare i dati dei voxel ricostruiti in due diversi formati di file: `.nrrd` e `.raw`. Questi formati permettono di esportare i dati in modalità compatibili con software di visualizzazione dedicati, rendendo possibile l'analisi dei risultati della ricostruzione all'utente finale.

- Il formato `.nrrd` è utilizzato per ottenere una visualizzazione volumetrica di dati;
- Il formato `.raw`, invece, essendo un formato di file immagine, conterrà sezioni bidimensionali del volume ricostruito.

6.1.1 File tridimensionale

Il formato `.nrrd` (Nearly Raw Raster Data) [18] è un formato flessibile usato per rappresentare dati multidimensionali in contesti scientifici e medici, in questo caso immagini 3D. Strutturato per essere facilmente leggibile e modificabile, `.nrrd` prevede un file principale che contiene metadati descrittivi (ad esempio, dimensioni, spaziatura e unità dei voxel) e opzionalmente anche un file dati separato. Supporta sia scrittura in modalità testuale che binaria, e anche la compressione dei dati. Il programma consente all'utente di scegliere il formato preferito tra i due per garantire la massima compatibilità con gli strumenti di visualizzazione.

6.1.2 File immagine

Il formato `.raw` rappresenta un file binario grezzo di dati dei pixel che compongo le immagini, privo di informazioni aggiuntive come le dimensioni delle immagini o il tipo di dati. Poiché non contiene metadati, è necessario che l'utente specifichi manualmente le dimensioni dell'immagine per visualizzarla correttamente. Il programma esporta i dati `.raw` in formato binario, fornendo inoltre i parametri essenziali per la visualizzazione. Questo semplice approccio rende l'output compatibile con un'ampia gamma di software di visualizzazione, anche quelli più datati.

6.2 Visualizzazione dei risultati

Questi due formati sono stati scelti per la loro particolare semplicità sia implementativa che di utilizzo.

Visualizzazione del formato .nrrd

Per i file in formato `.nrrd`, è possibile utilizzare software come *ITK/VTK viewer* [19], un software open source per la visualizzazione di immagini scientifiche, che supporta vari formati, tra cui `.nrrd`. Questo strumento consente di visualizzare e manipolare i dati in modo interattivo, offrendo funzionalità avanzate come:

- Regolazione della trasparenza, luminosità e contrasto;
- Rotazione, traslazione e zoom;
- Filtri per migliorare la qualità dell'immagine;
- Sovrapposizione di più immagini;
- Visualizzazione di sezioni 2D (di tutti gli assi).

ITK/VTK viewer dispone anche di un'interfaccia web gratuita, che permette di visualizzare i file `.nrrd` direttamente dal browser. Inoltre è possibile integrarlo in pagine web preesistenti.

In figura 6.1 è mostrata una schermata di *ITK/VTK viewer* con un file `.nrrd` caricato. Il modello 3D visualizzato rappresenta la ricostruzione di un cubo con un foro al suo interno, ottenuto con il programma sviluppato.

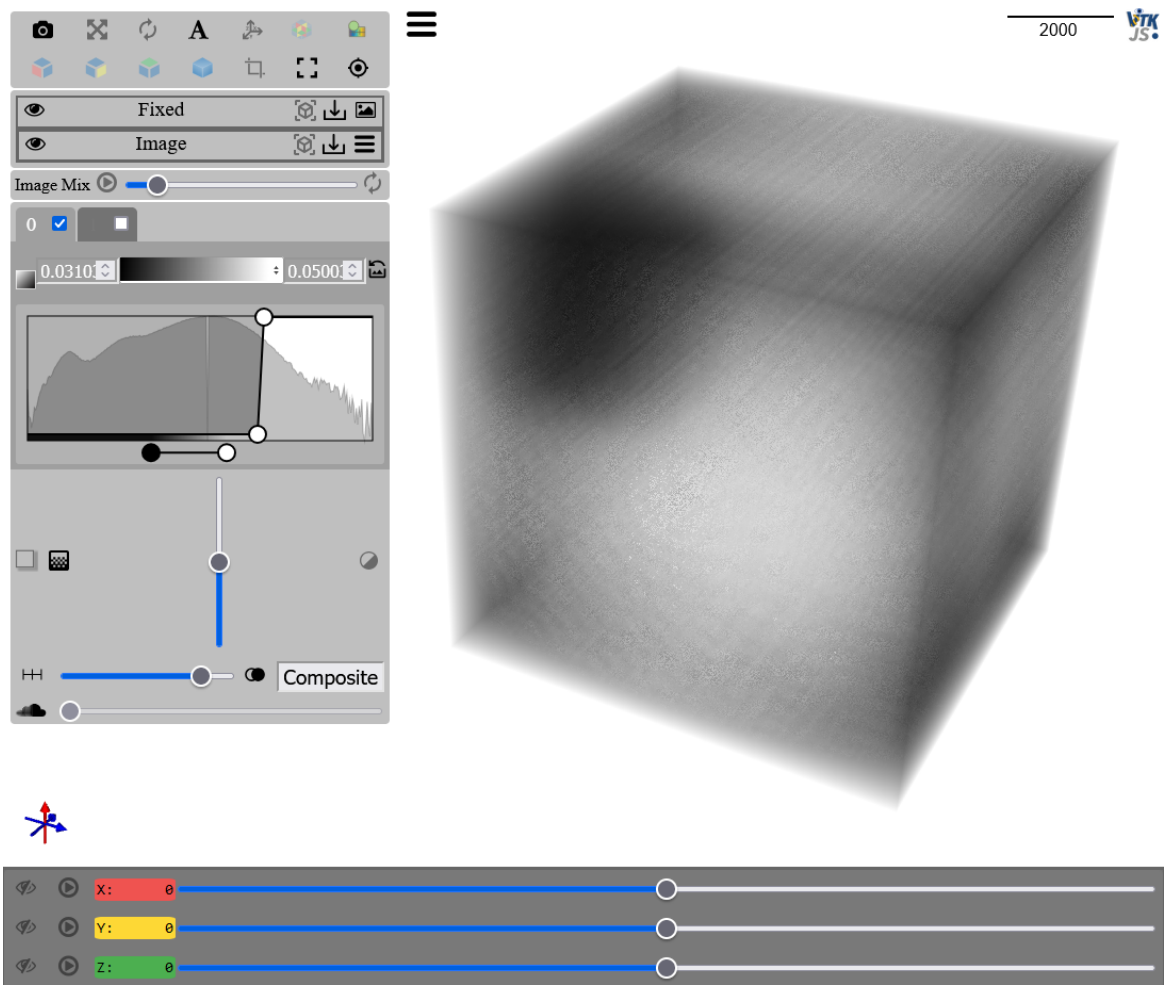


Figura 6.1: Interfaccia di *ITK/VTK viewer* con un file `.nrrd` caricato

Visualizzazione del formato .raw

Per i file `.raw`, invece, è consigliato l'uso di *ImageJ* [20], un software di pubblico dominio specializzato nella visualizzazione di immagini. Non dispone di molte delle funzionalità avanzate di ITK/VTK viewer, ma è molto semplice da usare, ha alte prestazioni, e offre una compatibilità universale grazie al fatto che è scritto in Java. [1]

In figura 6.2 è mostrata una schermata di *ImageJ* con un file `.raw` caricato. Il modello tridimensionale visualizzato è lo stesso del precedente, ma in questo caso è rappresentato come un'immagine 2D. Sono state scelte porzioni del volume per mostrare le sezioni che presentano il foro all'interno del cubo.

[1] Java è un linguaggio di programmazione ad alto livello, orientato agli oggetti e a tipizzazione statica, specificatamente progettato per essere il più possibile indipendente dalla piattaforma di esecuzione.

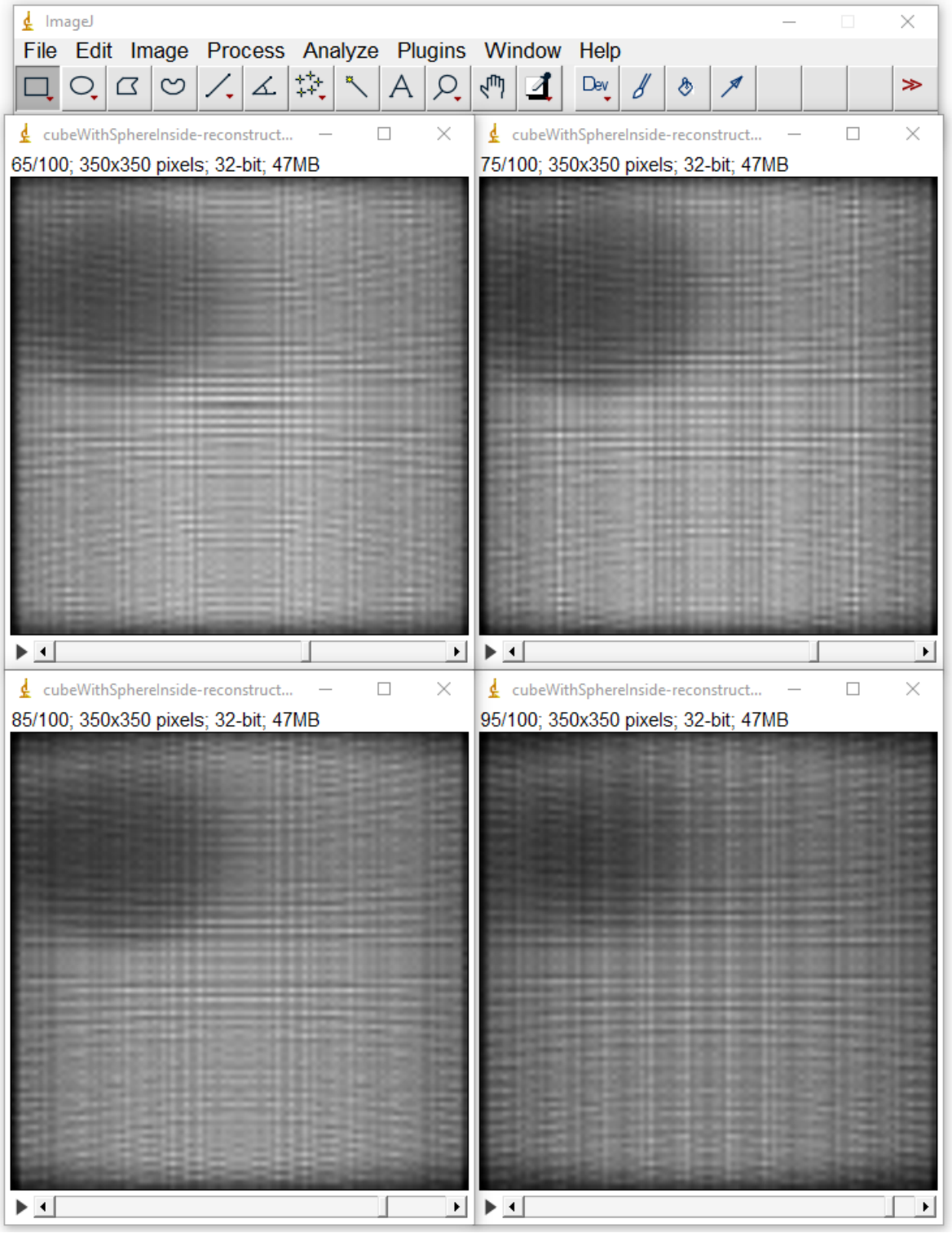


Figura 6.2: Interfaccia di *ImageJ* con un file `.raw` caricato, visualizzato in 4 diverse sezioni

6.3 Verifica dell'accuratezza

ITK/VTK viewer offre una funzionalità di confronto tra file multipli, che risulta utile per verificare la qualità delle ricostruzioni. Grazie a strumenti di miscelazione e sovrapposizione, è possibile confrontare il risultato della ricostruzione del volume con una simulazione di un fantoccio di riferimento. Inoltre, è possibile rappresentare i due volumi con colori e trasparenze diverse, mettendo in evidenza le differenze in modo visivo.

Questo tipo di verifica consente di valutare la fedeltà della ricostruzione, facilitando l'identificazione di eventuali errori o aree di miglioramento. Tra gli sviluppi futuri (sez.7.2.4), c'è proprio un caso di errore minore identificato con questa tecnica.

Possiamo vedere un esempio di confronto tra il cubo ricostruito, mostrato precedentemente in figura 6.1, e il suo modello di riferimento in figura 6.3. L'interfaccia consente di sfumare dinamicamente tra i due modelli, ma per evidenziare le differenze in un fermo immagine, il modello di riferimento è stato colorato con un contrasto elevato.

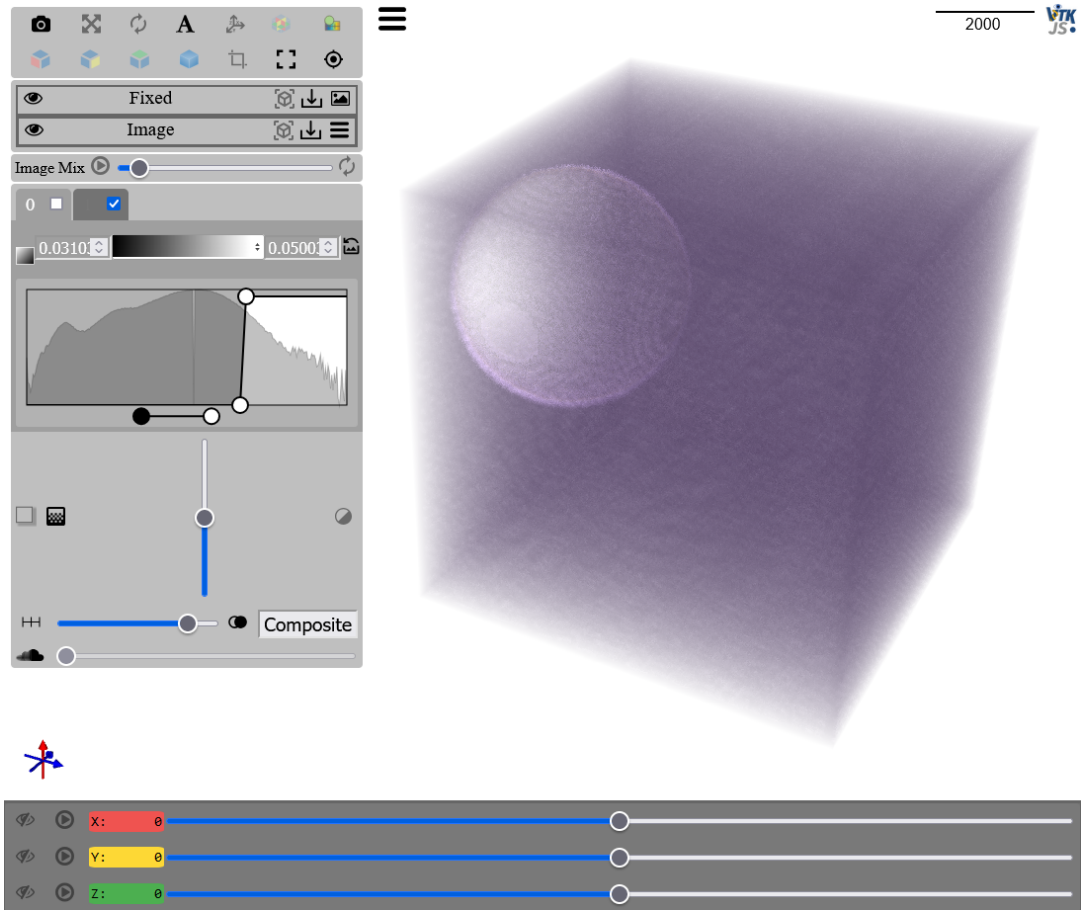


Figura 6.3: Modello di riferimento del cubo con foro al suo interno

Capitolo 7

Conclusioni e sviluppi futuri

7.1 Conclusione e riflessioni

In conclusione, gli obiettivi prefissati all'inizio del progetto sono stati raggiunti con successo, realizzando un programma ad alte prestazioni con OpenMP. Questo lavoro mi ha permesso di applicare in modo concreto le conoscenze acquisite durante il mio percorso universitario, trasformandole in questo progetto completo e approfondito. Spero che questo progetto possa essere utile per future applicazioni pratiche o fare da base per ulteriori sviluppi, o magari portato avanti da altri studenti o ricercatori.

7.2 Sviluppi futuri

In quest'ultima sezione vengono proposte alcune idee di sviluppo per migliorare l'accuratezza, le prestazioni e la flessibilità del programma. Queste proposte includono soluzioni che, per complessità o limiti temporali, non sono state implementate, ma potrebbero aumentare significativamente la qualità complessiva del software.

7.2.1 Implementazione alternativa dell'algoritmo di Siddon

Negli anni successivi alla pubblicazione dell'algoritmo di Siddon, sono state proposte diverse varianti e ottimizzazioni che potrebbero migliorare le prestazioni del programma. Un esempio è quello proposto nel documento "*A new algorithm for calculating the radiological path in CT image reconstruction*" [21].

Questo algoritmo accelera il calcolo del percorso radiologico rispetto a quello di Siddon, sfruttando la successione degli indici dei voxel intersecati dal raggio e le coordinate delle intersezioni. Dopo aver determinato il primo voxel attraversato dal raggio, l'algoritmo usa

l'incremento delle coordinate tra le intersezioni successive per individuare rapidamente gli indici dei voxel seguenti. Questo approccio riduce i calcoli richiesti per ogni intersezione, semplificando i passaggi necessari e velocizzando significativamente l'intero processo. Implementare questa variante potrebbe portare a uno speedup fino a quattro volte superiore rispetto alla versione originale [21], specialmente per volumi di grandi dimensioni, migliorando l'efficienza computazionale senza stravolgere la struttura del programma.

7.2.2 Implementazione di altri formati per input e output

Attualmente, il programma supporta solo l'output in formato `.nrrd` e `.raw`, come discusso nel capitolo 6.1. Visto che il programma fa uso di moduli separati per la lettura e scrittura dei file, l'aggiunta di nuovi formati di output è triviale.

Implementare ulteriori formati di output, come `.vtk` per i dati volumetrici o `.stl` per le superfici, amplierebbe le possibilità di visualizzazione in altri software di imaging.

L'aggiunta del formato DICOM (`.dcm`) potrebbe essere un'opzione interessante per rendere il programma più versatile nell'ambito medico. DICOM è uno standard diffuso in radiologia e medicina che offre diverse funzionalità, tra cui:

- **Miglioramento della qualità dell'immagine**: i visualizzatori DICOM consentono all'utente di regolare luminosità e contrasto per evidenziare i dettagli delle immagini, facilitandone l'analisi;
- **Ricostruzione 3D**: sebbene le immagini DICOM siano bidimensionali, spesso vengono acquisite da tre piani che permettono di creare visualizzazioni tridimensionali. Ciò è utile per esaminare strutture complesse da angolazioni diverse;
- **Misurazioni**: i visualizzatori DICOM permettono la misurazione accurata di distanze e valori dei modelli in 3D;
- **Comparazione e combinazione delle immagini**: DICOM consente di confrontare immagini di scansioni diverse, per monitorare le differenze tra esami o valutare l'efficacia di trattamenti medici;
- **Archiviazione e compressione**: i file DICOM possono essere compressi per ridurre lo spazio di archiviazione, senza compromettere la qualità dell'immagine. Inoltre, i file DICOM possono essere archiviati come *PACS* (*Picture Archiving and Communication System*), consentendo di accedere alle immagini da altri dispositivi connessi a una rete.

Anche l'input potrebbe essere migliorato con formati standardizzati, dato che attualmente si limita a formati `.pgm` testuali e `.dat` binari che non aderiscono a standard specifici. Utilizzare qualche standard di input permetterebbe di leggere dati provenienti da altri software già in uso.

7.2.3 Aumentare la porzione di codice parallelo

Lettura e scrittura parallele

Rifacendoci alla sezione precedente, esistono formati di input e output strutturati in modo da permettere la lettura e scrittura parallela dei dati. Questi formati non sono semplici da implementare, se ne potrebbe valutare l'utilità in esigenza di file di grandi dimensioni.

Per esempio lo standard `VTK` supporta formati come:

- `.pvti` (*Parallel vtkImageData*): rappresenta una griglia strutturata con voxel uniformi, utile per dati volumetrici come le scansioni CT.
- `.pvts` (*Parallel vtkStructuredGrid*): consente la rappresentazione di geometrie complesse con celle di vari tipi.

Questi formati usano la sintassi `XML` per la descrizione dei dati. Il formato XML di VTK facilita lo streaming dei dati e la gestione di input/output parallelo. Questo approccio offre vari vantaggi, tra cui: supporto per la compressione; e la codifica binaria portatile, fornendo compatibilità con gli ordini di byte big e little endian. Inoltre, consente di rappresentare dati in più file separati, con estensioni specifiche per le diverse tipologie di dati; migliorando la portabilità e la scalabilità del programma a discapito della complessità aggiunta. [22]

Questa idea non è stata implementata, perché il rendimento sarebbe stato marginale rispetto al tempo necessario per implementare un nuovo formato di file.

Algoritmi di ordinamento paralleli

Una delle funzioni che occupa molto tempo di esecuzione è `mergeIntersections` (sez. 3.3.2), che ordina i set di intersezioni di ogni raggio con i piani usando *merge sort*. Per incrementare le prestazioni, sarebbe possibile implementare una versione parallela di questo algoritmo; oppure usare algoritmi sviluppati appositamente per sistemi paralleli, come il *bitonic sort* o il *Samplesort*. [23]

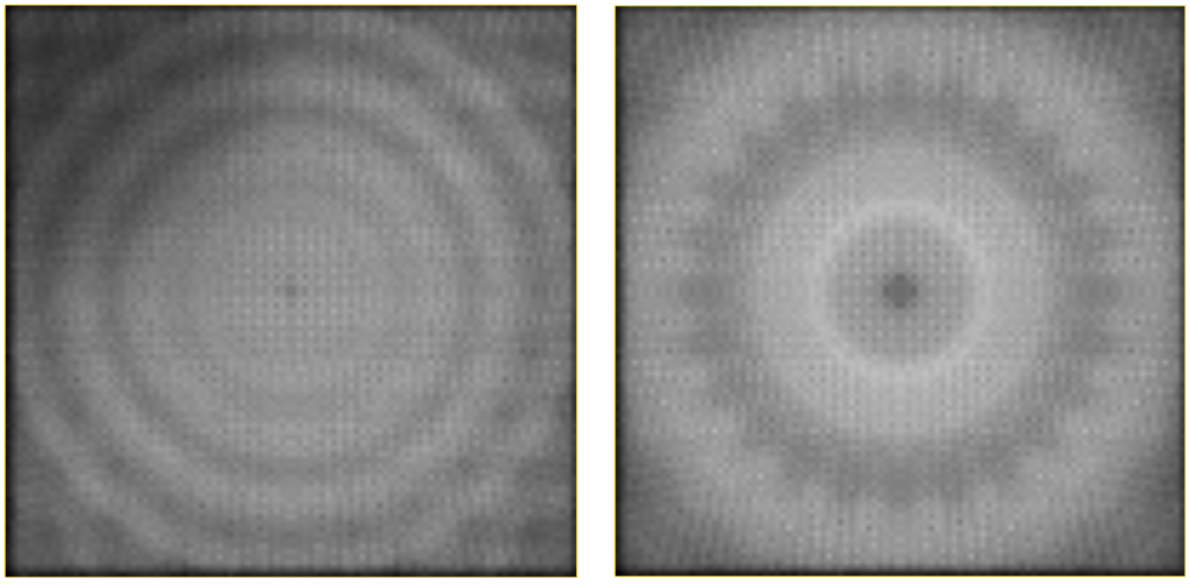
7.2.4 Riduzione degli artefatti

Le ricostruzioni 3D, attualmente, mostrano alcuni artefatti visivi sotto forma di cerchi concentrici, disposti lungo l'asse Y. La causa di questi effetti non è ancora chiara, potrebbero derivare da: un'imprecisione nelle proiezioni; da limitazioni intrinseche dell'algoritmo; o semplicemente dalla natura rotativa del tomografo.

Esistono molte tecniche per ridurre gli artefatti circolari, come la normalizzazione dell'intensità, spiegata in questo documento: "*Dynamic intensity normalization using eigen flat fields in X-ray imaging*" [24]. Molte di queste tecniche però sono applicabili solo per le scansioni CT eseguite su corpi e strumenti fisici. Non ho trovato molte tecniche applicabili alla simulazione della ricostruzione che ho implementato. Queste tecniche sono abbastanza complesse e non rientrano nell'argomento di questa tesi, ma potrebbero essere affrontate da specialisti di imaging medico.

Una possibile soluzione potrebbe essere la conversione delle immagini dei piani, da coordinate cartesiane (X, Z) a polari (ρ, θ). Trasformando gli artefatti circolari in linee rette. Non è una soluzione definitiva, ma potrebbe essere un punto di partenza per tramutare gli artefatti in una forma meno aggravante alla qualità dell'immagine.

In figura 7.1 sono mostrati due esempi di artefatti visivi presenti a diversi livelli di profondità nella ricostruzione (fig. 6.1).



(a) Artefatto A (b) Artefatto B

Figura 7.1: Esempi di artefatti visivi nella ricostruzione 3D

7.2.5 Parallelizzazione con CUDA

Un ultimo e significativo sviluppo sarebbe l'implementazione di una parallelizzazione su GPU tramite CUDA.

CUDA è un'architettura di calcolo parallelo sviluppata da NVIDIA, che permette di sfruttare l'architettura delle GPU per eseguire operazioni di calcolo ad alte prestazioni in parallelo.

Questo approccio, nonostante sia notevolmente più complicato di quello mostrato con OpenMP, potrebbe portare a un incremento sostanziale delle prestazioni. Grazie alla modularità del codice che ho sviluppato, l'adattamento del progetto alla GPU dovrebbe risultare relativamente semplice per studenti o sviluppatori con esperienza in CUDA.

Non essendo una tecnologia nuova, risalente al 2006 [25], esistono già numerose implementazioni di algoritmi di tomografia su GPU [26], come *GPU-based cone beam computed tomography* [27].

In alcuni casi, l'uso di GPU per l'elaborazione di immagini mediche è più veloce e conveniente rispetto all'uso di CPU, sorpassando in prestazioni le ottimizzazioni algoritmiche come quella presentata in questa tesi. Questa affermazione è dimostrata in *Hardware acceleration vs. algorithmic acceleration: Can GPU-based processing beat complexity optimization for CT?* [28].

Un breve esempio di come potrebbe essere implementata la parallelizzazione con CUDA è mostrato di seguito; notare che il codice è solo una bozza e non è completo o funzionante.

Prendendo la funzione `computeBackProjection` (sez. 3.3.2) come esempio, si potrebbe scrivere una versione parallela in CUDA come segue:

```
__global__ void computeBackProjectionKernel(projection* d_projection, volume* d_volume) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int idy = blockIdx.y * blockDim.y + threadIdx.y;

    if (idx < d_projection->nSidePixels && idy < d_projection->nSidePixels) {
        // Calcolo del percorso radiologico per il pixel (idx, idy)
    }
}

void computeBackProjection(projection* projection, volume* volume) {
    // Allocazione della memoria sulla GPU
    projection* d_projection;
    volume* d_volume;
    cudaMalloc(&d_projection, sizeof(projection));
    cudaMalloc(&d_volume, sizeof(volume));
```

```
16
17      // Copia dei dati dalla CPU alla GPU
18      cudaMemcpy(d_projection, projection, sizeof(projection), cudaMemcpyHostToDevice);
19      cudaMemcpy(d_volume, volume, sizeof(volume), cudaMemcpyHostToDevice);
20
21      // Configurazione e lancio del kernel CUDA
22      dim3 threadsPerBlock(16, 16); // Aggiustare le dimensioni dei blocchi
23      dim3 numBlocks((projection->nSidePixels+threadsPerBlock.x-1)/threadsPerBlock.x,
24                     (projection->nSidePixels+threadsPerBlock.y-1)/threadsPerBlock.y);
25      computeBackProjectionKernel<<<numBlocks, threadsPerBlock>>>(d_projection, d_volume);
26
27      // Copia dei risultati dalla GPU alla CPU
28      cudaMemcpy(volume, d_volume, sizeof(volume), cudaMemcpyDeviceToHost);
29
30      // De-allocazione della memoria sulla GPU
31      cudaFree(d_projection);
32      cudaFree(d_volume);
33  }
```

Questo codice mostra alcuni dei passaggi alla base della tecnologia CUDA, come:

- Allocazione della memoria sulla GPU;
- Copia dei dati di proiezione e volume dalla CPU alla GPU;
- Scrittura del kernel CUDA dove avverranno i calcoli per la retroproiezione;
- Configurazione e lancio del kernel CUDA con un numero appropriato di thread;
- Copia dei risultati dalla GPU alla CPU una volta terminato il kernel.
- De-allocazione della memoria sulla GPU.

Appendice A

Appendice

A.1 Codice sorgente

Il codice sorgente di tutto il progetto è disponibile su GitHub al seguente URL: https://github.com/borgotto/3D-CT-backprojection-openmp

Visitando il link è possibile accedere a:

- Il codice sorgente del programma di retroproiezione 3D;
- Istruzioni per la compilazione e l'esecuzione del programma;
- La documentazione del codice;
- Gli script per la profilazione del codice.
- Gli script per le misure delle prestazioni e generazione dei grafici.
- Esempi di file di input e output, usati e generati dal programma.
- Una volta pubblicata la tesi, verrà aggiunta anch'essa alla repository.

In futuro è probabile che vengano aggiornate varie sezioni, in caso di errata corrige o aggiunta di nuove funzionalità.

A.2 Documentazione

La documentazione del codice può essere consultata al seguente URL:
https://borgotto.github.io/3D-CT-backprojection-openmp/

Copre tutti i file sorgente del progetto, con una descrizione dettagliata di ogni funzione e variabile. Si può fare riferimento a questa documentazione per capire meglio il funzionamento del codice e per eventuali modifiche.

È stata generata con Doxygen, uno strumento per la generazione automatica di documentazione del codice.
http://www.doxygen.nl/

A.3 Parametri di configurazione

Il programma di retroproiezione simula il funzionamento di un tomografo con queste caratteristiche:

- Dimensione di ogni voxel di $100\mu m$;
- Dimensione del volume di $100000\mu m$ per lato (1000 voxel);
- Dimensione di un singolo pixel del detettore di $85\mu m$;
- Dimensione del detettore di $200000\mu m$ per lato (2352 pixel);
- Distanza sorgente-oggetto di $150000\mu m$;
- Distanza sorgente-detettore di $600000\mu m$;
- Angolo di rotazione totale di 360°;
- Distanza angolare tra due proiezioni adiacenti di 15°, per un totale di 25 proiezioni.

Bibliografia

[1] Aaron Filler. The history, development and impact of computed imaging in neurological diagnosis and neurosurgery: Ct, mri, and dti. *Nature precedings*, pages 1–1, 2009.

[2] Elizabeth C Beckmann. Ct scanning the early days. *The British journal of radiology*, 79(937):5–8, 2006.

[3] Avinash C. Kak and Malcolm Slaney. *Principles of Computerized Tomographic Imaging*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2001. Originally published by IEEE Press, 1988. Available for free online from SIAM.

[4] Brian Nett. Animated ct generations [1st, 2nd, 3rd, 4th, 5th gen ct] for radiologic technologists, 2024. Accessed: 2024-10-10.

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

[6] H Thomas et al. Introduction to algorithms, 2009.

[7] OpenMP Architecture Review Board. *OpenMP Application Programming Interface*. OpenMP, 2024. Accessed: 2024-10-15.

[8] OpenMP Architecture Review Board. *OpenMP Application Programming Interface Version 1.0*, 1997. Accessed: 2024-10-15.

[9] Michael Voss, Eric Chiu, Patrick Man Yan Chow, Catherine Wong, and Kevin Yuen. An evaluation of auto-scoping in openmp. In Barbara M. Chapman, editor, *Shared Memory Parallel Programming with Open MP*, pages 98–109, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[10] Valve Corporation. Steam hardware & software survey, 2024. Accessed on 2024-09-03. Survey data indicates that 6-core CPUs are the most common among users.

[11] Vikren Sarkar, Chengyu Shi, Prema Rassiah-Szegedi, Aidnag Diaz, Tony Eng, and Niko Papanikolaou. The effect of a limited number of projections and reconstruction algorithms on the image quality of megavoltage digital tomosynthesis. *Journal of applied clinical medical physics*, 10(3):155–172, 2009.

[12] ISO/IEC 9899:1999: Programming Languages – C, 1999. Accessed: 2024-11-01.

[13] GCC Developers. Gcc 6 release series: Changes, new features, and fixes, 2016. Accessed: 2024-11-03.

[14] GCC Developers. Gcc support for openmp, 2024. Accessed: 2024-11-01.

[15] Jose Fonseca. gprof2dot, 2024. Accessed: 2024-11-04.

[16] John Ellson, Emden Gansner, Eleftherios Koutsofios, Stephen North, and Gordon Woodhull. *Graphviz - Graph Visualization Software*, 2024. Accessed: 2024-11-04.

[17] J. D. Hunter. Matplotlib: A 2d graphics environment. `https://matplotlib.org/`, 2007. Matplotlib is a plotting library for the Python programming language and its numerical mathematics extension NumPy.

[18] Gordon Kindlmann and Teem contributors. Teem: Tools to process and visualize scientific data and images. Credits available at `https://teem.sourceforge.net/credits.html`, 2024. Accessed: 2024-11-04.

[19] Inc. Kitware. Itk/vtk viewer: Web-based viewer for scientific images, 2024. Accessed: 2024-11-04.

[20] NIH Image. Imagej: Image processing and analysis in java, N.D. Accessed: 2024-11-04.

[21] Zhen Xue, Liangliang Zhang, and Jinxiao Pan. A new algorithm for calculating the radiological path in ct image reconstruction. In *Proceedings of 2011 International Conference on Electronic & Mechanical Engineering and Information Technology*, volume 9, pages 4527–4530, 2011.

[22] VTK Developers. Vtk file formats documentation, 2023. Accessed: 2024-11-06.

[23] Selim G Akl. *Parallel sorting algorithms*, volume 12. Academic press, 2014.

[24] Vincent Van Nieuwenhove, Jan De Beenhouwer, Francesco De Carlo, Lucia Mancini, Federica Marone, and Jan Sijbers. Dynamic intensity normalization using eigen flat fields in x-ray imaging. *Optics Express*, 23(21):27975, oct 2015.

[25] NVIDIA Corporation. About cuda, 2024. Accessed: 2024-10-06.

[26] Anders Eklund, Paul Dufort, Daniel Forsberg, and Stephen M. LaConte. Medical image processing on the gpu – past, present and future. *Medical Image Analysis*, 17(8):1073–1094, 2013.

[27] Peter B. Noël, Alan M. Walczak, Jinhui Xu, Jason J. Corso, Kenneth R. Hoffmann, and Sebastian Schafer. Gpu-based cone beam computed tomography. *Computer Methods and Programs in Biomedicine*, 98(3):271–277, 2010. HP-MICCAI 2008.

[28] Neophytos Neophytou, Fang Xu, and Klaus Mueller. Hardware acceleration vs. algorithmic acceleration: Can gpu-based processing beat complexity optimization for ct? In *Medical Imaging 2007: Physics of Medical Imaging*, volume 6510, pages 1733–1741. SPIE, 2007.

Ringraziamenti

Vorrei fare i miei più sinceri ringraziamenti al Prof. Moreno Marzolla e la Prof.ssa Elena Loli Piccolomini per la loro professionalità, disponibilità, pazienza e passione che hanno per il loro lavoro. Grazie per avermi seguito e supportato durante il progetto e la stesura della tesi.

Ringrazio mia mamma per avermi permesso di studiare e per avermi sostenuto sia moralmente che materialmente.

Grazie anche ai miei amici e colleghi di corso, per aver condiviso con me le fatiche e le gioie di questi anni di studio.

Un ringraziamento speciale, va a Cristina e Samuele per essere sempre stati al mio fianco.