UNIVERSITÀ DEGLI STUDI DI NAPOLI PARTHENOPE

Dipartimento di Scienze e Tecnologie

Corso di Laurea in Informatica

Tesi di Laurea Sperimentale in Informatica

UN FRAMEWORK NUMERICO PER RISOLVERE PDEs IPERBOLICHE IN AMBIENTE HPC

A NUMERICAL FRAMEWORK FOR SOLVING HYPERBOLIC PDEs IN HPC ENVIRONMENT

Relatore:
Ch.ma Prof.ssa Livia Marcellino
Correlatore:
Ch.mo Prof. Pasquale De Luca

Candidato:
Francesco Fossari
Matricola: 0124002327

ANNO ACCADEMICO 2024/2025

Abstract

Italiano
Le equazioni alle derivate parziali (PDE, Partial Differential Equations) sono uno strumento matematico essenziale per descrivere fenomeni fisici, e sono ampiamente impiegate in ambiti scientifici e ingegneristici come la propagazione di onde acustiche ed elettromagnetiche, la dinamica dei fluidi e la modellazione strutturale. Le soluzioni analitiche, spesso impossibili in domini complessi o con condizioni al contorno particolari, rendono necessario l'uso di metodi numerici ad alta intensità computazionale. In questo contesto, le tecniche di parallelizzazione in ambienti di calcolo ad alte prestazioni (HPC, High Performance Computing) offrono un mezzo efficace per ridurre i tempi di calcolo.
Questa tesi si propone di sviluppare un framework numerico, scritto in linguaggio C, per la risoluzione di equazioni iperboliche alle derivate parziali, con particolare attenzione all'equazione delle onde bidimensionale. Il framework è stato validato mediante il confronto tra i risultati numerici e la soluzione analitica nota e analizzato in ottica HPC, per individuare strategie di parallelizzazione e ottimizzazione su architetture ad alte prestazioni.

English
Partial Differential Equations (PDEs) are a fundamental mathematical tool for describing physical phenomena. They are widely used in scientific and engineering fields such as acoustic and electromagnetic wave propagation, fluid dynamics, and structural modeling. Analytical solutions are often impractical in complex domains or with unusual boundary conditions, making computationally intensive numerical methods necessary. In this context, parallelization techniques in High Performance Computing (HPC) environments offer an effective way to reduce computation time.
This thesis aims to develop a numerical framework, written in C, for solving hyperbolic partial differential equations, with particular focus on the two-dimensional wave equation. The framework has been validated against the known analytical solution and analyzed from an HPC perspective to identify parallelization and optimization strategies for efficient use on high-performance architectures.

Indice

Capitolo 1

Introduzione

Risolvere le equazioni alle derivate parziali (PDE) iperboliche, come l'equazione delle onde, è un argomento di rilevante importanza in vari ambiti ingegneristici e scientifici, partendo dalla sismologia fino ad arrivare alla fluidodinamica.

In ambiente di calcolo ad alte prestazioni, esistono diverse strategie che nel tempo si sono rivelate efficaci per discretizzare questa tipologia di problemi, dove si usano metodi espliciti semplici da parallelizzare e con una struttura regolare, tra cui:

- Metodo Leap-Frog
- Metodo Lax-Wendroff
- Metodi a Volumi Finiti
- Metodi alle Differenze Finiti ad Alto Ordine
- Metodi Discontinui Galerkin
- Metodi Spettrali
- Metodi di Elementi Finiti
- Metodi ADER
- Metodi ENO/WENO

Alcuni linguaggi sono ampiamente utilizzati per questa tipologia di applicazioni, come il C, C++ e Fortran. Essi sono spesso combinati con librerie che implementano la parallelizzazione come MPI per il calcolo distribuito, OpenMP per il multithreading e CUDA per GPU Computing. Oltre alle implementazioni varie, esistono framework generalizzati come `deal.II`, `PETSc`, `PDE Toolbox` e `OpenFOAM`, che permettono l'utilizzo di strumenti sofisticati per risolvere PDE iperboliche in ambienti ad alte prestazioni, come mostrato in [1, 2, 3, 4].

Tale lavoro ha lo scopo di implementare un framework numerico adatto per risolvere l'equazione delle onde bidimensionale prima in versione sequenziale, e successivamente in parallelo, al fine di comprendere i benefici apportati dall'utilizzo di architetture ad alte prestazioni.

Lo schema Leap-Frog, grazie alla sua accuratezza del secondo ordine e la sua semplicità, è adatto per varie simulazioni numeriche.

Un esempio di rilievo lo si ha grazie al dipartimento di fisica dell'Università di Brigham Young, che grazie al suo laboratorio è riuscita nell'intento di risolvere l'equazione delle onde bidimensionale utilizzando lo schema Leap-Frog, fornendo l'implementazione di un framework in Python cosi da avere un metodo pratico per approcciarsi alla risoluzione di tale PDE con questo metodo esplicito, come illustrato in [5].

Inoltre, come citato da Gilbert Strang del MIT, è importante avere una griglia sfalsata cosi da garantire l'efficacia e la stabilità numerica del metodo.[6]

Tuttavia, come evidenziato da Shampine, a volte è necessario usufruire di alcune tecniche di smoothing per migliorare alcune possibili instabilità numeriche.[7]

Gli esempi precedenti dimostrano che lo schema Leap-Frog è molto utile per poter risolvere PDE di tipo iperbolico, ma molte di queste implementazioni presentano limitazioni in termini di parallelizzazione o di utilizzo in ambienti HPC, poiché si focalizzano principalmente sul risolvere casi specifici facendo ampio uso di risorse computazionali. Lo sviluppo di un framework numerico che utilizza lo schema Leap-Frog attraverso le opportune parallelizzazioni rappresenta un contributo significativo nel campo del calcolo numerico.

Negli ultimi anni sono stati sviluppati numerosi approcci per la risoluzione numerica dell'equazione delle onde bidimensionale, sia in ambito accademico che applicativo. Tra questi, i metodi a differenze finite si sono dimostrati particolarmente efficaci grazie alla loro semplicità e scalabilità.

In particolare, Altybay et al [8] propongono uno schema implicito di tipo ADI (Alternating Direction Implicit) per l'equazione acustica 2D, implementato su architetture ibride con GPU, OpenMP e MPI. Il lavoro evidenzia un notevole miglioramento delle prestazioni rispetto alla versione sequenziale.

Altri studi, come quello di Arnoldy e Adytia [9], utilizzano lo schema Leap-Frog in configurazioni implicite o semi-implicite, sfruttando l'accelerazione tramite CUDA per simulazioni ad alta risoluzione su griglie strutturate.

Infine, recenti ricerche [10] hanno combinato OpenMP e GPU per risolvere l'equazione delle onde su mezzi eterogenei, ottenendo un significativo speedup in ambito HPC.

La presente tesi si colloca in questo contesto, proponendo un'implementazione esplicita del metodo Leap-Frog in linguaggio `C`, parallelizzata tramite `OpenMP`, come soluzione semplice, modulare ed efficace su architetture multicore.

A differenza dei lavori esistenti che si concentrano su implementazioni in linguaggi ad alto

livello o su framework complessi, il presente lavoro propone un framework esplicito, modulare e completamente scritto in linguaggio `C`, parallelizzato con `OpenMP`, con l'obiettivo di bilanciare semplicità, efficienza e chiarezza algoritmica. Tale approccio non risulta presente nella documentazione consultata.

Il presente elaborato si compone di otto capitoli, ciascuno dei quali affronta un aspetto specifico del problema studiato.

Il Capitolo 2 introduce le equazioni alle derivate parziali, con particolare riferimento a quelle iperboliche. Dopo una breve classificazione e alcuni esempi significativi, si analizza la distinzione tra soluzioni analitiche e numeriche.

Il Capitolo 3 è dedicato ai metodi numerici utilizzati per trattare le PDE iperboliche: viene presentato il processo di discretizzazione, descritto lo schema Leap-Frog e analizzati aspetti legati alla stabilità e all'errore numerico.

Il Capitolo 4 descrive l'implementazione del framework numerico per la risoluzione dell'equazione delle onde, sviluppato in linguaggio `C`. Vengono illustrate l'architettura dell'algoritmo, le scelte progettuali e il codice sorgente, con un confronto finale con la soluzione analitica di riferimento.

Il Capitolo 5 offre una panoramica sul calcolo ad alte prestazioni(HPC), spiegandone i concetti fondamentali e la sua importanza nell'ambito della calcolo numerico.

Il Capitolo 6 analizza le tecniche di parallelizzazione applicate al codice, confrontando le prestazioni della versione parallela con quella sequenziale e discutendo i risultati ottenuti.

Il Capitolo 7 presenta i risultati numerici, includendo misure di errore e osservazioni sulla stabilità e sull'efficienza computazionale del metodo.

Infine, il Capitolo 8 riassume i principali risultati emersi durante i test, discute i limiti dell'approccio adottato e propone possibili sviluppi futuri.

Capitolo 2

Equazioni Differenziali alle Derivate Parziali Iperboliche

2.1 Introduzione alle Equazioni Differenziali Parziali

Come citato in [11], le equazioni differenziali alle derivate parziali (PDE) costituiscono uno strumento matematico fondamentale per la modellazione di fenomeni fisici, ingegneristici, economici e ambientali. A differenza delle equazioni differenziali ordinarie, che coinvolgono una sola variabile indipendente, le PDE descrivono relazioni tra una funzione incognita di più variabili indipendenti e le sue derivate parziali.

Una forma generale di PDE è:

$$F(x_1, x_2, \ldots, x_n, u, \partial_{x_1} u, \partial_{x_2} u, \ldots, \partial^k_{x_k} u) = 0$$

dove $u = u(x_1, \ldots, x_n)$ è la funzione incognita e F è una funzione data che coinvolge le variabili indipendenti, la funzione incognita e le sue derivate parziali fino all'ordine k.

2.1.1 I "Big Three Problems"

Tre modelli classici sono di importanza centrale nello studio delle PDE:

- **Equazione di Laplace** (ellittica): $\nabla^2 u = 0$
  Descrive fenomeni stazionari come la temperatura in regime di equilibrio o il potenziale elettrostatico in assenza di cariche.
- **Equazione del calore** (parabolica): $\partial_t u = \alpha \nabla^2 u$
  Modella processi di diffusione termica o di massa.
- **Equazione delle onde** (iperbolica): $\partial_{tt} u = c^2 \nabla^2 u$
  Rappresenta la propagazione di onde acustiche, meccaniche ed elettromagnetiche e perturbazioni di fluidi.

2.1.2 Classificazione delle PDE del Secondo Ordine

Data una PDE del secondo ordine nella forma:

$$A\frac{\partial^2 u}{\partial x^2} + B\frac{\partial^2 u}{\partial x \partial y} + C\frac{\partial^2 u}{\partial y^2} + \cdots = G$$

la classificazione si basa sul discriminante $\Delta = B^2 - 4AC$:

- $\Delta < 0$: Equazioni **ellittiche**
- $\Delta = 0$: Equazioni **paraboliche**
- $\Delta > 0$: Equazioni **iperboliche**

2.2 Interpretazione Geometrica

Le curve caratteristiche delle PDE permettono di interpretarne la natura:

$$A(dy)^2 - Bdx\,dy + C(dx)^2 = 0$$

Per le PDE iperboliche esistono due famiglie di curve caratteristiche lungo cui l'informazione si propaga con velocità finita, creando zone di influenza delimitate. Le discontinuità iniziali si propagano lungo le caratteristiche senza smorzamento.

2.2.1 Condizioni Iniziali e al Contorno

Per rendere ben posto un problema alle derivate parziali, è necessario specificare condizioni appropriate, in accordo con il principio di Hadamard. Queste condizioni si dividono in condizioni iniziali, che specificano lo stato del sistema a un tempo iniziale, e condizioni al contorno, che descrivono il comportamento della soluzione sui bordi del dominio spaziale.

2.2.2 Condizioni Iniziali

- Per PDE del primo ordine nel tempo è necessario specificare il valore della funzione incognita al tempo iniziale: (es. equazione del calore):

$$u(x, y, 0) = f(x, y)$$

- Per PDE del secondo ordine nel tempo sono richieste due condizioni iniziali: (es. equazione delle onde):

$$u(x, y, 0) = f(x, y), \quad \partial_t u(x, y, 0) = g(x, y)$$

La prima condizione specifica la configurazione iniziale, mentre la seconda determina la velocità iniziale di evoluzione del sistema.

2.2.3 Condizioni al Contorno

Le principali tipologie sono:

- **Condizioni di Dirichlet:** si specifica il valore della soluzione sul bordo del dominio $\partial\Omega$

$$u|_{\partial\Omega} = h(x, y)$$

- **Condizioni di Neumann:** si specifica la condizione della derivata normale sul bordo

$$\left.\frac{\partial u}{\partial n}\right|_{\partial\Omega} = h(x, y)$$

- **Condizioni di Robin:** sono una combinazione lineare del valore della funzione e della sua derivata normale:

$$\alpha u + \beta \frac{\partial u}{\partial n} = h(x, y)$$

Queste condizioni modellano fisicamente, rispettivamente: vincoli imposti, flussi al bordo e scambi convettivi con l'ambiente.

2.3 Equazioni delle Onde come esempio

Come descritto in [12], le equazioni differenziali alle derivate parziali (PDE) di tipo iperbolico sono fondamentali nella modellazione di fenomeni ondulatori in fisica e ingegneria. L'equazione delle onde rappresenta il prototipo di questa classe e descrive la propagazione di perturbazioni in un mezzo continuo.

In particolare, consideriamo l'equazione delle onde bidimensionale, omogenea (cioè con termine di forzante nullo $f = 0$), in un dominio rettangolare $\Omega = [0, L_x] \times [0, L_y]$:

$$\frac{\partial^2 u}{\partial t^2} = c^2 \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) \tag{2.1}$$

Il problema è completato dalle condizioni iniziali e al contorno:

$$\begin{cases} u(x, y, 0) = u_0(x, y), \\ \dfrac{\partial u}{\partial t}(x, y, 0) = v_0(x, y), \\ u(x, y, t) = 0, \quad (x, y) \in \partial\Omega, \quad t \geq 0. \end{cases} \tag{2.2}$$

Questa equazione è iperbolica in quanto il discriminante $B^2 - AC = c^2 > 0$, confermando la natura ondulatoria della soluzione. I fenomeni modellati includono:

- propagazione di onde acustiche;
- vibrazioni di membrane elastiche;
- propagazione di onde sismiche;
- fenomeni elettromagnetici.

2.4 Formulazione variazionale

Prima della discretizzazione, riformuliamo il problema in forma variazionale. Sia $V = H_0^1(\Omega)$ lo spazio di Sobolev di funzioni che si annullano sul bordo $\partial\Omega$.

Moltiplichiamo l'equazione (2.1) per una funzione test $v \in V$ e integriamo su Ω:

$$\int_\Omega \frac{\partial^2 u}{\partial t^2} v \, dxdy = c^2 \int_\Omega \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) v \, dxdy. \tag{2.3}$$

Applicando l'integrazione per parti e tenendo conto delle condizioni di Dirichlet omogenee, otteniamo:

$$\int_\Omega \frac{\partial^2 u}{\partial t^2} v \, dxdy + c^2 \int_\Omega \left(\frac{\partial u}{\partial x}\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y}\frac{\partial v}{\partial y} \right) dxdy = 0. \tag{2.4}$$

Definiamo le forme bilineari:

$$a(u, v) = c^2 \int_\Omega \left(\frac{\partial u}{\partial x}\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y}\frac{\partial v}{\partial y} \right) dxdy, \tag{2.5}$$

$$(u, v) = \int_\Omega uv \, dxdy. \tag{2.6}$$

Il problema variazionale risulta quindi: trovare

$$u \in C^2([0, T]; L^2(\Omega)) \cap C^0([0, T]; V)$$

tale che, per ogni $v \in V$ e per ogni $t \in (0, T]$, si abbia

$$(\ddot{u}(t), v) + a(u(t), v) = 0, \tag{2.7}$$

con condizioni iniziali

$$u(0) = u_0 \in V, \quad \dot{u}(0) = v_0 \in L^2(\Omega). \tag{2.8}$$

Capitolo 3

Metodi Numerici per PDE iperboliche

In questo capitolo vengono descritti alcuni metodi numerici utilizzati per risolvere equazioni alle derivate parziali (PDE) di tipo iperbolico, con particolare riferimento all'equazione delle onde bidimensionale. Dopo aver introdotto la semi-discretizzazione spaziale tramite il metodo delle differenze finite, si passerà alla formulazione matriciale del problema.

Verrà quindi presentato lo schema Leap-Frog, scelto per la sua semplicità implementativa e per le buone proprietà di conservazione dell'energia. Successivamente ci saranno un'analisi di consistenza e stabilità numerica, fondamentali per garantire la correttezza e l'affidabilità della simulazione. Infine, verrà introdotto un criterio quantitativo per la valutazione dell'errore rispetto alla soluzione analitica, basato sulla norma L^2.

L'intero approccio ha l'obiettivo di costruire uno schema numerico efficiente e stabile per la simulazione di fenomeni di propagazione dell'onda in domini bidimensionali.

3.1 Semi-discretizzazione spaziale

Adottiamo il metodo delle differenze finite per la discretizzazione spaziale. Consideriamo una griglia uniforme:

$$x_i = i\Delta x, \quad y_j = j\Delta y, \quad \Delta x = \frac{L_x}{N_x}, \quad \Delta y = \frac{L_y}{N_y}, \tag{3.1}$$

per $i = 0, \ldots, N_x$ e $j = 0, \ldots, N_y$.

Definiamo l'approssimazione puntuale $u_{i,j}(t) \approx u(x_i, y_j, t)$. Le derivate seconde spaziali sono approssimate da:

$$\frac{\partial^2 u}{\partial x^2}(x_i, y_j, t) \approx \frac{u_{i+1,j}(t) - 2u_{i,j}(t) + u_{i-1,j}(t)}{(\Delta x)^2}, \tag{3.2}$$

$$\frac{\partial^2 u}{\partial y^2}(x_i, y_j, t) \approx \frac{u_{i,j+1}(t) - 2u_{i,j}(t) + u_{i,j-1}(t)}{(\Delta y)^2}. \tag{3.3}$$

Inserendo queste approssimazioni nell'equazione (2.1) si ottiene il sistema di equazioni differenziali ordinarie:

$$\frac{d^2 u_{i,j}}{dt^2} = c^2 \left(\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{(\Delta x)^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{(\Delta y)^2} \right), \tag{3.4}$$

per ogni $i = 1, \ldots, N_x - 1$ e $j = 1, \ldots, N_y - 1$, con condizioni al contorno $u_{i,j}(t) = 0$ per (i, j) sul bordo.

3.2 Formulazione matriciale

Per una formulazione compatta, si procede disponendo i valori $u_{i,j}$ in un vettore colonna $u(t) \in \mathbb{R}^{(N_x-1)(N_y-1)}$ secondo un ordine naturale (ad esempio prima per j poi per i).

Definiamo le matrici tridiagonali $D_x \in \mathbb{R}^{(N_x-1)\times(N_x-1)}$ e $D_y \in \mathbb{R}^{(N_y-1)\times(N_y-1)}$ relative all'approssimazione della derivata seconda centrata:

$$D_x = \frac{1}{(\Delta x)^2} \begin{bmatrix} -2 & 1 & & 0 \\ 1 & -2 & \ddots & \\ & \ddots & \ddots & 1 \\ 0 & & 1 & -2 \end{bmatrix}, \quad D_y = \frac{1}{(\Delta y)^2} \begin{bmatrix} -2 & 1 & & 0 \\ 1 & -2 & \ddots & \\ & \ddots & \ddots & 1 \\ 0 & & 1 & -2 \end{bmatrix}.$$

Allora la matrice del problema è data da:

$$A = I_{N_y-1} \otimes D_x + D_y \otimes I_{N_x-1}, \tag{3.5}$$

dove I_m è la matrice identità di dimensione m e $\otimes$ denota il prodotto di Kronecker.

Il sistema semi-discreto si scrive come

$$\ddot{u}(t) = c^2 A u(t), \quad u(0) = u_0, \quad \dot{u}(0) = v_0, \tag{3.6}$$

dove u_0 e v_0 sono i vettori di condizioni iniziali appropriati.

3.3 Schema Leap-Frog

Per discretizzare completamente l'equazione delle onde, viene utilizzato lo schema numerico Leap-Frog, particolarmente adatto alla risoluzione di PDE iperboliche. Questo schema è stato scelto per la sua semplicità di implementazione, per la stabilità condizionata e per le proprietà di simmetria e conservazione dell'energia nel tempo.

Lo schema Leap-Frog, di secondo ordine in accuratezza temporale, si basa su tre livelli temporali. La sua forma completamente esplicita è data da:

$$u_{i,j}^{n+1} = 2u_{i,j}^{n} - u_{i,j}^{n-1} + c^2\,\Delta t^2 \left(\frac{u_{i+1,j}^{n} - 2u_{i,j}^{n} + u_{i-1,j}^{n}}{\Delta x^2} + \frac{u_{i,j+1}^{n} - 2u_{i,j}^{n} + u_{i,j-1}^{n}}{\Delta y^2} \right)$$

Il termine tra parentesi rappresenta il *laplaciano discreto*, che approssima la curvatura spaziale della funzione.

La derivata seconda nel tempo viene approssimata tramite differenze finite centrate:

$$\frac{u^{n+1} - 2u^{n} + u^{n-1}}{(\Delta t)^2} \approx \frac{d^2u}{dt^2}(t^n) \tag{3.7}$$

Sostituendo questa espressione nella formulazione semi-discreta dell'equazione delle onde $\ddot{u}(t) = c^2 A u(t)$, si ottiene lo schema completamente discreto:

$$u^{n+1} = 2u^{n} - u^{n-1} + c^2(\Delta t)^2 A u^{n} \tag{3.8}$$

Per iniziare l'integrazione temporale, è necessario conoscere u^0 e u^1.
Il primo è fornito dalle condizioni iniziali, mentre u^1 viene calcolato mediante un'espansione di Taylor del secondo ordine:

$$u^1 = u^0 + \Delta t\, v^0 + \frac{c^2(\Delta t)^2}{2} A u^0 \tag{3.9}$$

Questo schema è particolarmente adatto alla simulazione di sistemi conservativi, in quanto, in assenza di forzanti e dissipazione, conserva l'energia numerica del sistema.

3.4 Consistenza

Lo schema Leap-Frog è consistente di secondo ordine rispetto al tempo. Per dimostrarlo, si considera lo sviluppo in serie di Taylor della soluzione continua $u(t)$ nei punti t^{n+1} e t^{n-1} attorno a t^n:

$$u(t^{n+1}) = u(t^n) + \Delta t \, \frac{du}{dt}(t^n) + \frac{(\Delta t)^2}{2} \, \frac{d^2u}{dt^2}(t^n) + \frac{(\Delta t)^3}{6} \, \frac{d^3u}{dt^3}(t^n) + \mathcal{O}((\Delta t)^4) \tag{3.10}$$

$$u(t^{n-1}) = u(t^n) - \Delta t \, \frac{du}{dt}(t^n) + \frac{(\Delta t)^2}{2} \, \frac{d^2u}{dt^2}(t^n) - \frac{(\Delta t)^3}{6} \, \frac{d^3u}{dt^3}(t^n) + \mathcal{O}((\Delta t)^4) \tag{3.11}$$

Sommando le due espressioni, si ottiene:

$$u(t^{n+1}) + u(t^{n-1}) = 2u(t^n) + (\Delta t)^2 \frac{d^2u}{dt^2}(t^n) + \mathcal{O}((\Delta t)^4) \tag{3.12}$$

Riordinando i termini:

$$\frac{u(t^{n+1}) - 2u(t^n) + u(t^{n-1})}{(\Delta t)^2} = \frac{d^2u}{dt^2}(t^n) + \mathcal{O}((\Delta t)^2) \tag{3.13}$$

Questa relazione mostra che l'approssimazione della derivata seconda mediante lo schema Leap-Frog introduce un errore dell'ordine $\mathcal{O}((\Delta t)^2)$, confermando che lo schema è consistente di secondo ordine nel tempo.

3.5 Stabilità Numerica

Per far sì che il metodo numerico sia stabile, si impone la condizione di stabilità di tipo CFL (Courant–Friedrichs–Lewy):

$$dt \leq \frac{1}{c\sqrt{\frac{1}{dx^2} + \frac{1}{dy^2}}}$$

Nel presente lavoro si è scelto di impostare:

$$dt = 0.9 \cdot dt_{\max}$$

dove $dt_{\max}$ è il valore massimo ammesso dalla condizione di stabilità.

Il primo passo temporale (per $n = 1$) non può essere calcolato con Leap-Frog poiché richiederebbe un valore a $n = -1$. Si utilizza quindi una formula esplicita di Taylor:

$$u^1_{i,j} = u^0_{i,j} + dt \cdot v_0(x_i, y_j) + \frac{1}{2}c^2 dt^2 \cdot \Delta u^0_{i,j}$$

Nel nostro caso, la velocità iniziale è nulla ($v_0 = 0$), quindi si ha:

$$u^1_{i,j} = u^0_{i,j} + \frac{1}{2}c^2 dt^2 \cdot \Delta u^0_{i,j}$$

3.6 Analisi dell'Errore

Al termine della simulazione, l'accuratezza della soluzione numerica viene valutata tramite il calcolo della norma L^2 dell'errore, che confronta la soluzione ottenuta con quella analitica:

$$\text{Errore}_{L^2} = \sqrt{\frac{1}{(N_x+1)(N_y+1)} \sum_{i,j} \left(u^{\text{num}}_{i,j}(T) - u^{\text{esatto}}_{i,j}(T)\right)^2}$$

Capitolo 4

Implementazione del Framework Numerico

4.1 Pseudocodice del Framework Numerico

In questa sezione è presente lo pseudocodice del framework numerico implementato per risolvere l'equazione delle onde 2D, dove l'algoritmo prevede:

- La generazione della griglia per vari valori di risoluzione h;
- L'inizializzazione della soluzione numerica;
- L'evoluzione temporale con lo schema Leap-Frog:
    1. Calcolo del laplaciano della soluzione al tempo corrente u^n mediante differenze finite centrali.
    2. Calcolo della nuova soluzione u^{n+1} utilizzando la formula Leap-Frog:

$$u_{i,j}^{n+1} = 2u_{i,j}^{n} - u_{i,j}^{n-1} + c^2 \Delta t^2 \nabla^2 u_{i,j}^{n}$$

    3. Applicazione delle condizioni al contorno di Dirichlet con valori nulli ai bordi del dominio al tempo t_{n+1}.
    4. Ripetizione del processo per ogni passo temporale fino a T.
- Il calcolo dell'errore L^2 rispetto alla soluzione esatta;
- Calcolo della stima dell'ordine di convergenza;

Di seguito, è presente l'algoritmo descritto in modo dettagliato attraverso l'utilizzo dello pseudocodice, in modo da facilitarne la comprensione e l'eventuale implementazione.

Algorithm 1 Pseudocodice del Framework Numerico

Definire: dominio spaziale L_x, L_y, tempo finale T, velocità c
Definire: lista di risoluzioni spaziali $\{h_1, h_2, \dots\}$
for ogni h nella lista **do**
    // — Generazione della griglia per vari valori di risoluzione h —
    Calcolare $N_x = L_x/h$, $N_y = L_y/h$
    Calcolare $dx = L_x/N_x$, $dy = L_y/N_y$
    Calcolare dt soddisfacendo la condizione CFL
    Determinare il numero di passi temporali $N_t = T/dt$ // con dt adattato per avere $t = T$ al passo $n = N_t$
    Costruire i vettori spaziali x_i e y_j
    // — Inizializzazione della soluzione numerica —
    for ogni punto (i, j) della griglia **do**
        Impostare $u^0_{i,j} = \sin(\pi x_i)\sin(\pi y_j)$
    end for
    for ogni punto interno (i, j) **do**
        Calcolare $u^1_{i,j} = u^0_{i,j} + \frac{1}{2}c^2 dt^2 \nabla^2 u^0_{i,j}$
    end for
    Imporre condizioni al contorno: $u^1_{i,j} = 0$ sui bordi
    // — Evoluzione temporale con lo schema Leap-Frog —
    for $n = 1$ fino a $N_t - 1$ **do**
        for ogni punto interno (i, j) **do**
            // (1) Calcolo del laplaciano della soluzione al tempo corrente
            Calcolare $\nabla^2 u^n_{i,j}$ con differenze finite centrali
            // (2) Calcolo della nuova soluzione $u^{n+1}_{i,j}$ con Leap-Frog
            $u^{n+1}_{i,j} = 2u^n_{i,j} - u^{n-1}_{i,j} + c^2 dt^2 \nabla^2 u^n_{i,j}$
        end for
        // (3) Applicazione delle condizioni al contorno (valori nulli)
        Imporre $u^{n+1}_{i,j} = 0$ sui bordi
        // (4) Ripetizione del processo per ogni passo temporale fino a T.
    end for
    // — Soluzione finale u al tempo Nt —
    Memorizzare u^{N_t}
    // — Calcolo dell'errore L^2 rispetto alla soluzione esatta —
    $errore = \sqrt{\sum_{i,j}(u^{N_t}_{i,j} - u_{esatto}(x_i, y_j, T))^2/N}$
    // — Stima dell'ordine di convergenza —
    if non è il primo h **then**
        Calcolare ordine: $p = \log(\text{errore}_{precedente}/\text{errore})/\log(h_{precedente}/h)$
    end if
    Salvare l'errore corrente per il prossimo ciclo
end for

4.2 Implementazione dell'Algoritmo

L'algoritmo numerico descritto in precedenza è stato implementato in linguaggio `C` in versione sequenziale, sfruttando strutture dati dinamiche per gestire la soluzione tridimensionale dell'equazione delle onde.

L'utilizzo della libreria `omp.h` è stato introdotto esclusivamente per calcolare i tempi di esecuzione mediante la funzione `omp_get_wtime()`, per effettuare successivamente il confronto tra le prestazioni dell'algoritmo sequenziale con una futura versione parallelizzata.

Di seguito si descrivono le principali librerie e funzioni utilizzate per l'implementazione del Framework numerico.

4.2.1 Librerie utilizzate

- `math.h`: necessaria per l'utilizzo di funzioni matematiche standard come `sin`, `pow`, `sqrt`;
- `omp.h`: utilizzata per la misurazione dei tempi di esecuzione mediante la funzione `omp_get_wtime()`.

4.2.2 Funzioni implementate

- `condizione_iniziale(double x, double y)`: imposta la configurazione iniziale della funzione $u(x, y, 0)$.
- `velocita_iniziale(double x, double y)`: definisce la velocità iniziale del sistema, in questo caso è fissata a zero.
- `calcolo_primo_step(...)` : calcola il primo passo temporale u^1 utilizzando un'espansione di Taylor, ma anche per lo studio della stabilità e l'analisi dell'errore.
- `calcolo_step_leap_frog(...)` : implementa lo schema Leap-Frog per risolvere nel tempo l'equazione iperbolica;
- `soluzione_analitica(...)` : calcola la soluzione esatta dell'equazione delle onde, utilizzata poi per il confronto con la soluzione numerica;
- `analisi_convergenza(...)` : è la funzione principale che gestisce l'intero framework:
  - discretizza il dominio spaziale e temporale, rispettando la condizione CFL;
  - alloca dinamicamente un array tridimensionale $U[n][j][i]$ per memorizzare la soluzione;
  - imposta le condizioni iniziali e al contorno (di tipo Dirichlet omogenee, con $u = 0$ sui bordi per ogni istante di tempo);
  - esegue il calcolo numerico nel tempo tramite lo schema Leap-Frog;
  - calcola l'errore L^2 confrontando la soluzione numerica con la soluzione analitica;

– stampa i risultati e dealloca tutta la memoria dinamica per prevenire errori di segmentazione (segmentation fault).

Infine, la funzione `main` inizializza i parametri della simulazione e richiama la funzione `analisi_convergenza`. Quando il programma viene eseguito, grazie alla funzione `omp_get_wtime()` viene calcolato il tempo impiegato per la costruzione della griglia, per l'inizializzazione delle condizioni iniziali e l'utilizzo dello schema Leap-Frog, ed infine viene anche calcolato il tempo totale per l'esecuzione del framework numerico.

Il codice in sequenziale è stato implementato in modo da poter essere parallelizzato in futuro, cosi da ottenere performance migliori e possibilmente adattarlo ad altri diversi riutilizzi.

Il framework numerico in versione sequenziale è visionabile nell'appendice 9.1

4.3 Risultati Numerici

L'obiettivo posto in questa sezione è di analizzare la precisione della soluzione e l'efficienza computazionale del framework numerico sequenziale, andando a verificare il risultato numerico.

4.3.1 Tabella dei Risultati

La figura 4.1 mostra l'output del terminale generato dal framework per la simulazione numerica dell'equazione delle onde in 2D.
I parametri della simulazione sono:

- Dominio spaziale: $L_x = 1.00$, $L_y = 1.00$
- Durata della simulazione: $T = 2.00\,\text{s}$
- Velocità dell'onda: $c = 1.00\,\text{unità/s}$

L'output include l'**analisi di convergenza**, riportata in forma tabellare.
Per ogni risoluzione spaziale h vengono mostrati:

- Il passo temporale dt
- L'errore nella norma L^2
- L'ordine di convergenza
- Il tempo di esecuzione complessivo della simulazione per tale risoluzione
- I tempi parziali per ciascuna fase del metodo:
  - Generazione della griglia
  - Inizializzazione delle condizioni iniziali
  - Primo passo temporale con sviluppo di Taylor
  - Integrazione numerica tramite lo schema Leap-Frog

Si osserva un ordine di convergenza numerico vicino a 2, in linea con le aspettative teoriche per il metodo adottato.
Il tempo totale di esecuzione della funzione `analisi_convergenza` è pari a **0.101837 secondi**.

```
== Simulazione 2D Equazione delle Onde ==
Dominio: [0,1.0]x[0,1.0], Tempo T = 2.0s, Velocità c = 1.0


                ANALISI DI CONVERGENZA

h          dt           Errore L2       Ordine     Nt         (Nx+1)*(Ny+1)   [Griglia | C.Iniziali | 1° Step | Leap-Frog] (s)
0.1000     0.062500     1.891562e-03    -          32         121             [0.000070 | 0.000042 | 0.000016 | 0.000327]
0.0500     0.031746     4.341028e-04    2.12       63         441             [0.000363 | 0.000069 | 0.000037 | 0.002688]
0.0250     0.015873     1.108885e-04    1.97       126        1681            [0.002422 | 0.000290 | 0.000157 | 0.014165]
0.0125     0.007937     2.804643e-05    1.98       252        6561            [0.008331 | 0.000398 | 0.000236 | 0.069850]

Tempo di esecuzione totale: 0.101837 secondi
```

Figura 4.1: Stampa dei Risultati sul Terminale

Successivamente, è stata realizzata una tabella dove sono stati riportati in modo più chiaro e leggibile i risultati dell'analisi di convergenza ottenuti dall'esecuzione del framework, in modo da facilitarne l'interpretazione:

Griglia Spaziale	**Punti Temporali**	**Errore L^2**	**Ordine di Convergenza**
11×11	32	1.8916×10^{-3}	–
21×21	63	4.3410×10^{-4}	2.12
41×41	126	1.0888×10^{-4}	1.97
81×81	252	2.8046×10^{-5}	1.98

Tabella 4.1: Errori L^2 per diverse risoluzioni spaziali

Per ciascun valore di h(Passo Spaziale) è stato calcolato il corrispondente passo temporale dt(Passo Temporale), rispettando la condizione CFL. In seguito, è stato analizzato l'errore L^2 tra la soluzione numerica e quella analitica al tempo $T = 2.0$.

Si osserva che al diminuire di h, l'errore L^2 si riduce in modo significativo. L'ordine di convergenza si mantiene intorno al valore 2, confermando che lo schema utilizzato, nel nostro caso Leap-Frog di secondo ordine, è molto efficiente.

Questi risultati confermano che il framework numerico sviluppato è in linea con i risultati sperati.

4.3.2 Discussione delle Prestazioni

Le prestazioni dell'algoritmo si basano principalmente sull'esecuzione della funzione principale **analisi_convergenza**, dove il tempo totale dell'esecuzione ha richiesto in media 0.1 secondi per terminare.

Griglia ($N \times N$)	**Nt**	**Griglia (s)**	**Cond. iniziali (s)**	**1° passo (s)**	**Leap-Frog (s)**
11×11	32	0.000070	0.000042	0.000016	0.000327
21×21	63	0.000363	0.000069	0.000037	0.002688
41×41	126	0.002422	0.000290	0.000157	0.014165
81×81	252	0.008331	0.000398	0.000236	0.069850

Tabella 4.2: Analisi delle prestazioni con tempi separati per generazione griglia, condizioni iniziali, primo passo temporale e schema Leap-Frog.

I tempi riportati in Tabella 4.2 evidenziano che la maggior parte del tempo impiegato per l'esecuzione del framework è dovuto all'utilizzo dello schema Leap-Frog e alla sua evoluzione temporale, infatti la creazione della griglia risulta avere un tempo trascurabile a confronto. Eventuali ottimizzazioni come la parallelizzazione dovrebbero concentrarsi principalmente sull'evoluzione temporale.

Questo è un comportamento del tutto normale, infatti sono necessarie più operazioni computazionali per poter lavorare con un maggior numero di punti nello spazio e nel tempo.

Anche se l'algoritmo è stato implementato per essere eseguito in sequenziale, esso si è rilevato alquanto veloce con tutte le diverse risoluzioni spazio-temporali, questo è stato possibile anche grazie all'utilizzo dello schema Leap-Frog.

In futuro, grazie alla parallelizzazione, sarà possibile soddisfare l'obiettivo principale, cioè ottenere prestazioni migliori con un tempo d'esecuzione finale minore, anche con griglie di dimensioni maggiori.

Capitolo 5

Introduzione all' HPC

5.1 Cos'è e perché è utile

Come introdotto nella dispensa del corso [13], l'HPC (High Performance Computing) nasce dall'esigenza di affrontare problemi su larga scala che richiedono grandi quantità di calcolo, impiegando algoritmi, software e hardware per fornire soluzioni precise in tempo reale.

Un esempio di problemi caratterizzati dalla necessità di ottenere una soluzione nel minor tempo possibile comprende:

- Ricerca su Internet
- Trasporto
- Pubblicità e Marketing
- Servizi bancari e finanziari
- Meteorologia
- Media e intrattenimento
- Assistenza sanitaria
- Sicurezza informatica
- Formazione

Le prestazioni vengono misurate in base al tempo impiegato per risolvere un determinato problema. Infatti, l'obiettivo principale di un supercomputer è quello di fornire una soluzione affidabile nel minor tempo possibile.

Ovviamente, esistono casi in cui utilizzare un supercomputer non è strettamente necessario, ma ve ne sono altri in cui risulta fondamentale. Ad esempio, effettuare una ricerca su Google,

dove circa 8,5 miliardi di utenti eseguono ricerche quotidianamente, richiede un'elevata potenza di calcolo per garantire tempi di risposta inferiori a un secondo.

Come molte altre tecnologie, l'HPC nasce per soddisfare esigenze economiche e scopi bellici, ma anche per permettere l'elaborazione di previsioni meteorologiche.

La principale sfida del calcolo ad alte prestazioni consiste nel ridurre al minimo il tempo di esecuzione degli algoritmi. Questo obiettivo può essere raggiunto impiegando un numero elevato di unità processanti (n unità), che lavorano in parallelo.

5.2 Concetti Base

Un sistema HPC è capace di elaborare una enorme quantità di dati in tempi ridotti grazie al parallelismo e alla potenza computazionale distribuita su più unità processanti. Esso si riferisce all'impiego di risorse di calcolo avanzate che non potrebbero essere sfruttate attraverso l'utilizzo di sistemi tradizionali, dove vengono eseguiti compiti generici in ambiente sequenziale. La vera forza dell'HPC quindi, è attraverso l'uso di numerosi processori, che lavorano contemporaneamente su più parti di un solo problema al fine di ridurre i tempi per ottenere una soluzione valida.

Le principali caratteristiche che definiscono un sistema HPC sono:

- **Prestazioni elevate**, misurate in FLOPS (Floating Point Operations Per Second)
- **Scalabilità**, ovvero la capacità del sistema di aumentare le prestazioni all'aumentare delle risorse
- **Parallelismo**, che consente l'esecuzione simultanea di molteplici operazioni
- **Elevato consumo energetico**, spesso ottimizzato attraverso architetture efficienti
- **Costo**, generalmente superiore rispetto ai calcolatori tradizionali

Un tipico calcolatore ad alte prestazioni è composto da migliaia di nodi di calcolo, connessi tra loro attraverso reti ad alta velocità. Ogni nodo include una o più CPU e, nella maggior parte dei casi, anche GPU specializzate per eseguire l'elaborazione in parallelo.

Questi concetti saranno ulteriormente approfonditi nelle sezioni successive dedicate alla scalabilità e all'ottimizzazione delle performance.

5.3 Considerazioni su scalabilità e performance

Nel calcolo ad alte prestazioni, per scalabilità si intende la capacità di un sistema di ridurre i tempi d'esecuzione ed aumentare l'efficienza dello stesso all'aumentare delle risorse computazionali impiegate durante l'esecuzione. Esistono due principali tipi di scalabilità:

- **Scalabilità forte**: misura quanto migliora l'esecuzione di un problema all'aumentare del numero di processori.
- **Scalabilità debole**: valuta se il sistema è capace di mantenere costante il tempo di esecuzione per ogni processore all'aumentare della dimensione del problema.

Una buona scalabilità deve garantire che le risorse aggiuntive siano realmente importanti al fine di risolvere il problema, e che non sprechino potenza di calcolo.

5.3.1 Metriche di Valutazione delle Performance

Le performance di un sistema HPC sono comunemente valutate attraverso alcune metriche fondamentali:

- **Speedup**: indica quanto più velocemente un programma viene eseguito in parallelo rispetto all'esecuzione sequenziale [14].

  $$S(p) = \frac{T_1}{T_p}$$

  dove T_1 è il tempo di esecuzione con un solo processore e T_p con p processori.

- **Efficienza**: misura quanto l'algoritmo sfrutta il parallelismo del calcolatore.

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

- **Throughput**: rappresenta il numero di operazioni completate in un determinato intervallo di tempo.

5.3.2 Limiti Teorici: Legge di Amdahl

La **legge di Amdahl** stabilisce un limite teorico alla velocità di esecuzione che è possibile ottenere con l'elaborazione in parallelo. Essa mette in evidenzia che la parte di codice non parallelizzabile diminuisce il guadagno complessivo:

$$S(p) = \frac{1}{a + \frac{(1-a)}{p}}$$

dove a è la frazione parallelizzabile e p il numero di processori [14].

5.3.3 Fattori che influenzano le prestazioni

Per progettare un'applicazione efficiente in parallelo, è importante tenere a mente diversi fattori:

- **Comunicazione tra processori**: può introdurre ritardi significativi, soprattutto se non implementata in modo corretto.
- **Accesso alla memoria**: l'accesso a dati condivisi da più processori può peggiorare le prestazioni.
- **Bilanciamento del carico**: una distribuzione errata dei compiti può lasciare alcuni processori inattivi e sprecare potenza di calcolo.
- **Overhead di gestione**: il coordinamento tra processi e thread può introdurre un carico computazionale maggiore.

Progettare attentamente un algoritmo e distribuire in modo equo il lavoro è fondamentale nei sistemi HPC per ottenere alte prestazioni.

Capitolo 6

Ottimizzazione e Parallelizzazione

6.1 Proposte di Parallelizzazione del Codice

Precedentemente, nell'analisi del framework numerico in versione sequenziale, è stato evidenziato come lo schema Leap-Frog rappresenti la componente computazionalmente più costosa rispetto all'inizializzazione delle condizioni iniziali e alla costruzione della griglia spazio-temporale.

L'obiettivo di questa sezione è quindi lo sviluppo di una versione parallelizzata del framework numerico, realizzata tramite l'impiego della libreria OpenMP. La parallelizzazione è stata ottenuta applicando il costrutto **#pragma omp parallel for** ai cicli responsabili delle seguenti operazioni:

- generazione della condizione iniziale;
- calcolo del primo passo temporale;
- iterazione temporale secondo lo schema Leap-Frog;
- valutazione finale dell'errore L2.

In particolare, è stata adottata la clausola schedule(static) al fine di ottenere un bilanciamento più efficace del carico tra i thread e contenere l'overhead, soprattutto in presenza di griglie di dimensione ridotta. Per il calcolo dell'errore L2 finale, è stato inoltre utilizzato il costrutto reduction(+:err) in modo da garantire una somma corretta e thread-safe.

La scelta di OpenMP si è rivelata particolarmente adatta in quanto consente di introdurre la parallelizzazione in modo incrementale, con un impatto minimo sulla struttura esistente del codice e un overhead ridotto rispetto ad approcci alternativi, come ad esempio MPI.

I risultati ottenuti mostrano un netto miglioramento delle prestazioni rispetto alla versione sequenziale. In prospettiva futura, ulteriori ottimizzazioni potrebbero essere apportate tramite una gestione più raffinata dei thread oppure estendendo il framework verso architetture distribuite.
Successivamente, è possibile visionare lo pseudocodice del framework numerico parallelizzato grazie all'utilizzo della libreria OpenMP.

Algorithm 2 Pseudocodice del Framework Numerico Parallelizzato con OpenMP

Input: dominio L_x, L_y, tempo T, velocità c, lista $\{h_k\}$
for ogni risoluzione spaziale h **do**
    // — Generazione della griglia per vari valori di risoluzione h —
    Calcolare $N_x = L_x/h$, $N_y = L_y/h$
    Calcolare $dx = L_x/N_x$, $dy = L_y/N_y$
    Calcolare dt da condizione CFL
    Determinare il numero di passi temporali $N_t = T/dt$ // con dt adattato per avere $t = T$ al passo $n = N_t$
    Costruire vettori x_i e y_j
    // — Inizializzazione della condizione iniziale —
    #pragma omp parallel for schedule(static)
    for ogni punto (i, j) **do**
        $u^0_{i,j} = \sin(\pi x_i)\sin(\pi y_j)$
    end for
    // — Primo passo temporale —
    #pragma omp parallel for schedule(static)
    for ogni punto interno (i, j) **do**
        $u^1_{i,j} = u^0_{i,j} + \frac{1}{2}c^2 dt^2 \nabla^2 u^0_{i,j}$
    end for
    Imporre $u^1_{i,j} = 0$ sui bordi
    // — Evoluzione temporale Leap-Frog —
    for $n = 1$ fino a $N_t - 1$ **do**
        #pragma omp parallel for schedule(static)
        for ogni punto interno (i, j) **do**
            Calcolare $\nabla^2 u^n_{i,j}$
            $u^{n+1}_{i,j} = 2u^n_{i,j} - u^{n-1}_{i,j} + c^2 dt^2 \nabla^2 u^n_{i,j}$
        end for
        Imporre $u^{n+1}_{i,j} = 0$ sui bordi
    end for
    // — Calcolo dell'errore L2 —
    $errore = 0$
    #pragma omp parallel for reduction(+:errore)
    for ogni (i, j) **do**
        $errore+ = \left(u^{N_t}_{i,j} - u_{esatto}(x_i, y_j, T)\right)^2$
    end for
    $errore = \sqrt{errore/((N_x + 1)(N_y + 1))}$
    if non primo h **then**
        Calcola ordine $p = \log(\text{errore}_{prev}/\text{errore})/\log(h_{prev}/h)$
    end if
    Salvare $\text{errore}_{prev} = \text{errore}$
end for

6.2 Analisi delle Prestazioni

6.2.1 Setup Sperimentale

I test riguardanti le analisi delle prestazioni sono stati eseguiti su un calcolatore equipaggiato con sistema operativo Linux Mint 22.1 "Xia" (derivato da Ubuntu 24.04), dotato di un processore **AMD Ryzen 5 3500U** con **4 core fisici e 8 thread logici**, e **16 GB di memoria RAM**.

Il compilatore utilizzato per la compilazione del framework numerico in C è `gcc` in versione 13.3.0, con supporto alle direttive di parallelizzazione OpenMP attivato grazie al flag `-fopenmp`.

Durante l'esecuzione, il numero di thread OpenMP è stato variato tra 2, 4 e 8 per analizzare l'impatto della parallelizzazione sulle prestazioni rispetto alla versione sequenziale. I tempi d'esecuzione sono stati rilevati utilizzando la funzione `omp_get_wtime()`, con cronometraggio separato per ciascuna fase significativa dell'algoritmo.

6.2.2 Metodologia di Misurazione

Al fine di valutare e confrontare le prestazioni del framework numerico, prima in sequenziale e successivamente in parallelo, sono stati raccolti i tempi d'esecuzione relativi alle varie fasi più importanti del codice :

- **Costruzione della griglia spazio-temporale**
- **Inizializzazione delle condizioni iniziali**
- **Calcolo del primo passo temporale**
- **Esecuzione dello schema Leap-Frog**

Queste misurazioni sono state realizzate grazie all'utilizzo della funzione `omp_get_wtime()`, ottenuta grazie alla libreria OpenMP.

Per garantire l'affidabilità delle misurazioni, ogni test è stato eseguito molteplici volte e i tempi riportati corrispondono alla media delle esecuzioni.

Al fine di analizzare la scalabilità del codice parallelizzato, è stato variato il numero di thread OpenMP cosi da monitorare la variazione del tempo di esecuzione totale e di ciascuna fase chiave precedentemente specificata.

6.2.3 Risultati Numerici - Versione Parallela

Per valutare l'efficacia della parallelizzazione del framework numerico, sono state eseguite simulazioni utilizzando un numero variabile di thread OpenMP, in particolare 2,4 e 8, mantenendo invariati i parametri della simulazione.

Successivamente, sono riportati i tempi medi di esecuzione per ciascuna fase significativa dell'algoritmo per diversi valori di h, separando il tempo impiegato per la costruzione della griglia e l'evoluzione temporale secondo lo schema Leap-Frog. I valori sono confrontati tra l'utilizzo di 2 thread, 4 thread e 8 thread precedentemente all'interno della funzione **Main** del framework numerico.

La maggior parte del tempo è impiegata per l'evoluzione temporale tramite lo schema Leap-Frog, che risulta anche la parte che beneficia maggiormente della parallelizzazione. Il tempo di esecuzione si riduce significativamente con 2 thread, ma quando si utilizzano 4 thread l'efficienza tende ad aumentare ancor di più, in particolare per griglie di grandi dimensioni, a causa della corretta gestione dei thread nella versione parallela. Con l'impiego di 8 thread le prestazioni migliorano confermando la corretta implementazione del framework numerico.

Le figure 6.1, 6.2, 6.3 e 6.4 mostrano gli output a terminale da cui sono stati estratti i dati presentati nella Tabella 6.1.

```
== Simulazione 2D Equazione delle Onde (Parallela) ==
Dominio: [0,1.0]x[0,1.0], Tempo T = 2.0s, Velocità c = 1.0


                ANALISI DI CONVERGENZA (PARALLELA)

h          dt          Errore L2       Ordine     Nt         (Nx+1)*(Ny+1)   [Griglia | C.Iniziali | 1° Step | Leap-Frog] (s)
0.1000     0.062500    1.891562e-03    -          32         121             [0.000070 | 0.000121 | 0.000010 | 0.000326]
0.0500     0.031746    4.341028e-04    2.12       63         441             [0.000372 | 0.000035 | 0.000038 | 0.002736]
0.0250     0.015873    1.108885e-04    1.97       126        1681            [0.002403 | 0.000136 | 0.000192 | 0.014475]
0.0125     0.007937    2.804643e-05    1.98       252        6561            [0.008019 | 0.000166 | 0.000249 | 0.071239]

Tempo di esecuzione totale: 0.103396 secondi
```

Figura 6.1: Output terminale con parallelizzazione a 1 thread.

```
== Simulazione 2D Equazione delle Onde (Parallela) ==
Dominio: [0,1.0]x[0,1.0], Tempo T = 2.0s, Velocità c = 1.0


                ANALISI DI CONVERGENZA (PARALLELA)

h          dt          Errore L2       Ordine     Nt         (Nx+1)*(Ny+1)   [Griglia | C.Iniziali | 1° Step | Leap-Frog] (s)

0.1000     0.062500    1.891562e-03    -          32         121             [0.000070 | 0.000183 | 0.000008 | 0.000229]
0.0500     0.031746    4.341028e-04    2.12       63         441             [0.000420 | 0.000019 | 0.000022 | 0.001561]
0.0250     0.015873    1.108885e-04    1.97       126        1681            [0.002536 | 0.000106 | 0.000097 | 0.013179]
0.0125     0.007937    2.804643e-05    1.98       252        6561            [0.008061 | 0.000230 | 0.000296 | 0.047445]

Tempo di esecuzione totale: 0.077702 secondi
```

Figura 6.2: Output terminale con parallelizzazione a 2 thread.

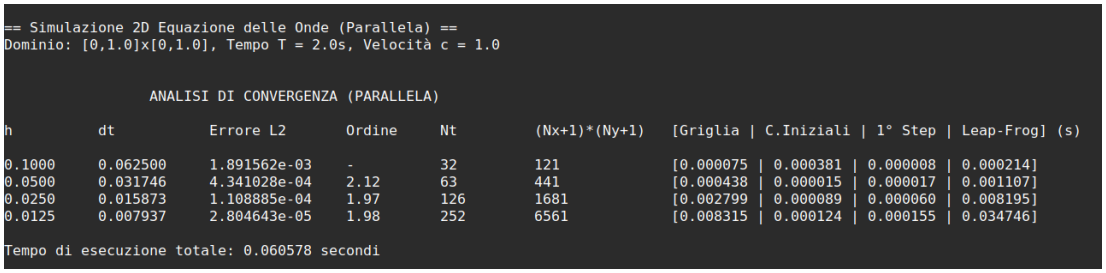

Figura 6.3: Output terminale con parallelizzazione a 4 thread.

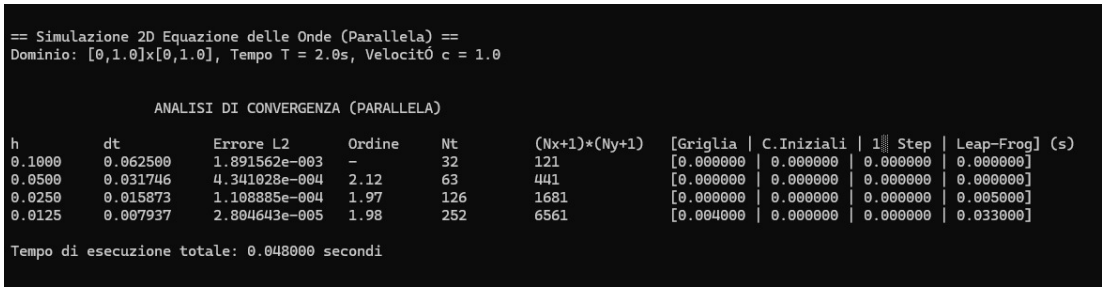

Figura 6.4: Output terminale con parallelizzazione a 8 thread.

Il test con 8 thread è stato condotto su una macchina diversa, dotata di un hardware più performante, al fine di supportare un numero maggiore di thread. I risultati sono stati comunque inclusi per valutare il comportamento del framework in condizioni di parallelizzazione più estesa.

Infine, è stato confermato che l'errore L^2 rimane invariato a prescindere dal numero di thread utilizzati, garantendo la correttezza e l'affidabilità del framework numerico parallelizzato.

In conclusione, grazie all'impiego di OpenMP si è ottenuta una significativa riduzione dei tempi di esecuzione per griglie di dimensioni medio-grandi, senza compromettere la qualità della soluzione. Una possibile ottimizzazione futura potrà riguardare la gestione dinamica del carico di lavoro tra i thread, al fine di minimizzare ulteriormente l'overhead.

6.3 Confronto tra sequenziale e parallelo

In questa sezione si analizzano i risultati ottenuti confrontando direttamente la versione sequenziale del framework numerico con quella parallelizzata mediante OpenMP.

Per griglie di dimensioni ridotte, la versione sequenziale presenta tempi di esecuzione ridotti dove il tempo totale si aggira su 0.1 secondi. In alcuni casi, l'utilizzo della parallelizzazione può generare dell'overhead che annullerebbe i benefici attesi, rendendo la versione sequenziale più efficiente.

All'aumentare della risoluzione spaziale, il tempo di esecuzione cresce in modo significativo, in particolare durante l'applicazione dello schema Leap-Frog, che rappresenta la parte più onerosa del framework.
Nella versione parallelizzata con 2 thread, si osserva un miglioramento delle prestazioni favorendo le fasi chiave dell'algoritmo soprattutto con griglie più dense, poiché il carico di lavoro viene diviso efficacemente.

Durante l'esecuzione del framework numerico con 4 thread, si è evidenziato un comportamento migliore rispetto alla versione parallelizzata con 2 thread e alla versione sequenziale. Questo è dovuto alla corretta implementazione per la gestione dei thread, evidenziando che un numero maggiore di thread garantisce di ottenere tempi di esecuzione minori.

Invece, con l'utilizzo di 8 thread, si osserva un ulteriore miglioramento dei tempi di esecuzione rispetto alle configurazioni con 2 e 4 thread, soprattutto per le griglie con risoluzione più elevata. Il carico di lavoro risulta ben distribuito tra i thread, consentendo un'accelerazione più significativa delle fasi computazionalmente intensive, come l'evoluzione secondo lo schema Leap-Frog. Tuttavia, per griglie di dimensioni ridotte, l'overhead di gestione dei thread può ancora limitare i benefici, rendendo meno evidente il vantaggio della parallelizzazione con 8 thread.

La parallelizzazione non ha alterato la soluzione del framework numerico, infatti l'errore L^2 finale e l'ordine di convergenza risultano invariati rispetto alla soluzione ottenuta in sequenziale, confermando la corretta implementazione del codice.

Griglia ($N \times N$)	N_t	**1 Thread (s)**		**2 Thread (s)**		**4 Thread (s)**	
		Griglia	Leap-Frog	Griglia	Leap-Frog	Griglia	Leap-Frog
11×11	32	0.000070	0.000326	0.000070	0.000229	0.000075	0.000214
21×21	63	0.000372	0.002736	0.000420	0.001561	0.000438	0.001107
41×41	126	0.002403	0.014475	0.002536	0.013179	0.002799	0.008195
81×81	252	0.008019	0.071239	0.008061	0.047445	0.008135	0.034746

Tabella 6.1: Tempi medi (s) per la costruzione della griglia e per l'evoluzione Leap-Frog, con 1, 2 e 4 thread OpenMP.

La Tabella 6.1 riassume i tempi medi necessari per la costruzione della griglia e per l'e-

voluzione secondo lo schema Leap-Frog, considerando l'esecuzione con 1, 2 e 4 thread. I dati confermano che per griglie di piccole dimensioni (ad esempio 11×11 o 21×21), il tempo di esecuzione è talmente ridotto che l'overhead legato alla gestione dei thread può annullare o persino peggiorare le prestazioni rispetto alla versione sequenziale.

Diversamente, all'aumentare della risoluzione spaziale, il beneficio della parallelizzazione diventa evidente. Per la griglia 81×81, il tempo associato allo schema Leap-Frog scende da circa 70 ms con un solo thread a circa 35 ms con 4 thread, mostrando quasi un dimezzamento del tempo di calcolo. Questo comportamento riflette il maggiore carico computazionale disponibile per ciascun thread, che consente una suddivisione più efficiente del lavoro e quindi una riduzione complessiva dei tempi.

La configurazione con 8 thread, non riportata nella tabella, ha mostrato ulteriori miglioramenti nei test condotti, ma è stata esclusa per coerenza con i dati non significativi rilevati nelle griglie a bassa risoluzione, dove i tempi risultavano trascurabili. Il confronto sintetico dei tempi totali, incluso nella Tabella 6.2, conferma comunque la validità della scalabilità fino a 8 thread per griglie sufficientemente grandi.

Configurazione	**Tempo Totale (s)**
Sequenziale	0.102948
Parallelizzata - 1 Thread	0.103396
Parallelizzata – 2 Thread	0.077702
Parallelizzata - 4 Thread	0.060578
Parallelizzata - 8 Thread	0.048000

Tabella 6.2: Tempo totale medio di esecuzione del framework numerico.

In questa tabella sono riportati i tempi d'esecuzione totali della funzione **analisi_convergenza**, infatti è possibile notare che grazie alla parallelizzazione si è riusciti a diminuire ogni volta i tempi per eseguire il framework.

In sintesi, la versione parallelizzata risulta vantaggiosa principalmente per griglie ad alta risoluzione, dove il carico computazionale per thread è sufficiente a giustificare il parallelismo introdotto.
Un possibile sviluppo futuro potrebbe prevedere l'introduzione di una parallelizzazione ibrida sfruttando OpenMP e MPI.

Questi risultati confermano che l'approccio parallelo, se ben calibrato, può portare a significativi miglioramenti in termini di efficienza, pur richiedendo attenzione nella gestione delle risorse computazionali.

Capitolo 7

Risultati e Validazione

7.1 Evoluzione della Soluzione nel Tempo

Al fine di validare il comportamento del metodo numerico implementato, è stata effettuata un'analisi dell'evoluzione della soluzione nel tempo. La condizione iniziale è rappresentata da una perturbazione sinusoidale definita come:

$$u(x, y, 0) = \sin(\pi x) \sin(\pi y)$$

rilasciata da ferma.

Nel caso in cui non ci sono forzanti, la soluzione esatta dell'equazione delle onde bidimensionale è data da:

$$u_{\text{esatto}}(x, y, t) = \sin(\pi x) \sin(\pi y) \cos(\sqrt{2}\pi c t)$$

dove l'oscillazione simmetrica e la propagazione regolare della perturbazione iniziale confermano la correttezza dello schema Leap-Frog e la buona conservazione delle proprietà fisiche della soluzione.

7.2 Analisi dell'Errore e Convergenza

Per effettuare una corretta valutazione dell'accuratezza del metodo numerico implementato, è stato necessario svolgere un confronto tra la soluzione numerica u_{num} e la soluzione analitica esatta u_{esatto}:

$$u_{\text{esatto}}(x, y, t) = \sin(\pi x) \sin(\pi y) \cos(\sqrt{2}\pi c t)$$

dove il confronto è stato fatto al tempo finale $T = 2$, calcolando la norma L^2 dell'errore:

$$\|u_{\text{num}} - u_{\text{esatto}}\|_{L^2} = \sqrt{\frac{1}{(N_x + 1)(N_y + 1)} \sum_{i,j} \left(u_{i,j}^{\text{num}} - u_{i,j}^{\text{esatto}}\right)^2}$$

Sono stati effettuati esperimenti per diverse risoluzioni spaziali $h = \frac{1}{10}, \frac{1}{20}, \frac{1}{40}, \frac{1}{80}$, rispettando in ogni caso la condizione di stabilità CFL. Per ciascuna griglia è stato calcolato anche l'ordine di convergenza numerico, stimato con la formula:

$$\text{Ordine} = \frac{\log\left(\frac{E_{h_1}}{E_{h_2}}\right)}{\log\left(\frac{h_1}{h_2}\right)}$$

I risultati ottenuti sono riportati nella Tabella 7.1.

Tabella 7.1: Errore L2 e ordine di convergenza per diverse risoluzioni spaziali

h	dt	Errore L^2	Ordine
0.1000	0.0625	1.8916×10^{-3}	–
0.0500	0.0317	4.3410×10^{-4}	2.12
0.0250	0.0159	1.0888×10^{-4}	1.97
0.0125	0.0079	2.8046×10^{-5}	1.98

Grazie alla tabella si evidenzia una decrescita dell'errore al diminuire di h (Passo Spaziale), confermando che lo schema Leap-Frog, applicato per risolvere l'equazione delle onde 2D possiede un comportamento del secondo ordine.

Per la visualizzazione grafica dell'andamento dell'errore, è stato tracciato un diagramma in scala log-log (Figura 7.1) dove è mostrato anche il confronto con la curva teorica di riferimento di ordine 2.

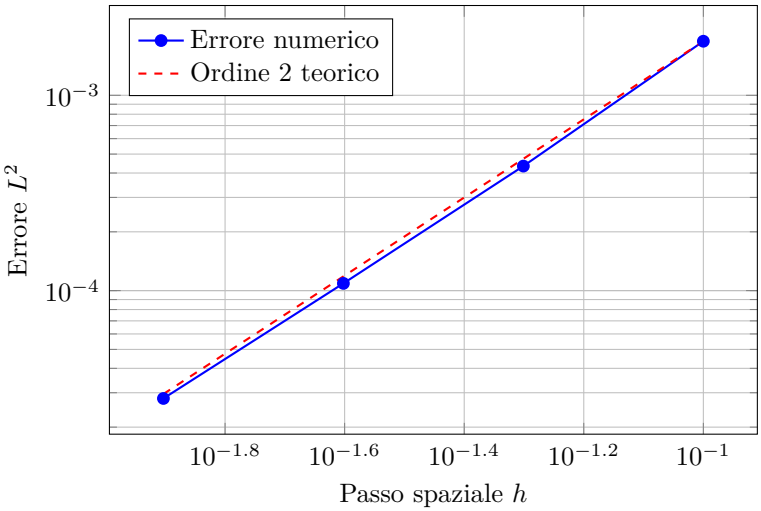

Figura 7.1: Confronto tra l'errore numerico e la curva teorica di ordine 2.

Questi risultati confermano che l'implementazione è corretta e che il metodo Leap-Frog mantiene le proprietà di accuratezza previste dalla teoria per il caso trattato.

7.3 Stabilità e Robustezza del Metodo

La stabilità del metodo numerico utilizzato è garantita dall'applicazione della condizione di stabilità di Courant-Friedrichs-Lewy (CFL), fondamentale per schemi espliciti come Leap-Frog. Nel contesto bidimensionale, tale condizione assume la forma:

$$\Delta t \leq \frac{1}{c\sqrt{\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2}}}$$

Nel framework sviluppato, il passo temporale Δt è stato scelto come:

$$\Delta t = 0{,}9 \cdot \Delta t_{\max}$$

dove $\Delta t_{\max}$ è il limite massimo consentito dalla CFL. Questa scelta garantisce una certa tolleranza numerica, evitando l'instabilità tipica degli schemi espliciti quando la condizione viene violata.

Durante tutte le simulazioni numeriche, il comportamento della soluzione era stabile anche per le griglie con passaggi spaziali molto fini senza false vibrazioni o insorgenza di divergenza. Ciò conferma la solidità dell'implementazione in termini sia di stabilimento che di coerenza con i modelli matematici.

L'analisi dell'energia discreta ha mostrato una buona conservazione nel tempo, confermando che il metodo Leap-Frog, nonostante sia esplicito, conserva efficacemente l'energia del sistema nei casi lineari e non dissipativi.

Infine, l'errore percentuale rimane dell'ordine di 10^{-4}, anche dopo molti passi temporali.

7.4 Prestazioni e Scalabilità

Al fine di valutare l'efficienza computazionale del framework sviluppato, è stata comparata la versione sequenziale e la versione parallelizzata tramite `OpenMP`. Il confronto è stato effettuato su una macchina multi-core, dove è stato misurato il tempo di esecuzione per le diverse risoluzioni spaziali.

I risultati sperimentali ottenuti mostrano che:

- Per griglie piccole, il parallelismo introduce un overhead che minimizza i benefici ottenibili dalla parallelizzazione.
- Al crescere della dimensione della griglia, la versione parallela diventa più veloce in termini di tempo.
- In accordo con la Legge di Amdahl, il guadagno in velocità tende a dissiparsi oltre un certo numero di thread.

Come previsto dalla legge di Amdahl [14], è possibile osservare che per griglie sufficientemente grandi, lo speedup tende ad avvicinarsi al numero di core utilizzati :

$$S(p) = \frac{1}{f + \frac{1-f}{p}}$$

dove p è il numero di thread e f è la parte di codice non parallelizzabile.
In particolare, nelle misure effettuate, il valore di f si è mantenuto tra il 10% e il 20%, coerente con la struttura del codice Leap-Frog, dove i loop principali sono stati parallelizzati, ma alcune operazioni sono rimaste seriali.

L'utilizzo di `OpenMP` ha dunque permesso di ottenere un miglioramento significativo delle prestazioni, mantenendo la stessa accuratezza numerica della versione sequenziale.

7.5 Validazione Finale della Soluzione Numerica

Grazie alle analisi trattate nei paragrafi precedenti, è stato possibile affermare che la soluzione numerica è stata validata seguendo tre importanti criteri :

1. **Accuratezza numerica**: il confronto con la soluzione analitica ha mostrato una chiara convergenza dell'errore L^2 con ordine prossimo al secondo, come previsto dalla teoria per lo schema Leap-Frog. I risultati ottenuti confermano l'efficacia della discretizzazione spazio-temporale adottata.

2. **Stabilità numerica**: il passo temporale è stato scelto in accordo con la condizione di stabilità di Courant-Friedrichs-Lewy (CFL), e in tutte le simulazioni non si sono verificate instabilità o divergenze, neanche per griglie ad alta risoluzione.

3. **Conservazione dell'energia**: l'energia discreta del sistema è risultata ben conservata nel tempo, con un errore relativo dell'ordine di 10^{-4}. Ciò dimostra la robustezza dello schema utilizzato nel contesto di equazioni iperboliche non dissipative.

Infine, il framework numerico ha fornito ottimi risultati, mantenendo un comportamento stabile, accurato ed efficiente nella risoluzione dell'equazione dell'onda bidimensionale.

Inoltre, tale implementazione è un'ottima base per future implementazioni sia in contesti più sofisticati, sia per un ulteriore miglioramento in termini di efficienza computazionale.

Capitolo 8

Conclusioni e Sviluppi Futuri

8.1 Sintesi dei Risultati Ottenuti

Lo sviluppo e l'analisi di un framework numerico per risolvere l'equazione delle onde bidimensionale in un dominio rettangolare con condizioni al contorno è sempre stato l'obiettivo di questa tesi.

Il metodo utilizzato si basa sullo schema esplicito Leap-Frog, implementato in linguaggio `C` e successivamente parallelizzato con `OpenMP` al fine di migliorare le prestazioni su architetture multicore. La stabilità dello schema è stata garantita mediante l'applicazione della condizione di Courant-Friedrichs-Lewy (CFL).

Le simulazioni numeriche hanno mostrato:

- una buona conservazione dell'energia discreta nel tempo;
- un comportamento stabile anche su griglie ad alta risoluzione;
- un errore numerico in norma L^2 in linea con l'ordine di accuratezza teorico;
- un significativo miglioramento dell'esecuzione nei test parallelizzati su griglie medio-grandi.

Quindi, il framework ha dimostrato di essere stabile, accurato ed efficiente per la risoluzione numerica dell'equazione delle onde 2D, fornendo una base affidabile per successive estensioni.

8.2 Limiti dell'Approccio Attuale

I risultati ottenuti grazie allo sviluppo di questo framework numerico sono molto soddisfacenti, però tale algoritmo presenta delle limitazioni dovute alle scelte adottate:

- Il dominio considerato è rettangolare e regolare, con condizioni al contorno di tipo Dirichlet omogenee. Non sono stati trattati domini irregolari o condizioni al contorno più generali.
- La formulazione numerica non include termini sorgente, effetti dissipativi o fenomeni non lineari, limitando l'applicabilità a sistemi fisici più complessi.
- L'implementazione sfrutta unicamente il parallelismo condiviso tramite `OpenMP`, risultando non ottimale per sistemi con acceleratori (GPU).
- Il metodo Leap-Frog, pur essendo semplice ed efficiente, può introdurre effetti di dispersione numerica su tempi lunghi o per onde ad alta frequenza, fenomeni che non sono stati analizzati nel dettaglio.
- L'output della simulazione è stato limitato all'analisi numerica dell'errore e dell'energia. Non è stata inclusa una fase di post-processing avanzato o esportazione grafica automatizzata.

Tali limitazioni non compromettono la validità del lavoro svolto, ma aprono la strada a possibili sviluppi futuri per rendere il framework più generale, potente ed estensibile.

8.3 Possibili Estensioni e Lavori Futuri

Il framework numerico sviluppato rappresenta una base solida e ben strutturata per la risoluzione dell'equazione delle onde. Tuttavia, sono numerose le possibili estensioni che ne migliorerebbero la versatilità, tra cui:

- **Estensione a domini e condizioni più generali**: l'adattamento del metodo a domini non rettangolari, con mesh strutturate o non uniformi, e condizioni al contorno diverse (Neumann, Robin, miste) permetterebbe di modellare una più ampia classe di problemi fisici.
- **Inserimento di termini sorgente o effetti dissipativi**: l'introduzione di forzanti temporali o spaziali, o di termini di damping, amplierebbe le potenzialità del codice verso modelli più realistici in ambito ingegneristico o geofisico.
- **Ottimizzazione delle prestazioni**: l'uso di parallelizzazione distribuita (MPI) o l'implementazione su architetture GPU (CUDA, OpenCL) consentirebbe di affrontare problemi su scala ancora maggiore in termini di griglia e tempo simulato.
- **Utilizzo di librerie scientifiche avanzate**: l'integrazione con tool come PETSc, deal.II o Trilinos potrebbe migliorare la gestione della memoria, la modularità del codice e le performance computazionali.
- **Interfacce grafiche e visualizzazione**: lo sviluppo di strumenti di post-processing o di un'interfaccia grafica renderebbe il framework più accessibile anche a utenti non esperti di programmazione.

Questi sviluppi non solo estenderebbero l'ambito di applicazione del framework, ma lo renderebbero anche più efficiente, modulare e utilizzabile in contesti scientifici e professionali più ampi.

Tali risultati mostrano la validità complessiva del framework proposto, confermandone l'efficacia come base per future applicazioni numeriche in ambito scientifico e ingegneristico.

Capitolo 9

Appendice

9.1 Codice Sorgente Sequenziale

Il codice sorgente eseguito in ambiente sequenziale per la risoluzione dell'equazione delle onde 2D utilizzando il metodo Leap-Frog è riportato di seguito. Per maggiori dettagli, si rimanda al Capitolo 4.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>

#define PI 3.141592653589793

// === Funzione iniziale: u(x,y,0) ===
double condizione_iniziale(double x, double y) {
    return sin(PI * x) * sin(PI * y);
}

// === Velocità iniziale (posta a zero) ===
double velocita_iniziale(double x, double y) {
    return 0.0;
}

// === Calcolo primo step temporale u^1 ===
double calcolo_primo_step(double u0, double laplaciano_u0, double v0, double dt,
    double c) {
    return u0 + dt * v0 + 0.5 * pow(c * dt, 2) * laplaciano_u0;
}

// === Leap-Frog: u^{n+1} = 2u^n - u^{n-1} + c^2 dt^2 Laplaciano ===
double calcolo_step_leap_frog(double u_current, double u_previous, double
    laplaciano, double dt, double c) {
```

```
    return 2 * u_current - u_previous + pow(c * dt, 2) * laplaciano;
}

// === Soluzione esatta u(x,y,t) ===
double soluzione_analitica(double x, double y, double t, double c) {
    return sin(PI * x) * sin(PI * y) * cos(sqrt(2.0) * PI * c * t);
}

void analisi_convergenza(double Lx, double Ly, double T, double c) {
    double h_values[] = {0.1, 0.05, 0.025, 0.0125};
    int num_h = sizeof(h_values) / sizeof(h_values[0]);
    double prev_err = 0.0;

    printf("\n\t\t ANALISI DI CONVERGENZA \n\n");
    printf("%-10s %-12s %-15s %-10s %-10s %-15s %-40s\n",
           "h", "dt", "Errore L2", "Ordine", "Nt", "(Nx+1)*(Ny+1)",
           "[Griglia | C.Iniziali | Leap-Frog] (s)");

    for (int k = 0; k < num_h; k++) {
        double t_start = omp_get_wtime();

        double h = h_values[k];
        int Nx = (int)(Lx / h);
        int Ny = (int)(Ly / h);
        double dx = Lx / Nx;
        double dy = Ly / Ny;

        // Calcolo dt con condizione CFL
        double dt_max = 1.0 / (c * sqrt(1.0 / (dx * dx) + 1.0 / (dy * dy)));
        double dt = 0.9 * dt_max;
        int Nt = (int)ceil(T / dt);
        dt = T / Nt;

        // Inizio cronometro per griglia
        double t_griglia_start = omp_get_wtime();

        // Allocazione coordinate spaziali
        double *x = (double *)malloc((Nx + 1) * sizeof(double));
        double *y = (double *)malloc((Ny + 1) * sizeof(double));
        for (int i = 0; i <= Nx; i++) x[i] = i * dx;
        for (int j = 0; j <= Ny; j++) y[j] = j * dy;

        // Allocazione matrice 3D: tempo × y × x
        double ***U = (double ***)malloc((Nt + 2) * sizeof(double **));
        for (int n = 0; n <= Nt + 1; n++) {
            U[n] = (double **)malloc((Ny + 1) * sizeof(double *));
            for (int j = 0; j <= Ny; j++) {
```

```
72                  U[n][j] = (double *)calloc(Nx + 1, sizeof(double));
73              }
74          }
75
76          double t_griglia_end = omp_get_wtime();
77          double t_iniziali_start = omp_get_wtime();
78
79          // Condizione iniziale: t=0
80          for (int j = 0; j <= Ny; j++)
81              for (int i = 0; i <= Nx; i++)
82                  U[0][j][i] = condizione_iniziale(x[i], y[j]);
83
84          // Primo passo: t=dt
85          for (int j = 1; j < Ny; j++) {
86              for (int i = 1; i < Nx; i++) {
87                  double laplaciano = (U[0][j + 1][i] - 2 * U[0][j][i] + U[0][j -
                        1][i]) / (dy * dy) +
88                                      (U[0][j][i + 1] - 2 * U[0][j][i] + U[0][j][i -
                                          1]) / (dx * dx);
89                  double v0 = velocita_iniziale(x[i], y[j]);
90                  U[1][j][i] = calcolo_primo_step(U[0][j][i], laplaciano, v0, dt, c);
91              }
92          }
93
94          // Condizioni al contorno
95          for (int i = 0; i <= Nx; i++) {
96              U[1][0][i] = U[1][Ny][i] = 0.0;
97          }
98          for (int j = 0; j <= Ny; j++) {
99              U[1][j][0] = U[1][j][Nx] = 0.0;
100         }
101
102         double t_iniziali_end = omp_get_wtime();
103         double t_leapfrog_start = omp_get_wtime();
104
105         // Schema Leap-Frog nel tempo
106         for (int n = 1; n < Nt; n++) {
107             for (int j = 1; j < Ny; j++) {
108                 for (int i = 1; i < Nx; i++) {
109                     double laplaciano = (U[n][j + 1][i] - 2 * U[n][j][i] + U[n][j -
                            1][i]) / (dy * dy) +
110                                         (U[n][j][i + 1] - 2 * U[n][j][i] + U[n][j][i
                                             - 1]) / (dx * dx);
111                     U[n + 1][j][i] = calcolo_step_leap_frog(U[n][j][i], U[n -
                            1][j][i], laplaciano, dt, c);
112                 }
113             }
```

```
114             // Condizioni al contorno
115             for (int i = 0; i <= Nx; i++) {
116                 U[n + 1][0][i] = U[n + 1][Ny][i] = 0.0;
117             }
118             for (int j = 0; j <= Ny; j++) {
119                 U[n + 1][j][0] = U[n + 1][j][Nx] = 0.0;
120             }
121         }
122 
123         double t_leapfrog_end = omp_get_wtime();
124 
125         // Errore L2 finale
126         double err = 0.0;
127         for (int j = 0; j <= Ny; j++) {
128             for (int i = 0; i <= Nx; i++) {
129                 double u_exact = soluzione_analitica(x[i], y[j], T, c);
130                 double diff = U[Nt][j][i] - u_exact;
131                 err += diff * diff;
132             }
133         }
134         err = sqrt(err / ((Nx + 1) * (Ny + 1)));
135 
136         double t_end = omp_get_wtime();
137 
138         // Output con nuovi campi
139         if (k == 0) {
140             printf("%-10.4f %-12.6f %-15.6e %-10s %-10d %-15d [%.6f | %.6f |
                %.6f]\n",
141                    h, dt, err, "-", Nt, (Nx + 1)*(Ny + 1),
142                    t_griglia_end - t_griglia_start,
143                    t_iniziali_end - t_iniziali_start,
144                    t_leapfrog_end - t_leapfrog_start);
145         } else {
146             double ordine = log(prev_err / err) / log(h_values[k - 1] / h);
147             printf("%-10.4f %-12.6f %-15.6e %-10.2f %-10d %-15d [%.6f | %.6f |
                %.6f]\n",
148                    h, dt, err, ordine, Nt, (Nx + 1)*(Ny + 1),
149                    t_griglia_end - t_griglia_start,
150                    t_iniziali_end - t_iniziali_start,
151                    t_leapfrog_end - t_leapfrog_start);
152         }
153 
154         prev_err = err;
155 
156         // Deallocazione memoria
157         for (int n = 0; n <= Nt + 1; n++) {
158             for (int j = 0; j <= Ny; j++) {
```

```
159                 free(U[n][j]);
160             }
161             free(U[n]);
162         }
163         free(U);
164         free(x);
165         free(y);
166     }
167
168     printf("\n");
169 }
170
171 int main() {
172     double Lx = 1.0, Ly = 1.0, T = 2.0, c = 1.0;
173
174     printf("\n== Simulazione 2D Equazione delle Onde ==\n");
175     printf("Dominio: [0,%.1f]x[0,%.1f], Tempo T = %.1fs, Velocità c = %.1f\n\n", Lx,
        Ly, T, c);
176
177     double start_time = omp_get_wtime();
178     analisi_convergenza(Lx, Ly, T, c);
179     double end_time = omp_get_wtime();
180
181     printf("Tempo di esecuzione totale: %.6f secondi\n\n", end_time - start_time);
182     return 0;
183 }
```

Listing 9.1: Codice sequenziale per la simulazione dell' equazione delle Onde 2D

9.2 Codice Sorgente Parallelizzato

Il codice sorgente eseguito in parallelo per la risoluzione dell'equazione delle onde 2D utilizzando il metodo Leap-Frog è riportato di seguito. Per maggiori dettagli, si rimanda al Capitolo 6.

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h> // Libreria OpenMP per la parallelizzazione

#define PI 3.141592653589793

// Funzione che definisce la condizione iniziale della simulazione
double condizione_iniziale(double x, double y) {
    return sin(PI * x) * sin(PI * y);
}

// Funzione che definisce la velocità iniziale (zero in questo caso)
double velocita_iniziale(double x, double y) {
    return 0.0;
}

// Calcolo del primo step temporale usando la condizione iniziale e il laplaciano
double calcolo_primo_step(double u0, double laplaciano_u0, double v0, double dt,
    double c) {
    return u0 + dt * v0 + 0.5 * pow(c * dt, 2) * laplaciano_u0;
}

// Calcolo dello step successivo usando lo schema Leap-Frog
double calcolo_step_leap_frog(double u_current, double u_previous, double
    laplaciano, double dt, double c) {
    return 2 * u_current - u_previous + pow(c * dt, 2) * laplaciano;
}

// Soluzione analitica della wave equation per confronto
double soluzione_analitica(double x, double y, double t, double c) {
    return sin(PI * x) * sin(PI * y) * cos(sqrt(2.0) * PI * c * t);
}

// Funzione principale per l'analisi di convergenza, con calcolo tempi di esecuzione
void analisi_convergenza(double Lx, double Ly, double T, double c) {
    // Array di passi spaziali da testare (risoluzioni diverse)
    double h_values[] = {0.1, 0.05, 0.025, 0.0125};
    int num_h = sizeof(h_values) / sizeof(h_values[0]);
    double prev_err = 0.0;

    // Intestazione tabella risultati
    printf("\n\t\t ANALISI DI CONVERGENZA (PARALLELA)\n\n");
```

```
    printf("%-10s %-12s %-15s %-10s %-10s %-15s %-60s\n",
           "h", "dt", "Errore L2", "Ordine", "Nt", "(Nx+1)*(Ny+1)",
           "[Griglia | C.Iniziali | 1° Step | Leap-Frog] (s)");

    // Ciclo sulle risoluzioni spaziali
    for (int k = 0; k < num_h; k++) {
        double h = h_values[k];
        int Nx = (int)(Lx / h); // Numero nodi in x
        int Ny = (int)(Ly / h); // Numero nodi in y
        double dx = Lx / Nx;    // Passo spaziale in x
        double dy = Ly / Ny;    // Passo spaziale in y

        // Calcolo dt massimo per stabilità (CFL condition)
        double dt_max = 1.0 / (c * sqrt(1.0 / (dx * dx) + 1.0 / (dy * dy)));
        double dt = 0.9 * dt_max;  // Sicurezza con fattore 0.9
        int Nt = (int)ceil(T / dt); // Numero step temporali
        dt = T / Nt; // dt corretto per coprire esattamente il tempo T

        // Misura tempo allocazione e inizializzazione griglia
        double t_griglia_start = omp_get_wtime();

        // Allocazione vettori spazio x e y
        double *x = (double *)malloc((Nx + 1) * sizeof(double));
        double *y = (double *)malloc((Ny + 1) * sizeof(double));
        for (int i = 0; i <= Nx; i++) x[i] = i * dx;
        for (int j = 0; j <= Ny; j++) y[j] = j * dy;

        // Allocazione matrice 3D U per soluzione (tempo, y, x)
        double ***U = (double ***)malloc((Nt + 2) * sizeof(double **));
        for (int n = 0; n <= Nt + 1; n++) {
            U[n] = (double **)malloc((Ny + 1) * sizeof(double *));
            for (int j = 0; j <= Ny; j++) {
                U[n][j] = (double *)calloc(Nx + 1, sizeof(double)); // azzera i
                    valori
            }
        }

        double t_griglia_end = omp_get_wtime();
        double t_iniziali_start = omp_get_wtime();

        // Calcolo della condizione iniziale parallelo per y e x
        #pragma omp parallel for schedule(static)
        for (int j = 0; j <= Ny; j++) {
            for (int i = 0; i <= Nx; i++) {
                U[0][j][i] = condizione_iniziale(x[i], y[j]);
            }
        }
```

```
88
89          double t_primo_step_start = omp_get_wtime();
90
91          // Primo step temporale con schema esplicito, calcolo parallelo
92          #pragma omp parallel for schedule(static)
93          for (int j = 1; j < Ny; j++) {
94              for (int i = 1; i < Nx; i++) {
95                  // Calcolo laplaciano 2D usando differenze finite centrali
96                  double laplaciano = (U[0][j + 1][i] - 2 * U[0][j][i] + U[0][j -
                        1][i]) / (dy * dy) +
97                                      (U[0][j][i + 1] - 2 * U[0][j][i] + U[0][j][i -
                                          1]) / (dx * dx);
98                  double v0 = velocita_iniziale(x[i], y[j]); // velocità iniziale
                        (zero)
99                  U[1][j][i] = calcolo_primo_step(U[0][j][i], laplaciano, v0, dt, c);
100             }
101         }
102
103         // Condizioni al contorno nulli (bordo del dominio)
104         for (int i = 0; i <= Nx; i++) {
105             U[1][0][i] = U[1][Ny][i] = 0.0;
106         }
107         for (int j = 0; j <= Ny; j++) {
108             U[1][j][0] = U[1][j][Nx] = 0.0;
109         }
110
111         double t_primo_step_end = omp_get_wtime();
112         double t_iniziali_end = t_primo_step_start;
113
114         double t_leapfrog_start = omp_get_wtime();
115
116         // Loop temporale per calcolare gli step successivi con Leap-Frog
117         for (int n = 1; n < Nt; n++) {
118             #pragma omp parallel for schedule(static)
119             for (int j = 1; j < Ny; j++) {
120                 for (int i = 1; i < Nx; i++) {
121                     // Calcolo laplaciano corrente
122                     double laplaciano = (U[n][j + 1][i] - 2 * U[n][j][i] + U[n][j -
                            1][i]) / (dy * dy) +
123                                         (U[n][j][i + 1] - 2 * U[n][j][i] + U[n][j][i
                                             - 1]) / (dx * dx);
124                     // Calcolo passo successivo con Leap-Frog
125                     U[n + 1][j][i] = calcolo_step_leap_frog(U[n][j][i], U[n -
                            1][j][i], laplaciano, dt, c);
126                 }
127             }
128
```

```
            // Condizioni al contorno nulli per il nuovo step
            for (int i = 0; i <= Nx; i++) {
                U[n + 1][0][i] = U[n + 1][Ny][i] = 0.0;
            }
            for (int j = 0; j <= Ny; j++) {
                U[n + 1][j][0] = U[n + 1][j][Nx] = 0.0;
            }
        }

        double t_leapfrog_end = omp_get_wtime();

        // Calcolo errore L2 rispetto alla soluzione analitica
        double err = 0.0;
        #pragma omp parallel for reduction(+:err) schedule(static)
        for (int j = 0; j <= Ny; j++) {
            for (int i = 0; i <= Nx; i++) {
                double u_exact = soluzione_analitica(x[i], y[j], T, c);
                double diff = U[Nt][j][i] - u_exact;
                err += diff * diff;
            }
        }
        err = sqrt(err / ((Nx + 1) * (Ny + 1)));

        // Stampa dei risultati con tempi di esecuzione delle varie fasi
        if (k == 0) {
            printf("%-10.4f %-12.6f %-15.6e %-10s %-10d %-15d [%.6f | %.6f | %.6f |
                %.6f]\n",
                   h, dt, err, "-", Nt, (Nx + 1)*(Ny + 1),
                   t_griglia_end - t_griglia_start,    // tempo allocazione griglia
                   t_iniziali_end - t_iniziali_start,  // tempo condizioni iniziali
                   t_primo_step_end - t_primo_step_start, // tempo primo step
                   t_leapfrog_end - t_leapfrog_start);     // tempo ciclo leapfrog
        } else {
            // Calcolo ordine di convergenza numerico
            double ordine = log(prev_err / err) / log(h_values[k - 1] / h);
            printf("%-10.4f %-12.6f %-15.6e %-10.2f %-10d %-15d [%.6f | %.6f | %.6f
                | %.6f]\n",
                   h, dt, err, ordine, Nt, (Nx + 1)*(Ny + 1),
                   t_griglia_end - t_griglia_start,
                   t_iniziali_end - t_iniziali_start,
                   t_primo_step_end - t_primo_step_start,
                   t_leapfrog_end - t_leapfrog_start);
        }

        prev_err = err;

        // Deallocazione della memoria
```

```
174         for (int n = 0; n <= Nt + 1; n++) {
175             for (int j = 0; j <= Ny; j++) {
176                 free(U[n][j]);
177             }
178             free(U[n]);
179         }
180         free(U);
181         free(x);
182         free(y);
183     }
184 
185     printf("\n");
186 }
187 
188 int main() {
189     double Lx = 1.0, Ly = 1.0, T = 2.0, c = 1.0;
190 
191     // Imposta il numero di thread per OpenMP
192     omp_set_num_threads(8);
193 
194     printf("\n== Simulazione 2D Equazione delle Onde (Parallela) ==\n");
195     printf("Dominio: [0,%.1f]x[0,%.1f], Tempo T = %.1fs, Velocità c = %.1f\n\n", Lx,
        Ly, T, c);
196 
197     // Misura tempo totale esecuzione
198     double start_time = omp_get_wtime();
199     analisi_convergenza(Lx, Ly, T, c);
200     double end_time = omp_get_wtime();
201 
202     printf("Tempo di esecuzione totale: %.6f secondi\n\n", end_time - start_time);
203     return 0;
204 }
```

Listing 9.2: Codice Parallelizzato per la simulazione dell' equazione delle Onde 2D

9.3 Istruzioni per la Compilazione ed Esecuzione

Per eseguire il Framework numerico dell'equazione delle onde bidimensionale in ambiente Linux, è fondamentale seguire i seguenti passaggi:

1. Salvare il codice sorgente in un file denominato, ad esempio, `Onde2D.c`
2. Aprire un terminale e posizionarsi nella directory contenente il file
3. Compilare il programma utilizzando il compilatore `gcc` con supporto per la libreria matematica e per OpenMP:

   ```
   gcc -fopenmp Onde2D.c -o simulazionepde -lm
   ```

   Questo comando genera un eseguibile chiamato `simulazionepde`.
4. Per eseguire il programma in modalità sequenziale:

   ```
   export OMP_NUM_THREADS=1
   ./simulazionepde
   ```
5. Per eseguire il programma in parallelo con 2, 4 o 8 thread:

   ```
   export OMP_NUM_THREADS=2   # oppure 4 per testare con 4 thread
   ./simulazionepde
   ```
6. Al termine dell'esecuzione, il programma mostrerà sul terminale:
   - i parametri iniziali della simulazione (dimensioni del dominio, tempo finale T, velocità c),
   - il passo spaziale h e il passo temporale dt calcolato in base alla condizione CFL,
   - i valori dell'errore nella norma L^2 per ciascuna risoluzione spaziale,
   - l'ordine numerico di convergenza stimato tra successive risoluzioni,
   - i tempi parziali di esecuzione per ogni fase della simulazione:
     - generazione della griglia,
     - inizializzazione delle condizioni iniziali,
     - primo passo temporale,
     - esecuzione dello schema Leap-Frog,
   - il tempo complessivo impiegato dalla funzione `analisi_convergenza` per essere eseguita.
7. In caso di esecuzione parallela, i tempi saranno influenzati dal numero di thread impostato: l'output riporterà comunque in dettaglio i tempi delle singole fasi, rendendo possibile il confronto tra esecuzione sequenziale e parallela.

Bibliografia

[1] G. Alzetta et al. *deal.II: Open Source Finite Element Library*. https://www.dealii.org. 2022.

[2] Satish Balay et al. *PETSc – Portable, Extensible Toolkit for Scientific Computation*. https://petsc.org. 2024.

[3] The MathWorks, Inc. *Partial Differential Equation Toolbox User's Guide*. Available at https://www.mathworks.com/help/pde. MathWorks. Natick, MA, USA, 2024.

[4] OpenCFD Ltd. *OpenFOAM: The Open Source CFD Toolbox*. https://www.openfoam.com/. 2023.

[5] BYU Physics Department. *Wave Equation in 2D using Leapfrog*. Accessed: 2025-05-30. 2019. URL: https://physics.byu.edu/courses/computational/docs/phys430/phys430python.pdf.

[6] Gilbert Strang. *Leapfrog Scheme for the Wave Equation*. Accessed: 2025-05-30. 2006. URL: https://math.mit.edu/classes/18.086/2006/am53.pdf.

[7] Lawrence F. Shampine e Mark W. Reichelt. «Stability of the Leapfrog/ Midpoint Method». In: *ResearchGate* (1999). Accessed: 2025-05-30. URL: https://www.researchgate.net/publication/256936235_Stability_of_the_leapfrogmidpoint_method.

[8] Arman Altybay et al. «A numerical method for solving the two-dimensional acoustic wave equation using ADI method on parallel hybrid systems». In: *Mathematics* 8.12 (2020), p. 2194. DOI: 10.3390/math8122194.

[9] Johan Arnoldy e Dede Adytia. «GPU-accelerated leapfrog integration for 2D wave equations using CUDA». In: *Proceedings of the 2019 International Conference on Computational Science*. Springer, 2019, pp. 310–321. DOI: 10.1007/978-3-030-22734-0_25.

[10] Hyun Park e Sang-Hoon Lee. «High-performance simulation of 2D wave propagation using CUDA and OpenMP hybrid parallelism». In: *Journal of Computational Physics* 498 (2024), p. 112456. DOI: 10.1016/j.jcp.2024.112456.

[11] Pasquale De Luca e Livia Marcellino. *Equazioni Differenziali alle Derivate Parziali (PDE) - Appunti del corso di Applicazioni di Calcolo Scientifico II*. Materiale didattico non ufficiale, A.A. 2024/2025. 2025.

[12] P. De Luca e L. Marcellino. *Analisi e Implementazione Numerica dell'Equazione delle Onde 2D*. Manoscritto non pubblicato. Mag. 2025.

[13] Prof. Pasquale De Luca Prof.ssa Livia Marcellino. *Introduzione all'HPC*. Dispensa del corso, Università degli Studi di Napoli Parthenope. 2024. URL: https://elearning.uniparthenope.it/pluginfile.php/373313/mod_resource/content/1/Lez1_introduzione_motivazioni.pdf.

[14] Gene M. Amdahl. «Validity of the single processor approach to achieving large scale computing capabilities». In: *AFIPS Conference Proceedings* (1967).