Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

SVILUPPO DI UNA LIBRERIA C++ PARALLELA DISTRIBUITA PER LA RISOLUZIONE DI PROBLEMI DI ALGEBRA LINEARE

Relatore:
Chiar.mo Prof.
MORENO MARZOLLA

Presentata da:
MICHELE RAVAIOLI

Correlatore:
Dott.ssa MARIA ROSARIA TONDI
Ing. GIAMPAOLO ZERBINATO

Sessione 11/12/2025
Anno Accademico 2024-2025

Indice

Sommario

L'obiettivo di questa tesi è la progettazione e implementazione di una libreria in C++ per il calcolo parallelo distribuito di problemi di algebra lineare (ad esempio operazioni tra matrici e risoluzione di sistemi lineari). Questa libreria, una volta sviluppata, sarà utilizzata dall'ente INGV [14] all'interno del codice per il calcolo del problema di inversione sismico-gravimetrica. Il loro codice originale, scritto in Fortran, fa uso delle routine della libreria *S*caLAPACK [23] per eseguire in modo efficiente calcoli paralleli su cluster: in particolare, ScaLAPACK implementa una decomposizione a blocchi ciclici (*block-cyclic distribution*) [1] delle matrici sui vari nodi e algoritmi progettati per macchine con memoria distribuita. Lo scopo della libreria realizzata per questa tesi (denominata *S*LAC - Scalable Linear Algebra for C++ [21]) è quello di proporre un'alternativa a questa libreria ormai consolidata come standard nell'ambito fisico-computazionale per problemi algebrici.

L'utilizzo del linguaggio C++ offre diversi vantaggi rispetto al Fortran: migliori opportunità di integrazione con altri tool e programmi, maggiore facilità d'uso e manutenzione, e uso di un linguaggio più affermato nel contesto software ingegneristico moderno. Il vero problema - e la reale sfida - nella creazione di una libreria di questo tipo risiede nella progettazione della libreria e nella logica di utilizzo: come ScaLAPACK, anche SLAC si propone di adottare la distribuzione ciclica a blocchi per suddividere le matrici sui vari nodi del cluster. La distribuzione ciclica a blocchi non solo consente di distribuire il carico di lavoro in modo uniforme fra i nodi, ma fa sì che ogni nodo conservi solo una porzione della matrice globale, con conseguente utilizzo più efficiente della memoria locale, a costo di comunicazioni intra-nodo e fra i nodi per coordinamento e condivisione dei dati necessari ai calcoli. Questo approccio rappresenta un compromesso fra uso della memoria locale, flessibilità e comunicazione di rete.

La libreria si appoggia su tecnologie consolidate ed efficienti: per le operazioni matriciali a livello locale viene utilizzata la libreria *E*igen [12], implementata in C++ e progettata per manipolare matrici, vettori, decomposizioni e operazioni numeriche in modo efficiente. Per quanto riguarda le comunicazioni tra nodi, invece, viene impiegato il modello *M*PI (Message-Passing Interface) [9], standard de facto per la parallelizzazione su cluster con memoria distribuita.

In questa tesi verranno presentati i fondamenti teorici, la struttura architetturale della libreria, gli algoritmi implementati, e le ottimizzazioni adottate per ottenere una libreria efficiente e facile da comprendere e utilizzare.

Capitolo 1

Introduzione

1.1 Contesto e obiettivi

Il problema affrontato in questa tesi riguarda l'inversione geofisica. L'inversione geofisica mira a ricostruire modelli delle proprietà fisiche della Terra (come densità, conducibilità e velocità delle onde elastiche nel mezzo) in grado di riprodurre adeguatamente le anomalie osservate nei dati acquisiti tramite indagini geofisiche quali rilievi gravimetrici, di resistività in corrente continua e sismici, rimanendo al contempo coerenti con le informazioni geologiche. Le proprietà fisiche sono legate alla composizione delle rocce, alla struttura e allo stato fisico del mezzo. Di conseguenza, i modelli di proprietà fisiche ottenuti mediante inversione forniscono informazioni essenziali non solo per l'esplorazione mineraria, ma anche per la comprensione più generale della struttura e dei processi del pianeta Terra.

A causa dell'incertezza dei dati e di altri aspetti intrinseci del problema inverso geofisico sottodeterminato, in cui il numero di osservazioni è inferiore a quello dei parametri fisici da determinare, esiste un numero infinito di modelli che possono adattarsi ai dati geofisici con il grado di accuratezza desiderato [17]: il problema non è univoco. Informazioni aggiuntive sono quindi essenziali per ottenere una soluzione univoca. L'integrazione di conoscenze geologiche pregresse e la combinazione di diversi tipi complementari di dati geofisici raccolti sulla stessa area terrestre possono ridurre l'ambiguità e migliorare i risultati dell'inversione, portando a modelli della Terra più affidabili. Phillips [19], Williams [26] e Farquharson et al. [8, 7] forniscono esempi di come l'integrazione di informazioni sulle proprietà fisiche possa migliorare in modo significativo i risultati dell'inversione.

L'algoritmo [24] originariamente sviluppato in linguaggio Fortran, adotta una strategia iterativa in cui i due dataset vengono trattati in modo sequenziale: l'informazione sismica viene utilizzata per derivare il modello di velocità e i relativi vincoli, mentre i dati gravimetrici contribuiscono all'aggiornamento sia del modello di densità, in accordo con la relazione velocità-densità assunta o stimata, sia del modello di velocità iniziale. Questo processo viene ripetuto iterativamente fino al raggiungimento della convergenza verso il modello finale. L'uso di una funzione di densità di probabilità [22] colloca l'algoritmo in una posizione intermedia tra un approccio di inversione puramente congiunta,

in cui i due dataset sono trattati simultaneamente [25], e un'inversione strettamente sequenziale, in cui i dati vengono invertiti separatamente fino al raggiungimento di un misfit accettabile per entrambi i set [15]. In particolare, l'obiettivo pratico del lavoro di tesi è consistito nella riscrittura del codice originale da Fortran a C++. Un elemento fortemente originale di questo lavoro è rappresentato dallo sviluppo ex novo di una libreria in C++ per il calcolo parallelo di operazioni di algebra lineare su matrici distribuite, progettata e implementata durante questo studio. Tale libreria, ispirata alla struttura e alle funzionalità di ScaLAPACK [23] in Fortran, adotta uno schema di distribuzione ciclica a blocchi.

1.2 Inversione geofisica

Nel contesto delle geoscienze, il termine *inversione* indica il processo matematico mediante il quale, a partire da misure effettuate in superficie (ad esempio tempi di arrivo di onde sismiche o anomalie di gravità), si ricava un modello plausibile delle proprietà fisiche del sottosuolo. Si tratta, in pratica, di "risalire dagli effetti alle cause": poiché non è possibile osservare direttamente densità o velocità in profondità, si utilizzano modelli fisici che descrivono come tali proprietà generano le osservazioni (*forward modeling*), e si cerca il modello che riproduce meglio i dati. Questo procedimento è tuttavia instabile e non univoco: molti modelli diversi possono produrre osservazioni simili, e piccoli errori nei dati possono propagarsi in variazioni significative del modello. Per questo motivo ogni inversione richiede l'introduzione di vincoli aggiuntivi, o *regolarizzazioni*, che guidano la soluzione verso configurazioni fisicamente realistiche oppure coerenti con informazioni indipendenti [22].

Un tipico schema di inversione lineare, ampiamente utilizzato in tomografia sismica e gravimetria, è illustrato in Fig. 1.1. L'algoritmo prevede una sequenza iterativa di passi: a partire da un modello iniziale si calcolano le risposte teoriche tramite il *forward model*; queste vengono confrontate con i dati osservati, ottenendo il misfit; sulla base di tale misfit si calcola un aggiornamento del modello secondo una relazione lineare (spesso espressa tramite matrici di sensibilità o derivate del modello); il modello viene quindi modificato e il processo viene ripetuto fino a ottenere una soluzione stabile o fino al soddisfacimento di criteri di arresto predeterminati. La struttura iterativa del metodo evidenzia come l'inversione sia in realtà un problema di ottimizzazione.

Dal punto di vista concettuale, questa formulazione risulta particolarmente utile anche nei metodi di integrazione di dataset differenti, nei quali ciascun tipo di misura (ad esempio sismica e gravimetrica) fornisce vincoli complementari sulle proprietà fisiche del mezzo, e constribuisce quindi a ridurre l'ambiguità del problema inverso.

1.3 Strategia di inversione integrata

La strategia considerata in questo lavoro di tesi è di tipo *multi-step sequential integrated inversion*. Sinteticamente si possono indicare i passi essenziali:

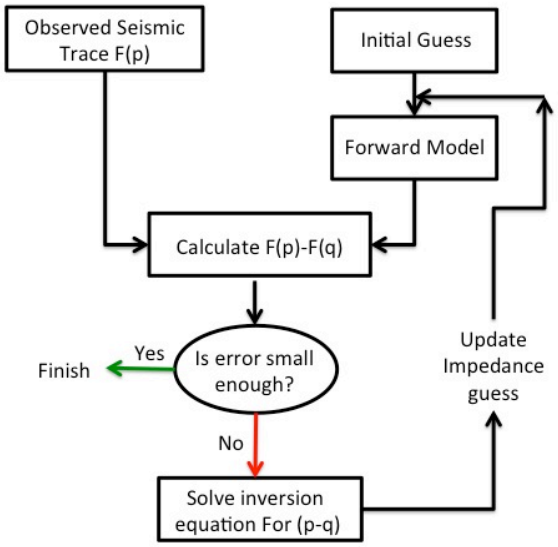

Figura 1.1: Schema classico dell'algoritmo di inversione lineare. L'immagine illustra la sequenza iterativa di forward modeling, confronto con i dati e aggiornamento del modello [4].

1. **Inversione sismica**: si stima un modello di velocità delle onde sismiche utilizzando i dati sismici disponibili. Qualsiasi tecnica di inversione sismica (traveltime tomography, inversione full waveform, tomografia a rifrazione/riflessione ecc.) compatibile con i dati può essere impiegata in questo passo. Il risultato è un modello di velocità tridimensionale che risolve, entro limiti di errore, le osservazioni sismiche locali.

2. **Relazione velocità-densità**: si assume o si stima mediante misure di tipo petrografico o in pozzo una relazione spazio-dipendente tra velocità e densità (es. modelli lineari o parametrici locali) che permette di tradurre il modello di velocità in un modello iniziale di densità a priori per l'analisi gravimetrica.

3. **Integrazione gravimetrica**: a partire dal modello di velocità ottenuto dall'inversione sismica, viene derivato un modello di densità iniziale mediante una relazione velocità-densità assunta o stimata. Su questo modello di densità viene quindi calcolata la risposta gravimetrica prevista. Successivamente, si costruisce una *matrice di covarianza* che tiene conto sia della propagazione delle incertezze associate al modello sismico (tramite la sua matrice di covarianza a posteriori) sia degli errori legati alla stima della relazione velocità-densità utilizzata per ricavare il modello di densità iniziale. A partire da queste informazioni, il problema gravimetrico viene

formulato in termini probabilistici e viene determinata la correzione al modello di densità che massimizza la funzione di densità di probabilità a posteriori. In questo modo si ottiene un modello di densità ottimale, coerente sia con le osservazioni gravimetriche sia con i vincoli a priori derivanti dall'informazione sismica.

4. **Iterazione**: all'aggiornamento del modello di densità fa seguito l'aggiornamento del modello di velocità, ottenuto attraverso la relazione densità-velocità, che può essere a sua volta aggiornata nel corso del processo di ottimizzazione. Il modello di velocità così stimato viene quindi utilizzato come vincolo a priori nella successiva iterazione dell'inversione sismica. L'intero procedimento viene ripetuto iterativamente fino al raggiungimento della convergenza per entrambi i dataset, sismico e gravimetrico.

Questo approccio sequenziale mantiene separate le stime per i due dataset, ma le integra iterativamente per sfruttare la complementarità informativa: i dati sismici offrono risoluzione strutturale dettagliata in determinate regioni, mentre i dati di gravità forniscono vincoli integrali sulla distribuzione delle masse.

1.4 Problema diretto

Il problema diretto della gravità consiste nel calcolare la componente verticale dell'accelerazione gravitazionale misurata in N stazioni nota la distribuzione di densità in un volume discretizzato in M elementi (cubi, prismi o poliedri). In forma discretizzata lineare:

$$\mathbf{g} \;=\; \mathbf{G}\,\Delta\boldsymbol{\rho} \;+\; \boldsymbol{\varepsilon} \tag{1.1}$$

dove

- $\mathbf{g} \in \mathbb{R}^N$ è il vettore che rappresenta il campo di gravità (componente verticale della gravità), osservato o calcolato,
- $\Delta\boldsymbol{\rho} \in \mathbb{R}^M$ è il vettore delle anomalie di densità (incognite),
- $\mathbf{G} \in \mathbb{R}^{N\times M}$ è la matrice delle derivate parziali: ogni elemento $G_{n,m} = \partial g_n/\partial \rho_m$ misura l'effetto della variazione di densità nel volume m sulla misura della gravità sul punto n,
- $\boldsymbol{\varepsilon}$ rappresenta l'errore associato alle macchine di calcolo.

Per il calcolo di $\mathbf{G}$ e della previsione $\mathbf{g}$ si usano formule analitiche o semi-analitiche per corpi poliedrici o prismatici; esistono metodi efficienti e numericamente stabili (per corpi poliedrici con densità variabile, vedi Pohánka [20]).

1.5 Problema inverso

L'inversione lineare regolarizzata può essere formulata come un problema di minimi quadrati con termine di regolarizzazione (fare riferimento a Tondi et al. [24] per maggiori dettagli). Applicando il principio di massima probabilità alla funzione densità di probabilità si ottiene la soluzione del problema:

$$\Delta\boldsymbol{\rho}^{(i)} \;=\; (\mathbf{G}^T\mathbf{C}_{gg}^{-1}\mathbf{G} + \mathbf{C}_{mm}^{-1})^{-1}(\mathbf{G}^T\mathbf{C}_{gg}^{-1}\Delta\mathbf{g} + \alpha\mathbf{C}_{mm}^{-1}\Delta\mathbf{v}) \tag{1.2}$$

dove

- $\Delta\mathbf{g}$ è il vettore dei residui di gravità;
- $\mathbf{G}$ è la matrice delle derivate parziali che lega variazioni di densità alle osservazioni;
- $\mathbf{C}_{gg}$ e $\mathbf{C}_{mm}$ sono rispettivamente le matrici di covarianza degli errori sui dati e del modello a priori;
- $\Delta\mathbf{v}$ è il vettore delle velocità;

1.6 Problematiche

La principale limitazione computazionale del lavoro originale è rappresentata dalla costruzione e dalla gestione della matrice delle derivate parziali $\mathbf{G}$. Questa matrice è in generale *densa*: ogni elemento della discretizzazione del volume influisce, seppur con intensità decrescente con la distanza, su ciascuna stazione di misura. Di conseguenza la costruzione esplicita di $\mathbf{G}$ e la risoluzione del sistema 1.2 diventano rapidamente onerose sia in termini di tempo di calcolo ma soprattutto di memoria quando il numero di osservazioni N e il numero di celle di discretizzazione M aumentano.

La memorizzazione completa di $\mathbf{G}$ può richiedere risorse superiori a quelle disponibili su una singola macchina; analogamente, la soluzione diretta delle equazioni necessarie alla risoluzione del problema può risultare computazionalmente troppo onerosa. Per la gestione di grandi volumi di dati, corrispondenti a volumi di terreno di grandi dimensioni, e per rendere questi calcoli praticabili è quindi necessario distribuire i dati (matrici e vettori) e il carico computazionale su più processori di un cluster.

Il codice sviluppato da Tondi et al. [24] affronta precisamente questo problema mediante calcolo distribuito: le matrici vengono suddivise tra i nodi del cluster invece di essere tutte conservate nella memoria di un unico calcolatore. Questo è reso possibile dall'uso della libreria ScaLAPACK, che consente di eseguire operazioni di algebra lineare in parallelo su matrici distribuite secondo lo schema della distribuzione ciclica a blocchi. Il funzionamento e i dettagli implementativi di ScaLAPACK verranno approfonditi in un capitolo successivo.

Il software originario è tuttavia scritto in Fortran: il presente lavoro di tesi consiste nel porting dell'applicazione e delle routine necessarie per la gestione distribuita delle matrici dal Fortran al C++, con particolare attenzione agli aspetti legati alla formulazione

e alla risoluzione del problema inverso. L'obiettivo è quello di di ottenere un codice più moderno, manutenibile e integrabile nell'ecosistema C++ contemporaneo. Questo intervento comprende la riscrittura delle procedure di inversione sismico-gravimetrica, nonché l'implementazione di una libreria di funzioni in C++ in grado di riprodurre le funzionalità della libreria ScaLAPACK nell'ambito delle operazioni di algebra lineare su matrici distribuite.

Nei capitoli successivi verranno descritti nel dettaglio gli aspetti implementativi, gli algoritmi algebrici adottati e i risultati delle misure di prestazione sulle funzionalità sviluppate.

Capitolo 2

Distribuzione delle matrici per il calcolo parallelo

2.1 Matrici su memoria distribuita

La crescente dimensione dei problemi numerici in scienze e ingegneria rende spesso impossibile o inefficiente l'uso di strutture dati monolitiche residenti su un unico nodo: memorie locali limitate, limiti di I/O e costi computazionali spingono verso soluzioni parallele. Distribuire gli elementi di una matrice su più processi permette di:

- aumentare la memoria totale disponibile (ogni processo conserva solo una porzione della matrice globale);
- parallelizzare le operazioni aritmetiche (ogni processo esegue calcoli sulla propria porzione);
- ridurre i tempi di esecuzione sfruttando l'hardware del cluster in modo scalabile;

Esistono diverse tecniche per la distribuzione delle matrici su più processi, ognuna con i propri vantaggi e svantaggi: la *distribuzione a blocchi*, la *distribuzione ciclica* e la *distribuzione ciclica a blocchi*.

2.1.1 Distribuzione a blocchi

Le prime strategie di distribuzione sono basate su suddivisioni lungo righe o colonne o su blocchi contigui. Dato una matrice globale $A \in \mathbb{R}^{m \times n}$, si possono definire:

- **Block-row**: ogni processo riceve un insieme contiguo di righe (Fig. 2.1). Se p è il numero di processi e $m_i = \frac{m}{p}$ il numero di righe assegnate al processo i, allora la porzione locale é $A_{loc}^{(i)} \in \mathbb{R}^{m_i \times n}$.
- **Block-column**: analogamente per le colonne (Fig. 2.2), con porzioni $A_{loc}^{(i)} \in \mathbb{R}^{m \times n_i}$.

- **Block**: la matrice viene suddivisa in blocchi rettangolari contigui secondo una griglia regolare; ogni processo memorizza uno o più blocchi contigui (Fig. 2.3).

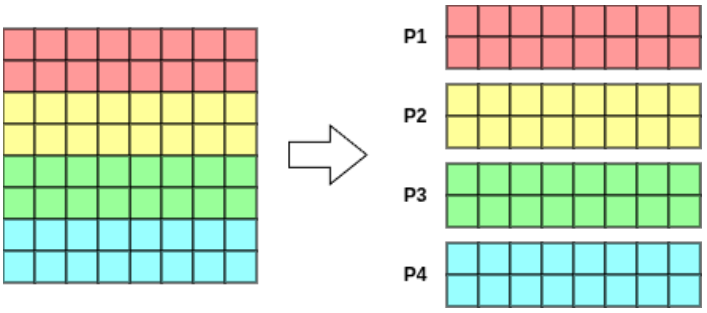

Figura 2.1: Distribuzione block-row di una matrice tra 4 processi.

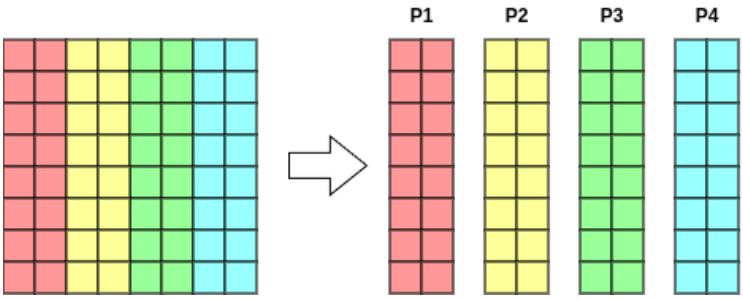

Figura 2.2: Distribuzione column-row di una matrice tra 4 processi.

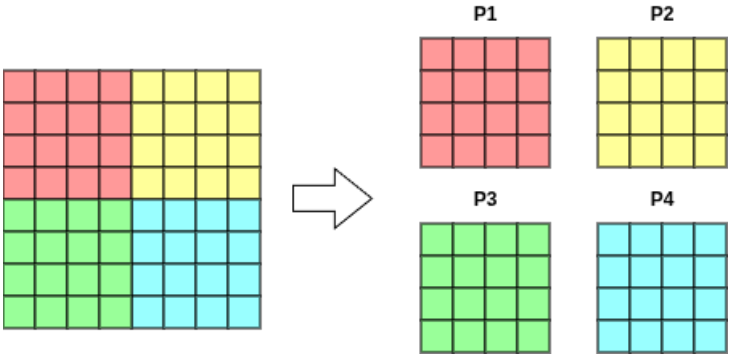

Figura 2.3: Distribuzione a blocchi di una matrice tra 4 processi.

Queste strategie sono semplici da implementare e consentono di sfruttare l'adiacenza in memoria dei dati, ottimizzando le prestazioni grazie all'utilizzo della cache e la vettorizzazione. Tuttavia, tali vantaggi si limitano alle operazioni su una singola matrice. Quando due o più matrici hanno distribuzioni differenti, i processi possono dover accedere a porzioni di dati non allineate, con conseguente squilibrio di carico e costosi scambi di dati tra nodi.

2.1.2 Distribuzione ciclica

La distribuzione ciclica assegna righe o colonne (o blocchi di dimensione 1) ai processi in modo periodico: il primo elemento al processo 0, il secondo al processo 1, e così via, tornando al processo 0 dopo l'ultimo (Fig. 2.4).

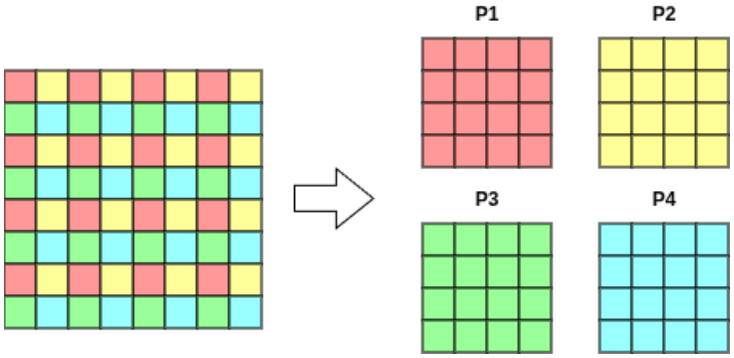

Figura 2.4: Distribuzione ciclica di una matrice tra 4 processi.

Questo approccio migliora il bilanciamento del carico sia nelle operazioni su singole matrici, sia nelle operazioni fra più matrici, poiché la distribuzione dei dati risulta più uniforme. In questo modo, elementi corrispondenti di matrici diverse tendono a essere mappati sugli stessi processi, semplificando l'esecuzione di operazioni locali. Tuttavia, la mancanza di adiacenza in memoria penalizza le prestazioni, e la frequente necessità di comunicazioni tra processi può introdurre overhead significativi.

2.1.3 Distribuzione block-cyclic

La distribuzione *block-cyclic* generalizza la precedente: si definisce una dimensione di blocco locale (b_r, b_c) (righe per blocco, colonne per blocco) e si mappa la griglia di blocchi globali sui processi seguendo uno schema ciclico bidimensionale su una griglia di processori $P_r \times P_c$. In altre parole, la matrice globale viene suddivisa in celle di dimensione $b_r \times b_c$ e queste celle vengono assegnate ai processi in ordine ciclico secondo la disposizione dei processori (Fig. 2.5).

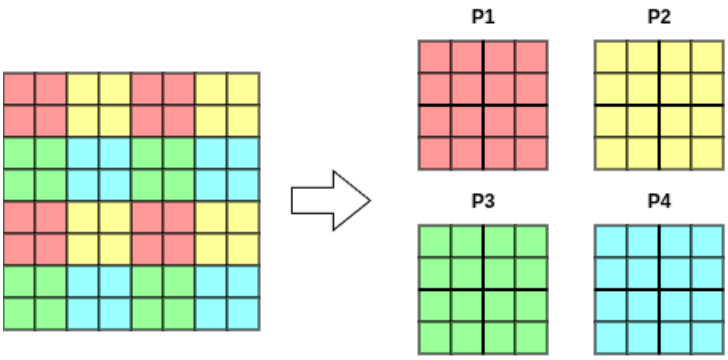

Figura 2.5: Immagine che mostra la distribuzione ciclica a blocchi di una matrice tra 4 processi.

Questa soluzione combina i vantaggi dei due approcci precedenti:

- bilanciamento del carico computazionale più uniforme;
- riduzione del volume complessivo di comunicazioni;
- possibilità di sfruttare ottimizzazioni locali basate su blocchi contigui in memoria.

Per queste ragioni, la distribuzione block-cyclic è quella adottata nelle principali librerie di algebra lineare parallela, come ScaLAPACK.

2.2 ScaLAPACK

ScaLAPACK (*Scalable Linear Algebra PACKage*) [23] è una libreria standard per il calcolo numerico parallelo di algebra lineare su architetture a memoria distribuita. Basata sui concetti di BLAS [6] e PBLAS [3] per le operazioni locali, estende LAPACK [2] per ambienti distribuiti, fornendo routine per fattorizzazioni (LU, QR, Cholesky), risoluzione di sistemi lineari, calcolo di autovalori e autovettori, e altre operazioni di algebra lineare. Le routine sono progettate per essere eseguite su griglie di processi organizzate attraverso BLACS (*Basic Linear Algebra Communication Subprograms*) [5] o MPI.

ScaLAPACK è ampiamente diffuso nella comunità scientifica per la sua stabilità numerica, l'efficienza degli algoritmi implementati e la portabilità su differenti architetture HPC. Esso fornisce un'infrastruttura solida per lo sviluppo di applicazioni scientifiche di larga scala.

Il disegno concettuale di ScaLAPACK si basa su pochi elementi chiave:

1. **Griglia di processori**: i processi vengono organizzati in una griglia bidimensionale $P_r \times P_c$. Questa struttura facilita la mappatura delle matrici distribuite e delle comunicazioni a livello di riga/colonna di blocchi.

2. **Distribuzione block-cyclic 2D**: usando questa distribuzione, si hanno i vantaggi descritti sopra e si possono riutilizzare degli algoritmi di LAPACK per eseguire i calcoli localmente.

3. **Array descriptor**: ogni matrice distribuita possiede un descrittore che specifica le dimensioni globali m, n, le dimensioni di blocco b_r, b_c, e la posizione/offset della porzione locale memorizzata. Le routine ScaLAPACK usano questo descrittore per calcolare gli indici locali e orchestrare comunicazioni.

4. **PBLAS e BLACS**: ScaLAPACK si appoggia su Primitive BLAS parallele (PBLAS) per le operazioni di base a livello distribuito, e su BLACS per la gestione delle comunicazioni e dei contesti di processo.

L'utilizzo di ScaLAPACK consente di risolvere efficientemente problemi di algebra lineare di grandi dimensioni, motivo per cui è divenuto un riferimento nel calcolo scientifico. Tuttavia, la libreria presenta alcune complessità d'uso: richiede una configurazione preliminare (inizializzazione di BLACS e MPI) e, essendo scritta principalmente in Fortran, risulta meno accessibile a utenti non esperti di programmazione. Sebbene siano disponibili interfacce C, l'interazione resta complessa per chi desidera utilizzare la libreria come strumento di ricerca numerica piuttosto che come framework di sviluppo.

Queste considerazioni, unite all'esigenza di tradurre il codice d'inversione sismico-gravimetrica dell'INGV in C++, hanno motivato la creazione di una nuova libreria, denominata *SLAC (Scalable Linear Algebra for C++)*, con l'obiettivo di offrire prestazioni e affidabilità paragonabili a ScaLAPACK, ma con una progettazione più moderna, intuitiva e integrata nell'ecosistema C++.

Capitolo 3

Scalable Linear Algebra for C++

La libreria *SLAC* [21] (acronimo di *Scalable Linear Algebra for C++*) è un framework progettato per lo sviluppo e la risoluzione di problemi di algebra lineare con matrici dense in contesti distribuiti. Scritta in C++20, essa propone, attraverso un linguaggio moderno, una valida alternativa alle librerie tradizionali per l'algebra lineare parallela.

L'intera libreria è ottimizzata per il calcolo parallelo, non soltanto su singoli nodi multi-core, ma soprattutto su cluster con memoria distribuita. Analogamente a librerie come ScaLAPACK, SLAC adotta una distribuzione delle matrici secondo lo schema di distribuzione ciclica a blocchi, che è uno standard per queste tipologie di problemi.

Inoltre, SLAC si distingue per una maggiore semplicità d'uso e configurazione: è una libreria *header-only*, facile da includere nei progetti, e al contempo nasconde all'utente l'intera logica di distribuzione, calcolo e comunicazione. Questo la rende più adatta anche ad utenti non esperti di programmazione parallela, pur mantenendo elevata efficienza e affidabilità.

3.1 Tecnologie

Le tecnologie impiegate per lo sviluppo di questa libreria sono ormai consolidate tanto in ambito scientifico quanto informatico, per la risoluzione di problemi di algebra lineare e per le comunicazioni tra processi in memoria non condivisa. In particolare:
Ogni processo esegue le proprie computazioni sulle matrici locali mediante la libreria Eigen, e si coordina e scambia dati con gli altri processi tramite MPI.

3.1.1 Eigen

La libreria Eigen [12] è una libreria C++ a template per l'algebra lineare: vettori, matrici dense e sparse, risolutori numerici e algoritmi correlati. Tra le motivazioni della sua scelta occorrono evidenziare i seguenti punti:

- è *header-only*: non richiede una fase di linking o installazione complessa, basta includere gli header nel progetto.

- fornisce una interfaccia un semplice e familiare, che consente rapido sviluppo ed è utilizzata in molti ambiti industriali e di ricerca.
- fornisce prestazioni elevate grazie all'uso di tecniche come *expression templates*, ottimizzazione del codice, supporto per vettorizzazione, e supporto per matrici dinamiche.
- è ampiamente diffusa e supportata da una comunità attiva, ciò garantisce robustezza e affidabilità.

3.1.2 MPI (Message Passing Interface)

La *Message Passing Interface* (MPI) [9] è uno standard di libreria per la comunicazione fra processi in architetture a memoria distribuita. Nel contesto di SLAC, MPI consente ai processi di coordinarsi, scambiare porzioni di matrici distribuite e cooperare nel calcolo parallelo. Le ragioni della sua scelta includono:

- è lo standard de facto nelle applicazioni HPC su cluster con memoria distribuita, garantisce portabilità e scalabilità.
- consente la comunicazione punto-a-punto, le operazioni collettive, la gestione di gruppi di processi e topologie logiche di comunicazione.
- dispone di implementazioni mature e performanti (come Open MPI [18]), con supporto per processi distribuiti su nodi eterogenei.

3.2 Struttura

3.2.1 Glossario

Prima di mostrare la struttura interna della libreria, è opportuno definire un breve glossario dei termini che verranno utilizzati più avanti, al fine di evitare ambiguità. Si useranno frequentemente i termini *nodo*, *matrice locale*, *matrice distribuita* e *matrice reale*. Con questi termini si intende:

- **Nodo**: rappresenta un nodo logico di un cluster; non si intende necessariamente un nodo fisico, bensì un processo identificato da una posizione nella griglia dei processi. Si preferisce la parola "nodo" per enfatizzare che il codice è completamente trasparente alla collocazione fisica dei processi: più nodi potrebbero risiedere nella stessa macchina fisica e l'applicazione li tratterebbe come processi indipendenti, senza conoscere la loro localizzazione fisica.
- **Matrice locale**: è una matrice residente in un singolo nodo. Se un nodo possiede una matrice locale, allora detiene in memoria tutti gli elementi di quella matrice. I calcoli su tale matrice non richiedono comunicazioni inter-nodi e perciò sono definiti locali.

- **Matrice distribuita**: è una matrice suddivisa fra più nodi secondo la distribuzione ciclica a blocchi. Ciò implica che un nodo che gestisce una matrice distribuita non necessariamente detiene tutti gli elementi della matrice globale; se ha bisogno di accedere ad elementi che risiedono su altri nodi, può essere necessario richiederli via comunicazione.

- **Matrice reale**: indica la matrice dal punto di vista globale e logico. Indipendentemente dal modo in cui è memorizzata (localmente o in forma distribuita) questo termine viene usato principalmente per descrivere il comportamento degli algoritmi distribuiti.

3.2.2 Funzionalità

La libreria SLAC opera mediante funzioni collettive, ossia chiamate che devono essere invocate simultaneamente da tutti i processi coinvolti per ottenere un risultato comune e cooperativo. Tutti i nodi eseguono lo stesso programma e, in base alle matrici in gioco e alla posizione logica del nodo, calcolano il risultato.

I nodi di SLAC corrispondono ai processi MPI, decisi all'avvio del programma: in base al numero di processi disponibili viene creata una griglia logica che associa a ogni nodo una posizione. Tale posizione è utilizzata principalmente per coordinare i nodi ed eseguire algoritmi basati sulla distribuzione ciclica a blocchi.

Nella libreria sono previste due classi principali per l'utente:

- `LocalMatrix`: rappresenta una matrice locale. Ogni nodo che crea una `LocalMatrix` conserva in memoria tutti gli elementi della matrice. Internamente utilizza le strutture della libreria Eigen per eseguire in modo efficiente i calcoli locali.

- `DistributedMatrix`: rappresenta una matrice distribuita. Ogni nodo che crea una `DistributedMatrix` conserva in memoria solo gli elementi mappati su quel nodo secondo la distribuzione ciclica a blocchi. In sostanza, è un wrapper su `LocalMatrix`: contiene una matrice locale e include le informazioni riguardanti la porzione della matrice globale a cui il nodo è assegnato. L'assegnazione degli elementi ai rispettivi nodi è gestita da una classe ausiliaria, `Mapper`, il cui compito è stabilire una corrispondenza biunivoca tra ogni elemento della matrice reale e il nodo corretto, determinandone anche la posizione all'interno della matrice locale.

LocalMatrix e DistributedMatrix implementano una medesima interfaccia astratta, `Matrix`, che definisce le informazioni strutturali di base (dimensioni) e i metodi di accesso in lettura e scrittura ai singoli elementi. La logica di accesso è delegata all'interfaccia `ElementLogic`, che fornisce un meccanismo unificato per la gestione degli elementi. A seconda dell'implementazione concreta, questa può operare su memoria locale (`LocalElementLogic`) oppure su memoria distribuita (`DistributedElementLogic`).

La classe `LocalMatrix` non eredita direttamente da `Matrix`, ma da una classe astratta intermedia, `BaseMatrix`. Questa scelta progettuale permette di incapsulare qualunque tipo di matrice di Eigen, sfruttandone le ottimizzazioni. Un'altra estensione utilizzata

di `BaseMatrix` è `LocalSubmatrix`, che rappresenta una vista su una porzione di una `LocalMatrix`. Questa classe permette di manipolare sottomatrici senza effettuare copie dei dati, consentendo aggiornamenti ottimizzati direttamente sulla matrice originale grazie alle operazioni su blocchi offerte da Eigen.

Nella Fig. 3.1 è mostrato lo schema delle classi definite nella libreria.

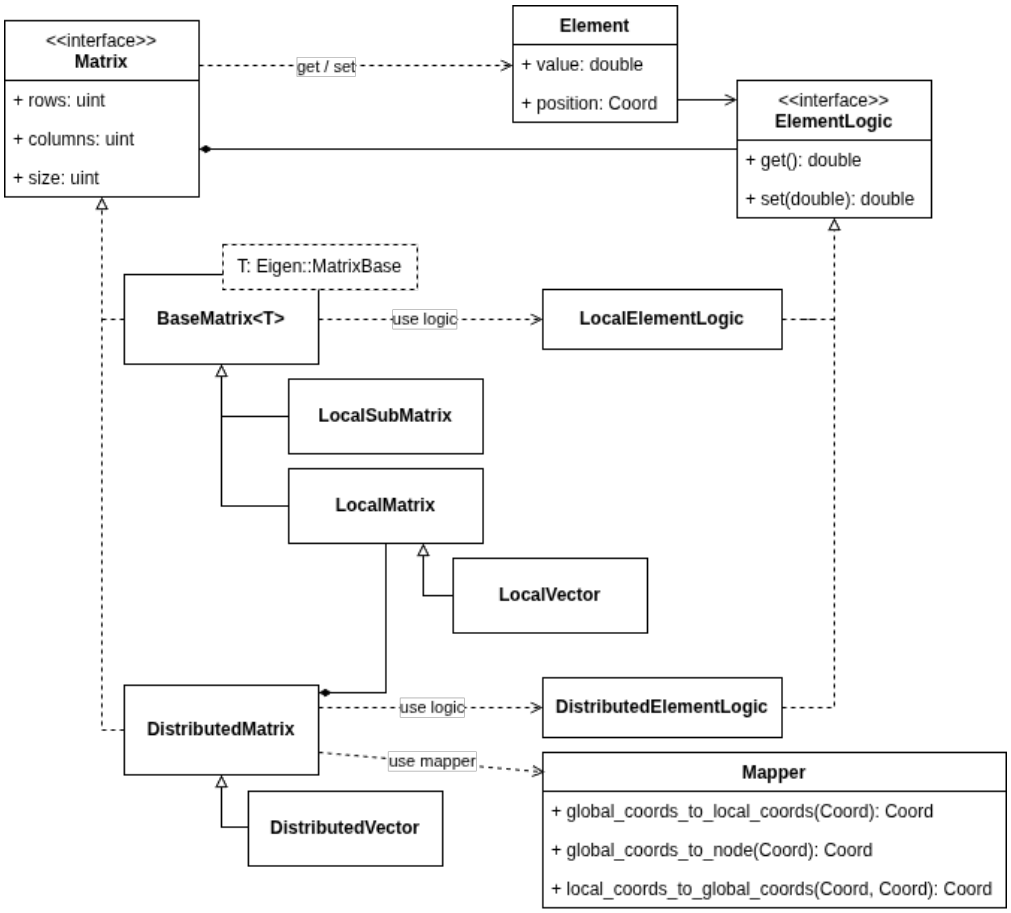

Figura 3.1: Schema UML delle classi definite nella libreria SLAC.

È importante osservare che, se su un nodo si definiscono due matrici con le stesse dimensioni - una `LocalMatrix` e una `DistributedMatrix` - la `DistributedMatrix` occuperà sempre meno spazio in memoria rispetto alla `LocalMatrix`. La matrice locale alloca tutti gli elementi della matrice globale, mentre quella distribuita alloca soltanto gli elementi mappati sul nodo, che sono in numero inferiore rispetto al totale. Di conseguenza, la `DistributedMatrix` risulta più efficiente in termini di occupazione di memoria, a fronte di algoritmi più complessi e della necessità di comunicazione e coordinamento tra nodi.

Nel sistema sono previste anche le classi `LocalVector` e `DistributedVector`, estensioni rispettivamente di `LocalMatrix` e `DistributedMatrix` che rappresentano rispettivamente vettori locali e distribuiti (i quali possono essere visti come matrici con una delle due dimensioni unitaria) utilizzate per ottimizzare alcuni algoritmi e imporre vincoli sulle dimensioni degli input.

Uno dei punti di forza della libreria è che le stesse funzioni e operazioni sono identiche sia per matrici locali sia per matrici distribuite. Grazie all'overload delle funzioni (che selezionano l'algoritmo appropriato in base al tipo), le classi `LocalMatrix` e `DistributedMatrix` sono completamente intercambiabili e integrabili dal punto di vista dell'utente, mentre la logica sottostante rimane nascosta. Anche l'accesso agli elementi delle matrici è uniformato tramite la classe `Element`, che consente di leggere e scrivere un singolo elemento sia su matrici locali che distribuite, in base alla `ElementLogic` utilizzata.

3.3 Test e prestazioni

Tutte le funzionalità implementate in SLAC sono state verificati tramite un'ampia suite di test e risultano correttamente funzionanti. Lo sviluppo è avvenuto seguendo il paradigma del *test-driven development*: per ciascuna funzione della libreria sono stati scritti uno o più test che verificano tutti gli scenari d'uso previsti.

Per i test è stato adottato il framework Catch2 [16], un popolare framework di testing in C++ che fornisce una sintassi semplice, leggibile e basata su macro per definire casi di test. Catch2 supporta anche il micro-benchmarking e può essere facilmente integrato nello sviluppo TDD grazie alla sua natura header-only.

Oltre ai test di correttezza, sono stati creati benchmark dedicati per misurare le prestazioni di SLAC. In questa tesi si presentano le misurazioni di tempo di esecuzione, la scalabilità (in modalità strong scaling) e l'efficienza delle varie operazioni al variare delle dimensioni delle matrici, delle dimensioni dei blocchi e del numero di processi.

Le prove di performance sono state condotte su un server dotato di processore Intel Xeon E5-2603 (12 core a 1,70 GHz), 64 GB di RAM e 3 GPU NVIDIA GTX 1070.

Capitolo 4

Algoritmi per matrici distribuite

Le matrici locali, essendo interamente memorizzate nella memoria di un singolo nodo, possono sfruttare senza limitazioni gli algoritmi ottimizzati forniti dalla libreria Eigen. Le matrici distribuite, invece, non possono delegare direttamente a Eigen le operazioni algebriche, poiché nessun nodo possiede tutte le informazioni necessarie. Per eseguire tali calcoli è quindi necessario ricorrere ad algoritmi specifici, in grado di orchestrare lo scambio dei dati tra i nodi affinché ciascuno possa disporre localmente degli elementi richiesti dalle routine di Eigen.

Gli algoritmi distribuiti sviluppati in questo lavoro sono progettati per operare su matrici distribuite secondo lo schema ciclico a blocchi (Fig. 2.5). Tale modalità di distribuzione presenta alcune proprietà strutturali che possono essere sfruttate per ottimizzare le operazioni:

- i dati sono suddivisi in blocchi quadrati di lato B, il che implica che gli elementi con indici appartenenti all'intervallo $[iB \dots (i+1)B]$, per ogni $i \in \mathbb{N}$, risultano adiacenti anche nella matrice reale;
- i nodi del cluster sono organizzati in una griglia di processi di dimensione (G_x, G_y), e ciascun nodo è identificato dalla coppia (C_x, C_y);
- tutti i nodi appartenenti alla stessa colonna della griglia (stesso valore di C_x) memorizzano localmente lo stesso numero di colonne;
- tutti i nodi appartenenti alla stessa riga della gridlia (stesso valore di C_y) memorizzano localmente lo stesso numero di righe;
- il nodo $(0, 0)$ contiene sempre almeno un elemento, poiché è il nodo che possiede la posizione reale $(0, 0)$.

Gli elementi locali della `DistributedMatrix` vengono assegnati ai nodi corretti seguendo uno schema ciclico a blocchi, tramite una formula che, data la posizione reale (P_x, P_y) di un elemento, determina sia il nodo a cui esso appartiene (C_x, C_y), sia la sua

posizione locale nel nodo $\left(P_x', P_y'\right)$:

$$\begin{aligned} P_x^{'} &\leftarrow \left\lfloor \frac{P_x}{G_x B} \right\rfloor B + (P_x \bmod G_x B) \bmod B \\ P_y^{'} &\leftarrow \left\lfloor \frac{P_y}{G_y B} \right\rfloor B + (P_y \bmod G_y B) \bmod B \end{aligned} \tag{4.1}$$

$$\begin{aligned} C_x &\leftarrow \left\lfloor \frac{P_x}{B} \right\rfloor \bmod G_x \\ C_y &\leftarrow \left\lfloor \frac{P_y}{B} \right\rfloor \bmod G_y \end{aligned} \tag{4.2}$$

La formula inversa permette invece di determinare la posizione reale di un elemento a partire dalla sua posizione locale e dal nodo che lo contiene:

$$\begin{aligned} P_x &\leftarrow \left(\left\lfloor \frac{P_x^{'}}{B} \right\rfloor G_x + C_x \right) + P_x^{'} \bmod B \\ P_y &\leftarrow \left(\left\lfloor \frac{P_y^{'}}{B} \right\rfloor G_y + C_y \right) + P_y^{'} \bmod B \end{aligned} \tag{4.3}$$

È importante osservare che questo mapping è universale: dato un qualunque elemento di una matrice, indipendentemente dalle dimensioni della matrice stessa, è sempre possibile determinare in anticipo su quale nodo verrà collocato e in quale posizione della matrice locale di quel nodo sarà memorizzato. Ciò implica che due elementi che occupano la stessa posizione in matrici diverse verranno comunque assegnati allo stesso nodo e alla medesima posizione locale. Questo rende gli algoritmi distribuiti significativamente più semplici ed efficienti, riducendo il costo delle comunicazioni e facilitando l'esecuzione dei calcoli.

Grazie a tali relazioni, ogni nodo può determinare in modo univoco dove reperire e verso chi inviare i dati necessari per le operazioni distribuite, consentendo una comunicazione efficiente.

4.1 Operazioni embarrassingly parallel

4.1.1 Design

Alcune operazioni possono essere classificate come *embarrassingly parallel*, ovvero composte da insiemi di task completamente indipendenti tra loro. In questi casi ogni operazione locale può essere eseguita senza richiedere alcuna forma di comunicazione o sincronizzazione con gli altri nodi del cluster, portando a un parallelismo perfetto.

Rientrano in questa categoria tutte le operazioni elemento-per-elemento: ciascun nodo lavora esclusivamente sugli elementi della porzione locale della matrice, già presente in memoria, e può quindi completare il proprio calcolo in autonomia. Esempi tipici sono

la somma tra matrici (eseguita come somma indipendente dei rispettivi elementi locali) e la moltiplicazione elemento-per-elemento.

A queste si aggiunge una funzione generica implementata in SLAC, denominata `apply`, che consente di applicare una funzione unaria a ogni elemento della matrice. Il risultato della funzione viene quindi memorizzato direttamente in corrispondenza della posizione dell'elemento originale.

4.1.2 Implementazione

Poiché le operazioni embarrassingly parallel possono essere svolte interamente in locale, la loro implementazione è delegata a Eigen, che offre primitive ottimizzate e ad alte prestazioni per il calcolo matriciale. Non è richiesta alcuna comunicazione tra i nodi: anche nel caso della somma tra due matrici distribuite, il fatto che esse condividano la stessa dimensione globale garantisce che, su ogni nodo, le corrispondenti matrici locali abbiano dimensioni identiche e rappresentino la stessa porzione della matrice globale.

Questo allineamento naturale permette di eseguire l'operazione in perfetto parallelismo, con ogni nodo che effettua il proprio calcolo in modo autonomo. Le funzioni che seguono questo schema sono `Slac::add`, `Slac::cwise_mul`, `Slac::cwise_div` e `Slac::apply`.

4.1.3 Analisi delle prestazioni

Per la valutazione delle prestazioni è stata analizzata esclusivamente la funzione `Slac::add`. Le tabelle 4.1 e 4.2 riportano i tempi di esecuzione per diversi numeri di nodi e dimensioni della matrice, utilizzando rispettivamente blocchi di lato 32 e 64. Sono inoltre presentati i grafici relativi ai tempi di esecuzione (Fig. 4.1), nonché lo speedup e l'efficienza per matrici di dimensione fissata 2048×2048 (Fig. 4.2).

Tabella 4.1: Tempi di esecuzione della funzione `Slac::add` ($blocksize = 32$)

Dimensione matrice (px)	Tempo di esecuzione (s)			
	1 nodo	4 nodi	8 nodi	12 nodi
256×256	0.00008	0.0001	0.00003	0.00002
512×512	0.0003	0.0002	0.0001	0.00010
1024×1024	0.0015	0.0003	0.0002	0.0003
1536×1536	0.0051	0.0010	0.0006	0.0005
2048×2048	0.0091	0.0026	0.0015	0.0017
2560×2560	0.0142	0.0034	0.0023	0.0021

Essendo un'operazione completamente parallela e priva di comunicazioni, ci si attende uno speedup quasi lineare. Dai risultati sperimentali si osserva effettivamente tale comportamento, con una leggera perdita di efficienza all'aumentare del numero di nodi. Questo effetto è spiegabile con la distribuzione a blocchi: a differenza della distribuzione ciclica, il carico computazionale non è ripartito in modo perfettamente uniforme, e

Tabella 4.2: Tempi di esecuzione della funzione `Slac::add` ($blocksize = 64$)

Dimensione matrice (px)	Tempo di esecuzione (s)			
	1 nodo	4 nodi	8 nodi	12 nodi
256×256	0.00008	0.0001	0.00002	0.00002
512×512	0.0003	0.0002	0.0001	0.0001
1024×1024	0.0015	0.0003	0.0002	0.0001
1536×1536	0.0051	0.0011	0.0006	0.0006
2048×2048	0.0091	0.0024	0.0015	0.0011
2560×2560	0.0142	0.0035	0.0022	0.0018

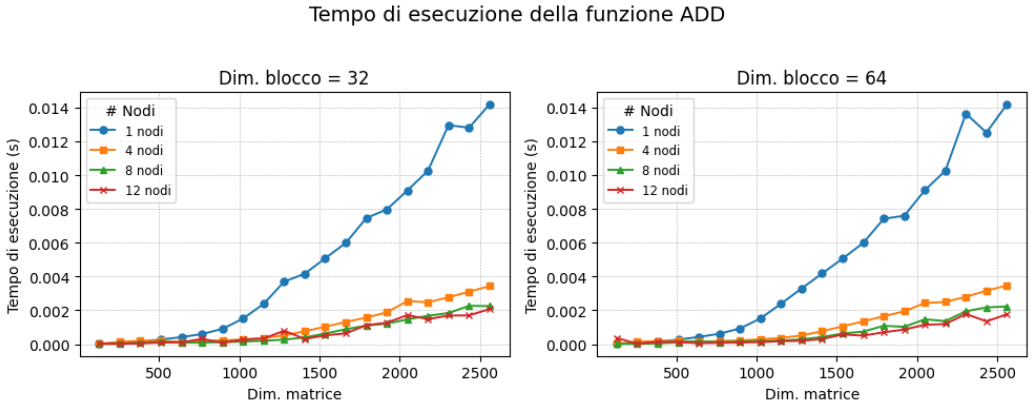

Figura 4.1: Tempi di esecuzione della funzione `Slac::add` misurati su diverse dimensioni dei blocchi.

alcuni nodi possono ricevere un numero di blocchi maggiore rispetto ad altri, riducendo l'efficienza complessiva.

4.2 Trasposizione

4.2.1 Design

La trasposizione di una matrice consiste nel rimappare ogni elemento dalla sua posizione originale alla posizione speculare rispetto alla diagonale principale. Data una matrice A, la sua matrice trasposta A^T si costruisce tramite la relazione:

$$A^T_{j,i} = A_{i,j}. \tag{4.4}$$

Ciò implica che gli elementi in posizione (i, j) della matrice originale vengono spostati in posizione (j, i) della matrice trasposta.

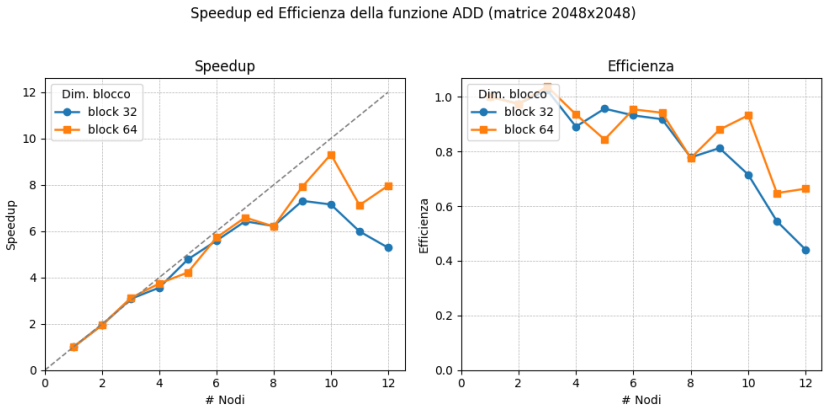

Figura 4.2: Speedup ed efficienza della funzione `Slac::add` con dimensione della matrice fissa 2048×2048.

In un contesto distribuito, tuttavia, questa operazione non è banale: le coordinate (i, j) e (j, i) si riferiscono alla *posizione globale* all'interno della matrice distribuita. Di conseguenza, l'elemento destinato alla nuova posizione può non appartenere al nodo che detiene la posizione originale. Ogni nodo deve quindi inviare gli elementi ai nodi che ospiteranno le rispettive posizioni trasposte, e ricevere gli elementi che completano la propria porzione locale della matrice (Fig. 4.3).

Questa fase coinvolge tutti i nodi e richiede numerose comunicazioni punto-punto, rendendo l'operazione di trasposizione una delle più complesse da gestire in memoria distribuita.

4.2.2 Implementazione

L'utilizzo della distribuzione ciclica a blocchi riduce in modo significativo il costo delle comunicazioni necessarie per la trasposizione. Poiché gli elementi della matrice sono raggruppati in blocchi, è possibile manipolare ciascun blocco localmente prima e dopo il trasferimento al nodo corretto. Questo approccio permette di inviare dati più strutturati e di dimensione maggiore, invece di effettuare comunicazioni elemento-per-elemento. L'algoritmo usato è descritto dal seguente pseudocodice:

1: **function** BLOCKED_TRANSPOSE(A)
2: **for each** block $(i, j) \in A$ **do**
3: $t \leftarrow$ node that will own block (j, i) of A^T
4: **if** this node owns block (i, j) of A **then**
5: send block (i, j) of A to node t
6: **end if**
7: **end for**
8: allocate A^T
9: **for each** block $(j, i) \in A^T$ **do**

```
10:         t ← node that owns block (i, j) of A
11:         if this node owns block (j, i) of A^T then
12:             receive block b from node t
13:             block (j, i) of A^T ← b^T
14:         end if
15:     end for
16:     return A^T
17: end function
```

MPI offre strumenti ottimizzati per l'invio di buffer di grandi dimensioni, includendo la possibilità di definire tipi derivati personalizzati che rappresentano strutture complesse come blocchi rettangolari di una matrice. Questo rende il trasferimento dei blocchi particolarmente efficiente.

La complessità principale rimane comunque legata al numero totale di blocchi della matrice: per completare la trasposizione è necessario che ogni nodo invii e riceva un numero di blocchi proporzionale alla dimensione globale della matrice. Di conseguenza, l'algoritmo comporta un numero di operazioni di comunicazione pari al numero totale di blocchi, pur beneficiando dell'ottimizzazione data dal loro trattamento aggregato.

L'algoritmo può essere ulteriormente migliorato quando si utilizza una griglia di processi quadrata. In questo caso, ogni nodo comunica sempre con un unico nodo specifico, quello simmetrico rispetto alla diagonale della griglia: il nodo (x, y) scambia i blocchi esclusivamente con (y, x). Questo consente di sostituire le molteplici operazioni di `send`/`recv` per ciascun blocco con una singola operazione di invio e una di ricezione contenenti tutti i blocchi necessari, con un notevole miglioramento delle prestazioni complessive.

La funzione che realizza la trasposizione di una matrice distribuita è `Slac::transpose`.

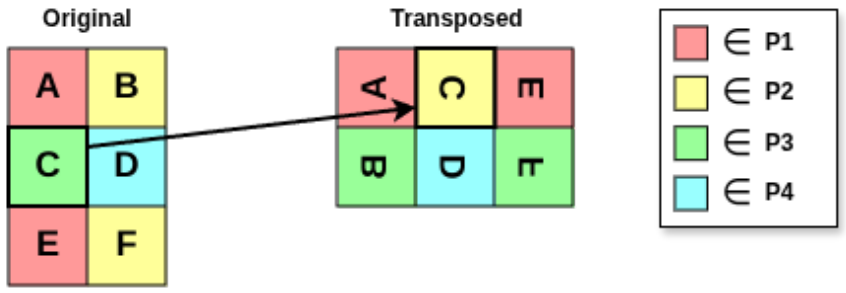

Figura 4.3: Logica di trasposizione delle matrici distribuite.

4.2.3 Analisi delle prestazioni

Le tabelle 4.3 e 4.4 riportano i tempi di esecuzione della funzione `Slac::transpose` per diversi numeri di nodi e dimensioni della matrice, con blocchi di lato 32 e 64. Sono

inoltre mostrati i grafici dei tempi di esecuzione (Fig. 4.4) e quelli relativi a speedup ed efficienza (Fig. 4.5).

Tabella 4.3: Tempi di esecuzione della funzione `Slac::transpose` (*blocksize* = 32)

Dimensione matrice (px)	Tempo di esecuzione (s)			
	1 nodo	4 nodi	8 nodi	12 nodi
256×256	0.0007	0.0003	0.0001	0.0001
512×512	0.0021	0.0007	0.0004	0.0005
1024×1024	0.0088	0.0029	0.0017	0.0020
1536×1536	0.0253	0.0073	0.0046	0.0049
2048×2048	0.0513	0.0134	0.0089	0.0094
2560×2560	0.0918	0.0208	0.0140	0.0147

Tabella 4.4: Tempi di esecuzione della funzione `Slac::transpose` (*blocksize* = 64)

Dimensione matrice (px)	Tempo di esecuzione (s)			
	1 nodo	4 nodi	8 nodi	12 nodi
256×256	0.0006	0.0003	0.00006	0.00006
512×512	0.0021	0.0008	0.0004	0.0005
1024×1024	0.0090	0.0028	0.0013	0.0014
1536×1536	0.0247	0.0073	0.0034	0.0046
2048×2048	0.0509	0.0135	0.0065	0.0067
2560×2560	0.0937	0.0208	0.0102	0.0108

Dalle curve in Fig. 4.5 emerge chiaramente la presenza di due picchi corrispondenti a 4 e 9 nodi, cioè alle griglie quadrate 2×2 e 3×3. Questo comportamento conferma l'efficacia dell'ottimizzazione introdotta per le griglie quadrate: in tali configurazioni ogni nodo comunica con un unico nodo "simmetrico", riducendo drasticamente il numero totale di messaggi e privilegiando il lavoro locale. L'ottimizzazione risulta pertanto determinante per migliorare prestazioni ed efficienza complessiva dell'algoritmo di trasposizione.

4.3 Moltiplicazione matriciale

4.3.1 Design

La moltiplicazione tra matrici rappresenta una delle operazioni fondamentali dell'algebra lineare e, al tempo stesso, una delle più complesse da distribuire in modo efficiente su cluster. Il problema nasce dal fatto che il calcolo di ciascun elemento (i, j) della matrice risultato $R = P \times Q$ richiede l'intera riga i della matrice P e l'intera colonna j della matrice Q. Nelle matrici distribuite, tuttavia, nessun nodo possiede per intero una riga

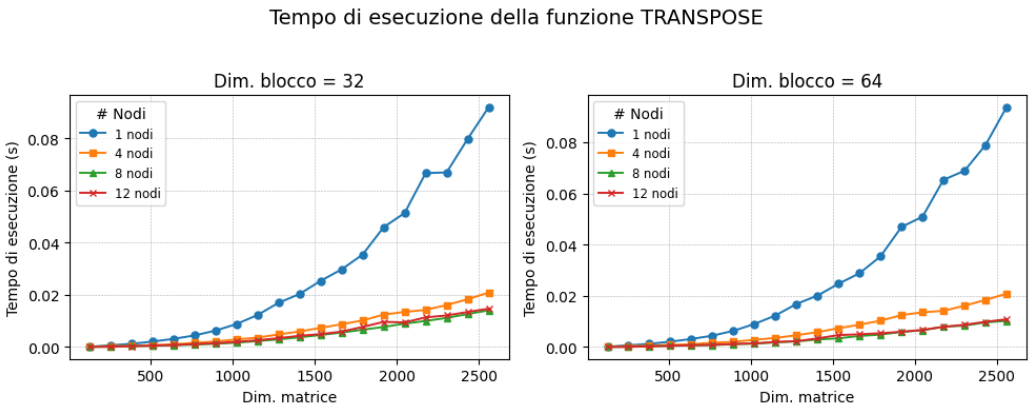

Figura 4.4: Tempi di esecuzione della funzione `Slac::transpose` misurati su diverse dimensioni dei blocchi.

o una colonna: ogni processo detiene solo una porzione locale della matrice globale, rendendo necessarie comunicazioni coordinate tra nodi.

Per eseguire la moltiplicazione matrice per matrice in modo efficiente viene utilizzato una variazione dell'algoritmo SUMMA (Scalable Universal Matrix Multiplication Algorithm) [11]. SUMMA suddivide il prodotto in "fasi" successive: a ogni iterazione, una banda di colonne di P e una banda di righe di Q vengono trasmesse rispettivamente lungo le righe e lungo le colonne della griglia di processi. I nodi, una volta ricevuti i blocchi rilevanti, possono aggiornare localmente la porzione del risultato. La variazione usata invece consiste nell'aggregare tutti i blocchi di una riga/colonna prima di aggiornare la matrice risultato, e poi eseguire il calcolo; in questo modo si sfruttano meglio le ottimizzazioni locali (Fig. 4.6).

Un semplice pseudocodice dell'algoritmo usato è il seguente:

1: **function** SUMMA(P, Q)
2: allocate R to store $P \times Q$
3: $max\_it \leftarrow$ iterations needed to fetch all row and columns
4: **for** $it = 0$ **to** $max\_it - 1$ **do**
5: $P^{(it)}_{rows} \leftarrow P$ rows aggregated from same process row
6: $Q^{(it)}_{cols} \leftarrow Q$ columns aggregated from same process column
7: $R^{(it)} \leftarrow$ local portion of R corresponding to the rows of $P^{(it)}_{rows}$ and the columns of $Q^{(it)}_{cols}$
8: $R^{(it)} \leftarrow P^{(it)}_{rows} \times Q^{(it)}_{cols}$
9: **end for**
10: **return** R
11: **end function**

La distribuzione ciclica a blocchi offre un vantaggio sostanziale: le comunicazioni necessarie riguardano esclusivamente i nodi appartenenti alla stessa riga (per i blocchi

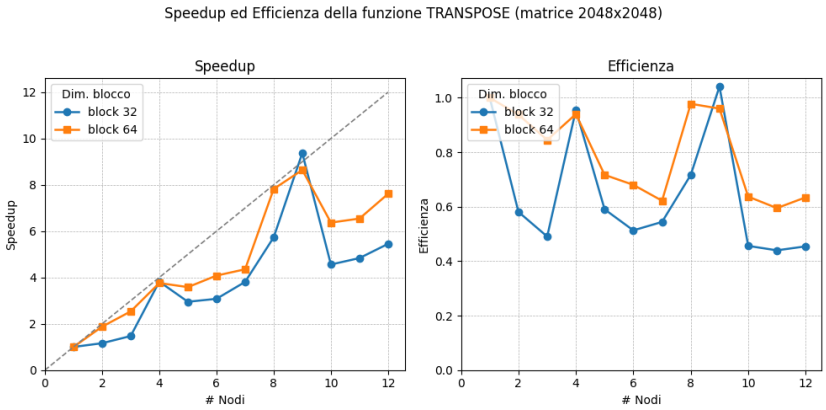

Figura 4.5: Speedup ed efficienza della funzione `Slac::transpose` con dimensione della matrice fissa 2048×2048.

della prima matrice) o alla stessa colonna (per i blocchi della seconda). Questo perché ogni nodo necessita soltanto dei blocchi posseduti dai processi della propria riga o colonna logica. Ne deriva che nodi appartenenti a righe diverse possono eseguire in parallelo la fase di aggregazione.

4.3.2 Implementazione

In SLAC la moltiplicazione matriciale è progettata per funzionare non solo tra matrici entrambe distribuite, ma anche tra matrici di natura diversa (ad esempio: matrice locale $\times$ matrice distribuita, matrice distribuita $\times$ matrice locale, ecc.). Per ottenere questo comportamento flessibile è stata introdotta una gerarchia di classi ausiliarie, denominate `MatmulInfo`. Queste classi incapsulano tutte le informazioni necessarie per recuperare, all'occorrenza, le righe o le colonne di una matrice, indipendentemente dalla sua rappresentazione interna.

Ogni `MatmulInfo` fornisce due funzionalità principali:

- un metodo per ottenere i dati necessari al calcolo (righe o colonne, eventualmente distribuite o locali);
- un metodo per determinare il numero di iterazioni richieste dal prodotto. Questo è necessario quando il volume di dati da scambiare è troppo grande per essere mantenuto simultaneamente in memoria: in tal caso il calcolo viene suddiviso in più fasi, riducendo l'impatto sulla memoria locale.

La classe `MatmulInfo` è stata estesa in quattro varianti specifiche:

- `DistrRow_MatmulInfo` per estrarre le righe da una `DistributedMatrix`;
- `DistrCol_MatmulInfo` per estrarre le colonne da una `DistributedMatrix`;

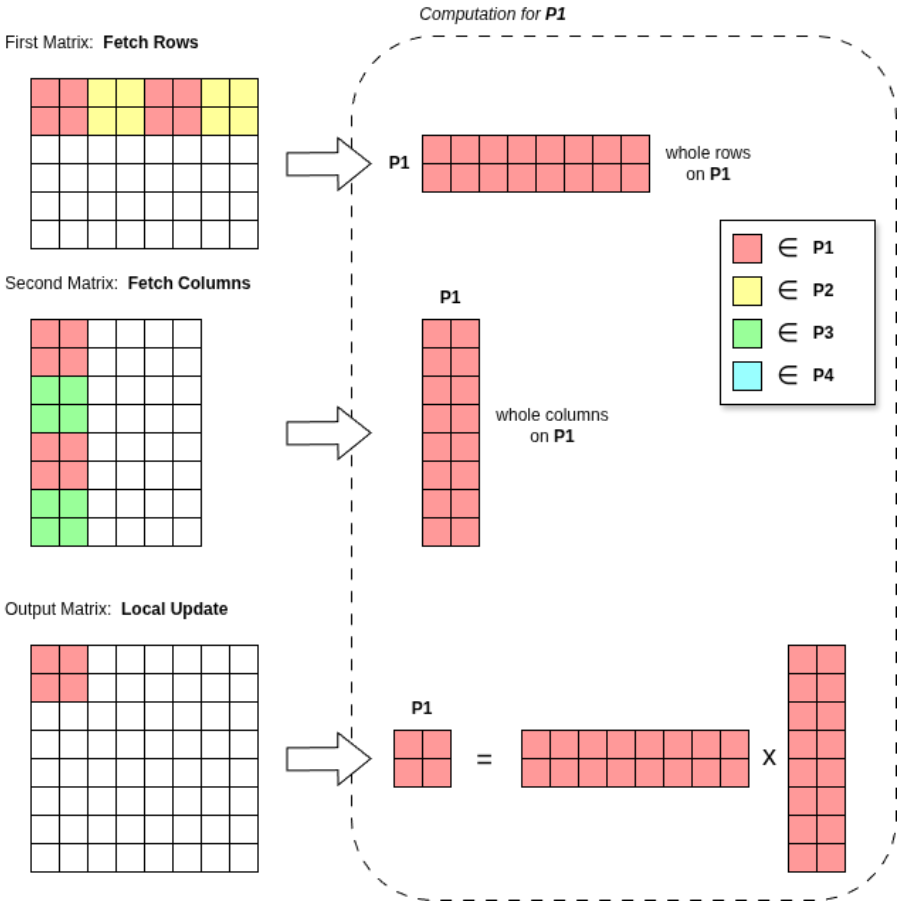

Figura 4.6: Principali fasi dell'algoritmo per la moltiplicazione matriciale per il nodo $P1$. Prima ottiene le righe e colonne comunicando con gli altri nodi, poi esegue localmente la moltiplicazione.

- `LocalRow_MatmulInfo` per estrarre le righe da una `LocalMatrix`;
- `LocalCol_MatmulInfo` per estrarre le colonne da una `LocalMatrix`.

La combinazione di queste classi permette di comporre qualsiasi forma di moltiplicazione, mantenendo la logica di gestione dei dati completamente astratta rispetto al tipo concreto della matrice (vedi Fig. 4.7 per uno schema riassuntivo).

Il modo in cui si inviano e ricevono le righe e le colonne delle matrici è stato progettato per sfruttare al meglio le capacità di MPI attraverso la definizione di *datatype* personalizzati. Questi tipi derivati permettono di trasmettere dati in modo efficiente, soprattutto quando non sono contigui in memoria. Nel nostro caso, la necessità di ricostruire righe e colonne della matrice reale a partire dai blocchi distribuiti rende questo

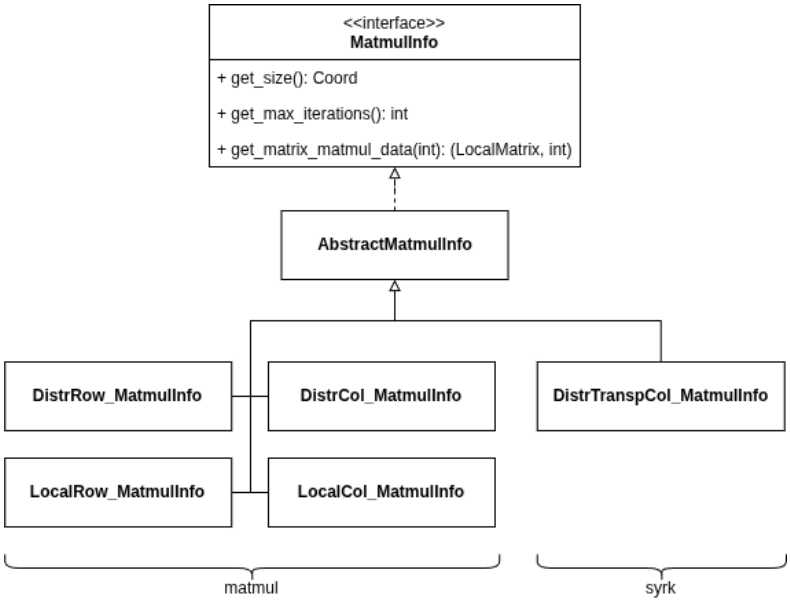

Figura 4.7: Schema UML delle classi MatmulInfo e le sue estensioni

meccanismo particolarmente adatto: i blocchi provenienti dai vari nodi possono essere ricevuti ed inseriti automaticamente nelle posizioni corrette.

In ogni nodo i dati locali sono contigui in memoria, quindi l'operazione di invio è semplice ed efficiente. Sul lato della ricezione, invece, i datatype MPI gestiscono lo stride tra i blocchi, consentendo di depositare ciascun elemento direttamente nella posizione finale che avrà nella riga o colonna globale, senza passaggi intermedi né copie aggiuntive.

Le funzioni incaricate dell'invio e dell'assemblaggio delle righe e delle colonne sono `fetch_global_rows`, utilizzata da `DistrRow_MatmulInfo`, e `fetch_global_cols`, utilizzata da `DistrCol_MatmulInfo`. All'interno di esse vengono impiegati rispettivamente le strutture dati `rows_datatype` e `cols_datatype`, che definiscono la struttura dei segmenti da inviare e ricevere, permettendo una ricostruzione diretta e ordinata dei blocchi nella matrice globale.

Il calcolo della moltiplicazione matriciale viene eseguito nel metodo `generic_matmul`, che implementa una versione generica del prodotto tra due `MatmulInfo`. Al suo interno viene applicato l'algoritmo SUMMA, adattato alle esigenze della libreria. La divisione del calcolo in iterazioni è resa possibile grazie a un meccanismo essenziale fornito da Eigen: le *block operations*. Questo consente di operare direttamente su porzioni di matrice, migliorando l'efficienza, riducendo le copie e permettendo alla procedura di essere compatibile con matrici di dimensioni anche molto elevate.

La funzione che esegue la moltiplicazione matriciale è `Slac::matmul`.

4.3.3 Analisi delle prestazioni

Le tabelle 4.5 e 4.6 mostrano i tempi di esecuzione della funzione `Slac::matmul` per diverse dimensioni della matrice, utilizzando blocchi di lato 32 e 64. I grafici dei tempi (Fig. 4.8) e quelli relativi a speedup ed efficienza (Fig. 4.9) completano l'analisi.

Tabella 4.5: Tempi di esecuzione della funzione `Slac::matmul` ($blocksize = 32$)

Dimensione matrice (px)	Tempo di esecuzione (s)			
	1 nodo	4 nodi	8 nodi	12 nodi
256×256	0.0074	0.0311	0.0713	0.1335
512×512	0.0494	0.0668	0.1359	0.2420
1024×1024	0.3659	0.1811	0.2752	0.4026
1536×1536	1.1894	0.3173	0.3717	0.4890
2048×2048	2.7711	0.6150	0.6609	0.7608
2560×2560	5.3506	0.8520	0.9158	1.1570

Tabella 4.6: Tempi di esecuzione della funzione `Slac::matmul` ($blocksize = 64$)

Dimensione matrice (px)	Tempo di esecuzione (s)			
	1 nodo	4 nodi	8 nodi	12 nodi
256×256	0.0074	0.0287	0.0655	0.1507
512×512	0.0494	0.0714	0.1329	0.2753
1024×1024	0.3671	0.1774	0.2681	0.4673
1536×1536	1.1887	0.3042	0.3837	0.5185
2048×2048	2.7702	0.6048	0.6568	0.7937
2560×2560	5.3665	0.9081	0.9069	1.1951

Dal grafico in Fig. 4.9 emerge chiaramente che la moltiplicazione distribuita non scala in modo perfettamente lineare. Il motivo principale risiede nelle numerose comunicazioni necessarie per scambiare i blocchi tra nodi: mentre le moltiplicazioni locali avvengono in parallelo, il costo dominante diventa proprio il traffico MPI.

Nel caso con tre nodi si osserva uno speedup apparentemente superlineare. Questo comportamento, sebbene controintuitivo, può essere spiegato considerando alcune ottimizzazioni interne di Eigen. Riducendo la dimensione dei blocchi locali su cui vengono eseguite le moltiplicazioni, la libreria può selezionare varianti più efficienti dei propri kernel di calcolo, ottenendo dunque prestazioni migliori rispetto all'esecuzione sequenziale su matrici più grandi.

I benefici derivanti dall'impiego di un numero crescente di nodi diventano particolarmente evidenti per matrici di dimensioni elevate, nelle quali il costo computazionale della moltiplicazione domina su quello delle comunicazioni. In questi scenari, la paralle-

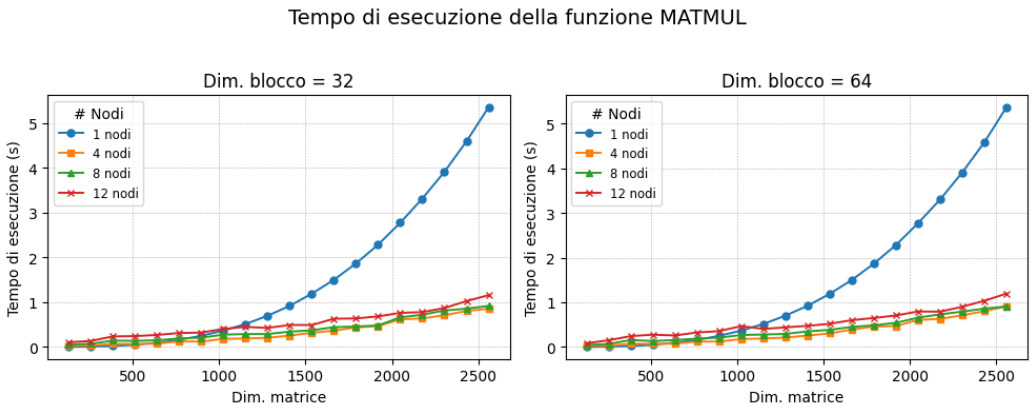

Figura 4.8: Tempi di esecuzione della funzione `Slac::matmul` misurati su diverse dimensioni dei blocchi.

lizzazione consente una riduzione significativa del tempo totale di esecuzione e permette di sfruttare in modo più efficace la memoria locale di ciascun nodo.

4.4 Aggiornamento simmetrico di rango k

4.4.1 Design

L'aggiornamento simmetrico di rango k (*symmetric rank-k update*, spesso indicato come SYRK) rappresenta un caso particolare di moltiplicazione matriciale in cui la matrice risultante è necessariamente simmetrica. L'operazione di base consiste nel calcolo:

$$C \leftarrow AA^T$$

dove A è la matrice di input e C è la matrice di output, che risulta simmetrica per costruzione. In altre parole, invece di combinare righe di una matrice con colonne di un'altra, si combinano righe di A con le corrispondenti righe di A interpretate come colonne di A^T. Questo consente di evitare l'esplicita trasposizione della matrice, operazione costosa e non necessaria ai fini del calcolo.

L'idea chiave è quindi quella di riformulare il prodotto AA^T come un'aggregazione riga-per-riga, sfruttando il fatto che la colonna da utilizzare in ogni passo non è altro che la riga corrispondente della matrice originale. Ciò permette ottimizzazioni sia sul piano computazionale sia su quello della comunicazione nei contesti distribuiti.

L'operazione è stata ulteriormente estesa includendo una matrice diagonale D con diagonale d, consentendo il calcolo:

$$C \leftarrow ADA^T$$

Questa generalizzazione è di grande utilità in contesti fisico-computazionali, dove aggiornamenti pesati lungo specifiche direzioni - rappresentati proprio da una matrice diago-

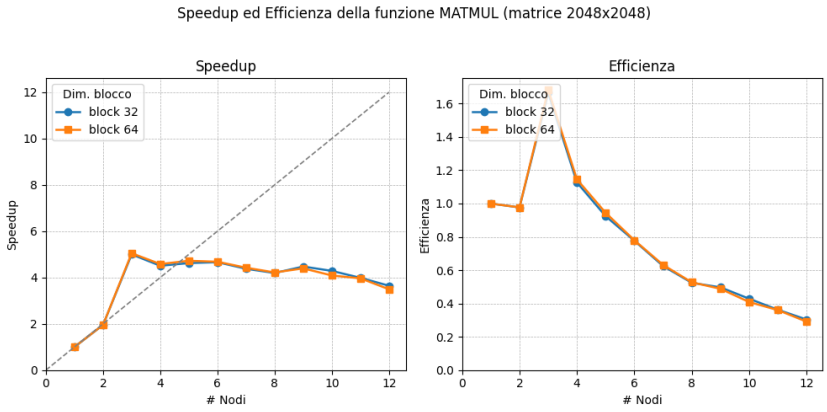

Figura 4.9: Speedup ed efficienza della funzione `Slac::matmul` con dimensione della matrice fissa 2048×2048.

nale - sono frequenti. Integrare tale funzionalità direttamente nell'operazione permette di ridurre passaggi intermedi e di migliorare prestazioni e chiarezza del codice.

L'operazione si ottiene applicando una piccola modifica all'algoritmo SUMMA usato per la moltiplicazione matriciale: durante un'iterazione, prima di eseguire il prodotto locale sulle righe e colonne aggregate, si moltiplicano le colonne per la matrice diagonale D. Uno schema semplificato dell'algoritmo è il seguente:

function SYRK(P, d)
    allocate R to store $P \times P^T$
    $D \leftarrow$ diagonal matrix with diagonal d
    $max\_it \leftarrow$ iterations needed to fetch all row and columns
    for $it = 0$ **to** $max\_it - 1$ **do**
        $P^{(it)}_{rows} \leftarrow P$ rows aggregated from same process row
        $P^{(it)}_{cols} \leftarrow P$ rows corresponding to P^T columns, aggregated from same process row
        $R^{(it)} \leftarrow$ local portion of R corresponding to the rows of $P^{(it)}_{rows}$ and the columns of $Q^{(it)}_{cols}$
        $R^{(it)} \leftarrow P^{(it)}_{rows} \times (D \times P^{(it)}_{cols})$
    end for
    return R
end function

4.4.2 Implementazione

All'interno di SLAC, l'aggiornamento simmetrico di rango k è implementato sfruttando la stessa infrastruttura utilizzata per la moltiplicazione generica, ossia la funzione `generic_matmul`. Per rendere possibile il calcolo senza costruire esplicitamente la ma-

trice trasposta A^T, è stata introdotta una variante dedicata della classe `MatmulInfo` (Fig. 4.7).

Questa nuova classe, chiamata `DistrTranspCol_MatmulInfo`, fornisce al sistema le colonne logiche di A^T senza eseguire alcuna trasposizione reale della matrice: in pratica, quando è necessario ottenere la colonna j della trasposta, essa si limita a recuperare la riga j della matrice originale, che contiene esattamente gli stessi elementi nella stessa sequenza. In questo modo si evita sia il costo computazionale della trasposizione, sia la necessità di mantenere in memoria una copia aggiuntiva della matrice.

L'algoritmo risultante si integra perfettamente con SUMMA, mantenendo la stessa struttura distribuita ma riducendo sensibilmente la quantità di dati da generare o trasformare.

La funzione che esegue questa operazione è `Slac::syrk`.

4.4.3 Analisi delle prestazioni

Le tabelle 4.7 e 4.8 riportano i tempi di esecuzione della funzione `Slac::syrk` con blocchi di lato 32 e 64. I corrispondenti grafici dei tempi (Fig. 4.10) e quelli relativi a speedup ed efficienza (Fig. 4.11) mostrano l'andamento delle prestazioni.

Tabella 4.7: Tempi di esecuzione della funzione `Slac::syrk` ($blocksize = 32$)

Dimensione matrice (px)	Tempo di esecuzione (s)			
	1 nodo	4 nodi	8 nodi	12 nodi
256×256	0.0079	0.0289	0.0730	0.1502
512×512	0.0505	0.0730	0.1327	0.2705
1024×1024	0.3711	0.1744	0.2730	0.4746
1536×1536	1.2060	0.3131	0.4086	0.5599
2048×2048	2.8237	0.6541	0.7869	0.9277
2560×2560	5.4644	0.9975	1.1093	1.3713

Tabella 4.8: Tempi di esecuzione della funzione `Slac::syrk` ($blocksize = 64$)

Dimensione matrice (px)	Tempo di esecuzione (s)			
	1 nodo	4 nodi	8 nodi	12 nodi
256×256	0.0079	0.0324	0.0761	0.1545
512×512	0.0505	0.0715	0.1458	0.2692
1024×1024	0.3705	0.1900	0.2782	0.4502
1536×1536	1.2057	0.3386	0.4068	0.5549
2048×2048	2.8223	0.6318	0.7849	0.9344
2560×2560	5.4570	1.0158	1.1159	1.3500

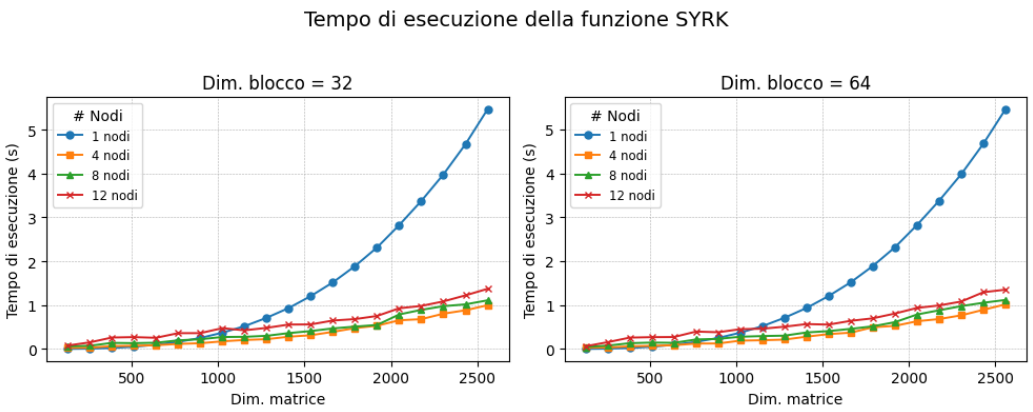

Figura 4.10: Tempi di esecuzione della funzione `Slac::syrk` misurati su diverse dimensioni dei blocchi.

Dal punto di vista delle prestazioni, la `syrk` presenta un comportamento simile alla moltiplicazione matriciale, di cui rappresenta una variante specializzata. Si osservano tempi leggermente superiori rispetto alla `matmul`: la causa principale è legata alla modalità con cui vengono reperite le colonne necessarie per la moltiplicazione, secondo il metodo descritto nel paragrafo precedente.

Questa fase richiede un maggior coordinamento tra i nodi rispetto alla moltiplicazione standard, nella quale i blocchi devono essere scambiati solo all'interno della stessa colonna di processi. Nel caso della `syrk`, invece, i nodi appartenenti alla stessa riga inviano più blocchi rispetto agli altri, aumentando il traffico e riducendo l'efficienza. Il principale vantaggio della `syrk` risiede tuttavia nel non richiedere l'allocazione di una seconda matrice completa, caratteristica molto importante in contesti con memoria limitata o in presenza di matrici di dimensioni molto elevate.

Anche per questa computazione emerge uno speedup superlineare nel caso con tre nodi. Il fenomeno ha la stessa origine osservata per `matmul`: trattandosi di un caso particolare di moltiplicazione matriciale, eredita sia il comportamento che i meccanismi di ottimizzazione della funzione generale. Gli stessi effetti legati alla riduzione dei blocchi locali e alla selezione di kernel più efficienti spiegano quindi l'incremento di prestazioni riscontrato in questo caso.

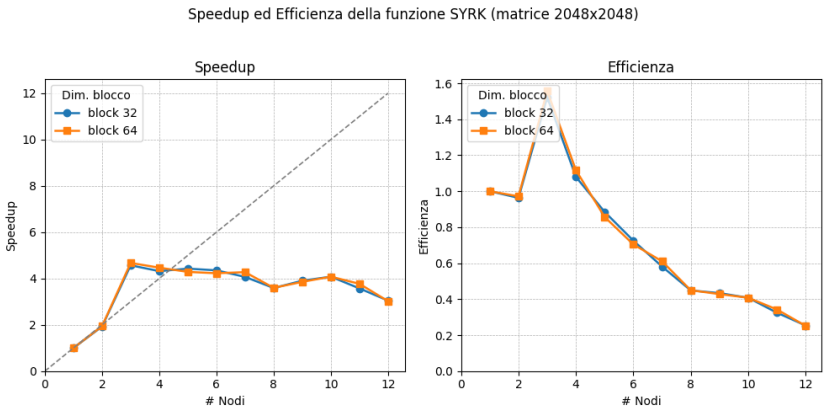

Figura 4.11: Speedup ed efficienza della funzione `Slac::syrk` con dimensione della matrice fissa 2048×2048.

Capitolo 5

Risoluzione di sistemi lineari distribuiti

La risoluzione di sistemi lineari costituisce uno degli strumenti fondamentali in numerosi ambiti scientifici e ingegneristici. Gran parte dei modelli matematici alla base della fisica computazionale, dell'analisi numerica, dell'ingegneria strutturale e dei metodi di ottimizzazione porta infatti alla formulazione di sistemi lineari di grandi dimensioni, spesso malcondizionati e caratterizzati da strutture particolari. Per questo motivo, la disponibilità di strumenti efficienti per la loro risoluzione è un requisito centrale per qualunque libreria di algebra lineare distribuita.

Sebbene nelle sue prime versioni e con un insieme di funzionalità ancora limitato, SLAC offre già metodi in grado di risolvere sistemi lineari triangolari e sistemi simmetrici definiti positivi. Nel seguito vengono descritti nel dettaglio i metodi implementati e le strategie progettuali adottate.

5.1 Risoluzione di sistemi triangolari

La risoluzione di sistemi triangolari rappresenta un'operazione di base nei solver più diffusi: molte procedure di fattorizzazione (come LU, QR e Cholesky) portano infatti alla risoluzione di sequenze di sistemi triangolari. Affrontare tali sistemi in modo efficiente è quindi essenziale. SLAC mette a disposizione due risolutori distinti: `Slac::trsm_lower` per matrici triangolari inferiori e `Slac::trsm_upper` per matrici triangolari superiori.

5.1.1 Design

Per risolvere un sistema triangolare inferiore si utilizza l'algoritmo di *forward substitution* [13], mentre per un sistema triangolare superiore si usa la corrispondente *backward substitution*. Entrambi si basano sulla stessa idea: ogni incognita viene calcolata sfruttando le soluzioni già determinate alle equazioni precedenti (o successive, nel caso della matrice superiore).

Un classico pseudocodice per la forward substitution è il seguente:

```
1: function LOCAL_TRSM(L, b)
2:     for i = 0 to n − 1 do
3:         s ← b[i]
4:         for j = 0 to i − 1 do
5:             s ← s − L[i][j] · x[j]
6:         end for
7:         x[i] ← s/L[i][i]
8:     end for
9:     return x
10: end function
```

Questi algoritmi hanno complessità lineare e presentano una struttura intrinsecamente sequenziale: ogni valore di x_i dipende da tutti i valori precedenti, il che limita il parallelismo possibile. Tuttavia, è comunque possibile parallelizzare il calcolo dei prodotti intermedi $L[i][j] \cdot x[j]$, distribuendoli tra i nodi che possiedono i blocchi rilevanti.

Con la distribuzione ciclica a blocchi, l'algoritmo può essere trasformato in una versione *blocked*. Gli algoritmi di tipo blocked traggono vantaggio dal lavorare su blocchi quadrati contigui in memoria, migliorando l'efficienza grazie alla località dei dati e all'uso dei metodi di risoluzione ottimizzati per le matrici locali.

Una descrizione semplificata dello schema blocked è la seguente:

```
1: function BLOCKED_TRSM(M, b)
2:     s ← b
3:     for each diagonal block D_{i,i} in M do
4:         if node owns block D_{i,i} then
5:             s ← LOCAL_TRSM(D_{i,i}, s)
6:             send s to nodes owning block A_{i,y}, ∀y
7:         else if node owns block A_{i,y}, ∀y then
8:             receive s′ (solution of block D_{i,i})
9:             s ← A_{i,y} · s′
10:        end if
11:    end for
12:    return s
13: end function
```

Questo approccio permette a ciascun nodo di contribuire al calcolo mantenendo la massima località possibile e riducendo il volume di comunicazione alla sola necessaria propagazione delle soluzioni parziali.

5.1.2 Implementazione

L'implementazione basata su `DistributedMatrix` segue fedelmente la struttura dell'algoritmo a blocchi. La soluzione viene calcolata dai soli nodi che ospitano le componenti finali del vettore soluzione: poiché si tratta di un `DistributedVector` colonna, tali nodi corrispondono a quelli situati nella prima colonna della griglia dei processi (Fig. 5.1).

Questi nodi recuperano dai nodi disposti lungo le righe i blocchi di riga necessari, così da costruire localmente la porzione completa dei dati utili al calcolo. Una volta ricevute le informazioni, essi eseguono la versione blocked dell'algoritmo, riducendo drasticamente le comunicazioni e beneficiando della località dei dati e del caching.

Le righe globali si ottengono tramite la classe `DistrRow_MatmulInfo`. Sebbene questa fosse stata inizialmente progettata per supportare esclusivamente la moltiplicazione matriciale, essa fornisce già le funzionalità necessarie per recuperare le righe globali; per questo motivo si è scelto di riutilizzarla anche in questo contesto.

Le righe globali estratte vengono quindi rappresentate come `LocalMatrix`, che si appoggiano alle primitive ottimizzate di Eigen. Tale scelta consente di eseguire in modo efficiente tutta la parte computazionale locale, come la risoluzione dei sistemi triangolari e gli aggiornamenti progressivi delle soluzioni, garantendo prestazioni elevate.

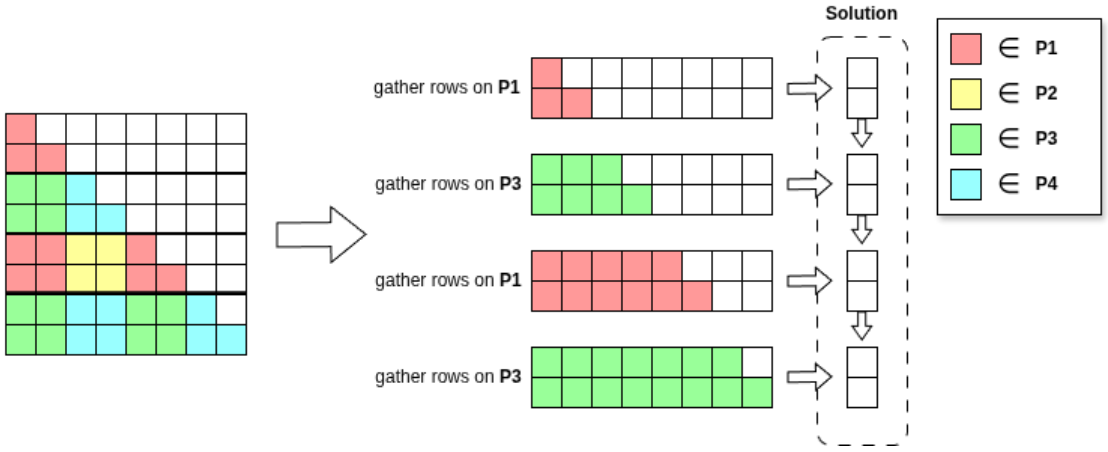

Figura 5.1: Calcolo della soluzione di un sistema triangolare distribuito. Le righe vengono aggregate sui nodi della prima colonna dei processi, che calcoleranno localmente la soluzione.

5.1.3 Analisi delle prestazioni

Le tabelle 5.1 e 5.2 riportano i tempi di esecuzione per diversi numeri di nodi e dimensioni della matrice, utilizzando rispettivamente blocchi di lato 32 e 64. Sono inoltre presentati i grafici relativi ai tempi di esecuzione (Fig. 5.2), nonché lo speedup e l'efficienza per matrici di dimensione fissata 2048×2048 (Fig. 5.3).

Dal grafico in Fig. 5.3 emerge un comportamento apparentemente anomalo dello speedup, spiegabile conoscendo la dimensione della griglia dei processi e l'algoritmo utilizzato. In particolare, quando la griglia si riduce a una sola riga di processi (situazione che si verifica quando non è possibile costruire una griglia bidimensionale bilanciata), l'esecuzione effettiva ricade interamente sul primo processo della riga. Ne consegue che lo speedup risulta pari a uno: il comportamento è quindi assimilabile alla versione

Tabella 5.1: Tempi di esecuzione della funzione `Slac::trsm_lower` ($blocksize = 32$)

Dimensione matrice (px)	Tempo di esecuzione (s)			
	1 nodo	4 nodi	8 nodi	12 nodi
256×256	0.0005	0.0002	0.0002	0.0002
512×512	0.0020	0.0005	0.0006	0.0005
1024×1024	0.0082	0.0015	0.0017	0.0015
1536×1536	0.0204	0.0052	0.0054	0.0043
2048×2048	0.0401	0.0103	0.0111	0.0083
2560×2560	0.0633	0.0163	0.0177	0.0131

Tabella 5.2: Tempi di esecuzione della funzione `Slac::trsm_lower` ($blocksize = 64$)

Dimensione matrice (px)	Tempo di esecuzione (s)			
	1 nodo	4 nodi	8 nodi	12 nodi
256×256	0.0005	0.0002	0.0002	0.0002
512×512	0.0021	0.0006	0.0005	0.0005
1024×1024	0.0084	0.0014	0.0016	0.0015
1536×1536	0.0213	0.0047	0.0044	0.0036
2048×2048	0.0380	0.0089	0.0087	0.0068
2560×2560	0.0632	0.0143	0.0143	0.0126

sequenziale, con l'aggravante che i dati non sono inizialmente disponibili in memoria locale.

Questo approccio può essere migliorato distribuendo il carico di lavoro in modo più uniforme tra i processi. L'implementazione corrente privilegia invece il riutilizzo di funzioni già esistenti e la riduzione del numero di comunicazioni.

5.2 Decomposizione di Cholesky

La decomposizione di Cholesky è una fattorizzazione fondamentale nel calcolo scientifico: permette di riscrivere una matrice simmetrica definita positiva A nella forma

$$A = LL^T$$

dove L è triangolare inferiore. Questa operazione costituisce la base del solver per sistemi lineari definiti positivi, ed è quindi stata implementata direttamente in `Slac` nel metodo `Slac::cholesky`.

5.2.1 Design

L'algoritmo classico di Cholesky presenta una complessità computazionale pari a

$$\mathcal{O}(n^3)$$

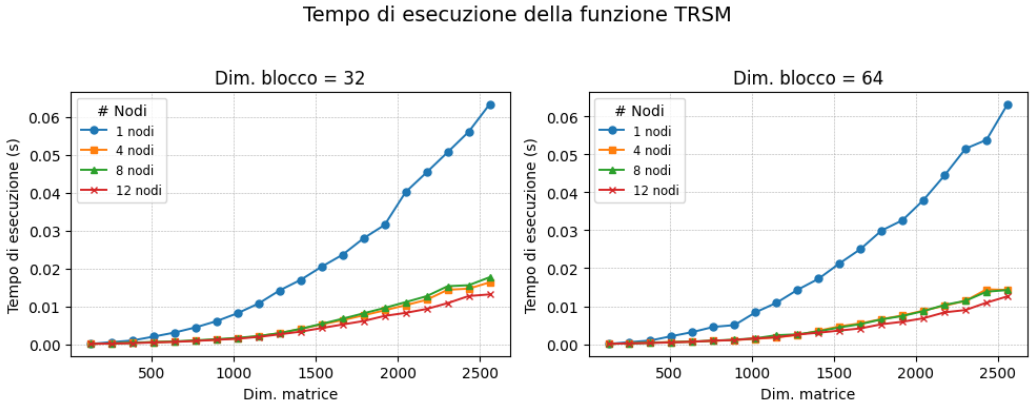

Figura 5.2: Tempi di esecuzione della funzione `Slac::trsm_lower` misurate su diverse dimensioni dei blocchi.

ed è intrinsecamente difficile da parallelizzare: ogni riga (o colonna) dipende da tutte le precedenti, e la propagazione delle dipendenze limita il parallelismo. In un contesto distribuito, questo si traduce in comunicazioni frequenti e sincronizzazioni costose.

Per mitigare queste difficoltà si utilizza la versione *blocked* dell'algoritmo [10] (Fig. 5.4). Una matrice simmetrica definita positiva può essere partizionata nel seguente modo:

$$\left(\begin{array}{c|c} A_{11} & \star \\ \hline A_{21} & A_{22} \end{array}\right) = \left(\begin{array}{c|c} L_{11} & 0 \\ \hline L_{21} & L_{22} \end{array}\right) \left(\begin{array}{c|c} L_{11} & 0 \\ \hline L_{21} & L_{22} \end{array}\right)^T = \left(\begin{array}{c|c} L_{11}L_{11}^T & \star \\ \hline L_{21}L_{11}^T & L_{21}L_{21}^T + L_{22}L_{22}^T \end{array}\right).$$

Dal quale si deriva che:

$$\begin{aligned} L_{11} &= \mathrm{Chol}(A_{11}), \\ L_{21} &= A_{21}L_{11}^{-T}, \\ L_{22} &= \mathrm{Chol}\big(A_{22} - L_{21}L_{21}^T\big). \end{aligned}$$

L'idea dell'algoritmo è quella di suddividere la matrice in blocchi quadrati e procedere ricorsivamente (Fig. 5.4):

1. si calcola la fattorizzazione di Cholesky sul blocco diagonale corrente;
2. si aggiorna la colonna relativa al blocco appena calcolato;
3. si aggiorna la sottomatrice rimanente;
4. si ripete sul blocco successivo.

Questo approccio riduce drasticamente la quantità di comunicazione, massimizza il lavoro locale sui blocchi e permette un'elevata efficienza anche su matrici distribuite.

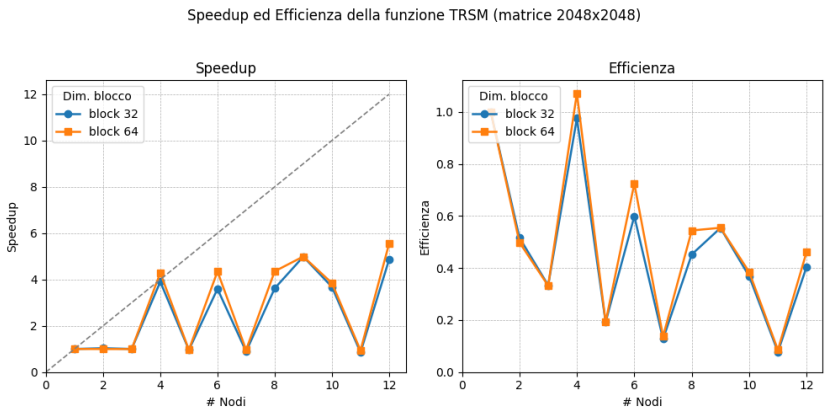

Figura 5.3: Speedup ed efficienza della funzione `Slac::trsm_lower` con dimensione della matrice fissa 2048×2048.

5.2.2 Implementazione

L'implementazione della fattorizzazione di Cholesky in SLAC segue fedelmente lo schema distribuito a blocchi, riproducendo passo per passo la struttura dell'algoritmo classico per matrici suddivise su una griglia di processi. Le `DistributedMatrix` sono progettate per delegare tutte le operazioni locali a Eigen, incluse le *block operations*, che permettono un aggiornamento estremamente efficiente sia dei blocchi locali sia delle sottomatrici.

L'algoritmo procede secondo le sue tre fasi canoniche. Nel primo passo, il processo che possiede il blocco diagonale calcola localmente e *in-place* la fattorizzazione di Cholesky del proprio blocco. Questa operazione è completamente locale e non richiede alcuna comunicazione.

Successivamente, il blocco appena fattorizzato deve essere propagato ai nodi che possiedono i blocchi nella stessa riga e nella stessa colonna. Questa fase è realizzata tramite operazioni di broadcast. Nella versione attuale il broadcast è implementato in modo semplice e sequenziale: ciò garantisce correttezza, ma può diventare un collo di bottiglia sulle configurazioni con molti nodi. Una versione più avanzata, non ancora implementata, integrerebbe un broadcast ad albero binario che ridurrebbe drasticamente il numero di messaggi e la latenza complessiva.

Una volta ricevuto il blocco fattorizzato, ciascun nodo aggiorna il proprio blocco locale applicando un solver triangolare (inferiore o superiore a seconda del ruolo nella griglia di processo). Questa scelta è cruciale: sfruttare un solver triangolare locale permette di distribuire il calcolo senza dover ricorrere a inversioni o operazioni meno strutturate, rendendo l'aggiornamento dei pannelli notevolmente più efficiente.

Completata la fase di risoluzione sui pannelli di riga e colonna, i nodi che li possiedono inviano i blocchi aggiornati ai processi che memorizzano la sottomatrice rimanente. Questi ultimi eseguono quindi l'operazione di aggiornamento tramite moltiplicazione matriciale distribuita, sfruttando ancora una volta le primitive locali ottimizzate di Ei-

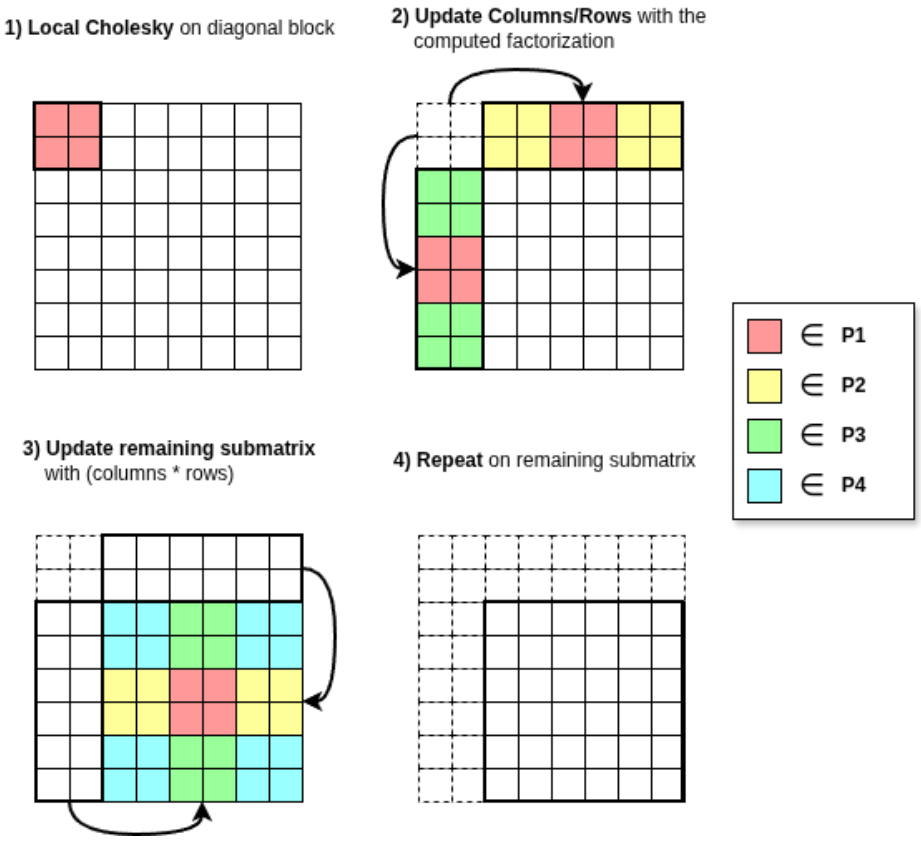

Figura 5.4: Fasi ricorsive dell'algoritmo a blocchi della decomposizione di Cholesky su matrici distribuite. Le fasi comprendono: (1) calcolo locale della fattorizzazione, (2) aggiornamento delle righe e delle colonne con la fattorizzazione trovata, (3) aggiornamento della sottomatrice tramite moltiplicazione matriciale distribuita. L'algoritmo viene ripetuto sulla sottomatrice (4), fino all'elaborazione di tutti i blocchi della diagonale.

gen. A questo punto l'algoritmo procede ricorsivamente alla sottomatrice successiva, ripetendo lo stesso schema fino alla completa fattorizzazione della matrice.

La matrice globale viene sovrascritta interamente con il fattore L nella parte inferiore e con L^T nella parte superiore. Nel contesto distribuito risulta più efficiente aggiornare anche la parte superiore: ciò evita comunicazioni aggiuntive nella fase di aggiornamento e consente di utilizzare direttamente un solver triangolare superiore nella risoluzione dei sistemi simmetrici definiti positivi. Limitarsi alla sola parte inferiore richiederebbe invece l'esecuzione di un aggiornamento simmetrico, meno efficiente sia in termini di messaggi inviati.

Sebbene l'implementazione sia pienamente funzionante e riproduca correttamente tutte le fasi dell'algoritmo distribuito, non rappresenta ancora la versione ottimale. Oltre alla mancanza di broadcast ad albero binario, l'algoritmo non sfrutta ancora il pipelining

delle fasi, che permetterebbe di sovrapporre comunicazione e calcolo riducendo significativamente i tempi complessivi. Questi aspetti costituiscono margini di miglioramento evidenti e saranno al centro dei futuri sviluppi della libreria.

5.2.3 Analisi delle prestazioni

Le tabelle 5.3 e 5.4 riportano i tempi di esecuzione variando il numero di nodi e la dimensione della matrice, utilizzando blocchi di lato 32 e 64. Sono inoltre presentati i grafici relativi ai tempi (Fig. 5.5) e, a dimensione fissata 2048×2048, speedup ed efficienza (Fig. 5.6).

Tabella 5.3: Tempi di esecuzione della funzione `Slac::cholesky` ($blocksize = 32$)

Dimensione matrice (px)	Tempo di esecuzione (s)			
	1 nodo	4 nodi	8 nodi	12 nodi
256×256	0.0036	0.0736	0.1000	0.1358
512×512	0.0244	0.2868	0.6075	0.9748
1024×1024	0.1865	0.7642	1.8796	3.0479
1536×1536	0.6708	1.3136	2.8565	5.2953
2048×2048	1.6436	1.8821	4.1474	7.8447
2560×2560	3.7517	2.7033	5.4660	10.0438

Tabella 5.4: Tempi di esecuzione della funzione `Slac::cholesky` ($blocksize = 64$)

Dimensione matrice (px)	Tempo di esecuzione (s)			
	1 nodo	4 nodi	8 nodi	12 nodi
256×256	0.0034	0.0505	0.1453	0.2537
512×512	0.0221	0.1731	0.4250	0.7322
1024×1024	0.1638	0.4206	1.0345	1.8244
1536×1536	0.5599	0.7216	1.6258	2.9377
2048×2048	1.3202	1.1189	2.3114	4.1773
2560×2560	2.7984	1.4674	3.2595	5.2623

Dai grafici risulta evidente che il problema non scala bene con il numero di nodi: l'aumento dei processi tende spesso a peggiorare le prestazioni. Tale comportamento deriva dalla natura fortemente iterativa e dipendente della decomposizione di Cholesky, che limita il parallelismo e aumenta il numero di comunicazioni. Con un numero ridotto di nodi è inoltre più probabile che i dati necessari siano già presenti localmente, riducendo quindi la latenza comunicativa. Inoltre, proprio perché l'algoritmo non è ancora ottimizzato, il numero di comunicazioni broadcast dipendono dal numero di nodi coinvolti, come emerge chiaramente dai dati sperimentali, e costituiscono un collo di bottiglia

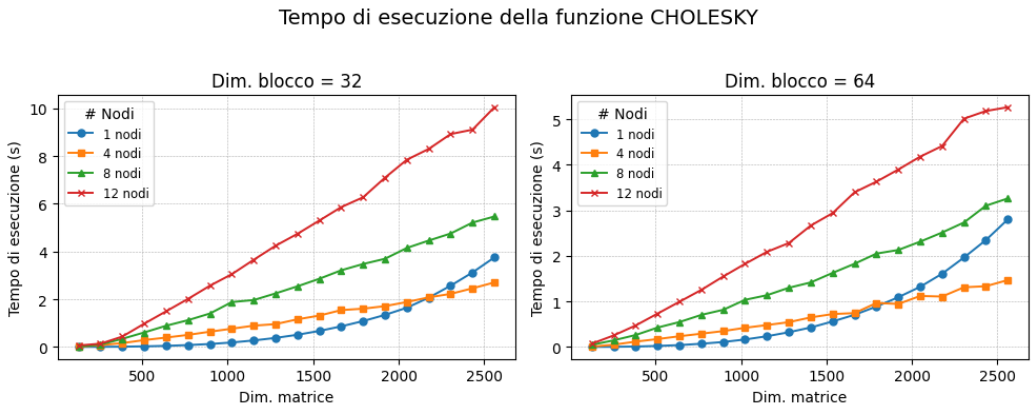

Figura 5.5: Tempi di esecuzione della funzione `Slac::cholesky` misurate su diverse dimensioni dei blocchi.

significativo. L'implementazione delle ottimizzazioni citate dovrebbe ridurre in maniera sostanziale le latenze introdotte da queste comunicazioni.

È tuttavia interessante notare che il tempo di esecuzione non cresce linearmente: nel caso a singolo nodo, il tempo per matrici molto grandi supera quello della configurazione a quattro nodi. Ciò è dovuto agli aggiornamenti delle sottomatrici eseguiti mediante moltiplicazioni matriciali, operazioni che beneficiano significativamente dell'uso di più nodi. Ne consegue che la decomposizione di Cholesky trae vantaggio dal parallelismo solo per matrici sufficientemente grandi; per dimensioni ridotte risulta più efficiente un approccio sequenziale.

Un altro aspetto rilevante è la forte dipendenza dei tempi dalla dimensione dei blocchi, che cresce quasi linearmente. Blocchi più grandi favoriscono le operazioni locali, e Eigen dispone di routine altamente ottimizzate per questo tipo di computazioni.

5.3 Risoluzione di sistemi simmetrici definiti positivi

5.3.1 Design e implementazione

La risoluzione di sistemi lineari simmetrici definiti positivi è una delle operazioni più comuni nel calcolo fisico e nell'analisi numerica: problemi di ottimizzazione, simulazioni meccaniche e modelli ellittici portano quasi sempre a sistemi SPD.

Per questo caso è inutile e inefficiente usare una fattorizzazione LU generale. La soluzione più rapida per risolvere il sistema $Ax = b$ consiste nell'eseguire una decomposizione di Cholesky:

$$A = LL^T$$

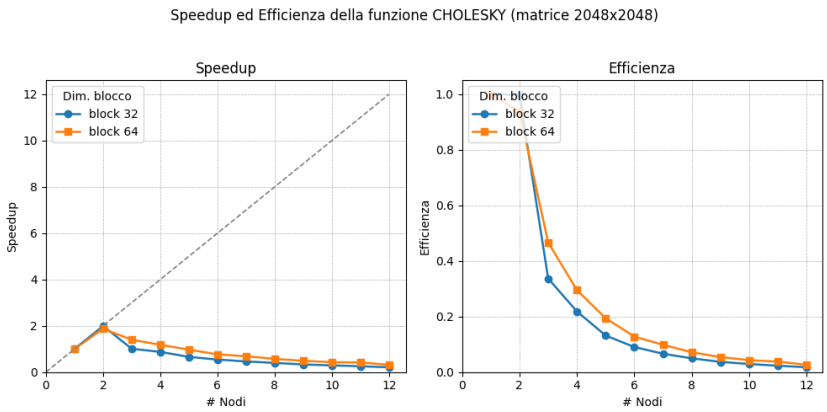

Figura 5.6: Speedup ed efficienza della funzione `Slac::cholesky` con dimensione della matrice fissa 2048×2048.

e poi risolvere due sistemi triangolari:

$$Ly = b$$
$$L^T x = y$$

L'algoritmo è estremamente semplice e si basa unicamente sulle primitive già fornite da SLAC.

1: **function** POSV(A, b)
2: $L \leftarrow$ CHOLESKY(A)
3: $y \leftarrow$ TRSM_LOWER(L, b)
4: $x \leftarrow$ TRSM_UPPER(L^T, y)
5: **return** x
6: **end function**

L'implementazione è quindi diretta: una volta ottenuta la fattorizzazione, la soluzione si ottiene tramite una forward e una backward substitution, entrambe già ottimizzate in forma distribuita. La funzione che esegue questo calcolo è `Slac::posv`.

5.3.2 Analisi delle prestazioni

Le tabelle 5.5 e 5.6 riportano i tempi di esecuzione variando il numero di nodi e la dimensione della matrice, utilizzando blocchi di lato 32 e 64. Sono inoltre mostrati i grafici dei tempi di esecuzione (Fig. 5.7) e, per matrici di dimensione fissata 2048×2048, speedup ed efficienza (Fig. 5.8).

Come prevedibile, il costo computazionale è dominato dalla decomposizione di Cholesky. Sebbene entrambe le fasi risolutive (Cholesky e le due risoluzioni triangolari) siano intrinsecamente iterative, il peso delle comunicazioni necessarie alla decomposizione supera di gran lunga quello della computazione locale. Di conseguenza, le due risoluzioni dei sistemi triangolari risultano molto più rapide ed efficienti.

Tabella 5.5: Tempi di esecuzione della funzione `Slac::posv` ($blocksize = 32$)

Dimensione matrice (px)	Tempo di esecuzione (s)			
	1 nodo	4 nodi	8 nodi	12 nodi
256×256	0.0046	0.0740	0.1004	0.1361
512×512	0.0284	0.2879	0.6086	0.9757
1024×1024	0.2028	0.7673	1.8830	3.0508
1536×1536	0.7117	1.3240	2.8672	5.3038
2048×2048	1.7239	1.9027	4.1696	7.8612
2560×2560	3.8783	2.7360	5.5013	10.0701

Tabella 5.6: Tempi di esecuzione della funzione `Slac::posv` ($blocksize = 64$)

Dimensione matrice (px)	Tempo di esecuzione (s)			
	1 nodo	4 nodi	8 nodi	12 nodi
256×256	0.0045	0.0509	0.1458	0.2541
512×512	0.0263	0.1742	0.4261	0.7331
1024×1024	0.1806	0.4234	1.0377	1.8275
1536×1536	0.6025	0.7311	1.6346	2.9449
2048×2048	1.3961	1.1366	2.3289	4.1910
2560×2560	2.9248	1.4960	3.2881	5.2875

Il modo più efficace per migliorare le prestazioni è aumentare la dimensione dei blocchi, favorendo il calcolo locale. Tuttavia, ciò comporta una distribuzione più irregolare dei dati e un potenziale sbilanciamento del carico di lavoro.

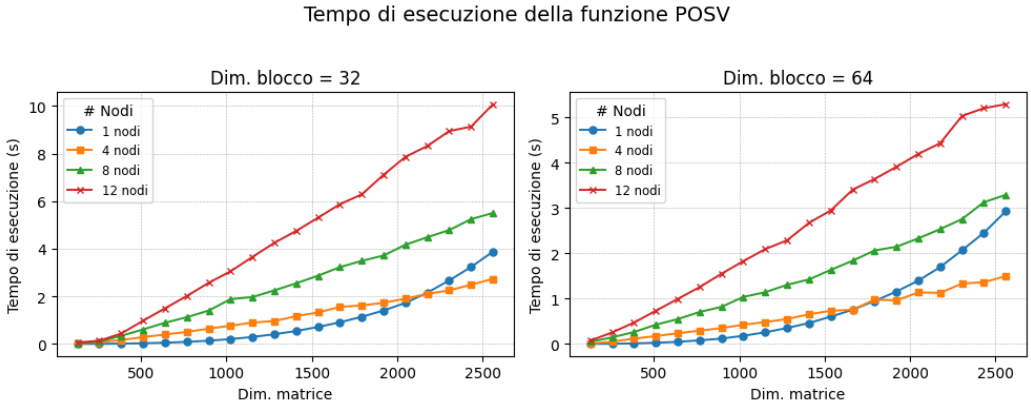

Figura 5.7: Tempi di esecuzione della funzione `Slac::posv` misurate su diverse dimensioni dei blocchi.

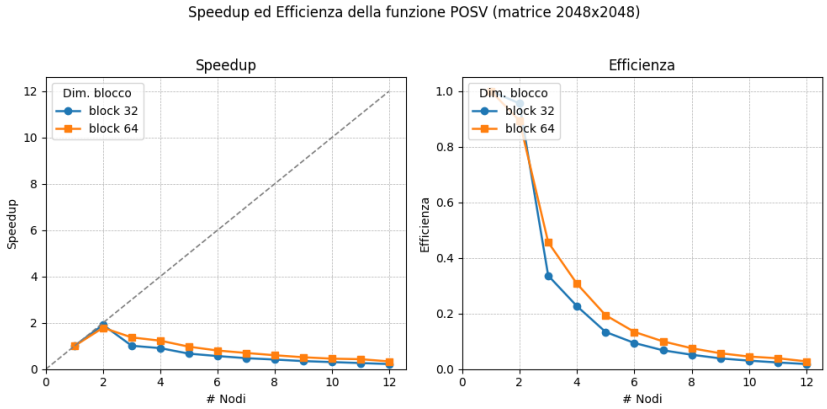

Figura 5.8: Speedup ed efficienza della funzione `Slac::posv` con dimensione della matrice fissa 2048×2048.

Capitolo 6

Implementazione del codice di inversione

L'intero flusso di lavoro dell'inversione sismico-gravimetrica è composto da una sequenza di eseguibili che, eseguiti nell'ordine corretto, producono il modello finale di densità del sottosuolo. Alcuni di questi moduli erano già disponibili in C++, mentre la parte relativa al calcolo dell'inversione (quella più numericamente complessa e direttamente coinvolta nell'uso dei dati di velocità e gravità) è stata oggetto della riscrittura presentata in questo capitolo.

La nuova implementazione sfrutta la libreria SLAC, che fornisce tutte le operazioni necessarie: gestione di matrici distribuite, calcolo di prodotti matriciali ottimizzati, aggiornamenti simmetrici e solutori per sistemi lineari simmetrici definiti positivi.

6.1 Inversione sismico-gravimetrica

Il codice implementa il calcolo descritto dall'equazione 1.2, che rappresenta lo schema dell'inversione lineare con regolarizzazione congiunta sismico-gravimetrica. I dati necessari provengono da file forniti dall'INGV: dimensioni della griglia, posizione delle stazioni osservative, misure di gravità e velocità, relazione densità-velocità, matrici di covarianza degli errori e la matrice delle derivate parziali $\mathbf{G}$, quest'ultima precomputata tramite l'algoritmo di Pohanka [20].

Una volta letti i dati, il calcolo dell'inversione viene eseguito interamente tramite funzioni SLAC. La scelta delle strutture dati è stata fatta considerando dimensioni e costi di comunicazione:

- i vettori e le matrici di covarianza sono memorizzati come `LocalVector`, poiché di dimensioni moderate e facilmente mantenibili in memoria locale;
- la matrice $\mathbf{G}$, invece, è enormemente più grande e viene gestita come `DistributedMatrix`;
- $\mathbf{G}$ è mantenuta direttamente in forma trasposta, poiché tutte le operazioni successive utilizzano $\mathbf{G}^T$: evitare una trasposizione esplicita riduce notevolmente il

tempo di preprocessamento, dato che la trasposizione distribuita è un'operazione costosa.

Le matrici di covarianza inverse C_{gg}^{-1} e C_{mm}^{-1} sono diagonali e vengono rappresentate come `LocalVector`. Nel caso di C_{gg}^{-1} è possibile scegliere, tramite l'opzione `-cgg-dim`, se interpretare il valore come diagonale vera e propria o come costante scalare (utile quando gli errori sono omogenei).

Le principali operazioni dell'espressione dell'inversione si traducono direttamente in primitive SLAC:

1. Il termine
$$\mathbf{G}^T C_{gg}^{-1} \mathbf{G}$$
è implementato come una chiamata a `Slac::syrk`, passando la diagonale C_{gg}^{-1} per applicare il peso ai blocchi corretti.

2. Il termine
$$\mathbf{G}^T C_{gg}^{-1} \Delta\mathbf{g}$$
si ottiene con `Slac::matmul` tra $\mathbf{G}^T$ e il vettore $C_{gg}^{-1} \odot \Delta\mathbf{g}$, calcolato tramite `Slac::cwise_mul`.

3. La regolarizzazione sismica
$$\alpha\, C_{mm}^{-1} \Delta\mathbf{v}$$
è implementata anch'essa tramite `Slac::cwise_mul`.

4. Le somme sono implementate con `Slac::add`.

5. Il sistema finale simmetrico definito positivo viene risolto tramite
$$\Delta\boldsymbol{\rho} = A^{-1} x,$$
usando direttamente `Slac::posv`.

Una volta ottenuto il modello finale di densità, questo viene salvato su file insieme alle differenze rispetto al modello di partenza e alle velocità ricostruite tramite la relazione densità-velocità.

6.2 Calcolo del campo di gravità

La fase successiva della riscrittura ha riguardato il calcolo diretto del campo gravitazionale e la costruzione della matrice $\mathbf{G}^T$. Questa parte ha richiesto particolare attenzione a causa della gestione dei dati di input.

Nel codice Fortran originale, ogni nodo calcolava autonomamente la propria porzione locale della matrice $\mathbf{G}$, evitando la necessità di memorizzarla globalmente su disco (impossibile a causa delle dimensioni). Ogni processo salvava su file solo il proprio blocco,

che il programma di inversione successivamente ricombinava assumendo la stessa griglia di processi, la stessa dimensione dei blocchi e la stessa politica di distribuzione.

Questo approccio rigido ha generato problemi nella nuova implementazione: leggere la matrice precomputata richiedeva che la configurazione del cluster fosse identica a quella dell'eseguibile Fortran. Bastava una variazione nel numero di processi o nella dimensione dei blocchi per rendere il file inutilizzabile.

Per superare definitivamente questo limite, si è deciso di implementare direttamente anche il calcolo del problema diretto, generando $\mathbf{G}^T$ come `DistributedMatrix` senza passare da file intermedi.

La modifica del codice C++ fornito, originalmente seriale e basato su array lineari, è stata semplice grazie agli oggetti `Element` di SLAC, che gestiscono automaticamente la localizzazione dei dati nel processo corretto. È stato sufficiente sostituire l'array con una `DistributedMatrix` e correggere gli indici per salvare direttamente la matrice trasposta: in questo modo si ottiene $\mathbf{G}^T$ senza dover effettuare la trasposizione distribuita, operazione molto costosa.

L'approccio risultante è più robusto, più efficiente, e soprattutto indipendente dalla configurazione del cluster utilizzata per generare i file di input. Questo completa la riscrittura dell'intero workflow in C++ e rende l'inversione sismico-gravimetrica pienamente portabile e scalabile.

6.3 Prestazioni

Le prove delle prestazioni sono state eseguite su *Ada*, il cluster di calcolo della sede di Bologna dell'INGV. Il sistema è composto da un nodo master, dodici nodi di computazione e due workstation HPC. La gestione delle risorse e dell'esecuzione è affidata a *Slurm*, che partiziona logicamente i nodi: per i test è stata utilizzata una partizione costituita da sei nodi *CentOS 7.9.2009*, ciascuno dotato di quattro processori *Intel Xeon Gold 6140* e di 7 GB di memoria RAM per core, connessi da una rete *Infiniband MT27800 100Gbs*.

L'INGV ha inoltre fornito i dati necessari per testare l'intera procedura di inversione: misure di gravità, densità iniziale, velocità e le relative matrici di covarianza. Si tratta di un dataset relativamente contenuto, rappresentativo di un problema di dimensioni ridotte: le osservazioni di gravità sono 6794, mentre quelle di densità e velocità sono 6760. Ne deriva una matrice $\mathbf{G} \in \mathbb{R}^{6794 \times 6760}$.

Nei grafici seguenti sono riportati i tempi di esecuzione (Fig. 6.1) e lo speedup (Fig. 6.2) ottenuti impiegando un numero variabile di processi del cluster e con diverse dimensioni di blocco.

Dall'analisi dei risultati emerge chiaramente che la scalabilità non è ancora ottimale. Questo è dovuto sia alla struttura intrinsecamente sequenziale di alcune operazioni del problema, sia al fatto che le implementazioni attuali dei risolutori non sfruttano ancora tutte le possibili ottimizzazioni. Nonostante ciò, il comportamento osservato fornisce una base solida su cui intervenire: le ottimizzazioni discusse nei capitoli precedenti dovrebbero portare a miglioramenti significativi in termini di tempi di esecuzione e utilizzo efficiente delle risorse del cluster.

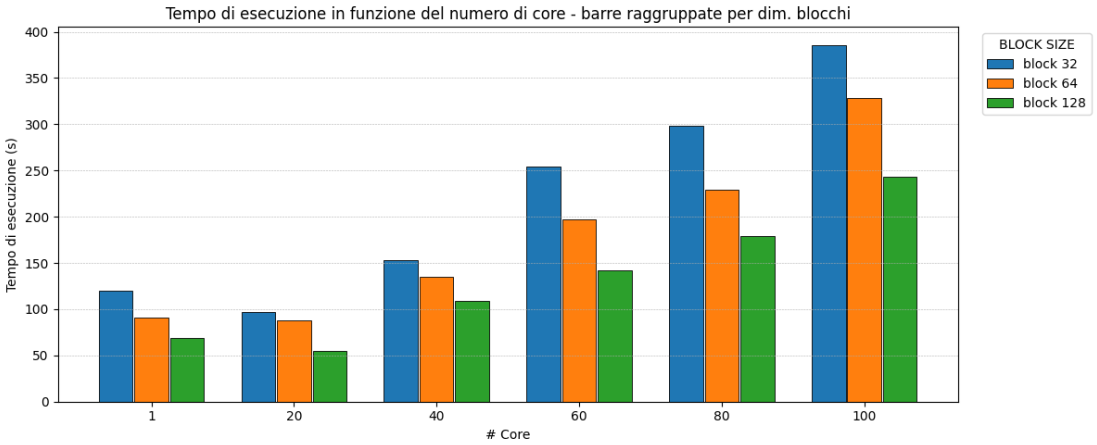

Figura 6.1: Tempi di esecuzione del programma di inversione misurati su un input di grandezza fissa 6794×6760.

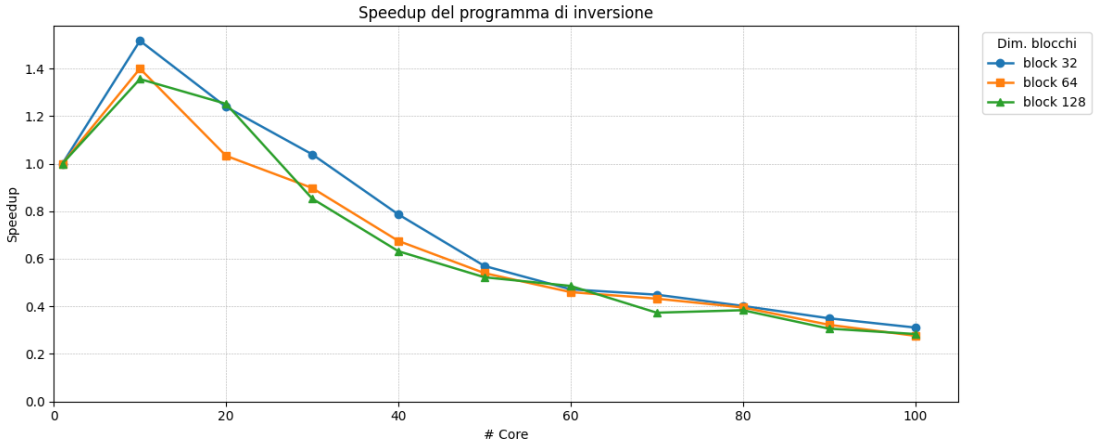

Figura 6.2: Speedup del programma di inversione misurati su un input di grandezza fissa 6794×6760.

Capitolo 7

Conclusioni

In questa tesi è stato presentato *SLAC*, un primo prototipo di libreria per l'algebra lineare distribuita progettata con l'obiettivo di essere semplice da utilizzare, facilmente estendibile e integrabile in differenti contesti applicativi. Pur trovandosi ancora in una fase iniziale di sviluppo, la libreria offre già un insieme coerente di funzionalità di base per operare su matrici distribuite, con un'interfaccia chiara e un modello di programmazione che punta a minimizzare la complessità per l'utente.

I risultati sperimentali mostrano come molte operazioni, in particolare quelle altamente parallelizzabili (come l'addizione di matrici), riescano a sfruttare efficacemente l'architettura distribuita, ottenendo speedup significativi e comportamenti in linea con le aspettative teoriche nonostante l'assenza di ottimizzazioni avanzate. Altre operazioni più complesse, come `matmul`, `syrk`, `trsm` e soprattutto `chol`, evidenziano invece i limiti attuali del sistema: le comunicazioni rappresentano spesso il principale collo di bottiglia, e l'implementazione dei risolutori non è ancora in grado di sfruttare pienamente il parallelismo disponibile.

È importante sottolineare che tali limiti non sono solo attesi, ma costituiscono un punto di partenza per sviluppi futuri. Le sezioni precedenti hanno evidenziato diverse ottimizzazioni potenzialmente decisive: l'introduzione di broadcast ad albero binario per ridurre le latenze, la possibilità di eseguire alcune fasi degli algoritmi in pipeline, e un migliore bilanciamento del carico. Queste ottimizzazioni, se implementate, permetteranno a SLAC di migliorare sensibilmente la scalabilità, soprattutto per algoritmi iterativi e a forte dipendenza comunicativa come la decomposizione di Cholesky.

Nonostante ciò, SLAC dimostra già oggi di essere una soluzione flessibile e, soprattutto, semplice da utilizzare e configurare rispetto a librerie consolidate come ScaLAPACK. Questo rappresenta uno degli obiettivi principali del progetto: offrire uno strumento accessibile per sperimentare, integrare e comprendere i metodi dell'algebra lineare distribuita senza dover affrontare la complessità delle grandi librerie HPC.

Il progetto ha inoltre ampi margini di estensione: l'aggiunta di nuove operatozioni, l'introduzione di algoritmi di fattorizzazione avanzati, l'esplorazione di strategie di scheduling e comunicazione alternative, l'integrazione con GPU e acceleratori. Tutti questi elementi rendono SLAC una base promettente per un sistema più completo ed efficiente.

Guida alla compilazione

Questa sezione fornisce una guida completa alla compilazione di programmi che utilizzano la libreria SLAC.

SLAC è una libreria *header-only*: ciò significa che non richiede alcuna fase di compilazione separata, ma è sufficiente includere gli header direttamente nel proprio codice sorgente. Gli header possono essere scaricati dal repository ufficiale [21]. La libreria è suddivisa in più file di intestazione, ciascuno dedicato a una specifica componente funzionale: `local_matrix.hpp`, `distributed_matrix.hpp`, `local_vector.hpp`, `distributed_vector.hpp`, `env.hpp`, `operators.hpp`. Ogni file definisce il relativo tipo di matrice o vettore, e può essere incluso singolarmente se si desidera utilizzare solo una parte delle funzionalità. Per caricare l'intero set di strumenti forniti da SLAC, è sufficiente includere l'header principale:

```
#include <Slac/core.hpp>
```

Dipendenze

SLAC richiede due componenti esterni già installati nel sistema:

- **Eigen** (versione C++): utilizzato per le operazioni locali su matrici e vettori;
- **MPI**: necessario per la gestione della distribuzione dei dati e delle comunicazioni tra processi.

È inoltre necessario un compilatore C++ che supporti pienamente lo standard **C++20**.

Compilazione

Poiché il programma risultante è un eseguibile MPI, la compilazione deve essere effettuata tramite `mpic++`, includendo la directory contenente gli header di SLAC. Un esempio di comando è il seguente:

```
mpic++ -std=c++20 -fopenmp -I<slac_directory> ...
```

Macro configurabili

Durante la compilazione è possibile definire alcune macro che controllano parametri interni della libreria e permettono di ottimizzare l'esecuzione:

- **SLAC_BLOCK_SIZE** Definisce la dimensione del lato dei blocchi utilizzati nella distribuzione a blocchi ciclici. La dimensione predefinita è 32, quindi i blocchi risultano essere matrici locali 32×32. Un valore maggiore può aumentare le prestazioni, ma a costo di un maggiore uso di memoria locale.
- **SLAC_MAX_TEMP_MATRIX_BLOCKS** Stabilisce il numero massimo di blocchi che possono comporre una matrice temporanea utilizzata per calcoli intermedi. Aumentare questo valore consente di ridurre le riallocazioni, migliorando la velocità al prezzo di un consumo di memoria superiore.
- **ENABLE_DEBUG** Abilita stampe diagnostiche per analizzare il comportamento della comunicazione tra processi. Questa opzione rallenta significativamente l'esecuzione ed è quindi consigliata solo per il debugging.

Esecuzione

Una volta compilato, il programma può essere eseguito tramite `mpirun`. La dimensione dei blocchi è fissata in fase di compilazione, mentre il numero di processi può essere scelto liberamente al momento dell'esecuzione:

```
mpirun -np <num_processi> ./programma
```

La creazione della griglia dei processi e la gestione delle comunicazioni avvengono automaticamente all'interno dell'ambiente di SLAC, permettendo di scalare il programma senza ulteriori modifiche al codice sorgente.

Bibliografia

[1] ScaLAPACK contributors / Netlib UTK. *Block Cyclic Data Distribution (ScaLAPACK / UTK Node 3)*. Netlib / UTK. Discussion of the two-dimensional block-cyclic decomposition used by ScaLAPACK. 1996. URL: https://www.netlib.org/utk/papers/factor/node3.html.

[2] E. Anderson et al. *LAPACK Users' Guide*. 3rd. Standard reference document for LAPACK; underlying routines rely on BLAS. SIAM, 1999. ISBN: 978-0-89871-397-5.

[3] Jaeyoung Choi, Jack J. Dongarra e David Walker. *PB-BLAS: A Set of Parallel Block Basic Linear Algebra Subprograms*. Rapp. tecn. Proceedings of the Scalable High Performance Computing Conference, May 1994. Foundational for PBLAS routines. Institute for Supercomputing Applications, University of Illinois at UrbanaChampaign, 1994. URL: https://www.netlib.org/scalapack/pblasqref.pdf.

[4] DBoyd13. *Linear Seismic Inversion Flow Chart*. Wikimedia Commons. Licensed under CC BY-SA 3.0. 2013. URL: https://commons.wikimedia.org/wiki/File:Linear_Seismic_Inversion_Flow_Chart.png.

[5] Jack J. Dongarra e R. Clint Whaley. *BLACS Basic Linear Algebra Communication Subprograms*. Netlib. Used as communication layer for ScaLAPACK; standard interface for process grid, contexts, array-based communication. 1997. URL: https://www.netlib.org/blacs/.

[6] Jack J. Dongarra et al. «A Set of Level 3 Basic Linear Algebra Subprograms». In: *ACM Transactions on Mathematical Software* 16.1 (1990). Defines the Level 3 BLAS, foundational for high-performance dense linear algebra., pp. 1–17. DOI: 10.1145/77626.77627.

[7] C. G. Farquharson, M. R. Ash e H. G. Miller. «Geologically constrained gravity inversion for the Voiseys Bay ovoid deposit». In: *Leading Edge* 27 (2008), pp. 64–69. DOI: 10.1190/1.2831681.

[8] C. G. Farquharson e D. W. Oldenburg. «Nonlinear inversion using general measures of data misfit and model structure». In: *Geophysical Journal International* 134 (1998), pp. 213–227. DOI: 10.1046/j.1365-246x.1998.00555.x.

[9] MPI Forum. *MPI Documents (standards and drafts)*. Access to MPI standard documents (MPI-2.x, MPI-3.x, etc.). 2024. URL: https://www.mpi-forum.org/docs/.

[10] Robert A. van de Geijn. *Notes on Cholesky Factorization*. FLAME notes, March 11, 2011. Good reference for blocked Cholesky / algorithmic variants. 2011. URL: https://www.cs.utexas.edu/~flame/Notes/NotesOnCholReal.pdf.

[11] Robert A. van de Geijn e Jerrell Watts. *SUMMA: Scalable Universal Matrix Multiplication Algorithm*. Rapp. tecn. LAWN 96 / LAPACK Working Note. Paper and MPI implementation details; introduces SUMMA algorithm. Department of Computer Sciences, The University of Texas at Austin / Scalable Concurrent Programming Laboratory, Caltech, 1996. URL: https://www.netlib.org/lapack/lawnspdf/lawn96.pdf.

[12] Gaël Guennebaud, Benot Jacob et al. *Eigen A C++ template library for linear algebra (Eigen v3)*. Header-only C++ linear algebra library; see BibTeX/citation page on the Eigen site. 2010. URL: https://libeigen.gitlab.io/.

[13] Michael T. Heath e Edgar Solomonik. *Parallel Numerical Algorithms Chapter 3: Dense Linear Systems (Section 3.3: Triangular Systems)*. Course slides (CS 554 / CSE 512) covering triangular solves, wavefront algorithms and TRSM. 2021. URL: https://relate.cs.illinois.edu/course/cs554-f21/f/slides/slides_07.pdf.

[14] Istituto Nazionale di Geofisica e Vulcanologia (INGV). *INGV Sito ufficiale*. Sito ufficiale dell'ente fornente i dati e utilizzatore del codice; informazioni istituzionali e dataset. 2025. URL: https://www.ingv.it/.

[15] L. Lines, A. K. Schultz e S. Treitel. «Cooperative inversion of geophysical data». In: — (1988).

[16] Phil Nash e il team di Catch2. *Catch2 Modern C++ Testing Framework*. Framework header-only per unit testing in C++ (TDD, BDD, micro-benchmarking). 2025. URL: https://catch2.org/.

[17] D. W. Oldenburg e D. A. Pratt. «Geophysical Inversion for Mineral Exploration: a Decade of Progress in Theory and Practice». In: 2007.

[18] Open MPI Community. *Open MPI Open Source High Performance Computing*. Sito ufficiale del progetto Open MPI (homepage, documentazione e download). 2024. URL: https://www.open-mpi.org/.

[19] N. D. Phillips. «Geophysical inversion in an integrated exploration program examples from the San Nicolás deposit». Tesi di laurea mag. Vancouver, Canada: University of British Columbia, 2001.

[20] V. Pohánka. *Optimum expression for computation of the gravity field of a homogeneous polyhedral body*. Public technical report / article on gravity calculations for polyhedral bodies (Pohánka). 1997. URL: http://gpi.savba.sk/GPIweb/ogg/pohanka/Pro46.pdf.

[21] M. Ravaioli. *SLAC Scalable Linear Algebra for C++*. Libreria header-only in C++ per calcolo parallelo distribuito su problemi di algebra lineare. 2025. URL: https://github.com/RavaMichi/scalable-linear-algebra-for-cpp.

[22] A. Tarantola. *Inverse Problem Theory*. Amsterdam: Elsevier, 1987.

[23] ScaLAPACK Team. *ScaLAPACK Scalable Linear Algebra PACKage (home)*. Netlib. Version 1.8 home page; includes users' guide, install guide and references. 2007. URL: https://www.netlib.org/scalapack/.

[24] R. Tondi et al. «Parallel, "large", dense matrix problems: Application to 3D sequential integrated inversion of seismological and gravity data». In: *Computers & Geosciences* 48 (2012). DOI: 10.1016/j.cageo.2012.05.026.

[25] K. Vozoff e D. Jupp. «Joint Inversion of Geophysical Data». In: *Geophysical Journal International* (2007). DOI: 10.1111/J.1365-246X.1975.TB06462.X.

[26] N. C. Williams. «Applying UBC-GIF potential field inversions in greenfields or brownfields exploration». In: *Australian Earth Sciences Convention, ASEG/GSA, Abstracts*. 2006.

Ringraziamenti

Desidero esprimere la mia sincera gratitudine a tutte le persone che hanno contribuito, direttamente o indirettamente, alla realizzazione di questo lavoro.

Ringrazio innanzitutto il mio relatore, Moreno Marzolla, per la disponibilità, i consigli e il sostegno continuo durante tutto il percorso.

Un ringraziamento speciale va a Maria Rosaria Tondi e Giampaolo Zerbinato, personale dell'INGV persso la sede di Bologna, per avermi seguito lungo tutto il percorso di tirocinio, e per aver messo a disposizione le risorse di calcolo e i dati necessari per la parte applicativa della tesi.

Un grazie va anche ai miei colleghi e amici, che hanno condiviso momenti di confronto sempre preziosi.

Infine, un ringraziamento affettuoso alla mia famiglia, per il sostegno costante, la pazienza e l'incoraggiamento che non mi hanno mai fatto mancare.

A tutti voi, grazie.