

Università degli Studi di Salerno

**DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE ED
ELETTRICA E MATEMATICA APPLICATA**

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA



HIGH PERFORMANCE COMPUTING IMPLEMENTAZIONE E OTTIMIZZAZIONE DELL'ALGORITMO VF2++ PER GRAPH ISOMORPHISM

Studente

Giuseppe Squitieri - 0622702339 - g.squitieri8@studenti.unisa.it

Docente

Prof. Francesco Moscato – fmoscato@unisa.it
Anno Accademico 2024/2025

Questo documento è distribuito con licenza
Creative Commons Attribution-NonCommercial-ShareAlike 4.0
International (CC BY-NC-SA 4.0)
<https://creativecommons.org/licenses/by-nc-sa/4.0/>



Indice

1 Introduzione al Problema	5
1.1 Graph Isomorphism Problem	5
1.1.1 Condizioni di isomorfismo	5
1.1.2 Definizione finale	6
1.2 VF2	7
1.2.1 Principi fondamentali	7
1.2.2 Strutture dati principali	7
1.2.3 Definizione di $P(s)$	7
1.2.4 Regole di fattibilità	7
1.2.5 Algoritmo principale	8
1.2.6 Complessità computazionale	8
1.3 VF2++	9
1.3.1 Motivazioni per VF2++	9
1.3.2 Innovazioni principali	9
1.3.3 Algoritmo principale	10
1.3.4 Vantaggi rispetto a VF2	10
1.3.5 Complessità e prestazioni	11
2 Componenti	12
2.1 Specifiche Hardware	12
2.1.1 Sistema Base	12
2.1.2 Processore	12
2.1.3 Memoria e Storage	12
2.1.4 Grafica	12
2.2 Ambiente Software	13
2.2.1 Sistema Operativo	13
2.2.2 Compilatore e Toolchain	13
2.2.3 Ottimizzazioni del Compilatore	13
2.3 Software e Misure	13
2.3.1 Linguaggi e runtime	13
2.3.2 Metodologia di misura	14
3 Implementazione dell'Algoritmo	15
3.1 Struttura del progetto	15
3.2 Rappresentazione dei grafi	15
3.3 Calcolo dell'ordine dei nodi	15
3.4 Generazione dei candidati	16
3.5 Gestione dello stack	16
3.6 Algoritmo principale	16
3.7 Funzione principale	16
4 Implementazione MPI VF2++	17
4.1 Ordinamento dei nodi	17
4.2 La funzione propose_candidates	18
4.3 Parallelizzazione MPI effettiva di VF2++	19

4.4	Implementazione alternativa: Work Stealing	23
4.5	Considerazioni sulla scalabilità teorica	24
5	Risultati	25
5.1	Metodologia sperimentale e raccolta dati	25
5.2	Test: VF2++ (sequenziale) vs VF2 (NetworkX)	25
5.3	Conclusioni — VF2++ (sequenziale) vs VF2	30
5.4	Test VF2++ vs VF2++ con MPI	31
5.4.1	2 processi	31
5.4.2	Tabella riassuntiva media Speedup 2 processi	35
5.4.3	4 processi	36
5.5	Tabella Riassuntiva media Speedup 4 processi	41
5.6	Conclusioni — VF2++ (sequenziale) vs VF2++ con MPI	41
6	Conclusioni e considerazioni finali	44

1 Introduzione al Problema

1.1 Graph Isomorphism Problem

Il **Graph Isomorphism Problem** rappresenta uno dei problemi computazionali più studiati e complessi dell'informatica teorica. Dal punto di vista della complessità computazionale, il problema dell'isomorfismo di grafi riveste un ruolo particolare: esso appartiene alla classe NP, ma non è noto se sia risolvibile in tempo polinomiale né se sia NP-completo. Si tratta quindi di uno dei pochi problemi "intermedi", che sfugge ancora a una classificazione definitiva. Questa incertezza teorica contribuisce a renderlo un tema di grande interesse nella ricerca informatica e matematica.

Per spiegare tale problema, prendiamo come esempio due grafi:

- G con struttura $\{a, b, c\}$
- H con struttura $\{x, y, z\}$

Come possiamo osservare, G e H non sono lo stesso grafo, quindi non possiamo scrivere $G = H$. Tuttavia, se rispettano determinate regole, potrebbero avere la stessa struttura. In tal caso si dicono **isomorfi** e si scrive:

$$G \simeq H \quad \text{o} \quad H \simeq G$$

1.1.1 Condizioni di isomorfismo

Affinché due grafi siano isomorfi, devono rispettare le seguenti condizioni:

1. **Numero di vertici e archi** Devono avere lo stesso numero di vertici e lo stesso numero di archi.
2. **Corrispondenza tra vertici** Se G è isomorfo ad H , allora è possibile stabilire una corrispondenza biunivoca tra i vertici di G e quelli di H .
Esempio: se in G il vertice b ha due vicini e lo associamo al vertice y di H , allora anche y deve avere due vicini.

Questa corrispondenza si indica con una funzione:

$$\varphi : V(G) \rightarrow V(H)$$

Ogni vertice di G deve avere un corrispondente in H , e non è possibile che due vertici distinti di G abbiano la stessa immagine in H .

Possiamo rappresentarla anche in forma di "matrice di associazione":

$$\varphi : \begin{cases} a \mapsto x \\ b \mapsto y \\ c \mapsto z \end{cases}$$

3. Proprietà di biettività La funzione φ deve essere:

- *iniettiva*: vertici diversi di G hanno immagini diverse in H ;
- *suriettiva*: ogni vertice di H deve avere almeno una preimmagine in G .

In questo modo φ risulta **biettiva**, cioè una corrispondenza uno-a-uno tra i vertici.

4. Preservazione delle adiacenze La corrispondenza deve preservare la struttura degli archi. In particolare, per ogni coppia di vertici $u, v \in V(G)$:

$$uv \in E(G) \iff \varphi(u)\varphi(v) \in E(H)$$

Ciò significa che:

- se due vertici in G sono adiacenti, i corrispondenti in H devono esserlo;
- se due vertici in G non sono adiacenti, i corrispondenti in H non devono esserlo.

1.1.2 Definizione finale

Due grafi G e H si dicono **isomorfi** se esiste una biezione:

$$\varphi : V(G) \rightarrow V(H)$$

tale che:

$$uv \in E(G) \iff \varphi(u)\varphi(v) \in E(H)$$

In tal caso si scrive:

$$G \simeq H$$

È importante sottolineare che oltre al problema dell'isomorfismo classico, la letteratura affronta anche varianti più generali, come il *subgraph isomorphism* e l'*induced subgraph isomorphism*. Nel primo caso si cerca di verificare se un grafo sia contenuto come sottografo in un altro, mentre nel secondo si richiede che le adiacenze siano preservate esattamente come nell'isomorfismo classico, ma limitatamente a un sottografo. Entrambi i problemi sono noti per essere NP-completi, e rappresentano scenari in cui algoritmi efficienti come VF2 e VF2++ trovano applicazione concreta.

1.2 VF2

L'algoritmo VF2 (Vento-Foggia 2), sviluppato da Cordella et al. nel 2004, rappresenta uno dei metodi più efficienti per risolvere il *Graph Isomorphism Problem*. VF2 è un algoritmo di backtracking che costruisce incrementalmente un mapping tra i nodi di due grafi.

1.2.1 Principi fondamentali

VF2 si basa su tre concetti chiave:

- **State Space Representation:** ogni stato rappresenta un mapping parziale tra i nodi dei due grafi.
- **Feasibility Rules:** regole che permettono di eliminare rami non promettenti dell'albero di ricerca.
- **Incremental Matching:** il mapping viene esteso nodo per nodo, con ritorno indietro in caso di incompatibilità.

1.2.2 Strutture dati principali

VF2 mantiene strutture per tracciare lo stato della ricerca:

- $M(s)$: mapping parziale corrente.
- $T_1^{out}(s), T_2^{out}(s)$: successori non ancora mappati di nodi già nel mapping.
- $T_1^{in}(s), T_2^{in}(s)$: predecessori non ancora mappati di nodi già nel mapping.

1.2.3 Definizione di $P(s)$

L'insieme delle coppie candidate $P(s)$ è definito come:

- se $T_1^{out}(s)$ e $T_2^{out}(s)$ non sono vuoti: $P(s) = T_1^{out}(s) \times T_2^{out}(s)$;
- altrimenti, se $T_1^{in}(s)$ e $T_2^{in}(s)$ non sono vuoti: $P(s) = T_1^{in}(s) \times T_2^{in}(s)$;
- altrimenti: $P(s) = (N_1 - M_1(s)) \times (N_2 - M_2(s))$.

1.2.4 Regole di fattibilità

La funzione di fattibilità $F(s, n, m)$ verifica che:

1. siano rispettati eventuali vincoli di attributi sui nodi/archi;
2. tutte le adiacenze già mappate siano consistenti tra n e m ;
3. le cardinalità dei terminal set corrispondano.

1.2.5 Algoritmo principale

```
PROCEDURE Match( $s$ )
    INPUT: an intermediate state  $s$ ; the initial state  $s_0$  has  $M(s_0)=\emptyset$ 
    OUTPUT: the mappings between the two graphs

    IF  $M(s)$  covers all the nodes of  $G_2$  THEN
        OUTPUT  $M(s)$ 
    ELSE
        Compute the set  $P(s)$  of the pairs candidate for inclusion in  $M(s)$ 
        FOREACH  $(n, m) \in P(s)$ 
            IF  $F(s, n, m)$  THEN
                Compute the state  $s'$  obtained by adding  $(n, m)$  to  $M(s)$ 
                CALL Match( $s'$ )
            END IF
        END FOREACH
        Restore data structures
    END IF
END PROCEDURE
```

Figura 1: VF2 Alghoritm

1.2.6 Complessità computazionale

- **Caso peggiore:** $O(N! \times N)$, dove N è il numero di nodi.
- **Caso medio:** molto migliore grazie alle regole di potatura.

La complessità spaziale è $O(N)$ grazie alle strutture dati condivise.

1.3 VF2++

VF2++ rappresenta l’evoluzione dell’algoritmo VF2, sviluppata da Jüttner e Madarasi nel 2018, con l’obiettivo di superare alcune limitazioni strutturali della versione precedente e garantire prestazioni notevolmente superiori.

1.3.1 Motivazioni per VF2++

Nonostante l’efficacia di VF2, l’algoritmo presentava tre limiti principali. In primo luogo, l’ordine con cui i nodi venivano esplorati non era ottimale e spesso portava ad un’espansione non necessaria dello spazio di ricerca. In secondo luogo, le regole di potatura applicate risultavano poco incisive, lasciando sopravvivere molti rami dell’albero di ricerca che non avrebbero condotto a soluzioni valide. Infine, la gestione della memoria non era del tutto efficiente, soprattutto nei casi in cui i grafi coinvolti avevano dimensioni considerevoli.

1.3.2 Innovazioni principali

Per risolvere queste criticità, VF2++ introduce tre innovazioni fondamentali. La prima è l’adozione di un *node ordering* strategico: i nodi vengono visitati seguendo un ordine calcolato in modo intelligente, basato su più criteri combinati. L’algoritmo tiene conto del grado dei nodi, della rarità delle etichette presenti e della struttura locale, ossia del numero di vicini già mappati. Questo approccio permette di anticipare i nodi più “critici” e ridurre drasticamente lo spazio di ricerca.

La seconda innovazione riguarda le cosiddette *cutting rules avanzate*. A differenza delle regole di potatura tradizionali, VF2++ utilizza condizioni più potenti, che analizzano i vicini etichettati e confrontano le cardinalità nei terminal set dei due grafi. In questo modo è possibile escludere in anticipo numerose corrispondenze non valide, evitando esplorazioni infruttuose.

Infine, VF2++ abbandona l’approccio ricorsivo tipico di VF2, sostituendolo con una gestione esplicita dello stack per il *backtracking*. Questa scelta consente un controllo più diretto sulla profondità della ricerca e sulla selezione dei candidati, oltre a rendere più efficiente l’aggiornamento delle strutture dati. L’implementazione non ricorsiva riduce l’overhead computazionale del backtracking e permette di sfruttare meglio le risorse di memoria.

1.3.3 Algoritmo principale

L'implementazione del core algorithm segue questa logica:

Algorithm 1 VF2++ Core Algorithm

Require: Grafi G_1 e G_2

Ensure: **true** se isomorfi, **false** altrimenti

```
1: Inizializza strutture dati ( $T_2^\sim$ , stack, mapping)
2: Calcola ordinamento ottimale dei nodi di  $G_1$ 
3: Trova candidati per il primo nodo e push nello stack
4: while stack non vuoto do
5:    $current\_node \leftarrow$  top dello stack
6:    $found \leftarrow$  false
7:   for ogni candidato non testato do
8:     if candidato è valido per mapping then
9:       if tutti i nodi sono mappati then
10:        return true
11:       end if
12:       Aggiorna mapping e  $T_2^\sim$ 
13:       Trova candidati per il prossimo nodo
14:       Push nello stack
15:        $found \leftarrow$  true
16:       break
17:     end if
18:   end for
19:   if not found then
20:     Pop dallo stack
21:     Ripristina stato precedente
22:   end if
23: end while
24: return false
```

1.3.4 Vantaggi rispetto a VF2

Rispetto al suo predecessore, VF2++ introduce miglioramenti significativi sotto diversi aspetti. Dal punto di vista delle **prestazioni**, l'algoritmo può risultare fino a dieci o addirittura cento volte più veloce su grafi di grandi dimensioni, grazie alle strategie di ordinamento e alle regole di potatura avanzate. Anche la **scalabilità** ne beneficia: mentre VF2 tende a degradare rapidamente all'aumentare del numero di nodi, VF2++ è in grado di gestire senza difficoltà grafi contenenti migliaia di vertici. Inoltre, l'algoritmo si dimostra più **robusto**, garantendo un comportamento costante e prevedibile sia su grafi sparsi sia su grafi densi. Un ulteriore vantaggio è rappresentato dall'**efficienza nella gestione della memoria**: l'implementazione non-ricorsiva consente un uso più razionale delle risorse, evitando gli sprechi tipici dell'approccio classico.

1.3.5 Complessità e prestazioni

Dal punto di vista teorico, la complessità nel caso peggiore rimane invariata, ossia $O(N! \times N)$, come per VF2. Tuttavia, nella pratica, le ottimizzazioni introdotte consentono di ridurre drasticamente i tempi di esecuzione. I test empirici mostrano chiaramente questi benefici: nei grafi **sparsi** il guadagno varia da 5 a 20 volte, nei grafi **densi** può raggiungere un fattore di 10–50, mentre su grafi molto grandi (con più di mille nodi) lo speedup arriva fino a 100 volte rispetto a VF2.

In sintesi, VF2++ rappresenta oggi lo *stato dell'arte* per il problema dell'isomorfismo di grafi, riuscendo a coniugare l'eleganza della formulazione teorica con un livello di efficienza pratica difficilmente eguagliabile.

2 Componenti

2.1 Specifiche Hardware

Il sistema utilizzato per lo sviluppo e il testing presenta le seguenti caratteristiche tecniche:

2.1.1 Sistema Base

- **Modello:** Acer Aspire A315-55G-7045
- **Architettura:** x86_64
- **Form Factor:** Laptop ultrabook

2.1.2 Processore

Tabella 1: Specifiche dettagliate del processore

Parametro	Valore
CPU	Intel® Core™ i7-10510U
Frequenza Base	1.80 GHz
Frequenza Massima	4.90 GHz (Turbo Boost)
Core Fisici	4
Thread Logici	8 (Hyperthreading)
Cache L3	8 MB
Architettura	Comet Lake (14nm)
TDP	15W (Ultra Low Power)
Set di Istruzioni	SSE4.1, SSE4.2, AVX2

2.1.3 Memoria e Storage

- **RAM Totale:** 16 GB DDR4
- **Configurazione:** Dual Channel
- **Banda Teorica:** ~25.6 GB/s
- **Latenza:** CL22 (tipica per DDR4-2666)

2.1.4 Grafica

- **GPU Integrata:** Intel® UHD Graphics (CML GT2)
- **Driver:** Mesa Intel® UHD Graphics
- **Identificativo:** NV138
- **Execution Units:** 24 EU

2.2 Ambiente Software

2.2.1 Sistema Operativo

- **Distribuzione:** Ubuntu 22.04.5 LTS (Jammy Jellyfish)
- **Kernel:** Linux 5.15+ (64-bit)
- **Desktop Environment:** GNOME 42+
- **Package Manager:** APT

2.2.2 Compilatore e Toolchain

Tabella 2: Toolchain di sviluppo utilizzata

Componente	Versione/Dettagli
Compilatore Principale	GCC (GNU Compiler Collection)
Standard C	C99/C11
Librerie Standard	glibc 2.35+
Librerie Matematiche	libm
Build System	Make con Makefile personalizzato
Debugger	GDB
Profiler	gprof, Valgrind

2.2.3 Ottimizzazioni del Compilatore

Il progetto supporta multiple configurazioni di ottimizzazione:

- 00 Nessuna ottimizzazione (modalità debug)
- 01 Ottimizzazioni base, tempo di compilazione ridotto
- 02 Ottimizzazioni standard (configurazione default)
- 03 Ottimizzazioni aggressive, massime prestazioni

2.3 Software e Misure

2.3.1 Linguaggi e runtime

- **Linguaggio:** C (standard C99/C11).
- **Compilatore:** GCC (toolchain GNU) con ottimizzazioni -00, -01, -02, -03.
- **Parallelismo:** MPI conforme allo standard, utilizzata per la versione parallela di VF2++.
- **Scripting:** Python per l'automazione dei test (lancio batch, raccolta log) e la generazione di grafici e tabelle a partire dai CSV.

2.3.2 Metodologia di misura

- **Isolamento del carico:** tutti i test sono stati eseguiti con il computer *dedicato* all'esperimento (nessun altro carico utente attivo) per ridurre rumore da schedulazione, I/O e interferenze.
- **Riproducibilità:** stesso ambiente software, stesse opzioni di compilazione e identica pipeline di esecuzione tra run sequenziali e MPI.
- **Raccolta dati:** tempi misurati dagli eseguibili e salvati in .csv; i grafici (tempi e speedup) sono stati generati con script Python a partire dagli stessi file.

3 Implementazione dell'Algoritmo

3.1 Struttura del progetto

Il progetto è stato realizzato in linguaggio C, con una chiara separazione tra interfacce (`.h`) e implementazioni (`.c`). La cartella `include` contiene tutti i file header, che definiscono le strutture dati e le funzioni principali, mentre la cartella `src` raccoglie le implementazioni operative.

L'architettura del codice segue un'organizzazione modulare:

- **graph** – definisce e gestisce la struttura dei grafi, i nodi, gli archi e le operazioni di base.
- **vertex_ordering** – implementa l'algoritmo per il calcolo dell'ordine dei nodi, elemento chiave di VF2++.
- **propose_candidates** – calcola l'insieme dei nodi candidati in G_2 per l'estensione del mapping.
- **stack** – gestisce uno stack esplicito per memorizzare gli stati durante la ricerca, evitando la ricorsione.
- **vf2pp** – contiene il cuore dell'algoritmo VF2++, coordinando la generazione dei candidati, l'applicazione delle regole di consistenza e delle cutting rules.
- **main** – costituisce il punto di ingresso del programma, gestisce l'input e avvia la procedura di matching.

3.2 Rappresentazione dei grafi

I grafi sono rappresentati tramite strutture dati dedicate definite in `graph.h`. Ogni grafo è descritto da un insieme di nodi e archi, con funzioni che permettono di creare, modificare e interrogare la struttura. Questa astrazione rende l'algoritmo indipendente dalla specifica modalità di input, facilitando l'estensione a diversi formati di dati.

3.3 Calcolo dell'ordine dei nodi

Il file `vertex_ordering.c` implementa la strategia di ordinamento introdotta in VF2++. L'algoritmo calcola un matching order ottimizzato, basato su tre criteri:

1. il grado dei nodi,
2. la rarità delle etichette,
3. la struttura locale, cioè il numero di vicini già mappati.

Questa procedura, richiamata all'inizio della ricerca, consente di ridurre lo spazio esplorato e migliorare le prestazioni complessive.

3.4 Generazione dei candidati

Il modulo `propose_candidates.c` è responsabile della selezione dei nodi di G_2 che possono essere accoppiati al nodo corrente di G_1 . La scelta si basa sulle regole di consistenza: se esistono vicini già mappati, i candidati sono limitati ai corrispondenti vicini nel secondo grafo; altrimenti, la ricerca si estende all'insieme dei nodi non ancora utilizzati.

3.5 Gestione dello stack

Una delle principali innovazioni di VF2++ è l'adozione di una gestione non ricorsiva. Il file `stack.c` implementa uno stack esplicito che memorizza gli stati della ricerca (mapping parziali, profondità corrente, nodi da esplorare). Questa scelta consente un migliore controllo sul flusso dell'algoritmo e una gestione della memoria più efficiente, evitando l'overhead tipico delle chiamate ricorsive.

3.6 Algoritmo principale

Il cuore del sistema è racchiuso in `vf2pp.c`. Qui viene gestito il ciclo principale dell'algoritmo:

1. Inizializzazione delle strutture dati e caricamento del matching order.
2. Iterazione sugli stati tramite lo stack esplicito.
3. Per ogni nodo u di G_1 , determinazione dei candidati v in G_2 .
4. Applicazione delle regole di consistenza e delle cutting rules per escludere rapidamente le corrispondenze non valide.
5. Estensione del mapping e push del nuovo stato nello stack.
6. Identificazione e output degli isomorfismi trovati.

3.7 Funzione principale

Infine, il file `main.c` rappresenta il punto di ingresso del programma. Esso si occupa della lettura dei dati di input (grafo da confrontare), dell'inizializzazione delle strutture e dell'avvio della procedura di matching tramite le funzioni di `vf2pp.c`.

4 Implementazione MPI VF2++

In questa sezione verrà spiegato il ragionamento alla base dell'implementazione MPI, illustrando le motivazioni delle scelte effettuate, le strade percorse e quelle non implementate, e fornendo una panoramica di tutto ciò che si trova dietro lo sviluppo dell'algoritmo in parallelo.

Per farlo, si analizzerà l'algoritmo VF2++ passo per passo, a partire dalla funzione `vertex_ordering`. Tale funzione, come spiegato nel capitolo precedente, ha il compito di stabilire l'ordine con cui i nodi del grafo vengono considerati durante il processo di isomorfismo.

4.1 Ordinamento dei nodi

La funzione `vertex_ordering` costruisce la sequenza dei nodi di G_1 secondo la seguente logica:

1. sceglie come primo nodo quello con **grado massimo** (cioè con il maggior numero di vicini);
2. esegue una **BFS** a partire da tale nodo;
3. all'interno di ogni livello della BFS ordina i nodi per **grado decrescente**;
4. se restano nodi non visitati (ad esempio in caso di grafo disconnesso), riparte dal nodo con grado massimo tra quelli rimanenti.

L'ordine finale privilegia quindi nodi con **molti vicini e vicini tra loro**, riducendo lo spazio di ricerca dell'algoritmo VF2++.

Perché non è parallelizzabile facilmente

La funzione non si presta bene al parallelismo perché:

- mantiene uno **stato incrementale** (`ordered[]` e `num_nodes_ordered`) che cambia ad ogni inserimento;
- ogni scelta dipende dalle precedenti: per determinare il prossimo nodo da ordinare bisogna sapere quali nodi sono già stati inseriti e in quale ordine la BFS li ha prodotti;
- vi è quindi un **riferimento diretto e sequenziale** tra i livelli della BFS e l'array `ordered`, che impedisce di calcolare i risultati in parallelo senza rischiare inconsistenze o duplicazioni.

Si tratta dunque di una procedura **strettamente sequenziale**, in cui ogni passo dipende dal precedente.

Perché non è stata parallelizzata la BFS

In teoria la sola BFS interna sarebbe parallelizzabile, poiché i nodi di uno stesso livello possono essere esplorati contemporaneamente. Tuttavia, in questo caso:

- la BFS viene eseguita solo su **porzioni molto piccole del grafo**, limitata ai livelli vicini del nodo scelto;
- essa serve esclusivamente a costruire l'ordine iniziale, quindi il **carico computazionale è ridotto** rispetto al resto dell'algoritmo VF2++;
- introdurre thread o comunicazioni avrebbe comportato un **overhead di sincronizzazione** superiore al guadagno reale;

Per queste ragioni si è scelto di mantenere la BFS sequenziale: **più semplice, leggera e priva di overhead inutile**.

4.2 La funzione propose_candidates

La funzione `propose_candidates` si occupa di determinare, dato un nodo u di G_1 , quali nodi di G_2 possono essere suoi candidati validi. L'operazione coinvolge soltanto i **vicini già mappati** di u e un insieme ristretto di nodi in G_2 .

Dove sarebbe parallelizzabile

In teoria alcune parti della funzione sono parallelizzabili:

- nei **CASI 1 e 2**, i cicli che filtrano i candidati in base a condizioni indipendenti (grado, mapping, disponibilità) potrebbero essere eseguiti in parallelo, assegnando ogni nodo candidato a un thread diverso;
- anche nel **CASO 3**, i controlli su ciascun nodo candidato rispetto ai vicini mappati possono essere suddivisi tra più thread, dato che ogni verifica è indipendente dalle altre.

Quindi, a livello teorico, la funzione è **parallelizzabile**, poiché i controlli sui candidati non hanno dipendenze forti tra loro.

Perché non è stato fatto

Tuttavia, in pratica:

- il **numero di nodi da analizzare è molto ridotto** (solo i vicini di un nodo, non l'intero grafo);
- il **costo computazionale è trascurabile** rispetto al backtracking complessivo dell'algoritmo VF2++;
- introdurre parallelismo richiederebbe sincronizzazione sull'array dei candidati e gestione della memoria condivisa, generando un **overhead maggiore del guadagno reale**.

Per questo motivo la funzione è stata mantenuta sequenziale: pur essendo parallelizzabile in teoria, la quantità di lavoro è così piccola da rendere la parallelizzazione **inutile e controproducente**.

4.3 Parallelizzazione MPI effettiva di VF2++

L'idea alla base della parallelizzazione di VF2++ con MPI è quella di ridurre il tempo necessario ad esplorare lo spazio di ricerca distribuendo il lavoro tra più processi. In particolare, è stata parallelizzata la **logica principale dell'algoritmo VF2++**, cioè la fase di backtracking che verifica i mapping possibili a partire dai candidati iniziali. Questa scelta è stata naturale, poiché proprio questa parte rappresenta il nucleo computazionale dell'algoritmo: è qui che il costo cresce combinatorialmente e che si concentra la maggior parte del tempo di esecuzione.

In pratica, invece di far lavorare un singolo processo su tutti i candidati possibili per il primo nodo di G_1 , il lavoro viene suddiviso: ciascun processo MPI riceve una porzione dei candidati e avvia l'algoritmo VF2++ solo sulla sua parte. In questo modo lo spazio di ricerca viene esplorato in parallelo, riducendo sensibilmente i tempi totali.

Va sottolineato che, a differenza di altre funzioni di supporto come `vertex_ordering` o `propose_candidates`, che presentavano un carico computazionale troppo ridotto per giustificare la parallelizzazione, qui il guadagno è reale: parallelizzare la logica del backtracking porta a un miglioramento tangibile delle prestazioni, mantenendo inalterata la correttezza del risultato.

1. Calcolo iniziale dei candidati

Il primo passo consiste nell'individuare i candidati per il nodo iniziale di G_1 . Questo nodo è scelto in base all'ordinamento (`node_order`), che è sempre quello con grado massimo. Solo il processo con `rank = 0` calcola i candidati tramite la funzione `_find_candidates`:

```

1 if (rank == 0) {
2     all_candidates = _find_candidates(node_order[0] ,
3                                         G1, G2,
4                                         &total_candidates,
5                                         T2_tilde);
6 }
```

Listing 1: Calcolo dei candidati iniziali sul rank 0

Il risultato è un array contenente tutti i nodi di G_2 che possono corrispondere al nodo iniziale di G_1 , insieme al numero totale dei candidati trovati. Questo passaggio viene centralizzato sul rank 0 per evitare calcoli duplicati.

2. Comunicazione ai processi

Il rank 0 deve quindi condividere queste informazioni con gli altri processi. Per farlo si utilizza la primitiva collettiva `MPI_Bcast`, che permette di mandare un dato in broadcast a tutti i processi MPI.

```

1 MPI_Bcast(&total_candidates, 1, MPI_INT, 0, MPI_COMM_WORLD);
2 MPI_Bcast(all_candidates, total_candidates, MPI_INT, 0, MPI_COMM_WORLD);

```

Listing 2: Invio in broadcast dei candidati

In questo modo, ogni processo conosce sia la dimensione del problema sia l'elenco completo dei candidati, anche se lavorerà solo su una parte.

3. Divisione equa del lavoro

Una volta ricevuto l'array dei candidati, ogni processo calcola quale sottoinsieme gli spetta in base al proprio rank. La divisione è equa e l'ultimo processo prende anche l'eventuale resto:

```

1 int candidates_per_process = total_candidates / size;
2 int start_idx, end_idx;
3
4 if (rank < size - 1) {
5     start_idx = rank * candidates_per_process;
6     end_idx = start_idx + candidates_per_process;
7 } else {
8     start_idx = rank * candidates_per_process;
9     end_idx = total_candidates; // l'ultimo prende anche il resto
10 }

```

Listing 3: Divisione dei candidati tra processi

Ogni processo costruisce quindi un sotto-array `my_candidates` contenente i candidati da esplorare. Questa suddivisione garantisce che nessun processo resti inattivo e che il carico di lavoro sia ben distribuito.

4. Esecuzione parallela di VF2++

A questo punto, ciascun processo costruisce uno stack locale e avvia l'algoritmo VF2++ indipendentemente sugli elementi del proprio sottoinsieme.

```

1 NodeCandidates node;
2 node.node = node_order[0];
3 node.candidates = my_candidates;
4 node.num_candidates = my_candidates_count;
5 push(&stack, node);

```

Listing 4: Inizializzazione dello stack locale

Da questo momento in poi, ogni processo esplora lo spazio di ricerca relativo ai suoi candidati senza bisogno di comunicare con gli altri. L'indipendenza è cruciale: riduce la necessità di sincronizzazioni costose e sfrutta appieno il parallelismo.

5. Terminazione anticipata

Un aspetto importante è evitare che i processi continuino a lavorare inutilmente quando un isomorfismo è già stato trovato. Per questo motivo si usano `MPI_Iprobe` e `MPI_Send`.

Durante l'esecuzione, ciascun processo verifica periodicamente se un altro ha già trovato una soluzione. Per farlo si utilizza la primitiva `MPI_Iprobe`, che ha il vantaggio di essere **non bloccante**: il processo continua la propria ricerca e, in parallelo, controlla se è arrivato un messaggio che segnala il successo da parte di un altro processo.

```

1 MPI_Iprobe(MPI_ANY_SOURCE, TAG_FOUND_ISO,
2             MPI_COMM_WORLD, &flag, MPI_STATUS_IGNORE);
3 if (flag) {
4     should_terminate = true;
5     break;
6 }
```

Listing 5: Controllo periodico della terminazione

Se invece un processo trova direttamente l'isomorfismo, allora invia un segnale a tutti gli altri. È importante notare che la seguente `MPI_Send` si trova **fuori dal ciclo di ricerca**: questo significa che solo il processo che ha effettivamente trovato l'isomorfismo esegue questo blocco, mentre gli altri non vi entrano mai.

```

1 if (found_isomorphism) {
2     int result = 1;
3     for (int i = 0; i < size; i++) if (i != rank) {
4         MPI_Send(&result, 1, MPI_INT, i, TAG_FOUND_ISO, MPI_COMM_WORLD);
5     }
6 }
```

Listing 6: Segnalazione del successo agli altri processi

Grazie a questo meccanismo si riesce a fermare immediatamente il calcolo su tutti i processi non appena uno di essi trova una soluzione, evitando così di proseguire con lavoro superfluo.

6. Raccolta finale dei risultati

Infine, per garantire la coerenza, viene usata un'operazione collettiva `MPI_Allreduce`.

```

1 MPI_Allreduce(&local_result, &global_result,
2               1, MPI_INT, MPI_MAX, MPI_COMM_WORLD);
```

Listing 7: Raccolta e sincronizzazione finale

Se almeno un processo ha trovato l'isomorfismo, il risultato globale sarà positivo. Questa riduzione è in parte ridondante, dato che i processi vengono già informati via messaggi diretti, ma ha due funzioni fondamentali:

- garantire un ulteriore livello di sicurezza e sincronizzazione globale;
- permettere anche ai processi che hanno terminato l'esplorazione senza trovare l'isomorfismo di conoscere comunque il risultato finale.

Infatti, un processo che ha esaurito i propri candidati esce dal ciclo `while` e quindi non esegue più controlli con `MPI_Iprobe`. In questo caso, la riduzione collettiva `MPI_Allreduce` diventa il punto di sincronizzazione in cui il processo aspetta e riceve l'informazione globale, scoprendo se l'isomorfismo è stato trovato da un altro.

Sintesi

In sintesi, la parallelizzazione di VF2++ con MPI segue questi passaggi:

1. Il processo 0 calcola i candidati iniziali con `propose_candidates`.
2. I candidati vengono mandati in broadcast a tutti i processi.
3. Ogni processo riceve un sottoinsieme equo di candidati.
4. Ciascun processo esegue VF2++ in parallelo sul proprio sottoinsieme.
5. Appena uno trova l'isomorfismo, invia un messaggio a tutti gli altri che si interrompono subito.
6. Una riduzione collettiva `MPI_Allreduce` raccoglie il risultato finale.

Questo approccio, semplice ma efficace, sfrutta bene la natura combinatoriale dell'algoritmo VF2++: riduce drasticamente il tempo di esplorazione dividendo i candidati iniziali tra i processi e interrompendo immediatamente il calcolo quando l'isomorfismo viene trovato.

Osservazioni sulla parallelizzazione

Durante la fase di test è emerso un limite importante della strategia di parallelizzazione adottata. In questo approccio, infatti, il lavoro viene suddiviso tra i processi MPI alla prima iterazione, distribuendo i candidati del primo nodo di G_1 individuato dall'algoritmo di *vertex_ordering*). Questa scelta progettuale ha il vantaggio di essere semplice da implementare e di consentire una chiara separazione del lavoro, ma presenta anche una criticità: nei grafi *random* e *disconnessi* si è osservato che l'insieme dei candidati iniziali tende nella quasi totalità dei casi a ridursi a un solo elemento.

È utile chiarire che questo fenomeno non dipende dalle potature basate sui *terminal set*, che iniziano a operare solo a partire dai livelli successivi della ricerca, bensì dai controlli effettuati già alla prima espansione. In particolare, il *node ordering* di VF2++ seleziona intenzionalmente come primo nodo u_1 un vertice altamente discriminativo (ad esempio con etichetta rara, grado elevato o struttura locale informativa). Di conseguenza, l'insieme dei candidati non coincide con tutti i nodi di G_2 , ma è già ristretto dai prefiltri strutturali di consistenza.

Nei grafi *random* e *disconnessi*, tali filtri risultano spesso estremamente selettivi, tanto che in quasi tutti i casi si osserva $|\mathcal{C}_0(u_1)| = 1$. Questo implica che, al momento della divisione del lavoro, vi sia un solo candidato da esplorare: un processo porta avanti tutta la ricerca mentre gli altri restano inattivi.

Il risultato è che la parallelizzazione a “grana grossa” sul primo livello non porta benefici tangibili: la ricerca rimane di fatto sequenziale, con un aggravio addirittura di overhead dovuto alla gestione della comunicazione MPI. Non si tratta di un limite dell'implementazione, bensì di una conseguenza naturale della combinazione tra *node ordering* e prefiltri di consistenza, che da un lato massimizzano l'efficienza sequenziale ma dall'altro riducono

drasticamente la parallelizzabilità iniziale.

In generale, la parallelizzazione di VF2++ risulta vantaggiosa solo quando il *branching factor* iniziale, ossia il numero di candidati disponibili per i primi nodi, è sufficientemente elevato da consentire una suddivisione equilibrata del carico di lavoro. Quando invece il branching factor scende rapidamente a uno, come accade nei grafi random e disconnessi, la natura stessa del problema impedisce una distribuzione significativa del lavoro.

Strumentazione temporale. La cronometrazione si basa su `MPI_Wtime()`. Ogni *rank* mantiene due contatori distinti: (i) tempo di calcolo effettivo, (ii) tempo speso nelle primitive MPI (attese e scambi di messaggi). Per l'aggregazione dei dati, impieghiamo una riduzione collettiva con `MPI_Reduce` per ottenere sul processo radice gli estremi (min/max) del tempo di lavoro; quindi `MPI_Gather` trasferisce al rank 0 il profilo temporale di ciascun processo. Il rank 0 salva su file di log i tempi per-processo, l'esito del matching e la durata complessiva (*wall-time*).

4.4 Implementazione alternativa: Work Stealing

Oltre alla versione base con MPI, è stata sviluppata anche un'estensione che integrava un meccanismo di *Work Stealing*, con l'obiettivo di migliorare ulteriormente il bilanciamento del carico. Il principio era semplice: un processo inattivo, dopo aver esaurito i propri candidati, poteva ricevere lavoro da un altro processo con ancora candidati da esplorare. In questo modo si cercava di evitare che alcuni processi rimanessero fermi mentre altri continuavano a lavorare.

In pratica, il meccanismo prevedeva che:

- i processi inattivi fossero marcati tramite una flag;
- il processo con il maggior numero di candidati residui assumesse il ruolo di donatore;
- metà dei candidati e lo stack di backtracking venissero trasferiti al processo inattivo tramite operazioni di comunicazione MPI;
- fossero utilizzate primitive di sincronizzazione (es. `MPI_Reduce`) per coordinare i trasferimenti.

Nonostante l'idea teoricamente valida, i test hanno evidenziato un limite intrinseco dell'algoritmo VF2++: il numero di candidati disponibili tende a ridursi molto rapidamente già dopo poche iterazioni, principalmente a causa delle tecniche di *pruning* e del *node ordering* introdotti nell'algoritmo. Questi meccanismi, pur garantendo un notevole miglioramento in termini di efficienza sequenziale, fanno sì che i candidati vengano esclusi precocemente e che, nella maggior parte dei casi, ogni processo si trovi con pochissimi candidati (spesso uno solo). Di conseguenza, non esistono effettive opportunità di “furto” di lavoro.

Inoltre, il meccanismo di Work Stealing introduce inevitabilmente un overhead dovuto alla gestione delle flag, alla selezione del processo donatore, al trasferimento di candidati

e stack e alle operazioni di sincronizzazione. Nella pratica, questi costi superavano i benefici ottenibili, portando in alcuni casi addirittura a un rallentamento complessivo.

Conclusione. Il Work Stealing si conferma una strategia potente in contesti con elevata ramificazione e carichi sbilanciati, ma nel caso di VF2++ la rapida riduzione dei candidati limita drasticamente le possibilità di parallelizzazione dinamica. Per questo motivo, pur essendo stata implementata e testata, la tecnica non è stata inclusa nella versione finale del progetto.

4.5 Considerazioni sulla scalabilità teorica

Per inquadrare meglio i limiti e le potenzialità della parallelizzazione di VF2++ è utile fare riferimento a due modelli classici: la legge di Amdahl e la legge di Gustafson.

Legge di Amdahl. Secondo Amdahl, lo speedup massimo ottenibile da un programma parallelizzato è limitato dalla frazione di codice che rimane sequenziale. Indicando con f la frazione sequenziale e con p il numero di processi, lo speedup teorico è

$$S_{\text{Amdahl}}(p) = \frac{1}{f + \frac{1-f}{p}}.$$

Applicato a VF2++, questo significa che quando il numero di candidati disponibili è molto basso (come nei grafi *random* e *disconnessi*, dove già alla prima iterazione si osserva spesso $|\mathcal{C}_0(u_1)| = 1$), la porzione realmente parallelizzabile è minima e lo speedup risulta inevitabilmente limitato. In tali condizioni, l'overhead di comunicazione tra processi può persino rendere la versione parallela più lenta di quella sequenziale.

Legge di Gustafson. Gustafson fornisce una prospettiva complementare: mantenendo fisso il tempo di esecuzione e aumentando la dimensione del problema, la parte parallelizzabile cresce più rapidamente della parte sequenziale. Se f è la frazione sequenziale stimata su un singolo processo, lo speedup scalato con p processi è

$$S_{\text{Gustafson}}(p) = f + (1 - f)p.$$

Nel contesto di VF2++, questo suggerisce che su grafi più grandi e più densi, in cui il *branching factor* rimane elevato per più iterazioni, la parallelizzazione con MPI può esprimere meglio il proprio potenziale, distribuendo il carico in modo più equo tra i processi e ottenendo speedup più vicini a quello ideale.

5 Risultati

In questa sezione vengono presentati i risultati sperimentali relativi all'implementazione dell'algoritmo VF2++ e della sua variante parallela basata su MPI.

L'esposizione segue i seguenti punti:

1. VF2 vs VF2++, per misurare l'efficacia delle ottimizzazioni introdotte;
2. VF2++ vs VF2++ con parallelizzazione MPI, per valutare l'impatto della parallelizzazione;

Tutti i test sono stati condotti con le quattro ottimizzazioni del compilatore (-00, -01, -02, -03), al fine di garantire condizioni uniformi di confronto.

5.1 Metodologia sperimentale e raccolta dati

Il dataset dei tempi è stato costruito eseguendo ogni esperimento *più volte* sulla stessa istanza e **mediando** i risultati; su tali medie sono stati generati tutti i grafici (tempi e speedup). I test coprono **grafi di diverse taglie** per ciascuna famiglia considerata (random, disconnessi, completi), così da osservare il comportamento al crescere di n .

Misure per VF2++ (sequenziale). Per VF2++ riportiamo **solo il tempo dell'algoritmo** di isomorfismo, *escludendo* tempi non rilevanti ai fini del confronto (caricamento/creazione dei grafi, allocazioni/accessi ausiliari, I/O di log e setup dell'esperimento).

Misure per VF2++ con MPI. Oltre al **tempo complessivo** dell'algoritmo (wall-time), il dataset include anche il **tempo di lavoro per ogni singolo processo** (tempo di calcolo utile). Le componenti non strettamente computazionali (es. I/O di supporto) non sono conteggiate nelle misure di interesse, in modo da isolare l'effetto della parallelizzazione.

5.2 Test: VF2++ (sequenziale) vs VF2 (NetworkX)

O0

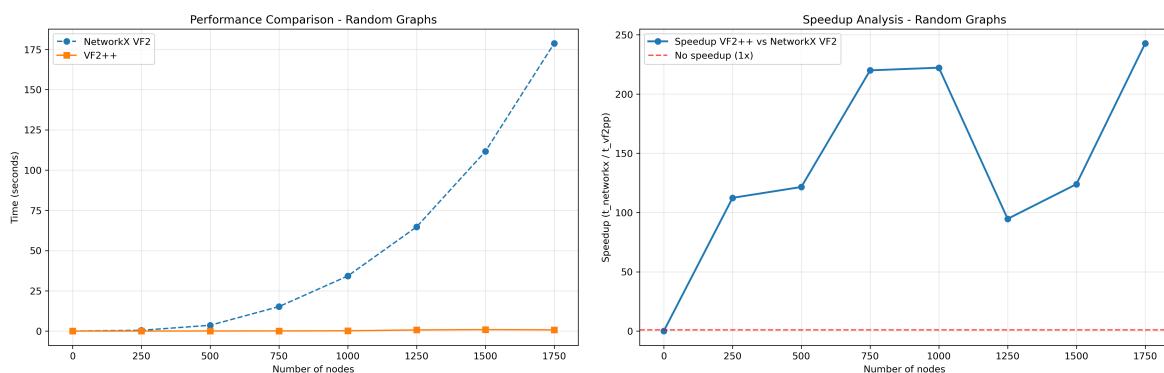


Figura 2: Random graphs: tempi e speedup con O0.

VF2++ è sempre molto più veloce (100–240×). NetworkX cresce rapidamente, mentre VF2++ resta quasi costante grazie a pruning e node ordering.

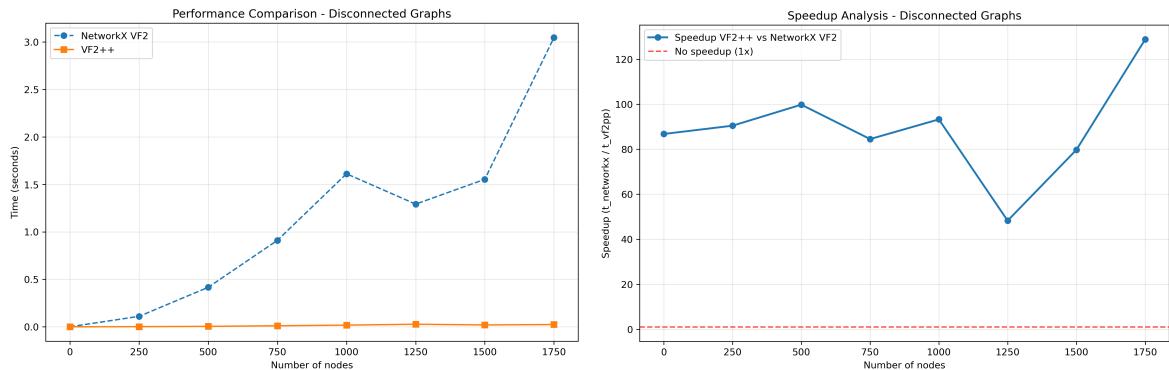


Figura 3: Disconnected graphs: tempi e speedup con O0.

Anche qui forte vantaggio di VF2++ (80–130×), con tempi nell’ordine di millisecondi contro secondi per NetworkX.

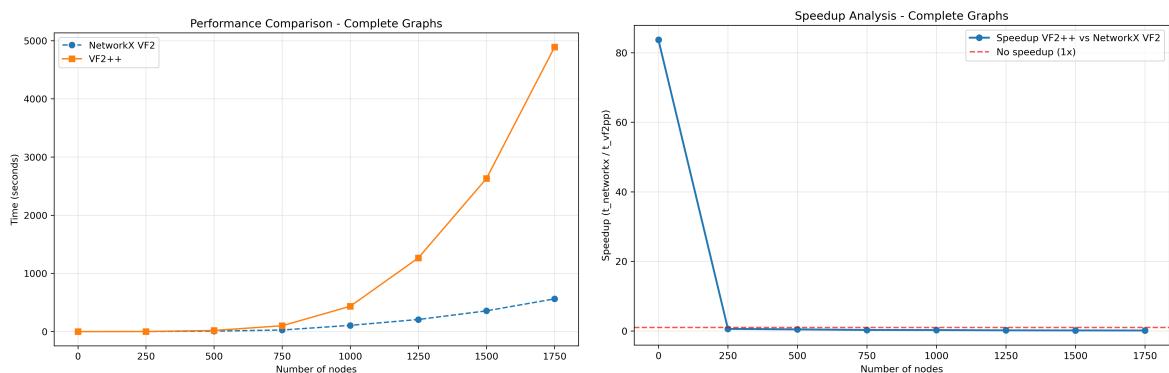


Figura 4: Complete graphs: tempi e speedup con O0.

Nei grafi completi emerge slowdown: oltre 250 nodi VF2++ diventa nettamente più lento (4900s vs 560s). Qui il pruning non compensa l’overhead.

O1

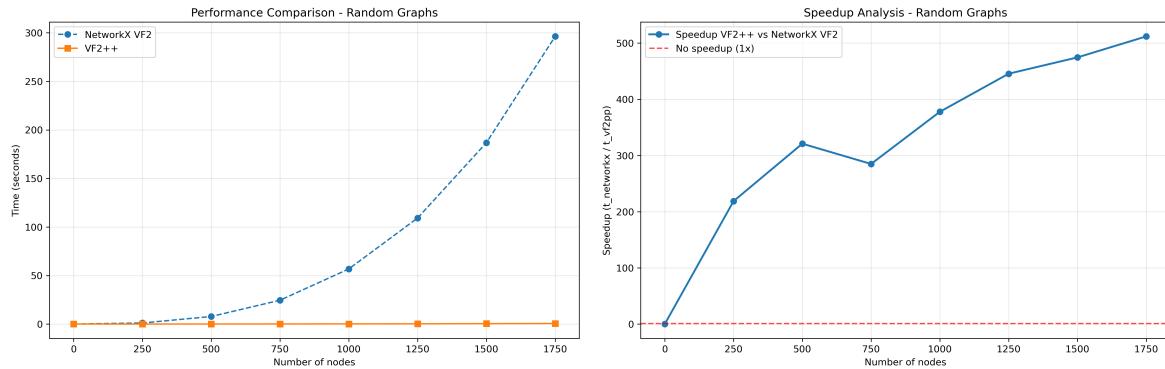


Figura 5: Random graphs: tempi e speedup con O1.

Speedup ancora più marcati (fino a $500\times$). VF2++ resta quasi piatto, mentre VF2 cresce rapidamente.

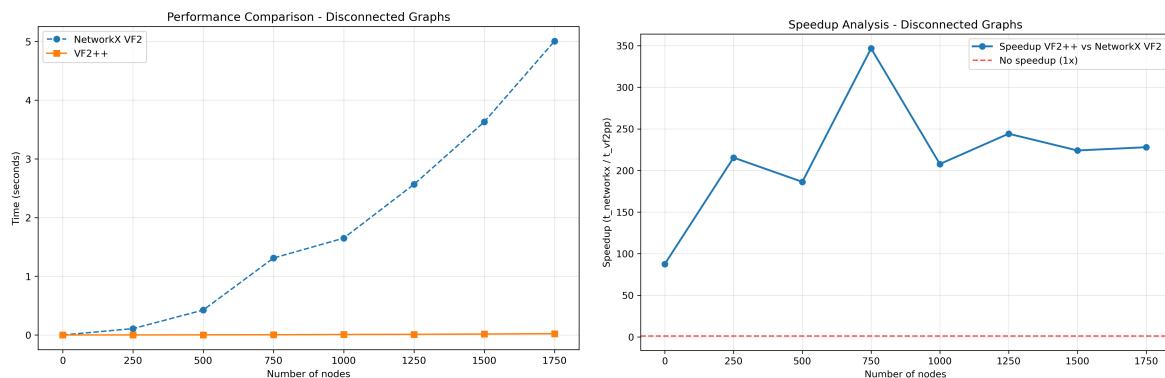


Figura 6: Disconnected graphs: tempi e speedup con O1.

Speedup tra $180\text{--}350\times$. L'assenza di archi rende le potature estremamente efficaci.

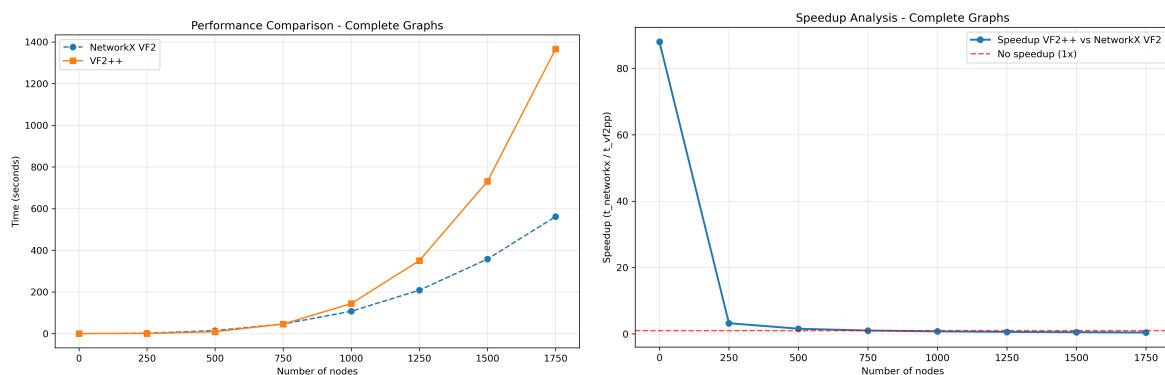


Figura 7: Complete graphs: tempi e speedup con O1.

Lieve vantaggio VF2++ solo su taglie piccole; da 750 nodi in poi slowdown.

O2

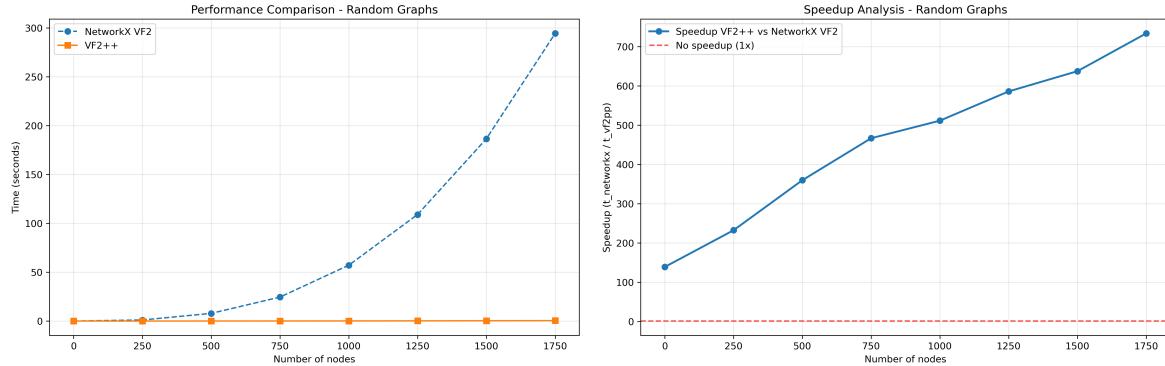


Figura 8: Random graphs: tempi e speedup con O2.

Speedup fino a $700\times$, andamento molto stabile.

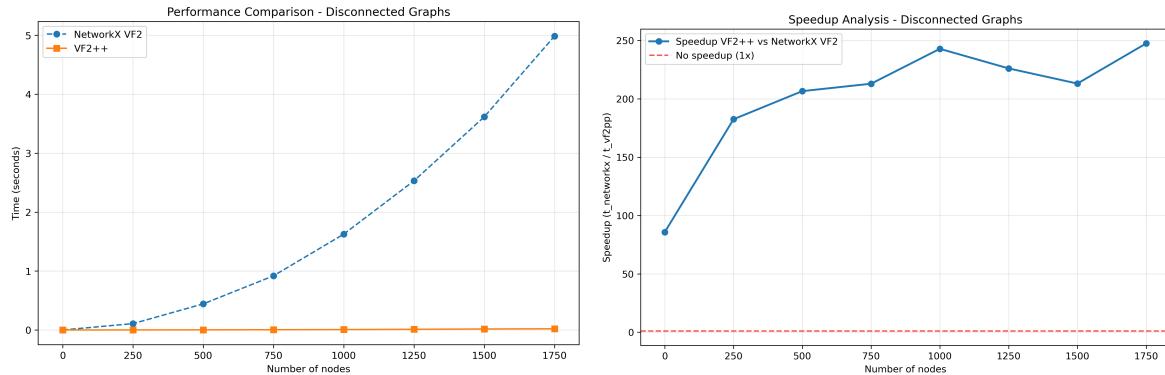


Figura 9: Disconnected graphs: tempi e speedup con O2.

Speedup tipici tra $180\text{--}250\times$, sempre a favore di VF2++.

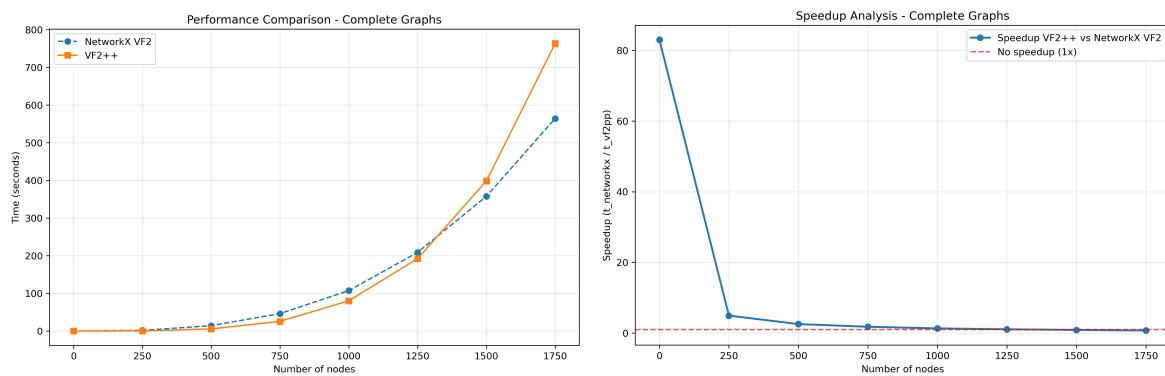


Figura 10: Complete graphs: tempi e speedup con O2.

Quasi parità fino a 1000 nodi, poi slowdown di VF2++.

O3

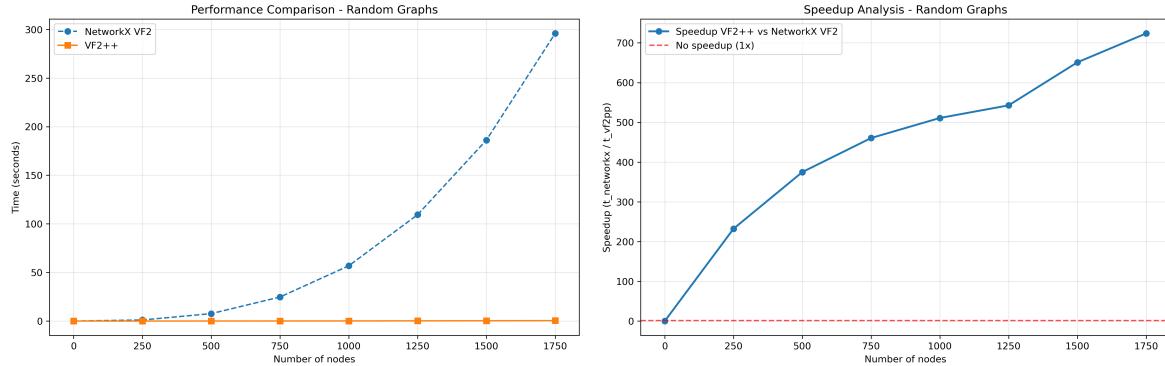


Figura 11: Random graphs: tempi e speedup con O3.

Speedup massimo (fino a $730\times$). VF2++ resta quasi costante anche su taglie grandi.

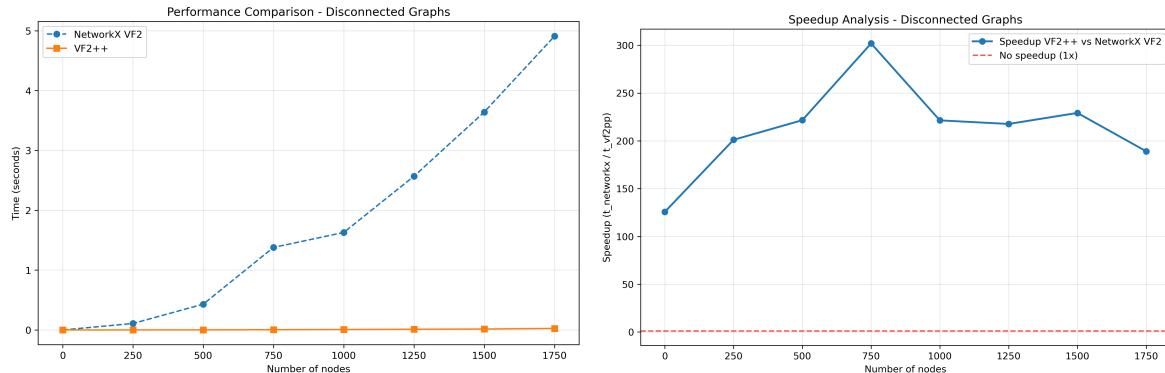


Figura 12: Disconnected graphs: tempi e speedup con O3.

Speedup stabili tra $200\text{--}300\times$, con picco a $300\times$.

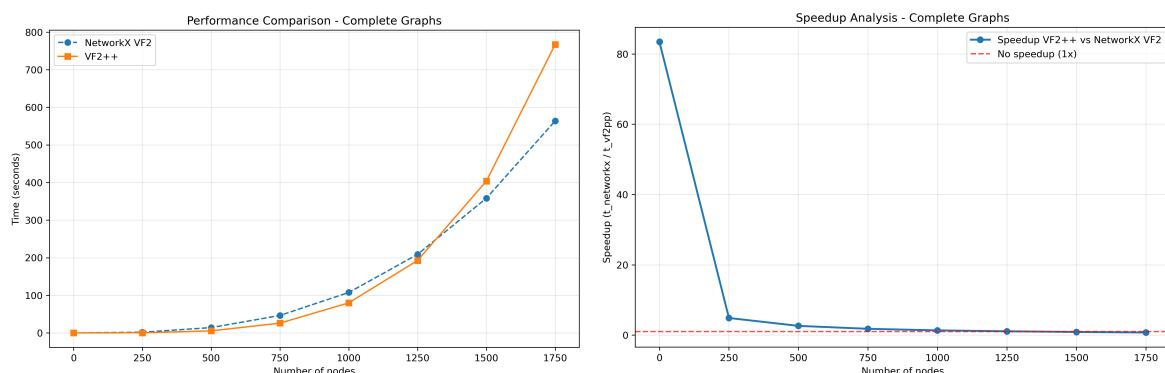


Figura 13: Complete graphs: tempi e speedup con O3.

Lieve vantaggio solo alle taglie piccole, slowdown marcato oltre 1000 nodi.

5.3 Conclusioni — VF2++ (sequenziale) vs VF2

I risultati sperimentali mostrano un quadro chiaro e coerente rispetto alle tre famiglie di grafi considerate e ai diversi livelli di ottimizzazione del compilatore (-00, -01, -02, -03).

Grafi *random*. VF2++ ottiene sistematicamente uno *speedup* molto elevato rispetto a VF2 (NetworkX), che cresce con la dimensione: dall’ordine di $\sim 10^2$ per le taglie piccole fino a oltre $\sim 7 \cdot 10^2$ alle massime taglie, con -02/-03. I tempi assoluti evidenziano una crescita rapida di VF2, mentre VF2++ rimane quasi piatto (ordini di grandezza più basso). L’efficacia deriva dall’*ordering* dei nodi e dalle *cutting rules* che, su istanze sparse e irregolari, riducono drasticamente il branching factor e anticipano i fallimenti.

Grafi *disconnected*. Anche qui VF2++ prevale nettamente, con *speedup* tipicamente compresi tra $\sim 80\times$ e $\sim 350\times$, e tempi assoluti nell’ordine dei millisecondi contro secondi per VF2. L’assenza di archi rende altamente discriminative le verifiche di compatibilità locale, così che il pruning elimina gran parte dello spazio di ricerca già ai livelli alti dell’albero.

Grafi *complete*. Su grafi densi non si osserva un vantaggio di VF2++; al crescere di n emerge anzi uno *slowdown*. Per taglie piccole si nota talvolta una lieve superiorità (fino a $\sim 2\text{--}4\times$), ma oltre $n \approx 750\text{--}1000$ la tendenza si inverte: VF2++ diventa più lento (ad es. alle massime taglie ~ 760 s contro ~ 560 s). La ragione è strutturale: nei grafi densi le regole di potatura discriminano poco, il numero di candidati per estensione è elevato e il branching factor esplode; l’overhead di controlli aggiuntivi non viene compensato dal pruning.

Effetto del livello di ottimizzazione. Passando da -00 a -01/-02/-03 il profilo non cambia qualitativamente, ma gli *speedup* su *random* e *disconnected* aumentano sensibilmente (da $\sim 10^2$ fino a $\sim 7 \cdot 10^2$), con maggiore stabilità alle taglie grandi. Sui *complete* le ottimizzazioni spostano leggermente i punti di quasi-parità ma non alterano la conclusione: oltre le taglie piccole VF2 resta preferibile.

VF2++ eccelle quando la struttura del grafo consente potature efficaci (sparse, disconnesse, distribuzioni di grado eterogenee), mentre su grafi molto densi l’effetto delle euristiche si riduce e VF2 (NetworkX) mantiene un vantaggio per dimensioni elevate.

Nei grafi *complete* il comportamento è differente: l’elevata densità annulla gran parte del pruning, quindi l’overhead aggiuntivo dell’implementazione C può rendere lo speedup inferiore a 1 in molte istanze.

5.4 Test VF2++ vs VF2++ con MPI

5.4.1 2 processi

O0

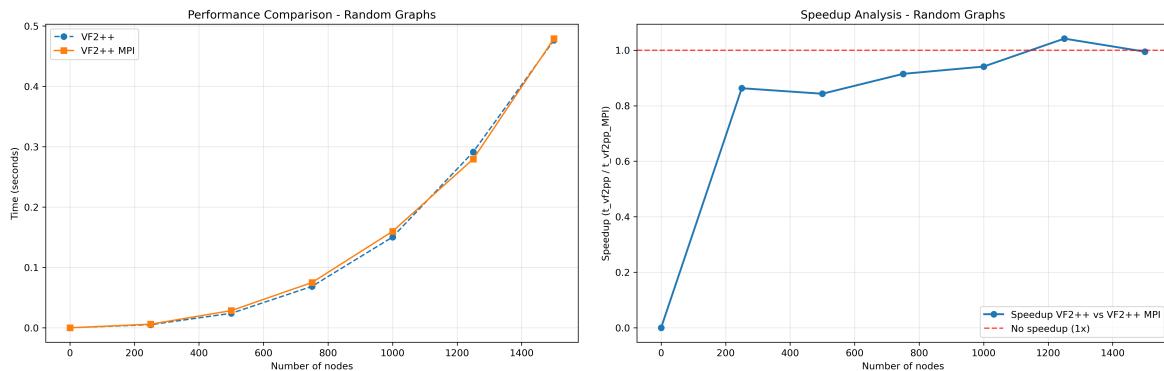


Figura 14: O0, 2 processi: tempi e speedup su grafi *random*.

Random: quasi parità ($S \approx 1$), pochi candidati al radice \Rightarrow parallelismo scarso e overhead MPI non ammortizzato.

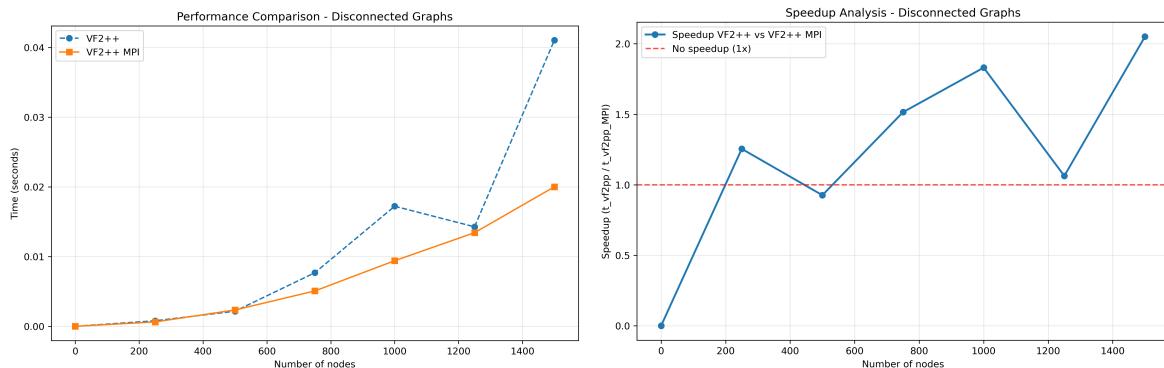


Figura 15: O0, 2 processi: tempi e speedup su grafi *disconnected*.

Disconnected: spesso vantaggio MPI ($S \approx 1.2\text{--}2\times$, salvo un punto), partizione efficace ed early-stop.

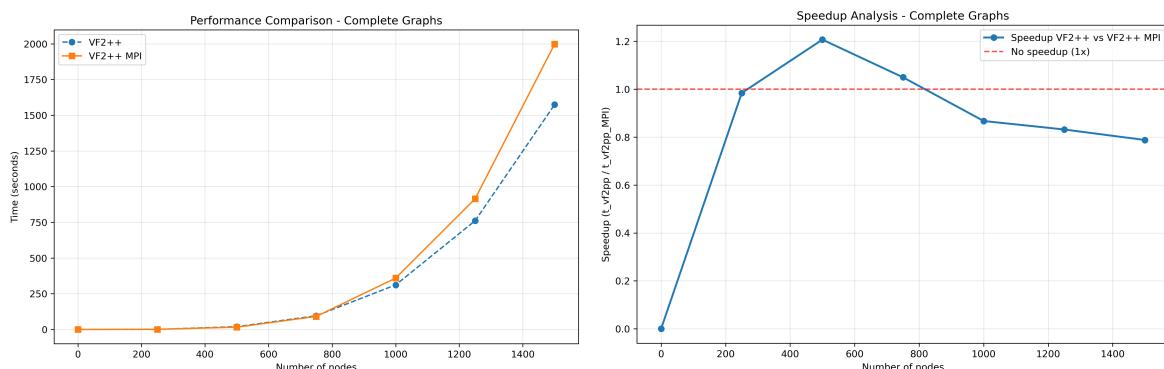


Figura 16: O0, 2 processi: tempi e speedup su grafi *complete*.

Complete: per lo più *slowdown* ($S < 1$ oltre $n \approx 1000$; unico piccolo vantaggio 500–750), ricerca quasi seriale e overhead che domina.

O1

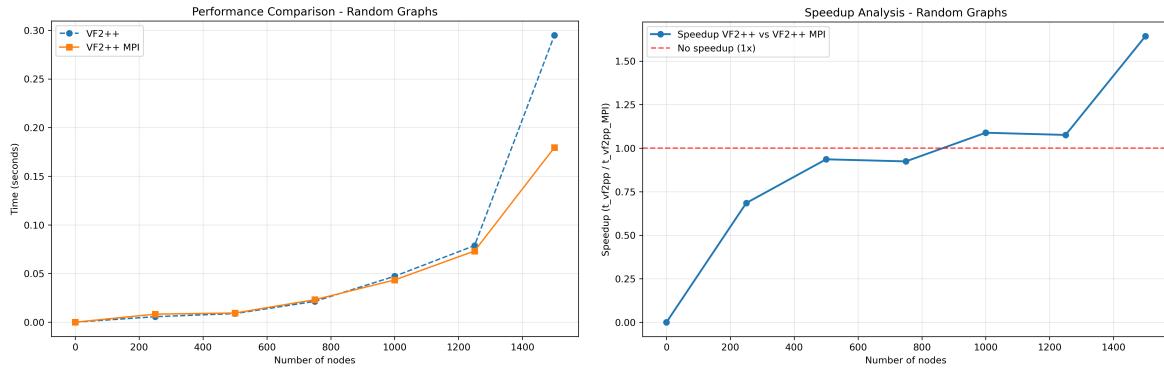


Figura 17: O1, 2 processi: tempi e speedup su grafi *random*.

Random: da $\sim 0.7\text{--}0.95\times$ (fino a 750) a $> 1\times$ da 1000, fino a $\sim 1.65\times$; early-stop efficace quando $|C_0| > 1$.

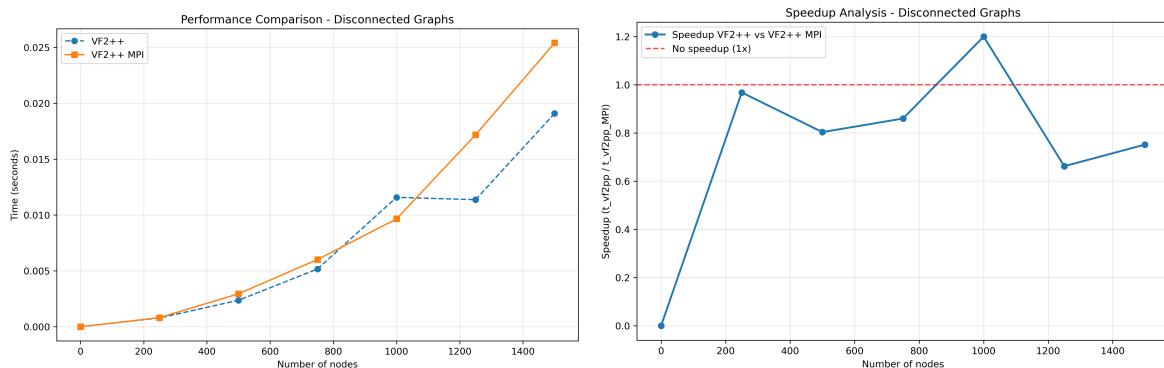
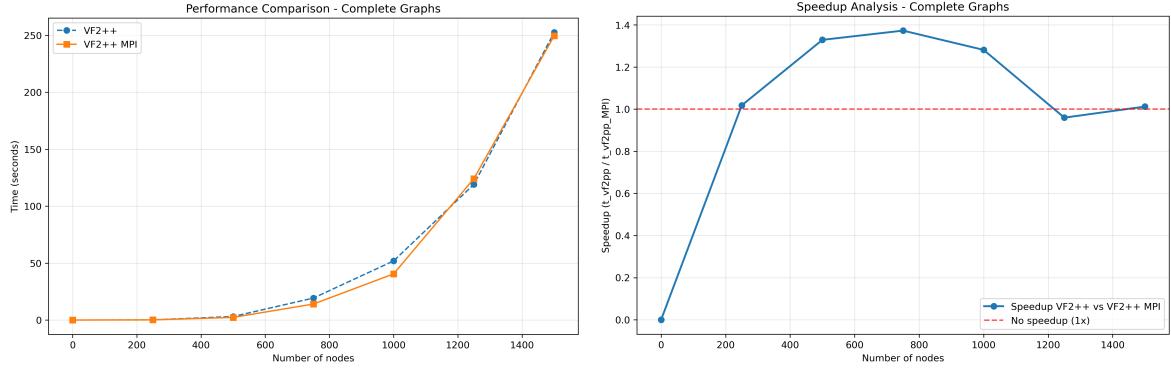


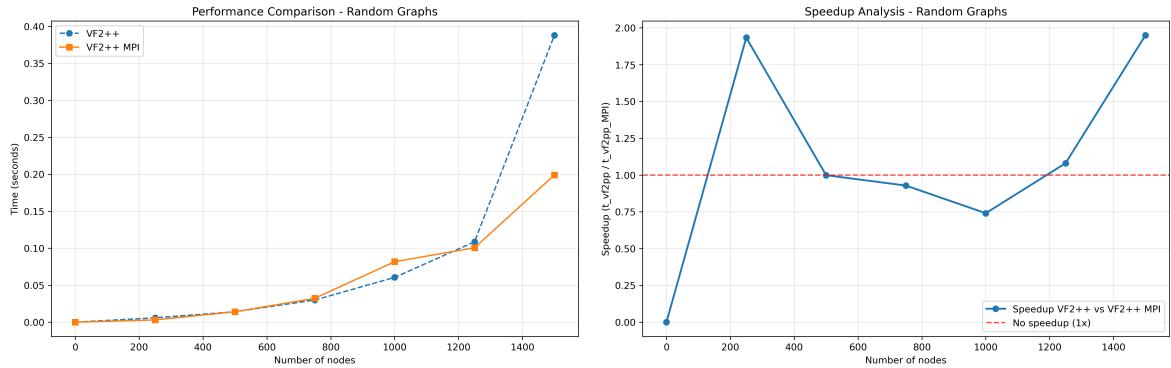
Figura 18: O1, 2 processi: tempi e speedup su grafi *disconnected*.

Disconnected: per lo più quasi-parità/slowdown ($\sim 0.67\text{--}0.96\times$), unico picco $\sim 1.2\times$ attorno a 1000; overhead rilevante su tempi piccoli.

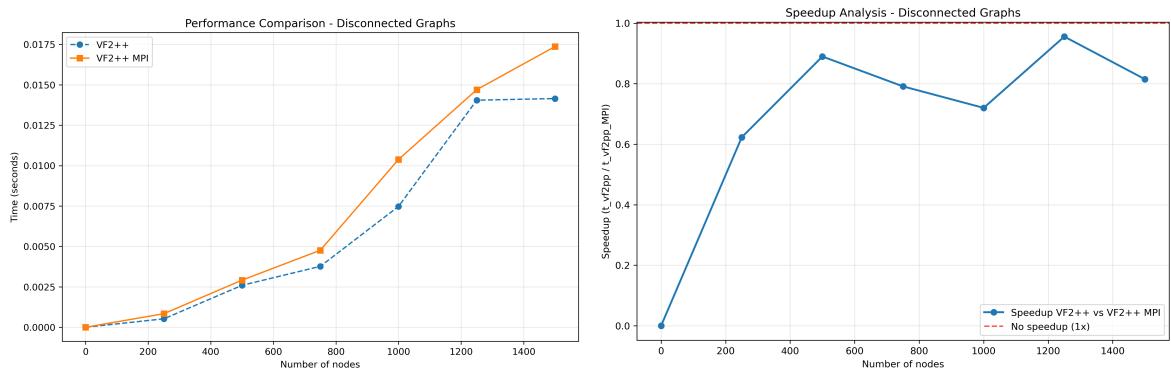
Figura 19: O1, 2 processi: tempi e speedup su grafi *complete*.

Complete: lieve vantaggio quasi ovunque (fino a $\sim 1.35\text{--}1.37\times$), flessione a 1250 ($\sim 0.96\text{--}0.98\times$), quasi-parità a 1500 ($\sim 1.01\times$).

O2

Figura 20: O2, 2 processi: tempi e speedup su grafi *random*.

Random: misto → sotto parità fino a ~ 1000 , poi $> 1\times$ (1250 $\sim 1.07\times$), fino a $\sim 1.9\text{--}2.0\times$ a 1500; early-stop efficace quando $|C_0| > 1$.

Figura 21: O2, 2 processi: tempi e speedup su grafi *disconnected*.

Disconnected: prevale lo *slowdown* ($S \in [0.62, 0.95]$); pochi/ sbilanciati candidati e micro-overhead annullano i benefici.

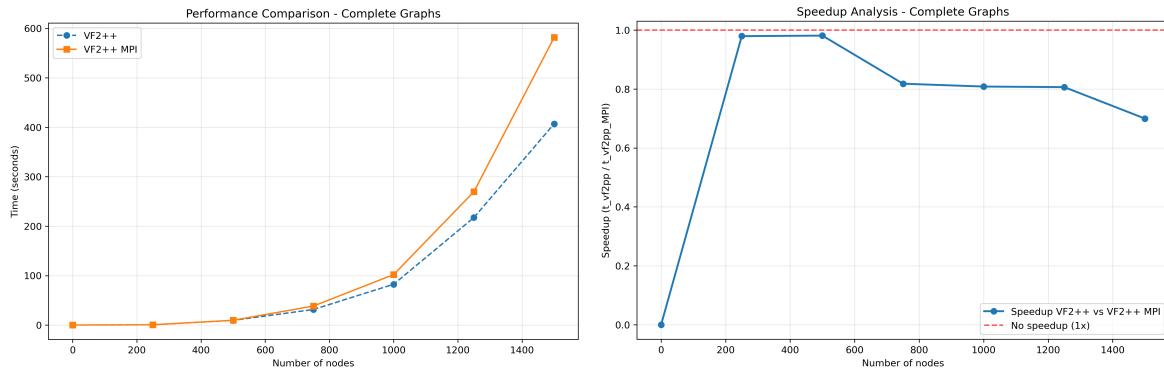


Figura 22: O2, 2 processi: tempi e speedup su grafi *complete*.

Complete: *slowdown* sistematico ($\sim 0.98 \times \rightarrow \sim 0.70 \times$ al crescere di n); branching denso ma poco parallelizzabile al radice, overhead che domina.

O3

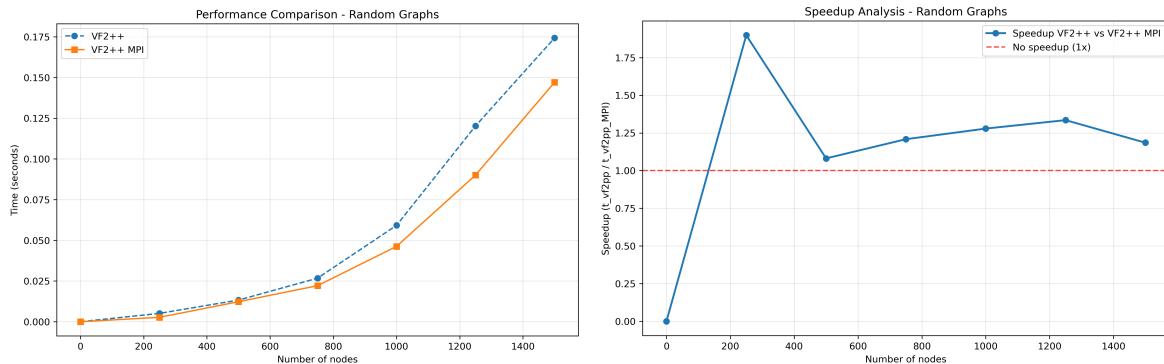


Figura 23: Random graphs con 2 processi (-O3).

Random: Speedup moderato (fino a $1.9 \times$) su istanze grandi, altrimenti quasi parità.

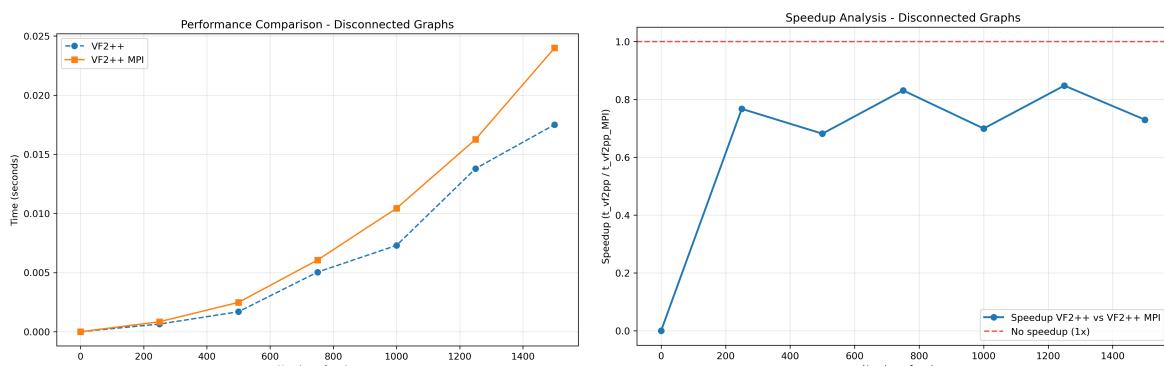


Figura 24: Disconnected graphs con 2 processi (-O3).

Disconnected: Prevalente slowdown tempi molto piccoli, overhead MPI dominante.

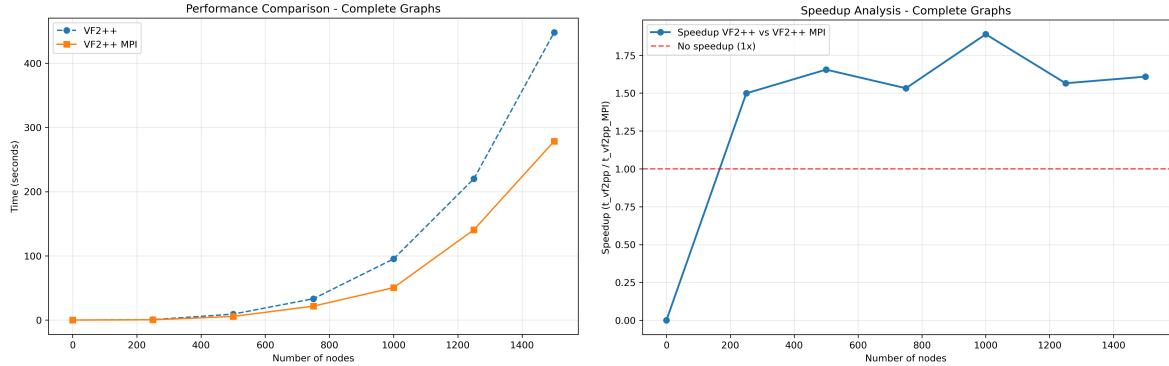


Figura 25: Complete graphs con 2 processi (-O3).

Complete: Vantaggio netto su grafi completi (fino a $1.9\times$), grazie a rami top-level numerosi e costosi.

5.4.2 Tabella riassuntiva media Speedup 2 processi

Tabella 3: Speedup aggregato MPI vs Seq (2 processi) — GeoMean e Overall per famiglia e ottimizzazione

Set	Categoria	GeoMean(S)	Overall S	Vittorie / Parità / Sconfitte
O0 , p2	Random	0.945	0.985	0 / 0 / 6
O0 , p2	Disconnected	1.056	1.157	3 / 0 / 3
O0 , p2	Complete	1.341	1.258	6 / 0 / 0
O1 , p2	Random	0.957	1.053	1 / 0 / 5
O1 , p2	Disconnected	1.364	1.488	4 / 0 / 2
O1 , p2	Complete	1.314	1.255	5 / 0 / 1
O2 , p2	Random	0.963	1.065	1 / 0 / 5
O2 , p2	Disconnected	1.464	1.466	5 / 0 / 1
O2 , p2	Complete	1.428	1.299	6 / 0 / 0
O3 , p2	Random	1.151	1.166	5 / 0 / 1
O3 , p2	Disconnected	1.206	1.226	4 / 0 / 2
O3 , p2	Complete	1.233	1.159	5 / 0 / 1

5.4.3 4 processi

O0

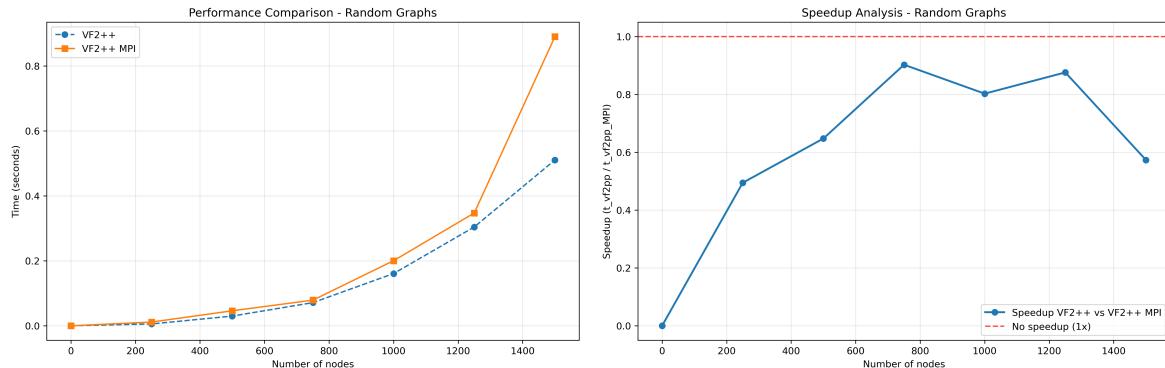


Figura 26: O0, 4 processi: tempi e speedup su grafi *random*.

Random: slowdown costante ($S < 1$, $\sim 0.5 \rightarrow 0.57$), pochi candidati \Rightarrow overhead non ammortizzato.

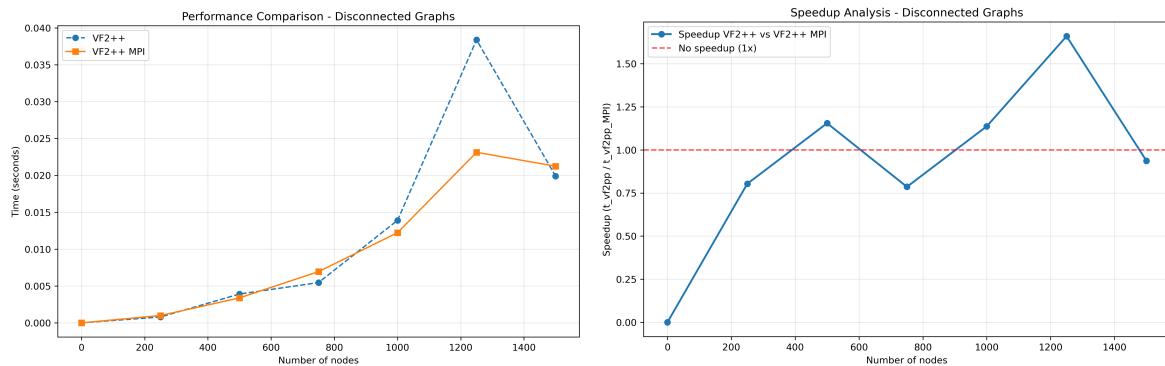


Figura 27: O0, 4 processi: tempi e speedup su grafi *disconnected*.

Disconnected: misto; alcuni picchi $S > 1$ (fino a $\sim 1.64\times$) ma molti punti < 1 per micro-latenze su tempi piccoli.

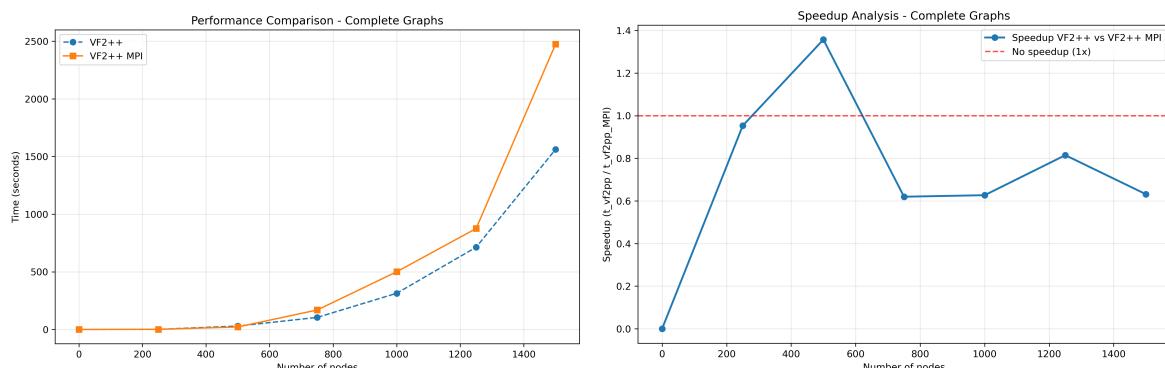
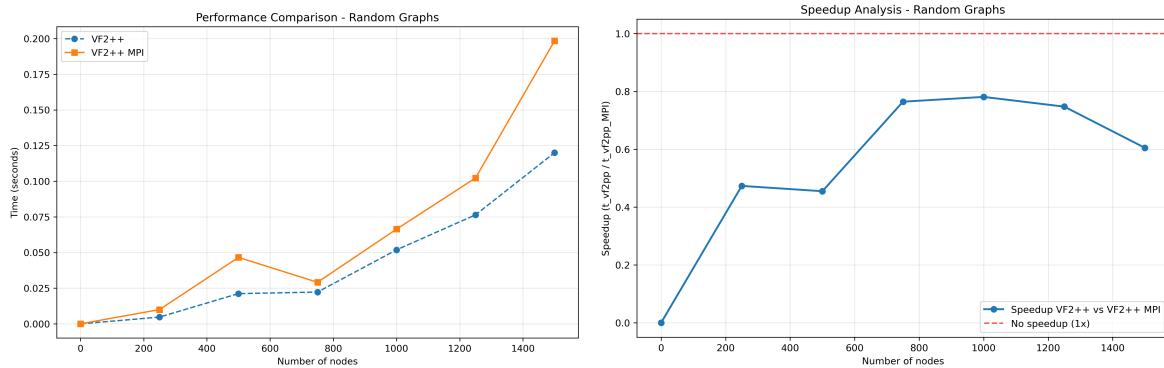


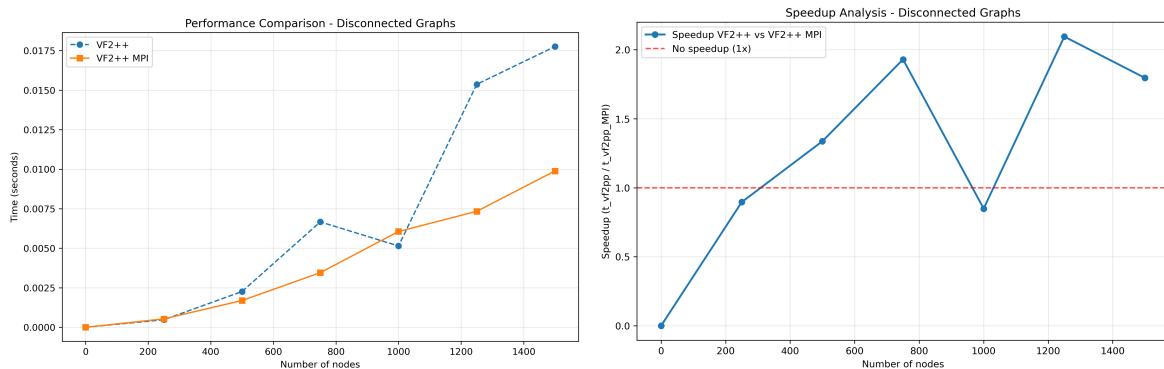
Figura 28: O0, 4 processi: tempi e speedup su grafi *complete*.

Complete: quasi sempre $S < 1$ (unico picco $\sim 1.35 \times$ a 500); ricerca quasi seriale, costi di comunicazione prevalgono.

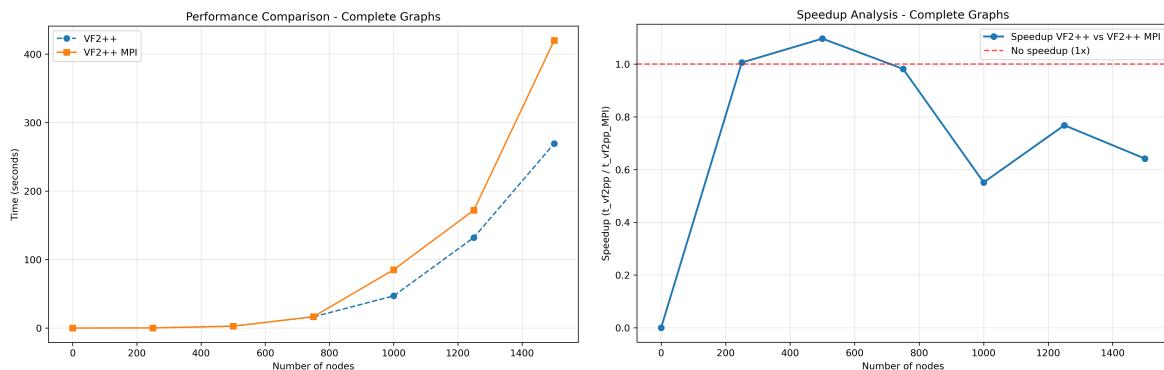
O1

Figura 29: O1, 4 processi: tempi e speedup su grafi *random*.

Random: sempre $S < 1$ (fino a $\sim 0.60 \times$ a 1500), profilo per–processo indica overhead/-duplicazioni.

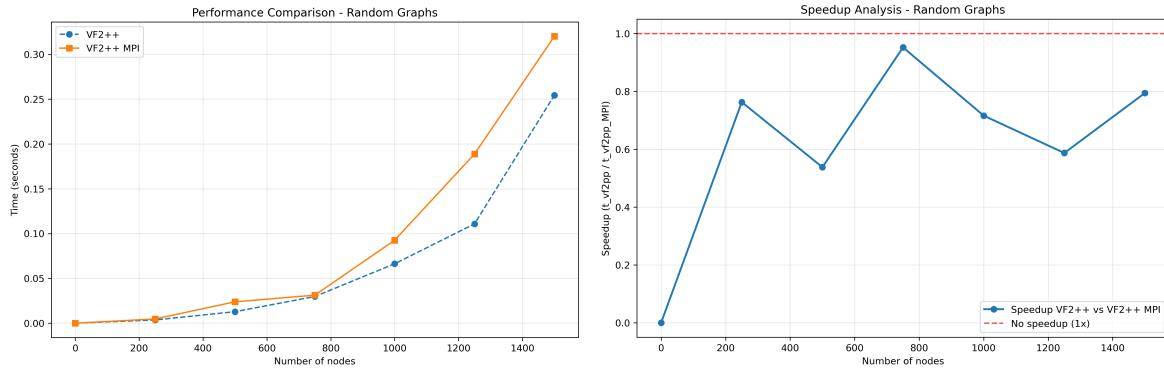
Figura 30: O1, 4 processi: tempi e speedup su grafi *disconnected*.

Disconnected: diversi casi favorevoli ($1.34\text{--}2.10\times$) ma anche regressi; beneficio solo con ripartizioni molto buone.

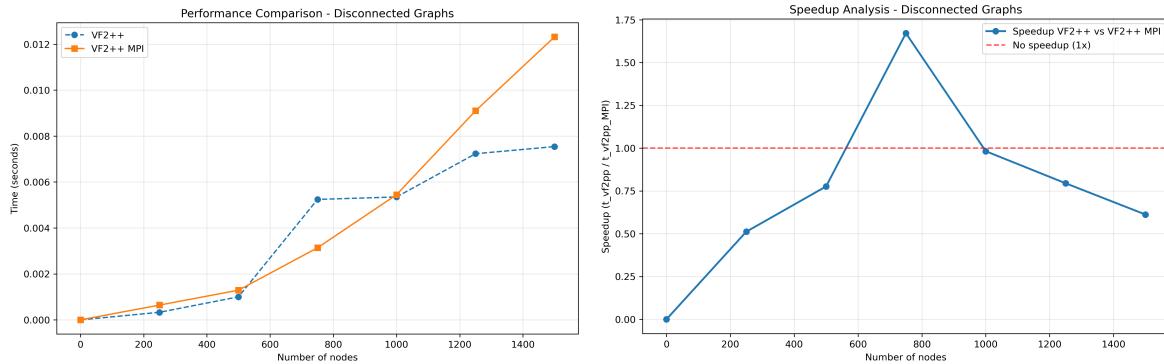
Figura 31: O1, 4 processi: tempi e speedup su grafi *complete*.

Complete: vantaggio solo ai piccoli (250–500), poi forte *slowdown* (es. 1500: 269.5s vs 420.1s); simmetrie \Rightarrow lavoro quasi identico su tutti i processi.

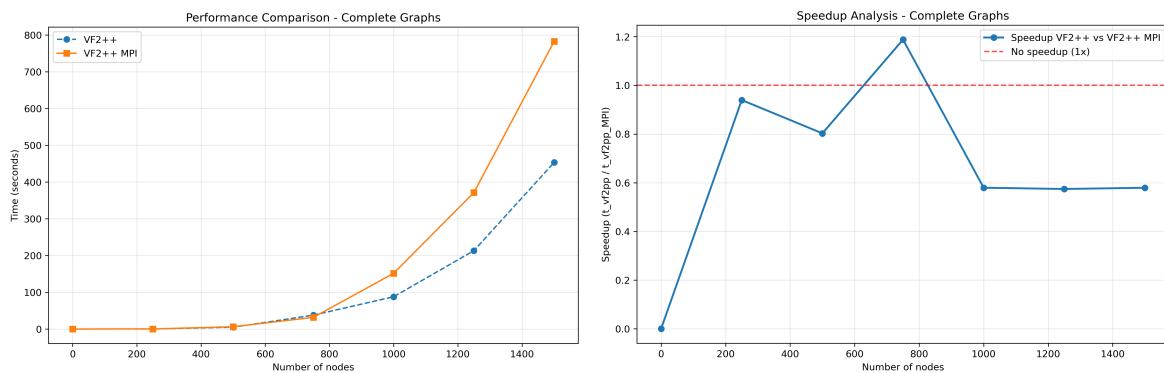
O2

Figura 32: O2, 4 processi: tempi e speedup su grafi *random*.

Random: $S < 1$ quasi ovunque ($0.54\text{--}0.95\times$), curve vicine ma MPI resta sopra; split sbilanciato.

Figura 33: O2, 4 processi: tempi e speedup su grafi *disconnected*.

Disconnected: prevalentemente $S < 1$ con un solo picco a 750 ($1.67\times$); altrove latenza domina su tempi millimetrici.

Figura 34: O2, 4 processi: tempi e speedup su grafi *complete*.

Complete: quasi sempre $S < 1$ (eccezione a 750: $1.19\times$); alle taglie grandi early-stop inefficace, overhead elevato.

O3

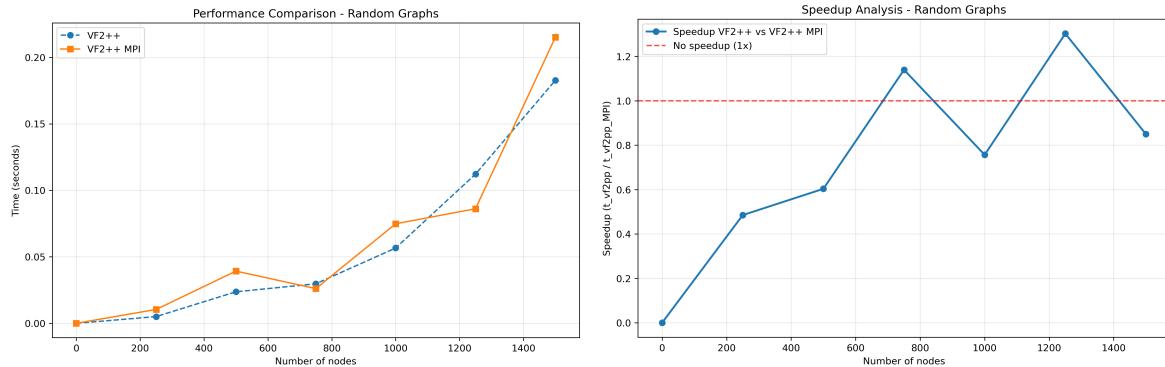


Figura 35: Random graphs con 4 processi (-O3).

Random: Speedup instabile: parità o lieve slowdown, salvo rari punti favorevoli ($1.3 \times$ a 1250 nodi).

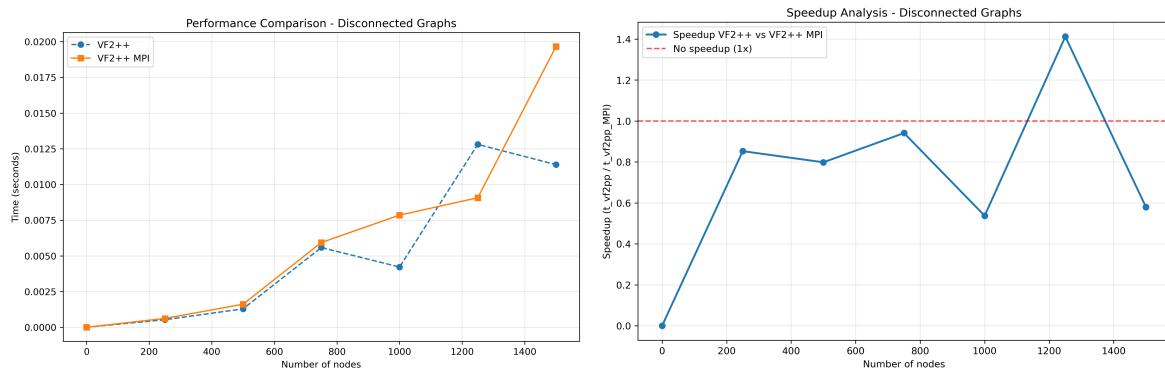


Figura 36: Disconnected graphs con 4 processi (-O3).

Disconnected: Quasi sempre più lenta la versione MPI; unico speedup a 1250 nodi.

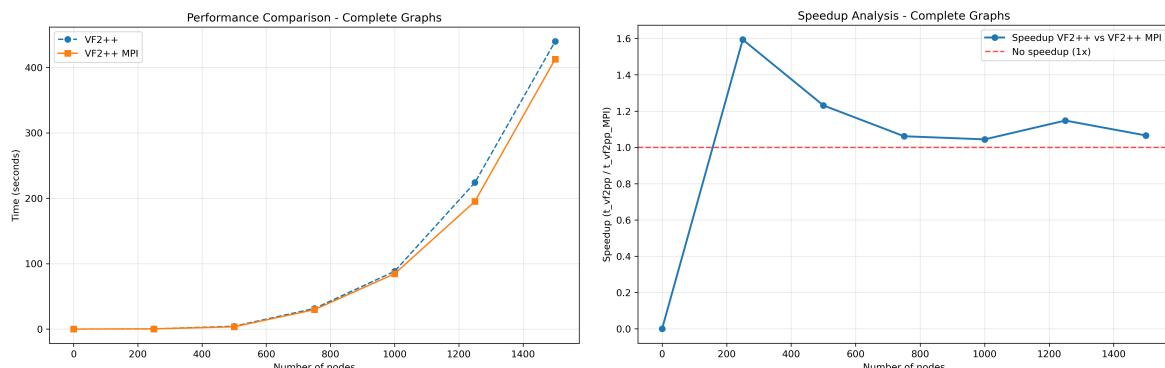


Figura 37: Complete graphs con 4 processi (-O3).

Complete: MPI sempre più veloce ($1.05\text{--}1.6\times$): overhead ammortizzato da lavoro abbondante.

5.5 Tabella Riassuntiva media Speedup 4 processi

Tabella 4: Speedup aggregato MPI vs Seq (4 processi) — GeoMean e Overall per famiglia e ottimizzazione

Set	Categoria	GeoMean(S)	Overall S	Vittorie / Parità / Sconfitte
O0 , p4	Random	0.699	0.687	0 / 0 / 6
O0 , p4	Disconnected	1.043	1.212	3 / 0 / 3
O0 , p4	Complete	0.798	0.674	1 / 1 / 4
O1 , p4	Random	0.622	0.655	0 / 0 / 6
O1 , p4	Disconnected	1.396	1.646	4 / 0 / 2
O1 , p4	Complete	0.815	0.672	1 / 2 / 3
O2 , p4	Random	0.712	0.722	0 / 1 / 5
O2 , p4	Disconnected	0.826	0.836	1 / 1 / 4
O2 , p4	Complete	0.746	0.593	1 / 0 / 5
O3 , p4	Random	0.808	0.907	2 / 0 / 4
O3 , p4	Disconnected	0.810	0.800	1 / 0 / 5
O3 , p4	Complete	1.177	1.086	5 / 1 / 0

5.6 Conclusioni — VF2++ (sequenziale) vs VF2++ con MPI

Definiamo lo *speedup* come $S = T_{\text{seq}}/T_{\text{mpi}}$. Sulla base di tutti i test (O0–O3, 2 e 4 processi) e delle considerazioni progettuali iniziali, emerge un quadro coerente: VF2++ è *intrinsecamente sequenziale* e la parallelizzazione sul solo livello radice fornisce benefici *solo* quando l’insieme iniziale dei candidati $|C_0(u_1)|$ è *sufficientemente numeroso e “diverso”*. In assenza di ciò, l’overhead di comunicazione/sincronizzazione (`MPI_Bcast`, split, stop-signal, `Allreduce`) prevale e $S < 1$.

Dove MPI aiuta (e perché).

- **Grafi densi (*complete*) con `-03`.** Lo spazio di ricerca è costoso, i rami top-level sono numerosi: l’esecuzione parallela su più candidati riduce il *time-to-solution* anche senza early-stop precoce. Nei test con 4 processi e `-03` lo speedup è sistematicamente $S > 1$ (tipicamente 1.05–1.60×), e con 2 processi arriva anche oltre 1.8×.
- **Istanza “fortunata” (più candidati iniziali realmente alternativi).** In alcune configurazioni *random/disconnected* con 2 processi e `-01/-02/-03`, la presenza di candidati multipli e bilanciati consente a un processo di trovare presto il mapping valido: l’early-stop taglia il tempo totale (picchi fino a ∼2.0× nei casi migliori).

Dove MPI penalizza (e perché).

- **Radice “stretta” (spesso $|C_0(u_1)| \approx 1$).** È il caso tipico delle istanze *random* con 4 processi e `-00/-02`, e di molte *disconnected*: il lavoro resta di fatto seriale e l’overhead fa scendere S in fascia 0.5–0.9×.

- **Tempi assoluti piccoli.** Sulle *disconnected* i tempi della sequenziale sono dell’ordine dei millisecondi: *micro-overhead* di comunicazione e sincronizzazione diventano dominanti, dando $S < 1$ nella maggior parte dei punti, con rari picchi pro-MPI quando la partizione è particolarmente favorevole.
- **Quattro processi non “riempiti”.** Quando un solo processo porta il carico utile (profilo per-processo sbilanciato), gli altri restano quasi sempre in attesa: la pipeline di broadcast/split/stop non viene ammortizzata e S cala.

Lettura per famiglia di grafi.

- **Random.** Con 2 processi e `-01/-03` compaiono diversi casi $S > 1$ (fino a $\sim 1.9\text{--}2.0\times$ alle taglie maggiori); con 4 processi, invece, prevale $S \leq 1$ (soprattutto `-00/-02`), salvo sporadici punti pro-MPI con `-03`. *Conclusione:* bene 2 processi ad alte ottimizzazioni su istanze grandi; 4 processi spesso controproducenti.
- **Disconnected.** Andamento *altalenante*: si osservano sia speedup significativi (fino a $\sim 2.1\times$) sia slowdown marcati ($S \sim 0.6\text{--}0.9$). Il risultato dipende dal bilanciamento tra componenti e dal costo assoluto: quando i tempi sono molto piccoli, l’overhead domina. *Conclusione:* potenziale beneficio solo se la partizione iniziale produce rami realmente indipendenti e comparabili.
- **Complete.** Con `-03` MPI è *affidabilmente* migliore: $S > 1$ con 2 e 4 processi su quasi tutte le taglie; con `-00/-02` prevale invece $S < 1$ (rarissimi picchi positivi). *Conclusione:* su grafi densi conviene attivare MPI *insieme* a `-03`; altrimenti è preferibile la sequenziale.

Effetto del livello di ottimizzazione.

- Le ottimizzazioni del compilatore (`-01/-02/-03`) non cambiano la natura del problema, ma *ridimensionano* l’overhead relativo di MPI. Con `-03` lo speedup pro-MPI diventa più frequente dove esiste parallelismo reale (specie sui *complete*); con `-00` l’overhead di comunicazione pesa molto di più e porta spesso a $S < 1$.

Nota interpretativa

- **Limite teorico (Amdahl).** Se la frazione sequenziale f è alta (radice stretta, $|C_0| \approx 1$), lo speedup massimo è

$$S_{\text{tot}} = \frac{1}{f + \frac{1-f}{p}},$$

con $p \in \{2, 4\}$. Per esempio, con $f \approx 0.9$ si ha $S_{\text{tot}} \leq 1.18$ anche in condizioni ideali; in pratica l’overhead di `MPI_Bcast`, `MPI_Iprobe/stop` e `Allreduce` riduce ulteriormente S , portando a parità o slowdown.

- **“Casualità” del branching iniziale.** Speedup $S > 1$ emergono *solo* quando $|C_0|$ è > 1 e i candidati sono realmente *diversi/bilanciati*: processi distinti esplorano rami alternativi ed è probabile un early-stop precoce. Con node ordering discriminativo, però, su *random/disconnected* $|C_0|$ tende spesso a 1: il beneficio atteso è nullo e l’overhead domina.
- **Trascurabilità degli speedup MPI nel quadro globale.** Rispetto a VF2 (NetworkX), VF2++ sequenziale ottiene guadagni enormi ($\sim 10^2\text{--}10^3\times$). Variazioni MPI dell’ordine $0.7\text{--}1.3\times$ sono spesso *trascutibili* sia *relativamente* (spostano poco un gap di $\sim 300\times$) sia *assolutamente* (differenze di millisecondi o microsecondi su istanze già veloci).

6 Conclusioni e considerazioni finali

Questo lavoro ha valutato le prestazioni di **VF2++** rispetto a **VF2** (baseline NetworkX) e l'effetto della parallelizzazione **MPI** applicata a VF2++. I test hanno coperto tre famiglie di grafi (*random*, *disconnessi*, *completi*), varie taglie, livelli di ottimizzazione del compilatore (-00, -01, -02, -03) e due configurazioni di processi (2 e 4). Le metriche primarie sono il *tempo di esecuzione* e lo *speedup* $S = T_{\text{seq}}/T_{\text{MPI}}$; per i riassunti sono stati usati la *media geometrica* degli speedup (GeoMean) e l'*overall* $S = \sum T_{\text{seq}}/\sum T_{\text{MPI}}$. La correttezza dei risultati è stata preservata in tutti i casi (*match* tra soluzioni).

VF2++ vs VF2. La scelta dell'algoritmo è il fattore determinante: **VF2++** risulta sistematicamente *molto* più veloce di VF2, con guadagni che si collocano nell'ordine di grandezza di $\sim 10^2\text{--}10^3\times$ a seconda della famiglia e della taglia. Questa evidenza rende chiaro che l'adozione di VF2++ è la leva primaria di prestazione; le differenze introdotte da MPI, nell'intorno $\sim 0.7\text{--}1.6\times$ nella maggior parte dei casi, sono marginali rispetto al salto ottenuto passando da VF2 a VF2++.

VF2++ (sequenziale) vs VF2++ (MPI). Il comportamento della versione MPI è coerente con la natura *intrinsecamente sequenziale* di VF2++: la parallelizzazione opera al solo livello radice e porta benefici *solo* quando l'insieme dei candidati iniziali $|C_0|$ è *numeroso e diversificato*. In mancanza di ciò, l'overhead di comunicazione/sincronizzazione (**MPI_Bcast**, partizionamento, stop asincrono, **Allreduce**) prevale e $S \leq 1$. In sintesi:

- **Random.** Con **2 processi** e ottimizzazioni alte (-03) compaiono diversi casi $S > 1$ alle taglie grandi (anche $\sim 2\times$); con **4 processi** prevalgono *parità/slowdown* salvo pochi casi favorevoli con -03. Il numero di candidati al radice è spesso ridotto \Rightarrow poco lavoro parallelizzabile.
- **Disconnessi.** Con **2 processi** il vantaggio è frequente (GeoMean/Overall > 1 su più livelli di ottimizzazione); con **4 processi** il quadro diventa altalenante: bene con -01, spesso in parità o sotto con -02/-03, perché i tempi assoluti sono piccoli e i *micro-overhead* pesano.
- **Completi.** Con -03 la versione MPI è *affidabilmente migliore* sia con 2 sia con 4 processi (GeoMean/Overall > 1 in modo consistente), grazie al costo elevato dell'esplorazione e ai molti candidati top-level; con -00/-02 prevale invece lo *slowdown*.

Lettura degli indicatori aggregati. La **GeoMean** sintetizza in modo robusto i rapporti istanza-per-istanza, mentre l'**Overall** S evidenzia l'effetto sulle istanze più costose. Divergenze fra i due (ad es. Overall > 1 ma GeoMean ≈ 1) indicano che i benefici MPI si concentrano proprio dove il costo è alto, mentre su molte istanze rapide prevalgono parità o piccoli rallentamenti fisiologici.

Elenco delle figure

1	VF2 Alghoritm	8
2	Random graphs: tempi e speedup con O0.	25
3	Disconnected graphs: tempi e speedup con O0.	26
4	Complete graphs: tempi e speedup con O0.	26
5	Random graphs: tempi e speedup con O1.	27
6	Disconnected graphs: tempi e speedup con O1.	27
7	Complete graphs: tempi e speedup con O1.	27
8	Random graphs: tempi e speedup con O2.	28
9	Disconnected graphs: tempi e speedup con O2.	28
10	Complete graphs: tempi e speedup con O2.	28
11	Random graphs: tempi e speedup con O3.	29
12	Disconnected graphs: tempi e speedup con O3.	29
13	Complete graphs: tempi e speedup con O3.	29
14	O0, 2 processi: tempi e speedup su grafi <i>random</i> .	31
15	O0, 2 processi: tempi e speedup su grafi <i>disconnected</i> .	31
16	O0, 2 processi: tempi e speedup su grafi <i>complete</i> .	31
17	O1, 2 processi: tempi e speedup su grafi <i>random</i> .	32
18	O1, 2 processi: tempi e speedup su grafi <i>disconnected</i> .	32
19	O1, 2 processi: tempi e speedup su grafi <i>complete</i> .	33
20	O2, 2 processi: tempi e speedup su grafi <i>random</i> .	33
21	O2, 2 processi: tempi e speedup su grafi <i>disconnected</i> .	33
22	O2, 2 processi: tempi e speedup su grafi <i>complete</i> .	34
23	Random graphs con 2 processi (-O3).	34
24	Disconnected graphs con 2 processi (-O3).	34
25	Complete graphs con 2 processi (-O3).	35
26	O0, 4 processi: tempi e speedup su grafi <i>random</i> .	36
27	O0, 4 processi: tempi e speedup su grafi <i>disconnected</i> .	36
28	O0, 4 processi: tempi e speedup su grafi <i>complete</i> .	36
29	O1, 4 processi: tempi e speedup su grafi <i>random</i> .	38
30	O1, 4 processi: tempi e speedup su grafi <i>disconnected</i> .	38
31	O1, 4 processi: tempi e speedup su grafi <i>complete</i> .	38
32	O2, 4 processi: tempi e speedup su grafi <i>random</i> .	39
33	O2, 4 processi: tempi e speedup su grafi <i>disconnected</i> .	39
34	O2, 4 processi: tempi e speedup su grafi <i>complete</i> .	39
35	Random graphs con 4 processi (-O3).	40
36	Disconnected graphs con 4 processi (-O3).	40
37	Complete graphs con 4 processi (-O3).	40

Riferimenti bibliografici

- [1] Cordella, L. P., Foggia, P., Sansone, C., & Vento, M. (2004). *A (sub) graph isomorphism algorithm for matching large graphs*. IEEE transactions on pattern analysis and machine intelligence, 26(10), 1367-1372.
- [2] Jüttner, A., & Madarasi, P. (2018). *VF2++-An improved subgraph isomorphism algorithm*. Discrete Applied Mathematics, 242, 69-81.
- [3] Foggia, P., Sansone, C., & Vento, M. (2001). *A performance comparison of five algorithms for graph isomorphism*. Proceedings of the 3rd IAPR TC-15 workshop on graph-based representations in pattern recognition, 188-199.