# HPC project

COO & CSR for sparce matrix multiplication

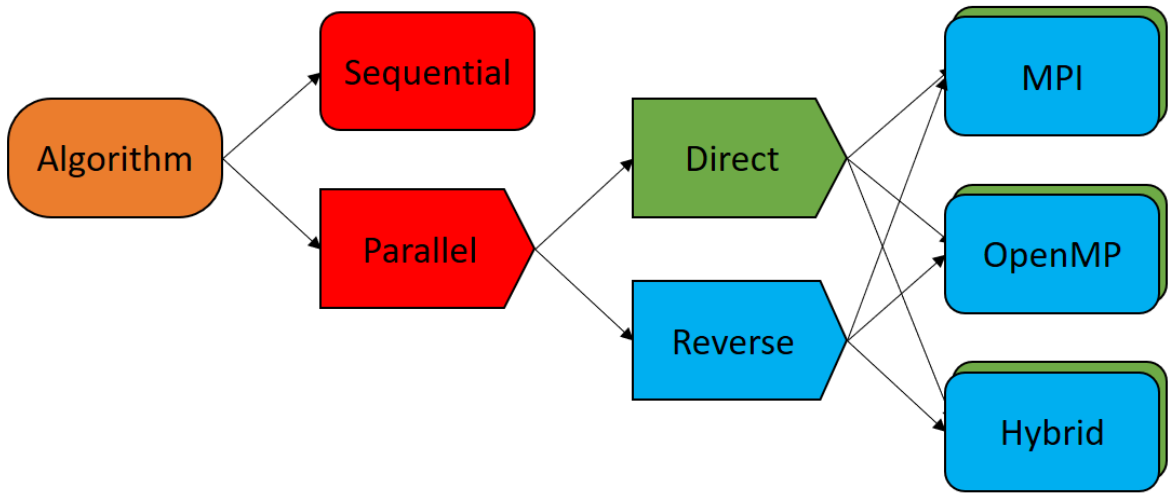
## Overview:

In questa parte presentiamo lo studio sugli algoritmi per la moltiplicazione di matrici sparse, ovvero matrici aventi la maggior parte degli elementi nulli. Gli algoritmi utilizzati si differenziano per il formato delle matrici (COO e CSR) e le strutture utilizzate per l'implementazione (array o struct):

- **COO con struct**
- **COO con array**
- **CSR con array**

Il formato COO prevede il salvataggio di tutte le informazioni di ogni elemento, quindi riga, colonna e valore. Per questo motivo sono possibili più strade, in particolare analizzeremo l'utilizzo di tre array distinti e un array di strutture contenente i tre valori. Nel formato CSR (o CRC) invece al posto di salvare tutte le righe (o colonne) di ogni valore, questi sono ordinati per righe in maniera da permetterci di poter salvare solo gli indici di inizio e fine di ogni riga senza perdere informazioni. Proprio per questo motivo utilizzeremo solo 3 array e non una rappresentazione con strutture in quanto è inappropriata.

Per ognuno di questi algoritmi è stata sviluppata una serie di implementazioni, risultante dalla combinazione di due approcci, modalità di suddivisione dei dati per la moltiplicazione e tipi di parallelizzazione utilizzati. In particolare quindi abbiamo per ogni algoritmo una versione sequenziale e 6 versioni parallele.




## Direct e Reverse:

La modalità Direct è quella classica, nella quale, dato due matrici A e B, ad ogni processo viene inviata una parte delle **righe di A** e **tutta B**. Dalla quale poi ogni processo potrà ricavare una parte delle righe della matrice finale C, e dopo che tutte queste parti sono state aggregate otterremo la matrice finale completa.

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \quad B = \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix} \quad C = \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{bmatrix}$$

La modalità Reverse invece è il risultato del tentativo di minimizzare la duplicazione dei dati nella fase di distribuzione e eliminare i confronti necessari nella fase di moltiplicazione della modalità precedente. In questo caso infatti ogni processo otterrà una parte delle **colonne di A** e delle **righe di B**, così facendo non ci sono dati duplicati, il contro è che ogni processo otterrà, dopo la moltiplicazione, una matrice delle stesse dimensioni di quella finale, **Ci**. E per ottenere C dobbiamo sommare tutti questi risultati parziali.

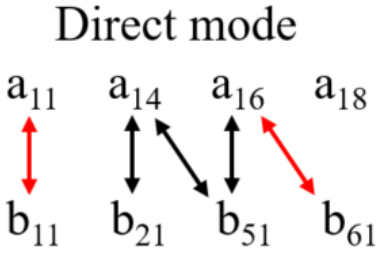
$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \quad B = \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix} \quad C = \Sigma_i C_i = \Sigma \begin{bmatrix} c_{i11} & c_{i12} & c_{i13} & c_{i14} \\ c_{i21} & c_{i22} & c_{i23} & c_{i24} \\ c_{i31} & c_{i32} & c_{i33} & c_{i34} \\ c_{i41} & c_{i42} & c_{i43} & c_{i44} \end{bmatrix}$$

## Pro e contro (general case):

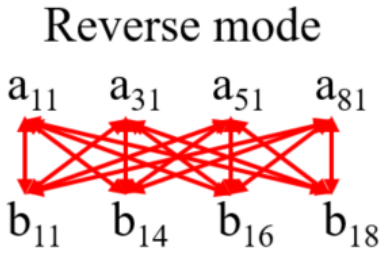
| **Mode:** | **Direct** | **Reverse** |
|---|---|---|
| Distribution phase | Data duplication | No data duplication |
| Multiplication phase | Comparisons needed | No comparisons needed |
| Aggregation phase | Simple aggregation | Sum of matrices needed |

Mentre tutte queste parti influenzano il tempo di esecuzione, quella più impattante è sicuramente la necessità o meno di fare confronti durante la moltiplicazione. Per capire perché questo sia il caso analizziamo i due meccanismi, utilizzeremo come esempio la moltiplicazione di due vettori sparsi.

Nel caso **Direct** siccome i vettori sono sparsi per trovare le corrette coppie di elementi tra cui computare i prodotti dobbiamo scorrere i due vettori e confrontare tutti gli indici, nell'esempio affianco le frecce rappresentano i confronti effettuati, se la coppia è corretta sono rosse, altrimenti nere.



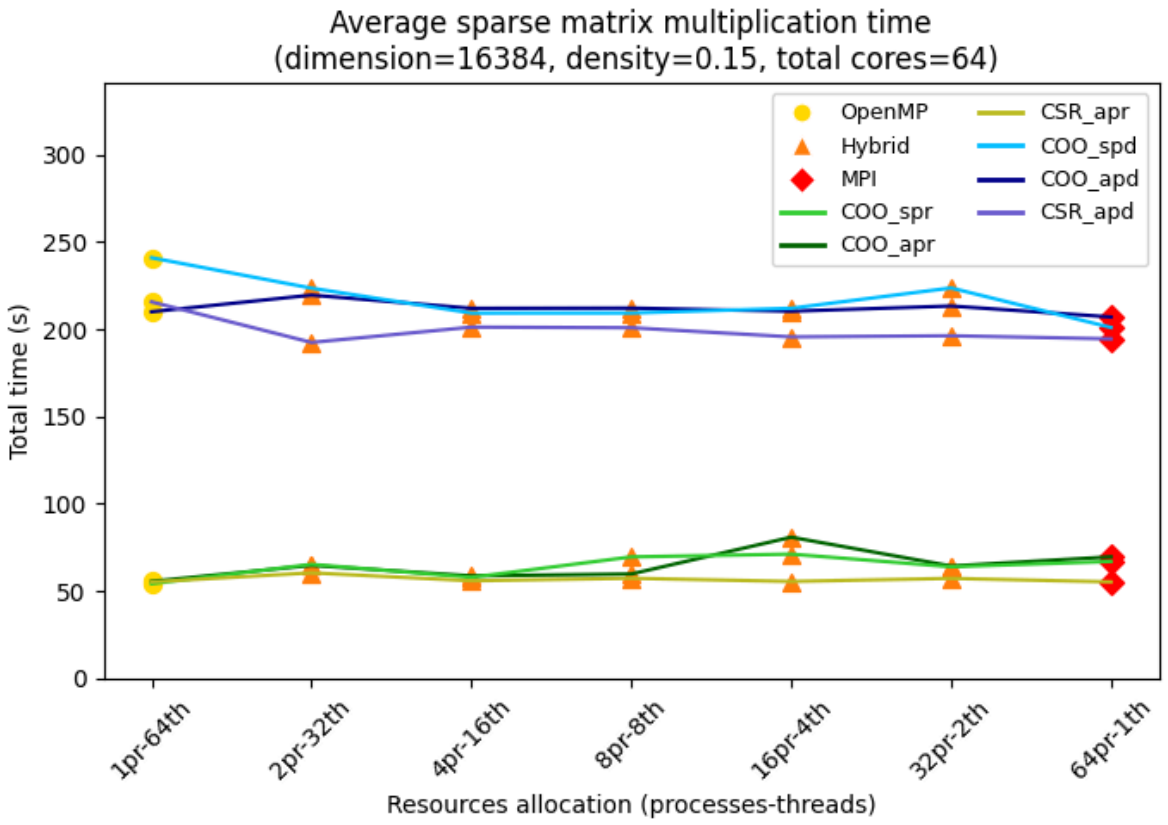

Nel caso **Reverse** invece, siccome i due vettori rappresentano una colonna di A e una riga di B, possiamo direttamente computare il prodotto tra ogni coppia di elementi. Permettendoci non solo di evitare **confronti** inutili, ma addirittura di **rimuoverli totalmente**.



Ricordo infine che **il numero totale di frecce rosse è lo stesso** dato due matrici, rappresentano infatti tutti i prodotti necessari per la computazione del risultato. Proprio per questo motivo, nella modalità Direct, per compensare le poché frecce rosse per iterazione **ogni processo svolgerà più iterazioni**, da qui la necessità di avere B nella sua interezza.

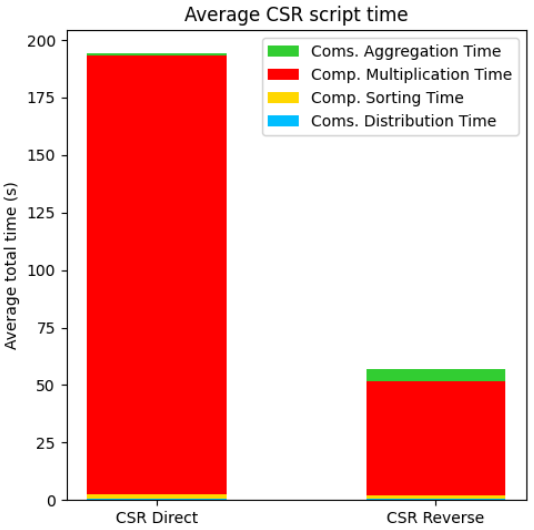
## Comparazione delle prestazioni

Per confrontare i vari programmi usiamo due matrici quadrate sparse con dimensione **16384** e densità di elementi non nulli pari al **15%**. Le risorse utilizzate sono **64** cores, suddivise su processi e thread. Confrontiamo ora le tempistiche dei nostri 3 algoritmi, ognuno con le sue 2 modalità, Direct e Reverse. Ciascuno di essi utilizzerà il tipo di parallelizzazione coerente con la distribuzione delle risorse utilizzata. Il tempo totale viene calcolato sulla media di dieci esecuzioni, e per irrobustire i risultati è stata usata l'opzione "**place=pack:excl**" che fa sì che tutti i processi siano sullo stesso nodo, quindi fisicamente vicini, e che non vi siano programmi di altri utenti che potrebbero influenzare i risultati. Nel grafico vengono usate notazioni contratte, ad esempio "COO struct parallel reverse" è abbreviato in "COO_spr".



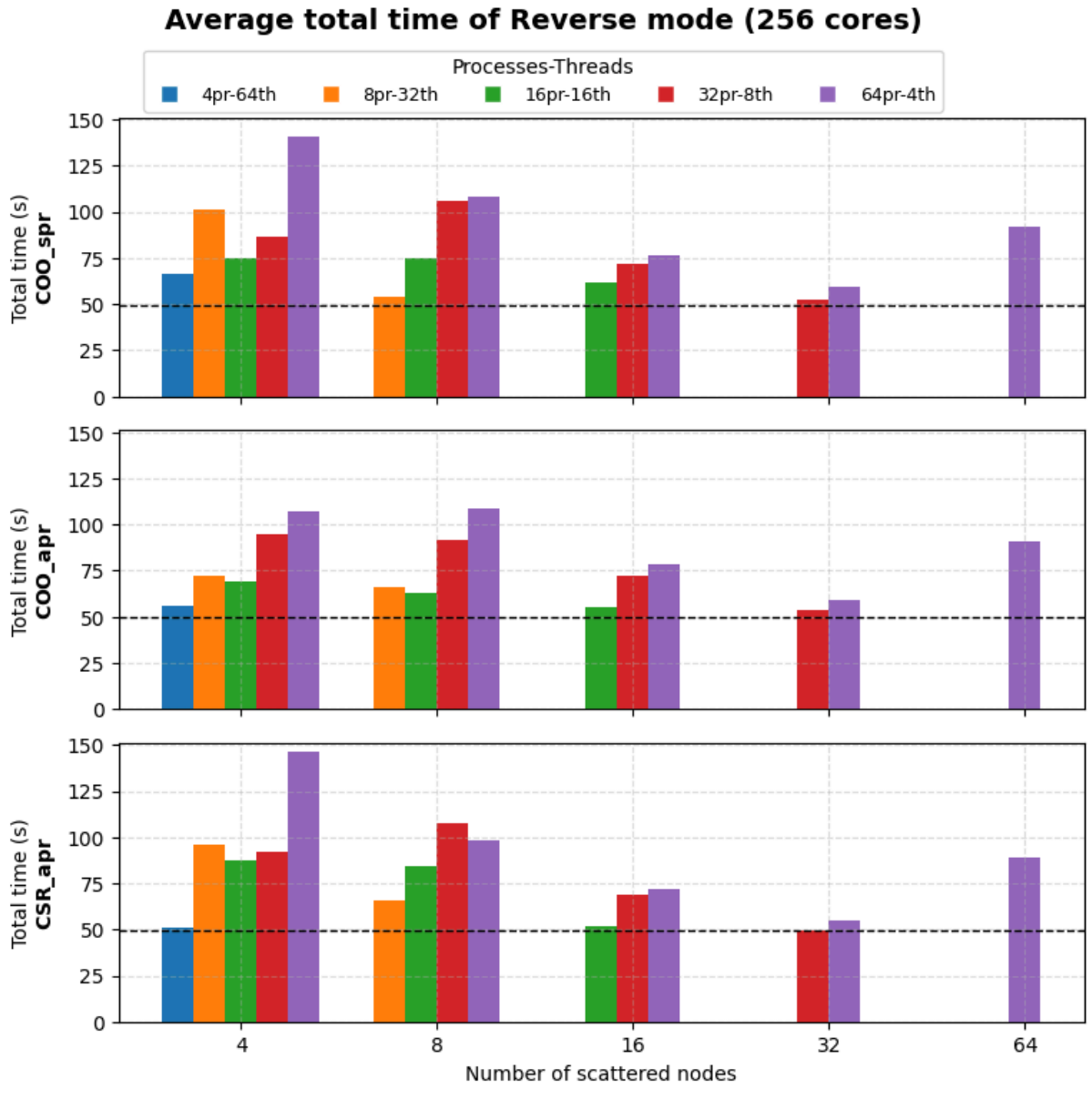

Dal grafico è evidente la differenza in prestazioni tra le modalità "**Direct**" e "**Reverse**". Analizzando i tempi delle singole sezioni possiamo vedere che cosa causi questo netto divario. Possiamo quindi notare tre cose:

- il tempo di distribuzione è pressoché lo stesso, questo indica che la duplicazione dei dati nella modalità Direct, dovuta al broadcast di una matrice non impatta molto sulle prestazioni.
- Il tempo di aggregazione è maggiore nella modalità Reverse, perché è necessario svolgere la somma di innumerevoli matrici.
- Il tempo di moltiplicazione è quello che comporta la differenza maggiore. Questo avviene perché, come spiegato in precedenza, la moltiplicazione nella modalità Direct necessità l'utilizzo di una grande quantità di confronti mentre la modalità Reverse non ne usa nessuno.



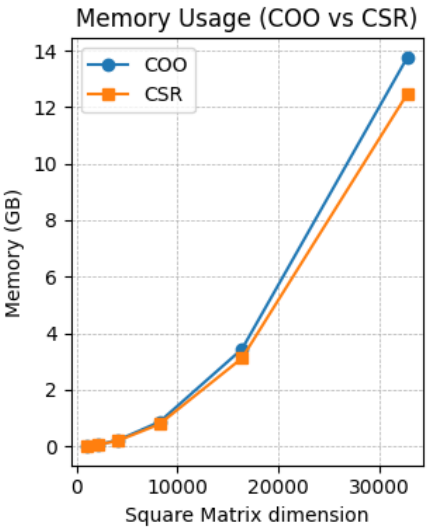

## Script migliore

A questo punto, possiamo concentrarci solo sugli script relativi alla modalità Reverse. Ora confrontiamo le prestazioni in base ai nodi utilizzati, forzando quindi ogni chunk ad essere allocato su un nodo diverso, tramite l'opzione "**place=scatter**". Inoltre come in precedenza utilizzeremo varie combinazioni di processi e threads, su un totale di 256 cores.



Dal grafico possiamo notare che anche variando la distribuzione delle risorse sui nodi tutti i programmi hanno prestazioni simili. Emergono inoltre alcuni pattern interessanti:

- Dato un certo numero di nodi, in generale, la distribuzione migliore è quella di un solo processo MPI per nodo e il resto threads.
- Le prestazioni hanno un andamento parabolico, avendo il minimo a 32 nodi, per mancanza di risorse ci siamo fermati a 64 nodi ma è probabile che questo andamento continui con l'aumentare dei nodi; questo sottolinea l'importanza di avere un corretto bilanciamento tra nodi, processi MPI e threads.
-




In conclusione il formato CSR è quello che statisticamente ha le prestazioni migliori. Esso viene comunque sempre preferito rispetto al formato COO per la memoria utilizzata. infatti una matrice sparsa in formato COO richiede l'allocazione di tre vettori, di dimensione pari al numero di elementi non nulli (nnz). Nel formato CSR (o CSC) invece, al posto del vettore delle righe (o colonne) abbiamo un vettore di dimensione pari al numero di righe più uno, rappresentate gli offset di ogni riga, risparmiando così memoria, Il divario tra i due aumenta con l'aumentare delle dimensioni delle matrice.

## Data Dependency

Analizziamo ora le dipendenze riscontrate, in particolare della versione CSR, in quanto rappresentativa di tutte le dipendenze riscontrate anche negli altri due algoritmi. Abbiamo quindi quattro sotto gruppi di dipendenze, ogni gruppo derivante da un ciclo diverso.

| Data Dependency Analysis (CSR) | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Memory Location | Earlier Statement | | | Later Statement | | | Loop-carried? | Kind of Dataflow |
| | Line | Iteration | Access | Line | Iteration | Access | | |
| col_counts[] | 101 | i | read | 101 | i | write | no | anti |
| col_counts[] | 101 | i | read | 101 | i + 1 | write | yes | anti |
| col_counts[] | 101 | i | write | 101 | i + 1 | read | yes | flow |
| col_counts[] | 101 | i | write | 101 | i + 1 | write | yes | output |
| col_ptr_csc[i] | 106 | i | write | 106 | i + 1 | read | yes | flow |
| col_counts[col] | 113 | i | read | 113 | i | write | no | anti |
| col_counts[col] | 113 | i | read | 113 | i + 1 | write | yes | anti |
| col_counts[col] | 113 | i | write | 113 | i + 1 | read | yes | flow |
| col_counts[col] | 113 | i | write | 113 | i + 1 | write | yes | output |
| C[...] | 150 | i | read | 150 | i | write | no | anti |
| C[...] | 150 | i | read | 150 | i + 1 | write | yes | anti |
| C[...] | 150 | i | write | 150 | i + 1 | read | yes | flow |
| C[...] | 150 | i | write | 150 | i + 1 | write | yes | output |

Per risolvere queste dipendenze abbiamo utilizzato diverse strategie, per i gruppi 1 e 3 è stata semplicemente utilizzata una direttiva “atomic”, per il gruppo 2 non è stato fatto nulla in quanto aveva un impatto trascurabile sulle prestazioni ed era ragionevole supporre che la rimozione della dipendenza potesse addirittura peggiorare la situazione. Per l’ultimo gruppo, relativo alla moltiplicazione delle matrici il codice seriale si presenta così:

```
int k, i, j;
for(k=0; k<n; k++){
    for(i=A_col_ptr[k]; i<A_col_ptr[k+1]; i++){
        for(j=B_row_ptr[k]; j<B_row_ptr[k+1]; j++){
            C[A_row[i]*n + B_col[j]] += A_value[i] * B_value[j];
        }
    }
}
```

Il problema si verifica nella scrittura nell’array C, infatti è possibile che due threads manipolino contemporaneamente i dati rendendoli inconsistenti. Per risolvere questo problema abbiamo due strade possibili:

- Parallelizzare il loop esterno, che ci permette di inizializzare i threads una volta sola, e utilizzare una direttiva “atomic” per serializzare l’accesso alla matrice, evitando così collisioni. Così facendo però introduciamo un collo di bottiglia non indifferente in quanto la scrittura in memoria è l’operazione più lenta e con questa strada si serializza un numero di operazioni che va **come il cubo di N**.
- Parallelizzare il secondo ciclo. In questo modo evitiamo di serializzare l’accesso alla matrice, ma in cambio dobbiamo inizializzare la sezione parallela più volte che introduce un overhead non indifferente ma che cresce **come N**. Questo approccio è possibile per una “proprietà” di questa modalità, superato il ciclo esterno infatti i due rimanenti si occupano di ciclare su una colonna di A e riga di B, moltiplicando ogni coppia di elementi. Per far si che si verifichi una collisione è necessario che vi siano due coppie distinte che abbiano **contemporaneamente** la stesso valore della riga dell’elemento di A e della colonna dell’elemento di B, questi due valori infatti sono gli indici della cella di destinazione. Ciò però non è possibile poiché, su una colonna di A non ci possono essere due elementi sulla stessa riga, allo stesso modo non ci sono due elementi su una riga di B sulla stessa colonna.

Qui di seguito sono mostrate entrambe le implementazioni, in verde la prima e in giallo la seconda. Dopo qualche test abbiamo verificato che la seconda opzione, come era da aspettarsi, è quella con le prestazioni migliori.

```
int k;
#pragma omp parallel for num_threads(num_threads)
for(k=0; k<n; k++){
    int i;
    #pragma omp parallel for num_threads(num_threads)
    for(i=A_col_ptr[k]; i<A_col_ptr[k+1]; i++){
        int j;
        for(j=B_row_ptr[k]; j<B_row_ptr[k+1]; j++){
            int index = A_row[i]*n + B_col[j];
            double val = A_value[i] * B_value[j];
            #pragma omp atomic
            C[index] += val;
        }
    }
}
```

## Speedup e Efficienza

Ci concentriamo ora ad analizzare in maniera più approfondita le prestazioni del programma migliore, formato CSR, usando array come rappresentazione e modalità Reverse. Siccome non è emersa una configurazione ottimale dallo studio precedente procediamo con l'analisi di tutte quante, con massimo di 64 cores complessivi. Abbiamo inoltre utilizzato entrambe le opzioni di distribuzione dei chunk viste in precedenza "pack:excl" e "scatter", che dovrebbero essere rispettivamente la migliore e la peggiore opzione tra quelle disponibili.

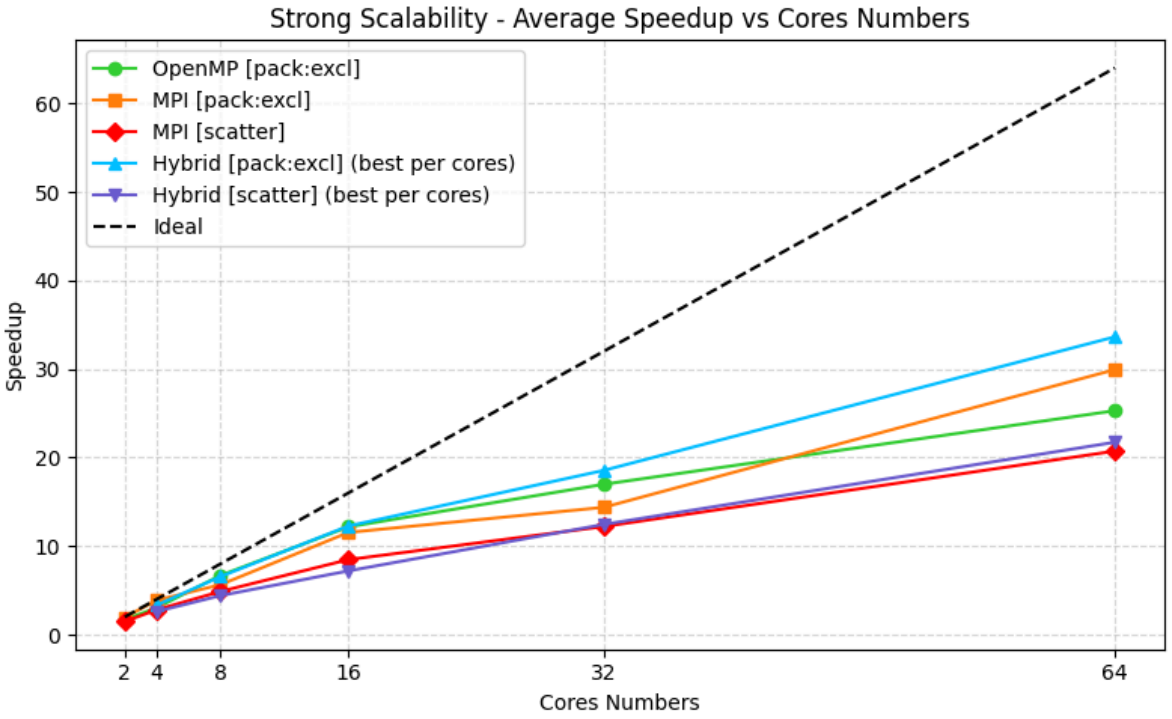
| Average Speedup CSR Array Parallel Reverse | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Place Option** | **Cores** | **1** | **2** | **4** | **8** | **16** | **32** | **64** |
| Pack Exclusive | OpenMP | 1.00 | 1.70 | 3.03 | 6.69 | 12.17 | 16.98 | 25.29 |
| | MPI | 1.00 | 1.95 | 3.89 | 5.64 | 11.54 | 14.38 | 29.93 |
| | Hybrid (2 pr) | 1.00 | - | 3.33 | 5.45 | 11.93 | 16.68 | 33.63 |
| | Hybrid (4 pr) | 1.00 | - | - | 6.56 | 11.11 | 16.90 | 29.28 |
| | Hybrid (8 pr) | 1.00 | - | - | - | 12.27 | 18.55 | 30.05 |
| | Hybrid (16 pr) | 1.00 | - | - | - | - | 15.68 | 31.31 |
| | Hybrid (32 pr) | 1.00 | - | - | - | - | - | 28.01 |
| Scatter | MPI | 1.00 | 1.48 | 2.81 | 4.88 | 8.47 | 12.21 | 20.73 |
| | Hybrid (2 pr) | 1.00 | - | 2.62 | 4.41 | 7.20 | 12.45 | 21.73 |
| | Hybrid (4 pr) | 1.00 | - | - | 3.90 | 7.02 | 12.00 | 20.07 |
| | Hybrid (8 pr) | 1.00 | - | - | - | 6.76 | 11.32 | 18.04 |
| | Hybrid (16 pr) | 1.00 | - | - | - | - | 11.26 | 17.99 |
| | Hybrid (32 pr) | 1.00 | - | - | - | - | - | 20.47 |

Allo stesso modo ora possiamo valutare l'efficienza di ogni combinazione. In particolare nella tabella sono evidenziate le celle relative all'efficienza migliore per ogni colonna.

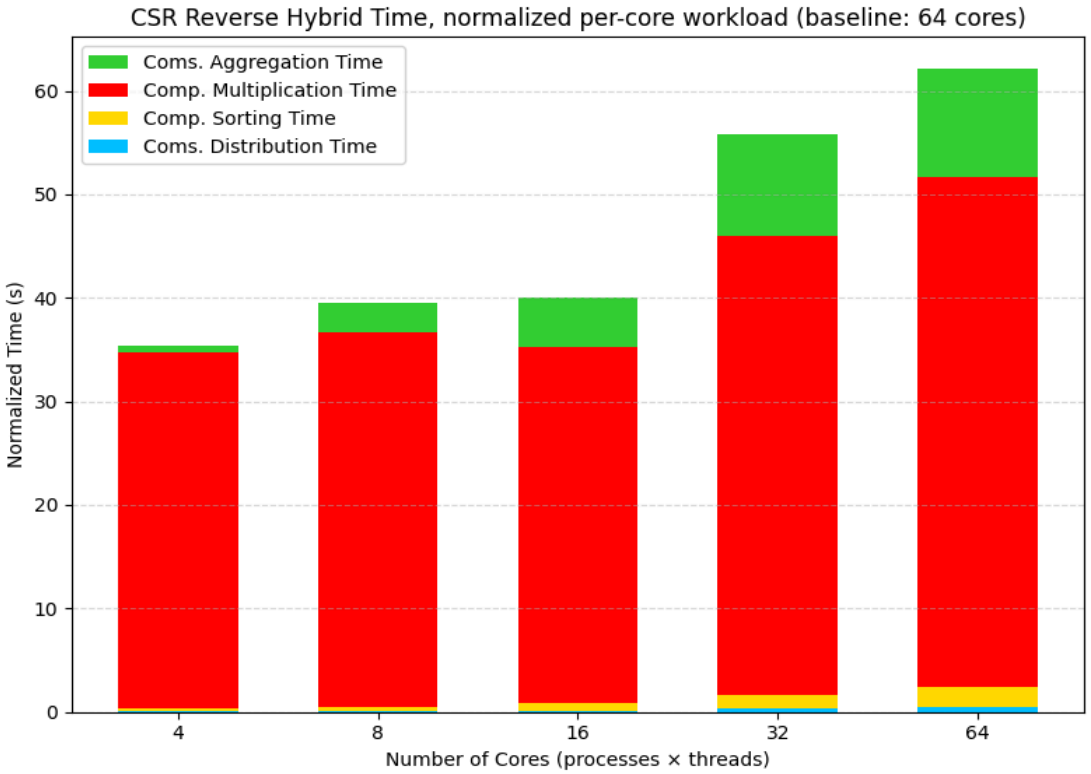
| Average Efficiency CSR Array Parallel Reverse | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Place Option** | **Cores** | **1** | **2** | **4** | **8** | **16** | **32** | **64** |
| **Pack Exclusive** | **OpenMP** | **1.00** | **0.85** | **0.76** | **0.84** | **0.76** | **0.53** | **0.40** |
| | **MPI** | **1.00** | **0.98** | **0.97** | **0.70** | **0.72** | **0.45** | **0.47** |
| | **Hybrid (2 pr)** | **1.00** | **-** | **0.83** | **0.68** | **0.75** | **0.52** | **0.53** |
| | **Hybrid (4 pr)** | **1.00** | **-** | **-** | **0.82** | **0.69** | **0.53** | **0.46** |
| | **Hybrid (8 pr)** | **1.00** | **-** | **-** | **-** | **0.77** | **0.58** | **0.47** |
| | **Hybrid (16 pr)** | **1.00** | **-** | **-** | **-** | **-** | **0.49** | **0.49** |
| | **Hybrid (32 pr)** | **1.00** | **-** | **-** | **-** | **-** | **-** | **0.44** |
| **Scatter** | **MPI** | **1.00** | **0.74** | **0.70** | **0.61** | **0.53** | **0.38** | **0.32** |
| | **Hybrid (2 pr)** | **1.00** | **-** | **0.65** | **0.55** | **0.45** | **0.39** | **0.34** |
| | **Hybrid (4 pr)** | **1.00** | **-** | **-** | **0.49** | **0.44** | **0.37** | **0.31** |
| | **Hybrid (8 pr)** | **1.00** | **-** | **-** | **-** | **0.42** | **0.35** | **0.28** |
| | **Hybrid (16 pr)** | **1.00** | **-** | **-** | **-** | **-** | **0.35** | **0.28** |
| | **Hybrid (32 pr)** | **1.00** | **-** | **-** | **-** | **-** | **-** | **0.32** |

## Scalabilità forte e debole

Analizziamo ora la scalabilità. Per la scalabilità forte possiamo partire direttamente dalla tabella dello speedup, ottenendo il seguente grafico.



Appare evidente che il programma non scali molto bene, questo aspetto emerge anche dalla tabella dell'efficienza dove essa cala con l'aumentare dei core. Per entrare più nello specifico nel che cosa causi questo calo di prestazioni vediamo di seguito le tempistiche di ogni core, normalizzate per il carico di lavoro, dell'implementazione Hybrid.




Vediamo quindi che, dal lato MPI il tempo di aggregazione aumenta in quanto devono essere sommate sempre più matrici; mentre dal lato OpenMP aumenta il tempo di computazione per l'overhead menzionato prima. Per le stesse ragioni anche l'analisi della scalabilità debole non è delle migliori.