# Prodotto Parallelo tra Matrici

Sistemi di Calcolo Parallelo e Applicazioni

Alessandro Lioi
0333693

***Abstract*—Lo scopo di questo documento è quello di descrivere lo sviluppo e le scelte progettuali effettuate nella realizzazione di un nucleo di calcolo parallelo per il prodotto tra due matrici: $C \leftarrow C + AB$**

## I. Introduzione

Il progetto richiede lo sviluppo di un nucleo di calcolo parallelo in grado di calcolare il prodotto $C \leftarrow C + AB$ con $A \in \mathbb{R}^{m \times k}$ e $B \in \mathbb{R}^{k \times n}$, quindi con $C \in \mathbb{R}^{m \times n}$.
La realizzazione di questo nucleo di computazione è stato effettuato tramite l'uso delle librerie **MPI** e **OpenMP** in modo tale da poter confrontare i due modelli di programmazione parallela su CPU:

- scambio di messaggi su processi che possono essere eseguiti su nodi diversi dello stesso cluster (MPI)
- modello di processamento a memoria condivisa, tramite l'uso di thread (OpenMP)

Inoltre è stata eseguita una ulteriore analisi sull'utilizzo di MPI e OpenMP in un approccio ibrido che permette di sfruttare le capacità di parallelismo tra più nodi, con MPI e all'interno di ogni singolo nodo, tramite OpenMP.
Gli indici prestazionali analizzati sono:

- FLOPS $= \frac{2 \cdot mnk}{T}$: operazioni in aritmetica floating-point per secondi
- Speed-Up $= \frac{T_s}{T_p}$
- Relative Error $= \frac{||C_s - C_p||}{||C_s||}$ per quantificare la differenza tra la matrice *vera*, calcolata tramite una computazione seriale e quella *approssimata*, calcolata tramite computazione parallela; è stata utilizzata la norma di **Frobenius** in quanto più adatta ad effettuare un analisi sull'errore considerando tutti gli elementi della matrice

Il progetto si pone quindi i seguenti obiettivi:

- identificare quale approccio alla computazione parallela permette di avere le performance migliori, minimizzando l'errore relativo
- verificare se un approccio ibrido porta dei benefici anche su nodo singolo

Tutte le versioni usano lo stesso nucleo di calcolo parallelo per il calcolo della matrice risultante $C$; per questo motivo la sua analisi viene rinviata in una sezione apposita dopo aver descritto le varie implementazioni.

## II. MPI

Sono state sviluppate due versioni per MPI che differiscono nella fase di generazione delle matrici iniziali, per rappresentare due possibili scenari realistici:

1) le matrici iniziali sono generate dal processo **root** e distribuite ai vari processo. È questo il caso dell'analisi dei dati o algoritmi centralizzati di machine learning: spesso i dati sono disponibili in formato tabellare/matriciale su un file system, spesso distribuito, che viene acceduto da un unico processo.
2) ogni processo genera la porzione delle matrici iniziali assegnata. Questo è invece il caso di simulazioni nell'ambito scientifico in cui si modella il fenomeno da analizzare tramite delle equazioni, spesso alle derivate parziali: ogni processo usa quindi delle equazioni specifiche per generare la sotto-matrice ad esso assegnata.

In entrambe le varianti, si ha la stessa distribuzione delle matrici e raccolta dei risultati.

### A. Distribuzione delle Matrici

Prendendo ispirazione dalla distribuzione implementata in ScaLAPACK, è stato adottato uno schema **two-dimensional block distribution**. Ciò ha portato quindi alla necessità di posizionare i processi in una topologia a griglia. All'avvio dell'applicazione, viene quindi creata una topologia cartesiana bidimensionale gestita completamente da MPI.
Tenendo conto dell'obiettivo di minimizzare l'errore relativo, si è scelto di adottare questo tipo di distribuzione sulla matrice risultate C. Ciò permette di annullare l'errore relativo, in quanto ogni elemento della matrice C viene calcolato in maniera indipendente da un singolo processo.
Per riuscire ad avere questa situazione, è necessario che ogni processo abbia a disposizione la riga $i$ della matrice $A$ e la colonna $j$ della matrice $B$ il cui prodotto scalare fornisce l'elemento $C_{i,j}$.
Al fine di aumentare le performance e semplificare la comunicazione tra processi, si è scelto di non adottare una distribuzione ciclica, come è implementata in ScaLAPACK, che ha il vantaggio di una distribuzione del carico automatica. Per poter sopperire a questa mancanza, è stato sviluppato un assegnamento personalizzato in grado di distribuire il carico in maniera più *fair* possibile, assegnando ad ogni processo il numero massimo di righe di $A$ e colonne di $B$.
Per il caso delle righe di $A$, ad ogni processo vengono assegnate $\frac{m}{\texttt{grid\_rows}}$ con `grid_rows` numero di processi per riga nella griglia definita da MPI.
Nel caso in cui $m$ non sia perfettamente divisibile per `grid_rows`, ai primi $\texttt{rest} = m \mod \texttt{grid\_rows}$ viene assegnata una riga in più. Nel calcolo di questo assegnamento, si calcola anche a partire da quale riga il processo si dovrà

occupare (indice $i$).
Per il caso delle colonne di $B$ si ha una situazione del tutto analoga.
A seguito di questa distribuzione ogni processo sa quindi quali righe e quali colonne sono ad esso assegnate e, quindi di conseguenza, quale blocco della matrice risultate $C$ dovrà calcolare.
È stata scelta questo tipo di distribuzione delle matrici poichè permette di gestire ogni tipo di dimensione e ogni numero di processi e disposizione di essi.

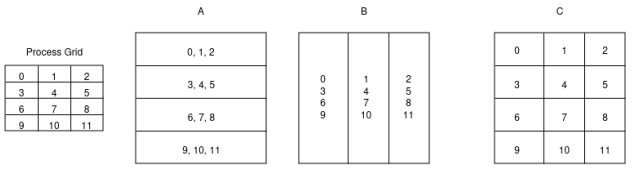


Fig. 1. Esempio di Distribuzione

### B. Generazione e Distribuzione Matrici - Variante 1

Nella prima variante, il processo root si occupa della generazione delle matrici $A$ e $B$ e della loro successiva comunicazione agli altri processi. La generazione avviene tramite numeri pseudo-casuali generati tramite la funzione `rand()` inclusa nella libraria standard con un *seed* predefinito. I valori generati vengono poi divisi per `RAND_MAX` in modo tale da avere un numero floating-point.
A seguito della generazione, la distribuzione delle righe e colonne avviene tramite la definizione di due `MPI_Datatype` di tipo `vector` che permettono di definire un vettore di una certa lunghezza e con un certo *stride*, formato da un numero definito di blocchi.
Per quanto riguarda le righe, il Datatype viene definito come un unico blocco contiguo di $k$ elementi con stride pari a $k$.
Per le colonne invece, si hanno $k$ blocchi di un elemento con stride $n$.
La distribuzione in entrambi i casi avviene tramite la funzione `MPI_Scatterv` che permette di inviare ad ogni processo un numero differente di elementi. Il numero di elementi varia da processo a processo ed è stato identificato calcolando per ogni processo quante righe/colonne deve processare e da quale deve iniziare (stesso algoritmo spiegato nella sezione precedente).

### C. Generazione e Distribuzione Matrici - Variante 2

Nella seconda variante ogni processo si genera la propria sotto-matrice. Poichè lo schema di distribuzione adottato è sempre lo stesso, è necessario che processi sulla stessa riga/colonna nella griglia di MPI, generino le stesse righe/-colonne delle matrici $A$/$B$. Per avere ciò, il processo root genera `grid_rows+grid_cols` *seeds* (tramite la funzione `rand()`) che vengono poi distribuiti ad ogni processo, con la funzione `MPI_Scatter` in base al posizione nella griglia.
Una volta ricevuti i seed, ogni processo genera la sua porzione di righe e colonne.

### D. Utilizzo del Nucleo di Calcolo Parallelo

A seguito della generazione delle matrici, ogni processo può quindi calcolare la sua porzione della matrice risultanante $C$ eseguendo, in maniera seriale, un prodotto tra matrici.
La matrice $C$ viene allocata per intero da ogni processo poichè si presuppone che il calcolo del prodotto tra matrici faccia parte di una computazione più complicata il cui risultato verrà poi utilizzato in seguito.
Grazie al calcolo dell'indice di riga $i$ e colonna $j$ assegnati ad ogni processo, si posiziona il risultato della computazione nel blocco corretto, eliminando la necessità di eseguire un fase di post-processamento dei risultati.

### E. Raccolta Matrice Finale

La ricostruzione della matrice finale avviene tramite l'esecuzione della funzione collettiva `MPI_Allreduce` con operazione di somma `MPI_SUM`. Di particolare nota è l'utilizzo di `MPI_IN_PLACE` come valore di `sendbuf`, che permette di riutilizzare lo stesso buffer di memoria sia per inviare che per ricevere, eliminando quindi la necessità di un buffer temporaneo aggiuntivo.

### F. Controllo Risultato

Il controllo avviene solo dal processo root, a seguito della finalizzazione dell'ambiente di MPI in quanto non è una parte integrante dell'applicazione ma è utile per il calcolo delle metriche di Speed-Up e dell'errore relativo. Il processo root quindi

- esegue il calcolo di $C$ usando un algoritmo seriale
- calcola l'errore relativo
- scrive su un file di output i vari valori delle metriche calcolate

Per quanto riguarda la prima variante, questa parte non richiede computazione aggiuntiva, in quanto le matrici $A$ e $B$ sono già disponibili. Per la seconda variante invece, per evitare una comunicazione aggiuntiva, il processo root genera le matrici $A$ e $B$ complete.

## III. OpenMP

È stata sviluppata una unica versione per OpenMP poichè, basandosi su un modello shared-memory, le matrici sono già disponibili su tutti i thread e non è quindi necessario una fase iniziale di scambio dei dati. Prima di eseguire il nucleo di calcolo è però necessario eseguire la trasposta della matrice $B$ poichè il nucleo di calcolo si aspetta che quest'ultima sia trasposta per motivi di efficienza di cache: Listing 1.
Nel calcolo della trasposta, si esegue in parallelo il primo ciclo, mentre si utilizzano le istruzioni SIMD del processore per vettorializzare l'esecuzione dell'assegnamento.
Come per l'implementazione di MPI, a seguito del calcolo parallelo, si esegue l'implementazione seriale per calcolare le metriche di interesse. Poichè questo controllo deve essere effettuato da entrambe le implementazioni, la funzionalità è stata astratta in una unica funzione.

## IV. MPI + OpenMP

L'implementazione dell'ibrido di MPI + OpenMP è stata immediata in quanto, grazie all'utilizzo di un unico kernel di calcolo, è stato possibile riutilizzare le due versioni implementate per MPI. Sono quindi stati prodotti due nuovi eseguibili, uno per versione di MPI, con l'opzione di compilazione riguardante OpenMP attiva.

## V. Nucleo di Calcolo Parallelo

L'implementazione del kernel parallelo è la seguente: Listing 2. Ogni processo/thread deve calcolare un blocco della matrice $C$ e per fare ciò si deve calcolare un prodotto tra matrici di dimensione ovviamente minori rispetto a quelle originarie. Tenendo conto dell'obiettivo di ottenere le performance migliori, è stato implementato, per ogni processo, un prodotto a blocchi poichè permette di sfruttare al meglio la gerarchia di memoria e le proprietà della cache. Infatti dividendo la matrice a blocchi, si minimizza la necessità di dover andare in RAM perchè si sfrutta la località spaziale e temporale. Inoltre, una volta caricato un blocco in cache, su di esso si possono effettuare più operazioni prima che venga rimosso dalla cache.
È quindi necessario identificare una dimensione del blocco adeguata affinchè tutto ciò sia valido: una linea di cache sull'architettura considerata è di 64 byte e un `float` è formato da 4 byte, si è scelto di usare come dimensione del blocco $\frac{64}{4} = 16$. Questa scelta si è rivelata essere adeguata anche a seguito di test empirici, tranne per un caso che verrà discusso successivamente.
Per migliorare i cache hits sulla sotto-matrice $B$, si assume che sia già trasposta. Questa assunzione decide quindi l'ordine con cui si devono eseguire i cicli: in questo caso $i \to j \to l$. Anche in questo caso, la scelta è supportata da vari test empirici.
La variante 1 di MPI permette di avere la matrice $B$ trasposta in automatico grazie a come viene definito il Datatype delle colonne. Per la variante 2 non si presentano problemi poichè è solamente necessario assicurarsi che la generazione avvenga nello stesso ordine per processi con lo stesso blocco di $B$. Invece per la versione in OpenMP è necessario eseguire la trasposta (Listing 1) prima di chiamare il nucleo. Questa operazione di trasposizione viene eseguita comunque in parallelo tramite l'uso di direttive di OpenMP.
Per facilitare il compilatore nell'uso di istruzione vettoriali si sfruttano due strategie:

- si allocano le matrici con `aligned_alloc` che permette di allocare in maniera allineata; ciò avviene solo se il prodotto delle dimensioni della matrice è multiplo esatto della dimensione del blocco; altrimenti si utilizza l'allocatore classico `malloc`
- si usa la keyword `restrict` per le matrici nella segnatura della funzione che si occupa di calcolare la matrice $C$; questa keyword indica al compilatore che il puntatore non è *aliased* e non è quindi usato in nessun altro contesto.

Per quanto riguarda OpenMP, si usano 2 direttive:

1) direttiva `parallel for` indicando come variabili private le variabili riguardanti l'iterazione corrente dei cicli e condivise le variabili riguardanti le matrici; si utilizza inoltre la direttiva `collapse(3)` per unire i primi 3 cicli (riguardanti i blocchi) in un unico ciclo, permettendo quindi ad ogni thread di occuparsi di un unico blocco
2) direttiva `simd` per eseguire con operazioni vettoriali il ciclo più interno, che si occupa di calcolare il valore effettivo di $C_{ii,jj}$, applicando una riduzione con somma su una variabile di accumulazione

Utilizzando la prima direttiva si cerca di suddividere il lavoro nel miglior modo possibile tra i vari thread. Con la seconda direttiva si cerca di ottenere le performance migliori su ogni thread, usando istruzioni vettoriali.
Il nucleo, inoltre, permette la scrittura sulla matrice $C$ con un offset sulle righe e sulle colonne: ciò è utilizzato dalle implementazioni di MPI in cui ogni processo scrive a partire dalla riga e colonna ad esso assegnata.
In generale, tenendo conto dell'obiettivo di voler ottenere le migliori performance possibili, tutte le versioni sono state compilate con:

- `O3` come livello di ottimizzazione
- `-march=native` per sfruttare al meglio le istruzioni specifiche per la CPU su cui vengono prodotti gli eseguibili
- `-ffast-math` per ridurre i controlli sui valori floating-point a seguito di ogni operazione, assumendo che questi siano sempre validi

## VI. Performance

Tutte le implementazioni sono state eseguite su matrici quadrate e rettangolari, con ogni misurazione eseguita per 4 volte al fine di avere dei risultati non influenzati da altri servizi. Il numero di iterazioni è comunque piccolo per avere dei risultati statisticamente rilevanti ma questo numero è stato scelto poichè la varianza dei risultati ottenuti era molto bassa.
Per le implementazioni di MPI e MPI+OpenMP, si mostrano 2 tipi grafici:

1) GFLOPS e Speed Up al variare del numero di processi considerando come $T$ il solo tempo di esecuzione parallela, e non la fase comunicazione iniziale e finale
2) percentuale di tempo speso in ogni fase, per identificare quale tipo di comunicazione causa il decremento maggiore nelle performance

Per la variante di OpenMP invece vengono mostrati solo i grafici riguardanti i GFLOPS e lo Speed Up in quanto non è necessario avere una comunicazione iniziale e finale.
Per quanto riguarda le matrici rettangolari sono stati analizzati i casi con $K = 32, 64, 128, 156$ e $M = N = 2500, 5000, 10000$

### A. MPI

Poichè si utilizza lo stesso nucleo di calcolo parallelo, entrambe le versioni presentano gli stessi valori di performance

rispetto ai GFLOPS e rispetto allo Speed Up. Per ogni variante verranno analizzate, nelle sotto-sezioni successive, l'impatto della comunicazione.
Per le matrici quadrate (Figure GFLOPS 2 e Figure Speedup 3), si riescono a raggiungere valori oltre ai 120 GFLOPS ($M = N = K = 5000$). Nonostante ciò è presente un calo delle performance per la dimensione massima considerata $M = N = K = 10000$. Ciò potrebbe essere causato da vari fattori:

- nonostante le matrici locali siano state allocate in maniera allineata tramite `aligned_alloc`, l'elevata dimensione di ogni blocco non permette di sfruttare al meglio le proprietà di località della cache
- l'utilizzo di un offset sulle righe e sulle colonne per accedere alla matrice $C$ durante le scritture, causa continui salti di memoria
- la dimensione del blocco scelta causa un ciclo aggiuntivo per ogni dimensione, provocando quindi la necessità di salti in memoria

Per quanto riguarda lo Speed Up ottenuto si riesce a raggiungere il massimo teorico, ovvero il numero di processi paralleli, nel caso $M = N = K = 5000$. Come analizzato precedentemente, si ha un calo dello Speed Up nel massimo caso considerato.
Per quanto riguarda le matrici rettangolari (Figure GFLOPS 4 e Figure Speed Up 5), si ha una situazione del tutto analoga con il picco di performance in $M = N = 5000$ e $K = 128$, arrivando poco sotto i 70 GFLOPS.

*1) MPI - Variante 1:* Per le matrici quadrate, il grafico (Figure 6) mostra come la prima parte della comunicazione, quindi la distribuzione iniziale delle matrici, sia la fase più onerosa in termini di tempo di esecuzione. Ciò era un risultato atteso poichè si ha come bottleneck il processo root con tutte le matrici che hanno una dimensione elevata.
Per quanto riguarda invece il caso delle matrici rettangolari (Figure 7), si ha un trend completamente opposto in cui la seconda comunicazione (ricezione della matrice finale $C$ da parte di tutti i processi) risulta essere la più pesante mentre la prima comunicazione risulta avere overhead quasi nullo. In realtà, analizzando i valori reali per questo tipo di comunicazione, si hanno tempi di comunicazione pressocchè uguali al caso quadrato (in quanto la dimensione finale della matrice è la stessa) e quindi l'uso di questo grafico risulta essere fuorviante per questo tipo di matrici. L'utilizzo della percentuale evidenzia il fatto che i tempi di esecuzione parallela e di invio delle matrici sia molto più basso. Questo risultato è atteso in quanto la dimensione $K$ è molto minore rispetto all'equivalente caso quadrato, e quindi è necessario inviare ed effettuare una moltiplicazione su delle matrici molto più piccole.

*2) MPI - Variante 2:* Sia per le matrici quadrate (Figure 8) che per le matrici rettangolari (Figure 9) il tempo della prima comunicazione è praticamente nullo, in quanto è necessario inviare solo i *seed* di generazione. Per quanto riguarda i tempi della seconda comunicazione, la situazione in entrambi i tipi di matrici è del tutto analoga alla variante 1 e valgono le stesse considerazioni.

### B. OpenMP

Nel caso di OpenMP si ha praticamente la stessa situazione sia per le matrici quadrate (Figure GFLOPS 10 e Figure Speed Up 11) che nel caso delle matrici rettangolari (Figure GFLOPS 12 e Figure Speed Up 13):

- picco delle performance in $M = N = K = 5000$ e $M = N = 5000$ e $K = 128$
- Speed Up massimo per $M = N = 5000$

È interessante notare come ci sia un calo nelle prestazioni nel caso di matrici quadrate con $M = N = K = 512$. Eseguendo il programma con `perf` e `valgrind` con `callgrind` come tool, non si è notato nessuna anomalia particolare riguardante un utilizzazione sbagliata della cache. Provando con varie configurazioni sulla dimensione del blocco, si è notato che con un valore pari a 32 si ha un aumento non indifferente delle performance; con questo valore si mantiene il trend rispetto alle altre dimensioni delle matrici. Poichè per le altre matrici il valore 32 non risulta ottimale, si è deciso di lasciare 16 come dimensione del blocco.
Da questa analisi si evince che ogni dimensione della matrice richiede un'attenta analisi di tutte le variabili utilizzate.

### C. MPI + OpenMP

L'analisi è stata effettuata impostando 2 o 4 processi e un numero di thread variabili in modo tale da avere `process·thread` $= 20$ ovvero il numero di core disponibili sulla CPU di riferimento.
Anche in questo caso per le matrici quadrate (Figure GFLOPS 14 e Figure Speed Up 15) si ha un comportamento analogo al caso MPI, con il picco raggiunto dalla configurazione con 4 processi e 5 threads. Mentre per le matrici rettangolari (Figure GFLOPS 16 e Figure Speed Up 17), le configurazioni con 2 processi riescono ad ottenere performance migliori rispetto a quelle con 4 processi. Ciò potrebbe quindi indicare come, su singola macchina, sia preferibile scalare sui threads con OpenMP rispetto al numero di processi con MPI. Questo è sicuramente dato dall'overhead di comunicazione introdotto dall'utilizzo dei processi, assente con i threads in quanto si lavora con memoria condivisa

### D. Errore Relativo

L'errore relativo in tutte le implementazioni è nullo. Questo risultato è come ci si aspettava poichè la distribuzione delle matrici permette ad ogni processo di calcolare un unico elemento della matrice $C$ in maniera seriale, e quindi si ha sempre lo stesso ordine di esecuzione delle operazioni floating-point.

## VII. Riferimenti

Il codice sorgente è disponibile su GitHub. Le immagini sono presenti in alta qualità nella cartella `output`. Nel file `README` sono presenti le istruzioni per poter compilare tramite `CMake`

```
void matrix_transpose(float *restrict source, float *restrict transposed, int rows, int cols)
  int i, j;
#pragma omp parallel for private(i, j) shared(source, transposed)
  for (j = 0; j < cols; j++)
#pragma omp simd
    for (i = 0; i < rows; i++)
      transposed[j * rows + i] = source[i * cols + j];
}
```

Listing 1: Matrix Transpose

```
void matrix_parallel_mult(float *restrict a, float *restrict b, float *restrict c,
                          int sub_m, int sub_n, int k, int n,
                          int row_offset, int col_offset) {
  int i, j, l, ii, jj, ll;
#pragma omp parallel for private(i, j, l, ii, jj, ll) shared(a, b, c) collapse(3)
  for (i = 0; i < sub_m; i += 16) {
    for (j = 0; j < sub_n; j += 16) {
      for (l = 0; l < k; l += 16) {
        // Block multiplication
        for (ii = i; ii < MIN(i + 16, sub_m); ++ii) {
          for (jj = j; jj < MIN(j + 16, sub_n); ++jj) {
            float sum = 0;
#pragma omp simd reduction(+ : sum)
            for (ll = l; ll < MIN(l + 16, k); ll++)
              sum += a[ii * k + ll] * b[jj * k + ll];
            c[(ii + row_offset) * n + (jj + col_offset)] += sum;
          }
        }
      }
    }
  }
}
```

Listing 2: Matrix Parallel Multiplication

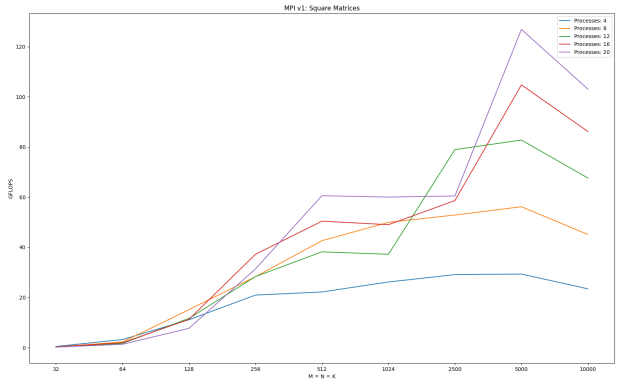


Fig. 2. MPI Square Matrices

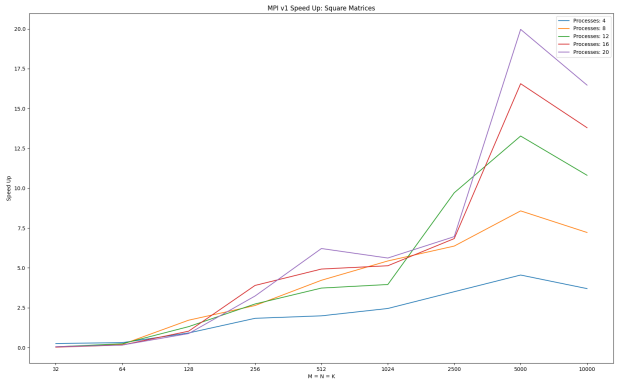


Fig. 3. MPI Speed Up Square Matrices

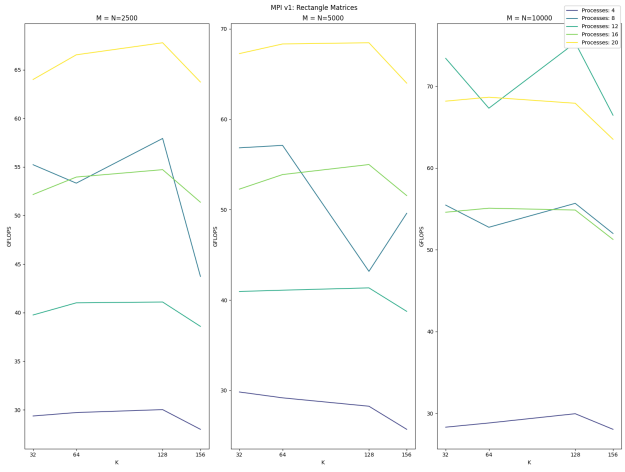


Fig. 4. MPI Rectangle Matrices

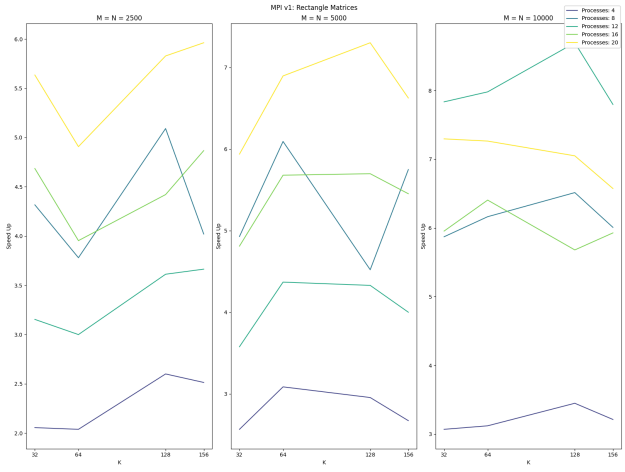


Fig. 5. MPI Speed Up Rectangle Matrices

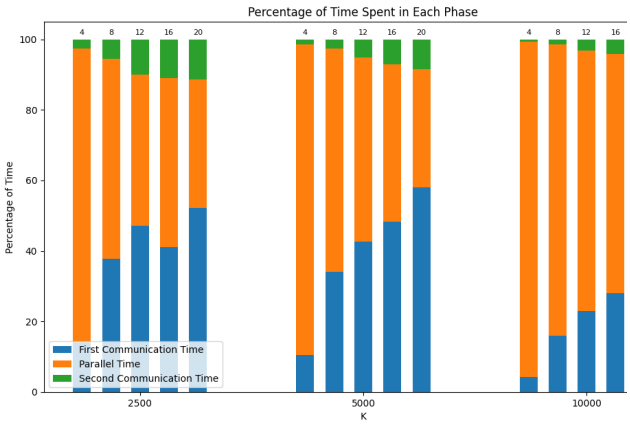


Fig. 6. MPIv1 Time Distribution Square Matrices

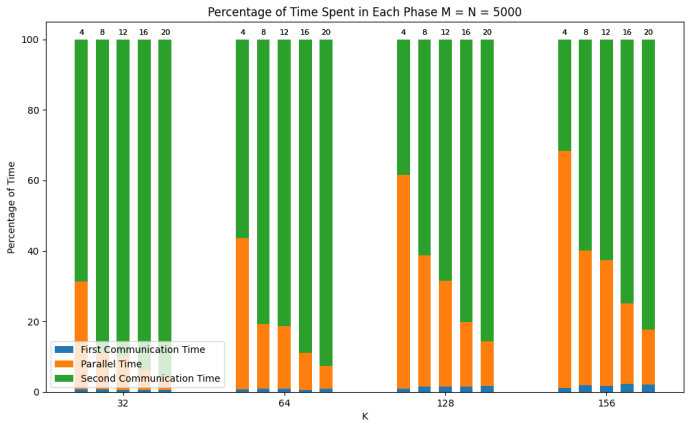


Fig. 7. MPIv1 Time Distribution Rectangle Matrices

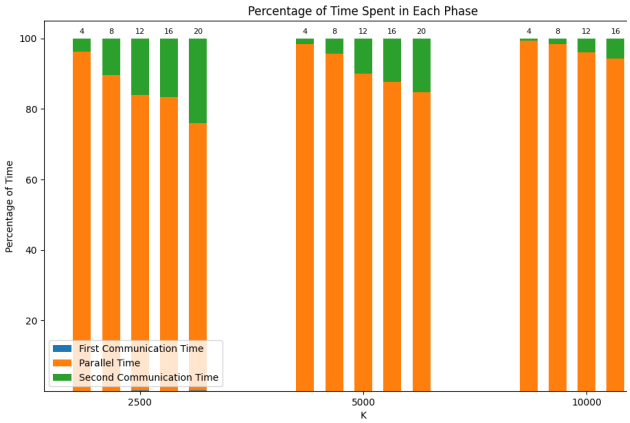


Fig. 8. MPIv2 Time Distribution Square Matrices


Fig. 9. MPIv2 Time Distribution Rectangle Matrices

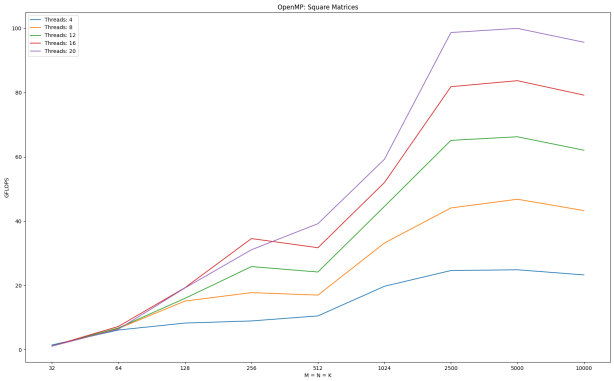


Fig. 10. OpenMP Square Matrices

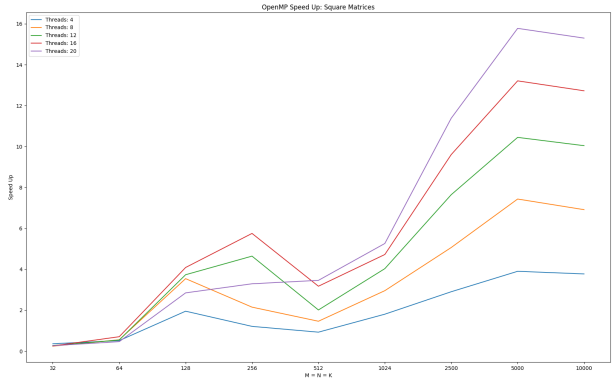


Fig. 11. OpenMP Speed Up Square Matrices

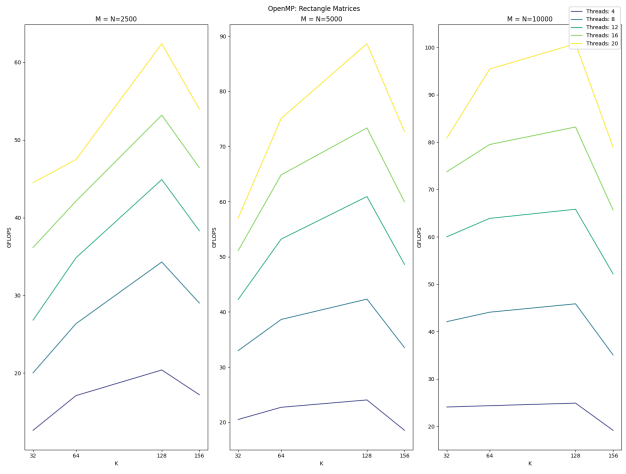


Fig. 12. OpenMP Rectangle Matrices

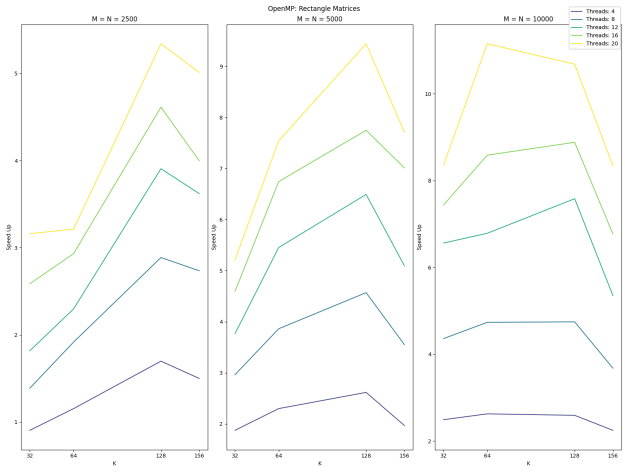


Fig. 13. OpenMP Speed Up Rectangle Matrices

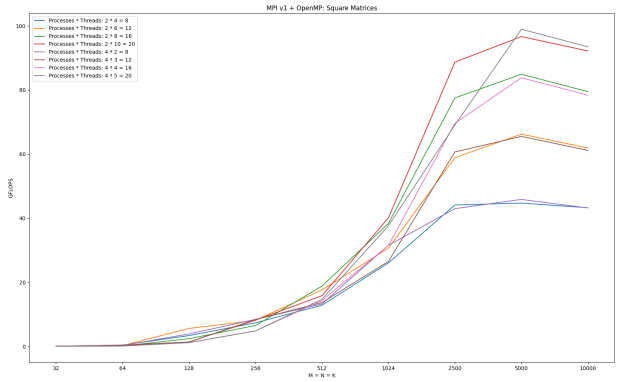


Fig. 14. MPI+OpenMP Square Matrices

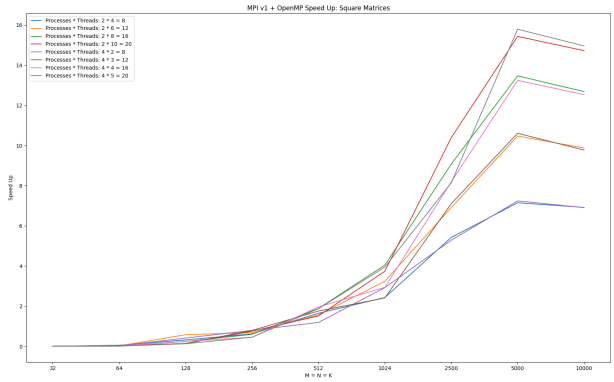


Fig. 15. MPI+OpenMP Speed Up Square Matrices

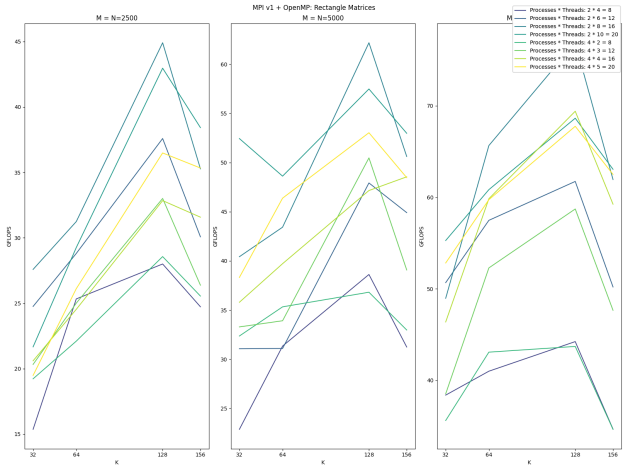


Fig. 16. MPI+OpenMP Rectangle Matrices

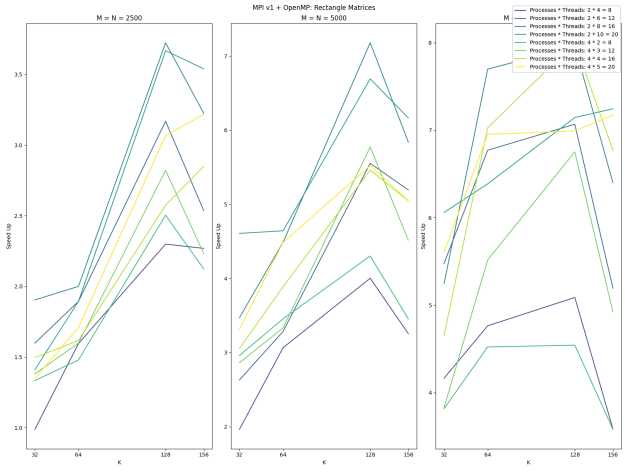


Fig. 17. MPI+OpenMP Speed Up Rectangle Matrices