

---

# **Calcolo parallelo e distribuito mod. B**

---

## **HW1**

**Implementazione algoritmo di Cannon o SUMMA in MPI**

**Boukara Djihad N97000275**

**Cicala Crispino N97000264**

# Indice

<b>1</b>	<b>Descrizione del problema</b>	<b>3</b>
1.1	Cannon . . . . .	3
1.2	SUMMA . . . . .	4
1.3	Implementazione . . . . .	7
<b>2</b>	<b>Tempi di esecuzione</b>	<b>9</b>
2.1	Grafici . . . . .	17
2.2	Analisi dei tempi . . . . .	23
<b>3</b>	<b>Appendice: Codice</b>	<b>24</b>
3.1	Utility . . . . .	24
3.2	Matmat . . . . .	30
3.3	Main . . . . .	33

# 1 Descrizione del problema

Lo scopo di questo progetto è implementare l'algoritmo di Cannon o SUMMA utilizzando la libreria MPI per il calcolo parallelo e di analizzare i tempi ottenuti dall'esecuzione su cluster. In particolare è stato implementato l'algoritmo SUMMA, che pur essendo leggermente meno efficiente dell'algoritmo di Cannon, è più generico rispetto alle dimensioni delle matrici date in input.

## 1.1 Cannon

L'algoritmo di Cannon prende in input due matrici A e B e ottiene C risolvendo  $C = C + A * B$ . L'algoritmo di Cannon può essere usato soltanto su matrici quadrate perfettamente divisibili per il numero di processori  $p$ , è richiesto inoltre che la griglia dei processori sia un quadrato perfetto.

Il primo step è quello di dividere le matrici A e B in blocchi di grandezza  $\frac{N}{s}$  dove  $s = \sqrt{p}$ . Ogni blocco ha  $\frac{N^2}{p}$  elementi. Ognuno di questi blocchi viene considerato come la più piccola unità di somma e moltiplicazione. Ai processi viene associata un'etichetta che va da  $P_{0,0}$  a  $P_{\sqrt{p}-1, \sqrt{p}-1}$  ed inizialmente viene associato il sottoblocco  $A_{i,j}$  e  $B_{i,j}$  al processo  $P_{i,j}$ . Ogni processore calcola il blocco  $C_{i,j}$  di output.

Per ogni processo della riga  $i$ -esima sono necessarie le  $\sqrt{p}$  sottomatrici  $A_{i,k}$  con  $0 \leq k < \sqrt{p}$ . Affinché i blocchi possano essere ruotati sistematicamente tra i processi dopo ogni operazione tra le sottomatrici, in modo tale da far ottenere ad ogni processo un nuovo sottoblocco  $A_{i,k}$ , è possibile gestire la computazione dei processi nell' $i$ -esima riga in modo tale che in un dato istante di tempo, ogni processo utilizzi un differente sottoblocco  $A_{i,k}$ . Applicando questa gestione dei blocchi alle colonne nessun processo necessita di più di un sottoblocco di ogni matrice per una singola iterazione. È fondamentale allineare le due matrici A e B in modo che ogni processo possa moltiplicare i propri sottoblocchi in maniera indipendente, e per farlo si opera uno shift a sinistra di  $i$  step a tutti i sottoblocchi  $A_{i,j}$  della matrice A, mentre i sottoblocchi  $B_{i,j}$  della matrice B sono shiftati in alto di  $j$  step. È fondamentale il fatto che le matrici siano considerate *toroidali*, quindi l'ipotetica riga o colonna  $n+1$  corrisponde alla riga o colonna 1. Queste operazioni fanno in modo che il processo  $P_{i,j}$  abbia as-

sociato il sottoblocco  $A_{i,(j+i)mod\sqrt{p}}$  e  $B_{(i+j)mod\sqrt{p},j}$ . A queste condizioni ogni processo può eseguire la moltiplicazione tra sottomatrici. Un volta completata l'operazione di moltiplicazione ogni blocco della matrice A *shifta* di uno step a sinistra, così come ogni blocco della matrice B *shifta* di uno step in alto. L'operazione termina quando tutti i processi hanno eseguito  $\sqrt{p}$  moltiplicazioni e la matrice C viene *assemblata* utilizzando tutti i blocchi calcolati  $C_{i,j}$ .

## 1.2 SUMMA

L'algoritmo SUMMA (Scalable Universal Matrix Multiplication Algorithm) come già accennato è leggermente meno efficiente dell'algoritmo di Cannon ma più facile da generalizzare. Può infatti gestire qualunque distribuzione, dimensione e allineamento. A differenza dell'algoritmo di Cannon viene utilizzato il broadcast di blocchi invece dello shift circolare. Dal punto di vista computazionale effettua più operazioni rispetto all'algoritmo di Cannon, ma utilizza meno memoria.

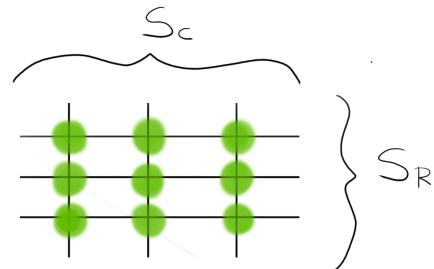


Figura 1.1: Griglia

Consideriamo una griglia di  $S_C$  colonne e  $S_R$  righe, le tre matrici A, B e C, dove A ha N righe ed M colonne, B ha M righe e P colonne, e C ha N righe e P colonne.

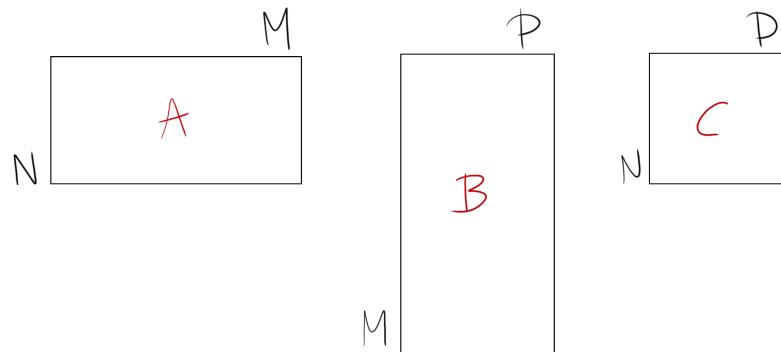


Figura 1.2: Matrici A, B e C

Le matrici vengono divise in blocchi:

A: K1 x K2 blocchi, con  $K1=S_R$  e  $K2=S_RS_C$

B: K2 x K3 blocchi, con  $K2=S_RS_C$  e  $K3=S_C$

C: K1 x K3 blocchi, con  $K1=S_R$  e  $K3=S_C$

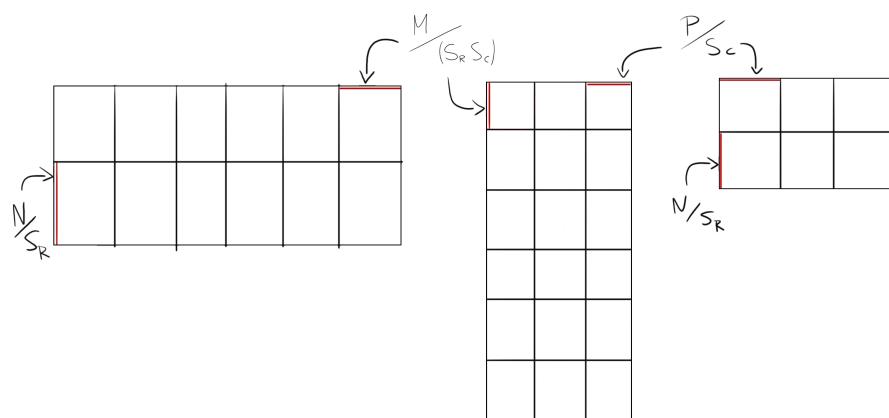


Figura 1.3: Divisione matrici

Un generico  $C_{i,j}$  è dato da  $\sum_k A_{i,k} * B_{k,j}$  con  $k = 0, \dots, K2 - 1$ . Solo  $A_{i,j}$ ,  $B_{i,j}$  e  $C_{i,j}$  sono in possesso di  $P_{ij}$  e ogni nodo  $P_{ij}$  calcola  $C_{ij}$  in parallelo. Si osserva però che per  $k = 0$  non tutti i nodi necessari alla determinazione del generico  $C_{ij}$  sono disponibili e quindi non tutti i nodi possono iniziare il calcolo. Per ottenere i dati necessari affinché tutti i nodi possano eseguire il calcolo per  $k = 0$  è necessario effettuare un broadcast di un blocco di A lungo la riga di nodi e un broadcast di un blocco di B lungo la colonna dei nodi. Effettuando lo stesso tipo di broadcast per  $k = 1$ , quindi l'iterazione successiva, otteniamo tutti i dati necessari affiché i nodi possano eseguire correttamente il calcolo. L'algoritmo SUMMA SPMD (single program, multiple data) è riportato di seguito:

```

for  $k \leftarrow 0$  to  $K2 - 1$  do
     $c = k \% S_C$ 
     $r = k \% S_R$ 
     $P_{ic}$  broadcast  $A_{i,k}$  lungo la riga  $i$       ( $i = 0, \dots, K1 - 1$ )
     $P_{rj}$  broadcast  $B_{k,j}$  lungo la colonna  $j$   ( $j = 0, \dots, K3 - 1$ )
    Receive  $A_{i,k}$  in  $A_{COL}$ 
    Receive  $B_{k,j}$  in  $B_{ROW}$ 
     $C = C + A_{COL} * B_{ROW}$ 
end

```

## 1.3 Implementazione

Si è deciso di implementare l'algoritmo SUMMA per le ragioni esposte in precedenza. Vista la più articolata struttura del progetto, si è deciso di dividere le funzioni in tre diversi *blocchi*: *utility*, *matmat* e *main*.

In *utility* sono presenti le seguenti funzioni:

- **lcm**: dati due interi, ritorna il minimo comune multiplo tra i due.
- **rnd\_flt\_matrix**: dati due interi **n** ed **m**, crea e ritorna una matrice  $n \times m$  di *float* casuali.
- **zeros\_flt\_matrix**: ritorna il puntatore ad una matrice di *float* di dimensioni  $n \times m$  con tutti elementi uguali a 0.
- **clear\_matrix**: data una matrice, le sue dimensioni ed un offset, riempi la matrice di 0.
- **cmp\_matrix**: date due matrici ritorna il numero di elementi differenti.
- **print\_matrix**: stampa la matrice data come parametro.
- **time\_elapsed**: dato un tempo di inizio ed uno di fine ritorna un numero che indica il numero di secondi passati.
- **init\_mpi\_env**: inizializza l'ambiente mpi attraverso le chiamate a **MPI\_Init**, **MPI\_Comm\_rank** e **MPI\_Comm\_size**.
- **create\_cart\_grid**: attraverso le chiamate a **MPI\_Cart\_create**, **MPI\_Comm\_rank** e **MPI\_Cart\_coords** inizializza la griglia di processori.
- **cart\_sub**: attraverso la chiamata a **MPI\_Cart\_sub** partiziona il comunicatore in sottogruppi che formano sottogrigli di dimensioni inferiori.
- **cp\_matrix**: date due matrici **S** e **D** copia il contenuto della matrice **D** in **S**.
- **cyclic\_distribution**: si occupa di tutte le operazioni necessarie ad effettuare la distribuzione delle sottomatrici ai processori. Include sia le operazioni di calcolo che di comunicazione, effettuate attraverso le chiamate ad **MPI\_Send** ed **MPI\_Recv**.

- **gather\_result**: riunisce tutti i risultati calcolati in un'unica soluzione. Anche in questo caso sono incluse le comunicazioni effettuate attraverso le chiamate ad **MPI\_Send** ed **MPI\_Recv**.

Per quanto riguarda il blocco *matmat*, sono incluse le funzioni descritte nei progetti precedenti per il calcolo del prodotto tra matrici. Sono quindi incluse la funzione per il *calcolo a singolo blocco* del prodotto matrice per matrice (**matmat**), quella per il *calcolo con logica blocchi* (**matmat\_blocks**) e quella per il calcolo con logica a blocchi che sfrutta i *thread* (**matmat\_threads**), infine la funzione **SUMMA** risolve il prodotto tra matrici utilizzando l'algoritmo SUMMA.

Il blocco *main* raccoglie tutte le funzioni descritte per eseguire il calcolo del prodotto tra matrici utilizzando l'algoritmo SUMMA utilizzando la libreria MPI per le comunicazioni tra processori. L'algoritmo è stato eseguito per la raccolta dei tempi computazionali dal cluster dell'architettura SCoPE dell'Università degli studi di Napoli "Federico II".

# 2 Tempi di esecuzione

Sr	Sc	ntrow	ntcol	total core	matrix size	time	gflops	speedup	efficiency
1	1	1	1	1	1680	6,953	1,364	1	1
1	1	1	1	1	3360	55,491	1,367	1	1
1	1	1	1	1	5040	193,130	1,326	1	1
1	1	1	1	1	6720	444,160	1,366	1	1
1	1	1	1	1	8400	865,936	1,369	1	1
1	1	1	1	1	10080	1,560,971	1,312	1	1
1	1	2	1	2	1680	3,498	2,711	1,988	0,994
1	1	2	1	2	3360	27,885	2,721	1,990	0,995
1	1	2	1	2	5040	96,751	2,646	1,996	0,998
1	1	2	1	2	6720	222,546	2,727	1,996	0,998
1	1	2	1	2	8400	433,721	2,733	1,997	0,998
1	1	2	1	2	10080	781,841	2,620	1,997	0,998
1	1	3	1	3	1680	2,341	4,050	2,970	0,990
1	1	3	1	3	3360	18,577	4,084	2,987	0,996
1	1	3	1	3	5040	64,598	3,964	2,990	0,997
1	1	3	1	3	6720	148,449	4,088	2,992	0,997
1	1	3	1	3	8400	289,421	4,096	2,992	0,997
1	1	3	1	3	10080	521,515	3,928	2,993	0,998
1	1	2	2	4	1680	1,782	5,323	3,902	0,975
1	1	2	2	4	3360	14,018	5,412	3,959	0,990
1	1	2	2	4	5040	48,592	5,269	3,975	0,994
1	1	2	2	4	6720	111,443	5,446	3,986	0,996
1	1	2	2	4	8400	217,279	5,456	3,985	0,996
1	1	2	2	4	10080	391,707	5,229	3,985	0,996
1	1	5	1	5	1680	1,418	6,687	4,903	0,981
1	1	5	1	5	3360	11,237	6,751	4,938	0,988
1	1	5	1	5	5040	38,886	6,585	4,967	0,993
1	1	5	1	5	6720	89,450	6,785	4,965	0,993
1	1	5	1	5	8400	174,304	6,801	4,968	0,994
1	1	5	1	5	10080	313,310	6,538	4,982	0,996
1	1	2	3	6	1680	1,196	7,929	5,814	0,969
1	1	2	3	6	3360	9,452	8,027	5,871	0,978
1	1	2	3	6	5040	32,675	7,836	5,911	0,985
1	1	2	3	6	6720	74,754	8,119	5,942	0,990
1	1	2	3	6	8400	145,878	8,126	5,936	0,989
1	1	2	3	6	10080	261,979	7,819	5,958	0,993
1	1	7	1	7	1680	1,022	9,279	6,803	0,972
1	1	7	1	7	3360	8,051	9,423	6,892	0,985
1	1	7	1	7	5040	27,906	9,175	6,921	0,989
1	1	7	1	7	6720	64,066	9,473	6,933	0,990
1	1	7	1	7	8400	124,899	9,491	6,933	0,990
1	1	7	1	7	10080	224,150	9,138	6,964	0,995
1	1	2	4	8	1680	0,916	10,353	7,591	0,949
1	1	2	4	8	3360	7,139	10,627	7,773	0,972
1	1	2	4	8	5040	24,513	10,446	7,879	0,985
1	1	2	4	8	6720	56,423	10,757	7,872	0,984
1	1	2	4	8	8400	109,709	10,805	7,893	0,987
1	1	2	4	8	10080	196,944	10,401	7,926	0,991

Sr	Sc	ntrow	ntcol	total core	matrix size	time	gflops	speedup	efficiency
2	1	1	1	2	1680	3,492	2,715	1,991	0,996
2	1	1	1	2	3360	27,828	2,726	1,994	0,997
2	1	1	1	2	5040	96,809	2,645	1,995	0,997
2	1	1	1	2	6720	222,675	2,726	1,995	0,997
2	1	1	1	2	8400	433,814	2,733	1,996	0,998
2	1	1	1	2	10080	781,775	2,620	1,997	0,998
2	1	2	1	4	1680	1,771	5,354	3,926	0,982
2	1	2	1	4	3360	13,971	5,430	3,972	0,993
2	1	2	1	4	5040	48,545	5,274	3,978	0,995
2	1	2	1	4	6720	111,714	5,433	3,976	0,994
2	1	2	1	4	8400	217,521	5,450	3,981	0,995
2	1	2	1	4	10080	391,729	5,229	3,985	0,996
2	1	3	1	6	1680	1,201	7,896	5,789	0,965
2	1	3	1	6	3360	9,382	8,086	5,915	0,986
2	1	3	1	6	5040	32,520	7,874	5,939	0,990
2	1	3	1	6	6720	74,711	8,124	5,945	0,991
2	1	3	1	6	8400	145,479	8,148	5,952	0,992
2	1	3	1	6	10080	261,703	7,827	5,965	0,994
2	1	2	2	8	1680	0,924	10,269	7,525	0,941
2	1	2	2	8	3360	7,120	10,655	7,794	0,974
2	1	2	2	8	5040	24,579	10,418	7,858	0,982
2	1	2	2	8	6720	56,440	10,754	7,870	0,984
2	1	2	2	8	8400	109,872	10,789	7,881	0,985
2	1	2	2	8	10080	197,324	10,381	7,911	0,989
2	1	5	1	10	1680	1,256	7,551	5,536	0,554
2	1	5	1	10	3360	8,469	8,958	6,552	0,655
2	1	5	1	10	5040	29,309	8,736	6,589	0,659
2	1	5	1	10	6720	73,078	8,305	6,078	0,608
2	1	5	1	10	8400	135,726	8,734	6,380	0,638
2	1	5	1	10	10080	230,152	8,900	6,782	0,678
2	1	2	3	12	1680	1,183	8,019	5,877	0,490
2	1	2	3	12	3360	8,350	9,085	6,646	0,554
2	1	2	3	12	5040	32,581	7,859	5,928	0,494
2	1	2	3	12	6720	74,790	8,115	5,939	0,495
2	1	2	3	12	8400	127,728	9,281	6,780	0,565
2	1	2	3	12	10080	261,862	7,822	5,961	0,497
2	1	7	1	14	1680	1,199	7,911	5,799	0,414
2	1	7	1	14	3360	9,110	8,327	6,091	0,435
2	1	7	1	14	5040	32,227	7,945	5,993	0,428
2	1	7	1	14	6720	72,358	8,388	6,138	0,438
2	1	7	1	14	8400	143,363	8,269	6,040	0,431
2	1	7	1	14	10080	257,409	7,958	6,064	0,433
2	1	2	4	16	1680	1,280	7,408	5,432	0,340
2	1	2	4	16	3360	9,632	7,877	5,761	0,360
2	1	2	4	16	5040	33,330	7,682	5,794	0,362
2	1	2	4	16	6720	76,203	7,965	5,829	0,364
2	1	2	4	16	8400	150,073	7,899	5,770	0,361
2	1	2	4	16	10080	266,045	7,699	5,867	0,367

Sr	Sc	ntrow	ntcol	total core	matrix size	time	gflops	speedup	efficiency
3	1	1	1	3	1680	2,347	4,041	2,963	0,988
3	1	1	1	3	3360	18,666	4,064	2,973	0,991
3	1	1	1	3	5040	64,705	3,957	2,985	0,995
3	1	1	1	3	6720	149,038	4,072	2,980	0,993
3	1	1	1	3	8400	289,675	4,092	2,989	0,996
3	1	1	1	3	10080	521,704	3,926	2,992	0,997
3	1	2	1	6	1680	1,203	7,881	5,780	0,963
3	1	2	1	6	3360	9,416	8,057	5,893	0,982
3	1	2	1	6	5040	32,533	7,870	5,936	0,989
3	1	2	1	6	6720	78,822	7,700	5,635	0,939
3	1	2	1	6	8400	145,716	8,135	5,943	0,990
3	1	2	1	6	10080	283,605	7,223	5,504	0,917
3	1	3	1	9	1680	1,173	8,083	5,928	0,659
3	1	3	1	9	3360	8,577	8,846	6,470	0,719
3	1	3	1	9	5040	29,867	8,573	6,466	0,718
3	1	3	1	9	6720	68,675	8,838	6,468	0,719
3	1	3	1	9	8400	134,888	8,788	6,420	0,713
3	1	3	1	9	10080	241,870	8,469	6,454	0,717
3	1	2	2	12	1680	1,038	9,137	6,698	0,558
3	1	2	2	12	3360	7,847	9,668	7,072	0,589
3	1	2	2	12	5040	24,970	10,254	7,734	0,645
3	1	2	2	12	6720	58,970	10,292	7,532	0,628
3	1	2	2	12	8400	113,852	10,412	7,606	0,634
3	1	2	2	12	10080	212,518	9,639	7,345	0,612
3	1	5	1	15	1680	1,032	9,188	6,737	0,449
3	1	5	1	15	3360	7,833	9,686	7,084	0,472
3	1	5	1	15	5040	26,446	9,682	7,303	0,487
3	1	5	1	15	6720	60,878	9,970	7,296	0,486
3	1	5	1	15	8400	119,342	9,933	7,256	0,484
3	1	5	1	15	10080	215,478	9,506	7,244	0,483
3	1	2	3	18	1680	1,089	8,709	6,385	0,355
3	1	2	3	18	3360	7,729	9,815	7,180	0,399
3	1	2	3	18	5040	26,578	9,634	7,267	0,404
3	1	2	3	18	6720	61,401	9,885	7,234	0,402
3	1	2	3	18	8400	119,175	9,947	7,266	0,404
3	1	2	3	18	10080	213,584	9,591	7,308	0,406
3	1	7	1	21	1680	1,122	8,451	6,197	0,295
3	1	7	1	21	3360	7,938	9,558	6,991	0,333
3	1	7	1	21	5040	27,599	9,278	6,998	0,333
3	1	7	1	21	6720	62,542	9,704	7,102	0,338
3	1	7	1	21	8400	118,582	9,997	7,302	0,348
3	1	7	1	21	10080	219,035	9,352	7,127	0,339
3	1	2	4	24	1680	1,107	8,564	6,281	0,262
3	1	2	4	24	3360	8,245	9,201	6,730	0,280
3	1	2	4	24	5040	29,856	8,576	6,469	0,270
3	1	2	4	24	6720	65,589	9,254	6,772	0,282
3	1	2	4	24	8400	129,153	9,178	6,705	0,279
3	1	2	4	24	10080	231,411	8,852	6,745	0,281

Sr	Sc	ntrow	ntcol	total core	matrix size	time	gflops	speedup	efficiency
2	2	1	1	4	1680	1,764	5,375	3,942	0,985
2	2	1	1	4	3360	15,294	4,960	3,628	0,907
2	2	1	1	4	5040	48,104	5,323	4,015	1,004
2	2	1	1	4	6720	111,541	5,441	3,982	0,996
2	2	1	1	4	8400	237,744	4,986	3,642	0,911
2	2	1	1	4	10080	408,400	5,016	3,822	0,956
2	2	2	1	8	1680	1,141	8,310	6,094	0,762
2	2	2	1	8	3360	9,458	8,021	5,867	0,733
2	2	2	1	8	5040	33,806	7,574	5,713	0,714
2	2	2	1	8	6720	79,017	7,681	5,621	0,703
2	2	2	1	8	8400	153,355	7,730	5,647	0,706
2	2	2	1	8	10080	266,879	7,675	5,849	0,731
2	2	3	1	12	1680	0,990	9,612	7,023	0,585
2	2	3	1	12	3360	7,955	9,537	6,976	0,581
2	2	3	1	12	5040	27,491	9,314	7,025	0,585
2	2	3	1	12	6720	63,876	9,502	6,953	0,579
2	2	3	1	12	8400	123,561	9,594	7,008	0,584
2	2	3	1	12	10080	218,550	9,373	7,142	0,595
2	2	2	2	16	1680	1,020	9,302	6,817	0,426
2	2	2	2	16	3360	7,212	10,519	7,694	0,481
2	2	2	2	16	5040	24,348	10,516	7,932	0,496
2	2	2	2	16	6720	56,482	10,746	7,864	0,491
2	2	2	2	16	8400	109,708	10,805	7,893	0,493
2	2	2	2	16	10080	195,281	10,489	7,993	0,500
2	2	5	1	20	1680	0,940	10,091	7,397	0,370
2	2	5	1	20	3360	7,259	10,451	7,644	0,382
2	2	5	1	20	5040	25,782	9,931	7,491	0,375
2	2	5	1	20	6720	61,607	9,852	7,210	0,360
2	2	5	1	20	8400	110,520	10,726	7,835	0,392
2	2	5	1	20	10080	204,589	10,012	7,630	0,381
2	2	2	3	24	1680	0,953	9,951	7,296	0,304
2	2	2	3	24	3360	7,865	9,646	7,055	0,294
2	2	2	3	24	5040	26,725	9,581	7,227	0,301
2	2	2	3	24	6720	61,376	9,889	7,237	0,302
2	2	2	3	24	8400	116,829	10,147	7,412	0,309
2	2	2	3	24	10080	205,795	9,954	7,585	0,316
2	2	7	1	28	1680	0,996	9,523	6,981	0,249
2	2	7	1	28	3360	7,599	9,984	7,302	0,261
2	2	7	1	28	5040	26,177	9,781	7,378	0,263
2	2	7	1	28	6720	60,304	10,064	7,365	0,263
2	2	7	1	28	8400	115,827	10,234	7,476	0,267
2	2	7	1	28	10080	209,019	9,800	7,468	0,267
2	2	2	4	32	1680	0,982	9,660	7,080	0,221
2	2	2	4	32	3360	7,763	9,773	7,148	0,223
2	2	2	4	32	5040	27,836	9,199	6,938	0,217
2	2	2	4	32	6720	60,317	10,062	7,364	0,230
2	2	2	4	32	8400	117,514	10,087	7,369	0,230
2	2	2	4	32	10080	209,399	9,782	7,455	0,233

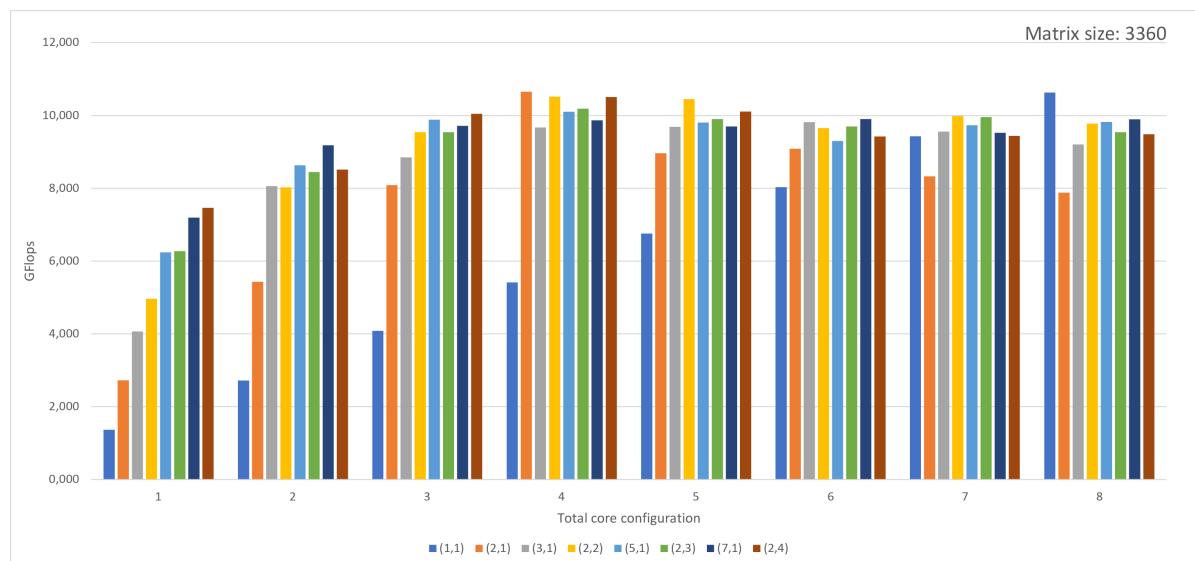
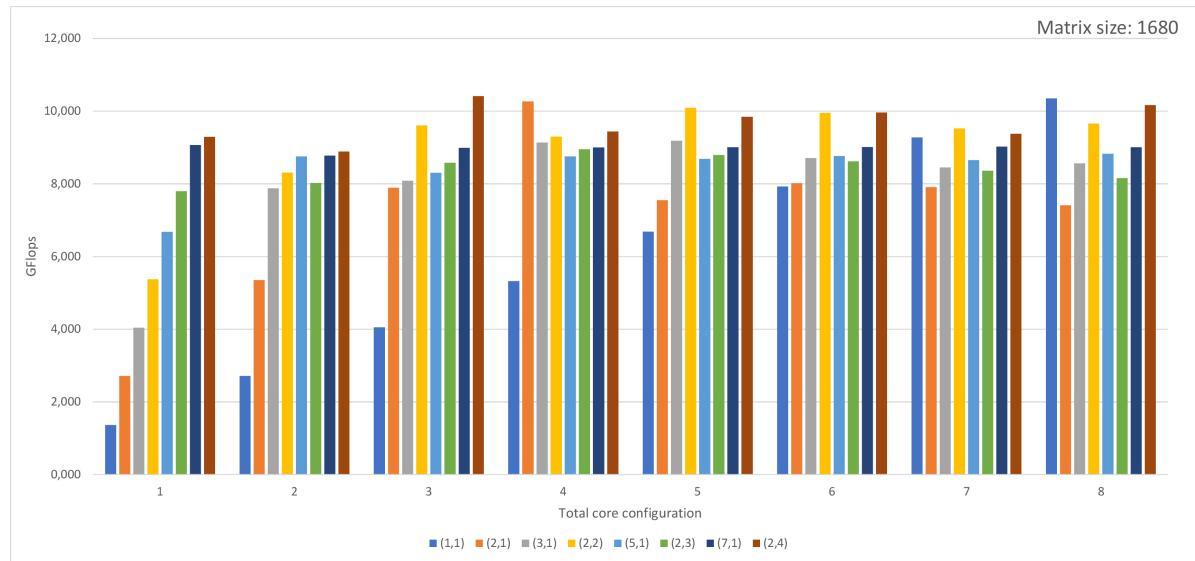
Sr	Sc	ntrow	ntcol	total core	matrix size	time	gflops	speedup	efficiency
5	1	1	1	5	1680	1,419	6,683	4,900	0,980
5	1	1	1	5	3360	12,158	6,24	4,564	0,913
5	1	1	1	5	5040	46,894	5,46	4,118	0,824
5	1	1	1	5	6720	111,822	5,428	3,972	0,794
5	1	1	1	5	8400	217,298	5,455	3,985	0,797
5	1	1	1	5	10080	391,417	5,233	3,988	0,798
5	1	2	1	10	1680	1,083	8,755	6,420	0,642
5	1	2	1	10	3360	8,792	8,629	6,312	0,631
5	1	2	1	10	5040	30,909	8,284	6,248	0,625
5	1	2	1	10	6720	71,397	8,501	6,221	0,622
5	1	2	1	10	8400	137,964	8,592	6,277	0,628
5	1	2	1	10	10080	241,264	8,49	6,470	0,647
5	1	3	1	15	1680	1,142	8,306	6,088	0,406
5	1	3	1	15	3360	7,677	9,882	7,228	0,482
5	1	3	1	15	5040	27,218	9,407	7,096	0,473
5	1	3	1	15	6720	62,911	9,647	7,060	0,471
5	1	3	1	15	8400	121,898	9,725	7,104	0,474
5	1	3	1	15	10080	219,352	9,338	7,116	0,474
5	1	2	2	20	1680	1,084	8,752	6,414	0,321
5	1	2	2	20	3360	7,513	10,098	7,386	0,369
5	1	2	2	20	5040	25,239	10,145	7,652	0,383
5	1	2	2	20	6720	57,903	10,482	7,671	0,384
5	1	2	2	20	8400	112,484	10,538	7,698	0,385
5	1	2	2	20	10080	202,361	10,122	7,714	0,386
5	1	5	1	25	1680	1,092	8,685	6,367	0,255
5	1	5	1	25	3360	7,74	9,802	7,169	0,287
5	1	5	1	25	5040	25,375	10,091	7,611	0,304
5	1	5	1	25	6720	57,885	10,485	7,673	0,307
5	1	5	1	25	8400	114,178	10,382	7,584	0,303
5	1	5	1	25	10080	205,408	9,972	7,599	0,304
5	1	2	3	30	1680	1,082	8,766	6,426	0,214
5	1	2	3	30	3360	8,157	9,301	6,803	0,227
5	1	2	3	30	5040	25,975	9,857	7,435	0,248
5	1	2	3	30	6720	60,092	10,1	7,391	0,246
5	1	2	3	30	8400	114,478	10,355	7,564	0,252
5	1	2	3	30	10080	205,428	9,971	7,599	0,253
5	1	7	1	35	1680	1,095	8,658	6,350	0,181
5	1	7	1	35	3360	7,797	9,73	7,117	0,203
5	1	7	1	35	5040	26,242	9,757	7,360	0,210
5	1	7	1	35	6720	60,591	10,017	7,330	0,209
5	1	7	1	35	8400	116,325	10,191	7,444	0,213
5	1	7	1	35	10080	207,741	9,86	7,514	0,215
5	1	2	4	40	1680	1,074	8,831	6,474	0,162
5	1	2	4	40	3360	7,727	9,819	7,181	0,180
5	1	2	4	40	5040	26,97	9,494	7,161	0,179
5	1	2	4	40	6720	61,256	9,908	7,251	0,181
5	1	2	4	40	8400	118,395	10,012	7,314	0,183
5	1	2	4	40	10080	210,491	9,731	7,416	0,185

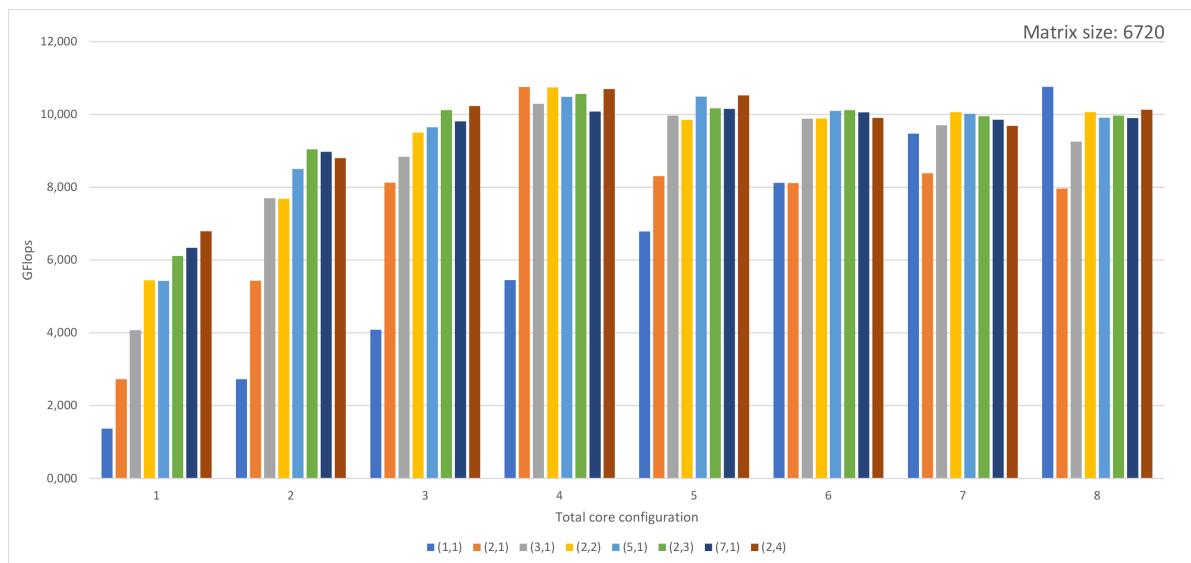
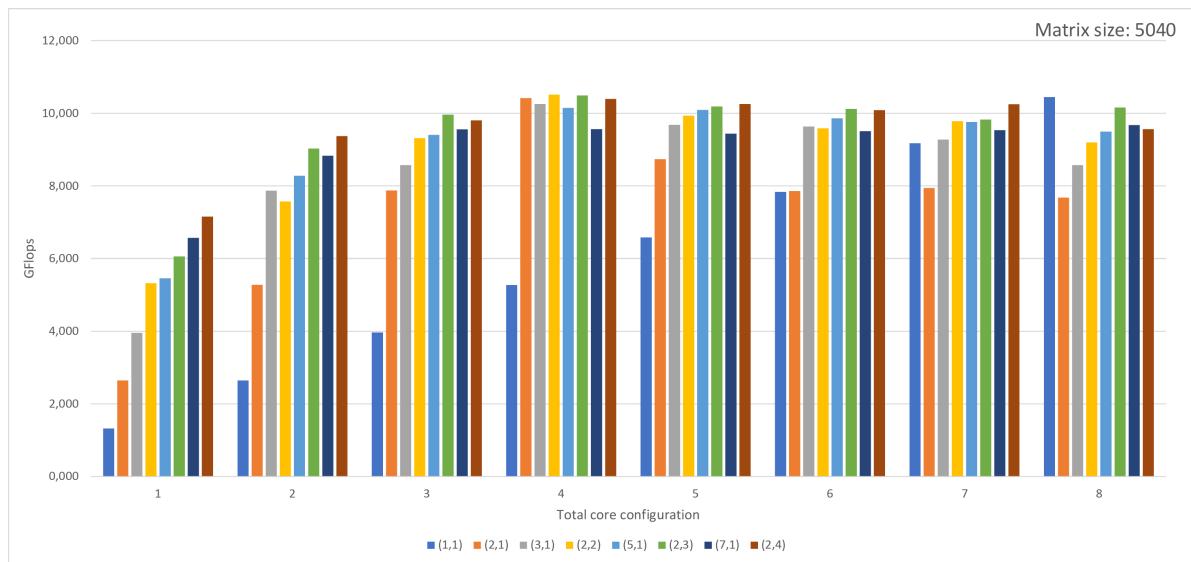
Sr	Sc	ntrow	ntcol	total core	matrix size	time	gflops	speedup	efficiency
2	3	1	1	6	1680	1,216	7,798	5,718	0,953
2	3	1	1	6	3360	12,095	6,273	4,588	0,765
2	3	1	1	6	5040	42,253	6,06	4,571	0,762
2	3	1	1	6	6720	99,278	6,113	4,474	0,746
2	3	1	1	6	8400	199,674	5,937	4,337	0,723
2	3	1	1	6	10080	343,132	5,97	4,549	0,758
2	3	2	1	12	1680	1,181	8,027	5,887	0,491
2	3	2	1	12	3360	8,986	8,443	6,175	0,515
2	3	2	1	12	5040	28,348	9,032	6,813	0,568
2	3	2	1	12	6720	67,127	9,041	6,617	0,551
2	3	2	1	12	8400	131,341	9,025	6,593	0,549
2	3	2	1	12	10080	231,404	8,852	6,746	0,562
2	3	3	1	18	1680	1,105	8,579	6,292	0,350
2	3	3	1	18	3360	7,955	9,537	6,976	0,388
2	3	3	1	18	5040	25,708	9,96	7,512	0,417
2	3	3	1	18	6720	59,976	10,12	7,406	0,411
2	3	3	1	18	8400	116,792	10,15	7,414	0,412
2	3	3	1	18	10080	203,002	10,09	7,689	0,427
2	3	2	2	24	1680	1,059	8,953	6,566	0,274
2	3	2	2	24	3360	7,446	10,189	7,452	0,311
2	3	2	2	24	5040	24,414	10,488	7,911	0,330
2	3	2	2	24	6720	57,464	10,562	7,729	0,322
2	3	2	2	24	8400	110,036	10,773	7,870	0,328
2	3	2	2	24	10080	191,179	10,714	8,165	0,340
2	3	5	1	30	1680	1,079	8,792	6,444	0,215
2	3	5	1	30	3360	7,663	9,9	7,241	0,241
2	3	5	1	30	5040	25,124	10,191	7,687	0,256
2	3	5	1	30	6720	59,659	10,173	7,445	0,248
2	3	5	1	30	8400	112,727	10,516	7,682	0,256
2	3	5	1	30	10080	200,063	10,239	7,802	0,260
2	3	2	3	36	1680	1,1	8,621	6,321	0,176
2	3	2	3	36	3360	7,823	9,698	7,093	0,197
2	3	2	3	36	5040	25,305	10,119	7,632	0,212
2	3	2	3	36	6720	59,973	10,12	7,406	0,206
2	3	2	3	36	8400	116,954	10,136	7,404	0,206
2	3	2	3	36	10080	199,041	10,291	7,842	0,218
2	3	7	1	42	1680	1,134	8,36	6,131	0,146
2	3	7	1	42	3360	7,62	9,956	7,282	0,173
2	3	7	1	42	5040	26,047	9,83	7,415	0,177
2	3	7	1	42	6720	61,004	9,949	7,281	0,173
2	3	7	1	42	8400	115,802	10,236	7,478	0,178
2	3	7	1	42	10080	198,805	10,303	7,852	0,187
2	3	2	4	48	1680	1,162	8,164	5,984	0,125
2	3	2	4	48	3360	7,954	9,538	6,976	0,145
2	3	2	4	48	5040	25,2	10,161	7,664	0,160
2	3	2	4	48	6720	60,875	9,97	7,296	0,152
2	3	2	4	48	8400	115,441	10,269	7,501	0,156
2	3	2	4	48	10080	201,033	10,189	7,765	0,162

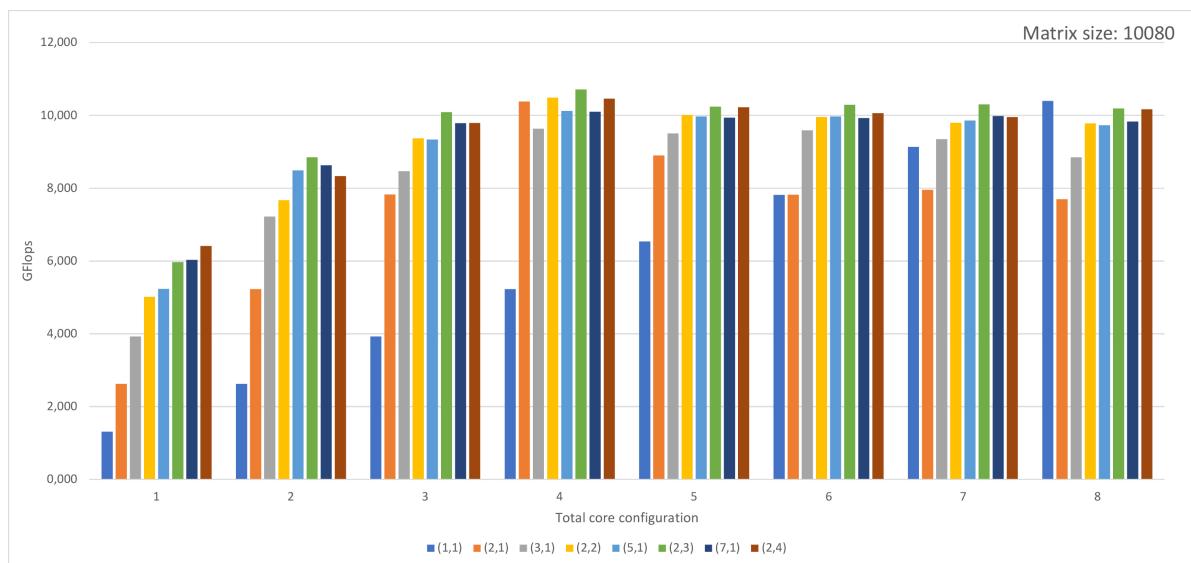
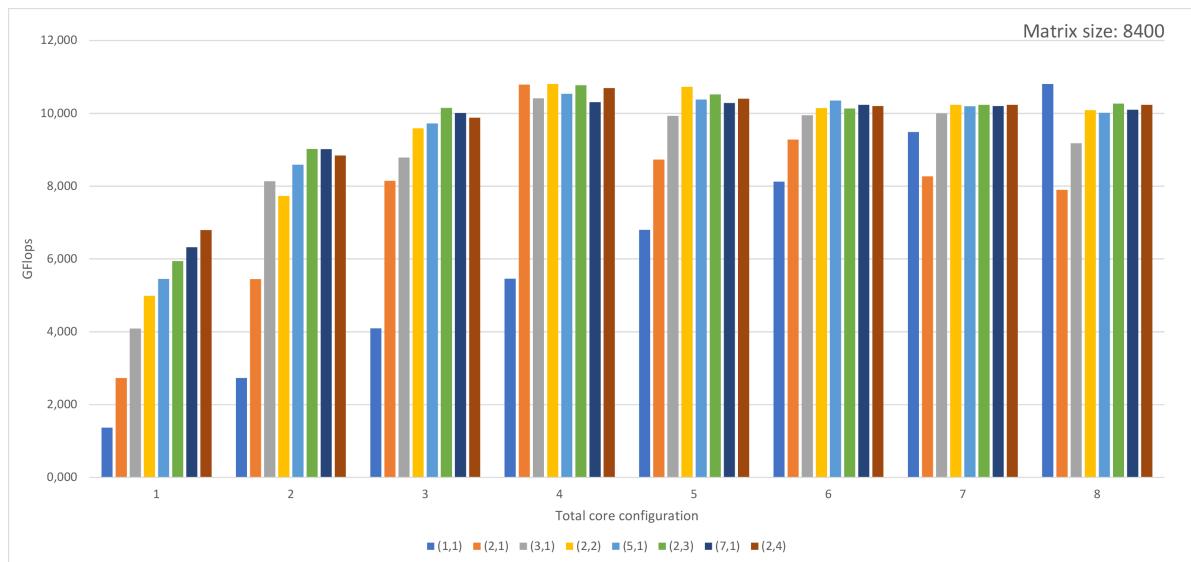
Sr	Sc	ntrow	ntcol	total core	matrix size	time	gflops	speedup	efficiency
7	1	1	1	7	1680	1,046	9,07	6,647	0,950
7	1	1	1	7	3360	10,545	7,195	5,262	0,752
7	1	1	1	7	5040	38,965	6,571	4,956	0,708
7	1	1	1	7	6720	95,834	6,333	4,635	0,662
7	1	1	1	7	8400	187,493	6,322	4,618	0,660
7	1	1	1	7	10080	339,557	6,033	4,597	0,657
7	1	2	1	14	1680	1,08	8,777	6,438	0,460
7	1	2	1	14	3360	8,266	9,178	6,713	0,480
7	1	2	1	14	5040	28,995	8,831	6,661	0,476
7	1	2	1	14	6720	67,648	8,972	6,566	0,469
7	1	2	1	14	8400	131,538	9,012	6,583	0,470
7	1	2	1	14	10080	237,344	8,63	6,577	0,470
7	1	3	1	21	1680	1,055	8,993	6,591	0,314
7	1	3	1	21	3360	7,812	9,711	7,103	0,338
7	1	3	1	21	5040	26,791	9,557	7,209	0,343
7	1	3	1	21	6720	61,834	9,815	7,183	0,342
7	1	3	1	21	8400	118,421	10,01	7,312	0,348
7	1	3	1	21	10080	209,262	9,789	7,459	0,355
7	1	2	2	28	1680	1,054	8,999	6,597	0,236
7	1	2	2	28	3360	7,691	9,864	7,215	0,258
7	1	2	2	28	5040	26,77	9,565	7,214	0,258
7	1	2	2	28	6720	60,224	10,078	7,375	0,263
7	1	2	2	28	8400	115,048	10,304	7,527	0,269
7	1	2	2	28	10080	202,735	10,104	7,700	0,275
7	1	5	1	35	1680	1,053	9,008	6,603	0,189
7	1	5	1	35	3360	7,825	9,695	7,092	0,203
7	1	5	1	35	5040	27,111	9,444	7,124	0,204
7	1	5	1	35	6720	59,79	10,151	7,429	0,212
7	1	5	1	35	8400	115,247	10,286	7,514	0,215
7	1	5	1	35	10080	206,123	9,938	7,573	0,216
7	1	2	3	42	1680	1,052	9,014	6,609	0,157
7	1	2	3	42	3360	7,667	9,895	7,238	0,172
7	1	2	3	42	5040	26,941	9,504	7,169	0,171
7	1	2	3	42	6720	60,369	10,054	7,357	0,175
7	1	2	3	42	8400	115,789	10,238	7,479	0,178
7	1	2	3	42	10080	206,309	9,929	7,566	0,180
7	1	7	1	49	1680	1,051	9,027	6,616	0,135
7	1	7	1	49	3360	7,967	9,523	6,965	0,142
7	1	7	1	49	5040	26,86	9,533	7,190	0,147
7	1	7	1	49	6720	61,57	9,857	7,214	0,147
7	1	7	1	49	8400	116,178	10,203	7,454	0,152
7	1	7	1	49	10080	205,242	9,98	7,606	0,155
7	1	2	4	56	1680	1,053	9,006	6,603	0,118
7	1	2	4	56	3360	7,668	9,893	7,237	0,129
7	1	2	4	56	5040	26,461	9,676	7,299	0,130
7	1	2	4	56	6720	61,315	9,899	7,244	0,129
7	1	2	4	56	8400	117,359	10,101	7,379	0,132
7	1	2	4	56	10080	208,306	9,834	7,494	0,134

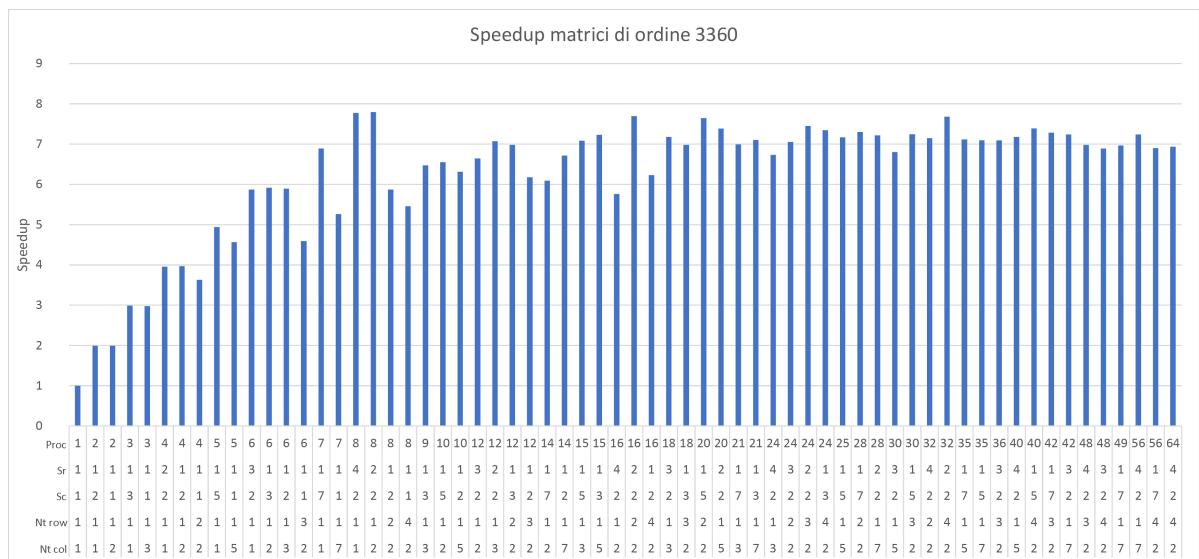
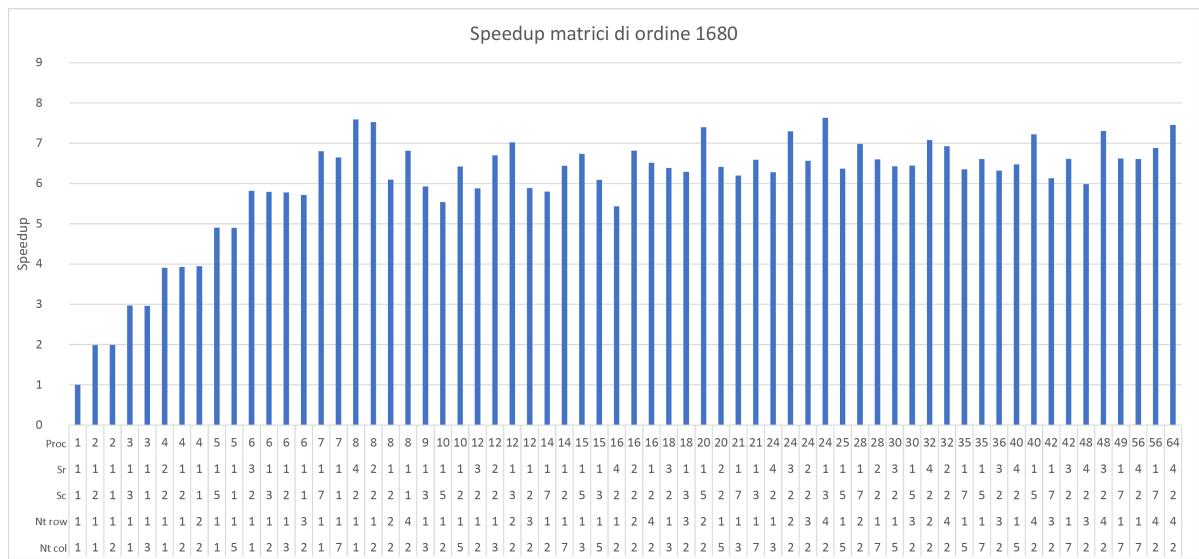
Sr	Sc	ntrow	ntcol	total core	matrix size	time	gflops	speedup	efficiency
2	4	1	1	8	1680	1,02	9,295	6,817	0,852
2	4	1	1	8	3360	10,168	7,461	5,457	0,682
2	4	1	1	8	5040	35,772	7,158	5,399	0,675
2	4	1	1	8	6720	89,393	6,789	4,969	0,621
2	4	1	1	8	8400	174,523	6,792	4,962	0,620
2	4	1	1	8	10080	319,517	6,411	4,885	0,611
2	4	2	1	16	1680	1,067	8,891	6,516	0,407
2	4	2	1	16	3360	8,912	8,513	6,227	0,389
2	4	2	1	16	5040	27,322	9,372	7,069	0,442
2	4	2	1	16	6720	68,973	8,799	6,440	0,402
2	4	2	1	16	8400	134,026	8,845	6,461	0,404
2	4	2	1	16	10080	245,806	8,333	6,350	0,397
2	4	3	1	24	1680	0,911	10,411	7,632	0,318
2	4	3	1	24	3360	7,553	10,044	7,347	0,306
2	4	3	1	24	5040	26,107	9,808	7,398	0,308
2	4	3	1	24	6720	59,333	10,229	7,486	0,312
2	4	3	1	24	8400	119,971	9,881	7,218	0,301
2	4	3	1	24	10080	209,181	9,792	7,462	0,311
2	4	2	2	32	1680	1,004	9,443	6,925	0,216
2	4	2	2	32	3360	7,223	10,504	7,683	0,240
2	4	2	2	32	5040	24,63	10,396	7,841	0,245
2	4	2	2	32	6720	56,736	10,697	7,829	0,245
2	4	2	2	32	8400	110,84	10,695	7,812	0,244
2	4	2	2	32	10080	195,883	10,457	7,969	0,249
2	4	5	1	40	1680	0,963	9,845	7,220	0,181
2	4	5	1	40	3360	7,507	10,106	7,392	0,185
2	4	5	1	40	5040	24,959	10,259	7,738	0,193
2	4	5	1	40	6720	57,699	10,519	7,698	0,192
2	4	5	1	40	8400	113,941	10,404	7,600	0,190
2	4	5	1	40	10080	200,361	10,223	7,791	0,195
2	4	2	3	48	1680	0,952	9,966	7,304	0,152
2	4	2	3	48	3360	8,054	9,42	6,890	0,144
2	4	2	3	48	5040	25,385	10,087	7,608	0,159
2	4	2	3	48	6720	61,267	9,906	7,250	0,151
2	4	2	3	48	8400	116,194	10,202	7,453	0,155
2	4	2	3	48	10080	203,577	10,062	7,668	0,160
2	4	7	1	56	1680	1,011	9,38	6,877	0,123
2	4	7	1	56	3360	8,042	9,433	6,900	0,123
2	4	7	1	56	5040	24,988	10,247	7,729	0,138
2	4	7	1	56	6720	62,665	9,685	7,088	0,127
2	4	7	1	56	8400	115,836	10,234	7,476	0,133
2	4	7	1	56	10080	205,799	9,953	7,585	0,135
2	4	2	4	64	1680	0,933	10,166	7,452	0,116
2	4	2	4	64	3360	8,002	9,481	6,935	0,108
2	4	2	4	64	5040	26,768	9,565	7,215	0,113
2	4	2	4	64	6720	59,916	10,13	7,413	0,116
2	4	2	4	64	8400	115,822	10,235	7,476	0,117
2	4	2	4	64	10080	201,51	10,165	7,746	0,121

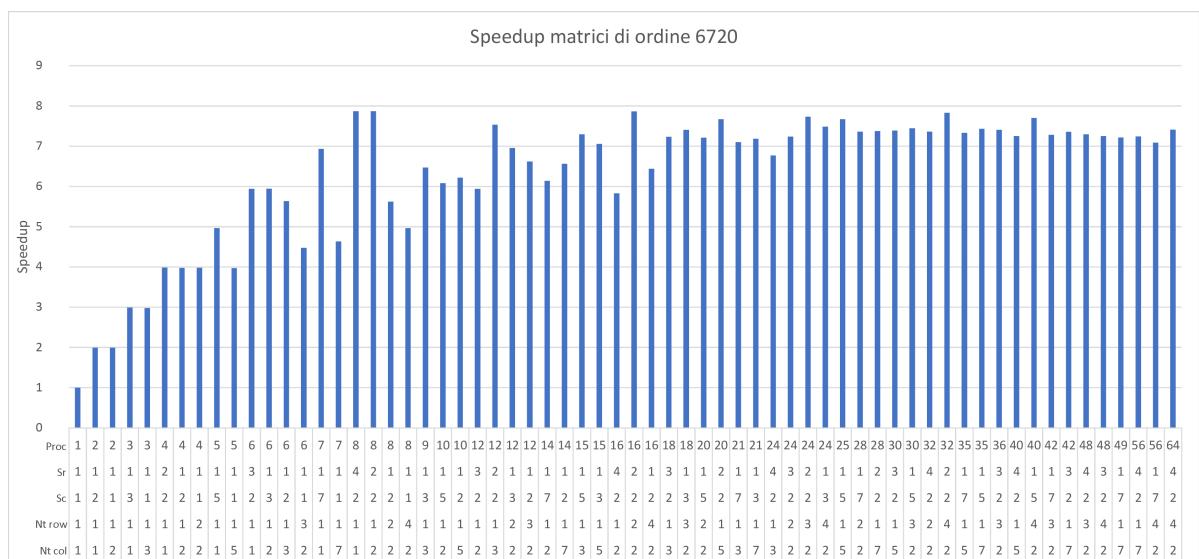
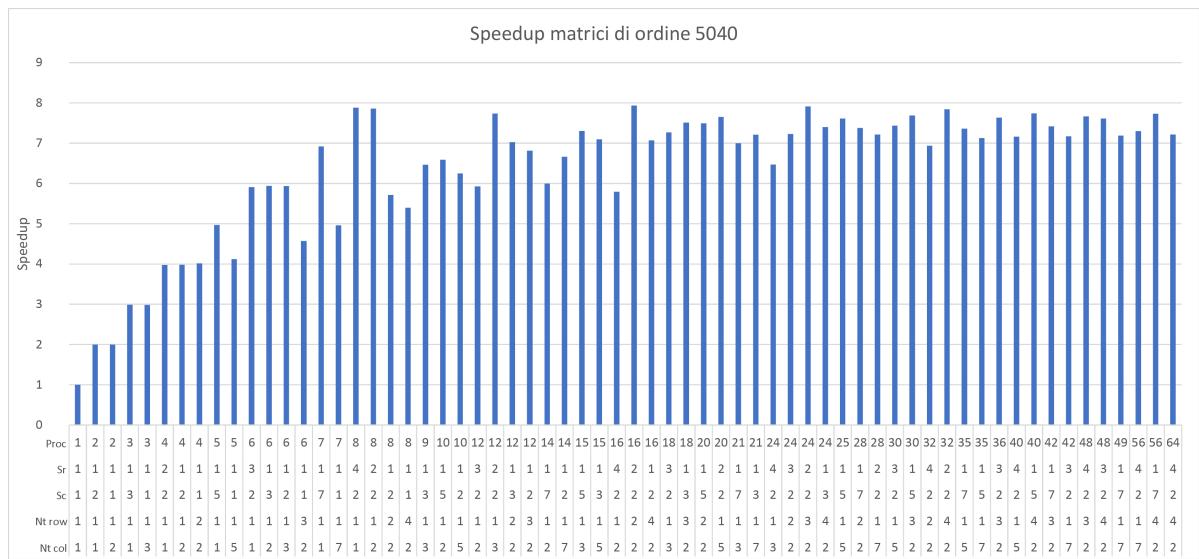
## 2.1 Grafici

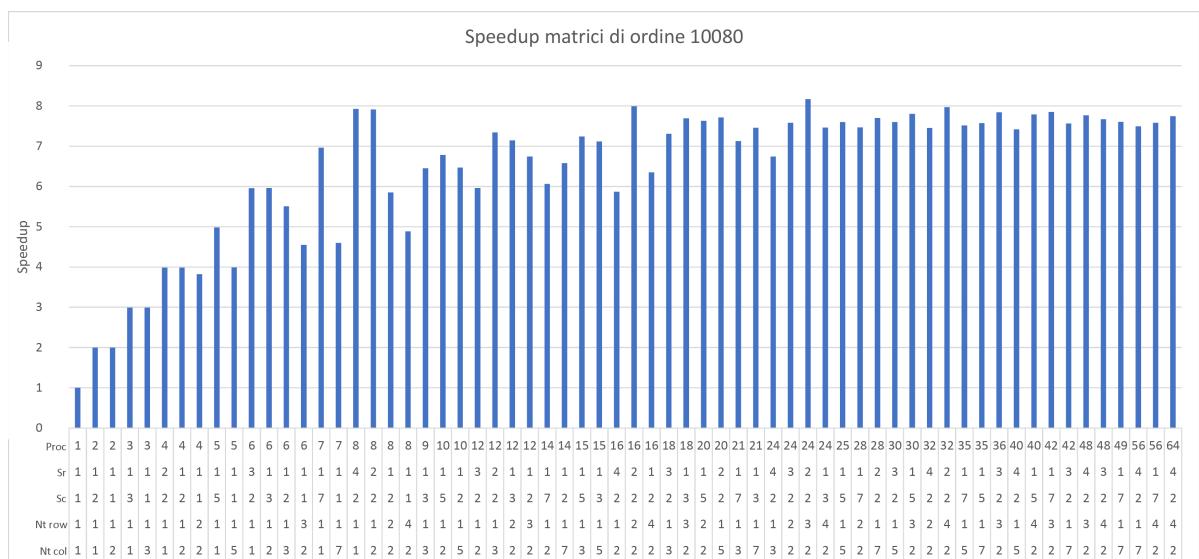
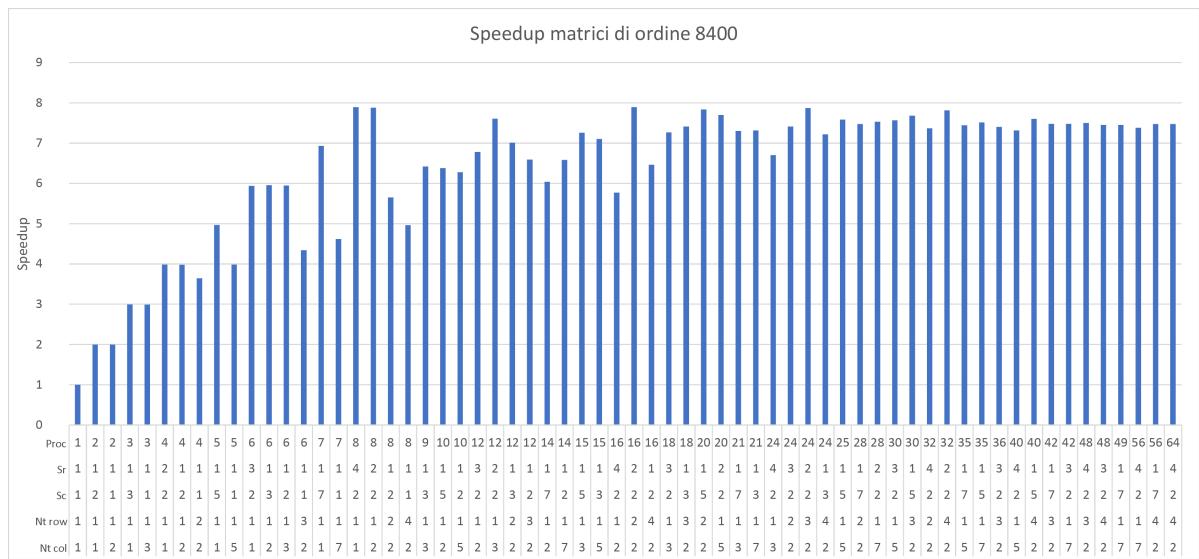












## 2.2 Analisi dei tempi

L'algoritmo SUMMA è stato eseguito su cluster utilizzando la funzione che esegue il prodotto tra matrici con logica a blocchi utilizzando i thread. Sono state testate diverse configurazioni per la divisione dei blocchi tra i processori e il numero di thread utilizzati per effettuare il calcolo. Le configurazioni relative alla dimensione della griglia dei processori e quindi alla divisione dei blocchi sono 8. Ognuna di queste prevede la divisione delle matrici in un numero di blocchi pari ad  $n \times m$ , nello specifico  $n$  ed  $m$  assumono i valori: (1,1),(2,1),(3,1),(2,2),(5,1),(2,3),(7,1) e (2,4). Ognuna di queste configurazioni è stata testata con un numero di processori che va da 1 fino ad 8, ovvero il massimo consentito dall'hardware. Di conseguenza il numero massimo di thread che è possibile far lavorare in parallelo è 64, ottenuto dalla configurazione che opera con una griglia 4x2, quindi 8 processori, e per ognuno di questi vengono utilizzati 8 thread. Le performance sono state valutate per matrici di ordine crescente, a partire da matrici di ordine 1680, fino a matrici di ordine 10080.

Analizzando i grafici ottenuti relativi ai Gflops abbiamo notato un costante miglioramento delle prestazioni per la configurazione (1,1), che nel caso di matrici di ordine 10080 parte da un valore inferiore ai 2 Gflops nel caso in cui utilizziamo un solo processore, fino ad arrivare ad un valore superiore ai 10 Gflops quando di processori ne utilizziamo 8. Questo comportamento si registra anche per matrici di ordine differente. Tuttavia per le altre configurazione il comportamento non è lo stesso, sempre analizzando i risultati delle matrici di ordine maggiore, si nota come per tutte le altre configurazioni, superato un certo numero di processori i Gflops diminuiscano, nel caso della configurazione (2,1) si passa dai circa 10 Gflops registrati su 4 processori, ai circa 9 su 5 processori, per poi stabilizzarsi. Una possibile spiegazione a questo comportamento potrebbe trovarsi nella configurazione hardware del cluster sul quale sono stati eseguiti i test, infatti ogni processore del cluster dispone di 4 core fisici ciascuno dei quali dotato di 2 core logici.

# 3 Appendice: Codice

## 3.1 Utility

```
1
2 #include "utility.h"
3
4 int lcm(int a, int b)
5 {
6     int lcm = (a > b) ? a : b;
7
8     while(lcm % a || lcm % b)
9         ++lcm;
10
11    return lcm;
12 }
13
14 float* rnd_flt_matrix(int n, int m)
15 {
16     int size = n * m;
17     float *A = (float*) malloc(sizeof(float) * size);
18
19     for(int i = 0; i < size; ++i)
20         A[i] = (float) rand() / RAND_MAX;
21
22     return A;
23 }
24
25 float* zeros_flt_matrix(int n, int m)
26 {
27     return (float*) calloc(n * m, sizeof(float));
28 }
29
30 void clear_matrix(float* A, int ld, int n, int m)
31 {
32     for(int i = 0; i < n; ++i)
33         for(int j = 0; j < m; ++j)
34             A[i*ld+j] = 0;
35 }
36
37 int cmp_matrix(float *A, float *B, int lda, int ldb, int n, int m)
38 {
39     int cnt = 0;
40
41     for(int i = 0; i < n; ++i)
42         for(int j = 0; j < m; ++j)
43             if(A[i*lda+j] != B[i*ldb+j])
```

```

44         cnt++;
45     return cnt;
46 }
48
49 void print_matrix(float *A, int lda, int n, int m)
50 {
51     for(int i = 0; i < n; ++i)
52     {
53         for(int j = 0; j < m; ++j)
54             printf("%5.0lf ", A[i*lda+j]);
55
56         putchar('\n');
57     }
58 }
59
60 double time_elapsed(struct timespec start, struct timespec finish)
61 {
62     double elapsed = (finish.tv_sec - start.tv_sec);
63     return elapsed + (finish.tv_nsec - start.tv_nsec) / NSEC;
64 }
65
66 void init_mpi_env(int argc, char** argv, int *nproc, int *menum)
67 {
68     MPI_Init(&argc, &argv);
69     MPI_Comm_rank(MPI_COMM_WORLD, menum); // get processor id
70     MPI_Comm_size(MPI_COMM_WORLD, nproc); // get number of processors
71 }
72
73 void create_cart_grid(int row, int col, int *coords, MPI_Comm* CART_COMM)
74 {
75     int menum_cart;
76
77     MPI_Cart_create(MPI_COMM_WORLD, 2, (int[]){row, col},
78                     (int[]){1, 1}, 1, CART_COMM);
79
80     MPI_Comm_rank(*CART_COMM, &menum_cart);
81
82     MPI_Cart_coords(*CART_COMM, menum_cart, 2, coords);
83 }
84
85 void cart_sub(MPI_Comm CART_COMM, MPI_Comm* ROW_COMM, MPI_Comm* COL_COMM)
86 {
87     MPI_Cart_sub(CART_COMM, (int[]){0, 1}, ROW_COMM);
88     MPI_Cart_sub(CART_COMM, (int[]){1, 0}, COL_COMM);
89 }
90
91 void cp_matrix(float *S, float *D, int n, int m, int lds, int ldd)
92 {
93     for(int i = 0; i < n; i++)
94         for (int j = 0; j < m; j++)

```

```

95         D[i*ldd+j] = S[i*lds+j];
96     }
97
98 void cyclic_distribution(MPI_Comm CART_COMM, int menum, int source,
99                         int Sr, int Sc, int n, int m, int p,
100                        int lda, int ldb, int ldc,
101                        float *A, float *B, float *C,
102                        float **locA, float **locB, float **locC)
103 {
104     float *bufferA, *bufferB, *tmpA, *tmpB, *tmpC;
105
106     int dest;
107     int local_offset, offsetA, offsetB;
108
109    /* calcolo della dimensione delle matrici locali */
110
111    // dimensione dei singoli blocchi
112    int n_size = n / Sr;      // numero di righe
113    int p_size = p / Sc;      // numero di colonne
114
115    // calcolo del numero di blocchi
116    // sulle colonne di A e sulle righe di B
117    int blk_num = lcm(Sr, Sc);
118    int proc_blk_a = blk_num / Sc;
119    int proc_blk_b = blk_num / Sr;
120
121    // numero di colonne di un blocco di A
122    // equivalente al numero di righe di un blocco di B
123    int m_size = m / blk_num;
124
125    int blk_size_a = n_size * m_size;
126    int blk_size_b = m_size * p_size;
127    int blk_size_c = n_size * p_size;
128
129    // allocazione matrici locali
130    tmpA = (float*) malloc(sizeof(float) *
131                           (n_size * m_size) * proc_blk_a);
132
133    tmpB = (float*) malloc(sizeof(float) *
134                           (m_size * proc_blk_b) * p_size);
135
136    tmpC = (float*) calloc(blk_size_c, sizeof(float));
137
138    // allocazione matrici di supporto per l'invio
139    bufferA = (float*) malloc(sizeof(float) * blk_size_a);
140    bufferB = (float*) malloc(sizeof(float) * blk_size_b);
141
142    /* distribuzione ciclica delle matrici */
143
144    local_offset = 0;
145

```

```

146 MPI_Barrier(MPI_COMM_WORLD);
147
148 // distribuzione di A
149 for (int i = 0; i < Sr; i++)
150 {
151     for (int j = 0; j < blk_num; j++)
152     {
153         // calcolo id destinatario
154         dest = (i * Sc) + (j % Sc);
155         offsetA = (i * lda * n_size) + (j * m_size);
156
157         if (menum == source)
158         {
159             if(dest == source)
160             {
161                 cp_matrix(&A[offsetA], &tmpA[local_offset*m_size], n_size, m_size,
162 lda, m_size * proc_blk_a);
163                 local_offset++;
164             }
165             else
166             {
167                 cp_matrix(&A[offsetA], bufferA, n_size, m_size, lda, m_size);
168                 MPI_Send(bufferA, blk_size_a, MPI_FLOAT, dest, 1, CART_COMM);
169             }
170             else if (menum == dest)
171             {
172                 MPI_Recv(bufferA, blk_size_a, MPI_FLOAT, source, 1, CART_COMM,
173 MPI_STATUS_IGNORE);
174                 cp_matrix(bufferA, &tmpA[local_offset*m_size], n_size, m_size, m_size,
175 m_size * proc_blk_a);
176                 local_offset++;
177             }
178         }
179         local_offset = 0;
180
181 // distribuzione di B
182 for (int i = 0; i < blk_num; i++)
183 {
184     for (int j = 0; j < Sc; j++)
185     {
186         // calcolo id destinatario
187         dest = ((i % Sr)*Sc) + j;
188         offsetB = (i * ldb * m_size) + (j * p_size);
189
190         if(menum == source)
191         {
192             if(dest == source)
193             {

```

```

194             cp_matrix(&B[offsetB], &tmpB[local_offset*blk_size_b], m_size,
195             p_size, ldb, p_size);
196             local_offset++;
197         }
198     else
199     {
200         cp_matrix(&B[offsetB], bufferB, m_size, p_size, ldb, p_size);
201         MPI_Send(bufferB, blk_size_b, MPI_FLOAT, dest, 1, CART_COMM);
202     }
203     else if(menum == dest)
204     {
205         MPI_Recv(bufferB, blk_size_b, MPI_FLOAT, source, 1, CART_COMM,
206         MPI_STATUS_IGNORE);
207         cp_matrix(bufferB, &tmpB[local_offset*blk_size_b], m_size, p_size,
208         p_size, p_size);
209         local_offset++;
210     }
211 }
212 *locA = tmpA;
213 *locB = tmpB;
214 *locC = tmpC;
215
216 free(bufferA);
217 free(bufferB);
218 }
219
220 void gather_result(MPI_Comm CART_COMM, int Sr, int Sc, int dest,
221                     int n, int p, int ldc, float* C, float *locC)
222 {
223     int menum, source, offset = 0;
224     int n_size = n / Sr, p_size = p / Sc,
225         c_blk_size = n_size * p_size;
226
227     MPI_Comm_rank(CART_COMM, &menum);
228     float *tmp = (float*) malloc(sizeof(float) * c_blk_size);
229
230     for(int i = 0; i < Sr; i++)
231     {
232         for(int j = 0; j < Sc; j++)
233         {
234             source = i*Sc + j;
235             offset = (i*n_size*ldc) + j*p_size;
236
237             if(menum == dest && source == menum)
238             {
239                 cp_matrix(locC, &C[offset], n_size, p_size, p_size, ldc);
240             }
241             else if(menum != dest && source == menum)

```

```

242     {
243         cp_matrix(locC, tmp, n_size, p_size, p_size, p_size);
244         MPI_Send(tmp, c_blk_size, MPI_FLOAT, dest, 1, CART_COMM);
245     }
246     else if (menum == dest)
247     {
248         MPI_Recv(tmp, c_blk_size, MPI_FLOAT, source, 1, CART_COMM,
249         MPI_STATUS_IGNORE);
250         cp_matrix(tmp, &C[offset], n_size, p_size, p_size, ldc);
251     }
252 }
253
254 free(tmp);
255 }
```

### 3.1: "Codice sorgente utility"

## 3.2 Matmat

```
1
2 #include "matmat.h"
3
4 void matmat(int lda, int ldb, int ldc, int n, int m, int p,
5             float* A, float* B, float* C)
6 {
7     for(int i = 0; i < n; ++i)
8         for(int k = 0; k < m; ++k)
9             for(int j = 0; j < p; ++j)
10                C[i*ldc+j] += A[i*lda+k] * B[k*ldb+j];
11 }
12
13 void matmat_block(int lda, int ldb, int ldc, int n, int m, int p,
14                     float* A, float* B, float* C, int bs)
15 {
16     int i_n, k_m, j_p;
17
18     for(int i = 0; i < n; i += bs)
19         for(int k = 0; k < m; k += bs)
20             for(int j = 0; j < p; j += bs)
21             {
22                 i_n = (i+bs > n) ? n-i : bs;
23                 k_m = (k+bs > m) ? m-k : bs;
24                 j_p = (j+bs > p) ? p-j : bs;
25
26                 matmat(lda, ldb, ldc, i_n, k_m, j_p,
27                         &A[i*lda+k], &B[k*ldb+j], &C[i*ldc+j]);
28             }
29 }
30
31 void matmat_threads(int ntrow, int ntcoll, int lda, int ldb, int ldc,
32                      int n, int m, int p, float* A, float* B, float* C, int bs)
33 {
34     int TH_N = ntrow * ntcoll, th_cnt = 0;
35     int sub_n, rest_n, sub_j, rest_j, span_n, span_j, offset_n, offset_j;
36     pthread_t tid[TH_N];
37     mat_struct th_arg[TH_N];
38
39     sub_n = n / ntrow;
40     rest_n = n % ntrow;
41
42     sub_j = p / ntcoll;
43     rest_j = p % ntcoll;
44
45     offset_n = 0;
46
47     for(int i = 0; i < ntrow; ++i)
48     {
```

```

49     span_n = (i < rest_n) ? sub_n + 1 : sub_n;
50
51     offset_j = 0;
52
53     for(int j = 0; j < ntncl; ++j)
54     {
55         span_j = (j < rest_j) ? sub_j + 1: sub_j;
56
57         th_arg[th_cnt].lda = lda;
58         th_arg[th_cnt].ldb = ldb;
59         th_arg[th_cnt].ldc = ldc;
60
61         th_arg[th_cnt].n = span_n;
62         th_arg[th_cnt].m = m;
63         th_arg[th_cnt].p = span_j;
64
65         th_arg[th_cnt].A = &A[offset_n * lda];
66         th_arg[th_cnt].B = &B[offset_j];
67         th_arg[th_cnt].C = &C[offset_n * ldc + offset_j];
68
69         th_arg[th_cnt].bs = bs;
70
71         pthread_create(&tid[th_cnt], NULL,
72                         matmat_thread, &th_arg[th_cnt]);
73         th_cnt++;
74
75         offset_j += span_j;
76     }
77     offset_n += span_n;
78 }
79
80 for(int i = 0; i < TH_N; ++i)
81     pthread_join(tid[i], NULL);
82 }
83
84 void* matmat_thread(void* mat_th_struct)
85 {
86     mat_struct *s = (mat_struct*) mat_th_struct;
87     matmat_block(s->lda, s->ldb, s->ldc, s->n, s->m, s->p,
88                  s->A, s->B, s->C, s->bs);
89 }
90
91 void SUMMA(MPI_Comm ROW_COMM, MPI_Comm COL_COMM, int Sr, int Sc,
92             int ntrow, int ntncl, int lda, int ldb, int ldc,
93             int n, int m, int p, float* locA, float* locB, float* Cloc, int bs)
94 {
95     int c, r, id_col, id_row,
96         offsetA, offsetB;
97
98     float *A_tmp, *B_tmp;
99

```

```

100     int n_size = n / Sr;
101     int p_size = p / Sc;
102
103     int blk_num = lcm(Sr, Sc);
104     int m_size = m / blk_num;
105
106     int blk_size_a = n_size * m_size;
107     int blk_size_b = m_size * p_size;
108
109     A_tmp = (float*) malloc(sizeof(float) * blk_size_a);
110     B_tmp = (float*) malloc(sizeof(float) * blk_size_b);
111
112     MPI_Comm_rank(ROW_COMM, &id_row);
113     MPI_Comm_rank(COL_COMM, &id_col);
114
115     offsetA = offsetB = 0;
116
117     for(int k = 0; k < blk_num; k++)
118     {
119         r = k % Sc;
120         c = k % Sr;
121
122         if(id_row == r)
123         {
124             cp_matrix(&locA[offsetA], A_tmp, n_size, m_size, lda, m_size);
125             offsetA += m_size;
126         }
127
128         if(id_col == c)
129         {
130             cp_matrix(&locB[offsetB], B_tmp, m_size, p_size, ldb, p_size);
131             offsetB += (m_size * ldb);
132         }
133
134         MPI_Bcast(A_tmp, blk_size_a, MPI_FLOAT, r, ROW_COMM);
135         MPI_Bcast(B_tmp, blk_size_b, MPI_FLOAT, c, COL_COMM);
136
137         matmat_threads(ntrow, ntcoll, m_size, p_size, ldc,
138                         n_size, m_size, p_size, A_tmp, B_tmp, Cloc, bs);
139     }
140
141     free(A_tmp);
142     free(B_tmp);
143 }
```

### 3.2: "Codice sorgente matmat"

### 3.3 Main

```
1 #include "include/matmat.h"
2
3 #define MAT_SIZE 10080
4
5 int main(int argc, char** argv)
6 {
7     int menum, nproc, coords[2], Sr, Sc,
8         ntrow, ntcoll, bs;
9
10    int ldlocA, ldlocB, ldlocC;
11
12    long nop;
13
14    double start, stop, proc_time, mpi_time,
15        tot_mpi_time, gflops_mpi;
16
17    float *A, *B, *C, *C2, *locA, *locB, *locC;
18
19    struct timespec start_timer_mpi, finish_timer_mpi;
20
21    MPI_Comm CART_COMM, ROW_COMM, COL_COMM;
22
23    init_mpi_env(argc, argv, &nproc, &menum);
24
25    bs = 720;
26
27    if (menum == 0)
28    {
29        if( argc > 4)
30        {
31            Sr = atoi(argv[1]);
32            Sc = atoi(argv[2]);
33            ntrow = atoi(argv[3]);
34            ntcoll = atoi(argv[4]);
35        }
36        else
37            MPI_Abort(MPI_COMM_WORLD, -1);
38
39        // inizializzazione con valori random delle matrici come vettori
40        A = rnd_flt_matrix(MAT_SIZE, MAT_SIZE);
41        B = rnd_flt_matrix(MAT_SIZE, MAT_SIZE);
42        // matrice risultante inizializzata con zeri
43        C = zeros_flt_matrix(MAT_SIZE, MAT_SIZE);
44        // DEBUG matrix for check result
45        //C2 = zeros_flt_matrix(MAT_SIZE, MAT_SIZE);
46    }
47
48    MPI_Bcast(&Sr, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

```

49 MPI_Bcast(&Sc, 1, MPI_INT, 0, MPI_COMM_WORLD);
50
51 // creazione griglia
52 create_cart_grid(Sr, Sc, coords, &CART_COMM);
53 cart_sub(CART_COMM, &ROW_COMM, &COL_COMM);
54
55 if(menum == 0)
56 {
57     printf("proc %d (%d, %d)- threads %d (%d, %d)\n",
58           nproc, Sr, Sc, ntrow*ntcol, ntrow, ntcoll);
59     printf("Matrix order; Time; GFlops\n");
60 }
61
62 for (int i = 1680; i <= 5040; i += 1680)
63 {
64     tot_mpi_time = 0.;
65
66     ldlocA = i/Sc;
67     ldlocB = i/Sc;
68     ldlocC = i/Sc;
69
70     // allocazione matrici locali
71     // e distribuzione ciclica delle matrici A e B
72     cyclic_distribution(CART_COMM, menu, 0, Sr, Sc, i, i, i,
73                         MAT_SIZE, MAT_SIZE, MAT_SIZE, A, B, C,
74                         &locA, &locB, &locC);
75
76     MPI_Barrier(MPI_COMM_WORLD);
77     clock_gettime(CLOCK_MONOTONIC, &start_timer_mpi);
78
79     // chiamata alla funzione SUMMA
80     SUMMA(ROW_COMM, COL_COMM, Sr, Sc, ntrow, ntcoll,
81           ldlocA, ldlocB, ldlocC, i, i, i, locA, locB, locC, bs);
82
83     clock_gettime(CLOCK_MONOTONIC, &finish_timer_mpi);
84     proc_time = time_elapsed(start_timer_mpi, finish_timer_mpi);
85
86     MPI_Barrier(MPI_COMM_WORLD);
87     MPI_Reduce(&proc_time, &mpi_time, 1, MPI_DOUBLE,
88                 MPI_MAX, 0, MPI_COMM_WORLD);
89
90     // mette insieme i risultati locali
91     gather_result(CART_COMM, Sr, Sc, 0, i, i, MAT_SIZE, C, locC);
92
93     if(menum == 0)
94         tot_mpi_time += mpi_time;
95
96     free(locA);
97     free(locB);
98     free(locC);
99

```

```

100     if (menum == 0)
101     {
102         nop = 2 * pow(i, 3);
103         gflops_mpi = (nop / tot_mpi_time) / GIGA;
104         printf("%d; %8.3lf; %8.3lf;\n", i, tot_mpi_time, gflops_mpi);
105         // -- DEBUG check result (time consuming) -- //
106         // matmat_threads(1,1, MAT_SIZE, MAT_SIZE, MAT_SIZE,
107         //                 i, i, i, A, B, C2, 150);
108         // long errors = cmp_matrix(C, C2, MAT_SIZE, MAT_SIZE, i, i);
109         // printf("error: %ld\n", errors);
110         // clear_matrix(C2, MAT_SIZE, i, i);
111     }
112 }
113
114 if (menum == 0)
115 {
116     free(A);
117     free(B);
118     free(C);
119 }
120
121 MPI_Finalize();
122
123 return 0;
124 }
```

### 3.3: "Codice sorgente *main*"