Relazione sul prodotto matrice-matrice
con un algoritmo parallelo ibrido

Corso di Calcolo Parallelo e Distribuito mod. B
Prof. M. Lapegna

Alessandro Serrapica N97000213

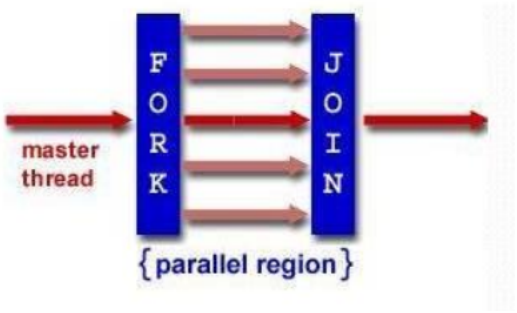
Giugno 2016

Indice

1 Introduzione

In questo progetto si é sviluppato un algoritmo per la risoluzione del prodotto matriciale, che fonde le due principali tecniche per lo sviluppo di algoritmi paralleli costituendo, quindi, una soluzione ibrida tra algoritmi a memoria distribuita e algoritmi a memoria condivisa.
Negli algoritmi a memoria distribuita l'utilitá dello strutturare gli algoritmi affinché lavorino su piú processori contemporaneamente é la divisione del carico di operazioni da svolgere, che velocizza il tempo di esecuzione. Un reale vantaggio si ottiene quando il numero di dati su cui lavorare é molto grande, essendo necessarie delle comunicazioni tra i processori. Inoltre la strutturazione di un algoritmo per il lavoro su piú processori aiuta a superare i limiti fisici dei singoli processori, ovvero riduce i tempi di calcolo per algoritmi che, se strutturati per il lavoro sequenziale su di un solo processore, non potrebbero scendere al di sotto di una soglia determinata dall'avanzamento tecnologico delle strutture fisiche, come la CPU. Il nucleo della strutturazione parallela di un algoritmo é la divisione in $\frac{N}{p}$ sotto problemi da passare ad ogni processore, ricomponendo poi i risultati per ottenere l'output cercato.
Negli algoritmi a memoria condivisa, invece, le componenti che eseguono operazioni concorrentemente non sono differenti processori, ma core di uno stesso processore. Sará quindi necessario suddividere le istruzioni in thread e far sí che vengano eseguite contemporaneamente istruzioni di thread diversi (si parla di multithreading). Per farlo, in questo corso avanzato di Calcolo Parallelo e Distribuito, si é fatto ricorso ai Pthreads (Posix Threads).
Il metodo di parallelizzazione implementato é il fork-join:

- *Fork*: un master thread apre quella che si dice una "regione parallela" creando un certo numero di thread (team di thread), ciascuno dei quali esegue delle istruzioni.
- *Join*: al termine di tali esecuzioni i threads si sincronizzano e terminano, facendo sopravvivere solo il master thread.



Tornando al nostro algoritmo, quindi, esso sfrutterá quanto detto sugli algoritmi a memoria distribuita dividendo le matrici in sottomatrici che saranno distribuite ai diversi processori, i quali, a loro volta, coerentemente con quanto descritto per gli algoritmi a memoria condivisa, suddivideranno virtualmente le loro matrici locali assegnando zone specifiche di esse ai loro thread. Infine, ciascun thread, ricorrerá ad una elementare funzione per il prodotto matriciale ottimizzata peró nella combinazione degli indici che assicura prestazioni migliori scorrendo le matrici in modo da garantire il minor numero di accessi in memoria. Ció é legato al modo in cui il linguaggio C salva le matrici e verrá mostrato nel paragrafo §3.3.

2 Descrizione dell'algoritmo

2.1 Algoritmo di Cannon

L'algoritmo di Cannon calcola il prodotto tra due matrici in parallelo con p processori.
Si considera una griglia di processori di dimensione $\sqrt{p} \times \sqrt{p}$. Le matrici A e B vengono distribuite tra i processori in blocchi di dimensioni $\frac{N}{\sqrt{p}} \times \frac{N}{\sqrt{p}}$. Per semplicitá verrá mostrato l'algoritmo nel caso di 9 processori e poi sará

esteso al caso generale.
Siano quindi

p_0	p_1	p_2
p_3	p_4	p_5
p_6	p_7	p_8

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$
$A_{1,0}$	$A_{1,1}$	$A_{1,2}$
$A_{2,0}$	$A_{2,1}$	$A_{2,2}$

$B_{0,0}$	$B_{0,1}$	$B_{0,2}$
$B_{1,0}$	$B_{1,1}$	$B_{1,2}$
$B_{2,0}$	$B_{2,1}$	$B_{2,2}$

rispettivamente la griglia di processori e le due matrici divise in blocchi. Ogni processore per completare il blocco di C che gli compete deve ovviamente calcolare

$$C_{i,j} = \sum_{k=0}^{2} A_{i,k} B_{k,j}$$

Consideriamo ad esempio p_7. Questo processore deve calcolare

$$C_{2,1} = A_{2,0}B_{0,1} + A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$$

Per come sono distribuiti inizialmente i blocchi, si deve effettuare un'inclinazione iniziale: i blocchi di A devono slittare di due posizioni verso sinistra, quelli di B di una posizione verso l'alto. Si puó vedere con semplici calcoli che in generale il blocco di $A_{i,j}$ deve slittare di i posizioni verso sinistra, il blocco $B_{i,j}$ di j posizioni verso l'alto. Alla fine di questo procedimento le matrici saranno cosí distribuite

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$
$A_{1,1}$	$A_{1,2}$	$A_{1,0}$
$A_{2,2}$	$A_{2,0}$	$A_{2,1}$

$B_{0,0}$	$B_{1,1}$	$B_{2,2}$
$B_{1,0}$	$B_{2,1}$	$B_{0,2}$
$B_{2,0}$	$B_{0,1}$	$B_{1,2}$

A questo punto ogni processore puó calcolare la prima parte del corrispondente blocco di C. Gli altri due blocchi da sommare a quello appena calcolato si ottengono facendo slittare tutti i blocchi di A a sinistra di una posizione e tutti quelli di B di una posizione in alto. Infatti, dopo il primo shift si ottengono le seguenti matrici

$A_{0,1}$	$A_{0,2}$	$A_{0,0}$
$A_{1,2}$	$A_{1,0}$	$A_{1,1}$
$A_{2,0}$	$A_{2,1}$	$A_{2,2}$

$B_{1,0}$	$B_{2,1}$	$B_{0,2}$
$B_{2,0}$	$B_{0,1}$	$B_{1,2}$
$B_{0,0}$	$B_{1,1}$	$B_{2,2}$

e dopo il secondo

$A_{0,2}$	$A_{0,0}$	$A_{0,1}$
$A_{1,0}$	$A_{1,1}$	$A_{1,2}$
$A_{2,1}$	$A_{2,2}$	$A_{2,0}$

$B_{2,0}$	$B_{0,1}$	$B_{1,2}$
$B_{0,0}$	$B_{1,1}$	$B_{2,2}$
$B_{1,0}$	$B_{2,1}$	$B_{0,2}$

In generale l'operazione che qui é stata eseguita 2 volte deve essere eseguita $\sqrt{p} - 1$ volte.

2.2 Algoritmo multithread per il prodotto matriciale

Come detto, l'algoritmo sfrutta la memoria condivisa per ottimizzare il calcolo del prodotto matriciale. A tal fine ciascun processore assegna una parte delle proprie matrici locali A e B ai thread messi a sua disposizione. In particolare, tali sottomatrici hanno dimensione $\frac{N}{\sqrt{t}}$, dove t é il numero di thread. Quindi ciascun thread con un'opportuna chiamata effettuerá $\sqrt{t}$ volte la funzione 'matikj'. Questo é sufficiente al calcolo corretto della matrice risultato in quanto la funzione 'matikj' implementata é della forma $C = C + AB$.

2.3 Complessitá di tempo e di spazio

Ogni processore, che ha inizialmente matrici di dimensione $\frac{N}{\sqrt{p}}$ effettua i seguenti passaggi:
`Inclinazione iniziale`

```
for i=0 to √p − 1
Chiama la funzione matmatthread
if (i<√p − 1)
Shift di A
Shift di B
endif
endfor
```

La funzione 'matmatthread' crea t thread. Ogni thread esegue le seguenti istruzioni

```
for i=0 to √t
Chiama la funzione matijk su sottomatrici di dimensione N/(√p√t)
endfor
```

Quindi si ha

$$T_{p,t} = T_{com} + T_{op}$$

dove

$$T_{com} = 2\left(T_{lat} + \frac{N}{\sqrt{p}}\frac{N}{\sqrt{p}}t_{com}\right) + 2\left(T_{lat} + \frac{N}{\sqrt{p}}\frac{N}{\sqrt{p}}t_{com}\right)(\sqrt{p}-1) = 2\sqrt{p}\left(T_{lat} + \frac{N^2}{p}t_{com}\right)$$

e

$$T_{op} = \sqrt{p}\sqrt{t}\left(\frac{N}{\sqrt{p}\sqrt{t}}\right)^3 t_{op} = \cancel{\sqrt{p}\sqrt{t}}\frac{N^3}{pt\cancel{\sqrt{p}\sqrt{t}}}t_{op} = \frac{N^3}{pt}t_{op}$$

In definitiva

$$T_{p,t} = \frac{N^3}{pt}t_{op} + 2\sqrt{p}\left(T_{lat} + \frac{N^2}{p}t_{com}\right)$$

Per quanto riguarda la complessitá di spazio, ogni processore alloca 3 matrici di dimensione $\frac{N}{\sqrt{p}} \times \frac{N}{\sqrt{p}}$ e 4 buffer di dimensione $\frac{N^2}{p}$. In totale lo spazio occupato da tutti i p processori é

$$S_p = 7N^2$$

Si noti che ovviamente il numero di thread non influisce sulla complessitá di spazio e sul tempo di comunicazione, essendo eseguiti in memoria condivisa.

3 Implementazione

In questo paragrafo si analizzano gli aspetti principali degli algoritmi implementati.

3.1 Algoritmo di Cannon

3.1.1 Variabili utilizzate

Variabili di input:

- MPI Comm comm : Comunicatore MPI per assicurare le comunicazioni tra i processori
- Int LDA : La Leading Dimension della matrice A
- Int LDB : La Leading Dimension della matrice B
- Int LDC : La Leading Dimension della matrice C
- Int N : il numero di righe di ciascuna matrice
- Float A[ ][LDA] : La matrice A

- Float B[ ][LDB] : La matrice B
- Int nt : Numero di thread

Variabili locali:

- Float Abuffero: Il buffer di output della matrice A per le comunicazioni tra processori
- Float Abufferi: Il buffer di input della matrice A per le comunicazioni tra processori
- Float Bbuffero: Il buffer di output della matrice B per le comunicazioni tra processori
- Float Bbufferi: Il buffer di output della matrice B per le comunicazioni tra processori
- Int Rank : il codice identificativo di ciascun processore
- Int nproc : il numero di processori utilizzati
- MPI Status status : una variabile di supporto alla funzione MPI Recv
- MPI Request request: una variabile di supporto alle funzioni MPI Isend e MPI Irecv
- Int nrow : numero di righe delle matrici locali di ciascun thread
- Int ncol : numero di colonne delle matrici locali di ciascun thread
- Int size : numero di righe e colonne della griglia dei processori
- Int myrow : riga della griglia del processore con codice identificativo rank
- Int mycol : colonna della griglia del processore con codice identificativo rank
- Int dest : il codice identificativo del processore destinatario del messaggio
- Int mitt : il codice identificativo del processore mittente del messaggio

Variabili di output:

- Float C[ ][LDC] : La matrice risultato C

3.1.2 Inclinazione iniziale

L'inclinazione iniziale prevede che tutti gli elementi della matrice A posti sull'i-esima riga si spostino a sinistra di i posizioni.
Ad esempio, se il numero dei processori é 9 , il processore p_7 posto sulla riga di indice 2 invierá i suoi elementi al processore p_8 e li riceverá dal processore p_6.
In generale il processore di identificativo *rank* invierá e riceverá rispettivamente dai processori *dest* e *mitt* cosí definiti.

```
//INCLINAZIONE INIZIALE  DI A. GLI ELEMENTI POSTI SULLA I-ESIMA RIGA SHIFTANO DI I POSIZIONI A SINISTRA

//CALCOLO IL RANK DEL PROCESSORE DESTINATARIO DEI DATI
int dest = rank - myrow;
if (dest < myrow*size)
    dest=dest + size;

//CALCOLO IL RANK DEL PROCESSORE CHE INVIERA' I DATI A QUESTO PROCESSORE
int mitt = rank + myrow;
if (mitt >= (myrow+1)*size)
    mitt=mitt-size;
```

Inoltre l'inclinazione iniziale prevede che tutti gli elementi della matrice B posti sull'i-esima colonna si spostino in

alto di i posizioni. Ad esempio, se il numero dei processori é 9, il processore p_7 posto sulla colonna 1 invierá i suoi elementi al processore p_4 e li riceverá dal processore p_1.
In generale il processore di identificativo *rank* invierá e riceverá rispettivamente dai processori *dest* e *mitt* cosí definiti.

```
//INCLINAZIONE INIZIALE  DI B. GLI ELEMENTI POSTI SULLA I-ESIMA COLONNA SHIFTANO DI I POSIZIONI IN ALTO

//CALCOLO IL RANK DEL PROCESSORE DESTINATARIO DEI DATI
dest = mod(rank-size*mycol,nproc);

//CALCOLO IL RANK DEL PROCESSORE DESTINATARIO DEI DATI
mitt = mod(rank+size*mycol,nproc);
```

3.1.3 Shift

Gli shift vengono eseguiti nel ciclo dell'algortimo di Cannon. In particolare essi saranno eseguiti $size - 1$ volte dove si ricorda che $size = \sqrt{p}$.
Per quanto riguarda la matrice A tutti gli elementi invieranno i propri dati al processore posto sulla griglia alla propria sinistra. Ad esempio, se il numero dei processori é 9, il processore p_7 invierá i suoi elementi al processore p_6 e li riceverá dal processore p_8.
In generale il processore di identificativo *rank* invierá e riceverá rispettivamente dai processori *dest* e *mitt* cosí definiti.

```
//SHIFT DI TUTTI GLI ELEMENTI DI A DI UNO A SINISTRA.
//CALCOLO, COME SEMPRE, DESTINATARIO E MITTENTE
if(mycol == 0 )
    dest=rank+(size-1);
else
    dest=rank - 1;
if(mycol == (size-1))
    mitt = (rank-size)+1;
else
        mitt= rank + 1;
```

Per quanto riguarda, infine, la matrice B tutti gli elementi invieranno i propri dati al processore posto sulla griglia sopra di essi ovvero quel processore con indice di colonna (mycol) uguale e indice di riga (myrow) inferiore di 1. Quindi:

```
//SHIFT DI TUTTI GLI ELEMENTI DI B DI UNO SOPRA.
//CALCOLO, COME SEMPRE, DESTINATARIO E MITTENTE
if(myrow == 0 )
    dest=rank+(size*(size-1));
else
    dest=rank - size;
if(myrow == (size-1))
    mitt = rank-(size*(size-1));
else
        mitt= rank + size;
```

3.2 Algoritmo multithread

3.2.1 Variabili utilizzate

Variabili di input

- Int N : il numero di righe della matrice
- Int LDA: la Leading Dimension di A
- Int LDB : la leading Dimension di B
- Int LDC : la leading Dimension di C

- Int nt : il numero di thread per ciascun processore
- Int nrow : il numero di righe della sottomatrice di ciascun thread
- Int ncol : il numero di colonne della sottomatrice di ciascun thread
- Float A[ ][LDA] : la matrice A
- Float B[ ][LDB] : la matrice B
- Float C[ ][LDC] : la matrice C

Variabili locali

- Infostruct info: una struttura dati d'appoggio per ciascun thread
- Pthread t thread id : vettore degli id dei thread
- Int irow : indice locale di riga per effettuare un'opportuna chiamata a matikj
- Int icol : indice locale di colonna per effettuare un'opportuna chiamata a matikj

3.2.2 Creazione dei thread e suddivisione delle matrici

La creazione dei thread é delegata alla funzione 'matmatthread'. Essa alloca le strutture dati necessarie alla creazione dei thread, le popola ed infine chiama la funzione 'pthread-create(..)' alla quale tra gli altri dati viene passato il riferimento alla funzione che ciascun thread dovrá eseguire.
In questa funzione i thread considerando irow ed icol, i blocchi di matrice di loro pertinenza e le leading dimension riescono ad effettuare un'opportuna chiamata alla funzione matikj.

```
//ESEGUO RADICE QUADRATA DEL NUMERO DI THREAD VOLTE L'ALGORITMO BASE DI MATRICE PER MATRICE
int j=0;
for(j=0; j<sqrt(nt); j++) {

    //LANCIO L'ALGORTIMO MATRICE PER MATRICE PASSANDO CORRETTAMENTE I PUNTI DI PARTENZA DI CIASCUNA MATRICE.
    //I PUNTI VENGONO TRASMESSI ATTRAVERSO LO SPOSTAMENTO.(OFFSET)
    Matikj(LDA,LDB,LDC,N/Nb,(float (*)[])(A+(N/Nb)*(irow*LDA+j)),(float (*)[])(B+(N/Nb)*(LDB*j+icol)),(float (*)[])(C+(N/Nb)*(irow*LDC+icol)));

}
```

3.3 Algoritmo per il prodotto matriciale

Come anticipato, é possibile ottimizzare la funzione per il prodotto di due matrici $C = AB$ diminuendo gli accessi in memoria centrale e rendendo piú frequenti quelli in memoria cache. In particolare tale ottimizzazione puó essere fatta scegliendo un'opportuna permutazione degli indici delle tre iterazioni necessarie a calcolare il prodotto. Poiché nel linguaggio C le matrici vengono salvate all'interno della memoria per righe, i blocchi delle matrici salvate in cache conterranno (per N non troppo grande) almeno un'intera riga della matrice stessa. Pertanto la permutazione di indici ottimale é quella in cui l'indice di scorrimento delle colonne di C regola il ciclo piú interno. Risulta evidente, inoltre, che nel caso in cui il secondo indice piú interno sia k, cioé quello che permette di scorrere le colonne di A e le righe di B, bisogna ad ogni iterazione aggiornare in cache solo il blocco relativo alla matrice B mentre restano invariati quelli di A e di C.
Nella tabella sottostante si riportano i tempi di esecuzione del prodotto tra matrici di dimensione $N = 2000$ al variare delle permutazioni degli indici.

ikj	6.052140
kij	6.304449
ijk	53.843947
jik	54.225775
kji	137.941120
jki	139.307539

4 Analisi dei tempi

Ricordiamo le definizioni di "speed-up" ed efficienza date per gli algoritmi eseguiti in parallelo tra p processori.
Si dice "speed-up" dell'algoritmo la quantitá

$$S_p = \frac{T_1}{T_p}$$

Tale quantitá é nel caso ideale uguale a p. Si definisce invece efficienza la quantitá

$$E_p = \frac{S_p}{p} = \frac{T_1}{pT_p}$$

L'efficienza vale 1 nel caso ideale.
I tempi analizzati sono calcolati nella funzione 'main' subito prima di chiamare la funzione 'Cannon' e subito dopo di essa.
I valori di N utilizzati nei paragrafi successivi si riferiscono alla dimensione totale della matrice, quindi in realtá sono stati eseguiti con $N_{loc} = \sqrt{\frac{N}{p}}$.

4.1 $N = 10^4$

Nella tabella sottostante sono riportati i tempi di esecuzione al variare di p e t con $N = 10^4$.

t \ p	1	4	9
1	0.002835	0.030180	0.047358
4	0.002402	0.023299	0.0461250

Quindi, in questo caso, non conviene utilizzare piú processori: si vede infatti che i tempi migliori si ottengono per $p = 1$. Si ottiene, invece, un miglioramento utilizzando 4 thread. Questo risultato non é sorprendente essendo N relativamente piccolo: il tempo impiegato per le comunicazioni prevale su quello delle operazioni.
Nella tabella sottostante é riportato il numero di operazioni al secondo (Gflop/s) al variare di p e t con $N = 10^4$.

t \ p	1	4	9
1	0.7055	0.0663	0.0422
4	0.8326	0.0858	0.0434

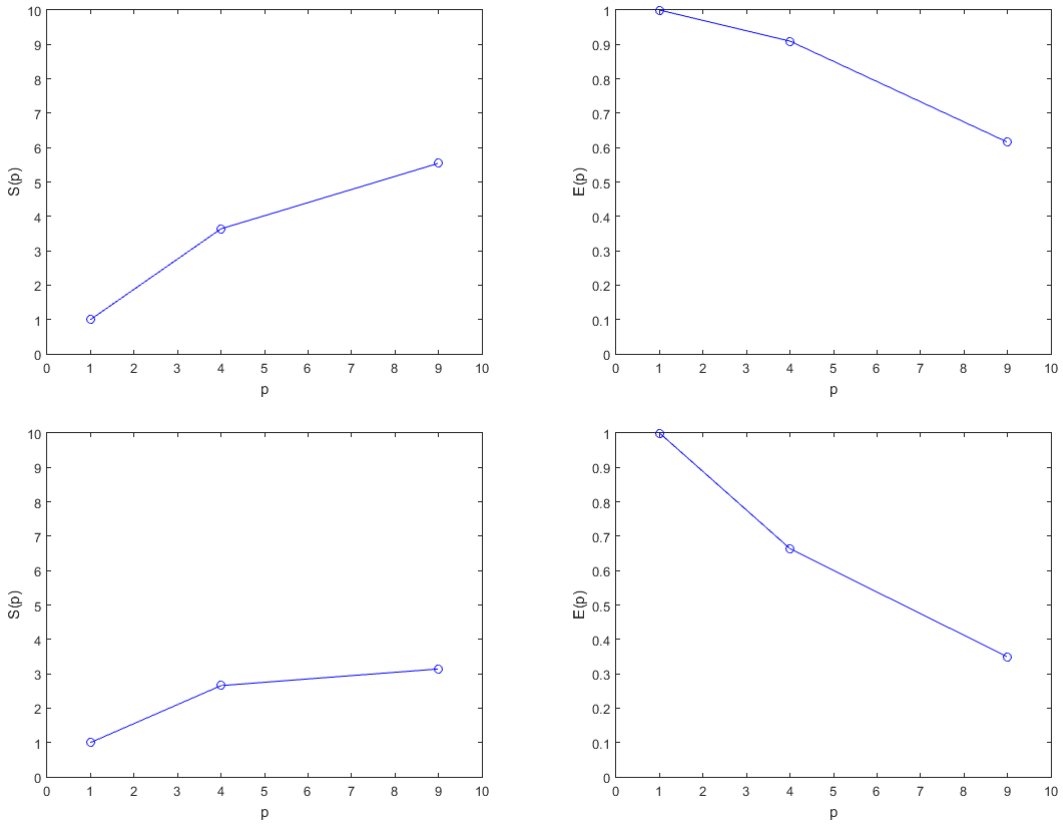
Si puó notare come i valori siano poco incoraggianti circa un impiego di soluzioni in parallelo.

4.2 $N = 10^6$

Nella tabella sottostante sono riportati i tempi di esecuzione al variare di p e t con $N = 10^6$.

t \ p	1	4	9
1	1.489598	0.494570	0.2688717
4	0.401576	0.151234	0.127937

In questo caso il tempo minore di esecuzione si ottiene, come ci si aspetta, per $p = 9$ e $t = 4$. Osserviamo che continua ad essere piú veloce l'esecuzione con 4 thread e un processore rispetto a quella con $p = 4$ e $t = 1$. Quindi le comunicazioni prevalgono ancora sulle operazioni matriciali.
I grafici che seguono mostrano lo speed-up e l'efficienza al variare dei processori con $t = 1$ e $t = 4$.



Nella tabella sottostante é riportato il numero di operazioni al secondo (Gflop/s) al variare di p e t con $N = 10^6$.

t \ p	1	4	9
1	1.3426	4.8845	7.4428
4	4.9804	13.2245	15.6327

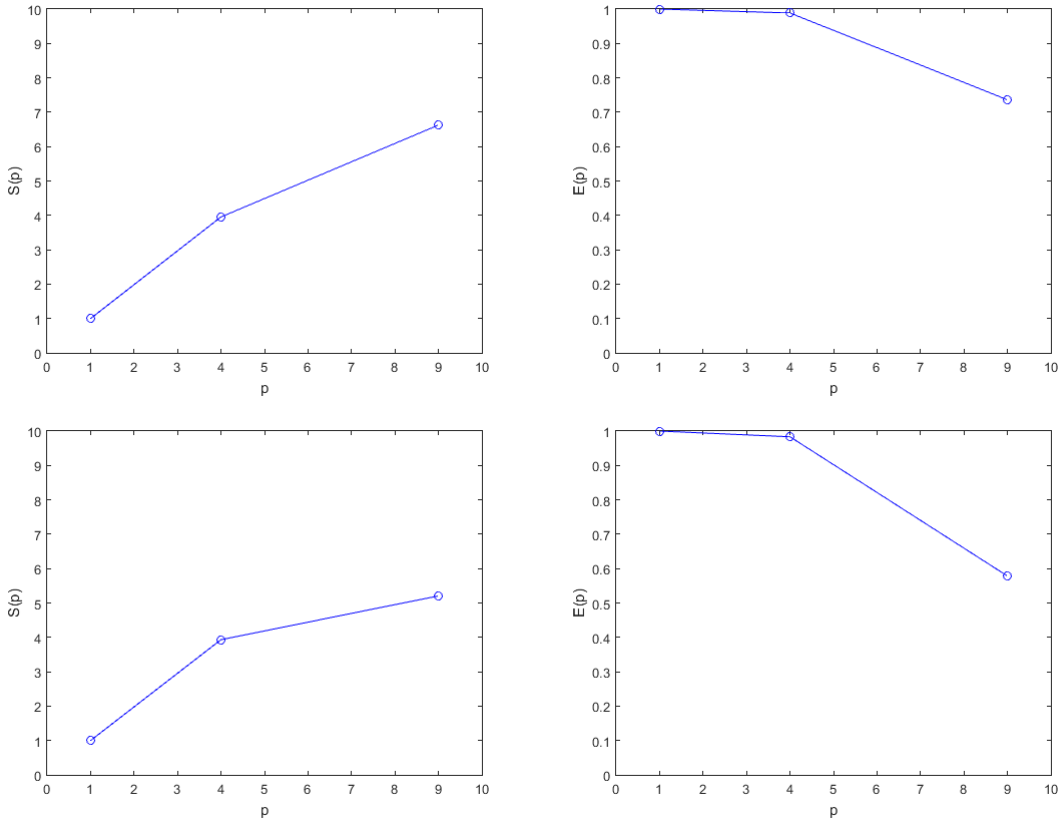
In questo caso, invece, si puó notare come i valori siano piú coerenti in quanto aumentano piú o meno proporzionalmente all'aumentare degli elementi di computazione. Si ricorda che il nono processore é solo simulato.

4.3 $N = 10^8$

Nella tabella sottostante sono riportati i tempi di esecuzione al variare di p e t con $N = 10^8$.

t \ p	1	4	9
1	1646.198341	416.283051	248.451935
4	719.868115	183.020462	138.283307

Anche per $N = 10^8$ il tempo migliore resta quello per $p = 9$ e $t = 4$. Inoltre, in questo caso le operazioni hanno un peso maggiore che nei casi precedenti, per cui l'esecuzione con 4 processori e 1 thread é piú vantaggiosa rispetto a quella con $p = 1$ e $t = 4$.
I grafici seguenti mostrano lo speed-up e l'efficienza al variare di p con $t = 1$ e $t = 4$.



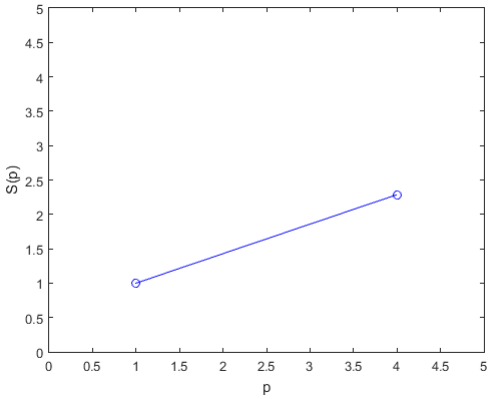
I valori ottenuti in questo caso sono piú vicini a quelli ideali.
Nella tabella sottostante é riportato il numero di operazioni al secondo (Gflop/s) al variare di p e t con $N = 10^8$.

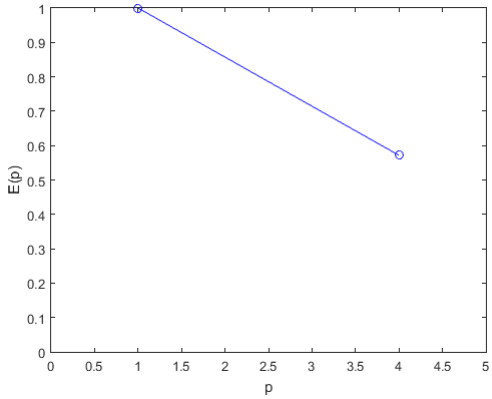
t \ p	1	4	9
1	1.2149	4.8044	8.0498
4	2.7783	10.9277	14.4631

Anche in questo caso si puó notare come i valori siano piuttosto coerenti con quanto ci si aspetta ma nonostante ció, per la maggior parte delle combinazioni tra p e t, i Gflop/s non superano i valori registrati per l'esempio precedente. Si ricorda che il nono processore é solo simulato.

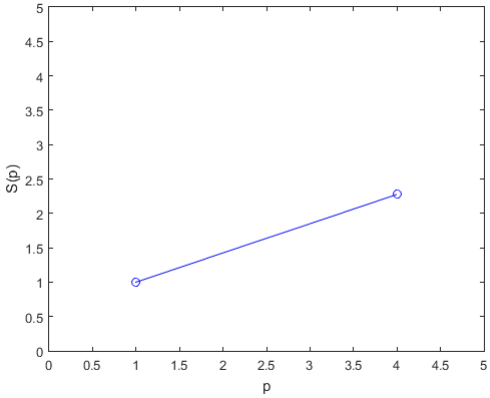
Mostriamo invece lo speed-up e l'efficienza al variare di t con p fissato:

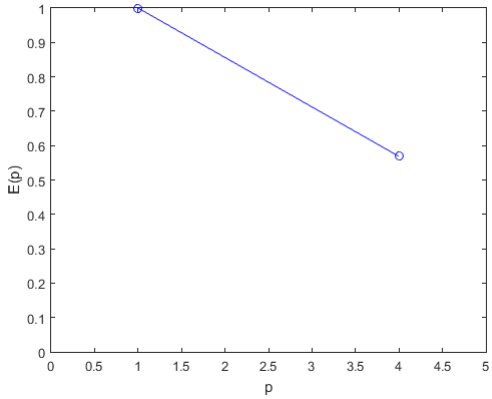
$p = 1$



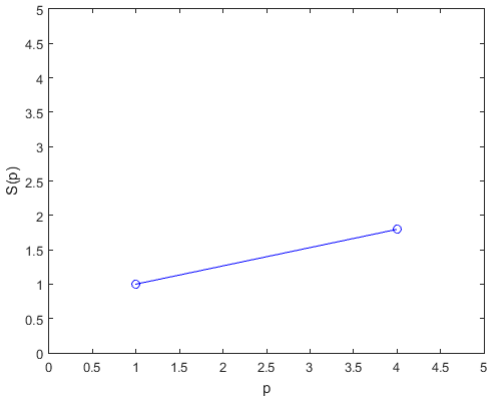


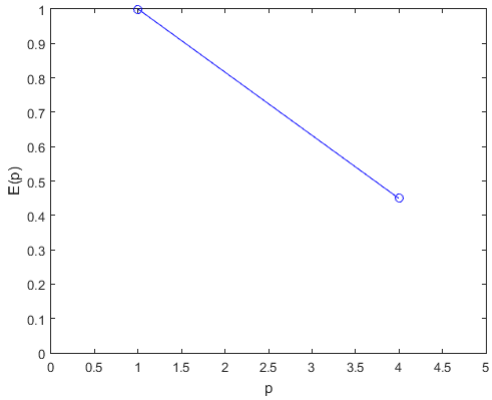
$p = 4$





$p = 9$

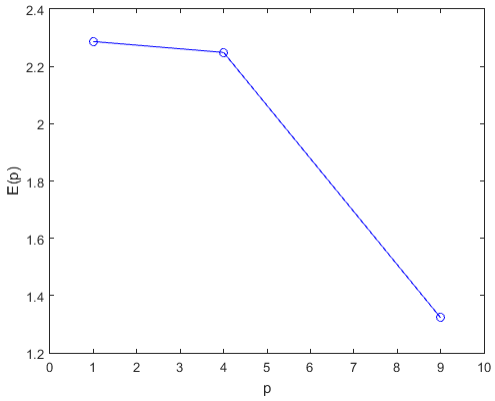




Osserviamo che, se si considera come efficienza la quantitá

$$E_p = \frac{T_{1,1}}{pT_{p,4}}$$

si ottengono valori migliori di quelli ideali.



Infatti il carico di lavoro non é suddiviso realmente tra p processori ma soltanto in parte tra p processori e 4 thread. Se il lavoro totale fosse inizialmente suddiviso tra i pt elementi di computazione in gioco, dovrebbe essere

$$T_{1,1} = ptT_{p,t}$$

Per questo proviamo a considerare le quantitá

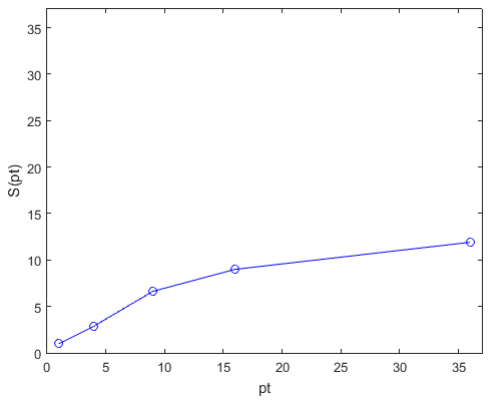
$$S_{p,t} = \frac{T_{1,1}}{T_{p,t}}$$

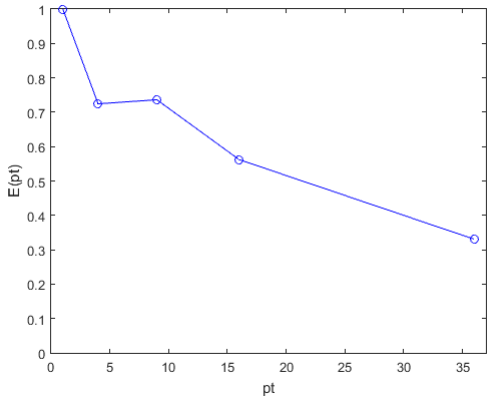
ed

$$E_{p,t} = \frac{S_{p,t}}{pt}$$

Anche qui nel caso ideale, si ha $S_{p,t} = pt$ e $E_{p,t} = 1$.
In realtá nel nostro algoritmo la suddivisione in thread viene effettuata solo per il calcolo matriciale vero e proprio, mentre le comunicazioni, che rappresentano una grossa parte del tempo impiegato, é ovviamente svolta solo dai p processori. Non ci aspettiamo dunque valori ottimali delle quantitá definite.

$p \times t$	1	4	9	16	36
$E_{p,t}$	1.0000	0.7245	0.7362	0.5622	0.3307





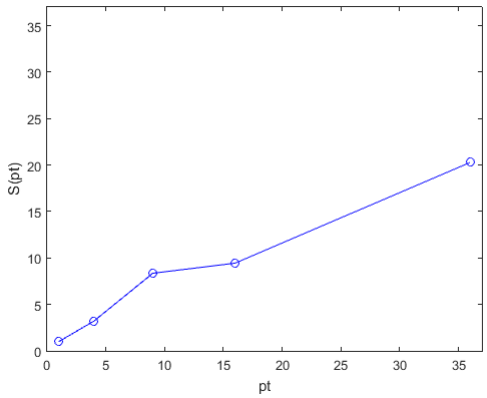
Come ci si aspettava, i valori ottenuti non sono vicini a quelli ideali, in quanto le comunicazioni influiscono negativamente su tali valori. Per i valori di speed-up ed efficienza nel punto $pt = 4$ si é scelto di calcolare una media aritmetica tra $t_{4,1}$ e $t_{1,4}$.
Proviamo a considerare il tempo di esecuzione relativo esclusivamente alla funzione 'matikj' eseguita da t thread per ogni processore. I tempi che si ottengono sono riportati nella tabella seguente:

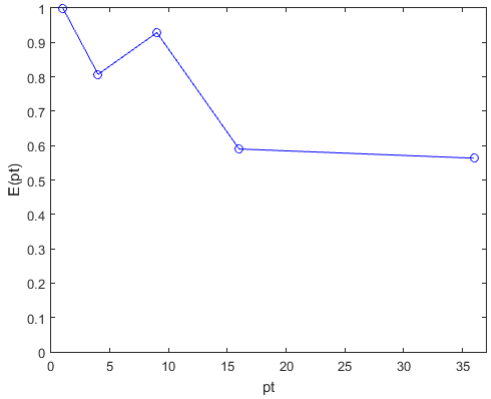
t \ p	1	4	9
1	1655.154062	410.856436	198.064226
4	614.249942	175.314453	81.603689

I valori dell'efficienza relativi sono i seguenti:

$p \times t$	1	4	9	16	36
$E_{p,t}$	1.0000	0.8073	0.9285	0.5901	0.5634

I grafici che si ottengono con questi valori e con quelli dello speed-up sono riportati di seguito.





4.4 Speed-up ed efficienza scalati

Si consideri il tempo necessario a risolvere un problema di dimensione N con un processore, $T(N,1)$, e quello per risolvere un problema di dimensione pN con p processori. Idealmente questi due tempi coincidono, quindi si definisce efficienza scalata la quantitá

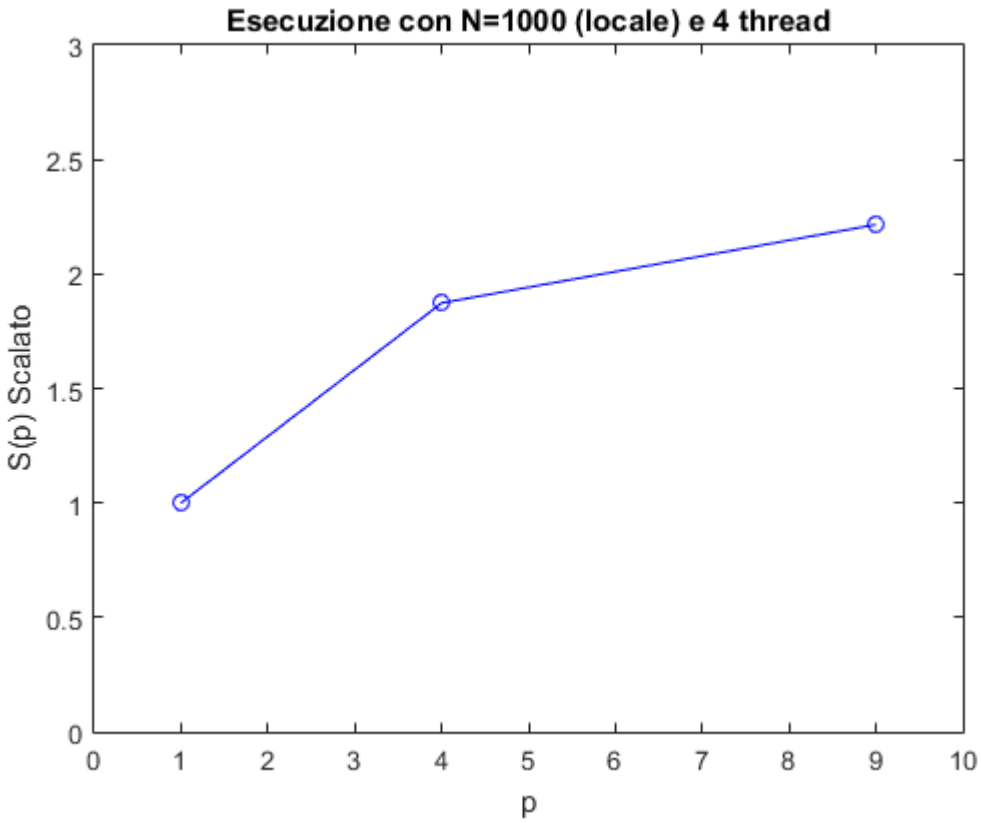
$$ES(p) = \frac{T(N,1)}{T(pN,p)}$$

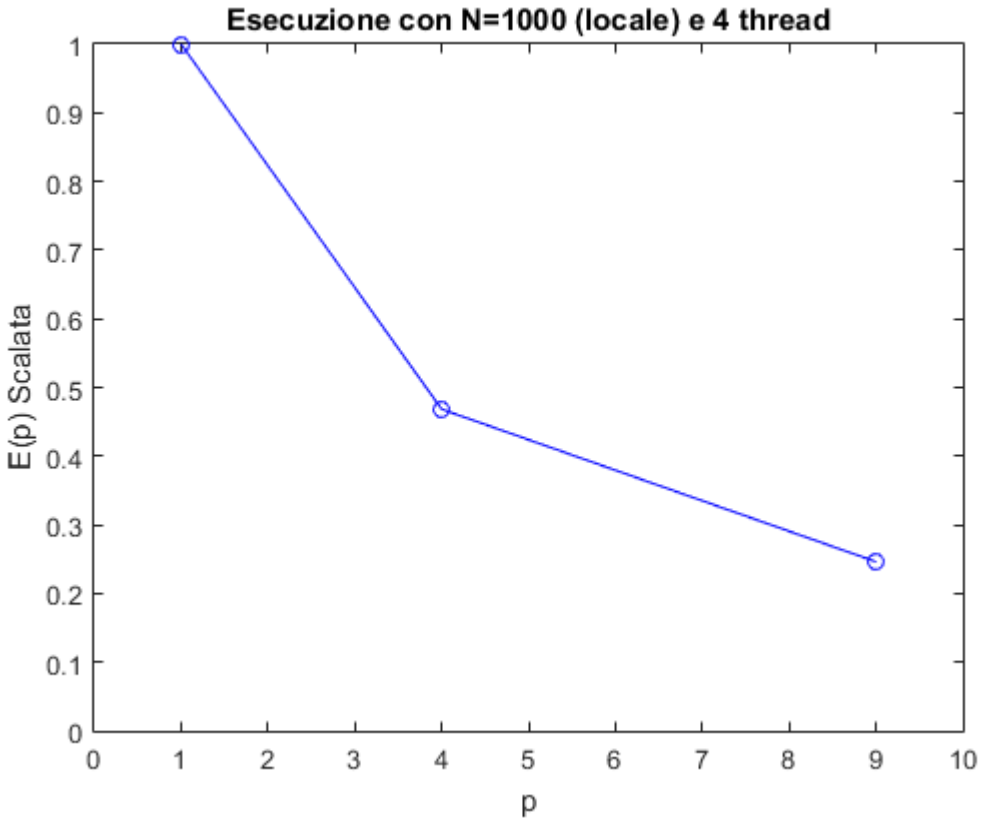
Si definisce invece speed-up scalato la quantitá

$$SS(p) = p\frac{T(N,1)}{T(pN,p)}$$

Nel caso ideale lo speed-up é uguale a p.
Nella tabella che segue si riportano i valori calcolati relativi alle esecuzioni $T(1000,1)$, $T(4000,4)$ e $T(9000,9)$.

p	T (s)	SS(p)	ES(p)
1	0.402365	1	1
4	0.859102	1.8734	0.4684
9	1.634238	2.2159	0.2462



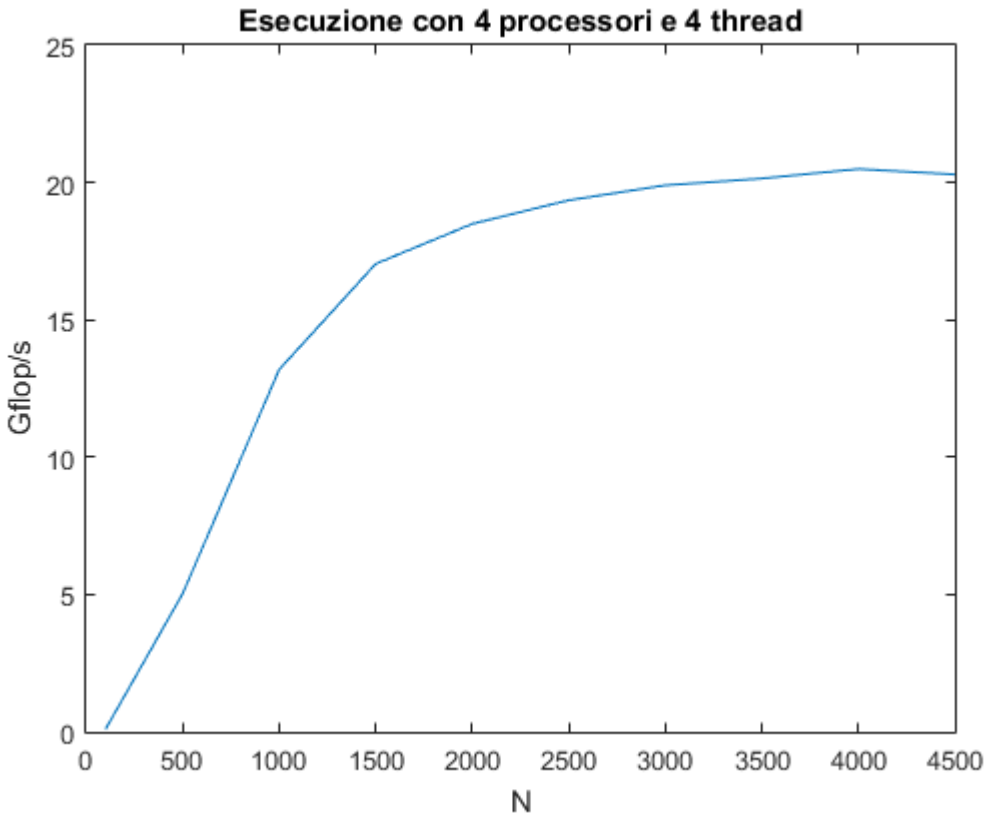


I valori non sono come ci si aspetta in quanto per poter scalare il problema fino a 9 processori é stato utilizzato un valore di N non sufficientemente grande.

4.5 Studio del numero di operazioni al secondo

Nella tabella che segue é riportato il numero di operazioni al secondo espresse in Gflop/s al variare di N.

N	Gflop/s
100	0.0897
500	5.0297
1000	13.1900
1500	17.0366
2000	18.4931
2500	19.3523
3000	19.8961
3500	20.1426
4000	20.4902
4500	20.2906



Si vede che il numero di Gflop/s si stabilizza intorno a 20.

5 Codice

```
/*
PROGETTO 1
ALGORTIMO DI CANNON SFRUTTANDO UNA FUNZIONE MATRICE PER MATRICE MULTITHREAD
*/


#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <pthread.h>
#include "mpi.h"
#include "c_timer.h"



void *thread(void *);
void randomMat(int ,int , float (*)[]);
void stampaMat (int ,int , float (*)[] );
void cannon(MPI_Comm, int , int, int, int , float (*)[],float (*)[],float (*)[], int );
void Matikj(int , int, int, int ,float (*)[], float (*)[], float (*)[] );
void matmatthread (int , int, int, int , int , int , int , float (*)[], float (*)[], float (*)[]);
int mod(int , int);

/*
Questa struttura dati contiene tutti i dati che devono essere trasmessi a ciascun thread
*/

typedef struct  {
 int nrow;
 int ncol;
 int idThread;
 int LDA;
 int LDB;
 int LDC;
 int nt;
 int dim;
 float *A;
 float *B;
 float *C;

}infostruct;

int main(int argc, char* argv[]) {

if(argc<2) {
        printf("ERRORE: e' necessario specificare due parametri di input. Esempio (...) progetto dimMatrice NThreads (...) .\n");
        return 0;
}

//INIZIALIZZAZIONE AMBIENTE MPI
int rank,nproc;
MPI_Init(&argc, &argv);

//DA QUI IN POI L'ALGORITMO E' SVOLTO IN PARALLELO
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Comm_size(MPI_COMM_WORLD,&nproc);
double tiloc,tfloc,tfglob;
```

```
//DA QUI IN POI L'ALGORITMO E' SVOLTO IN PARALLELO
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Comm_size(MPI_COMM_WORLD,&nproc);
double tiloc,tfloc,tfglob;

//LEADING DIMENSION DELLE TRE MATRICI
int LDA=10000;
int LDB=10000;
int LDC=10000;
float *A;
float *B;
float *C;
int N=atoi(argv[1]); //DIMENSIONI DELLE MATRICI LOCALI(NxN).
int nt=atoi(argv[2]); //Numero di thread
A=(float*)malloc(sizeof(float)*(N*LDA));
B=(float*)malloc(sizeof(float)*(N*LDB));
C=(float*)calloc((N*LDC),sizeof(float));

//INIZIALIZZO LE MATRICI LOCALI A E B
srand(rank);
randomMat(LDA,N,(float (*)[])A);
randomMat(LDB,N,(float (*)[])B);

//ESEGUO CANNON
tiloc=get_cur_time();
cannon(MPI_COMM_WORLD,LDA, LDB, LDC, N, (float (*)[])A, (float (*)[])B, (float (*)[])C, nt);
tfloc=get_cur_time()-tiloc;

//CALCOLO DEL TEMPO FINALE D'ESECUZIONE
MPI_Reduce(&tfloc, &tfglob, 1, MPI_DOUBLE, MPI_MAX, 0,MPI_COMM_WORLD);
if(rank==0)
        printf("Tempo finale di esecuzione: %f secondi. \n", tfglob);

MPI_Finalize();
return 0;
}


void cannon (MPI_Comm comm, int LDA, int LDB, int LDC, int N, float A[][LDA],float B[][LDB],float C[][LDC], int nt) {

        //CREO DEI BUFFER
        float *Abuffero;
        float *Bbuffero;
        float *Abufferi;
        float *Bbufferi;

        int rank,nproc;
        MPI_Status status;
        MPI_Request request;
        MPI_Comm_rank(MPI_COMM_WORLD,&rank);
        MPI_Comm_size(MPI_COMM_WORLD,&nproc);
        Abuffero=(float*)malloc(sizeof(float)*(N*LDA));
        Bbuffero=(float*)malloc(sizeof(float)*(N*LDB));
        Abufferi=(float*)malloc(sizeof(float)*(N*LDA));
        Bbufferi=(float*)malloc(sizeof(float)*(N*LDB));

       //COPIA DELLE MATRICI NEI BUFFER
       int i=0;
       int j=0;
       for(i=0; i<N; i++) {
               for(j=0; j<N; j++) {
                       Abuffero[i*LDA+j]= A[i][j];
                       Bbuffero[i*LDA+j]= B[i][j];
               }
       }

       //CALCOLO DIMENSIONI DELLE SOTTOMATRICI LOCALI PER CIASCUN THREAD
       int nrow=sqrt(nt);
       int ncol=sqrt(nt);
       int size= sqrt(nproc); //CALCOLO LA DIMENSIONE DELLA GRIGLIA DEI PROCESSORI. NPROC E' IL NUMERO DI PROCESSORI.
       int myrow = rank / size ;
       int mycol = rank % size ;

       //INCLINAZIONE INIZIALE  DI A. GLI ELEMENTI POSTI SULLA I-ESIMA RIGA SHIFTANO DI I POSIZIONI A SINISTRA

       //CALCOLO IL RANK DEL PROCESSORE DESTINATARIO DEI DATI
       int dest = rank - myrow;
       if (dest < myrow*size)
               dest=dest + size;

       //CALCOLO IL RANK DEL PROCESSORE CHE INVIERA' I DATI A QUESTO PROCESSORE
       int mitt = rank + myrow;
       if (mitt >= (myrow+1)*size)
               mitt=mitt-size;

       //INVIO E RICEVO I DATI
       MPI_Isend(Abuffero,N*N,MPI_FLOAT,dest,dest,comm, &request);
       MPI_Irecv(Abufferi,N*N,MPI_FLOAT,mitt,rank,comm, &request);
       MPI_Wait(&request,&status);

       //INCLINAZIONE INIZIALE  DI B. GLI ELEMENTI POSTI SULLA I-ESIMA COLONNA SHIFTANO DI I POSIZIONI IN ALTO

       //CALCOLO IL RANK DEL PROCESSORE DESTINATARIO DEI DATI
       dest = mod(rank-size*mycol,nproc);

       //CALCOLO IL RANK DEL PROCESSORE DESTINATARIO DEI DATI
       mitt = mod(rank+size*mycol,nproc);

       //INVIO E RICEVO I DATI
       MPI_Isend(Bbuffero,N*N,MPI_FLOAT,dest,dest+80,comm, &request);
       MPI_Irecv(Bbufferi,N*N,MPI_FLOAT,mitt,rank+80,comm, &request);
       MPI_Wait(&request,&status);
```

```
        //INIZIO DEL CICLO DI CANNON
        for(i=0; i<size; i++) {

                //LA FUNZIONE MATMATTHREAD EFFETTUA IL CALCOLO MATRICE*MATRICE SFRUTTANDO PIU' THREAD.
                matmatthread(N, N ,N , LDC, nt, nrow, ncol, (float (*)[])Abufferi, (float (*)[])Bbufferi, (float (*)[])C);

                //COPIO IL BUFFER DI INPUT NEL BUFFER DI OUTPUT. IN QUESTO MODO SONO PRONTO A INVIARE I PROSSIMI DATI
                Abuffero=&Abufferi[0];
                Bbuffero=&Bbufferi[0];

                //ALL'ULTIMO STEP E' INUTILE EFFETTUARE GLI SHIFT
                if(i < size -1 ) {

                //SHIFT DI TUTTI GLI ELEMENTI DI A DI UNO A SINISTRA.
                //CALCOLO, COME SEMPRE, DESTINATARIO E MITTENTE
                if(mycol == 0 )
                     dest=rank+(size-1);
                else
                     dest=rank - 1;
                if(mycol == (size-1))
                     mitt = (rank-size)+1;
                else
                     mitt= rank + 1;

                //INVIO E RICEVO I DATI
                MPI_Isend(Abuffero,N*N,MPI_FLOAT,dest,dest+160,comm, &request);
                MPI_Irecv(Abufferi,N*N,MPI_FLOAT,mitt,rank+160,comm, &request);
                MPI_Wait(&request,&status);

                //SHIFT DI TUTTI GLI ELEMENTI DI B DI UNO SOPRA.
                //CALCOLO, COME SEMPRE, DESTINATARIO E MITTENTE
                if(myrow == 0 )
                     dest=rank+(size*(size-1));
                else
                     dest=rank - size;
                if(myrow == (size-1))
                     mitt = rank-(size*(size-1));
                else
                     mitt= rank + size;

                //INVIO E RICEVO I DATI
                MPI_Isend(Bbuffero,N*N,MPI_FLOAT,dest,dest+240,comm, &request);
                MPI_Irecv(Bbufferi,N*N,MPI_FLOAT,mitt,rank+240,comm, &request);
                MPI_Wait(&request,&status);

        }
 }

}

  //CREA UNA MATRICE FLOAT RANDOM
  void randomMat(int LD,int N, float a[][LD]) {
          int i=0;
          int j=0;
          for(i=0; i<N; i++) {
                  for(j=0; j<N; j++) {
                          a[i][j]=(100.0*rand()/RAND_MAX);
                  }
          }
  }


  //EFFETTUA UN CALCOLO MATRICE*MATRICE SFRUTTANDO PIU' THREAD.
  void matmatthread(int N, int LDA, int LDB, int LDC, int nt, int nrow, int ncol, float A[][LDA], float B[][LDB], float C[][LDC]) {

          //ALLOCO LE STRUTTURE NECESSARIE ALLA CREAZIONE DEI THREAD
          infostruct *info;
          pthread_t *thread_id;
          thread_id = (pthread_t *)calloc(nt,sizeof(pthread_t));

          //ALLOCO E POPOLO LA STRUTTURA DATI INFO PER CIASCUN THREAD
          int i=0;
          for(i=0; i < nt; i++){
                  info = (infostruct *) malloc (sizeof(infostruct));
                  info->idThread = i;
                  info->nrow= nrow;
                  info->ncol = ncol;
                  info->dim = N;
                  info->LDA=LDA;
                  info->LDB=LDB;
                  info->LDC=LDC;
                  info->nt=nt;
                  info->A=*A;
                  info->B=*B;
                  info->C=*C;

                  //LANCIO IL THREAD
                  pthread_create(&thread_id[i], NULL, &thread, (void*)info);
          }

          //ATTENDO CHE I THREAD CONCLUDANO
          i=0;
          for(i=0; i < nt; i++){;
                  pthread_join(thread_id[i], NULL);
          }

  }
```

```
//LA FUNZIONE CHE ESEGUE CIASCUN THREAD
void *thread (void *arg){

        //ALLOCO LA STRUTTURA DATI PER RECUPERARE TUTTI I PARAMETRI E LI INSERISCO ALL'INTERNO DI VARIABILI
        infostruct *info;
        info=(infostruct *)malloc (sizeof(infostruct));
        info = (infostruct *) arg;
        float *A;
        float *B;
        float *C;
        int LDA;
        int LDB;
        int LDC;
        int Nb;
        int id;
        int nt;
        int N = info->dim;
        nt=info->nt;
        A=info->A;
        B=info->B;
        C=info->C;
        LDA=info->LDA;
        LDB=info->LDB;
        LDC=info->LDC;
        Nb=info->nrow;
        id=info->idThread;
        int irow=id/Nb;
        int icol=id%Nb;


        //ESEGUO RADICE QUADRATA DEL NUMERO DI THREAD VOLTE L'ALGORITMO BASE DI MATRICE PER MATRICE
        int j=0;
        for(j=0; j<sqrt(nt); j++) {

                //LANCIO L'ALGORTIMO MATRICE PER MATRICE PASSANDO CORRETTAMENTE I PUNTI DI PARTENZA DI CIASCUNA MATRICE.  I PUNTI VENGONO TRASMESSI ATTRAVERSO LO SPOSTAMEN
                Matikj(LDA,LDB,LDC,N/Nb,(float (*)[])(A+(N/Nb)*(irow*LDA+j)),(float (*)[])(B+(N/Nb)*(LDB*j+icol)),(float (*)[])(C+(N/Nb)*(irow*LDC+icol)));

        }
}

//ALGORTIMO BASE MATRICE PER MATRICE CON LA COMBINAZIONE DI INDICI PIU' EFFICIENTE
void Matikj(int LDA, int LDB, int LDC, int N,float A[][LDA], float B[][LDB], float C[][LDC] ){
        int i=0;
        int j=0;
        int k=0;
        for(i=0;i<N;i++){
                for(k=0;k<N;k++){
                        for(j=0;j<N;j++){
                                C[i][j]+=A[i][k]*B[k][j];
                        }
                }
        }
}

     //FUNZIONE AUSILIARIA MODULO. CALCOLA IL MODULO ANCHE DEI NUMERI NEGATIVI CHE C NON CALCOLA NATIVAMENTE.
     int mod(int a, int m) {
             if(a>=0)
                return a%m;
             else {
                     while (a<0) {
                             a+=m;
                     }
             }
     return a;

     }

     //FUNZIONE PER LA STAMPA DELLA MATRICE
     void stampaMat(int LD, int N, float a[][LD]){
             int i=0;
             int j=0;
             for(i=0; i<N; i++) {
                     for(j=0; j<N; j++) {
                             printf("%f ",a[i][j]);
                     }
                     printf("\n");
             }
     }
```