Università degli Studi di Napoli
"Parthenope"

Corso di Calcolo Parallelo e Distribuito

Progetto Matrice per Matrice
Strategia BMR

Virginia Bellino
Matr. 108/1570

Indice

INDIVIDUAZIONE ED ANALISI DEL PROBLEMA

DEFINIZIONE DEL PROBLEMA

Sviluppare un algoritmo per il calcolo del prodotto matrice per matrice in ambiente di calcolo parallelo su architettura MIMD a memoria distribuita, che utilizzi la libreria MPI.

Le matrici $A \in \Re^{nxm}$ e $B \in \Re^{mxk}$ devono essere distribuite a p^2 processori disposti secondo una topologia a griglia bidimensionale.

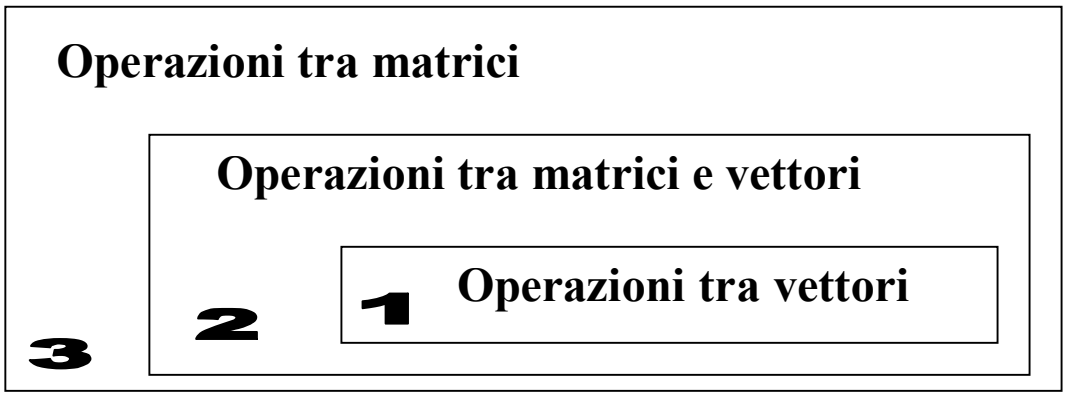
L'algoritmo deve implementare la strategia di comunicazione **BMR** ed essere organizzato in modo da utilizzare un sottoprogramma per la costruzione della griglia bidimensionale dei processi ed un sottoprogramma per il calcolo dei prodotti locali.

DESCRIZIONE DELL'ALGORITMO

- **Caratteristiche generali**

Nella risoluzione numerica di molti problemi della scienza e della tecnica, il nucleo computazionale fondamentale è rappresentato da un problema di algebra lineare, la cui risoluzione risulta combinazione di alcune operazioni, note come operazioni di base dell'algebra lineare.

Le operazioni di base dell'algebra lineare hanno la seguente struttura gerarchica:



Nelle operazioni di tipo matrice-matrice, l'operazione fondamentale è il prodotto, la cui importanza deriva dal fatto che molti algoritmi dell'algebra lineare possono essere descritti sinteticamente in termini di prodotto tra matrici.

Per effettuare il prodotto tra matrici, esistono varie tecniche, basate essenzialmente sulla decomposizione delle matrici di input in blocchi di righe o blocchi di colonne.

La tecnica descritta dalla presente relazione viene denominata…….

Broadcast **M**ultiply **R**olling (**BMR**)

Questa tecnica prevede la decomposizione delle matrici di input in blocchi quadrati, e ciascuno di tali blocchi verrà poi assegnato ai processori disposti lungo una griglia bidimensionale periodica.

Per semplicità si suppone dunque che:

1. il numero di processori concorrenti è del tipo p^2
2. l'ordine delle due matrici è proporzionale al numero di processori

- **Codice**

Il software realizzato si compone di 2 files, in modo da agevolare la leggibilità del codice.

Il primo file è la funzione main, che in primo luogo, prevede la dichiarazione delle variabili utilizzate e l'inizializzazione dell'ambiente di calcolo MPI utilizzando le apposite routines.

Successivamente, vengono controllate le condizioni di applicabilità della BMR.

La prima condizione è che il numero di processori deve essere tale da generare una griglia quadrata. Se ciò non si verifica il programma termina

```
//Controllo sul numero dei processori per garantire l'applicabilità della BMR
if(sqrt(nproc)*sqrt(nproc)!=nproc)
{

        if(menum==0)
        {
                printf("ERRORE:Impossibile applicare la strategia...\n");
                printf("Il numero di processori deve essere tale da generare una griglia quadrata.\n");

        }
```

```
        MPI_Finalize();
        return 0;

}
```

Dopo tale controllo, se l'identificativo corrisponde al processore P0, si effettua una verifica sulle dimensioni delle matrici inserite.

Tali dimensioni devono essere divisibili per la radice quadrata del numero di processori, altrimenti la BMR non si può applicare.

Vi è poi l'allocazione dinamica delle matrici e l'inserimento degli elementi delle due matrici di input.

Tale inserimento può essere….

```
//inserimento manuale
//printf("Digita elemento A[%d][%d]: ", i, j);
//scanf("%f", A+i*n+j);
```

…oppure….

```
//per valutare la prestazioni si attiva la riga seguente
//che genera random la matrice
*(A+i*n+j)=(float)rand()/((float)RAND_MAX+(float)1);
```

Le due matrici di input vengono stampate solo se le loro dimensioni non superano 100 x 100, e la stessa sorte verrà riservata anche alla matrice risultato.

Il processore P0 esegue poi un broadcast per comunicare a tutti i processori del comunicatore le dimensioni delle matrici e il valore di p (radice quadrata del numero di processori) che verrà utilizzato in seguito.

La chiamata di funzione ….

```
//Viene creata la griglia
crea_griglia(&griglia, &grigliar, &grigliac, menum, p, coordinate);
```

…provvede alla creazione della griglia bidimensionale periodica utilizzando una routine contenuta nel file denominato "**mxm_vb_aus.c**".

Dopo la creazione della griglia, troviamo poi una serie di istruzioni che allocano dinamicamente lo spazio necessario per le matrici di input, per la matrice risultato, e per una serie di blocchi di appoggio che verranno utilizzati nel calcolo.

A questo punto, inizia la fase di spedizione e di calcolo.

Il processore P0 calcola le dimensioni dei sottoblocchi delle due matrici che devono essere spediti e li spedisce a ciascun processore, poi alloca lo spazio per i rispettivi sottoblocchi.

Se invece non ci si trova nel processore P0, allora il processore in questione riceve i due sottoblocchi di A e di B da utilizzare per il calcolo.

Successivamente, iniziano le comunicazioni tra i processori della griglia per scambiarsi i sottoblocchi ed arrivare cosi alla elaborazione del risultato.

L'algoritmo si chiude con l'eventuale stampa della matrice risultato (o di un messaggio alternativo), e la successiva stampa dei paramenti di valutazione.

ANALISI DEL SOFTWARE

INDICAZIONI DI UTILIZZO PER L'UTENTE

- ***Compilazione ed esecuzione***

Per compilare il programma sorgente occorre digitare sul prompt dei comandi la seguente istruzione:

```
$ mpicc progetto_mxm_vb.c
```

Per eseguire il programma in luogo di far digitare all'utente sul prompt dei comandi una istruzione del tipo:

```
$ mpirun –np x a.out
```

o, più precisamente,considerando la struttura utilizzata per la fase di testing, una istruzione del tipo…

```
$ mpirun –np x –machinefile lista a.out
```

si è scelto di creare uno script di shell contenente l'istruzione che consente di mandare in esecuzione il programma. Tale script prende come argomento il numero di processori utilizzato.

Tale numero deve essere un *quadrato perfetto*, altrimenti la strategia prescelta risulta non applicabile.

L'istruzione di esecuzione diventa pertanto:

```
$ go.sh x
```

dove x rappresenta il numero di processori utilizzato che, si ribadisce, deve essere un quadrato perfetto.

Riepilogando:

mpicc è l'istruzione che consente di compilare il programma sorgente del tipo **nomesorgente.c**

mpirun -np è l'istruzione che consente di eseguire il programma

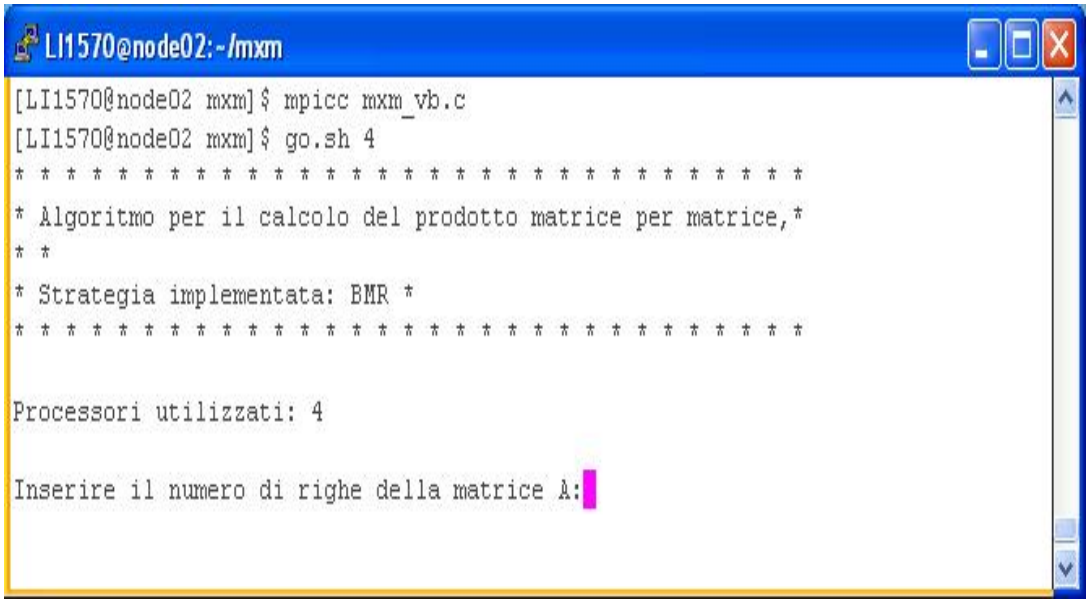
a.out indica generalmente il nome dell'eseguibile

x è il numero di processori che si desidera impiegare nel calcolo. Deve essere un quadrato perfetto

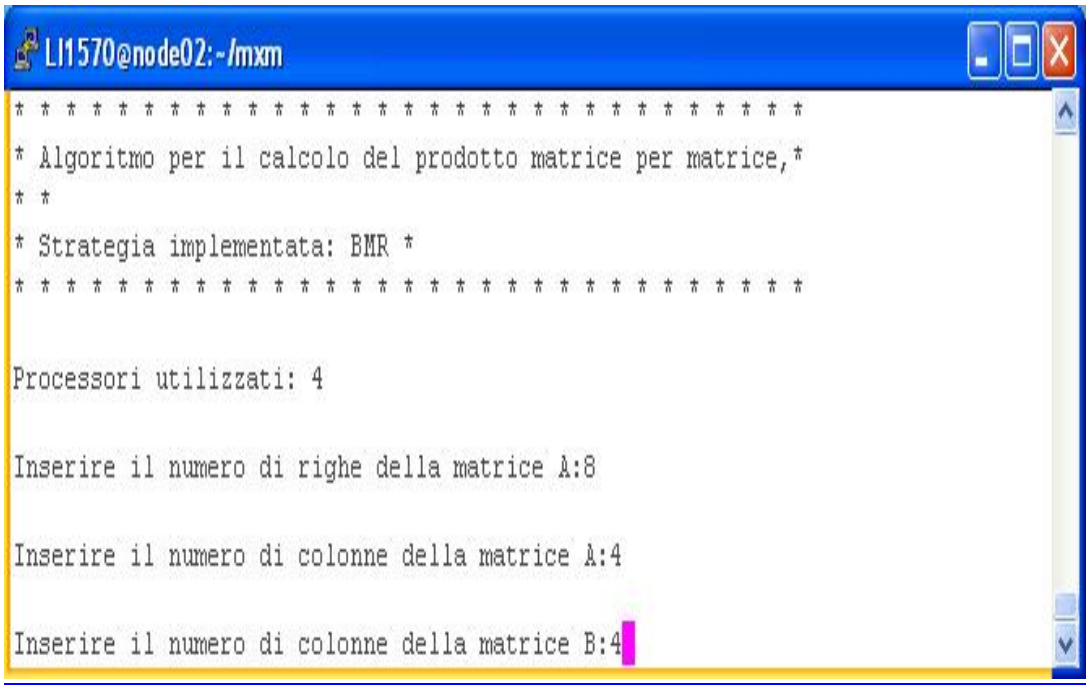
- ***Esempi d'uso***

Gli esempi che seguono mostrano i passi che occorre compiere per compilare ed eseguire il programma e che tipo di output l'utente riceverà a video.

Esempio 1:

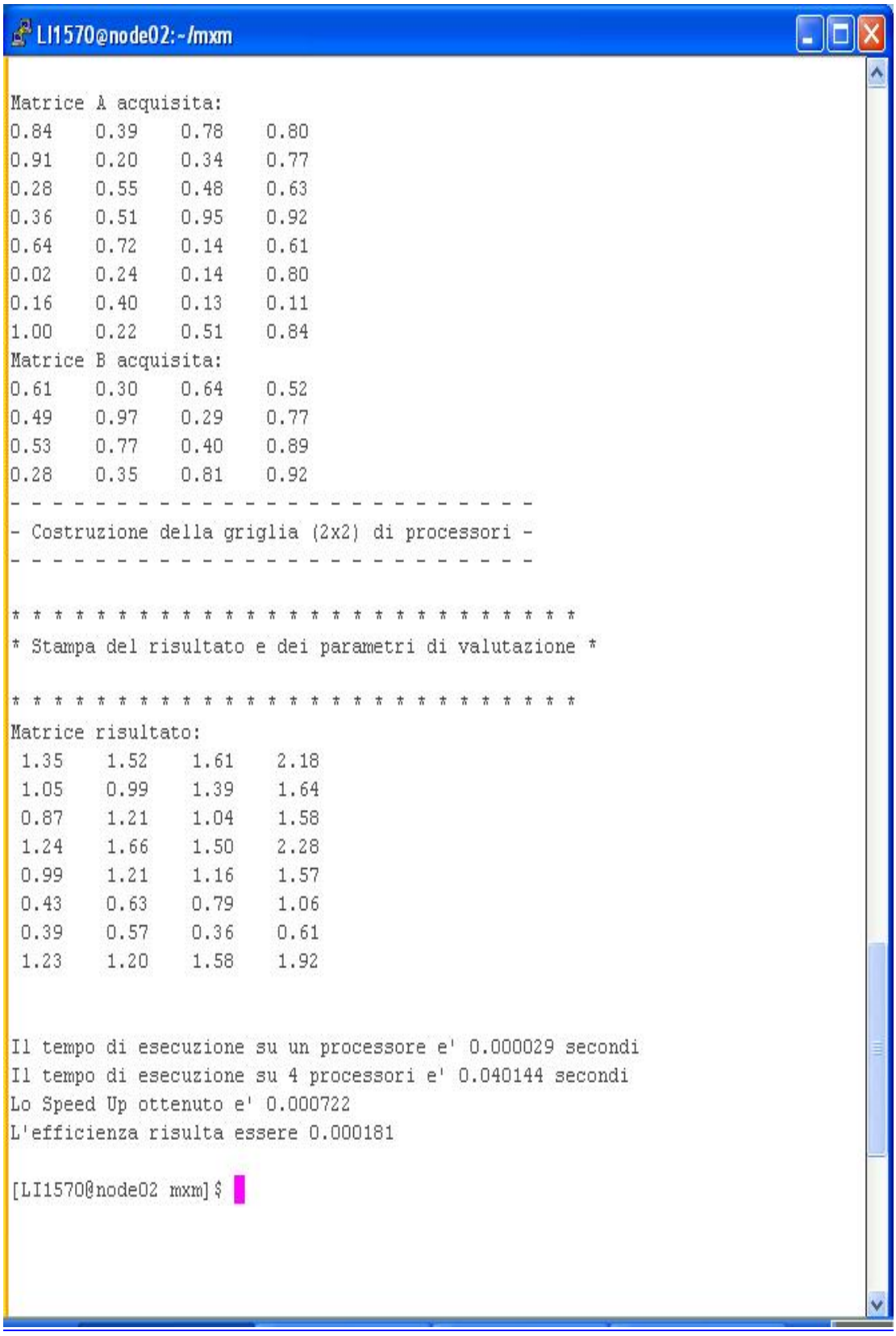


L'esempio mostra cosa appare all'utente dopo aver compilato e mandato in esecuzione il programma.

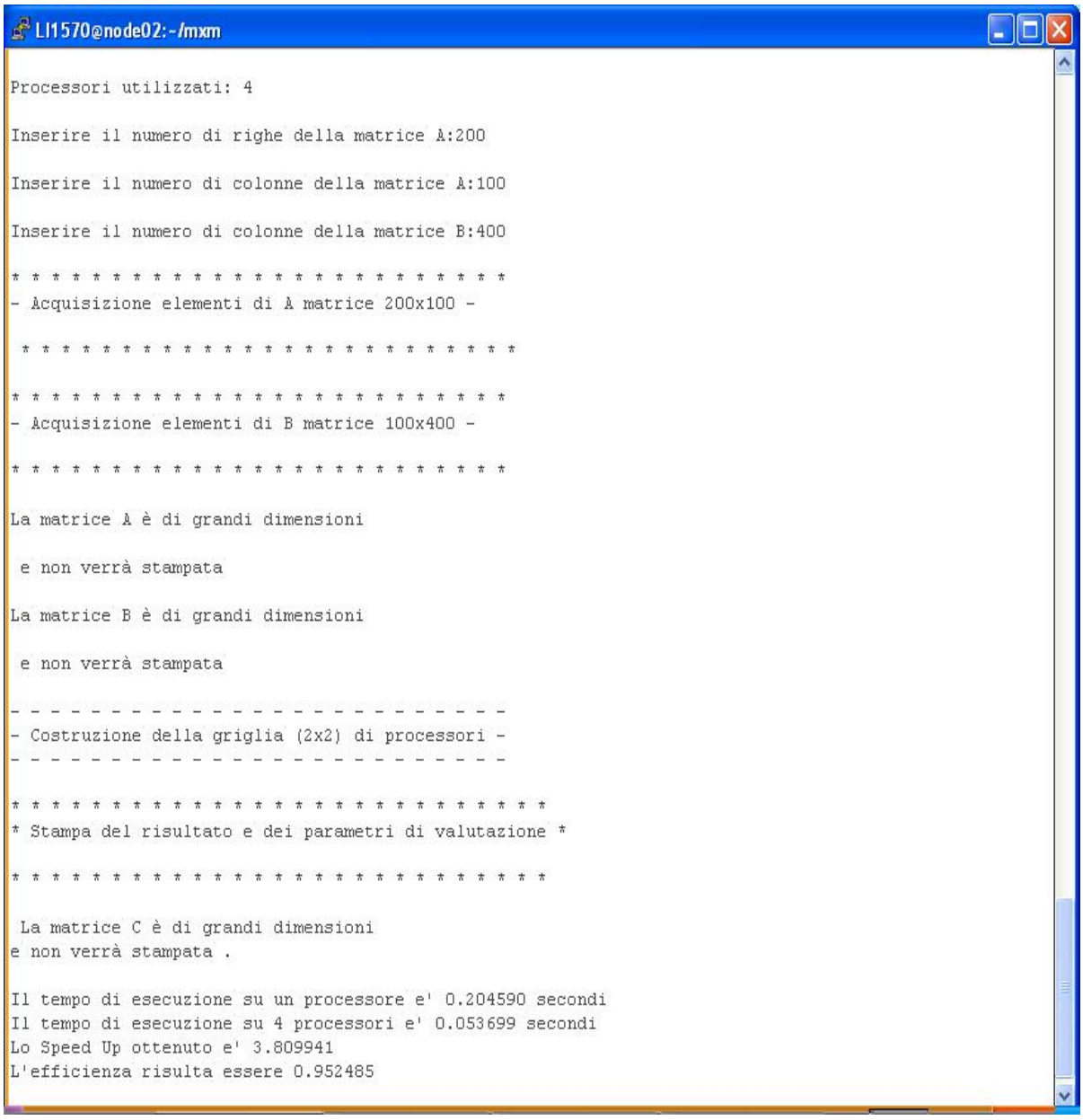


Viene chiesto innanzitutto di inserire le dimensioni delle matrici.

Successivamente, come si può notare osservando la schermata che segue, se le dimensioni delle matrici di input e della matrice risultato non superano 100 x 100, appare a video la stampa delle medesime seguita dalla stampa dei parametri di valutazione


LI1570@node02:~/mxm
Matrice A acquisita:
0.84 0.39 0.78 0.80
0.91 0.20 0.34 0.77
0.28 0.55 0.48 0.63
0.36 0.51 0.95 0.92
0.64 0.72 0.14 0.61
0.02 0.24 0.14 0.80
0.16 0.40 0.13 0.11
1.00 0.22 0.51 0.84
Matrice B acquisita:
0.61 0.30 0.64 0.52
0.49 0.97 0.29 0.77
0.53 0.77 0.40 0.89
0.28 0.35 0.81 0.92
- -
- Costruzione della griglia (2x2) di processori -
- -
* *
* Stampa del risultato e dei parametri di valutazione *
* *
Matrice risultato:
1.35 1.52 1.61 2.18
1.05 0.99 1.39 1.64
0.87 1.21 1.04 1.58
1.24 1.66 1.50 2.28
0.99 1.21 1.16 1.57
0.43 0.63 0.79 1.06
0.39 0.57 0.36 0.61
1.23 1.20 1.58 1.92
Il tempo di esecuzione su un processore e' 0.000029 secondi
Il tempo di esecuzione su 4 processori e' 0.040144 secondi
Lo Speed Up ottenuto e' 0.000722
L'efficienza risulta essere 0.000181
[LI1570@node02 mxm]$

Esempio 2:

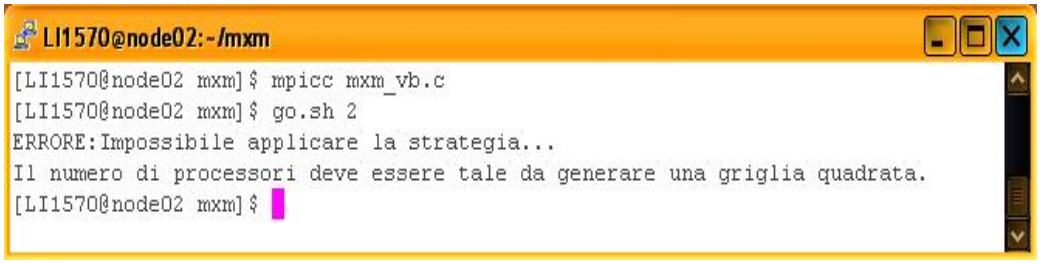


Esempio analogo al caso precedente, ma le matrici sono di grandi dimensioni e non vengono stampate

- ***Indicatori di errore***

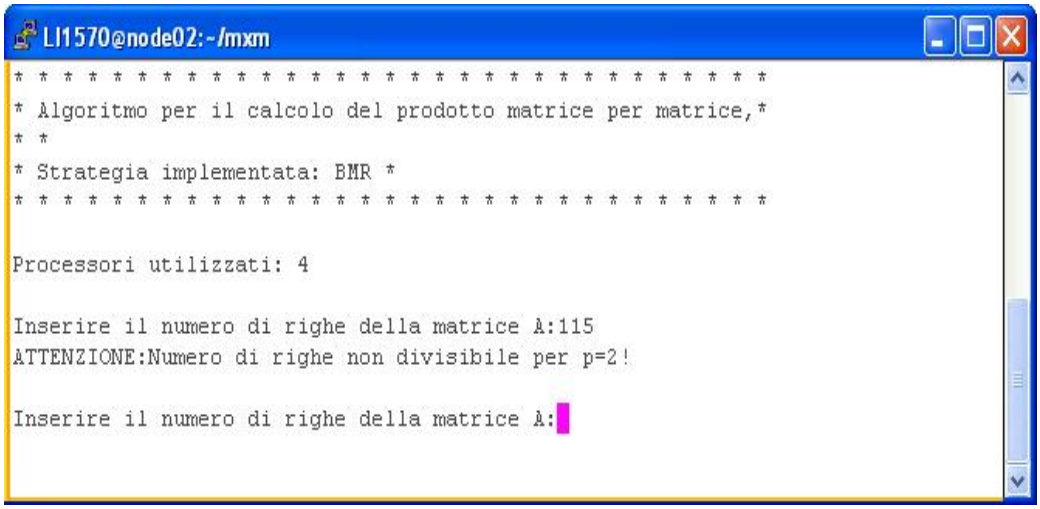
Le schermate che seguono mostrano le situazioni di errore in cui si può incorrere.

Situazione 1



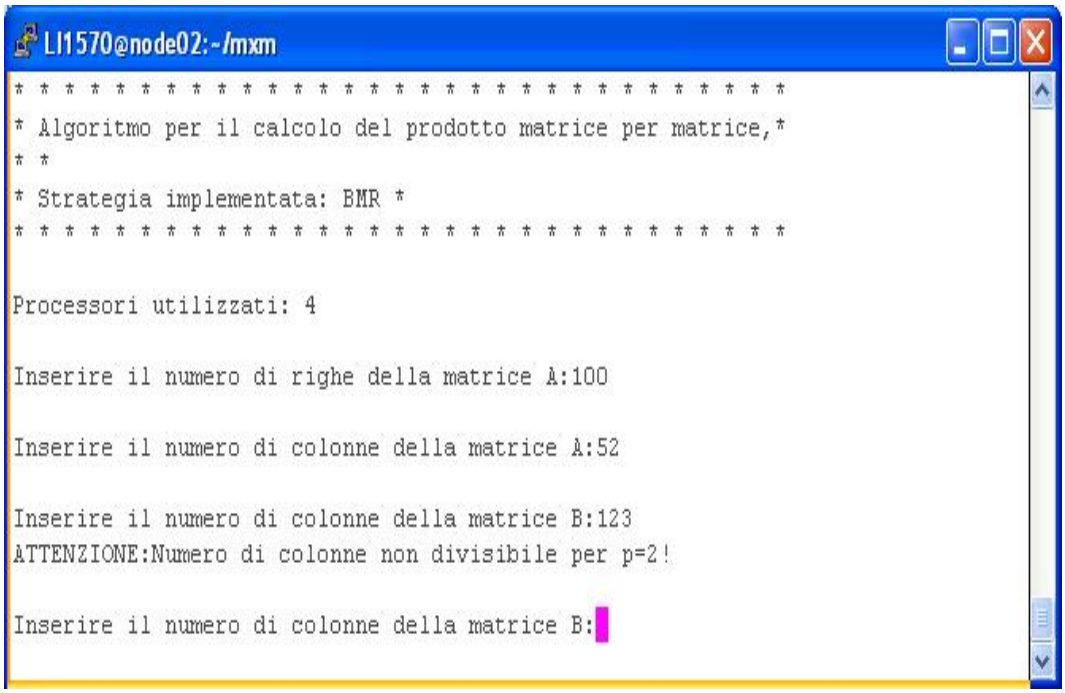
Questo errore si verifica quando si inserisce come argomento dello script di esecuzione **go.sh** un numero di processori che non è un quadrato perfetto. In questo caso infatti, non si può generare una griglia quadrata ne applicare la BMR, e quindi il programma termina.

Situazione 2



In questi caso, l'errore riguarda l'inserimento del numero di righe della prima matrice, che non è divisibile per la radice quadrata del numero di processori (condizione di applicabilità della BMR)

Situazione 3



Analogo alla situazione 2, ma riguarda il numero di colonne della seconda matrice

FUNZIONI AUSILIARIE UTILIZZATE

Il software descritto si compone di 2 files.

Il primo file rappresenta la funzione chiamante, e in esso troviamo l'utilizzo di alcune routine della libreria MPI per il calcolo parallelo.

Tali funzioni compaiono anche nel secondo file, costituito da una libreria utente contenente alcune funzioni usate dal programma principale

Di seguito, per ognuna di esse si fornisce il prototipo utilizzato nel programma e la descrizione dei relativi parametri di input e di output. Per alcune funzioni, poiché nel programma vi è più di una chiamata, si fornisce il prototipo della prima chiamata rilevata, sottintendendo che il ragionamento è analogo anche per gli altri casi.

Funzioni per inizializzare l'ambiente MPI

- **MPI_Init (&argc, &argv);**
  argc e *argv* sono gli argomenti del main

- **MPI_Comm_rank (MPI_COMM_WORLD, &menum);**
  MPI_COMM_WORLD (input): identificativo del comunicatore entro cui avvengono le comunicazioni

  menum (output): identificativo di processore nel gruppo del comunicatore specificato

- **MPI_Comm_size(MPI_COMM_WORLD, &nproc);**
  MPI_COMM_WORLD (input): nome del comunicatore entro cui avvengono le comunicazioni

  nproc (output): numero di processori nel gruppo del comunicatore specificato

Funzioni di comunicazione collettiva in ambiente MPI

- **MPI_Bcast(&m, 1, MPI_INT, 0, MPI_COMM_WORLD);**

comunicazione di un messaggio a tutti i processori appartenenti al comunicatore specificato.

I parametri sono:

m: indirizzo dei dati da spedire
1: numero dei dati da spedire
MPI_INT: tipo dei dati da spedire
0 : identificativo del processore che spedisce a tutti
MPI_COMM_WORLD : identificativo del comunicatore entro cui avvengono le comunicazioni

Funzioni di comunicazione bloccante in ambiente MPI

- **MPI_Send(A+offsetC*n+offsetR,n/p, MPI_FLOAT, i, tag, MPI_COMM_WORLD);**

  spedizione di dati

  I parametri sono:
  *A+offsetC*n+offsetR (input):* indirizzo del dato da spedire
  n/p (input): numero dei dati da spedire
  MPI_FLOAT (input): tipo del dato inviato
  i (input): identificativo del processore destinatario
  tag (input): identificativo del messaggio inviato
  MPI_COMM_WORLD (input): comunicatore usato per l'invio del messaggio

- **MPI_Recv (subA+n/p*j, n/p, MPI_FLOAT, 0, tag, MPI_COMM_WORLD, &info);**

  ricezione di dati

  I parametri sono:
  *subA+n/p*j*: indirizzo del dato su cui ricevere
  n/p : numero dei dati da ricevere
  MPI_FLOAT : tipo dei dati da ricevere
  0: identificativo del processore da cui ricevere
  tag (input): identificativo del messaggio
  MPI_COMM_WORLD (input): comunicatore usato per la ricezione del messaggio
  info: vettore che contiene informazioni sulla ricezione del messaggio

Funzioni di comunicazione non bloccante in ambiente MPI

- **MPI_Isend(subB, n/p*k/p, MPI_FLOAT, destinatarior, tag, grigliac, &rqst);**

  spedizione non bloccante di dati

  I parametri sono:

  *subB:*indirizzo del dato da spedire
  *n/p*k/p:*dimensione del dato da spedire
  *MPI_FLOAT:*tipo del dato da spedire
  *destinatarior:*identificativo del destinatario
  *tag:*identificativo del messaggio
  grigliac:comunicatore entro cuo avvengono le comunicazioni
  rqst: oggetto che crea un nesso tra la trasmissione e la ricezione del messaggio

Funzione di sincronizzazione MPI

- **MPI_Barrier(MPI_COMM_WORLD);**
  La funzione fornisce un meccanismo sincronizzante per tutti i processori del comunicatore MPI_COMM_WORLD

- **MPI_Wtime( )**
  Tale funzione restituisce un tempo in secondi

Funzione di chiusura ambiente MPI

- **MPI_Finalize( );**

La funzione determina la fine di un programma MPI. Dopo di essa non si può più chiamare nessuna altra routine MPI.

Funzione di creazione e gestione della topologia a griglia bidimensionale in ambiente MPI

- **MPI_Cart_create(MPI_COMM_WORLD, dim, ndim, period, reorder, griglia);**

Operazione collettiva che restituisce un nuovo comunicatore in cui i processi sono organizzati in una griglia di dimensione *dim* .

I parametri sono:

MPI_COMM_WORLD: identificativo del comunicatore di input
dim: numero di dimensioni della griglia
ndim: vettore di dimensione dim contenente le lunghezze di ciascuna dimensione
period: vettore di dimensione dim contenente la periodicità di ciascuna dimensione
reorder: permesso di riordinare i menum processori (1=si, 0=no)
griglia: nuovo comunicatore di output associato alla griglia

- **MPI_Comm_rank(*griglia, &menum_griglia);**

Analogo alla chiamata in sede di inizializzazione dell'ambiente MPI. Restituisce l'identificativo del processore nella griglia

griglia (input): identificativo del comunicatore entro cui avvengono le comunicazioni

menum_griglia (output): identificativo di processore nel gruppo del comunicatore specificato

- **MPI_Cart_coords(*griglia, menum_griglia, dim, coordinate);**

ogni processo calcola le proprie due coordinate nel nuovo ambiente, cioè nella griglia.

I parametri sono:

griglia : identificativo del comunicatore entro cui avvengono le comunicazioni

menum : identificativo di processore nel gruppo del comunicatore specificato, cioè nella griglia

dim: numero di dimensioni della griglia

coordinate: vettore di dimensione dim i cui elementi rappresentano le coordinate del processore all'interno della griglia

- **MPI_Cart_sub(*griglia, vc, grigliar);**
- **MPI_Cart_sub(*griglia, vc, grigliac);**

crea sotto-griglie di dimensione inferiore a quella originale. Ciascuna sotto-griglia viene identificata da un comunicatore differente.
Partiziona il communicator nel sottogruppo grigliar (o grigliac).

I parametri sono:

griglia: comunicatore di griglia
vc: dimensioni della sottogriglia
grigliar (o *grigliac*): comunicatore che include la sottogriglia contenente i processi desiderati

- **MPI_Cart_rank(grigliac, destRoll, destinatarior);**

determina il rango di un processore a partire dalle coordinate cartesiane del medesimo in una topologia a griglia bidimensionale.

In pratica, date le coordinate cartesiane del progetto ritorna il rank associato.

I parametri sono:

grigliac: comunicatore con topologia cartesiana

destRoll: coordinate del processore del quale si vuole conoscere il rango

destinatarior: rango del processore con coordinate *destRoll*

Il secondo file è una libreria utente contenente alcune function utilizzate per il calcolo, e cioè:

/*FUNZIONE crea_griglia*/

- **void crea_griglia(MPI_Comm *griglia, MPI_Comm *grigliar, MPI_Comm *grigliac, int menum, int lato, int*coordinate)**

crea una griglia bidimensionale periodica utile a combinare i risultati parziali ottenuti da tutti i processori

I parametri sono:

*griglia:*comunicatore che identifica la griglia
grigliar: comunicatore che identifica la sotto-griglia delle righe
grigliac: comunicatore che identifica la sotto-griglia delle colonne
menum: identificativo di processore
*lato:*dimensioni della griglia. In questo caso, poiché la dimensione è unica, la griglia risulterà quadrata
coordinate: vettore di dimensione dim i cui elementi rappresentano le coordinate del processore all'interno della griglia

/*Function per il CALCOLO DEL PRODOTTO MATRICE VETTORE */

- **double mat_mat_righe(float *A, , int m, int n, float *B, int k, float *C)**

esegue il calcolo del prodotto matrice per matrice

*A***:** matrice di input

B: matrice di input

m,n,k : dimensioni delle matrici

C: matrice risultato

ANALISI DEI TEMPI

Introduzione

Osserviamo ora il nostro algoritmo analizzando in dettaglio alcune caratteristiche atte a valutare le prestazioni di un software parallelo.

Esse consentiranno all'utente di capire in quale situazione è più opportuno utilizzare l'algoritmo e quando invece il suo utilizzo non reca alcun palese vantaggio.

Tali caratteristiche sono:

- *Tempo di esecuzione utilizzando un numero p>1 di processori.*
  Generalmente indicheremo tale parametro con il simbolo **T(p)**

- ***Speed – up***
  riduzione del tempo di esecuzione rispetto all'utilizzo di un solo processore, utilizzando invece p processori.
  In simboli ……

  $$S(p) = T(1) / T(p)$$

  Il valore dello speed – up ideale dovrebbe essere pari al numero p dei processori, perciò l'algoritmo parallelo risulta migliore quanto più S(p) è prossimo a p.

- **Efficienza**
  Calcolare solo lo speed-up spesso non basta per effettuare una valutazione corretta, poiché occorre *"rapportare lo speed-up al numero di processori "*, e questo può essere effettuato valutando l'efficienza.
  Siano dunque p il numero di processori ed S(p) lo speed - up ad esso relativi.
  Si definisce efficienza il parametro……

  $$E(p) = S(p) / p$$

  Essa fornisce *un'indicazione di quanto sia stato usato il parallelismo nel calcolatore.*
  Idealmente, dovremmo avere che:
  $$E(p) = 1$$
  e quindi l'algoritmo parallelo risulta migliore quanto più E(p) è vicina ad 1.

Valutazione dei tempi, dello speed-up e dell'efficienza

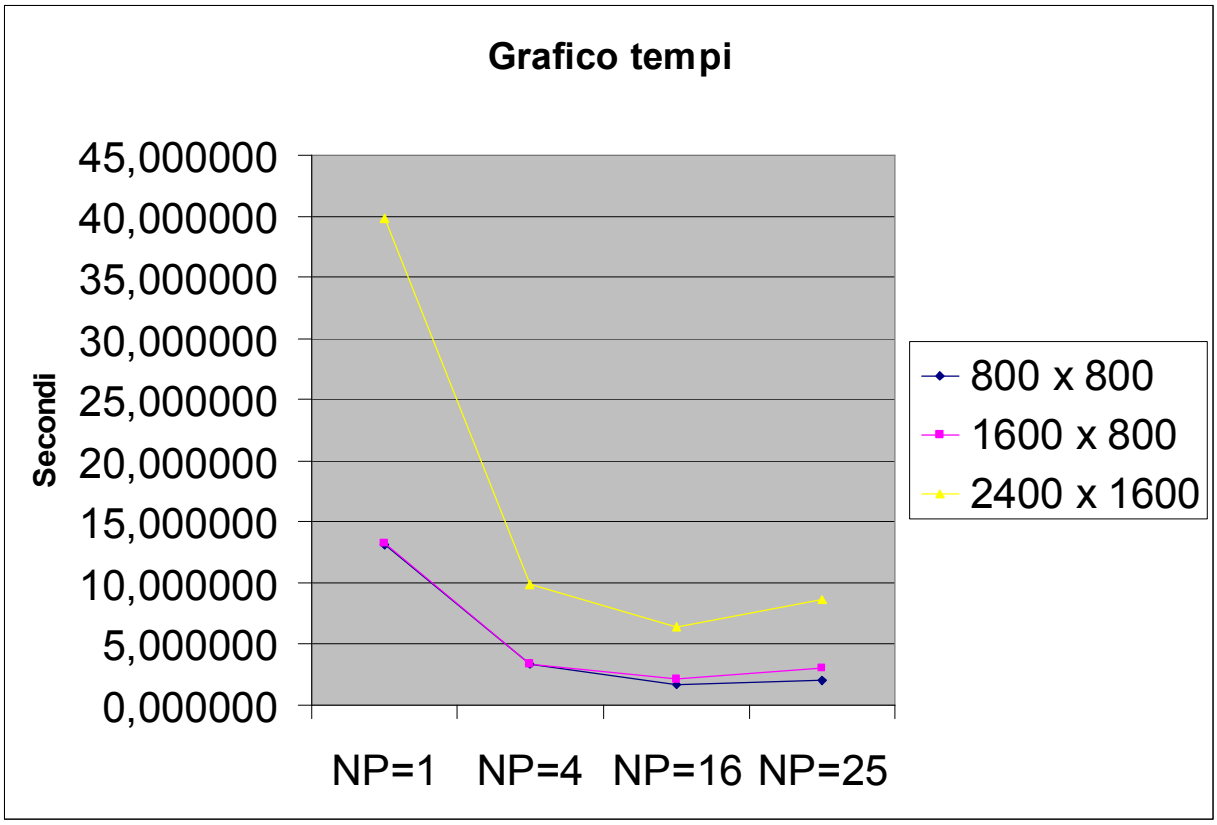
Le tabelle che seguono mostrano i dati raccolti analizzando ciascuna delle caratteristiche sopraelencate.

La prima riga di ogni tabella contiene il *valore dei dati forniti in input*, mentre la prima colonna elenca il *numero di processori impiegati nella computazione*.

Accanto a ciascuna tabella viene mostrato il relativo grafico che evidenzia l'andamento dei valori esaminati.

Tempi delle elaborazioni

	800 x 800	1600 x 800	2400 x 1600
NP=1	13,147956	13,197992	39,845088
NP=4	3,314249	3,420317	9,877586
NP=16	1,709943	2,164640	6,429235
NP=25	2,049647	3,039892	8,668707

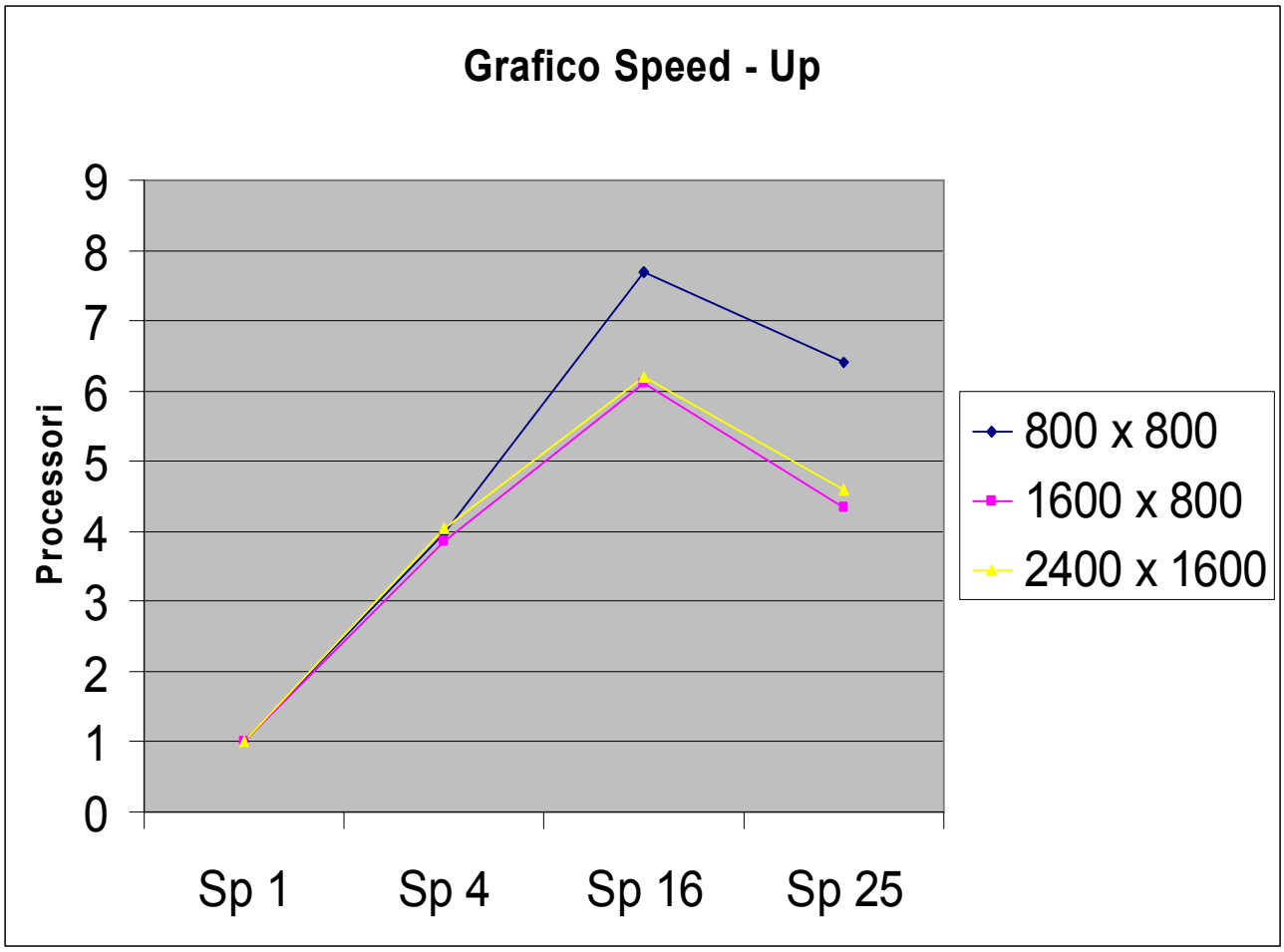


Come si può notare osservando il grafico, il tempo di esecuzione diminuisce sensibilmente aumentando il numero di processori, specialmente con grandi quantità di dati.

Tuttavia, forse a causa della struttura su cui è stata effettuata la fase di testing, la quale dispone di un *numero di processori "reali" inferiore a quello mostrato nel grafico*, è possibile osservare che, a 25 processori, il tempo riprende a salire.

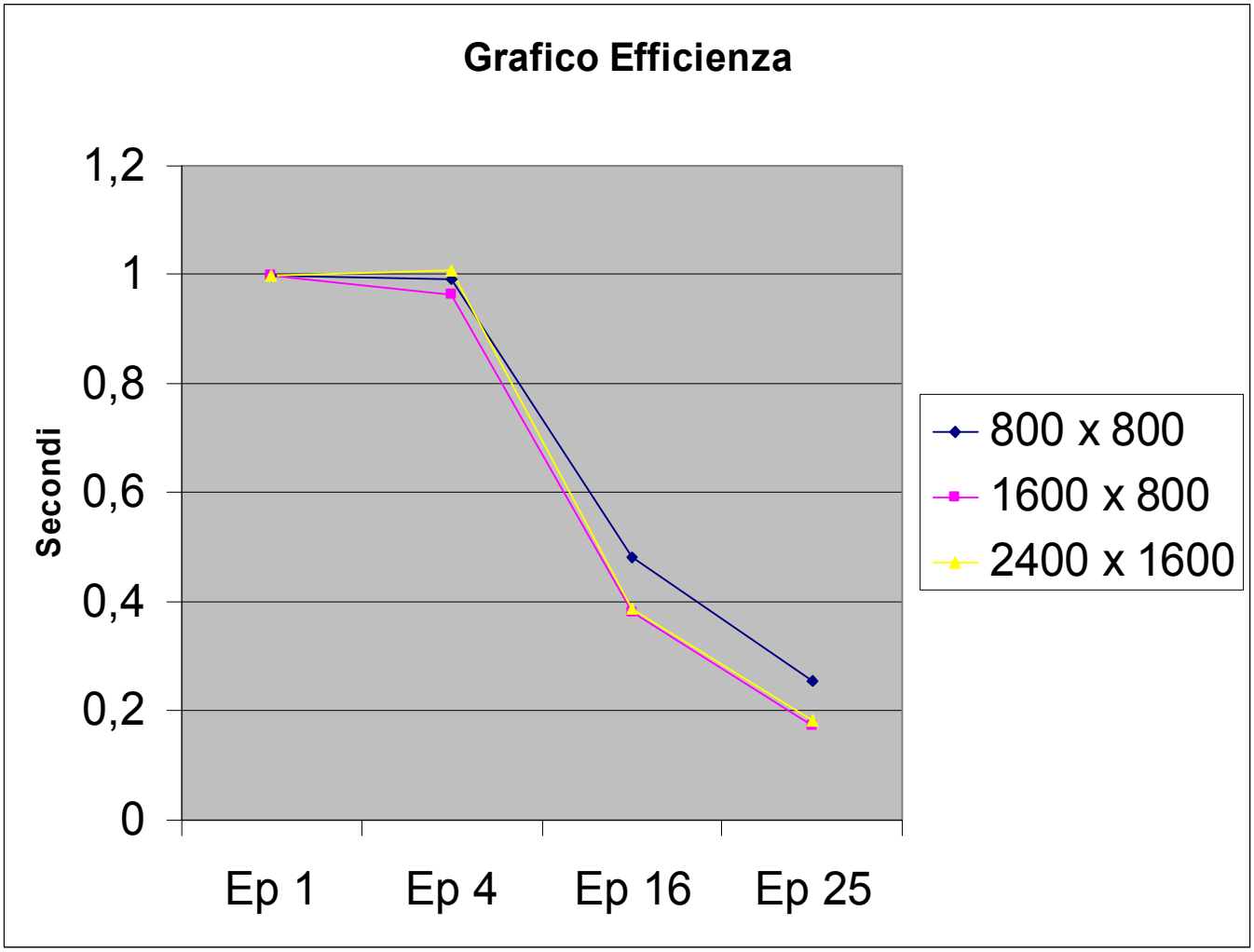
Speed - Up

	800 x 800	1600 x 800	2400 x 1600
Sp 1	1	1	1
Sp 4	3,96709964	3,858704325	4,033889252
Sp 16	7,68911946	6,097084042	6,197485082
Sp 25	6,41474166	4,341598978	4,596428049



Efficienza

	800 x 800	1600 x 800	2400 x 1600
Ep 1	1	1	1
Ep 4	0,99177491	0,964676081	1,008472313
Ep 16	0,48056997	0,381067753	0,387342818
Ep 25	0,25658967	0,173663959	0,183857122



Anche i valori di speed-up ed efficienza confermano quanto enunciato in precedenza, mostrando un peggioramento delle soglie all'aumentare del numero di processori.

Tuttavia,si ribadisce che questo può essere dovuto al fatto che la struttura utilizzata per il testing prevede molti processori virtuali, e questo può tradursi in un peggioramento delle prestazioni.

Essenzialmente, si potrebbe dunque esprimere un giudizio abbastanza positivo sull'algoritmo.

BIBLIOGRAFIA DI RIFERIMENTO

1. A.Murli – Lezioni di Calcolo Parallelo – Ed. Liguori

2. www.mat.uniroma1.it/centro-**calcolo**/HPC/materiale-corso/sliMPI.pdf
   (consultato per interessanti approfondimenti su MPI)

3. www.orebla.it/module.php?n=**c**_num_**casuali**
   (consultato per approfondimenti sulla funzione rand per generare numeri casuali)

4. Bellini – Guidi - LINGUAGGIO C/GUIDA ALLA PROGRAMMAZIONE – McGRAW HILL
   (per un utile ripasso del linguaggio C)

CODICE SORGENTE DEL PROGETTO

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mpi.h"

#include "mxm_vb_aus.c"


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

    int menum, nproc;
    int m, n, k;
    int p, q, flag=0, sinc=1, offsetR=0, offsetC=0, dim_prod_par;
    int N, i, j, z, y, tag, mittente;

    //nella funzione Cart_Rank, indicano il rango del processore avente coordinate prefissate
    int mittenter, destinatarior;

    float *A, *B, *C, *subA, *subB, *subC, *Temp, *printTemp;


    //parametri per valutare le prestazioni
    double t_inizio, t_fine, T1, Tp=0.F, speedup, Ep;

    MPI_Status info;
    MPI_Request rqst;//variabile per la spedizione non bloccante con ISend
    MPI_Comm griglia, grigliar, grigliac; //comunicatori di griglia

    int coordinate[2];
    int mittBcast[2];

    //nella funzione Cart_Rank contengono le coordinate del processore del quale si vuole conoscere il rango
    int destRoll[2];
    int mittRoll[2];

    //Inizializzazione dell'ambiente MPI
    MPI_Init(&argc, &argv);
    //Calcolo dell'identificativo in MPI_COMM_WORLD
    MPI_Comm_rank(MPI_COMM_WORLD, &menum);
    //Calcolo del numero di processori
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);


    //Controllo sul numero dei processori per garantire l'applicabilità della BMR
    if(sqrt(nproc)*sqrt(nproc)!=nproc)
    {
        if(menum==0)
        {
            printf("ERRORE:Impossibile applicare la strategia...\n");
            printf("Il numero di processori deve essere tale da generare una griglia
quadrata.\n");
        }
        MPI_Finalize();
        return 0;
```

```
}



if(menum==0)
{
    system("clear");
    printf("* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *\n");
    printf("* Algoritmo per il calcolo del prodotto matrice per matrice,*\n");
    printf("* \n");
    printf("* Strategia implementata: BMR *\n");
    printf("* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *\n");
    printf("\nProcessori utilizzati: %d\n", nproc);


    //Dimensionamento della matrice A
    while(flag==0)
    {
        printf("\nInserire il numero di righe della matrice A:");
        fflush(stdin);
        scanf("%d", &m);

        //Calcola il valore di p. Esso verrà utilizzato per verificare se le dimensioni delle matrici sono divisibili...
        //...per il numero di processori
        p=sqrt(nproc);

        //Si richiede che il numero di righe di A sia multiplo di p
        if(m%p!=0)
            printf("ATTENZIONE:Numero di righe non divisibile per p=%d!\n", p);
        else
            flag=1;
    }


    while(flag==1)
    {
        printf("\nInserire il numero di colonne della matrice A:");
        fflush(stdin);
        scanf("%d", &n);

        //Si richiede che il numero di righe di A sia multiplo di p
        if(n%p!=0)
            printf("ATTENZIONE:Numero di colonne non divisibile per p=%d!\n", p);
        else
            flag=0;
    }


    //Dimensionamento della matrice B
    while(flag==0)
    {
        printf("\nInserire il numero di colonne della matrice B:");
        fflush(stdin);
        scanf("%d", &k);
```

```
        //Si richiede che il numero di colonne di B sia multiplo di p
        if(k%p!=0)
            printf("ATTENZIONE:Numero di colonne non divisibile per p=%d!\n", p);
        else
            flag=1;
}



//Allocazione dinamica delle matrici A, B e C solo in P_0
//Il numero di righe di B è pari al numero di colonne di A
A=(float *)malloc(m*n*sizeof(float));
B=(float *)malloc(n*k*sizeof(float));
C=(float *)calloc(m*k, sizeof(float));



printf("\n* * * * * * * * * * * * * * * * * * * * * * * * * * * * * \n");
printf("- Acquisizione elementi di A matrice %dx%d -\n", m, n );
printf("\n * * * * * * * * * * * * * * * * * * * * * * * * * * * * * \n");

for(i=0;i<m;i++)
{
    for(j=0;j<n;j++)
    {
        //inserimento manuale
        //printf("Digita elemento A[%d][%d]: ", i, j);
        //scanf("%f", A+i*n+j);

        //per valutare la prestazioni si attiva la riga seguente
        //che genera random la matrice
        *(A+i*n+j)=(float)rand()/((float)RAND_MAX+(float)1);
    }
}


printf("\n* * * * * * * * * * * * * * * * * * * * * * * * * * * * * *\n");
printf("- Acquisizione elementi di B matrice %dx%d -\n",n, k );
printf("\n* * * * * * * * * * * * * * * * * * * * * * * * * * * * * *\n");

for(i=0;i<n;i++)
{
    for(j=0;j<k;j++)
    {
        //inserimento manuale
        //printf("Digita elemento B[%d][%d]: ", i, j);
        //scanf("%f", B+i*k+j);

        //per valutare la prestazioni si attiva la riga seguente
        //che genera random la matrice B
        *(B+i*k+j)=(float)rand()/((float)RAND_MAX+(float)1);
    }
}
```

```
//stampa di A solo se la matrice ha dimensioni minori di 100 x 100

if (m<100 && n<100)
{

            printf("\nMatrice A acquisita:");
            for(i=0;i<m;i++)
            {
                printf("\n");
                for(j=0;j<n;j++)
                    printf("%.2f\t", *(A+i*n+j));
            }


}

else

{

    printf("\nLa matrice A è di grandi dimensioni \n");
    printf("\n e non verrà stampata \n");


}


//stampa di B solo se la matrice ha dimensioni minori di 100X100

if (m<100 && n<100)
{

            printf("\nMatrice B acquisita:");
            for(i=0;i<n;i++)
            {
                printf("\n");
                for(j=0;j<k;j++)
                    printf("%.2f\t", *(B+i*k+j));
            }


}

else

{

    printf("\nLa matrice B è di grandi dimensioni \n");
    printf("\n e non verrà stampata \n");


}
```

```
        printf("\n- - - - - - - - - - - - - - - - - - - - - - - - -\n");
        printf("- Costruzione della griglia (%dx%d) di processori -", p, p);
        printf("\n- - - - - - - - - - - - - - - - - - - - - - - - -\n");
    }




    //Il processore P_0 esegue un Broadcast del numero di righe di A: m
    MPI_Bcast(&m, 1, MPI_INT, 0, MPI_COMM_WORLD);

    //Il processore P_0 esegue un Broadcast del numero di colonne di A: n
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

    //Il processore P_0 esegue un Broadcast del numero di colonne di B: k
    MPI_Bcast(&k, 1, MPI_INT, 0, MPI_COMM_WORLD);

    //Il processore P_0 esegue un broadcast del valore p
    MPI_Bcast(&p, 1, MPI_INT, 0, MPI_COMM_WORLD);

    //Viene creata la griglia
    crea_griglia(&griglia, &grigliar, &grigliac, menum, p, coordinate);

    //Tutti i processori allocano spazio per il blocco subA
    subA=(float *)malloc(m/p*n/p*sizeof(float));

    //Tutti allocano spazio per un blocco d'appoggio
    Temp=(float *)malloc(m/p*n/p*sizeof(float));

    //Tutti allocano spazio per il blocco subB
    subB=(float *)malloc(n/p*k/p*sizeof(float));

    //Tutti allocano spazio per il blocco subC
    subC=(float *)calloc(m/p*k/p, sizeof(float));

    printTemp=(float *)malloc(k/p*sizeof(float));


    if(menum==0)
    {

        for(i=1;i<nproc;i++)
        {

            //Gli offset sono necessari a P_0 per individuare in A il blocco da spedire
            offsetC=m/p*(i/p); //Individua la riga da cui partire
            offsetR=n/p*(i%p); //Individua la colonna da cui partire

            for(j=0;j<m/p;j++)
            {
                tag=10+i;
                //Spedisce gli elementi in vettori di dimensione n/p
                MPI_Send(A+offsetC*n+offsetR, n/p, MPI_FLOAT, i, tag, MPI_COMM_WORLD);
                offsetC++; //l'offset di colonna viene aggiornato
```

```
            }

        }


        for(i=1;i<nproc;i++)
        {

                //Stessa cosa avviene per spedire sottoblocchi di B
                offsetC=n/p*(i/p);
                offsetR=k/p*(i%p);

                for(j=0;j<n/p;j++)
                {
                    tag=20+i;
                    //Spedisce gli elementi di B in vettori di dimensione k/p
                    MPI_Send(B+offsetC*k+offsetR, k/p, MPI_FLOAT, i, tag,
MPI_COMM_WORLD);
                    offsetC++;  //l'offset di colonna viene aggiornato
                }

        }


        //P_0 inizializza il suo blocco subA
        for(j=0;j<m/p;j++)
        {

            for(z=0;z<n/p;z++)
                *(subA+n/p*j+z) = *(A+n*j+z);

        }


        //P_0 inizializza il suo blocco subB
        for(j=0;j<n/p;j++)
        {
                for(z=0;z<k/p;z++)
                    *(subB+k/p*j+z) = *(B+k*j+z);
        }

    } //end then


    else

    {

        //Tutti i processori ricevono il rispettivo subA....

        for(j=0;j<m/p;j++)
        {
                tag=10+menum;
                MPI_Recv(subA+n/p*j, n/p, MPI_FLOAT, 0, tag, MPI_COMM_WORLD, &info);
        }
```

```
        //....e ricevono il blocco subB
        for(j=0;j<n/p;j++)
            {
                tag=20+menum;
                MPI_Recv(subB+k/p*j, k/p, MPI_FLOAT, 0, tag, MPI_COMM_WORLD, &info);
            }

    }


        //Tutti i processori hanno i rispettivi blocchi subA e subB
        for(i=0;i<p;i++)
        {
            //al primo passo fa broadcast e prodotto
            if(i==0)
            {
                //Calcola le coordinate del processore che invierà in broadcast
                //la propria matrice subA ai processori sulla stessa riga
                mittBcast[0]=coordinate[0];
                mittBcast[1]=(i+coordinate[0])%p;

                //Calcola le coordinate del processore a cui inviare subB
                destRoll[0]=(coordinate[0]+p-1)%p;
                destRoll[1]=coordinate[1];

                //Calcola le coordinate del processore da cui ricevere la nuova subB
                mittRoll[0]=(coordinate[0]+1)%p;
                mittRoll[1]=coordinate[1];

                //Ricava il rango del destinatario sulla propria colonna
                MPI_Cart_rank(grigliac, destRoll, &destinatarior);

                //Ricava il rango del mittente sulla propria colonna
                MPI_Cart_rank(grigliac, mittRoll, &mittenter);

                //Ricava il rango del processore che effettua il Broadcast sulla propria riga
                MPI_Cart_rank(grigliar, mittBcast, &mittente);

                if(coordinate[0] == mittBcast[0] && coordinate[1]==mittBcast[1])
                {
                    //L'esecutore del broadcast copia subA nel blocco temporaneo Temp
                    memcpy(Temp, subA, n/p*m/p*sizeof(float));
                }

                t_inizio=MPI_Wtime();

                MPI_Bcast(Temp, n/p*m/p, MPI_FLOAT, mittente, grigliar);

                t_fine=MPI_Wtime();

                Tp+=t_fine-t_inizio;

                Tp+=mat_mat_righe(Temp, m/p, n/p, subB, k/p, subC);
```

```
            }

            else  //Se non è il primo passo esegue Broadcast, Rolling e Prodotto

            {
                mittBcast[0]=coordinate[0];
                mittBcast[1]=(i+coordinate[0])%p;


                if(coordinate[0] == mittBcast[0] && coordinate[1]==mittBcast[1])
                {
                    //L'esecutore del broadcast copia subA nel blocco temporaneo Temp
                    memcpy(Temp, subA, n/p*m/p*sizeof(float));
                }

                mittente=(mittente+1)%p;
                t_inizio=MPI_Wtime();

                //Broadcast di Temp
                MPI_Bcast(Temp, n/p*m/p, MPI_FLOAT, mittente, grigliar);

                //Il rolling vede l'invio del blocco subB al processore della riga superiore
                tag=30;  //La spedizione è non bloccante mentre la ricezione si
                MPI_Isend(subB, n/p*k/p, MPI_FLOAT, destinatarior, tag, grigliac, &rqst
);

                //E la ricezione del nuovo blocco subB dalla riga inferiore
                tag=30;
                MPI_Recv(subB, n/p*k/p, MPI_FLOAT, mittenter, tag, grigliac, &info);
                t_fine=MPI_Wtime();

                //Calcola il prodotto parziale e il tempo impiegato per eseguirlo
                Tp+=mat_mat_righe(Temp, m/p, n/p, subB, k/p, subC)+t_fine-t_inizio;

            }// end else

        }//end for


        //Tutti i processori in ordine inviano a P_0 la propria porzione di C
        if(menum==0)
        {
            //P_0 stampa così come le riceve le porzioni ricevute
            printf("\n* * * * * * * * * * * * * * * * * * * * * * * * * * * * * *\n");
            printf("* Stampa del risultato e dei parametri di valutazione *\n");
            printf("\n* * * * * * * * * * * * * * * * * * * * * * * * * * * * * *\n");



            //stampa di C solo se ha dimensioni minori di 100 X 100

            if (m < 100 && k < 100)

            {
```

```
                printf("Matrice risultato:\n");
                for(i=0;i<p;i++)
                {  //per quante sono le righe di C
                    for(z=0;z<m/p;z++)
                    {  //per quante sono le righe di subC
                        for(j=0;j<p;j++)
                        {  //per quanti sono i processori per riga
                            if(i*p+j!=0)
                            {
                                tag=70;
                                //Riceve le righe delle varie matrici subC nell'ordine di stampa
                                MPI_Recv(printTemp, k/p, MPI_FLOAT, i*p+j, tag,
MPI_COMM_WORLD, &info);
                                for(y=0;y<k/p;y++)
                                //Stampa la porzione di riga di C ricevuta
                                printf(" %.2f\t", *(printTemp+y));
                            }
                            else
                                //P_0 stampa le righe della propria matrice subC
                                for(y=0;y<k/p;y++)
                                printf(" %.2f\t", *(subC+k/p*z+y));
                        }
                        printf("\n");
                    }
                }

            }

            else

            {

            printf("\n La matrice C è di grandi dimensioni");
            printf("\ne non verrà stampata . ");

            }



            //Calcolo del prodotto con singolo processore e del tempo necessario ad eseguirlo
            T1=mat_mat_righe(A, m, n, B, k, C);

            speedup=T1/Tp;  //Calcola lo speed up
            Ep=speedup/nproc;  //Calcola l'efficienza

            //Stampa dei risultati ottenuti
            printf("\n\nIl tempo di esecuzione su un processore e' %f secondi\n", T1);
            printf("Il tempo di esecuzione su %d processori e' %f secondi\n", nproc, Tp
);
            printf("Lo Speed Up ottenuto e' %f\n", speedup);
            printf("L'efficienza risulta essere %f\n\n", Ep);


            //Deallocazione della memoria allocata da P_0 dinamicamente
```

```
            free(A);
            free(B);

        }

        else

        {
            for(i=0;i<m/p;i++)
            {
                tag=70;
                MPI_Send(subC+i*k/p, k/p, MPI_FLOAT, 0, tag, MPI_COMM_WORLD);
            }

        }



        //Deallocazione della memoria allocata dinamicamente da tutti i processori
        free(Temp);
        free(printTemp);
        free(subA);
        free(subB);

        MPI_Finalize();

        return(0);

}
```

```
/*
 ************************************************
 *    ALGORITMO PER IL CALCOLO PARALLELO  *
 *    DEL PRODOTTO MATRICE PER MATRICE   *
 *          III   STRATEGIA                      *
 *                                               *
 *        LIBRERIA AUSILIARIA                    *
 *           DI FUNZIONI                         *
 *                                               *
 *      VIRGINIA     BELLINO                     *
 *       MATR. 108/1570                          *
 ************************************************
*/


#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include "mpi.h"


/*PROTOTIPI DELLE FUNZIONI*/
double mat_mat_righe(float *, int, int, float *, int, float *);

void crea_griglia(MPI_Comm *, MPI_Comm *, MPI_Comm *, int, int, int *);




/*************************************************************************/
/*********FUNZIONI AUSILIARIE*********** *********/
/*************************************************************************/

/*FUNZIONE crea_griglia*/
//crea una griglia bidimensionale periodica utile a combinare i
//risultati parziali ottenuti da tutti i processori

void crea_griglia(MPI_Comm *griglia, MPI_Comm *grigliar, MPI_Comm *grigliac, int menum,
 int lato, int *coordinate){

    int menum_griglia, reorder;
    int *ndim, *period, dim=2; //Le dimensioni della griglia sono 2
    int vc[2]; //

    //Specifica il numero di processori in ogni dimensione
    //la griglia risulta quadrata,poichè entrambe le componenti di ndim hanno lo stesso valore
    ndim=(int*)calloc(dim, sizeof(int));
    ndim[0]=lato;
    ndim[1]=lato;

    period=(int*)calloc(dim, sizeof(int));
    period[0]=period[1]=1; //La griglia è periodica
    reorder=0; //la griglia non ha un ordine particolare

    //funzuine MPI che crea la griglia
```

```
    MPI_Cart_create(MPI_COMM_WORLD, dim, ndim, period, reorder, griglia);

    //Ricava il rango del processore all'interno della griglia
    MPI_Comm_rank(*griglia, &menum_griglia);

    /*Il processore di id menum calcola le proprie coordinate...*/
    /*..nel comunicatore griglia creato*/
    MPI_Cart_coords(*griglia, menum_griglia, dim, coordinate);
    //printf("\n");
    //printf("- Processore #%d, coordinate nella griglia: (%d,%d)", menum, *coordinate, *(coordinate+1));

    vc[0]=0;
    vc[1]=1;
    MPI_Cart_sub(*griglia, vc, grigliar); //Partiziona il communicator nel sottogruppo grigliar

    vc[0]=1;
    vc[1]=0;
    MPI_Cart_sub(*griglia, vc, grigliac); //Partiziona il communicator nel sottogruppo grigliac

}




//Function per il CALCOLO DEL PRODOTTO MATRICE MATRICE

double mat_mat_righe(float *A, int m, int n, float *B, int k, float *C)
{

    short i, j, z;
    double t_inizio, t_fine, tempo=0.F;

    t_inizio=MPI_Wtime();
    //L'azzeramento di C non viene effettuato  per consentire che chiamate
    //successive della function aggiornino la matrice C e non sovrascrivano

    for(i=0;i<m;i++)
        for(z=0;z<n;z++)
            for(j=0;j<k;j++)
            {
                //l'accesso ad A, B e C è per righe
                //calcolo del prodotto scalare tra la riga i-sima di A e la colonna j-sima di B
                *(C+i*k+j)+=*(A+i*n+z)*(*(B+k*z+j));
            }

    t_fine=MPI_Wtime();

    tempo+=t_fine-t_inizio;

    return tempo;
}
```

```
# Questo script serve a mandare in esecuzione il programma inserendo
# solo il numero di processori da usare

# Esempio d'uso in ambiente Linux

# $ go.sh 4

mpirun -np $1 -machinefile lista a.out
```