Università degli Studi di Napoli
"Parthenope"

Corso di Calcolo Parallelo e Distribuito

Progetto Matrice per Vettore
III Strategia

Virginia Bellino
Matr. 108/1570

Indice

INDIVIDUAZIONE ED ANALISI DEL PROBLEMA

DEFINIZIONE DEL PROBLEMA

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

La matrice deve essere distribuita a **p x q** processi, disposti secondo una ***topologia a griglia bidimensionale***.

L'algoritmo deve 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**

L'algoritmo implementa la III strategia di calcolo in parallelo per il prodotto matrice vettore.

Tale strategia prevede la decomposizione della matrice e del vettore di input in blocchi di dimensione fissata, i quali verranno poi assegnati ai processori che sono logicamente organizzati in una *topologia a griglia bidimensionale.*

Ciascun processore della griglia calcola una parte del risultato, che verrà successivamente scambiata con gli altri processori, in modo da ottenere, alla fine, il vettore risultato.

Riepilogando, la terza strategia prevede:

-decomposizione della matrice di input in blocchi di dimensione prefissata

-decomposizione del vettore di input

-assegnazione delle sottomatrici e dei sottovettori a ciascun processore situato lungo la griglia bidimensionale

-ciascun processore della griglia calcola parte del risultato

-scambio dei risultati parziali tra i processori che, dopo aver eseguito le opportune operazioni, ottengono il risultato finale

Si può quindi affermare che la terza strategia rappresenti un "ibrido" tra la prima strategia di calcolo (che prevede la decomposizione della matrice di input in blocchi di righe e l'assegnazione dell'intero vettore di input a ciascun processore), e la seconda strategia (che prevede la decomposizione della matrice di input in blocchi di colonne e l'assegnazione di una parte del vettore di input a ciascun processore).

- **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, se l'identificativo di processore corrisponde a quello del processore P0, si ha quanto segue:

-viene chiesto all'utente di inserire il numero di righe e di colonne della matrice

-un primo controllo verifica se le dimensioni inserite sono corrette

```
//controllo sulla correttezza dei dati inseriti
        if(n_righe*n_colonne<nproc)
                printf("ERRORE: Il numero di elementi risulta inferiore al numero di processori!\n");
        else if(primo(nproc) && n_righe<nproc && n_colonne<nproc)
                printf("ERRORE: Impossibile partizionare tale matrice per %d processori \n", nproc);
            else
                flag=1;//uscita
```

In primo luogo, viene controllato se il numero di elementi inseriti è inferiore o meno al numero di processori. In questo caso infatti, l'algoritmo segnala un errore e chiede nuovamente all'utente di inserire le dimensioni della matrice (**situazione 1**).

Analoga situazione si verifica se il numero di processori è primo e se la dimensione degli elementi inseriti risulta essere inferiore a nproc *(per schermate di esempio esplicative si consulti il paragrafo "indicatori di errore" presente nella sezione "Analisi del Software")*. (**situazione 2**)

-viene poi verificato, richiamando l'apposita funzione PRIMO, se le dimensioni inserite sono un numero primo.

In tal caso, viene applicata la prima o la seconda strategia, che sono più convenienti.

-se invece il numero di processori non è primo viene valutato se è possibile applicare la 3 strategia

```
else

/*....se il numero di processori non è primo.... */
/*....si valuta se è possibile applicare la terza strategia*/
{
flag=0;
        while(flag==0)
        {
                printf("\nIndicare in quante parti si vuole dividere il numero di righe:");
                fflush(stdin);
                scanf("%d", &p);
                q=nproc/p; //q è calcolato di conseguenza

                 /*Se ci sono errori ...*/
                 if(n_righe<p || p>nproc || (p*q)%nproc!=0 || n_colonne<q)
                        printf("ERRORE: Impossibile partizionare i dati nel modo indicato!\n");
                else
                        flag=1; /* esci*/
        }

 }
```

(per schermate di esempio esplicative si consulti il paragrafo "indicatori di errore" presente nella sezione "Analisi del Software") (**situazione 3**)

-si allocano poi le strutture dati per la matrice e per il vettore di cui verrà chiesto l'inserimento.

Tale inserimento potrà essere….

→ manuale
(consentendo cosi all'utente di verificare personalmente il corretto funzionamento dell'algoritmo)

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

→ random
(per consentire all'utente di verificare personalmente le prestazioni dell'algoritmo)

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

La scelta può essere effettuata decidendo quale delle suddette opzioni attivare, disattivando naturalmente l'altra.

-se matrice e vettore in input hanno dimensioni inferiori a 100, vengono stampati a video, altrimenti si prosegue.

Il processore P0 esegue poi un broadcast per comunicare a tutti i processori del comunicatore le dimensioni della matrice di input, tutti allocano la struttura dati che accoglierà il risultato e, dopo un nuovo broadcast per comunicare le dimensioni della griglia, viene chiamata l'apposita funzione per creare la medesima.

```
//chiamata della funzione che costruirà la topologia dei processori
//a griglia bidimensionale
   crea_griglia(&griglia, &grigliar, &grigliac, menum, p, q, coordinate);
```

La **funzione CREA_GRIGLIA**, usando l'apposita routine MPI, prima crea una griglia bidimensionale non periodica…..

```
period=(int*)calloc(dim, sizeof(int));
period[0]=period[1]=0; //La griglia non è periodica
```

…..e non ordinata con alcun particolare criterio….

```
reorder=0;
```

poi suddivide la griglia creata in due sottogriglie, una per le righe, e una per le colonne della matrice.

```
MPI_Cart_sub(*griglia, vc, grigliar);  //Partiziona il communicator nel sottogruppo grigliar

MPI_Cart_sub(*griglia, vc, grigliac);  //Partiziona il communicator nel sottogruppo grigliac
```

-Successivamente, vengono stabilite le dimensioni dei dati da inviare (blocco di matrice e frazione del vettore di input), allocando le relative strutture dati, poi il processore P0 provvede all'invio dei suddetti dati.

Terminata la FASE DI DISTRIBUZIONE, inizia la fase di Calcolo Locale, in cui viene richiamata la **funzione MAT_VET** che calcola il prodotto parziale matrice vettore di ciascun processore e il tempo impiegato

```
//Calcola il prodotto Blocco*sub_x=sub_y e il tempo per eseguirlo con p processori
Tp=mat_vet(Blocco, sub_x, Righe_Blocco, Colonne_Blocco, sub_y);
```

Tale function, essenzialmente….

-calcola il prodotto matrice vettore

-rileva il tempo di inizio e di fine della suddetta operazione, restituendo poi alla funzione chiamante il tempo totale impiegato per effettuare l'operazione con p processori

```
//rilevazione tempo di inizio
  t_inizio=MPI_Wtime();

*(C+i)=*(C+i)+*(A+i*n+j)*(*(B+j));

//rilevazione tempo di fine
 t_fine=MPI_Wtime();

//Si accumula l'incremento ad ogni ciclo
 tempo=tempo+t_fine-t_inizio;
```

La chiamata alla funzione ALL_Reduce consente di…..

```
//Con la routine MPI si sommano i vettori parziali sulle righe della griglia
MPI_Allreduce(sub_y, prod_par, Righe_Blocco, MPI_FLOAT, MPI_SUM, grigliar);
```

..e poi la fase di calcolo si chiude con il calcolo del risultato da parte del processore P0.

L'algoritmo si chiude con la *stampa del risultato finale* (il vettore risultato viene però stampato solo se il numero degli elementi è minore di 100) e con la *stampa dei valori dei parametri di valutazione delle prestazioni* del software parallelo (in base al numero di processori utilizzati nel calcolo).

```
//Calcolo del prodotto e del tempo con singolo processore
T1=mat_vet(A, x, n_righe, n_colonne, y);

speedup=T1/Tp: //Calcola lo speed up

Ep=speedup/nproc;  //Calcola l'efficienza

printf("\n\nIl tempo di esecuzione usando 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 con %d processori e' %f secondi\n\n",nproc, Ep)
```

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_matricexvet_vb.c
```

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

```
$ mpirun –np x a.out
```

Note:

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

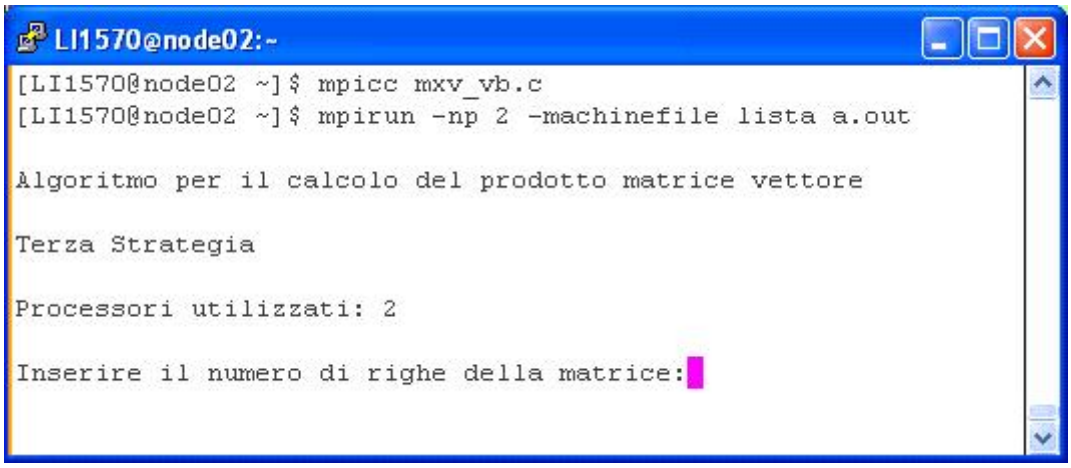
a.out indica generalmente il nome dell'eseguibile

x è il numero di processori che si desidera impiegare nel calcolo

- ***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.

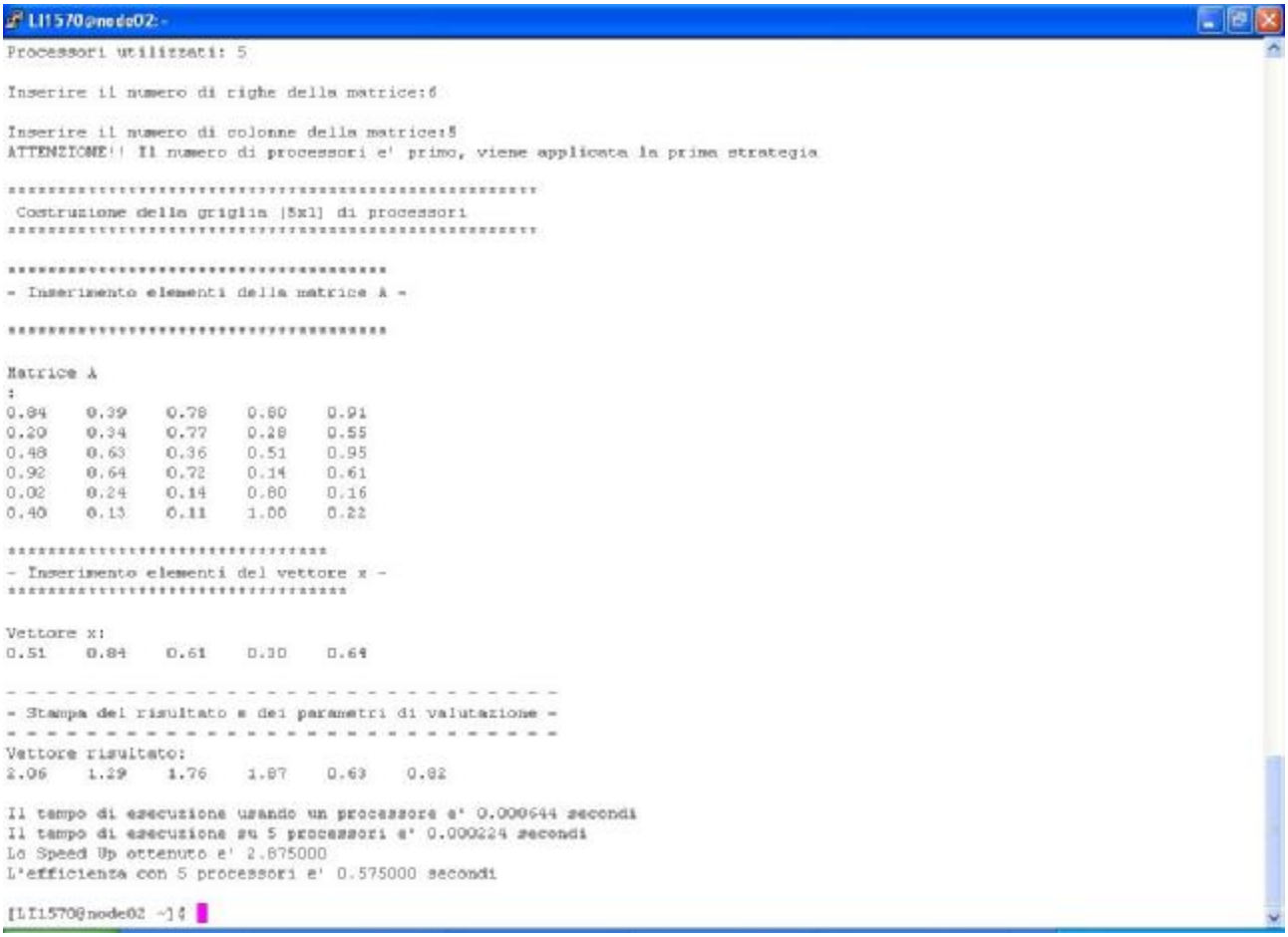
Esempio 1:



Come si può notare, dopo le istruzioni di compilazione ed esecuzione, viene stampato un messaggio di carattere informativo che evidenzia, tra l'altro, il numero di processori utilizzati nel calcolo, poi viene richiesto all'utente di inserire il numero di righe e di colonne della matrice (non evidenziato nell'esempio)

.

Esempio 2:

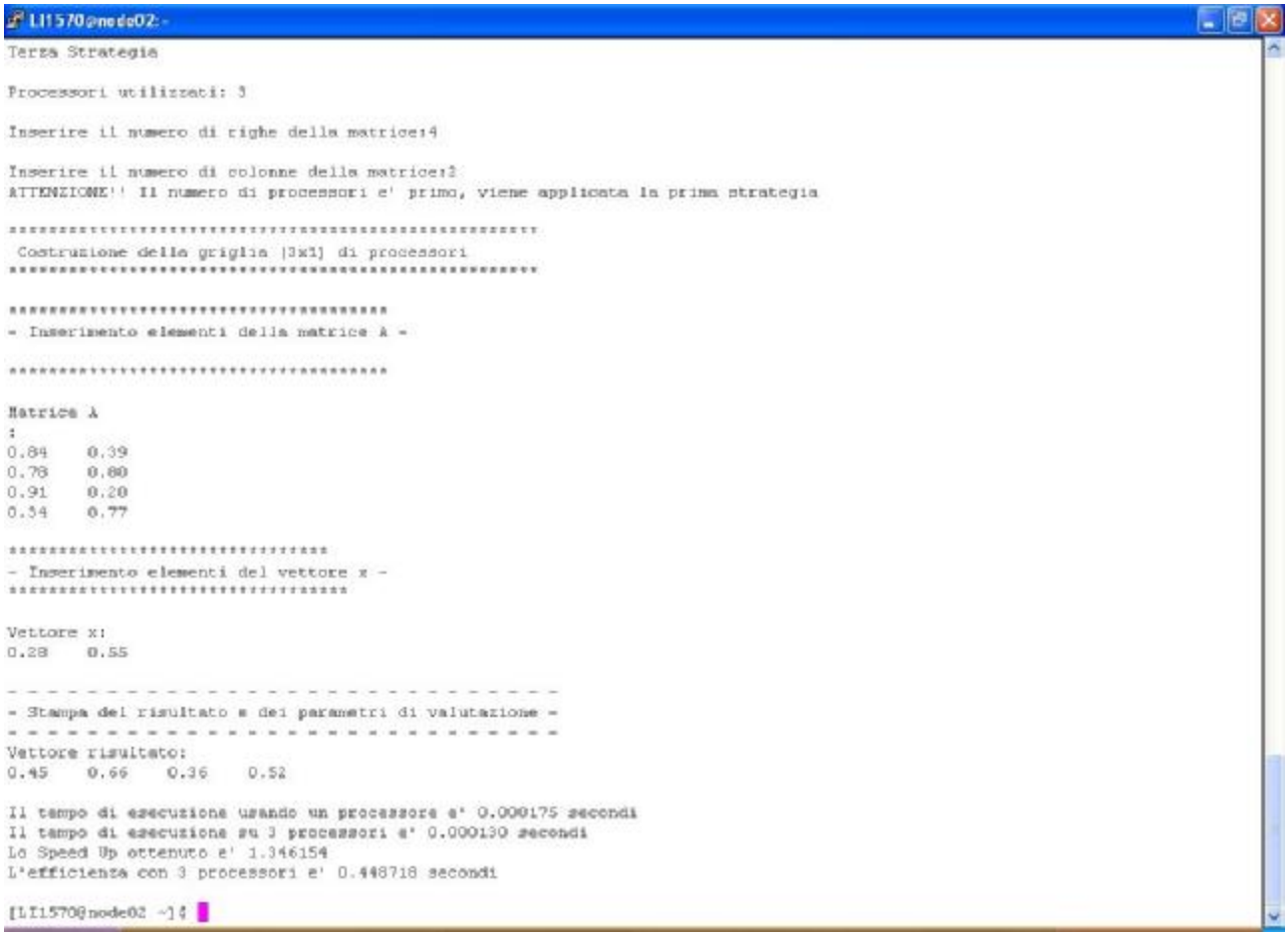


Questo secondo esempio mostra invece cosa appare all'utente dopo aver inserito il numero di righe e di colonne della matrice.

Se le dimensioni della matrice e del vettore di input (generati manualmente o random) non superano 100, vengono stampati a video (situazione che si ripete anche per il vettore risultato).

Segue la stampa dei parametri di valutazione delle prestazioni.

Esempio 3:

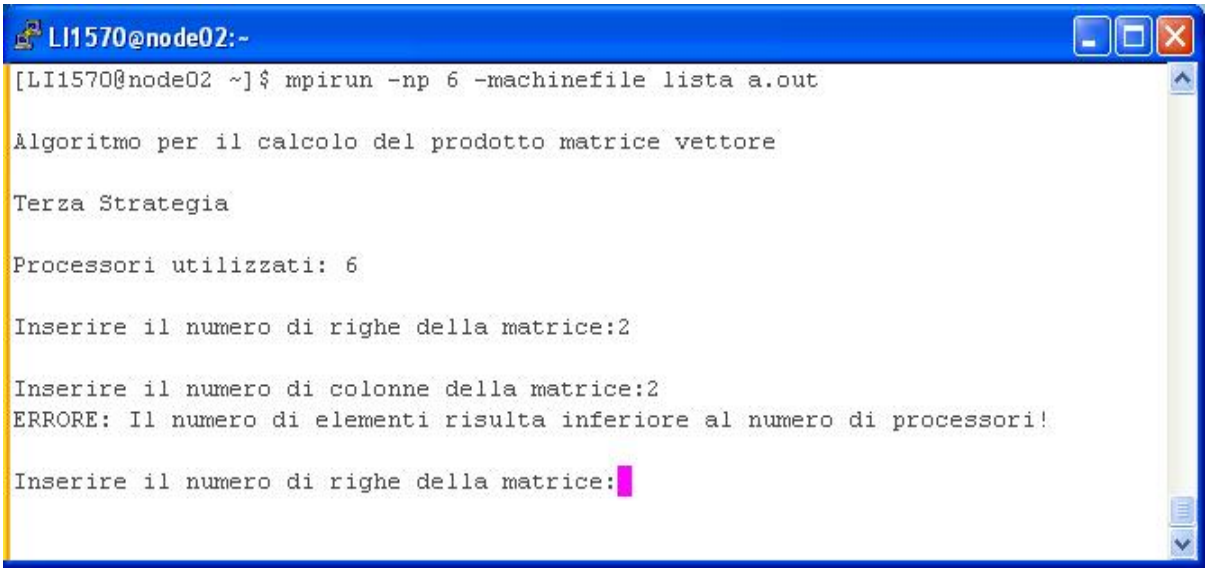


Esempio analogo al precedente.
Unica differenza è la mancata stampa della matrice di input,del vettore di input e del vettore risultato, perché la loro dimensione supera 100.

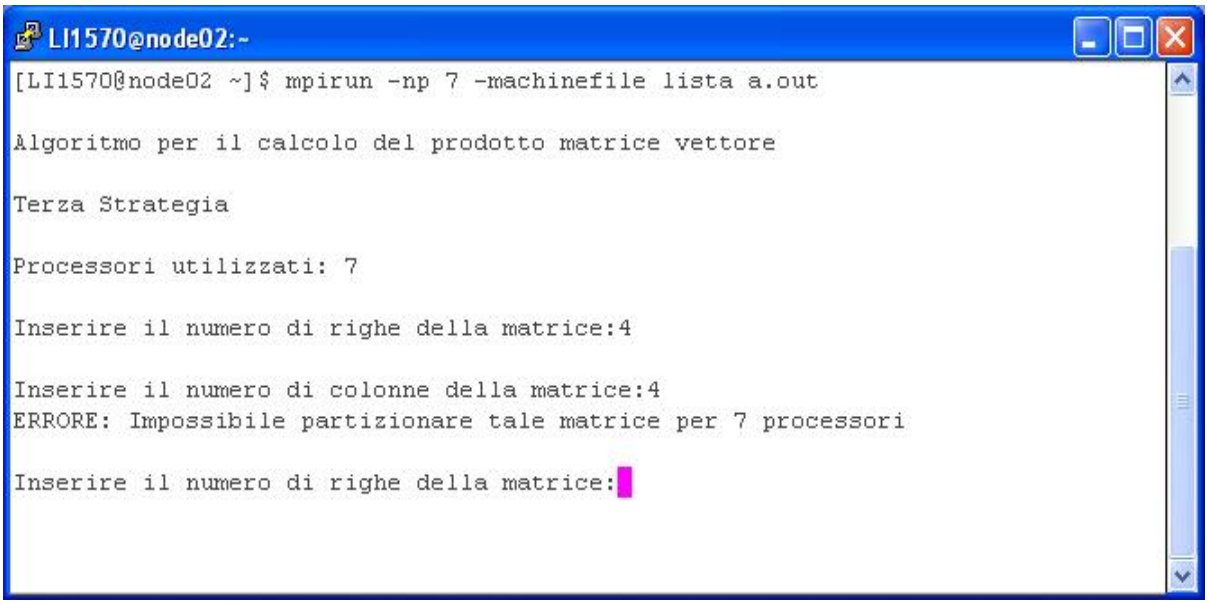
- ***Indicatori di errore***

  Le seguenti schermate mostrano gli errori descritti nella sezione *"Descrizione dell'algoritmo ->Codice"*

  Situazione 1

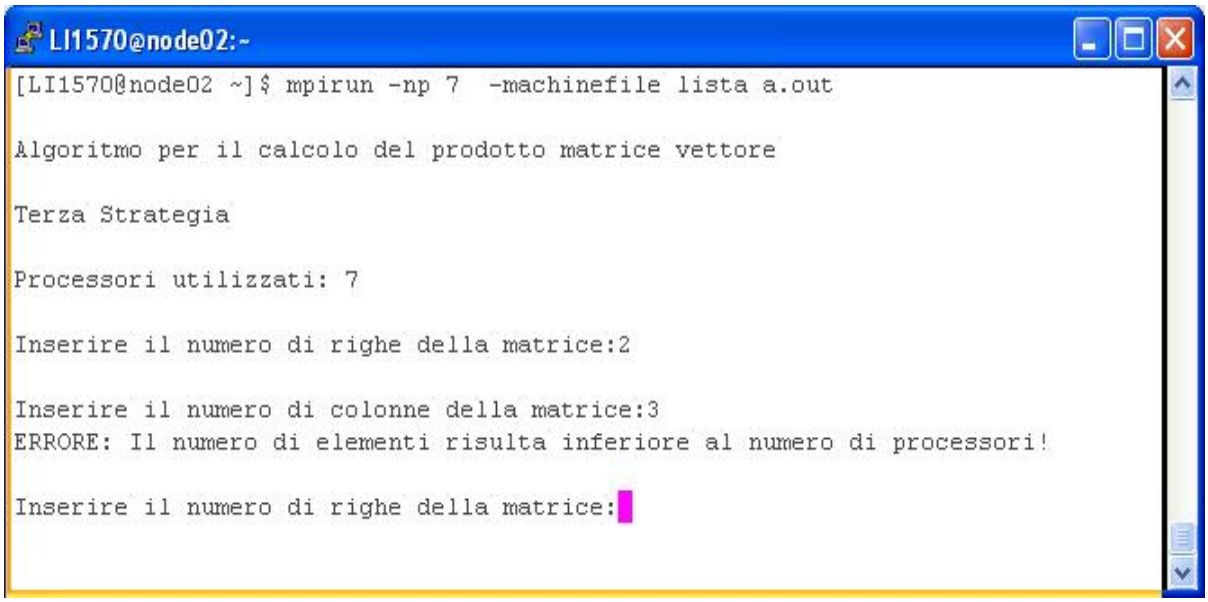


La prima schermata mostra che è stato inserito un numero di elementi inferiore al numero di processori, e questo genera un errore.



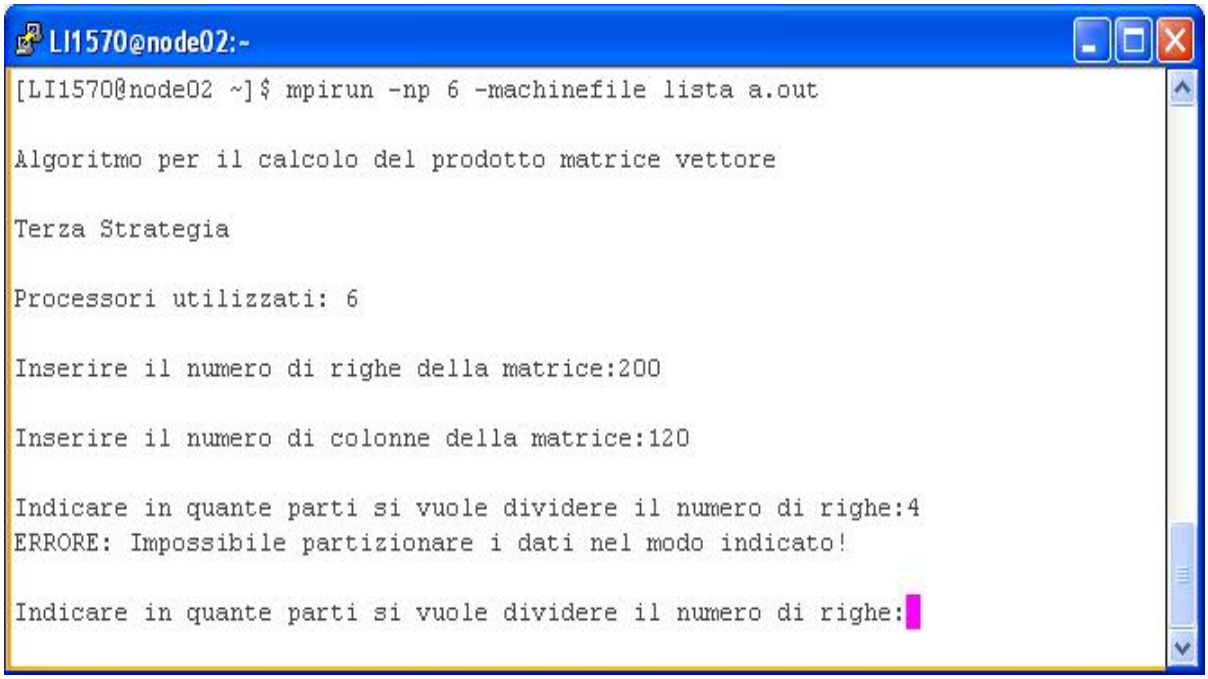
Altra situazione di errore, che si verifica quando il numero di processori è primo e viene inserito un numero di righe e di colonne inferiore ad esso.

Situazione 2



Errore analogo a quello mostrato nella prima schermata della situazione 1, solo che il numero di processori utilizzati è primo.

Situazione 3



Controllo sulla possibilità di applicare realmente la 3 strategia. Quando viene richiesto in quante parti dividere le righe della matrice, l'utente deve inserire un *divisore primo del numero di processori o il numero di processori stesso*.

Ad esempio, per 6 processori si può inserire 1 – 2 – 3 – 6

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(&n_righe, 1, MPI_INT, 0, MPI_COMM_WORLD);**

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

I parametri sono:

n_righe: 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+(j+1)*offset_r+offset_c+(n_colonne-offset_r)*j+k, 1, MPI_FLOAT, i,tag, MPI_COMM_WORLD);**
  spedizione di dati

  I parametri sono:
  *A+(j+1)*offset_r+offset_c+(n_colonne-offset_r)*j+k (input):* indirizzo del dato da spedire
  1 (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(dim_Blocco, 2, MPI_INT, i, tag, MPI_COMM_WORLD, &info);**
  ricezione di dati

  I parametri sono:
  dim_blocco: indirizzo del dato su cui ricevere
  2 : numero dei dati da ricevere
  MPI_INT : 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

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

Funzioni per operazioni collettive in ambiente MPI

- **MPI_Allreduce(sub_y, prod_par, Righe_Blocco, MPI_FLOAT, MPI_SUM, grigliar);**

  sub_y : indirizzo dei dati su cui effettuare l'operazione
  prod_par : indirizzo del dato contenente il risultato
  1 : numero dei dati su cui effettuare l'operazione
  MPI_FLOAT : tipo degli elementi da spedire
  MPI_sum : operazione effettuata
  grigliar: identificativo del processore che conterrà il risultato
  MPI_COMM_WORLD: identificativo del comunicatore

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

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

/*FUNZIONE primo*/

- **int primo(int N)**

ritorna 1 se il parametro in input è un numero primo.
Questa funzione controlla se il numero di processori (e le dimensioni della griglia) sono un numero primo. In tal caso, viene consigliata l'applicazione della prima o della seconda strategia, che risultano più convenienti.

Parametri:
N: dimensione della griglia (cfr. *N=n_righe*n_colonne;*)

/*FUNZIONE crea_griglia*/

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

crea una griglia bidimensionale non 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
*righe,colonne:*dimensioni della griglia
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_vet(float *A, float *B, int m, int n, float *C)**

esegue il calcolo del prodotto matrice vettore

*A***:** sottoblocco della matrice di input

B: frazione del vettore di input

m,n : dimensioni del sottoblocco di matrice ricevuto

C:vettore 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

	80.000	250.000	1.048.576
NP=1	1,555448	4,795473	20,308637
NP=2	0,776547	2,424454	10,161322
NP=3	0,523016	1,638161	6,696361
NP=4	0,388727	1,227661	5,070732
NP=5	0,311704	0,978842	4,086271
NP=6	0,260511	0,825524	3,376345
NP=7	0,225852	0,774564	2,883355
NP=8	0,278171	0,634556	2,568724

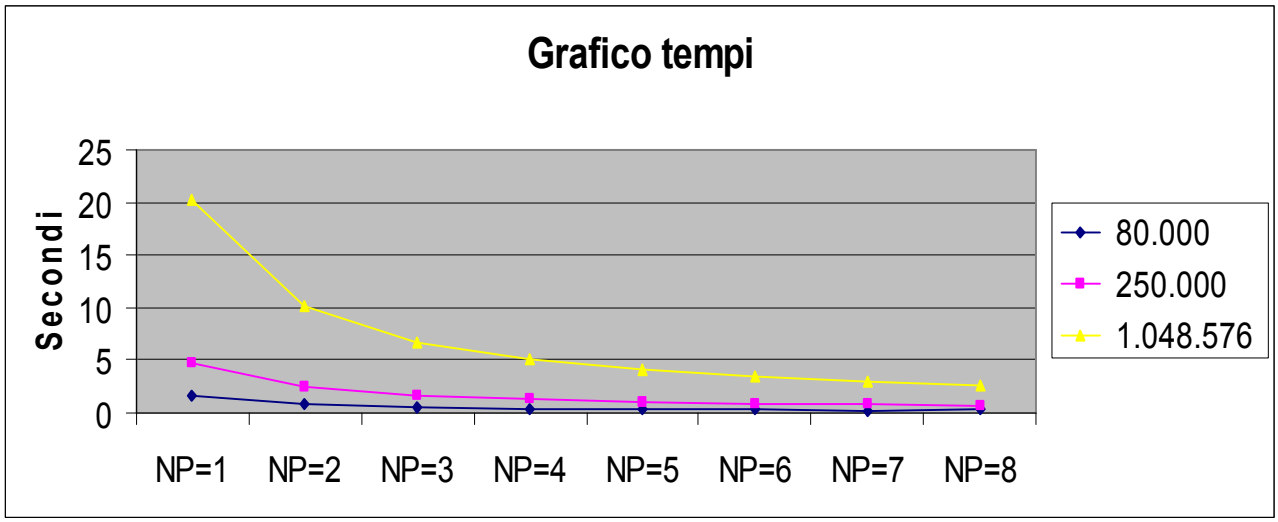


Tabella Speed-Up

	80.000	250.000	1.048.576
Sp 1	1	1	1
Sp 2	2,00303137	1,977959986	1,998621538
Sp 3	2,97399697	2,927351463	3,032787062
Sp 4	4,00138915	3,906186643	4,005070077
Sp 5	4,9901445	4,899128766	4,969968218
Sp 6	5,97075747	5,809004947	6,014976846
Sp 7	6,88702336	6,191190141	7,043404992
Sp 8	5,59169719	7,557210081	7,906118758

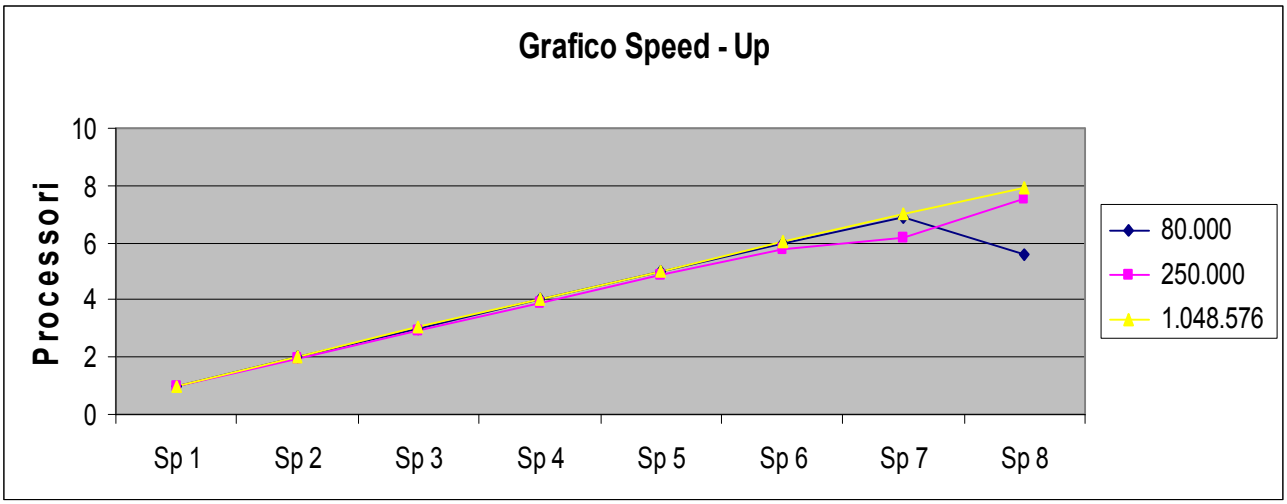
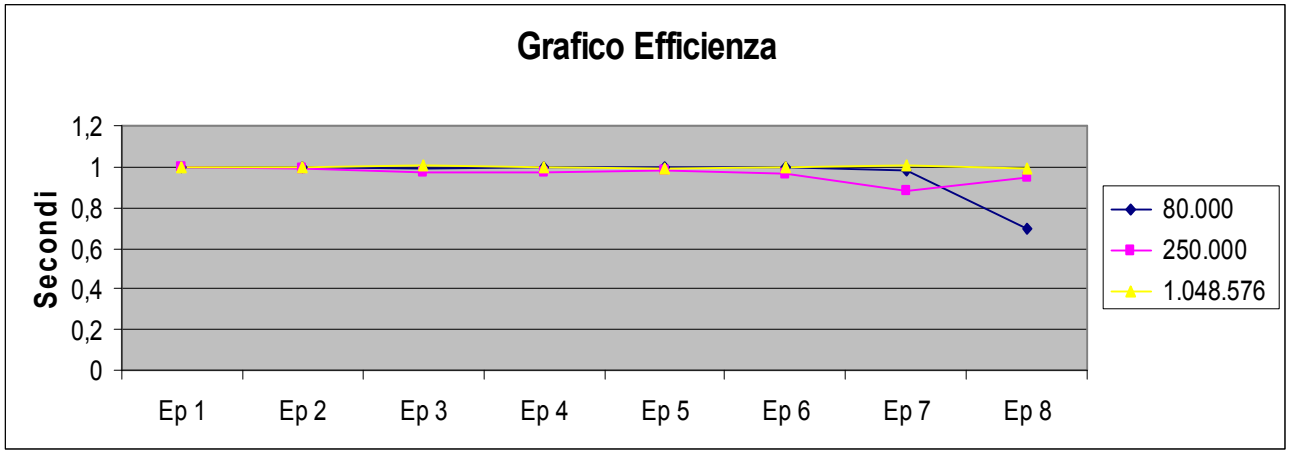


Tabella Efficienza

	80.000	250.000	1.048.576
Ep 1	1	1	1
Ep 2	1,00151568	0,988979993	0,999310769
Ep 3	0,99133232	0,975783821	1,010929021
Ep 4	1,00034729	0,976546661	1,001267519
Ep 5	0,9980289	0,979825753	0,993993644
Ep 6	0,99512625	0,968167491	1,002496141
Ep 7	0,98386048	0,884455734	1,006200713
Ep 8	0,69896215	0,94465126	0,988264845



Osservando l'andamento dei tempi di elaborazione, si può notare che con esigue quantità di dati non si riscontra alcun sensibile decremento di tempo aumentando il numero di processori, e questo vuol dire che, in tal caso, *il parallelismo è poco utilizzato*, come dimostrano anche i valori di speed up e di efficienza.

A partire da 8 processori infatti, osservando i relativi grafici si può notare che i valori cominciano a discostarsi dalle soglie ideali (si veda al riguardo, l'introduzione della presente sezione).

Incrementando sensibilmente le dimensioni dell'input, si può invece osservare che la curva dei tempi decresce drasticamente se si aumenta il numero di processori, e i valori di speed-up e di efficienza tendono a stabilizzarsi attorno alle soglie ideali, dimostrando che, in questo caso, il parallelismo trova pieno utilizzo.

Si può dunque concludere esprimendo un giudizio positivo se l'algoritmo lavora con grosse quantità di dati.

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

```
/*
 ************************************************
 *    ALGORITMO PER IL CALCOLO PARALLELO
 *     DEL PRODOTTO MATRICE PER VETTORE
 *               III  STRATEGIA
 *
 *           VIRGINIA    BELLINO
 *               MATR. 108/1570
 ************************************************
*/


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

/*INIZIO FUNZIONE MAIN*/
int main(int argc, char *argv[])
{


    /*dichiarazione di variabili*/
    int menum, nproc;
    int n_righe, n_colonne;//numero di righe e numero di colonne della matrice

    int offset_r=0, offset_c=0, dim_prod_par;
    int N, Righe_Blocco, Colonne_Blocco, resto, i, j, k, tag;

    int p, q;

    int flag=0; //flag di controllo

    float *A, *x, *y, *sub_x, *sub_y, *prod_par, *Blocco;

    /*variabili usate per rilevare le prestazioni dell'algoritmo*/
    double t_inizio, t_fine, T1, Tp=0.F, speedup, Ep;

    MPI_Status info;

    /*comunicatori di griglia*/
    MPI_Comm griglia, grigliar, grigliac;

    int coordinate[2];
    int dim_Blocco[2];

    dim_Blocco[0]=0;
    dim_Blocco[1]=0;

    /*Inizializzazione dell'ambiente MPI*/
    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &menum);
```

```
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);




    if(menum==0)//l'utente è posizionato nel processore PO
    {
        printf("\nAlgoritmo per il calcolo del prodotto matrice vettore\n");
        printf("\nTerza Strategia \n");
        printf("\nProcessori utilizzati: %d\n", nproc);


        while(flag==0)
        {

            printf("\nInserire il numero di righe della matrice:");
            fflush(stdin);
            scanf("%d", &n_righe);

            printf("\nInserire il numero di colonne della matrice:");
            fflush(stdin);
            scanf("%d", &n_colonne);


            //controllo sulla correttezza dei dati inseriti
            if(n_righe*n_colonne<nproc)

               printf("ERRORE: Il numero di elementi risulta inferiore al numero di
processori!\n");

            else if(primo(nproc) && n_righe<nproc && n_colonne<nproc)

                 printf("ERRORE: Impossibile partizionare tale matrice per %d
processori\n", nproc);

                 else

                 flag=1;//uscita
        }//end while



        //VALUTAZIONE SE E' POSSIBILE APPLICARE LA PRIMA O LA SECONDA
        //STRATEGIA DI CALCOLO DEL PRODOTTO MAT X VET
        //Se il numero di processori è primo viene scelta una delle 2,
        //che risulta più conveniente
        if(primo(nproc))
        {
            printf("ATTENZIONE!! Il numero di processori e' primo, viene applicata la "
);
            if(n_righe>=nproc)
            {  //ci sono abbastanza righe per applicare la prima che è favorita
                printf("prima strategia\n");
                q=1;
                p=nproc;
```

```
            }
            else
            {
                //si consiglia la seconda strategia
                printf("seconda strategia\n");
                q=nproc;
                p=1;
            }

        }

        else

        /*....se il numero di processori non è primo.... */
        /*....si valuta se è possibile applicare la terza strategia*/
        {
            flag=0;
            while(flag==0)
            {
                printf("\nIndicare in quante parti si vuole dividere il numero di
righe:");
                fflush(stdin);
                scanf("%d", &p);
                q=nproc/p;  //q è calcolato di conseguenza

                /*Se ci sono errori ...*/
                if(n_righe<p || p>nproc || (p*q)%nproc!=0 || n_colonne<q)
                    printf("ERRORE: Impossibile partizionare i dati nel modo
indicato!\n");
                else
                    flag=1; /* esci*/
            }

        }



        printf("\n*****************************************************\n");
        printf(" Costruzione della griglia (%dx%d) di processori ", p, q);
        printf("\n*****************************************************\n");

        //Allocazione dinamica della matrice A e del vettore x in P0
        x=(float *)calloc(n_colonne, sizeof(float));
        A=(float *)calloc(n_righe*n_colonne, sizeof(float));

        printf("\n*************************************\n");
        printf("- Inserimento elementi della matrice A -\n");
        printf("\n*************************************\n");
        for(i=0;i<n_righe;i++)
        {
            for(j=0;j<n_colonne;j++)
            {
                //inserimento manuale
                //printf("Inserire elemento A[%d][%d]: ", i, j);
                //scanf("%f", A+i*n_colonne+j);
```

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


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

if (n_righe<100 && n_colonne<100)
{

                  printf("\nMatrice A \n:");

                  for(i=0;i<n_righe;i++)

                  {

                    printf("\n");

                    for(j=0;j<n_colonne;j++)
                    printf("%.2f\t", *(A+i*n_colonne+j));


                  }


}

else

{

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


}


printf("\n\n*********************************\n");
printf("- Inserimento elementi del vettore x -");
printf("\n*********************************\n");
for(i=0;i<n_colonne;i++)
{
      //inserimento manuale
      //printf("Inserire elemento x[%d]: ", i);
      //scanf("%f", x+i);

      //per valutare la prestazioni si attiva la riga seguente...
      //..che genera random il vettore x

      *(x+i)=(float)rand()/((float)RAND_MAX+(float)1);
}
```

```
    //stampa di x solo se ha dimensioni minori di 100

    if ( n_colonne<100 )
    {

                    printf("\nVettore x:\n");
                    for(i=0;i<n_colonne;i++)
                    printf("%.2f\t", *(x+i));

                    printf("\n");




    }

    else

    {

        printf("\nIl vettore è di grandi dimensioni \n");
        printf("\n e non verrà stampato \n");


    }



}/* end if iniziale relativo al processore PO*/


//Il processore PO esegue un Broadcast per comunicare agli altri processori
//del comunicatore il numero di righe e di colonne della matrice

MPI_Bcast(&n_righe, 1, MPI_INT, 0, MPI_COMM_WORLD);

MPI_Bcast(&n_colonne, 1, MPI_INT, 0, MPI_COMM_WORLD);


//Tutti allocano il vettore y
y=(float *)calloc(n_righe, sizeof(float));


N=n_righe*n_colonne;

//Il processore PO esegue un Broadcast per comunicare agli altri processori
//del comunicatore le dimensioni che avra la topologia a griglia
MPI_Bcast(&p, 1, MPI_INT, 0, MPI_COMM_WORLD);

MPI_Bcast(&q, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

```
//chiamata della funzione che costruirà la topologia dei processori
//a griglia bidimensionale
crea_griglia(&griglia, &grigliar, &grigliac, menum, p, q, coordinate);



Righe_Blocco=n_righe/p; //Calcola il numero di righe di un blocco
resto=n_righe%p;
if(coordinate[0]<resto) //Nel caso in cui n_righe non sia divisibile per p
     Righe_Blocco++; //Alcuni blocchi avranno una riga in più



Colonne_Blocco=n_colonne/q; //Calcola il numero di colonne di un blocco
resto=n_colonne%q;
if(coordinate[1]<resto) //Nel caso in cui n_colonne non sia divisibile per q
     Colonne_Blocco++; //Alcuni blcchi avranno una colonna in più



//Si memorizzano le dimensioni per spedirle a P0
dim_Blocco[0]=Righe_Blocco;
dim_Blocco[1]=Colonne_Blocco;

//Allocazione dinamica del blocco di righe e colonne su tutti i processori
Blocco=(float *)malloc(Righe_Blocco*Colonne_Blocco*sizeof(float));

//Allocazione dei sottovettori di x e y
sub_x=(float *)malloc(Colonne_Blocco*sizeof(float));
sub_y=(float *)calloc(Righe_Blocco, sizeof(float));
prod_par=(float *)calloc(Righe_Blocco, sizeof(float));



/*DISTRIBUZIONE DATI da parte di P0 ai vari processori*/
if(menum==0)
{
     //for di spedizione dati a tutti i processori
     for(i=1;i<nproc;i++)
     {
          //P0 individua il blocco da spedire calcolando lo scostamento su
          //righe e colonne
          offset_r=offset_r+dim_Blocco[1];
          if(offset_r>=n_colonne)
               offset_c=offset_c+n_colonne*dim_Blocco[0]; //Scostamento sulle colonne
               offset_r=offset_r%n_colonne; //Scostamento sulle righe


          tag=10+i;

          //Attende di conoscere le dimensioni del blocco del processore a
          //cui spedire
          MPI_Recv(dim_Blocco, 2, MPI_INT, i, tag, MPI_COMM_WORLD, &info);
```

```
            //Ricevute le informazioni spedisce gli elementi uno alla volta
            for(j=0;j<dim_Blocco[0];j++)
            {
                for(k=0;k<dim_Blocco[1];k++)
                {
                    tag=20+i;
                    MPI_Send(A+(j+1)*offset_r+offset_c+(n_colonne-offset_r)*j+k, 1,
                             MPI_FLOAT, i,tag, MPI_COMM_WORLD);
                }
            }
            //Spedisce anche gli elementi di x
            tag=30+i;
            MPI_Send(x+offset_r, dim_Blocco[1], MPI_FLOAT, i, tag, MPI_COMM_WORLD);


        }//end for di spedizione dati


        //P0 inizializza il suo blocco
        for(j=0;j<Righe_Blocco;j++){
            for(k=0;k<Colonne_Blocco;k++)
                *(Blocco+Colonne_Blocco*j+k) = *(A+n_colonne*j+k);
        }
        //P0 inizializza il suo sottovettore di x
        for(j=0;j<Colonne_Blocco;j++)
            *(sub_x+j)=*(x+j);


    }


    else


    {
        //Tutti i processori spediscono a P0 le dimensioni del proprio blocco
        tag=10+menum;
        MPI_Send(dim_Blocco, 2, MPI_INT, 0, tag, MPI_COMM_WORLD);
        //E ricevono gli elementi di tale blocco da P0
        for(j=0;j<Righe_Blocco;j++)
        {
            for(k=0;k<Colonne_Blocco;k++)
            {
                tag=20+menum;
                MPI_Recv(Blocco+Colonne_Blocco*j+k, 1, MPI_FLOAT, 0, tag,
MPI_COMM_WORLD, &info);
            }
        }
        //Ricevono anche il sottovettore x
        tag=30+menum;
        MPI_Recv(sub_x, Colonne_Blocco, MPI_FLOAT, 0, tag, MPI_COMM_WORLD, &info);

    }

    /*FINE DISTRIBUZIONE DEI DATI*/
```

```
/*@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@*/
/* @@@@@@   INIZIO FASE DI CALCOLO   @@@@@@*/
/*@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@*/

//Calcola il prodotto Blocco*sub_x=sub_y e il tempo
//per eseguirlo con p processori
Tp=mat_vet(Blocco, sub_x, Righe_Blocco, Colonne_Blocco, sub_y);

t_inizio=MPI_Wtime(); //calcolo del tempo di inizio

//Con la routine MPI si sommano i vettori parziali sulle righe della griglia
MPI_Allreduce(sub_y, prod_par, Righe_Blocco, MPI_FLOAT, MPI_SUM, grigliar);

t_fine=MPI_Wtime(); // calcolo del tempo di fine

Tp=Tp+t_fine-t_inizio; //Rileva il tempo della Allreduce




if(menum==0)
{   //Il processore P0 riceve tutti i risultati parziali sub_y
    //dai processori che nella griglia sono sulla stessa colonna

    for(i=0;i<Righe_Blocco;i++) //Concatena prima il suo prodotto parziale in y
        *(y+i)=*(prod_par+i);

    for(j=q;j<p*q;j=j+q) //Poi attende i prodotti parziali dei restanti processori
    {
        tag=40+j;
        MPI_Recv(&dim_prod_par, 1, MPI_INT, j, tag, MPI_COMM_WORLD, &info);
        //Ricevendoli li colloca nella posizione giusta in y
        for(k=0;k<dim_prod_par;k++)
        {
            tag=50+j;
            MPI_Recv(y+i, 1, MPI_FLOAT, j, tag, MPI_COMM_WORLD, &info);
            i++;
        }
    }
}


//I processori in griglia sulla stessa colonna di P0 spediscono
//il prodotto parziale
if(coordinate[1]==0&&coordinate[0]!=0)
{
    tag=40+menum;
    MPI_Send(&Righe_Blocco, 1, MPI_INT, 0, tag, MPI_COMM_WORLD);

    for(k=0;k<Righe_Blocco;k++)
```

```
        {
            tag=50+menum;
            MPI_Send(prod_par+k, 1, MPI_FLOAT, 0, tag, MPI_COMM_WORLD);
        }
    }

    /*@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@*/
    /* @@@@@@   FINE FASE DI CALCOLO   @@@@@@*/
    /*@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@*/




    if(menum==0)
    {

        printf("\n- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -\n");
        printf("- Stampa del risultato e dei parametri di valutazione -\n");
        printf("- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -\n");


        if ( n_righe<100 )
        {

                        printf("Vettore risultato:\n");
                        for(k=0;k<n_righe;k++)
                        printf("%.2f\t", *(y+k));

        }

        else

        {

            printf("\nIl vettore risultato è di grandi dimensioni \n");
            printf("\n e non verrà stampato \n");


        }




        //Calcolo del prodotto e del tempo con singolo processore
        T1=mat_vet(A, x, n_righe, n_colonne, y);

        speedup=T1/Tp;  //Calcola lo speed up

        Ep=speedup/nproc;  //Calcola l'efficienza

        printf("\n\nIl tempo di esecuzione usando 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 con %d processori e' %f secondi\n\n",nproc, Ep);
       }
```

```
    MPI_Finalize();
    return(0);
}
/*FINE FUNZIONE MAIN*/
```

```
/*
 ************************************************
 *    ALGORITMO PER IL CALCOLO PARALLELO
 *     DEL PRODOTTO MATRICE PER VETTORE
 *               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*/
void crea_griglia(MPI_Comm *, MPI_Comm *, MPI_Comm *, int, int, int, int *);

double mat_vet(float *, float *, int, int, float *);

int primo(int);


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

/*FUNZIONE primo*/
/*ritorna 1 se il parametro in input è un numero primo*/
int primo(int N)
{
    int primo=1;
    int i=2;

    if(N==1)
        primo=0;
    else
    {
        while(primo==1 && i<=sqrt(N))  //La complessità è O(sqrt(N))
        {

            //Se viene trovato un solo divisore di N il ciclo si arresta
            if(N%i==0)
            {
                primo=0;
                break;
            }
            i++;
        }
```

```
    }
    return primo;
}



/*FUNZIONE crea_griglia*/
//crea una griglia bidimensionale non 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 righe, int colonne, int*coordinate)
{

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

    //Specifica il numero di processori in ogni dimensione
    ndim=(int*)calloc(dim, sizeof(int));
    ndim[0]=righe;
    ndim[1]=colonne;

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

    MPI_Cart_create(MPI_COMM_WORLD, dim, ndim, period, reorder, 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 VETTORE
double mat_vet(float *A, float *B, int m, int n, float *C)
```

```
{
    double tempo=0.F, t_inizio, t_fine;
    int i, j;
    for(i=0;i<m;i++)
    {
        *(C+i)=0.F;  //Per più attivazioni della function, azzeramento indispensabile
        for(j=0;j<n;j++)
        {
            //rilevazione tempo di inizio
            t_inizio=MPI_Wtime();

            *(C+i)=*(C+i)+*(A+i*n+j)*(*(B+j));

            //rilevazione tempo di fine
            t_fine=MPI_Wtime();

            //Si accumula l'incremento ad ogni ciclo
            tempo=tempo+t_fine-t_inizio;
        }
    }
    return tempo;  //Ritorna il tempo impiegato per il calcolo


}
```