

GIOVANNI DI CECCA

Calcolo

Parallelo e
Distribuito

Matrice

per Vettore

II

Strategia



dicecca.net
web site

© 2009 – Giovanni Di Cecca

© 2021 – MONITORE NAPOLETANO – www.monitorenapoletano.it

Direttore Responsabile: Giovanni Di Cecca

Collana **dicecca.net** – **Computer Science**

Anno II - № 19 – Supplemento al Numero 161 – Luglio 2021

Periodico Mensile Registrato presso il Tribunale di Napoli № 45 dell'8 giugno 2011

ISSN: 2239-7035

Indice

Definizione ed analisi del Problema.....	5
Descrizione dell’algoritmo.....	7
Introduzione.....	7
Inizializzazione dell’ambiente di calcolo.....	8
Inserimento dei dati.....	9
Distribuzione dei dati.....	11
Calcolo del prodotto matriciale parziale e totale.....	12
Compilazione ed esecuzione.....	15
Compilazione sotto Linux.....	15
Esecuzione del programma sotto Linux.....	15
Compilazione sotto Microsoft Windows XP.....	16
Esecuzione del programma sotto Microsoft Windows XP.....	16
Indicatori di errore.....	17
Funzioni utilizzate.....	19
Esempi d’uso.....	25
Analisi dei tempi.....	29
Introduzione.....	31
Tabella dei tempi di esecuzione.....	31
Tabella di Speed – Up.....	34
Tabella dell’Efficienza.....	37
Commenti ai tempi.....	39
Bibliografia.....	45
Codice sorgente.....	47

Definizione ed analisi del problema

Scopo: il software che si analizzerà di seguito ha lo scopo di effettuare il prodotto Matrice per Vettore usando un'architettura di tipo MIMD distribuendo il calcolo a $p \times q$ processi disposto secondo una griglia a topologia bidimensionale.

Una volta stabilito il numero di righe e di colonne che deve avere la nostra matrice, il programma genera dei numeri casuali che riempiono la matrice ed il vettore.

A seconda del numero di processori impiegato è possibile spezzare la matrice e redistribuire i blocchi di calcolo ai vari processori che effettuano i calcoli parziali ed infine totali.

Il programma contiene al suo interno le routines per calcolare lo speedup e l'efficienza dell'algoritmo

Per risolvere il problema, è stata usata l'infrastruttura del Message Passing Interface.

Descrizione dell'Algoritmo

Introduzione

La strategia usata per risolvere il calcolo del prodotto Matrice Vettore è quella di distribuire una matrice $A \in \mathfrak{R}^{m \times n}$ scorporandola in $p \times q$ processi su di una topologia di griglia bidimensionale.

Vediamo ora nel dettaglio le varie parti in gioco dell'algoritmo.

Per meglio gestire il problema il programma è stato scisso in due file: uno che contiene le routines del calcolo in senso stretto, ed un altro di funzioni ausiliarie

- 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

Descrizione dell'Algoritmo

L'algoritmo può essere suddiviso in cinque parti principali:

- Inizializzazione dell'ambiente di calcolo
- Inserimento dei dati
- Distribuzione dei dati
- Calcolo del prodotto mat vet parziale e totale
- Calcolo dei tempi

Per poter meglio analizzare le performances dell'algoritmo al suo interno è stato inserito il sistema di controllo del tempo.

Analizzeremo nel dettaglio le parti indicate

- Inizializzazione dell'ambiente di calcolo

```
// Dichiarazione di variabili
int menum, nproc;

//numero di righe e numero di colonne della matrice
int n_righe, n_colonne;

// offset_r=0 et offset_c=0 determinazione del blocco delle righe e delle
colonne da spedire
int offset_r=0, offset_c=0, dim_prod_par;

int N, Righe_Blocco, Colonne_Blocco, resto, i, j, k, tag;

// p=processori che si intendono usare per calcolare il prodotto righe x colonne
// q=il numero di righe x colonne calcolato dai processori p inseriti
precedentemente
int p, q;

int falg=0; //flag di controllo

float *A, *x, *y, *sub_x, *sub_y, *prod_par, *Blocco; // variabili puntatore
// *A = matrice
// *x = vettore
// *y = numero di colonne allocate dai processori
// *sub_x et *sub_y = sotto vettori x ed y
// *prod_par = prodotto parziale
// *Blocco = allocazione del blocco dati su tutti i processori

//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);
```

- **Inserimento dei dati**

Il processore P0, dopo aver inizializzato il vettore dinamicamente ed effettuato il broadcast alla rete viene effettuato l'inserimento dei dati con il metodo di generazione casuale Random, mediante la seguente funzione:

```
// Procedura di inserimento dati
while(falg==0)
{
    printf("\nInserire il numero di righe della matrice:");
    fflush(stdin); // pulisce lo standard input
    scanf("%d", &n_righe);

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

    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
        falg=1; // esce dal ciclo while
```

```

} //end while

...

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++)
    {
        // Generazione automatica dei valori di A
        *(A+i*n_colonne+j)=(float)rand()/((float)RAND_MAX+(float)1);
    }
}

printf("\n*****\n");
printf("Inserimento elementi della matrice x");
printf("\n*****\n");
for(i=0;i<n_colonne;i++)
{
    // Generazione automatica dei valori del vettore x
    *(x+i)=(float)rand()/((float)RAND_MAX+(float)1);
}

```

È possibile inserire i dati manualmente per eventuali operazioni di calcolo con dati reali, sostituendo le due funzioni in grassetto con le seguenti

```

// dati matrice
printf("Digita elemento A[%d][%d]: ", i, j);
scanf("%f", A+i*num_c+j);

// dati vettore
printf("Digita elemento x[%d]: ", i);
scanf("%f", x+i);

```

- Distribuzione dei dati

Il processore P0 invia i dati ai vari processori della rete:

```
//Il processore P0 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);

// Carica la funzione crea_griglia (dal file mxv-aux.c) per creare la griglia
in cui ripartire i dati
crea_griglia(&griglia, &grigliar, &grigliac, menum, p, q, coordinate);
```

crea_griglia è una funzione che fa parte delle routines ausiliarie

```
//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, dim=2; //Le dimensioni della griglia sono 2
    int vc[2];

    ndim=(int*)calloc(dim, sizeof(int)); //Specifica il numero di processori
in ogni dimensione
    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);

vc[0]=0;
vc[1]=1;

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

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

```

- **Calcolo delle prodotto matriciale parziale e totale**

A questo punto i dati sono stati inseriti e spediti.

Ora attendiamo la ricezione da parte dei processori

```

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

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

//Poi attende i prodotti parziali dei restanti processori
for (j=q; j<p*q; j=j+q)
{
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++;
        }
    }
} // end if

//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);
    }
}

```

Da questo punto in poi si tratta solo di visualizzare la nuova matrice con i dati calcolati

```

if(menum==0)
{
    printf("\n- - - - -\n");
    printf("- Stampa del risultato e dei parametri di valutazione -\n");
    printf("- - - - -\n");

    if (n_righe<100 || n_colonne<100)
    {
        printf("Vettore risultato:\n");
        for(k=0;k<n_righe;k++)
            printf("%.2f\t", *(y+k));
    }
}

```

```
else
    printf("\n\n Risultato troppo grande per essere stampato");
```

Per ragioni di test, ho optato per la non visualizzazione dei dati, in quanto il tempo di trasmissione per valori notevoli sarebbe stato particolarmente oneroso.

Nell'algoritmo, inoltre, c'è una subroutine che consente di calcolare i tempi di Efficienza e SpeedUp, che si trova nella libreria di funzioni ausiliarie:

```
//Calcolo del prodotto con singolo processore e tempo
T1=mat_vet(A, x, n_righe, n_colonne, y);
speedup=T1/Tp; //Calcola lo speed up
Ep=speedup/nproc; //Calcola l'efficienza
```

Compilazione ed esecuzione

Il programma è studiato per essere il più possibile indipendente dalla piattaforma usata (Linux o Windows).

- Compilazione sotto Linux

Se il programma viene compilato in un sistema Linux, la sintassi è la seguente

```
$ mpicc mxv.c <invio>
```

Successivamente sarà generato un file a eseguibile

DEVE ESSERE PERESEENTE NELLA CARTELLA DOVE SI COMPILA ANCHE IL FILE **MXV-AUX.C**

- Esecuzione del programma sotto Linux

Per eseguire il programma si deve seguire la seguente sintassi:

```
$ mpirun -np x -machinefile lista a.out <invio>
```

con **x** il numero di processori che si vuole usare.

lista è l'elenco di tutti processori della rete che coinvolgiamo nel calcolo

A questo punto il programma in esecuzione chiede quanti dati devono essere calcolati.

- Compilazione sotto Microsoft Windows XP

Se il programma viene compilato in Windows la sintassi è la seguente

Aprire il Prompt dei comandi¹:

```
C:\>gcc mxv.c -lmpich
```

Successivamente sarà generato un file **a.exe**

- Esecuzione del programma sotto Microsoft Windows²

Per eseguire il programma si deve seguire la seguente sintassi:

```
C:\> mpirun -np x a <invio>
```

con *x* il numero di processori che si vuole usare.

A questo punto il programma in esecuzione chiede quanti dati devono essere calcolati.

¹ Si presuppone che sia stato installato il compilatore MingW, le librerie MPICH NT e le **Variabili d'ambiente** contengano il path del compilatore e delle librerie

² Si presuppone che il sistema abbia identificato la presenza di 1 o più computer e che sia stato configurato il programma **MPICH Configuration tool**.

Indicatori di errore

La routine **primo** (nella libreria delle funzioni ausiliarie) calcola se il numero dei processori che si vanno ad usare è un numero primo o meno.

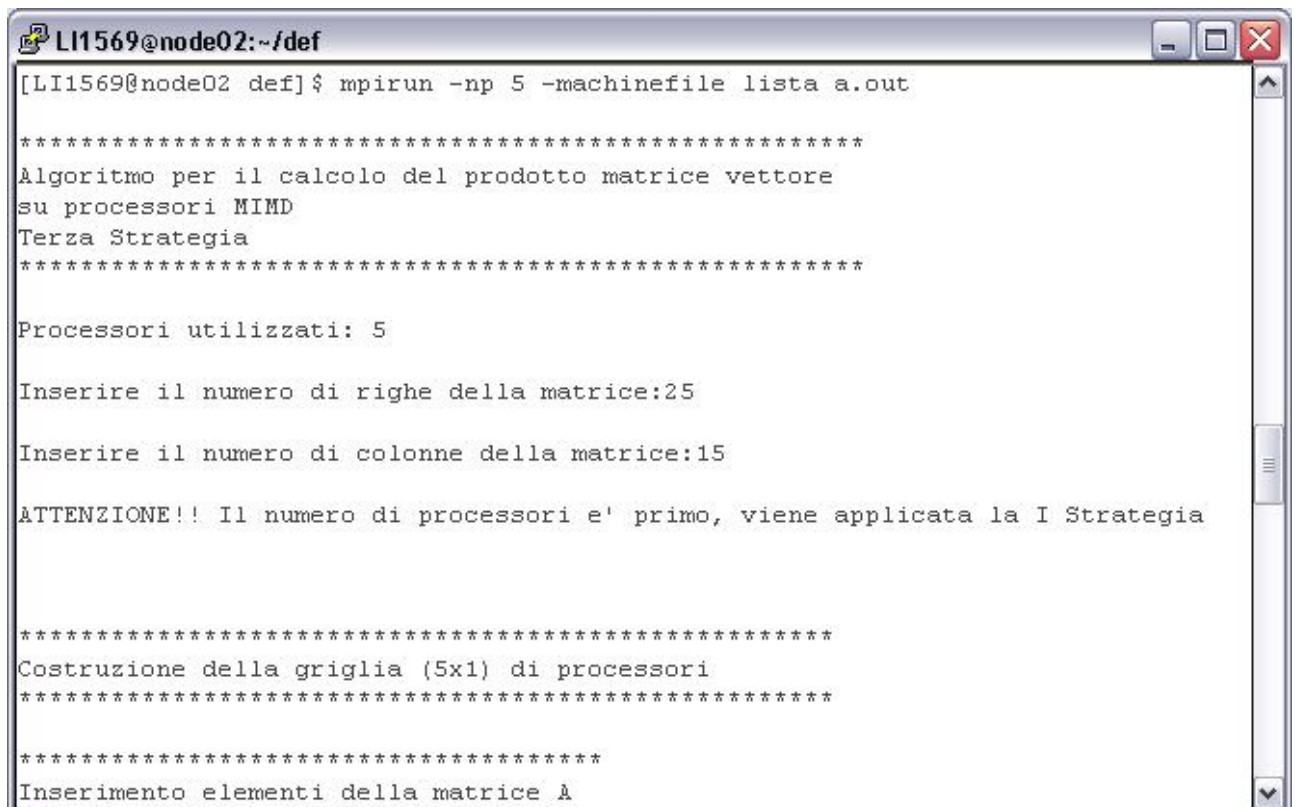
Nel caso si utilizzi un numero di processori pari a 1, 2, 3, 5, 7, 11,... (nel test reale riportato di seguito il numero massimo di processori è 8) il programma applica o la I o la II strategia.

La III strategia può essere applicata solo se il numero di processori non è primo, poiché si deve ripartire il numero di righe per i processori.

Inoltre come detto nell'analisi dell'algoritmo, per ridurre al massimo il tempo di attesa, sono stati evitati di inserire i dati e visualizzare la matrice.

Di seguito sono riportati alcuni screenshot di errori del programma:

- Applicazione della I Strategia



```
LI1569@node02:~/def
[LI1569@node02 def]$ mpirun -np 5 -machinefile lista a.out
*****
Algoritmo per il calcolo del prodotto matrice vettore
su processori MIMD
Terza Strategia
*****

Processori utilizzati: 5

Inserire il numero di righe della matrice:25

Inserire il numero di colonne della matrice:15

ATTENZIONE!! Il numero di processori e' primo, viene applicata la I Strategia

*****
Costruzione della griglia (5x1) di processori
*****

*****
Inserimento elementi della matrice A
```

- Applicazione della II Strategia

```

LI1569@node02:~/def
*****
Algoritmo per il calcolo del prodotto matrice vettore
su processori MIMD
Terza Strategia
*****

Processori utilizzati: 7

Inserire il numero di righe della matrice:6

Inserire il numero di colonne della matrice:8

ATTENZIONE!! Il numero di processori e' primo, viene applicata la II Strategia

*****
Costruzione della griglia (1x7) di processori
*****

*****
Inserimento elementi della matrice A
*****

```

- Errore del Numero di righe inferiore ai processori

```

LI1569@node02:~/def
[LI1569@node02 def]$ mpirun -np 7 -machinefile lista a.out

*****
Algoritmo per il calcolo del prodotto matrice vettore
su processori MIMD
Terza Strategia
*****

Processori utilizzati: 7

Inserire il numero di righe della matrice:5

Inserire il numero di colonne della matrice:6

ERRORE: Impossibile partizionare tale matrice per 7 processori

Inserire il numero di righe della matrice:█

```

Si deve inserire, quindi, un numero di righe maggiori o uguali al numero di processori usati.

Funzioni Utilizzate

Il software descritto si compone di 2 files.

Di seguito, per ognuna di esse si fornisce il prototipo utilizzato nel programma e la descrizione dei relativi parametri di input e di output.

- Funzioni per inizializzare l'ambiente MPI

- **MPI_Init (&argc, &argv);**
argc et *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
l: 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

l (*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

l : 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 menu 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

La libreria di funzioni ausiliare comprende

// **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.

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

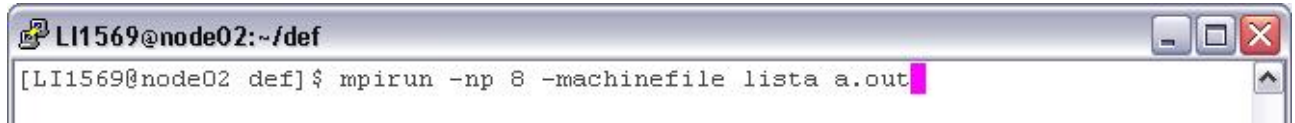
B: frazione del vettore di input

m,n : dimensioni del sottoblocco di matrice ricevuto

C:vettore risultato

Esempi d'uso

Esempio di lancio su 8 processori

A terminal window with a title bar that reads "LI1569@node02: ~/def". The window contains a single line of text: "[LI1569@node02 def]\$ mpirun -np 8 -machinefile lista a.out". The cursor is at the end of the command. The window has standard Linux window controls (minimize, maximize, close) and a scroll bar on the right side.

```
LI1569@node02: ~/def
[LI1569@node02 def]$ mpirun -np 8 -machinefile lista a.out
```

La seguente schermata mostra una esecuzione del programma su un numero eccezionalmente esiguo di valori, ma utili per mostrare come il programma visualizza i dati.

```
LI1569@node02:~/def

Inserire il numero di righe della matrice:4

Inserire il numero di colonne della matrice:4

Indicare in quante parti si vuole dividere il numero di righe:4

*****
Costruzione della griglia (4x1) di processori
*****

*****
Inserimento elementi della matrice A
*****

Matrice A
:
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
*****

Inserimento elementi della matrice x
*****

Vettore x:
0.64   0.72   0.14   0.61

Inizio l'elaborazione

-----
- Stampa del risultato e dei parametri di valutazione -
-----

Vettore risultato:
1.41   1.24   1.02   1.29

Il tempo di esecuzione su un processore e' 0.000347 secondi
Il tempo di esecuzione su 4 processori e' 0.000140 secondi
Lo Speed Up ottenuto e' 2.478571
L'efficienza risulta essere 0.619643
```

Di seguito, è riportato un altro esempio di come il sistema gestisce matrici di dimensione pari o superiore a 100 elementi:

```
LI1569@node02:~/def
Inserire il numero di colonne della matrice:200
Indicare in quante parti si vuole dividere il numero di righe:4

*****
Costruzione della griglia (4x2) di processori
*****

*****
Inserimento elementi della matrice A
*****

Matrice troppo grande per essere stampata

*****
Inserimento elementi della matrice x
*****

Vettore troppo grande per essere stampato

Inizio l'elaborazione

-----
- Stampa del risultato e dei parametri di valutazione -
-----

Risultato troppo grande per essere stampato

Il tempo di esecuzione su un processore e' 0.502640 secondi
Il tempo di esecuzione su 8 processori e' 0.089786 secondi
Lo Speed Up ottenuto e' 5.598200
L'efficienza risulta essere 0.699775
```

Esempio con esecuzione su un numero di processori primo

```
LI1569@node02:~/def
Inserire il numero di colonne della matrice:120
ATTENZIONE!! Il numero di processori e' primo, viene applicata la I Strategia

*****
Costruzione della griglia (2x1) di processori
*****

*****
Inserimento elementi della matrice A
*****

Matrice troppo grande per essere stampata

*****
Inserimento elementi della matrice x
*****

Vettore troppo grande per essere stampato

Inizio l'elaborazione

-----
- Stampa del risultato e dei parametri di valutazione -
-----

Risultato troppo grande per essere stampato

Il tempo di esecuzione su un processore e' 0.301727 secondi
Il tempo di esecuzione su 2 processori e' 0.150732 secondi
Lo Speed Up ottenuto e' 2.001745
L'efficienza risulta essere 1.000872
```

Analisi dei tempi

Introduzione

Ora osserveremo le prestazioni che ha il nostro algoritmo quando viene usato. Per fare ciò analizzeremo le seguenti caratteristiche:

- *Tempo di esecuzione utilizzando un numero $p > 1$ 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

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

Il valore dello speedup 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.....

$$E_p = \frac{S_P}{p}$$

Essa fornisce *un'indicazione di quanto sia stato usato il parallelismo nel calcolatore.*

Idealmente, dovremmo avere che:

$$E_p = 1$$

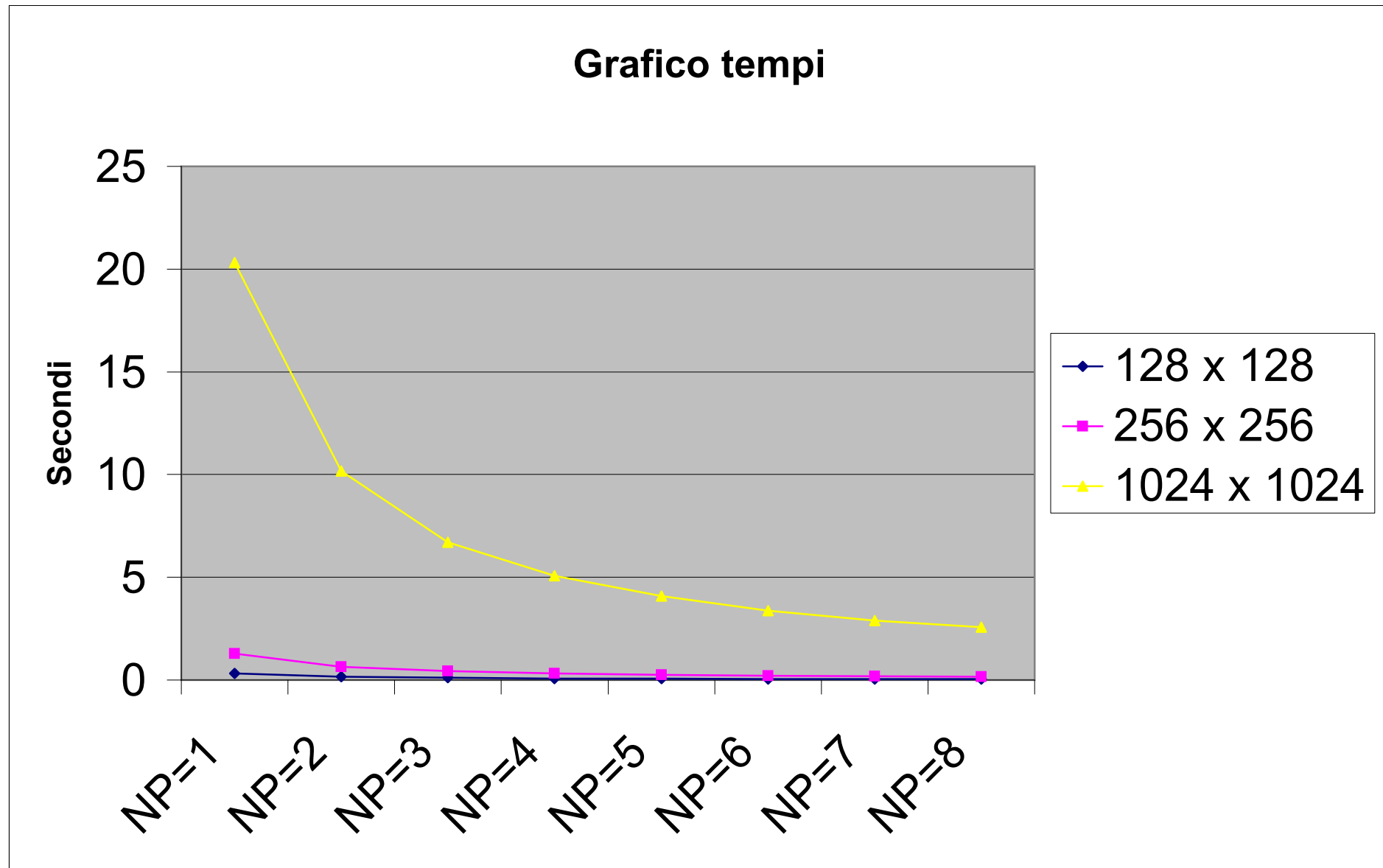
e quindi l'algoritmo parallelo risulta migliore quanto più E_p è vicina ad 1.

Nelle prossime pagine analizzeremo i tempi delle elaborazioni, presi direttamente dal calcolo, ed in seguito analizzeremo lo speed – up e l'efficienza.

Di seguito sono riportati i dati delle elaborazioni effettuate con il programma con test su valori significativi con matrici di dimensione quadrata pari a 2^7 (16.384 valori), 2^8 (65.536 valori), 2^{10} (1.048.576 valori).

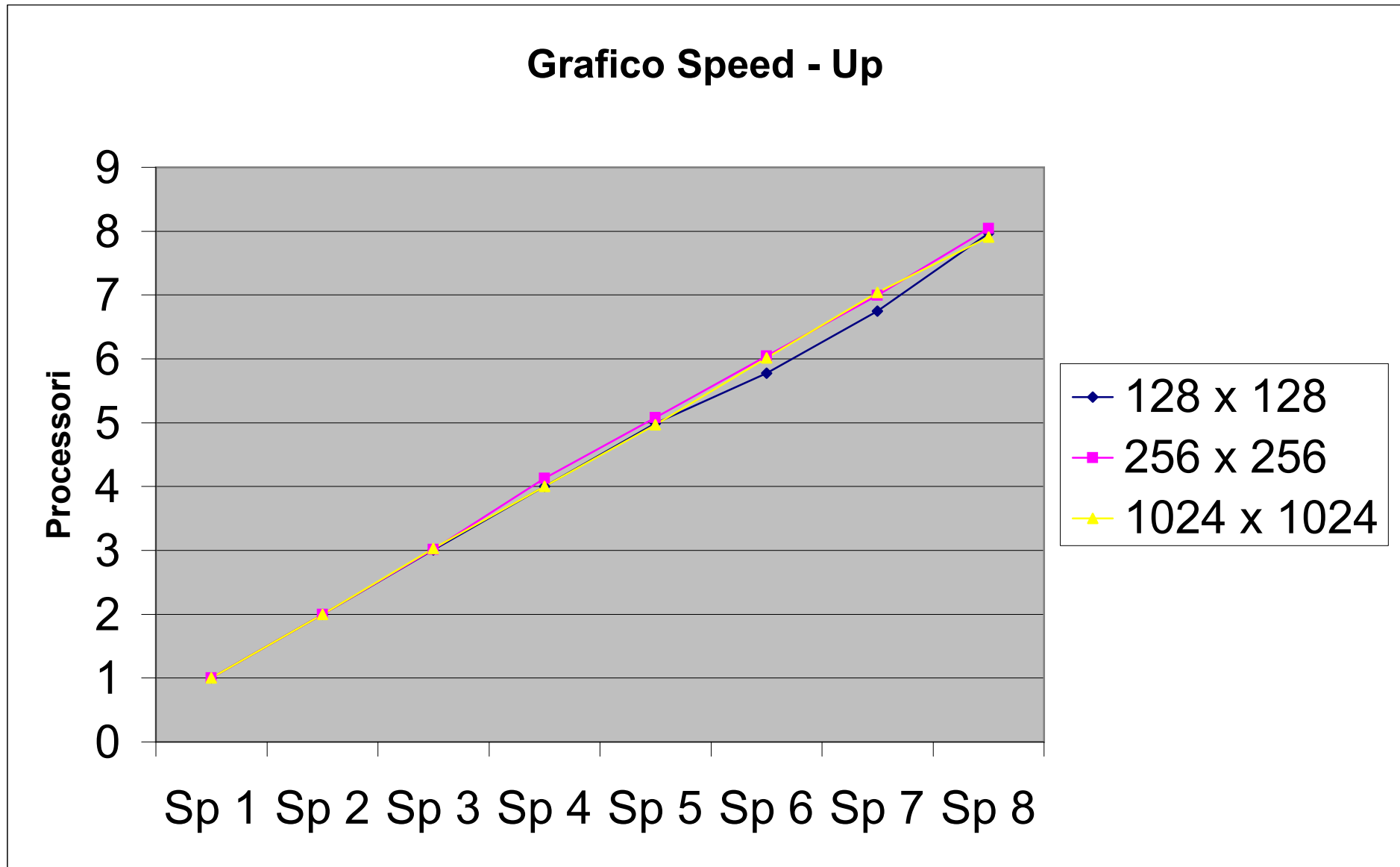
Tempi delle elaborazioni

	128 x 128	256 x 256	1024 x 1024
NP=1	0,320008	1,293631	20,308637
NP=2	0,160344	0,647796	10,161322
NP=3	0,106318	0,429488	6,6963610
NP=4	0,079991	0,313194	5,0707320
NP=5	0,064122	0,254830	4,0862700
NP=6	0,055396	0,214021	3,3763450
NP=7	0,047403	0,184979	2,8833490
NP=8	0,04016	0,160845	2,5687240



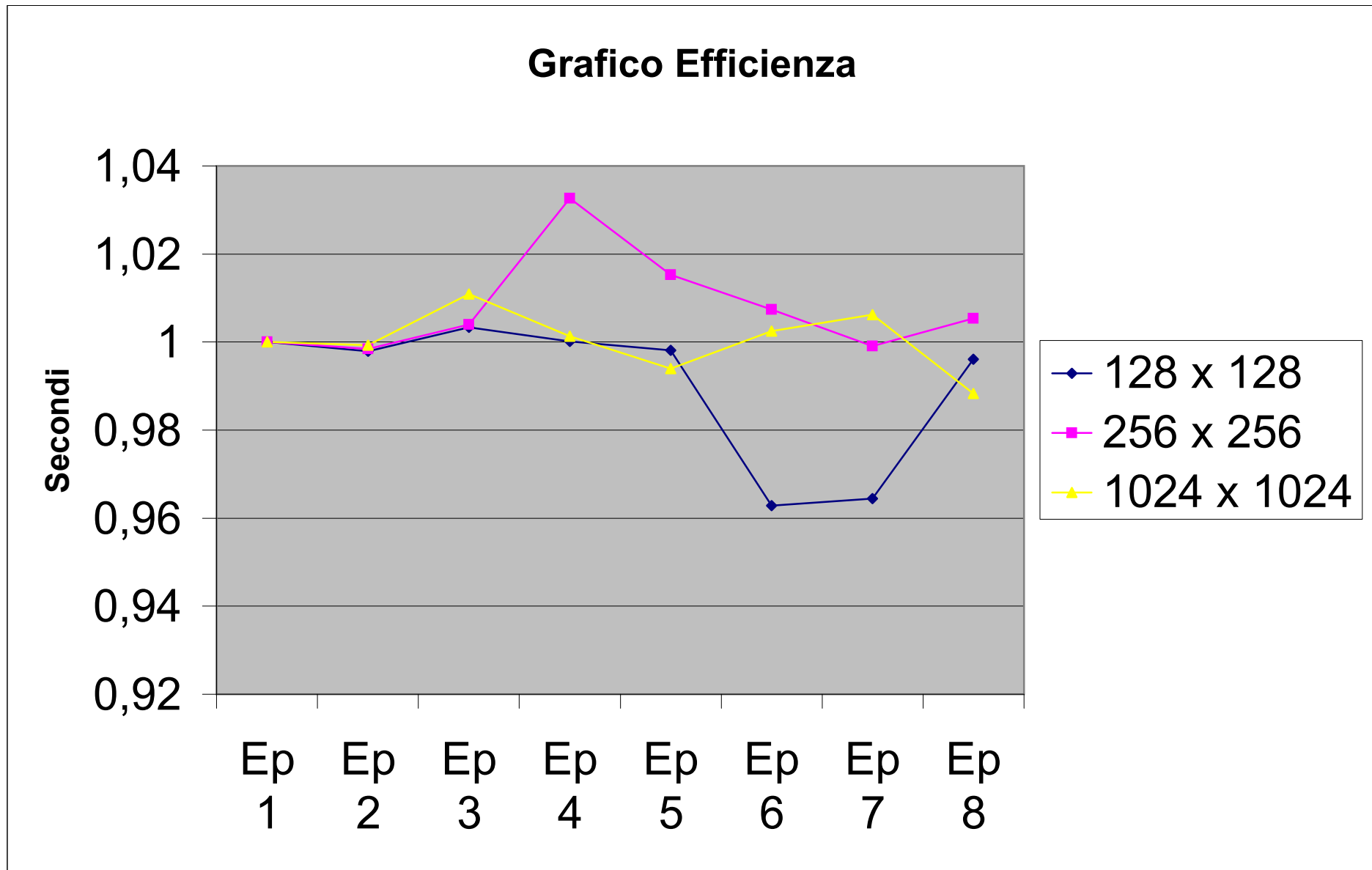
Speed - Up

	128 x 128	256 x 256	1024 x 1024
Sp 1	1	1	1
Sp 2	1,995759118	1,996972812	1,998621538
Sp 3	3,009913655	3,012030604	3,032787062
Sp 4	4,000550062	4,130446305	4,005070077
Sp 5	4,990611647	5,076447043	4,969969434
Sp 6	5,776734782	6,044411530	6,014976846
Sp 7	6,750796363	6,993393845	7,043419648
Sp 8	7,968326693	8,042718145	7,906118758



Efficienza

	128 x 128	256 x 256	1024 x 1024
Ep 1	1	1	1
Ep 2	0,997879559	0,998486406	0,999310769
Ep 3	1,003304552	1,004010201	1,010929021
Ep 4	1,000137515	1,032611576	1,001267519
Ep 5	0,998122329	1,015289409	0,993993887
Ep 6	0,96278913	1,007401922	1,002496141
Ep 7	0,96439948	0,999056264	1,006202807
Ep 8	0,996040837	1,005339768	0,988264845



Come è possibile osservare dai test effettuati (pag. 31) per questo tipo di problema la parallellizzazione porta degli oggettivi incrementi di prestazioni nell'ambito dell'efficienza delle prestazioni.

Le curve discendenti di pag. 33 mostrano, soprattutto con valori elevati un miglioramento effettivo delle prestazioni.

Per valori pari a 1.048.576 elementi floating point si passa dai 20,308637 secondi di un singolo processore ai 2,8833490 secondi di 7 processori, ai 2,5687240 secondi di 8 processori.

Ovviamente diminuendo il tempo aumenta lo speed-up (pagg. 33 e 35) e, l'efficienza, tende a quel valore ottimale di 1 (pagg. 39 e 41)

Bibliografia

1. A.Murli – Lezioni di Calcolo Parallelo – Ed. Liguori
2. Slide del corso di Calcolo Parallelo e distribuito
http://www.dma.unina.it/~murli/didattica/mat_didattico_nav_cp0708.html
3. www.mat.uniroma1.it/centro-calcolo/HPC/materiale-corso/sliMPI.pdf
(consultato per interessanti approfondimenti su MPI)
4. www.orebla.it/module.php?n=c_num_casuali
(consultato per approfondimenti sulla funzione rand per generare numeri casuali)
5. Aitken Peter G., Jones Bradley J. - Programmare in C – Apogeo
(consultato per ripasso di alcune funzioni)

Codice Sorgente

Di seguito viene proposto il codice sorgente del programma sin qui descritto, nella visualizzazione dell'editor di testo Notepad++ Portable per Windows

```
/*
*****
Algoritmo per il calcolo del prodotto matrice vettore,

Progetto Realizzato da

Giovanni Di Cecca

Web Site: http://www.dicecca.net

*****
*/

// Preprocessore
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include "mpi.h" // Carica la lib MPI

// Funzione Ausiliaria del programma
#include "mxv-aux.c"

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

    /*******
    // Inizio fase di inizializzazione delle variabili
    /*******

    // 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;

    // offset_r=0 et offset_c=0 determinazione del blocco delle righe e delle colonne da spedire

    int N, Righe_Blocco, Colonne_Blocco, resto, i, j, k, tag;

    int p, q; // p=processori che si intendono usare per calcolare il prodotto righe x colonne
            // q=il numero di righe x colonne calcolato dai processori p inseriti precedentemente

    int falg=0; //flag di controllo

    float *A, *x, *y, *sub_x, *sub_y, *prod_par, *Blocco; // variabili puntatore
    // *A = matrice
    // *x = vettore
    // *y = numero di colonne allocate dai processori
    // *sub_x et *sub_y = sotto vettori x ed y
```



```

// *prod_par = prodotto parziale
// *Blocco = allocazione del blocco dati su tutti i processori

//variabili usate per rilevare le prestazioni dell' algoritmo
double t_inizio, t_fine, Tl, 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);

//*****
// Inizio fase di immissione dei dati
//*****

if (menum==0)
{
    printf("\n*****\n");
    printf("Algoritmo per il calcolo del prodotto matrice vettore\n");
    printf("su processori MIMD\n");
    printf("Terza Strategia \n");
    printf("*****\n");

    printf("\nProcessori utilizzati: %d\n", nproc);

    // Procedura di inserimento dati
    while (falg==0)
    {

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

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

        if (n_righe*n_colonne<nproc)

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

```

```

else if(primo(nproc) && n_righe<nproc && n_colonne<nproc)
    printf("\n\nERRORE: Impossibile partizionare tale matrice per %d
processori\n", nproc);
else
    falg=1; // esce dal ciclo while
} //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("\nATTENZIONE!! Il numero di processori e' primo, viene applicata
la ");
    if(n_righe>=nproc)
    {
        //ci sono abbastanza righe per applicare la I Strategia
        printf("I Strategia\n");
        q=1;
        p=nproc;
    }
    else
    {
        printf("II Strategia\n");
        q=nproc;
        p=1;
    }
}

else

//....se il numero di processori non è primo....
//....si valuta se è possibile applicare la terza strategia
{

    falg=0; // reinizializza la variabile per il ciclo while

    while(falg==0)
    {
        // Questa procedura serve a partizionare il calcolo delle righe
        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\n Riprovare\n");

```

```

        else
            falg=1; // esci
    }

}

printf("\n\n\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");
printf("\n*****\n");
for(i=0;i<n_righe;i++)
{
    for(j=0;j<n_colonne;j++)
    {
        // Generazione automatica dei valori di A
        *(A+i*n_colonne+j)=(float)rand()/((float)RAND_MAX+(float)1);
    }
}

// Stampa della Matrice A solo se sono meno di 100 dati
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("\n\nMatrice troppo grande per essere stampata\n\n");

printf("\n*****\n");
printf("Inserimento elementi della matrice x");
printf("\n*****\n");
for(i=0;i<n_colonne;i++)
{
    // Generazione automatica dei valori del vettore x

```

```

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

    // Stampa del vettore x
    if (n_colonne<100)
    {
        printf("\nVettore x:\n");
        for(i=0;i<n_colonne;i++)
            printf("%.2f\t", *(x+i));
        printf("\n");
    }
    else
        printf("\n\n Vettore troppo grande per essere stampato\n\n");

    printf("\n\n Inizio l'elaborazione\n\n");

} // end if iniziale

//*****
// Inizio fase di distribuzione e spedizione dei dati
//*****

//Il processore P0 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;

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

// Carica la funzione crea_griglia (dal file mxv-aux.c) per creare la griglia in cui ripartire i dati
crea_griglia(&griglia, &grigliar, &grigliac, menun, p, q, coordinate);

//Calcola il numero di righe di un blocco
Righe_Blocco=n_righe/p;
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

```

```

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

//Si memorizzano le dimensioni per spedirle a PO
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 da parte di PO degli elementi ai vari processori
if (menum==0)
{
    //for di spedizione dati
    for (i=1; i<nproc; i++)
    {
        //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; //Lo scostamento sulle righe

        tag=10+i; // id di spedizione

        //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; // id di spedizione
        MPI_Send(x+offset_r, dim_Blocco[1], MPI_FLOAT, i, tag, MPI_COMM_WORLD);
    }
}

```

```

} //end for di spedizione dati

//PO 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);
}
//PO inizializza il suo sottovettore di x
for (j=0; j<Colonne_Blocco; j++)
    *(sub_x+j)=*(x+j);

}
// Altrimenti ricevi i dati
else
{
    //Tutti i processori spediscono a PO 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 PO
    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);

}

//*****
// Inizio fase di calcolo locale per valutazione dei tempi
//*****

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

t_inizio=MPI_Wtime(); //inizio del cronometro per il 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

//*****
// Fine fase di calcolo locale per valutazione dei tempi

```

```

//*****

//*****
// Inizio fase di ricezione dei risultati parziali
//*****

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);
    }
}

//*****
// Fase di stampa del risultato e dei tempi
//*****

if (menum==0)
{
    printf("\n- - - - -\n");
    printf("- Stampa del risultato e dei parametri di valutazione -\n");
    printf("- - - - -\n");

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

```

```
printf("Vettore risultato:\n");
    for(k=0;k<n_righe;k++)
        printf("%.2f\t", *(y+k));
}
else
    printf("\n\n Risultato troppo grande per essere stampato\n\n");

//Calcolo del prodotto con singolo processore e tempo
T1=mat_vet(A, x, n_righe, n_colonne, y);
speedup=T1/Tp; //Calcola lo speed up
Ep=speedup/nproc; //Calcola l'efficienza

// Stampa i tempi
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);
}

MPI_Finalize();
return(0);
}
```



```
/*
*****
Algoritmo per il calcolo del prodotto matrice vettore,

FUNZIONI AUSILIARIE AL PROGRAMMA

Progetto Realizzato da

Giovanni Di Cecca
Matr. 108/1569

E-Mail: giovanni.dicecca@gmail.com

Web Site: http://www.dicecca.net

*****
*/

//PROTOTIPI DELLE FUNZIONI UTILIZZATE
void crea_griglia(MPI_Comm *, MPI_Comm *, MPI_Comm *, int, int, int, int *);

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

int primo(int);

//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, dim=2; //Le dimensioni della griglia sono 2
    int vc[2];

    ndim=(int*) calloc (dim, sizeof(int)); //Specifica il numero di processori in ogni dimensione
    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);

    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++)
        {
            t_inizio=MPI_Wtime();
            *(C+i)=*(C+i)+*(A+i*n+j)*(*(B+j));
            t_fine=MPI_Wtime();
        }
    }
}

```

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

LIBERTÀ

EGUAGLIANZA

MONITORE NAPOLETANO

Fondato nel 1799 da
Carlo Lauberg ed Eleonora de Fonseca Pimentel

Rifondato nel 2010
Direttore: Giovanni Di Cecca

Anno CCXXII

Contatti



C.Ph.: +39 392 842 76 67



www.monitorenapoletano.it



info@monitorenapoletano.it