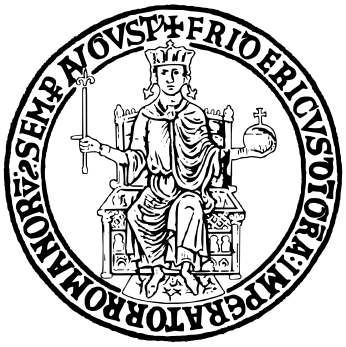
Università degli Studi di Napoli Federico II



Scuola Politecnica e delle Scienze di Base
Dipartimento di Ingegneria Elettrica e
Tecnologie dell'Informazione

Corso di Laurea Magistrale in Informatica

Parallel and Distributed Computing

Progettazione di un algoritmo per il calcolo del prodotto matrice-matrice in ambiente di calcolo parallelo su architettura MIMD a memoria distribuita

Docenti
Prof. Giuliano Laccetti
Prof.ssa Valeria Mele

Candidati
Marco Romano N97000395
Gianluca L'arco N97000393

Anno Accademico 2021 - 2022

Indice

Elenco delle figure

Elenco delle tabelle

1 Definizione e analisi del problema

Si vuole eseguire il prodotto righe per colonne tra due matrici $A \in \mathbb{R}^{n \times n}$ e $B \in \mathbb{R}^{n \times n}$ in parallelo su p processori. Il numero di processori e la dimensione n delle matrici vengono forniti in input mediante riga di comando, mentre i valori delle matrici vengono generati randomicamente all'interno del software.

Il prodotto tra matrici viene implementato mediante la strategia di comunicazione *Broadcast Multiply Rolling* (BMR), la quale si basa sulla suddivisione delle matrici A e B in blocchi di righe e blocchi di colonne, assegnando le due sottomatrici quadrate (quella di A e quella di B) ad ogni processore. Il risultato del prodotto è una matrice $C \in \mathbb{R}^{n \times n}$.

L'infrastruttura di calcolo parallelo utilizzata per gli esperimenti è un'architettura di tipo MIMD a memoria distribuita. La gestione dei processori avviene mediante la libreria *Message Parsing Interface* (MPI), che consente agli sviluppatori di creare programmi che possono essere eseguiti in modo efficiente sulla maggior parte delle architetture parallele (cluster).

Il linguaggio di programmazione adoperato per sviluppare il software è il linguaggio C.

2 Input e Output

Il software riceve in input un unico parametro: $n \in \mathbb{N}$, ovvero la dimensione delle matrici A e B. Tale valore deve essere multiplo del numero dei processori p.

L'input viene fornito all'interno del file *MatriceMatrice.pbs*, in cui viene indicato anche il numero di processori p da impiegare per l'esecuzione parallela. Il parametro p deve essere un quadrato perfetto. Per facilitare la comprensione circa i parametri di input del software, è stata implementata la funzionalità *help*, la quale può essere invocata direttamente da linea di comando mediante il comando `--help`.

L'output è contenuto nel file *MatriceMatrice.out* ed è strutturato nel seguente modo:

```
Esecuzione con NCPU: 4 - ORDER: 6

 > Generated Matrix A

 [1.706211] [1.974650] [3.345240] [1.800358] [2.101916] [1.651721]
 [3.787052] [0.256780] [4.441731] [1.240029] [1.858540] [2.358070]
 [1.942955] [4.861951] [0.279836] [4.967830] [4.138774] [3.777608]
 [0.132955] [1.581089] [4.245382] [4.611039] [2.126239] [3.754856]
 [3.674155] [0.217014] [1.435003] [1.295704] [3.839169] [3.742574]
 [2.306918] [0.545381] [0.717223] [0.652159] [2.345738] [2.819139]


 > Generated Matrix B

 [2.303879] [1.132790] [3.075919] [1.745611] [2.372819] [4.934459]
 [4.103680] [4.315774] [4.796410] [4.383516] [4.283604] [3.935184]
 [3.161124] [4.416560] [0.516274] [2.406507] [4.027598] [2.642513]
 [1.161363] [2.701754] [2.859527] [2.596366] [3.997458] [1.698696]
 [1.338940] [1.304376] [2.244077] [2.056163] [1.956535] [4.589815]
 [4.875302] [4.260414] [0.722605] [2.951222] [1.006025] [3.095424]


 > Product Matrix C

 [35.566803] [39.872202] [27.505029] [33.555497] [38.951431] [42.848121]
 [39.244418] [40.836152] [24.593990] [32.425591] [38.940980] [49.370869]
 [55.040831] [59.334506] [55.663846] [57.934373] [58.320741] [68.587917]
 [46.722834] [56.952769] [30.854400] [44.804612] [50.556909] [47.311083]
 [38.782970] [35.889718] [28.108041] [33.121521] [31.883439] [54.182908]
 [27.462513] [24.967052] [19.248052] [22.980037] [20.731402] [36.025592]


 > Max time: 0.000080
```

3 Indicatori di errore

Gli indicatori di errore, illustrati anche nella funzione *help*, sono descritti nella Tabella 1:

Codice	Errore	Descrizione
400	ERR_ARGC	Invalid number of arguments
401	ERR_NO_ORDER	Mandatory argument [-o --order] not provided
402	ERR_ORDER	Invalid orderof the square matrix provided
403	ERR_PROC	Invalid number of processors
404	ERR_MEMORY	Unable to allocate memory

Tabella 1: Indicatori di errori

4 Subroutine

In questa sezione vengono elencate e descritte tutte le funzioni adoperate all'interno del software, compreso quelle della libreria *mpi.h* (implementazione di MPI, un protocollo di comunicazione utilizzato nelle applicazioni per sistemi a memoria distribuita per il calcolo parallelo).

4.1 Funzioni MPI

```
int MPI_Init(int* argc, char* argv[])
```

Descrizione: inizializzare l'ambiente di esecuzione MPI consentendo la comunicazione e la sincronizzazione tra processori.

Input: *argc* puntatore al numero di argomenti; *argv* puntatore all'array degli argomenti.

Errori: Restituisce un codice di errore tra (MPI_SUCCESS, MPI_ERR_OTHER).

```
int MPI_Comm_rank(MPI_Comm comm, int* rank)
```

Descrizione: determina l'identificativo del processo chiamante nel comunicatore.

Input: *comm* communicator di riferimento.

Output: *rank* identificativo del processo chiamante nel gruppo *comm*.

Errori: Restituisce un codice di errore tra (MPI_SUCCESS, MPI_ERR_OTHER).

```
int MPI_Comm_size(MPI_Comm comm, int* size)
```

Descrizione: determina la dimensione del gruppo associato a un comunicatore.

Input: *comm* communicator di riferimento.

Output: *size* numero di processori nel gruppo *comm*.

Errori: Restituisce un codice di errore tra (MPI_SUCCESS, MPI_ERR_OTHER).

```
int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm
    comm)
```

Descrizione: trasmette un messaggio dal processo con identificativo *root* a tutti gli altri processi del comunicatore.

Input: *buffer* puntatore al buffer di lettura; *count* numero di elementi del buffer; *root* identificativo del processo root; *comm* communicator di riferimento.

Output: *buffer* puntatore al buffer di scrittura.

Errori: Restituisce un codice di errore tra (MPI_SUCCESS, MPI_ERR_COMM, MPI_ERR_COUNT, MPI_ERR_TYPE, MPI_ERR_BUFFER, MPI_ERR_ROOT).

```
int MPI_Abort(MPI_Comm comm, int errorcode)
```

Descrizione: termina l'ambiente di esecuzione MPI, nello specifico tutti i processi MPI associati alla comunicazione del comunicatore *comm*.

Input: *comm* communicator di riferimento; *errorcode* codice di errore per tornare all'ambiente chiamante.

Errori: Restituisce un codice di errore tra (MPI_SUCCESS, MPI_ERR_OTHER).

```
int MPI_Finalize(void)
```

Descrizione: termina l'ambiente di esecuzione MPI. Tutti i processi devono chiamare questa routine prima di uscire.

Errori: Restituisce un codice di errore tra (MPI_SUCCESS, MPI_ERR_OTHER).

```
int MPI_Barrier(MPI_Comm comm)
```

Descrizione: si blocca finché tutti i processi nel comunicatore non hanno raggiunto questa routine.

Input: *comm* communicator di riferimento.

Errori: Restituisce un codice di errore tra (MPI_SUCCESS, MPI_ERR_OTHER).

```
int MPI_Reduce(const void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype
    , MPI_Op op, int root, MPI_Comm comm)
```

Descrizione: riduce i valori su tutti i processi a un unico valore. É possibile effettuare la riduzione con diverse operazioni (MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, MPI_LAND, MPI_LOR, MPI_BAND, MPI_BOR, MPI_MAXLOC, MPI_MINLOC).

Input: *sendbuf* puntatore al buffer di trasmissione; *count* numero di elementi del buffer; *datatype* tipo degli elementi del buffer; *op* operazione di riduzione; *root* identificativo del processo root che riceverà il risultato; *comm* comunicatore di riferimento.

Output: *recvbuf* puntatore al buffer di ricezione.

Errori: Restituisce un codice di errore tra (MPI_SUCCESS, MPI_ERR_COMM, MPI_ERR_COUNT, MPI_ERR_TYPE, MPI_ERR_BUFFER).

```
double MPI_Wtime(void)
```

Descrizione: Restituisce il tempo trascorso sul processore chiamante.

Output: Tempo trascorso sul processore chiamante.

```
MPI_Cart_create(MPI_Comm comm_old, int dim, int *ndim, int *period, int reorder,
MPI_Comm *new_comm)
```

Descrizione: operazione collettiva che restituisce un nuovo communicator *new_comm* in cui i processi sono organizzati in una griglia di dimensioni *dim*. L'iesima dimensione ha lunghezza $ndim[i]$. Se $period[i] = 1$, l'iesima dimensione della griglia è periodica; non lo è se $period[i] = 0$.

Input: *comm_old* communicator 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*, **new_comm* communicator di output associato alla griglia.

Errori: Restituisce un codice di errore tra (MPI_SUCCESS, MPI_ERR_TOPOLOGY, MPI_ERR_DIMS, MPI_ERR_ARG).

```
MPI_Cart_coords(MPI_Comm comm_grid, int menum_grid, int dim, int *coordinate)
```

Descrizione: operazione collettiva che restituisce a ciascun processo di *comm_grid* con identificativo *menum_grid* le sue coordinate all'interno della griglia predefinita.

Input: **comm_grid* identificativo di communicator, *menum_grid* identificativo del processore del quale restituire le coordinate, **coordinate* vettore contenente le coordinate del processore, *dim* dimensione del vettore **coordinate*.

Errori: Restituisce un codice di errore tra (MPI_SUCCESS, MPI_ERR_TOPOLOGY, MPI_ERR_RANK, MPI_ERR_DIMS, MPI_ERR_ARG).

```
int MPI_Cart_sub(MPI_Comm comm, const int remain_dims[], MPI_Comm *newcomm)
```

Descrizione: partiziona un communicator in sottogruppi che formano sottogriglie cartesiane di dimensione inferiore

Input: *comm* identificativo di communicator, *remain_dims* specifica se la dimensione i-esima viene mantenuta nella sottogriglia (true) o viene eliminata (false) (vettore logico).

Errori: Restituisce un codice di errore tra (MPI_SUCCESS, MPI_ERR_TOPOLOGY, MPI_ERR_COMM, MPI_ERR_ARG).

```
int MPI_Cart_rank(MPI_Comm comm, const int coords[], int *rank)
```

Descrizione: determina il rank del processo nel communicator data la posizione cartesiana.

Input: *comm* identificativo del communicator, *coords* specifica le coordinate cartesiane di un processo; *rank* identificativo del processso.

Errori: Restituisce un codice di errore tra (MPI_SUCCESS, MPI_ERR_TOPOLOGY, MPI_ERR_COMM, MPI_ERR_ARG).

```
int MPI_Send(const void* buf, int count, MPI_Datatype datatype, int dest, int tag,
    MPI_Comm comm)
```

Descrizione: Funzione di trasmissione dati bloccante.

Input: *buf* puntatore al buffer di trasmissione; *count* numero di elementi del buffer; *datatype* tipo degli elementi del buffer; *dest* idenbtificativo del processo destinatario; *tag* identificativo della comunicazione; *comm* comunicatore di riferimento.

Errori: Restituisce un codice di errore tra (MPI_SUCCESS, MPI_ERR_COMM, MPI_ERR_COUNT, MPI_ERR_TYPE, MPI_ERR_TAG, MPI_ERR_RANK).

```
int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int tag,
 MPI_Comm comm, MPI_Status* status)
```

Descrizione: Funzione di ricezione dati bloccante.

Input: *count* numero di elementi del buffer, *datatype* tipo degli elementi del buffer, *dest* idenbtificativo del processo destinatari,; *tag* identificativo della comunicazione, *comm* comunicatore di riferimento.

Output: *buf* puntatore al buffer di ricezione; *status* informazioni sulla comunicazione.

Errori: Restituisce un codice di errore tra (MPI_SUCCESS, MPI_ERR_COMM, MPI_ERR_COUNT, MPI_ERR_TYPE, MPI_ERR_TAG, MPI_ERR_RANK).

```
MPI_Isend(void *buffer, int count, MPI_Datatype datatype, int dest, int tag,
    MPI_Comm comm, MPI_Request *request)
```

Descrizione: Funzione di trasmissione dati non bloccante.

Input: *buffer* puntatore al buffer di trasmissione; *count* numero di elementi del buffer; *datatype* tipo degli elementi del buffer; *dest* idenbtificativo del processo destinatario; *tag* identificativo della comunicazione; *comm* comunicatore di riferimento, *request* contiene informazione sulla fase di trasmissione del messaggio.

Errori: Restituisce un codice di errore tra (MPI_SUCCESS, MPI_ERR_COMM, MPI_ERR_COUNT, MPI_ERR_TYPE, MPI_ERR_TAG, MPI_ERR_RANK).

4.2 Funzioni codificate

```
void help(char* program_name)
```

Descrizione: stampa a video l'help del programma.

Input: *program_name* nome del programma.

```
int check_args(int argc, char** argv)
```

Descrizione: verifica l'integrità degli argomenti passati in ingresso al programma.

Input: *argc* numero di argomenti passati in ingresso al programma, *argv* argomenti passati in ingresso al programma.

Output: Codice errore/successo.

```
double* initialize_matrix(int matrix_order)
```

Descrizione: inizializza una matrice quadrata dinamica di ordine *matrix_order*.

Input: *matrix_order* ordine della matrice quadrata.

Output: Matrice quadrata inizializzata.

```
double* generate_random_matrix(int matrix_order, double lower, double upper)
```

Descrizione: genera una matrice quadrata pseudo-randomica di ordine *matrix_order*

Input: *matrix_order* ordine della matrice quadrata, *lower* limite inferiore del valore degli elementi, *upper* limite superiore del valore degli elementi.

Output: Matrice quadrata generata.

```
void print_matrix(double* matrix, int matrix_order)
```

Descrizione: effettua la stampa della matrice passata in ingresso.

Input: *matrix* matrice da stampare, *matrix_order* ordine della matrice quadrata.

```
void create_grid(MPI_Comm* grid, MPI_Comm* sub_rgrid, MPI_Comm* sub_cgrid, int
    mpi_rank, int mpi_size, int grid_order, int* periods, int reorder, int* coords)
```

Descrizione: Crea una griglia di processori di ordine *grid_order* e due sotto griglie su quest'ultima, di cui una di riga e l'altra di colonna.

Input: *grid* griglia creata, *sub_rgrid* sotto griglia riga creata, *sub_cgrid* sotto griglia colonna creata, *mpi_rank* rank del processore chiamante, *mpi_size* numero di processori impiegati, *periods* periodicità della griglia, *reorder* indica se riordinare il

rank dei processori, *coords* coordinate assegnate al processore.

```
void send_matrix_from_processor_0(double* matrix, int matrix_order, int
    sub_matrix_order, MPI_Comm grid, int mpi_size) {
```

Descrizione: suddivide e invia la matrice passata in ingresso, dal processore 0 ai restanti.

Input: *matrix* matrice da inviare, *matrix_order* ordine della matrice quadrata, *sub_matrix_order* ordine della sotto matrice da inviare, *grid* griglia di processori, *mpi_size* numero di processori impiegati.

```
void receive_matrix_from_processor_0(double* sub_matrix, int sub_matrix_order, int
    mpi_rank) {
```

Descrizione: riceve la sotto matrice inviata dal processore 0.

Input: *sub_matrix* sotto matrice, *sub_matrix_order* ordine della sotto matrice quadrata, *mpi_rank* rank del processore chiamante.

```
void bmr(double* sub_matrix_a, double* sub_matrix_b, double* sub_matrix_c, int
    sub_matrix_order, MPI_Comm grid, MPI_Comm sub_rgrid, MPI_Comm sub_cgrid, int
    grid_order, int coords[2]) {
```

Descrizione: effettua il prodotto righe per colonne parallelo mediante la strategia BMR (Broadcast Multiply Rolling).

Input: *sub_matrix_a* sotto matrice A (sinistra) da impiegare nel prodotto, *sub_matrix_b* sotto matrice B (destra) da impiegare nel prodotto, *sub_matrix_c* sotto matrice C risultante, *sub_matrix_order* ordine delle sotto matrici quadrate, *grid* griglia dei processori, *sub_rgrid* sotto griglia di riga, *sub_cgrid* sotto griglia di colonna, *grid_order* ordine della griglia quadrata di processori, *coords[2]* coordinate nella griglia del processore chiamante.

```
void merge(double* matrix_c, int matrix_order, double* sub_matrix_c, int
    sub_matrix_order, MPI_Comm grid, int mpi_size, int mpi_rank) {
```

Descrizione: unisce tutte le sotto matrici calcolate dai singoli processori in un'unica matrice C.

Input: *matrix_c* matrice risultate dall'operazione di unione, *matrix_order* ordine della matrice quadrata risultante, *sub_matrix_c* sotto matrice da inserire in quella finale, *sub_matrix_order* ordine della sotto matrice quadrata, *grid* griglia dei processori, *mpi_size* numero di processori impiegati, *mpi_rank* rank del processore chiamante.

```
void multiply(double* matrix_a, double* matrix_b, double* matrix_c, int
    matrix_order){
```

Descrizione: effettua Il prodotto righe per colonne delle matrici quadrate passate in ingresso.

Input: *matrix_a* matrice A (sinistra), *matrix_b* matrice B (destra), *matrix_c* matrice C (risultato del prodotto), *matrix_order* ordine delle matrici quadrate.

5 Descrizione dell'algoritmo

In questa sezione viene fornita una panoramica generale del funzionamento dell'algoritmo del prodotto tra due matrici quadrate in parallelo.

5.1 Controllo dell'input

Il processore 0 controlla che il valore n passato in input sia multiplo della radice quadrata del numero di processori p, ossia $(n \ mod \ \sqrt{p} = 0)$ e, per tale motivo, il numero di processori p deve essere necessariamente un quadrato perfetto. Se tali condizione non vengono soddisfatte, il software termina.

5.2 Allocazione e distribuzione dei blocchi

Le matrici A e B vengono allocate con la dimensione n fornita in input e inizializzate con valori reali casuali; inoltre, viene creata la griglia dei processori in base al parametro p.

Ciascun processore alloca nella propria memoria locale due sottomatrici, al fine di contenere i valori che verranno distribuiti da p_0 sia per la matrice A che per la matrice B. Il numero dei valori distribuiti da p_0 a ogni processore è pari al rapporto tra n e l'ordine della griglia dei processori.

```
if(!mpi_rank) {
  send_matrix_from_processor_0(matrix_a, matrix_order, sub_matrix_order, grid,
    mpi_size);
  send_matrix_from_processor_0(matrix_b, matrix_order, sub_matrix_order, grid,
    mpi_size);
  for(int i = 0; i < sub_matrix_order; i++)
    for(int j = 0; j < sub_matrix_order; j++) {
      sub_matrix_a[i*sub_matrix_order+j] = matrix_a[i*matrix_order+j];
      sub_matrix_b[i*sub_matrix_order+j] = matrix_b[i*matrix_order+j];
    }
} else {
  receive_matrix_from_processor_0(sub_matrix_a, sub_matrix_order, mpi_rank);
  receive_matrix_from_processor_0(sub_matrix_b, sub_matrix_order, mpi_rank);
}
```

Codice 1: Distribuzione delle matrici

5.3 Broadcast Multiply Rolling

Siano $A = (a_{i,j})$, $B = (b_{i,j})$ con $i, j = 0, 1, \ldots, n-1$ matrici quadrate $n \times n$, $C = A \cdot B = c_{i,j}$ dove $c_{i,j}$ è il prodotto scalare della i-esima riga di A con la j-esima colonna di B (approfondimento sottosezione 5.4).

Si assuma che i processori siano distribuiti secondo una griglia cartesiana e che le sottomatrici quadrate di A e B vengano distribuite in modo che al processore $p_{i,j}$ vengano assegnati i blocchi $A_{i,j}$ e $B_{i,j}$.

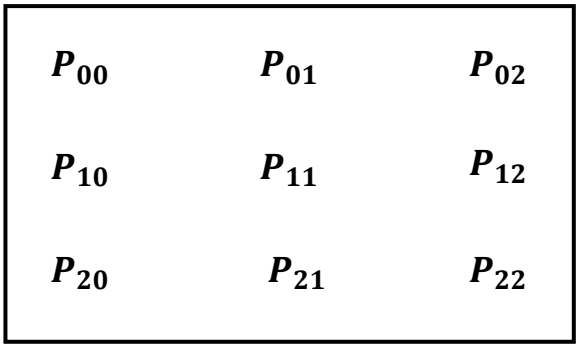


Figura 1: Esempio griglia $3 \times 3 = 9$ processori

L'algoritmo BMR per la moltiplicazione tra matrici procede in $\sqrt{p}$ fasi: una fase per ogni termine $a_{ik}b_{kj}$ nel prodotto scalare

$$c_{ij} = a_{i0}b_{0j} + a_{i1}b_{1j} + \cdots + a_{i,n-1}b_{n-1,j} \tag{1}$$

Durante la prima fase sul processore $(i, j) : c_{i,j} = a_{ii}b_{ij}$, viene effettuato il *broadcast* di a_{ii} attraverso la i-esima riga dei processori; dopodiché viene effettuata localmente la moltiplicazione con b_{ij} (sottosezione 5.4) e infine avviene lo *shift* degli elementi di B sulla riga superiore dei processori, mentre gli elementi nella riga superiore vengono spostati nella riga inferiore (shift circolare).

Nella fase successiva sul processore $(i, j) : c_{i,j} = c_{ij} + a_{i,i+1}b_{i+1,j}$ sull'ultima riga: $i + 1 \rightarrow (i + 1) mod \sqrt{p}$ avviene il *broadcast* di $a_{i,i+1}$ attraverso l'i-esima riga dei processori, viene effettuato il prodotto locale, dopo lo shift circolare della fase 0, con l'elemento locale di B che è $b_{i+1,j}$, e, infine, si effettua lo shift circolare degli elementi di B sulla riga dei processori superiore. In altri termini, ogni processore moltiplica l'elemento immediatamente a destra della diagonale di A (nella sua riga di processori) per l'elemento di B direttamente sotto il proprio elemento di B. Tale meccanismo si itera fino alla fase $k = \sqrt{p} - 1$ in cui si ha la moltiplicazione completata. In generale, durante la fase k-esima, ogni processore moltiplica l'elemento k colonne a destra della diagonale di A per l'elemento k righe sotto il proprio elemento di B. Ovviamente, non possiamo semplicemente aggiungere k a un pedice di riga o

colonna e aspettarci di ottenere sempre un numero di riga o colonna valido; per tale motivo si usa il modulo $\sqrt{p}$.

5.4 Prodotto tra sottomatrici

Moltiplicando la sottomatrice A di dimensione $n \times n$ per la sottomatrice B di dimensione $n \times n$ si ottiene la sottomatrice C di dimensione $n \times n$ con ogni elemento della sottomatrice C definito secondo l'espressione:

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} \cdot b_{kj}, 0 \leq i < n, 0 \leq j < n \tag{2}$$

Come si vede nella Equazione 2, ogni elemento della sottomatrice risultato C è il prodotto scalare della corrispondente riga della sottomatrice A e della colonna della sottomatrice B:

$$c_{ij} = \left(a_i, b_j^T\right), a_i = \left(a_{i_0}, a_{i_1}, \ldots, a_{i_n-1}\right), b_j^T = \left(b_{0_j}, b_{1_j}, \ldots, b_{n-1_j}\right)^T \tag{3}$$

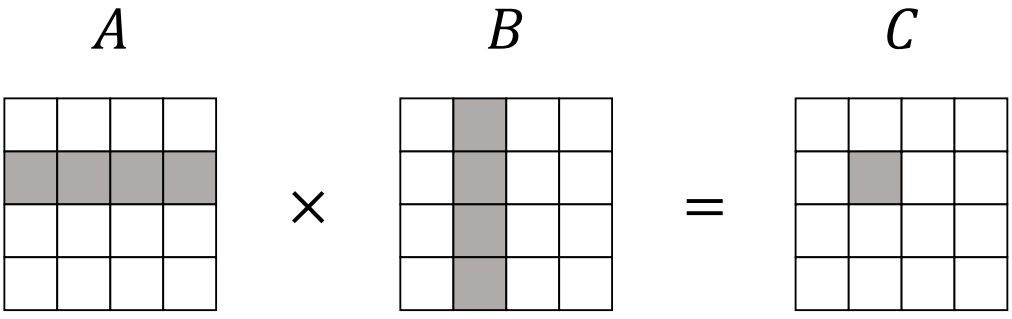


Figura 2: L'elemento della sottomatrice risultato C è il risultato della moltiplicazione scalare della riga della matrice A e della corrispondente colonna della matrice B

Ad esempio, il prodotto di due sottomatrici di dimensione 2:

$$\begin{pmatrix} 1 & 0 \\ 2 & 2 \end{pmatrix} \cdot \begin{pmatrix} 3 & 1 \\ 4 & 2 \end{pmatrix} = \begin{pmatrix} 3 & 1 \\ 14 & 6 \end{pmatrix} \tag{4}$$

Quindi, per ottenere la matrice risultato C $n \times n$, devono essere eseguite operazioni di moltiplicazione delle righe della matrice A per le colonne della matrice B. Ciascuna operazione di questo tipo prevede la moltiplicazione degli elementi di riga e di colonna e l'ulteriore somma dei prodotti ottenuti.

Lo pseudo-codice per l'implementazione della moltiplicazione matrice-matrice è il seguente:

```
// Algoritmo sequenziale del prodotto tra matrici
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        C[i][j] = 0;
        for (k=0; k<n; k++)
            C[i][j] += A[i][k]*B[k][j];
    }
}
```

6 Analisi dei tempi e delle prestazioni

La valutazione delle performance del software avvengono mediante i seguenti parametri:

1. Tempo medio impiegato: per ogni esperimento sono state effettuate 5 prove ed è stata, successivamente, considerata la media aritmetica dei tempi di ciascuna prova;

2. Speed Up: misura la riduzione del tempo di esecuzione rispetto all'algoritmo su 1 processore, $S(p) = \frac{T(1)}{T(p)}$ dove $T(1)$ rappresenta il tempo impiegato dall'algoritmo con singolo processore, mentre $T(p)$ rappresenta il tempo impiegato dall'algoritmo con p processori. Si osservi che $S(p)_{ideale} = p$;

3. Efficienza: misura quanto l'algoritmo sfrutta il parallelismo del calcolatore, $E(p) = \frac{S(p)}{p}$. Si osservi che $E(p)_{ideale} = 1$

Per misurare il tempo di esecuzione del software, è stata adoperata la funzione di libreria *MPI_Wtime*.

6.1 Dati raccolti

Di seguito vengono riportati, in formato tabellare, i tempi ottenuti dai vari test effettuati. Il numero di test totali per ogni tupla (n, p) è di cinque, dove n rappresenta l'ordine delle matrici quadrate (dimensione del problema) e p il numero di processori impiegati. Per ogni tupla (n, p) è riportato il tempo medio rilevato.

	Ordine 8			**Ordine 20**	
Processori →	**P1**	**P4**		**P1**	**P4**
Test N°1	0,000009	0,000098		0,000100	0,000535
Test N°2	0,000009	0,000104		0,000100	0,000582
Test N°3	0,000009	0,000139		0,000099	0,000600
Test N°4	0,000009	0,000106		0,000100	0,000563
Test N°5	0,000009	0,000050		0,000100	0,000621
Media	**0,000009**	**0,000099**		**0,000100**	**0,000580**

Tabella 2: Test $(n : 8,\ p : 1)$, $(n : 8,\ p : 4)$, $(n : 20,\ p : 1)$, $(n : 20,\ p : 4)$

	Ordine 80			**Ordine 200**	
Processori →	**P1**	**P4**		**P1**	**P4**
Test N°1	0,006220	0,002597		0,083774	0,026211
Test N°2	0,006216	0,002569		0,083025	0,026363
Test N°3	0,006219	0,002572		0,085099	0,026355
Test N°4	0,005891	0,002511		0,084403	0,026283
Test N°5	0,006226	0,002509		0,083792	0,025874
Media	**0,006154**	**0,002552**		**0,084019**	**0,026217**

Tabella 3: Test $(n : 80,\ p : 1)$, $(n : 80,\ p : 4)$, $(n : 200,\ p : 1)$, $(n : 200,\ p : 4)$

	Ordine 800			Ordine 2000	
Processori →	**P1**	**P4**		**P1**	**P4**
Test N°1	7,624933	1,962537		152,037466	40,825244
Test N°2	7,642270	1,920144		152,787322	40,538395
Test N°3	7,527564	1,920190		152,882391	40,933426
Test N°4	7,652961	1,971942		152,266537	40,675243
Test N°5	7,504089	1,934581		151,963301	40,675243
Media	**7,590363**	**1,941879**		**152,387403**	**40,729510**

Tabella 4: Test $(n : 800,\ p : 1)$, $(n : 800,\ p : 4)$, $(n : 2000,\ p : 1)$, $(n : 2000,\ p : 4)$

6.2 Speed Up ed Efficienza

	Ordine 8		
Processori	**Tempo Medio**	**Speed Up**	**Efficienza**
P1	0,000009	1,000000	1,000000
P4	0,000099	0,090543	0,022636

Tabella 5: Dati rilevati con matrici quadrate di ordine 8

	Ordine 20		
Processori	**Tempo Medio**	**Speed Up**	**Efficienza**
P1	0,000100	1,000000	1,000000
P4	0,000580	0,172010	0,043002

Tabella 6: Dati rilevati con matrici quadrate di ordine 20

	Ordine 80		
Processori	**Tempo Medio**	**Speed Up**	**Efficienza**
P1	0,006154	1,000000	1,000000
P4	0,002552	2,411977	0,602994

Tabella 7: Dati rilevati con matrici quadrate di ordine 80

	Ordine 200		
Processori	**Tempo Medio**	**Speed Up**	**Efficienza**
P1	0,084019	1,000000	1,000000
P4	0,026217	3,204713	0,801178

Tabella 8: Dati rilevati con matrici quadrate di ordine 200

	Ordine 800		
Processori	**Tempo Medio**	**Speed Up**	**Efficienza**
P1	7,590363	1,000000	1,000000
P4	1,941879	3,908773	0,977193

Tabella 9: Dati rilevati con matrici quadrate di ordine 800

	Ordine 2000		
Processori	**Tempo Medio**	**Speed Up**	**Efficienza**
P1	152,387403	1,000000	1,000000
P4	40,729510	3,741449	0,935362

Tabella 10: Dati rilevati con matrici quadrate di ordine 2000

6.3 Grafici

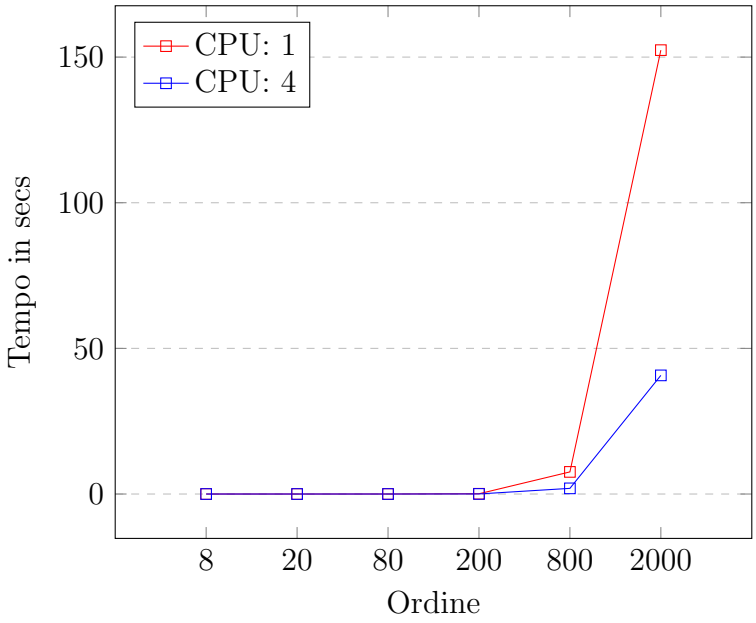


Figura 3: Grafico del tempo medio

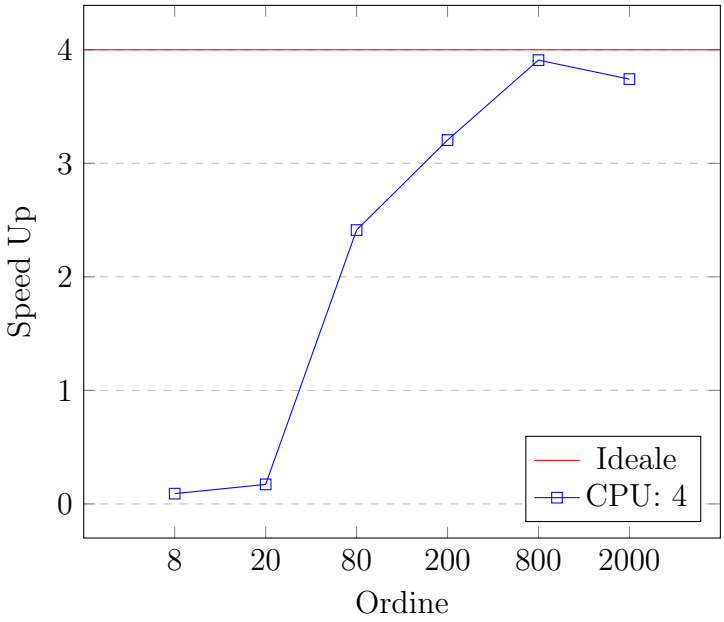


Figura 4: Grafico dello Speed Up (4 CPU)

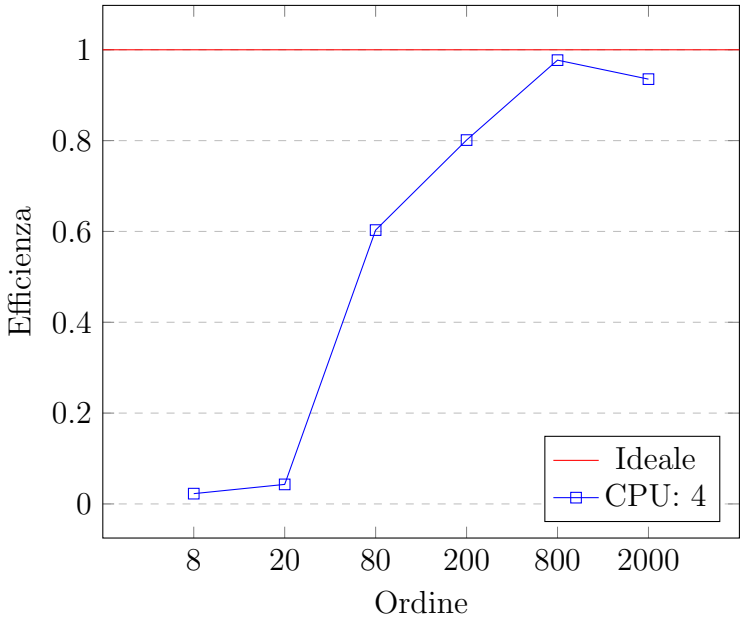


Figura 5: Grafico dell'Efficienza (4 CPU)

6.4 Considerazioni sui risultati ottenuti

In questo studio, sono stati condotti diversi test tenendo in considerazione i seguenti parametri:

- o: ordine delle matrici quadrate su cui effettuare il prodotto righe per colonne. Tale valore assume valori in $O = \{8, 20, 80, 200, 800, 2000\}$;
- p: numero di processori da impiegare. Tale parametro può assumere valori in $P = \{1, 4\}$.

Ogni test è identificato dalla tupla (o, p, i) dove i rappresenta l'i-esimo test ($(i \in I = [1, 5])$). Pertanto, essendo $(o, p, i) \in \{(o, p, i) : o \in O \land \ p \in P \land i \in I\} := O \times P \times I = T$, il numero di test totali effettuati è $|T| = 60$.
I risultati ottenuti durante i test hanno mostrato come sia poco vantaggioso utilizzare molti processori per matrici di piccole dimensioni, infatti, in questa circostanza, speed up ed efficienza assumono valori bassi. Nello specifico, l'uso di più processori ($p > 1$) con matrici piccole ($n < 100$) non comporta alcun vantaggio significativo rispetto all'utilizzo di un solo processore. Per ordini di matrici superiori ($n > 100$) e utilizzando 4 processori, lo speed up ottenuto tende a quello a ideale e, di conseguenza, l'efficienza tende a 1; ciò significa che il calcolatore viene sfruttato correttamente senza sprechi di risorse.

7 Esempi d'uso

7.1 Esecuzione dei test

L'esecuzione dei test sul cluster avviene mediante l'esecuzione del seguente script PBS. Per variare l'ordine delle matrici quadrate e il numero di processori da impiegare è necessario modificare rispettivamente le variabili ORDER e NCPU, le quali, nel seguente PBS, sono state impostate rispettivamente a 6 e 4. Si noti che per

semplicità di verifica la stampa della matrice e del vettore generati e del vettore risultante sono stati disattivati.

```
#!/bin/bash

############################
##  The PBS directives    ##
############################

#PBS -q studenti
#PBS -l nodes=4:ppn=8

#PBS -N MatriceMatrice
#PBS -o MatriceMatrice.out
#PBS -e MatriceMatrice.err



############################
## Informazioni sul Job ##
############################

NCPU=4

sort -u $PBS_NODEFILE > hostlist
echo ------------------------------
echo 'This Job is allocated on '${NCPU}' cpu(s)'
echo 'Job is running on node(s):'
cat hostlist

PBS_O_WORKDIR=$PBS_O_HOME/MatriceMatrice
echo ----------------------------------
echo PBS: qsub is running on $PBS_O_HOSTS
echo PBS: originating queue is $PBS_O_QUEUE
echo PBS: executing queue is $PBS_QUEUE
echo PBS: working directory is $PBS_O_WORKDIR
echo PBS: execution mode is $PBS_ENVIRONMENT
echo PBS: job identifier is $PBS_JOBID
echo PBS: job name is $PBS_JOBNAME
echo PBS: node file is $PBS_NODEFILE
echo PBS: current home directory is $PBS_O_HOME
echo PBS: PATH = $PBS_O_PATH
echo ------------------------------
```

```
41
42
43
44 ############################
45 ##      Compilazione      ##
46 ############################
47
48 /usr/lib64/openmpi/1.4-gcc/bin/mpicc -std=c99 -o $PBS_O_WORKDIR/MatriceMatrice
       $PBS_O_WORKDIR/MatriceMatrice.c
49
50
51
52 ############################
53 ##       Esecuzione       ##
54 ############################
55
56 ORDER=6
57
58 for TEST in {1..5}
59 do
60   echo "--------------------- NCPU: $NCPU - ORDER: $ORDER - TEST: $TEST
       ---------------------"
61   /usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile hostlist -np $NCPU
       $PBS_O_WORKDIR/MatriceMatrice -o $ORDER
62   echo -e "------------------------------------------------------------------------\n
       "
63 done
```

Il risultato ottenuto dall'esecuzione dello script PBS è contenuto nel file Matrice-Matrice.out mentre eventuali errori nel file MatriceMatrice.err:

```
-----------------------------
This Job is allocated on 4 cpu(s)
Job is running on node(s):
wn274.scope.unina.it
wn276.scope.unina.it
wn279.scope.unina.it
wn280.scope.unina.it
---------------------------------
PBS: qsub is running on
PBS: originating queue is studenti
PBS: executing queue is studenti
PBS: working directory is /homes/DMA/PDC/2021/LRCGLC98Q/MatriceMatrice
```

```
PBS: execution mode is PBS_BATCH
PBS: job identifier is 3995607.torque02.scope.unina.it
PBS: job name is MatriceMatrice
PBS: node file is /var/spool/pbs/aux//3995607.torque02.scope.unina.it
PBS: current home directory is /homes/DMA/PDC/2021/LRCGLC98Q
PBS: PATH = /usr/lib64/openmpi/1.2.7-gcc/bin:/usr/kerberos/bin:/opt/exp_soft/unina.
    it/intel/composer_xe_2013_sp1.3.174/bin/intel64:/opt/exp_soft/unina.it/intel/
    composer_xe_2013_sp1.3.174/mpirt/bin/intel64:/opt/exp_soft/unina.it/intel/
    composer_xe_2013_sp1.3.174/bin/intel64:/opt/exp_soft/unina.it/intel/
    composer_xe_2013_sp1.3.174/bin/intel64_mic:/opt/exp_soft/unina.it/intel/
    composer_xe_2013_sp1.3.174/debugger/gui/intel64:/opt/d-cache/srm/bin:/opt/d-
    cache/dcap/bin:/opt/edg/bin:/opt/glite/bin:/opt/globus/bin:/opt/lcg/bin:/usr/
    local/bin:/bin:/usr/bin:/opt/exp_soft/HADOOP/hadoop-1.0.3/bin:/opt/exp_soft/
    unina.it/intel/composerxe/bin/intel64/:/opt/exp_soft/unina.it/MPJExpress/mpj-
    v0_38/bin:/homes/DMA/PDC/2021/LRCGLC98Q/bin
------------------------------

-------------------- NCPU: 4 - ORDER: 6 - TEST: 1 --------------------
 > Max time: 0.000095
----------------------------------------------------------------------

-------------------- NCPU: 4 - ORDER: 6 - TEST: 2 --------------------
 > Max time: 0.000041
----------------------------------------------------------------------

-------------------- NCPU: 4 - ORDER: 6 - TEST: 3 --------------------
 > Max time: 0.000044
----------------------------------------------------------------------

-------------------- NCPU: 4 - ORDER: 6 - TEST: 4 --------------------
 > Max time: 0.000090
----------------------------------------------------------------------

-------------------- NCPU: 4 - ORDER: 6 - TEST: 5 --------------------
 > Max time: 0.000084
----------------------------------------------------------------------
```

7.2 Esecuzione utente

L'esecuzione del programma sul cluster con argomenti variabili a discrezione dell'utente avviene mediante l'esecuzione del seguente script PBS:

```
#!/bin/bash

############################
##  The PBS directives    ##
############################

#PBS -q studenti
#PBS -l nodes=4:ppn=8

#PBS -N MatriceMatrice
#PBS -o MatriceMatrice.out
#PBS -e MatriceMatrice.err



##########################
## Informazioni sul Job ##
##########################

NCPU=4

sort -u $PBS_NODEFILE > hostlist
echo ------------------------------
echo 'This Job is allocated on '${NCPU}' cpu(s)'
echo 'Job is running on node(s):'
cat hostlist

PBS_O_WORKDIR=$PBS_O_HOME/MatriceMatrice
echo ----------------------------------
echo PBS: qsub is running on $PBS_O_HOSTS
echo PBS: originating queue is $PBS_O_QUEUE
echo PBS: executing queue is $PBS_QUEUE
echo PBS: working directory is $PBS_O_WORKDIR
echo PBS: execution mode is $PBS_ENVIRONMENT
echo PBS: job identifier is $PBS_JOBID
echo PBS: job name is $PBS_JOBNAME
echo PBS: node file is $PBS_NODEFILE
echo PBS: current home directory is $PBS_O_HOME
echo PBS: PATH = $PBS_O_PATH
echo ------------------------------
```

```
44 ############################
45 ##      Compilazione      ##
46 ############################
47
48 /usr/lib64/openmpi/1.4-gcc/bin/mpicc -std=c99 -o $PBS_O_WORKDIR/MatriceMatrice
       $PBS_O_WORKDIR/MatriceMatrice.c
49
50
51
52 ############################
53 ##       Esecuzione       ##
54 ############################
55
56 ORDER=6
57
58 echo -e "\n-------------------- NCPU: $NCPU - ORDER: $ORDER --------------------"
59 /usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile hostlist -np $NCPU
       $PBS_O_WORKDIR/MatriceMatrice -o $ORDER
60 echo -e "\n-----------------------------------------------------------------"
```

```
-----------------------------
This Job is allocated on 4 cpu(s)
Job is running on node(s):
wn274.scope.unina.it
wn276.scope.unina.it
wn279.scope.unina.it
wn280.scope.unina.it
---------------------------------
PBS: qsub is running on
PBS: originating queue is studenti
PBS: executing queue is studenti
PBS: working directory is /homes/DMA/PDC/2021/LRCGLC98Q/MatriceMatrice
PBS: execution mode is PBS_BATCH
PBS: job identifier is 3995612.torque02.scope.unina.it
PBS: job name is MatriceMatrice
PBS: node file is /var/spool/pbs/aux//3995612.torque02.scope.unina.it
PBS: current home directory is /homes/DMA/PDC/2021/LRCGLC98Q
PBS: PATH = /usr/lib64/openmpi/1.2.7-gcc/bin:/usr/kerberos/bin:/opt/exp_soft/unina.
    it/intel/composer_xe_2013_sp1.3.174/bin/intel64:/opt/exp_soft/unina.it/intel/
    composer_xe_2013_sp1.3.174/mpirt/bin/intel64:/opt/exp_soft/unina.it/intel/
    composer_xe_2013_sp1.3.174/bin/intel64:/opt/exp_soft/unina.it/intel/
    composer_xe_2013_sp1.3.174/bin/intel64_mic:/opt/exp_soft/unina.it/intel/
    composer_xe_2013_sp1.3.174/debugger/gui/intel64:/opt/d-cache/srm/bin:/opt/d-
```

```
    cache/dcap/bin:/opt/edg/bin:/opt/glite/bin:/opt/globus/bin:/opt/lcg/bin:/usr/
    local/bin:/bin:/usr/bin:/opt/exp_soft/HADOOP/hadoop-1.0.3/bin:/opt/exp_soft/
    unina.it/intel/composerxe/bin/intel64/:/opt/exp_soft/unina.it/MPJExpress/mpj-
    v0_38/bin:/homes/DMA/PDC/2021/LRCGLC98Q/bin
------------------------------

-------------------- NCPU: 4 - ORDER: 6 --------------------

 > Generated Matrix A

 [4.259313] [4.701393] [0.226063] [0.896582] [2.503671] [0.528091]
 [0.647354] [4.764613] [2.737796] [0.682940] [2.985619] [3.239520]
 [0.308327] [0.455738] [2.769413] [0.422122] [0.174118] [1.373708]
 [1.958687] [4.698727] [4.078735] [3.674332] [1.388745] [3.573381]
 [3.139459] [3.646902] [1.075670] [2.568845] [1.725702] [2.279451]
 [2.182756] [0.985015] [1.980843] [2.408819] [1.881597] [4.484514]


 > Generated Matrix B

 [2.936910] [2.528951] [4.249127] [0.674706] [3.211891] [2.234746]
 [3.914226] [3.520218] [2.690484] [1.683639] [3.942340] [2.864602]
 [3.057347] [0.901027] [2.563329] [2.136082] [4.575359] [3.952074]
 [0.709463] [2.714818] [2.598976] [1.785133] [0.283663] [4.324679]
 [4.064584] [2.466419] [0.309694] [1.045427] [4.875237] [2.191291]
 [0.529941] [2.812147] [4.720243] [4.779068] [3.486853] [2.932134]


 > Product Matrix C

 [42.695010] [37.619438] [36.925140] [18.013828] [47.550946] [34.791653]
 [43.257962] [39.204283] [40.578570] [34.129094] [59.434413] [44.909928]
 [12.891626] [10.317855] [17.270434] [14.391638] [21.216568] [19.174445]
 [46.759632] [48.618259] [58.266511] [43.033447] [63.749332] [63.367694]
 [36.828541] [39.387039] [43.879555] [27.839475] [46.472508] [43.288545]
 [28.055660] [34.563768] [45.013704] [35.061305] [45.450475] [43.217713]


 > Max time: 0.000056

------------------------------------------------------------
```

7.2.1 Composizione argomenti

Il programma prende in ingresso i seguenti argomenti (necessariamente in ordine):

- **-o** o equivalentemente –**order**: ordine delle matrici quadrate

```
# Ordine delle matrici: 1000
/usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile hostlist -np $NCPU
    $PBS_O_WORKDIR/MatriceMatrice -o 1000

# Oppure, in modo equivalente:
/usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile hostlist -np $NCPU
    $PBS_O_WORKDIR/MatriceMatrice --order 1000
```

7.2.2 Help

Il programma offre anche la possibilità di stampare (sul file .out) l'help, ossia le informazioni sugli argomenti necessari per avviare il programmare correttamente:

```
...
...
...
# Per stampare l'help e' necessario passare al programma esclusivamente l'argomento
    -- help

##########################
##      Esecuzione      ##
##########################

echo -e "\n-------------------- NCPU: $NCPU - ORDER: $ORDER --------------------"
/usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile hostlist -np $NCPU
    $PBS_O_WORKDIR/MatriceMatrice --help
echo -e "\n------------------------------------------------------------"
```

Successivamente all'esecuzione dello script PBS riportato precedentemente, sul file .out troveremo:

```
...
...
...
-------------------- NCPU: 4 - ORDER: 6 --------------------

```

```
> Usage: /homes/DMA/PDC/2021/LRCGLC98Q/MatriceMatrice/MatriceMatrice [-o --order] <value>

 Mandatory arguments:
    -o  --order              Order of the square matrix

 Error codes:
    400 ERR_ARGC             Invalid number of arguments
    401 ERR_NO_ORDER         Mandatory argument [-o --order] not provided
    402 ERR_ORDER            Invalid order of the square matrix provided
    403 ERR_PROC             Invalid number of processors
    404 ERR_MEMORY           Unable to allocate memory

 Additional Info:
    The argument [-o --order] must be a multiple of the number of processors used
    The number of processors used must be a perfect square
    Communication strategy used: BMR

-------------------------------------------------------------
```

8 Codice Sorgente

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <ctype.h>
#include <limits.h>
#include <time.h>
#include <mpi.h>


#define SD_ARG_ORDER      "-o"

#define DD_ARG_ORDER      "--order"
#define DD_ARG_HELP       "--help"

#define SCC               200
#define SCC_ARGS          201
#define SCC_HELP          202

```

```
20 #define ERR_ARGC          400
21 #define ERR_NO_ORDER      401
22 #define ERR_ORDER         402
23 #define ERR_PROC          403
24 #define ERR_MEMORY        404
25
26 #define D_TAG              22
27
28
29 void help(char* program_name);
30 void print_matrix(double*, int);
31 void create_grid(MPI_Comm*, MPI_Comm*, MPI_Comm*,int, int, int, int*, int, int*);
32 void send_matrix_from_processor_0(double*, int, int, MPI_Comm, int);
33 void receive_matrix_from_processor_0(double*, int, int);
34 void merge(double*, int, double*, int, MPI_Comm, int, int);
35 void multiply(double*, double*, double*, int);
36 void bmr(double*, double*, double*, int, MPI_Comm, MPI_Comm, MPI_Comm, int, int[2])
       ;
37
38 int check_args(int, char**, int);
39
40 double* initialize_matrix(int);
41 double* generate_random_matrix(int, double, double);
42
43
44 int main(int argc, char** argv) {
45
46   /*
47     args_error: Codice errore ottenuto dalla verifica degli argomenti;
48
49     mpi_rank: Identificativo MPI del processore;
50     mpi_size: Numero di processori del communicator;
51
52     start_time: Tempo inizio somma;
53     end_time: Tempo fine somma;
54     delta_time: Differenza temporale tra inizio e fine somma;
55     max_time: Tempo di somma massimo;
56
57     matrix_order: Ordine delle matrici quadrate;
58     matrix_a: Matrice A (sinistra) da impiegare nel prodotto;
59     matrix_b: Matrice B (destra) da impiegare nel prodotto;
60     matrix_c: Matrice risultate dal prodotto;
61
```

```
62      sub_matrix_order: Ordine delle sotto matrici quadrate (sotto-problema);
63      sub_matrix_a: Sotto matrice A (sinistra) da impiegare nel prodotto (sotto-
        problema);
64      sub_matrix_b: Sotto matrice B (destra) da impiegare nel prodotto (sotto-
        problema);
65      sub_matrix_c: Sotto matrice risultate dal prodotto (sotto-problema);
66
67      periods: Array delle periodicita' della griglia di processori
68      coords: Coordinate del processore nella griglia
69
70      grid_order: Ordine della griglia quadrata di processori;
71      grid: Griglia di processori;
72      sub_rgrid: Sotto griglia di riga;
73      sub_cgrid: Sotto griglia di colonna;
74    */
75
76    int args_error;
77
78    int mpi_rank;
79    int mpi_size;
80
81    double start_time;
82    double end_time;
83    double delta_time;
84    double max_time;
85
86    int matrix_order;
87    double* matrix_a;
88    double* matrix_b;
89    double* matrix_c;
90
91    int sub_matrix_order;
92    double* sub_matrix_a;
93    double* sub_matrix_b;
94    double* sub_matrix_c;
95
96    int periods[2] = {0};
97    int coords[2];
98
99    int grid_order;
100   MPI_Comm grid;
101   MPI_Comm sub_rgrid;
102   MPI_Comm sub_cgrid;
```

```
103
104 MPI_Init(&argc, &argv);
105 MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
106 MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);
107
108
109 /* Controllo degli argomenti */
110
111 if(!mpi_rank) {
112
113   args_error = check_args(argc, argv, mpi_size);
114
115   switch(args_error) {
116
117     case SCC_HELP:
118       help(argv[0]);
119       break;
120
121     case ERR_ARGC:
122       printf(
123         "\n <!> ERROR: Invalid number of arguments! For additional info type %s.\
  n",
124         DD_ARG_HELP
125       );
126       break;
127
128     case ERR_NO_ORDER:
129       printf(
130         "\n <!> ERROR: Expected [%s %s] argument! For additional info type %s.\n"
  ,
131         SD_ARG_ORDER, DD_ARG_ORDER, DD_ARG_HELP
132       );
133       break;
134
135     case ERR_ORDER:
136       printf(
137         "\n <!> ERROR: Invalid value for argument [%s %s]! For additional info
  type %s.\n",
138         SD_ARG_ORDER, DD_ARG_ORDER, DD_ARG_HELP
139       );
140       break;
141
142     case ERR_PROC:
```

```
143         printf(
144           "\n <!> ERROR: Invalid number of processors used. For additional info
    type %s.\n",
145           DD_ARG_HELP
146         );
147         break;
148 
149     }
150 
151     if(args_error != SCC_ARGS && args_error != SCC_HELP)
152       MPI_Abort(MPI_COMM_WORLD, args_error);
153 
154   }
155 
156 
157   /* Propagazione del codice SCC_HELP */
158 
159   if(mpi_size != 1)
160     MPI_Bcast(&args_error, 1, MPI_INT, 0, MPI_COMM_WORLD);
161   if(args_error == SCC_HELP) {
162     MPI_Finalize();
163     return 0;
164   }
165 
166 
167   /* Lettura e distribuzione degli argomenti passati in ingresso */
168 
169   if(!mpi_rank)
170     matrix_order = atoi(argv[2]);
171   MPI_Bcast(&matrix_order, 1, MPI_INT, 0, MPI_COMM_WORLD);
172   grid_order = (int)sqrt(mpi_size);
173   sub_matrix_order = matrix_order/grid_order;
174 
175 
176   /* Generazione pseudo-randomica delle matrici da moltiplicare */
177 
178   if(!mpi_rank) {
179     srand(time(NULL));
180     matrix_a = generate_random_matrix(matrix_order, 0.0, 5.0);
181     matrix_b = generate_random_matrix(matrix_order, 0.0, 5.0);
182     if(!matrix_a || !matrix_b) {
183       printf("\n <!> ERROR: Unable to allocate memory.\n");
184       MPI_Abort(MPI_COMM_WORLD, ERR_MEMORY);
```

```
185     }
186   }
187
188
189   /* Stampa della matrici generate (solo se con ordine inferiore a 10) */
190
191   if(!mpi_rank && matrix_order <= 10) {
192     printf("\n > Generated Matrix A \n\n");
193     print_matrix(matrix_a, matrix_order);
194     printf("\n\n > Generated Matrix B \n\n");
195     print_matrix(matrix_b, matrix_order);
196   }
197
198
199   /* Se singolo processore allora effettua il prodotto sequenziale, altrimenti
        parallelo */
200
201   if (mpi_size != 1) {
202
203     /* Creazione della griglia e delle sotto-griglie riga e colonna */
204
205     create_grid(
206       &grid, &sub_rgrid, &sub_cgrid, mpi_rank, mpi_size, grid_order, periods, 0,
        coords
207     );
208
209
210     /*  Allocazione memoria */
211
212     sub_matrix_a = initialize_matrix(sub_matrix_order);
213     sub_matrix_b = initialize_matrix(sub_matrix_order);
214     sub_matrix_c = initialize_matrix(sub_matrix_order);
215     if(!mpi_rank && (!sub_matrix_a || !sub_matrix_b || !sub_matrix_c)) {
216       printf("\n <!> ERROR: Unable to allocate memory.\n");
217       MPI_Abort(MPI_COMM_WORLD, ERR_MEMORY);
218     }
219     if(!mpi_rank) {
220       matrix_c = initialize_matrix(matrix_order);
221       if(!matrix_c) {
222         printf("\n <!> ERROR: Unable to allocate memory.\n");
223         MPI_Abort(MPI_COMM_WORLD, ERR_MEMORY);
224       }
225     }
```

```
226
227
228     /* Distribuzione delle matrici */
229
230     if(!mpi_rank) {
231       send_matrix_from_processor_0(matrix_a, matrix_order, sub_matrix_order, grid,
    mpi_size);
232       send_matrix_from_processor_0(matrix_b, matrix_order, sub_matrix_order, grid,
    mpi_size);
233       for(int i = 0; i < sub_matrix_order; i++)
234         for(int j = 0; j < sub_matrix_order; j++) {
235           sub_matrix_a[i*sub_matrix_order+j] = matrix_a[i*matrix_order+j];
236           sub_matrix_b[i*sub_matrix_order+j] = matrix_b[i*matrix_order+j];
237         }
238     } else {
239       receive_matrix_from_processor_0(sub_matrix_a, sub_matrix_order, mpi_rank);
240       receive_matrix_from_processor_0(sub_matrix_b, sub_matrix_order, mpi_rank);
241     }
242
243     /* Sincronizzazione dei processori e salvataggio timestamp di inizio */
244
245     MPI_Barrier(MPI_COMM_WORLD);
246     start_time = MPI_Wtime();
247
248
249     /* Applicazione della strategia di comunicazione BMR */
250
251     bmr(
252       sub_matrix_a, sub_matrix_b, sub_matrix_c, sub_matrix_order,
253       grid, sub_rgrid, sub_cgrid, grid_order, coords
254     );
255
256
257     /* Composizione del risultato finale */
258
259     merge(matrix_c, matrix_order, sub_matrix_c, sub_matrix_order, grid, mpi_size,
    mpi_rank);
260
261
262     /* Calcolo del tempo impiegato */
263
264     end_time = MPI_Wtime();
265     delta_time = end_time - start_time;
```

```
266       MPI_Reduce(&delta_time, &max_time, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD)
      ;
267
268   } else {
269
270     /* Salvataggio timestamp di inizio */
271
272     start_time = MPI_Wtime();
273
274
275     /* Calcolo del prodotto */
276
277     matrix_c = initialize_matrix(matrix_order);
278     if(!matrix_c) {
279       printf("\n <!> ERROR: Unable to allocate memory.\n");
280       MPI_Abort(MPI_COMM_WORLD, ERR_MEMORY);
281     }
282     multiply(matrix_a, matrix_b, matrix_c, matrix_order);
283
284
285     /* Calcolo del tempo impiegato */
286
287     end_time = MPI_Wtime();
288     max_time = end_time - start_time;
289
290   }
291
292
293   /* Stampa della matrice calcolata (solo se con ordine inferiore a 10) e del tempo
        impiegato */
294
295   if(!mpi_rank) {
296     if(matrix_order <= 10) {
297       printf("\n\n > Product Matrix C \n\n");
298       print_matrix(matrix_c, matrix_order);
299     }
300     printf("\n\n > Max time: %lf\n", max_time);
301   }
302
303   MPI_Finalize();
304   return 0;
305
306 }
```

```
307
308
309 /*
310
311   Stampa a video l'help del programma
312
313   @params:
314     char* program_name: Nome del programma
315
316   @return:
317     void
318
319 */
320
321 void help(char* program_name) {
322
323   printf(
324     "\n > Usage: %s [%s %s] <value>",
325     program_name,
326     SD_ARG_ORDER, DD_ARG_ORDER
327   );
328
329   printf("\n\n\tMandatory arguments:");
330   printf(
331     "\n\t   %s  %-20s Order of the square matrix",
332     SD_ARG_ORDER, DD_ARG_ORDER
333   );
334
335   printf("\n\n\tError codes:");
336   printf("\n\t   %d %-20s Invalid number of arguments", ERR_ARGC, "ERR_ARGC");
337   printf(
338     "\n\t   %d %-20s Mandatory argument [%s %s] not provided",
339     ERR_NO_ORDER, "ERR_NO_ORDER",
340     SD_ARG_ORDER, DD_ARG_ORDER
341   );
342   printf(
343     "\n\t   %d %-20s Invalid order of the square matrix provided",
344     ERR_ORDER, "ERR_ORDER"
345   );
346   printf(
347     "\n\t   %d %-20s Invalid number of processors",
348     ERR_PROC, "ERR_PROC"
349   );
```

```
350   printf(
351     "\n\t   %d %-20s Unable to allocate memory",
352     ERR_MEMORY, "ERR_MEMORY"
353   );
354
355   printf("\n\n\tAdditional Info:");
356   printf(
357     "\n\t   The argument [%s %s] must be a multiple of the number of processors
      used",
358     SD_ARG_ORDER, DD_ARG_ORDER
359   );
360   printf(
361     "\n\t   The number of processors used must be a perfect square"
362   );
363   printf(
364     "\n\t   Communication strategy used: BMR\n"
365   );
366 }
367
368
369 /*
370
371   Verifica l'integrita' degli argomenti passati in ingresso al programma
372
373   @params:
374     int argc: Numero di argomenti passati in ingresso al programma
375     char* argv[]: Argomenti passati in ingresso al programma
376     int mpi_size: Numero di processori impiegati
377
378   @return:
379     int: Codice errore/successo
380
381 */
382
383 int check_args(int argc, char** argv, int mpi_size) {
384
385   if(argc == 2 && !strcmp(argv[1], DD_ARG_HELP))
386     return SCC_HELP;
387
388   if(argc == 3) {
389
390     if(sqrt(mpi_size) != (int)sqrt(mpi_size))
391       return ERR_PROC;
```

```
392
393     if(strcmp(argv[1], SD_ARG_ORDER) && strcmp(argv[1], DD_ARG_ORDER))
394       return ERR_NO_ORDER;
395
396     int matrix_order = atoi(argv[2]);
397
398     if(matrix_order <= 0 || (matrix_order % (int)sqrt(mpi_size)))
399       return ERR_ORDER;
400
401     return SCC_ARGS;
402
403   }
404
405   return ERR_ARGC;
406
407 }
408
409
410 /*
411
412   Inizializza una matrice quadrata dinamica di ordine  matrix_order
413
414   @params:
415     int matrix_order: Ordine della matrice quadrata
416
417   @return:
418     double*: Matrice quadrata inizializzata
419
420 */
421
422 double* initialize_matrix(int matrix_order) {
423
424   double* matrix = (double*) calloc((matrix_order * matrix_order), sizeof(double));
425   return matrix;
426
427 }
428
429
430 /*
431
432   Genera una matrice quadrata pseudo-randomica di ordine matrix_order
433
434   @params:
```

```
435     int matrix_order: Ordine della matrice quadrata
436     double lower: Limite inferiore del valore degli elementi
437     double upper: Limite superiore del valore degli elementi
438
439   @return:
440     double*: Matrice quadrata generata
441
442 */
443
444 double* generate_random_matrix(int matrix_order, double lower, double upper) {
445
446   double* matrix = initialize_matrix(matrix_order);
447   if(matrix) {
448     for(int i = 0; i < matrix_order; i++)
449       for(int j = 0; j < matrix_order; j++)
450         matrix[i*matrix_order+j] = (((double)rand()*(upper-lower))/(double)RAND_MAX
      +lower);
451   }
452     return matrix;
453
454 }
455
456
457 /*
458
459   Effettua la stampa della matrice passata in ingresso
460
461   @params:
462     double* matrix: Matrice da stampare
463     int matrix_order: Ordine della matrice quadrata
464
465   @return:
466     void
467
468 */
469
470 void print_matrix(double* matrix, int matrix_order) {
471
472   for(int i = 0; i < matrix_order; i++) {
473       for(int j = 0; j < matrix_order; j++)
474         printf(" [%lf]", matrix[i*matrix_order+j]);
475       printf("\n");
476   }
```

```
477
478 }
479
480
481 /*
482
483   Crea una griglia di processori di ordine grid_order e due
484   sotto griglie su quest'ultima, di cui una di riga e l'altra di colonna
485
486   @params:
487     MPI_Comm* grid: Griglia creata
488     MPI_Comm* sub_rgrid: Sotto griglia riga creata
489     MPI_Comm* sub_cgrid: Sotto griglia colonna creata
490     int mpi_rank: Rank del processore chiamante
491     int mpi_size: Numero di processori impiegati
492     int periods: Periodicita' della griglia
493     int reorder: Indica se riordinare il rank dei processori
494     int* coords: Coordinate assegnate al processore
495
496   @return:
497     void
498
499 */
500
501 void create_grid(
502   MPI_Comm* grid, MPI_Comm* sub_rgrid, MPI_Comm* sub_cgrid,
503   int mpi_rank, int mpi_size, int grid_order, int* periods, int reorder, int*
      coords) {
504
505   int dimensions[2] = {grid_order, grid_order};
506   MPI_Cart_create(MPI_COMM_WORLD, 2, dimensions, periods, reorder, grid);
507   MPI_Cart_coords(*grid, mpi_rank, 2, coords);
508   int remains[2] = {0, 1};
509   MPI_Cart_sub(*grid, remains, sub_rgrid);
510   remains[0] = 1;
511   remains[1] = 0;
512   MPI_Cart_sub(*grid, remains, sub_cgrid);
513
514 }
515
516
517 /*
518
```

```
519   Suddivide e invia la matrice passata in ingresso, dal processore 0 ai restanti
520
521   @params:
522     double* matrix: Matrice da inviare
523     int matrix_order: Ordine della matrice quadrata
524     int sub_matrix_order: Ordine della sotto matrice da inviare
525     MPI_Comm grid: Griglia di processori
526     int mpi_size: Numero di processori impiegati
527
528   @return:
529     void
530
531 */
532
533 void send_matrix_from_processor_0(
534   double* matrix, int matrix_order, int sub_matrix_order, MPI_Comm grid, int
      mpi_size) {
535
536   int coords[2];
537   int start_row;
538   int start_column;
539
540   for(int processor = 1; processor < mpi_size; processor++) {
541     MPI_Cart_coords(grid, processor, 2, coords);
542     start_row = coords[0] * sub_matrix_order;
543     start_column = coords[1] * sub_matrix_order;
544     for(int row_offset = 0; row_offset < sub_matrix_order; row_offset++)
545       MPI_Send(
546         &matrix[(start_row+row_offset)*matrix_order+start_column],
547         sub_matrix_order, MPI_DOUBLE, processor, D_TAG + processor, MPI_COMM_WORLD
548       );
549   }
550
551 }
552
553
554 /*
555
556   Riceve la sotto matrice inviata dal processore 0
557
558   @params:
559     double* sub_matrix: Sotto matrice
560     int sub_matrix_order: Ordine della sotto matrice quadrata
```

```
561     int mpi_rank: Rank del processore chiamante
562
563   @return:
564     void
565
566 */
567
568 void receive_matrix_from_processor_0(
569   double* sub_matrix, int sub_matrix_order, int mpi_rank) {
570
571   MPI_Status status;
572   for(int row = 0; row < sub_matrix_order; row++)
573     MPI_Recv(
574       &sub_matrix[row*sub_matrix_order], sub_matrix_order, MPI_DOUBLE,
575       0, D_TAG + mpi_rank, MPI_COMM_WORLD, &status
576     );
577
578 }
579
580
581 /*
582
583   Effettua il prodotto righe per colonne parallelo mediante la stregia BMR
584
585   @params:
586     double* sub_matrix_a: Sotto matrice A (sinistra) da impiegare nel prodotto
587     double* sub_matrix_b: Sotto matrice B (destra) da impiegare nel prodotto
588     double* sub_matrix_c: Sotto matrice C risultante
589     int sub_matrix_order: Ordine delle sotto matrice quadrata
590     MPI_Comm grid: Griglia di processori
591     MPI_Comm sub_rgrid: Sotto griglia di riga
592     MPI_Comm sub_cgrid: Sotto griglia di colonna
593     int grid_order: Ordine della griglia quadrata di processori
594     int coords[2]: Coordinate nella griglia del processore chiamante
595
596   @return:
597     void
598
599 */
600
601 void bmr(
602   double* sub_matrix_a, double* sub_matrix_b, double* sub_matrix_c, int
      sub_matrix_order,
```

```
603 MPI_Comm grid, MPI_Comm sub_rgrid, MPI_Comm sub_cgrid, int grid_order, int coords
      [2]) {
604
605   MPI_Request request;
606   MPI_Status status;
607
608   int sub_matrix_a_broadcaster_coords[2];
609   int sub_matrix_a_broadcaster_rank;
610
611   /* Matrice di appoggio per il broadcasting */
612
613   double* tmp_matrix = initialize_matrix(sub_matrix_order);
614
615   /* Calcolo rank sotto griglia colonna del receiver della sotto matrice B */
616
617   int sub_matrix_b_receiver_rank;
618   int sub_matrix_b_receiver_coords[2] = {
619     (coords[0] + grid_order - 1) % grid_order,
620     coords[1]
621   };
622   MPI_Cart_rank(sub_cgrid, sub_matrix_b_receiver_coords, &
        sub_matrix_b_receiver_rank);
623
624   /* Calcolo rank sotto griglia colonna del sender della sotto matrice B */
625
626   int sub_matrix_b_sender_rank;
627   int sub_matrix_b_sender_coords[2] = {
628     (coords[0] + 1) % grid_order,
629     sub_matrix_b_sender_coords[1] = coords[1]
630   };
631     MPI_Cart_rank(sub_cgrid, sub_matrix_b_sender_coords, &sub_matrix_b_sender_rank)
        ;
632
633     /* Calcolo rank nella sotto griglia colonna del processore chiamante */
634
635     int rank_cgrid;
636     MPI_Cart_rank(sub_cgrid, coords, &rank_cgrid);
637
638   /* Inizio BMR */
639
640   for(int step = 0; step < grid_order; step++) {
641
642     /* Coordinate del processore che deve inviare la sotto matrice A */
```

```
643
644     sub_matrix_a_broadcaster_coords[0] = coords[0];
645         sub_matrix_a_broadcaster_coords[1] = (coords[0] + step) % grid_order;
646
647
648     if(!step) {    // Primo passo
649
650       /* Broadcasting */
651
652       if(coords[0] == coords[1]){
653                 sub_matrix_a_broadcaster_coords[1] = coords[1];
654                 memcpy(tmp_matrix, sub_matrix_a, sub_matrix_order*sub_matrix_order*
    sizeof(double));
655             }
656
657       MPI_Cart_rank(sub_rgrid, sub_matrix_a_broadcaster_coords, &
    sub_matrix_a_broadcaster_rank);
658
659             MPI_Bcast(tmp_matrix, sub_matrix_order*sub_matrix_order,
660         MPI_DOUBLE, sub_matrix_a_broadcaster_rank, sub_rgrid
661       );
662
663       /* Multiply */
664
665             multiply(tmp_matrix, sub_matrix_b, sub_matrix_c, sub_matrix_order);
666
667     } else {  // Passi successivi
668
669       /* Broadcasting sulle diagonale superiori alla principale (k + step) */
670
671             if(coords[1] == sub_matrix_a_broadcaster_coords[1]){
672               memcpy(tmp_matrix, sub_matrix_a, sub_matrix_order*sub_matrix_order*
    sizeof(double));
673             }
674             sub_matrix_a_broadcaster_rank = (sub_matrix_a_broadcaster_rank+1)%
    grid_order;
675
676             MPI_Bcast(tmp_matrix, sub_matrix_order*sub_matrix_order,
677         MPI_DOUBLE, sub_matrix_a_broadcaster_rank, sub_rgrid
678       );
679
680             /* Rolling */
681
```

```
682             MPI_Isend(sub_matrix_b, sub_matrix_order*sub_matrix_order,
683         MPI_DOUBLE, sub_matrix_b_receiver_rank,
684         D_TAG + sub_matrix_b_receiver_rank, sub_cgrid, &request
685       );
686
687             MPI_Recv(sub_matrix_b, sub_matrix_order*sub_matrix_order,
688         MPI_DOUBLE, sub_matrix_b_sender_rank, D_TAG + rank_cgrid, sub_cgrid, &
      status
689       );
690
691             /* Multiply */
692
693             multiply(tmp_matrix, sub_matrix_b, sub_matrix_c, sub_matrix_order);
694
695     }
696
697   }
698
699 }
700
701
702 /*
703
704   Unisce tutte le sotto matrici calcolate dai singoli processori in un'unica
        matrice C
705
706   @params:
707     double* matrix_c: Matrice risultate dall'operazione di unione
708     int matrix_order: Ordine della matrice quadrata risultante
709     double* sub_matrix_c: Sotto matrice da inserire in quella finale
710     int sub_matrix_order: Ordine della sotto matrice quadrata
711     MPI_Comm grid: Griglia di processori
712     int mpi_size: Numero di processori impiegati
713     int mpi_rank: Rank del processore chiamante
714
715   @return:
716     void
717
718 */
719
720 void merge(
721   double* matrix_c, int matrix_order, double* sub_matrix_c,
722   int sub_matrix_order, MPI_Comm grid, int mpi_size, int mpi_rank) {
```

```
723
724   MPI_Status status;
725   int start_row;
726   int start_columns;
727   int coords[2];
728
729   if(!mpi_rank) {  // Se il chiamante e' il processore 0 allora si riceve
730
731     /* Si scorre su tutti i processori */
732
733     for(int processor = 0; processor < mpi_size; processor++) {
734
735       MPI_Cart_coords(grid, processor, 2, coords);
736       start_row = coords[0] * sub_matrix_order;
737       start_columns = coords[1] * sub_matrix_order;
738
739       if(processor) {  // Se non e' il processore 0 allora si riceve
740         for(int row_offset = 0; row_offset < sub_matrix_order; row_offset++) {
741           MPI_Recv(
742             &sub_matrix_c[row_offset*sub_matrix_order], sub_matrix_order,
743             MPI_DOUBLE, processor, D_TAG + processor, MPI_COMM_WORLD, &status
744           );
745           memcpy(
746             &matrix_c[(start_row+row_offset)*matrix_order+start_columns],
747             &sub_matrix_c[row_offset*sub_matrix_order],
748             sub_matrix_order*sizeof(double)
749           );
750         }
751       } else {  // Altrimenti si copia la propria sotto matrice C nella matrice C
    finale
752         for(int row_offset = 0; row_offset < sub_matrix_order; row_offset++)
753           memcpy(
754             &matrix_c[row_offset*matrix_order],
755             &sub_matrix_c[row_offset*sub_matrix_order],
756             sub_matrix_order*sizeof(double)
757           );
758       }
759
760     }
761
762   } else {   // Altrimenti si invia al processore 0
763
764     for(int row_offset = 0; row_offset < sub_matrix_order; row_offset++)
```

```
765       MPI_Send(
766         &sub_matrix_c[row_offset*sub_matrix_order], sub_matrix_order,
767         MPI_DOUBLE, 0, D_TAG + mpi_rank, MPI_COMM_WORLD
768       );
769
770   }
771
772 }
773
774
775 /*
776
777   Effettua Il prodotto righe per colonne delle matrici quadrate passate in ingresso
778
779   @params:
780     double* matrix_a: Matrice A (sinistra)
781     double* matrix_b: Matrice B (destra)
782     double* matrix_c: Risultato del prodotto
783     int matrix_order: Ordine delle matrici quadrate
784
785   @return:
786     void
787
788 */
789
790 void multiply(double* matrix_a, double* matrix_b, double* matrix_c, int
      matrix_order) {
791
792   for(int i = 0; i < matrix_order; i++)
793     for(int j = 0; j < matrix_order; j++)
794       for(int k = 0; k < matrix_order; k++)
795         matrix_c[i*matrix_order+j] += (matrix_a[i*matrix_order+k] * matrix_b[k*
      matrix_order+j]);
796
797 }
```