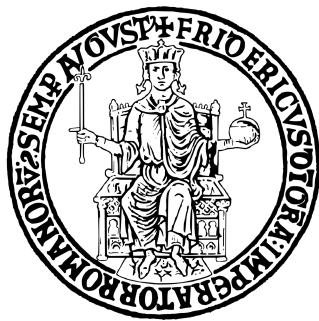


UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

SCUOLA POLITECNICA E DELLE SCIENZE DI BASE

DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELLE TECNOLOGIE DELL'INFORMAZIONE



CORSO DI LAUREA MAGISTRALE IN INFORMATICA

INSEGNAMENTO DI PARALLEL AND DISTRIBUTED COMPUTING

ANNO ACCADEMICO 2023/2024

**Sviluppo 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

Studenti

Francesco Jr. Iaccarino – N97000440  
Fabiola Salomone – N97000457

Questa pagina è stata lasciata intenzionalmente bianca.

# INDICE

<b>1. Definizione ed Analisi del problema.....</b>	<b>6</b>
<b>2. Descrizione dell'algoritmo .....</b>	<b>8</b>
2.1 Descrizione dell'algoritmo in fasi .....	8
<b>3 Input, Output ed errori.....</b>	<b>10</b>
3.1 Input.....	10
3.2 Restrizioni .....	11
3.3 Output.....	11
<b>4 Descrizione delle Subroutine .....</b>	<b>15</b>
<b>4.1 Subroutine personalizzate .....</b>	<b>15</b>
4.1.1 Subroutine – getRandomDoubleNumberInRange .....	15
4.1.2 Subroutine –getMatrixOfRandomNumbersOfSize.....	16
4.1.3 Subroutine –printSquareMatrix .....	17
4.1.4 Subroutine –parseInt.....	18
4.1.5 Subroutine –isPerfectSquare .....	19
4.1.6 Subroutine –mod.....	20
4.1.7 Subroutine – distributeMatrixes .....	21
4.1.8 Subroutine – receiveMatrix .....	23
4.1.9 Subroutine – getPartialMatrixResult.....	24
4.1.10 Subroutine – printResult .....	26
4.1.11 Subroutine – validateInput.....	27
<b>3.1 Subroutine MPI.....</b>	<b>29</b>
3.1.1 Routine – MPI_Init .....	29
3.1.2 Routine – MPI_Comm_rank.....	30
3.1.3 Routine – MPI_Comm_size .....	30
3.1.4 Routine – MPI_Bcast .....	31
3.1.5 Routine – MPI_Wtime.....	32
3.1.6 Routine – MPI_Reduce .....	32
3.1.7 Routine – MPI_Send.....	33
3.1.8 Routine – MPI_Recv.....	33
3.1.9 Routine – MPI_Barrier.....	34
3.1.10 Routine – MPI_Finalize .....	34
3.1.11 Routine – MPI_Cart_create .....	35
3.1.12 Routine – MPI_Cart_coords .....	35
3.1.13 Routine – MPI_Cart_rank .....	36
<b>5. Esempi d'uso .....</b>	<b>37</b>
5.1 Script PBS utilizzato per l'esecuzione dell'algoritmo .....	37
5.1 Esempi d'uso .....	39
5.1.1 Esempi 1 – (A ∈ R^(4X4) e B ∈ R^(4X4)) .....	39

5.1.2 Esempi 2 – ERROR - Number of processors provided cannot be distributed in a NN grid. number of processors:N.....	41
5.1.3 Esempi 3 – Provide only one argument: the number of rows (and columns) of the matrices.....	42
5.1.4 Esempi 4 – Errore - Matrix size must be >= of the grid and a multiple of number of processor matrixsize : N numberOfProcessors : M .....	44
5.1.5 Esempi 5 – Errore - Cannot parse the matrixSize with value provided. matrixSize : ABC.....	46
5.1.6 Esempi 6 – (A ∈ R^(2X2) e B ∈ R^(2X2)) .....	47
<b>6 Analisi delle performance .....</b>	<b>50</b>
6.4.1 Analisi sul numero di processori .....	51
6.4.2 Calcolo dello speed-up.....	52
6.4.3 Calcolo dell'efficienza.....	53
6.4.4 Conclusioni .....	54
<b>7 Codice Sorgente.....</b>	<b>57</b>
7.1 main.c.....	57
7.2 untils.h.....	57
7.3 untils.c .....	64
7.4 job-script.h .....	66



# 1. Definizione ed Analisi del problema

Si vuole sviluppare un algoritmo per il calcolo del prodotto matrice-matrice, in ambiente di calcolo parallelo su architettura MIMD (Multiple Instruction stream Multiple Data) a memoria distribuita, che utilizzi la libreria MPI.

L'obiettivo dell'algoritmo è quello di eseguire il prodotto scalare tra due matrici:  $A \in R^{(NxN)}$  e  $B \in R^{(NxN)}$  e restituire in output una matrice  $C \in R^{(NxN)}$ , le cui componenti sono ottenute come segue:

$$c_{i,j} = \sum_{k=1}^m a_{i,k} * b_{k,j}$$

L'algoritmo dovrà ricevere in input un unico parametro :

- Il numero  $n$  di righe e colonne delle matrici A, B, C

Si assume che le dimensioni di tali matrici siano quadrate e uguali.

L'algoritmo è composto da due parti principali: la prima parte, formata da uno script PBS, in cui vi è la raccolta del parametro in ingresso che è la grandezza della matrice da generare. Tale grandezza, ha il vincolo di essere un multiplo del numero dei processori utilizzati.

La seconda parte, performata mediante un algoritmo in C che prevede l'implementazione di una topologia, la generazione di una matrice di dimensione presa in input e l'applicazione della strategia Broadcast Multiply Rolling - BMR per il calcolo del prodotto matrice-matrice.

Una **topologia** è un’astrazione di tipo logico nella quale si immaginano i processori disposti secondo un determinato ordine, che può non coincidere con l’ordinamento fisico degli stessi.

In particolar modo la topologia adottata durante lo sviluppo di tale algoritmo è quella a *griglia bidimensionale* con dimensione  $p \times p$ .

Successivamente si analizzano come variano i tempi al variare della dimensione del problema.

## 2. Descrizione dell'algoritmo

In questo capitolo, viene descritto l'algoritmo implementato per il calcolo del prodotto matrice-matrice in ambiente di calcolo parallelo su architettura MIMD a memoria distribuita.

### 2.1 Descrizione dell'algoritmo in fasi

L'algoritmo che si presta al calcolo del prodotto matrice – matrice è strutturato in sette fasi distinte:

1. **Fase 1:** Inizializzazione e Validazione degli input;
2. **Fase 2:** Creazione della Griglia di Processori;
3. **Fase 3:** Distribuzione delle Matrici Casuali;
4. **Fase 4:** Calcolo del Prodotto Parziale;
5. **Fase 5:** Stampa del Tempo Trascorso;
6. **Fase 6:** Deallocazione della Memoria.

Nella prima fase l'algoritmo comincia inizializzando l'ambiente MPI, ottenendo l'ID del processo e il numero totale di processori. Viene poi eseguita la validazione degli input, assicurandosi che ci sia un solo argomento (la dimensione delle matrici) e che il numero di processori sia un multiplo della dimensione delle matrici, così rendendo possibile la creazione di una griglia bidimensionale di processori.

Successivamente viene creata una griglia di processori bidimensionale utilizzando la funzione ‘MPI\_Cart\_create’. Questa griglia viene utilizzata in seguito, per distribuire due matrici casuali generate dal processo master, tra tutti i processori. Le matrici vengono inviate in modo coordinato tramite la funzione ‘MPI\_Send’ utilizzando le coordinate cartesiane dei processori nella griglia.

Ogni processore riceve porzioni delle due matrici distribuite e calcola il prodotto parziale della matrice utilizzando la funzione ‘getPartialMatrixResult’. Durante questo calcolo le matrici vengono suddivise e inviate tra i processori della stessa riga e colonna mediante operazioni di comunicazione ‘MPI\_Send’ e ‘MPI\_Recv’.

Seguentemente, ogni processore invia il proprio risultato parziale al processo master, che riceve e memorizza il risultato parziale completo, dopodiché il processo master stampa il risultato completo utilizzando la funzione ‘printResult’.

Infine, viene calcolato il tempo totale di esecuzione dell’algoritmo come la differenza tra l’istante finale e l’istante di iniziale. Il tempo massimo tra tutti i processori è calcolato utilizzando la routine ‘MPI\_Reduce’, e il risultato viene stampato solo dal processore master.

In sintesi, il codice implementa la moltiplicazione di matrici in parallelo, sfruttando la comunicazione tra processori per distribuire dati e calcolare il prodotto parziale in modo coordinato. L’utilizzo di MPI consente di sfruttare al massimo le risorse disponibili migliorano le prestazioni dell’algoritmo rispetto a un’implementazione sequenziale.

# 3 Input, Output ed errori

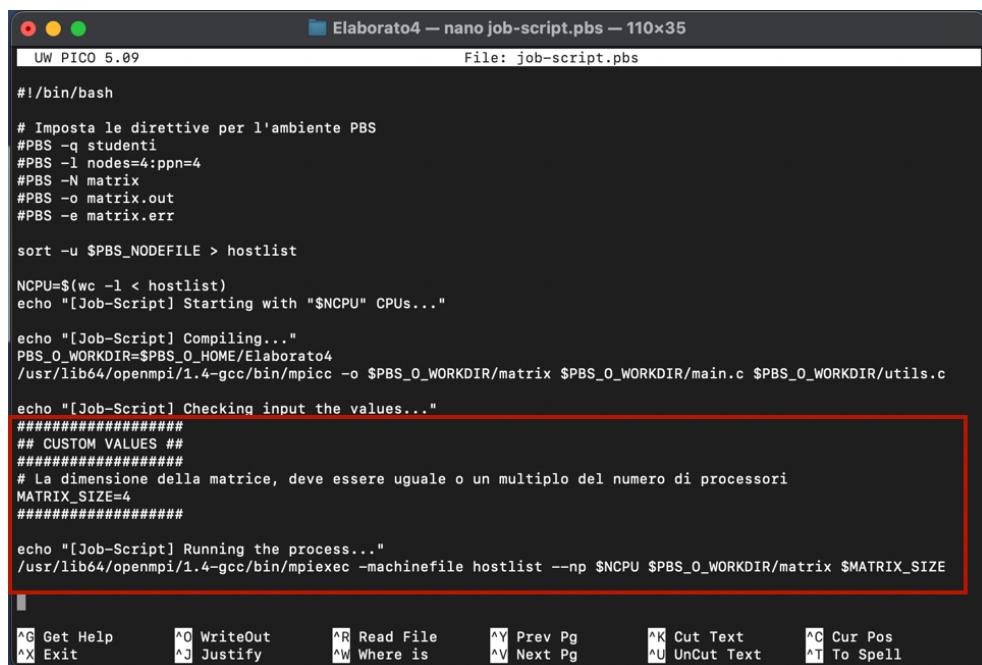
In questo capitolo, saranno esaminati gli aspetti riguardanti l'input, l'output e i potenziali errori associati all'esecuzione dell'algoritmo. Si affrontano dettagli relativi alla gestione degli input forniti dall'utente, alla presentazione dei risultati attraverso l'output, e alle procedure atte a gestire eventuali errori durante l'esecuzione dell'algoritmo.

## 3.1 Input

Come descritto nel primo capitolo, il parametro richiesto dall'algoritmo è seguente:

- Il numero  $N$  che sarà la grandezza della matrice  $N \times N$ , ed anche un multiplo del numero di processori utilizzati per l'esecuzione in ambiente parallelo.

Per comprendere meglio dove inserire tale parametro, si mostra (figura) di seguito il punto esatto in cui va inserito il parametro nel file PBS:



```
UW PICO 5.09 File: job-script.pbs
#!/bin/bash

# Imposta le direttive per l'ambiente PBS
#PBS -q studenti
#PBS -l nodes=4:ppn=4
#PBS -N matrix
#PBS -o matrix.out
#PBS -e matrix.err

sort -u $PBS_NODEFILE > hostlist

NCPU=$(wc -l < hostlist)
echo "[Job-Script] Starting with \"$NCPU\" CPUs..."

echo "[Job-Script] Compiling..."
PBS_O_WORKDIR=$PBS_O_HOME/Elaborato4
/usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS_O_WORKDIR/matrix $PBS_O_WORKDIR/main.c $PBS_O_WORKDIR/utils.c

echo "[Job-Script] Checking input the values..."
#####
## CUSTOM VALUES ##
#####
# La dimensione della matrice, deve essere uguale o un multiplo del numero di processori
MATRIX_SIZE=4
#####

echo "[Job-Script] Running the process..."
/usr/lib64/openmpi/1.4-gcc/bin/mpirun --np $NCPU $PBS_O_WORKDIR/matrix $MATRIX_SIZE
```

Cioè:

```
1. #####
2. ## CUSTOM VALUES ##
3. #####
4. # La dimensione della matrice, deve essere uguale o un multiplo del numero di processori
7. #####
8. COLUMN=4
9. #####
10. echo "[Job-Script] Running the process..."
11.
12. /usr/lib64/openmpi/1.4-gcc/bin/mpexec -machinefile hostlist --np $NCPU $PBS_O_WORKDIR/matrix
    $MATRIX_SIZE
```

## 3.2 Restrizioni

I parametri devono rispettare determinate restrizioni/costrizioni:

- La dimensione  $N$  deve essere un multiplo del numero di righe (o colonne) della griglia dei processori.

Vengono inoltre eseguiti controlli su tutti i valori passati come parametri, in caso di qualsiasi violazione di queste restrizioni, il programma genererà e mostrerà un messaggio a video (su terminale) indicando l'errore corrispondente.

## 3.3 Output

Vediamo ora alcuni esempi di output che ci possiamo aspettare rispetto ad alcuni input di esempio.

L'output dell'algoritmo, si presenta nella seguente forma:

**First random matrix :**

*<valori prima matrice >*

**Second random matrix :**

*<valori seconda matrice >*

**RESULT :**

*<valori matrice risultato >*

**Time elapsed :** *<tempo in secondi> seconds*

Vediamo ora all'atto pratico alcuni esempi di output che possiamo aspettarci in caso di determinati input.

Il numero di processi utilizzato nell'esempio è 4. La matrice A nell'algoritmo è inizializzata con i valori da 1 a NxN, mentre la matrice B è inizializzata con i valori da NxN a 1. L'input corrisponde al parametro N:

- **INPUT:** 2
- **OUTPUT:**

```
First random matrix:  
10.252473 2.021238 0.203122 8.353641  
5.691431 6.226150 6.079395 2.862472  
10.330067 9.746447 10.636822 2.942036  
11.113232 11.553806 7.363888 14.515747
```

```
Second random matrix:  
2.940140 7.048192 2.477590 12.167706  
6.283752 0.498549 11.122838 14.479766  
7.465564 2.708370 8.167510 13.684550  
3.690113 12.417508 7.392629 13.942587
```

```
RESULT:  
107.539922 97.404829 108.204738 192.654751  
288.806102 198.133285 270.601773 355.942738  
317.849874 209.304715 281.056398 415.509577  
263.138099 243.378954 287.311423 360.172110
```

```
Time elapsed: 0.000419 seconds
```

Il numero di processi utilizzato nell'esempio è 1. La matrice A nell'algoritmo è inizializzata con i valori da 1 a NxN, mentre la matrice B è inizializzata con i valori da NxN a 1. L'input corrisponde al parametro N:

- **INPUT:** 3
- **OUTPUT:**

```
First random matrix:  
2.113276 4.820949 14.772604  
6.136221 4.890598 5.249949  
0.416120 11.223005 13.164731
```

```
Second random matrix:  
6.160282 8.898859 12.693720  
7.213221 4.138748 13.521164  
5.275416 4.419135 12.342865
```

```
RESULT:  
32.614158 107.059148 157.215159  
99.072563 291.759602 442.792812  
50.010047 134.651001 224.286859
```

```
Time elapsed: 0.000036 seconds
```

Il numero di processi utilizzato nell'esempio è 4. La matrice A nell'algoritmo è inizializzata con i valori da 1 a NxN, mentre la matrice B è inizializzata con i valori da NxN a 1. L'input corrisponde al parametro N:

- **INPUT:** 10
- **OUTPUT:**

```
ERROR – Matrix size must be >= of the grid and  
a multiple of number of processors.  
matrixSize:10  
number0fProcessors:4
```

Il numero di processi utilizzato nell'esempio è 1. La matrice A nell'algoritmo è inizializzata con i valori da 1 a NxN, mentre la matrice B è inizializzata con i valori da NxN a 1. L'input corrisponde al parametro N:

- **INPUT:** ABF
- **OUTPUT:**

```
ERROR – Cannot parse the matrixSize with value  
provided.  
matrixSize:ABF
```

Il numero di processi utilizzato nell'esempio è 1. La matrice A nell'algoritmo è inizializzata con i valori da 1 a NxN, mentre la matrice B è inizializzata con i valori da NxN a 1. L'input corrisponde al parametro N:

- **INPUT:** 4
- **OUTPUT:**

First random matrix:

1.356461	2.688487	9.688815	13.913293
4.659664	10.902225	14.072307	2.025361
1.114079	10.958447	8.069032	4.141256
0.872076	9.328972	5.197837	9.926756

Second random matrix:

6.067318	7.783601	0.675695	13.164677
14.279775	10.735852	7.868346	11.122619
4.636614	7.875941	1.404162	11.708170
11.103472	5.807930	7.555573	12.459933

RESULT:

93.271043	103.469995	142.337749	194.318285
96.807607	299.824633	369.451852	329.058118
185.725983	243.824255	318.477624	400.976631
138.747424	182.822596	316.101530	353.306830

Time elapsed: 0.000055 seconds

Tutti gli output verranno trascritti all'interno del file “*matrix.out*” mentre eventuali errori relativi a compilazione o esecuzione verranno trascritti nel file “*matrix.err*” (non sono incisi i messaggi di violazione delle restrizioni degli input che sono invece presenti nel file “*matrix.out*”).

## 4 Descrizione delle Subroutine

In questo capitolo vengono descritte le subroutine utilizzate nell'algortimo, queste sono documentate all'interno del codice sorgente attraverso commenti che hanno la seguente forma:

- Breve descrizione delle funzioni;
- Lista dei parametri con descrizione dettagliata del loro utilizzo;
- Se presente un output, a seconda delle diverse condizioni possibili, viene descritta la sua forma.

Sono quindi riportati i prototipi delle funzioni, la loro documentazione interna ed eventualmente una descrizione aggiuntiva.

### 4.1 Subroutine personalizzate

Di seguito, verranno descritte alcune subroutine personalizzate utilizzate nel progetto.

#### 4.1.1 Subroutine – getRandomDoubleNumberInRange

La funzione “getRandomDoubleNumberInRange” si occupa dell’allocazione di matrici e vettori con generatore casuale.

Il prototipo della funzione è il seguente:

```
double getRandomDoubleNumberInRange(int min, int max);
```

Descriviamo i parametri presi in input dalla funzione:

- min: il valore minimo dell’intervallo desiderato(incluso).
- max: il valore massimo dell’intervallo desiderato (escluso).

La funzione restituisce un numero casuale nel range definito.

In pratica, la funzione sfrutta ‘rand()’ per generare un numero causale tra 0 e ‘RAND-MAX’, lo scala in un intervallo tra 0 e 1 e quindi lo trasforma nell’intervallo specificato ‘[min, max]’. Ad esempio, se chiamiamo la

funzione con ‘getRandomDoubleNumberInRange(5,10)’, il risultato sarà un numero casuale di tipo ‘double’ compreso tra 5 (incluso) e 10 (escluso)

#### 4.1.2 Subroutine –**getMatrixOfRandomNumbersOfSize**

La funzione “ getMatrixOfRandomNumbersOfSize” alloca una matrice di dimensione column \*row con valori casuali nello specifico è progettata per generare e restituire una matrice di numeri casuali in virgola mobile di dimensione specificata.

Il prototipo della funzione è il seguente:

```
double** getMatrixOfRandomNumbersOfSize(int column, int  
                                         row, int min, int max);
```

Descriviamo i parametri presi in input dalla funzione:

- column: rappresenta il numero di colonne
- row: rappresenta il numero di righe
- min: il valore minimo per i numeri casuali nella matrice
- max: il valore massimo per i numeri casuali nella matrice

Funzionamento: La funzione inizia allocando dinamicamente la memoria per un array di puntatori double di dimensione ‘column’. Ogni elemento dell’array è un puntatore a un array di double. Successivamente in un ciclo ‘for’, vengono allocati dinamicamente gli array di double per ciascuna colonna della matrice. Ogni array ha dimensione *row*.

All’interno di un secondo ciclo ‘for’ ogni elemento della matrice viene popolato con un numero casuale in virgola mobile ottenuto chiamando la funzione ‘getRandomDoubleNumberInRange(min, max)’.

Una volta completata la generazione della matrice, essa viene restituita come risultato della funzione.

Risultato: La funzione restituisce una matrice di numeri casuali in virgola mobile di dimensione specifiche, dove ciascun numero è generato nell'intervallo ‘[min, max]’.

Per il codice della funzione si veda il capitolo dal nome “Codice” con sottoparagrafo dal nome “Utils.c”

#### 4.1.3 Subroutine –printSquareMatrix

La funzione “ printSquareMatrix ” è progettata per stampare una matrice quadrata di numeri in virgola mobile su standard output.

Il prototipo della funzione è il seguente:

```
void printSquareMatrix(double** matrix, int size);
```

Descriviamo i parametri presi in input dalla funzione:

- matrix: una matrice quadrata di numeri in virgola mobile
- size: la dimensione della matrice, ovvero il numero di righe e colonne

Funzionamento: Utilizzando due cicli ‘for’, uno annidato dentro l’altro, la funzione attraversa ogni elemento della matrice.

Per ogni elemento viene utilizzata la funzione ‘printf’ per stampare il valore del numero in virgola mobile seguito da uno spazio.

Dopo aver completato una riga della matrice, viene stampato un carattere di nuova linea, spostando così la stampa sulla riga successiva.

Risultato: La funzione non restituisce alcun valore, (tipo di ritorno ‘void’), ma ha l’effetto di stampare la matrice fornita su standard output.

Per il codice della funzione si veda il capitolo dal nome “Codice” con sottoparagrafo dal nome “Utils.c”.

#### 4.1.4 Subroutine –parseInt

La funzione “ parseInt ” è progettata per convertire una stringa rappresentante un numero intero in un valore intero.

Il prototipo della funzione è il seguente:

```
bool parseInt(char* arg, int* val);
```

Descriviamo i parametri presi in input dalla funzione:

- arg: rappresenta la stringa da convertire
- val: un puntatore a un intero, dove verrà memorizzato il risultato della conversione

Funzionamento: La funzione utilizza la funzione ‘strtol’ per convertire la stringa in un numero intero a ‘long’.

La variabile ‘temp’ è un puntatore a carattere che memorizzerà il primo carattere non convertito dalla stringa. La variabile ‘ret’ contiene il risultato della conversione della stringa in un numero a ‘long’.

Viene verificato se la conversione è avvenuta con successo. Ciò include la verifica che la stringa non sia vuota, che tutti i caratteri siano stati utilizzati nella conversione e che non ci siano errori di overflow o underflow.

Se la conversione non è riuscita, la variabile ‘result’ viene impostata su ‘false’, altrimenti, il risultato viene memorizzato nella variabile puntata da ‘val’.

Infine, la funzione restituisce il valore booleano ‘result’ indicando se la conversione è stata eseguita con successo.

Risultato: La funzione restituisce ‘true’ se la conversione è riuscita e memorizza il risultato nella variabile puntata da ‘val’, mentre restituisce ‘false’ se la conversione non è riuscita.

Per il codice della funzione si veda il capitolo dal nome “Codice” con sottoparagrafo dal nome “Utils.c”

#### 4.1.5 Subroutine –isPerfectSquare

La funzione “ isPerfectSquare ” è progettata per verificare se un dato intero è un quadrato perfetto.

Il prototipo della funzione è il seguente:

```
bool isPerfectSquare(int number);
```

Descriviamo i parametri presi in input dalla funzione:

- number: rappresenta il valore intero da verificare se è un quadrato perfetto

Funzionamento: La funzione inizia calcolando la radice quadrata intera (‘s’) del numero dato utilizzando la funzione ‘sqrt’ della libreria matematica standard di c.

Successivamente, la funzione restituisce il risultato della condizione booleana: il prodotto della radice quadrata ('s') per se stessa è uguale al numero originale ('number').

- Se l'uguaglianza è vera, la funzione restituisce ‘true’ indicando che il numero è un quadrato perfetto.
- Se l'uguaglianza è falsa, la funzione restituisce ‘false’, indicando che il numero non è un quadrato perfetto.

Risultato: La funzione restituisce un valore booleano (‘true’ o ‘false’) che indica se il numero dato è un quadrato perfetto o meno.

Per il codice della funzione si veda il capitolo dal nome “Codice” con sottoparagrafo dal nome “Utils.c”

#### 4.1.6 Subroutine –mod

La funzione “ isPerfectSquare ” è progettata per calcolare il resto della divisione intera di due numeri interi, gestendo correttamente i casi in cui il risultato è negativo.

Il prototipo della funzione è il seguente:

```
int mod(int a, int b);
```

Descriviamo i parametri presi in input dalla funzione:

- a: rappresenta il dividendo
- b: rappresenta il divisore

Funzionamento: La funzione calcola il resto della divisione intera di ‘a’ per ‘b’ utilizzando l’operatore modulo ‘%’.

Successivamente, viene utilizzata una condizione ternaria per verificare se il risultato ('r') è negativo ('r<0'):

- Se il resto è negativo, viene aggiunto il divisore ('b') per garantire che il risultato finale sia non negativo.
- Se il resto è già non negativo, il valore di 'r' rimane inalterato.

Infine, il risultato corretto viene restituito.

Risultato: La funzione restituisce il resto della divisione intera di 'a' per 'b', garantendo che il risultato sia sempre non negativo.

Per il codice della funzione si veda il capitolo dal nome “Codice” con sottoparagrafo dal nome “Utils.c”.

#### 4.1.7 Subroutine – distributeMatrixes

La funzione “distributeMatrixes” è progettata per distribuire le matrici generate casualmente fra tutti i processori, compreso il chiamante.

Il prototipo della funzione è il seguente:

```
void distributeMatrixes(int matrixSize, int gridSize,
                      MPI_Comm mpi_comm);
```

Descriviamo i parametri presi in input dalla funzione:

- matrixSize: rappresenta la dimensione della matrice da generare casualmente
- gridSize: rappresenta la dimensione della matrice di processori

- `mpi_comm`: è il Communicator, che deve essere una griglia di dimensione `gridSize`.

Funzionamento: La funzione inizia inizializzando il ‘seme’ (un valore iniziale utilizzato per inizializzare l’algoritmo di generazione di numeri casuali) dei valori casuali utilizzando la funzione ‘`srad(time(NULL))`’

Successivamente, genera due matrici casuali chiamando la funzione ‘`getMatrixRandomNumbersOfSize`’, una per la matrice ‘`firstMatrix`’ e una per ‘`secondMatrix`’.

Stampa le matrici casuali generate utilizzando la funzione ‘`printSquareMatrix`’.

La distribuzione delle porzioni della matrice viene eseguita utilizzando una doppia iterazione attraverso la griglia di processori. Per ogni processore, la funzione calcola le coordinate sulla griglia (‘`receiverProcessor`’) utilizzando la funzione ‘`MPI_Cart_rank`’.

Successivamente, utilizza una doppia iterazione per inviare porzioni della matrice ‘`firstMatrix`’ e ‘`secondMatrix`’ al processore destinatario utilizzando la funzione ‘`MPI_Send`’. Vengono utilizzati due tag distinti (‘`TAG_MATRIX_FIRST`’ e ‘`TAG_MATRIX_SECOND`’) per identificare i valori.

Dopo aver completato l’invio delle porzioni, libera la memoria allocata per le matrici casuali utilizzando la funzione ‘`free`’.

Risultato: La funzione distribuisce porzioni delle matrici casuali generate tra tutti i processori nella griglia MPI, preparandoli per l’esecuzione parallela dell’algoritmo di moltiplicazione di matrice.

Per il codice della funzione si veda il capitolo dal nome “Codice” con sottoparagrafo dal nome “Utils.c”.

#### 4.1.8 Subroutine – receiveMatrix

La funzione “receiveMatrix” è progettata per ricevere una matrice inviata con un determinato tag.

Il prototipo della funzione è il seguente:

```
double** receiveMatrix(int matrixSize, int tag, MPI_Comm  
mpi_comm);
```

Descriviamo i parametri presi in input dalla funzione:

- matrixSize: rappresenta il numero di righe e colonne della matrice ‘MxM’ che si prevede di ricevere
- tag: rappresenta il tag specifico utilizzato per identificare il tipo di messaggio da ricevere
- mpi\_comm: è il Communicator MPI

Funzionamento: La funzione alloca dinamicamente una matrice bidimensionale (‘matrix’) delle dimensioni specificate da ‘matrixSize’

Successivamente, utilizza un doppio ciclo ‘for’ per iterare attraverso le righe e le colonne della matrice e riceve i valori da MPI utilizzando la funzione ‘MPI\_Recv’

- La ricezione avviene dal processo con ID ‘MASTER\_ID’

- Il tag specificato ('tag?) identifica il tipo di messaggio da ricevere, garantendo che corrisponda al tag utilizzato per l'invio

Ogni elemento della matrice viene ricevuto e memorizzato nella corrispondente posizione della matrice allocata dinamicamente.

Risultato: La funzione restituisce la matrice ricevuta, alloca dinamicamente e popolata con i valori ricevuti attraverso MPI .

Per il codice della finzione si veda il capitolo dal nome “Codice” con sottoparagrafo dal nome “Utils.c”.

#### 4.1.9 Subroutine – `getPartialMatrixResult`

La funzione “`getPartialMatrixResult`” è progettata per effettuare un prodotto parziale matrice per matrice.

Il prototipo della funzione è il seguente:

```
double** getPartialMatrixResult(int numberOfRows, int
gridSize, double * startTime, MPI_Comm mpi_comm_grid);
```

Descriviamo i parametri presi in input dalla funzione:

- `numberOfValues`: rappresenta il numero di righe e colonne della porzione di matrice su cui ogni processore effettuerà il calcolo
- `gridSize`: dimensione della griglia MPI, rappresenta il numero totale di processori lungo un lato della griglia (assumendo una griglia quadrata)
- `mpi_comm_grid`: è il Communicator MPI che rappresenta la griglia virtuale dei processori.

Funzionamento: Il processo identifica le sue coordinate all'interno della griglia MPI utilizzando la funzione ‘MPI\_CART\_coords’.

Vengono ricevute due porzioni di matrice (‘matrixOne’ e ‘matrixTwo’) attraverso la funzione ‘receiveMatrix’. Queste porzioni rappresentano i due fattori della moltiplicazione di matrici.

Viene allocata dinamicamente la matrice ‘result’ che conterrà il prodotto parziale.

La funzione sincronizza tutti i processori mediante ‘MPI\_Barrier’ e memorizza l'istante di tempo iniziale con ‘MPI\_Wtime’.

Vengono eseguite iterazioni per calcolare il prodotto parziale:

- I processori diagonalmente allineati inviano la loro porzione di matrice ‘matrixOne’ alle righe corrispondenti nella griglia.
- I processori non diagonalmente allineati ricevono una porzione di matrice ‘matrixOne’ dalla diagonale
- Vengono scambiate le porzioni di matrice ‘matrixTwo’ tra processori adiacenti nella griglia.
- Viene effettuato il calcolo del prodotto parziale moltiplicando la porzione di ‘matrixOne’ ricevuta per ‘matrixTwo’
- Le matrici temporanee non più necessarie vengono liberate dalla memoria.

Alla fine, la funzione restituisce la matrice ‘result’ contenente il prodotto parziale.

Risultato: La funzione restituisce un puntatore a una matrice di tipo ‘double’.

Nota: La sincronizzazione mediante ‘MPI\_Barrier’ e la misurazione del tempo sono utilizzate per calcolare il tempo trascorso nell’esecuzione dell’algoritmo

Per il codice della funzione si veda il capitolo dal nome “Codice” con sottoparagrafo dal nome “Utils.c”.

#### 4.1.10 Subroutine – printResult

La funzione “printResult” è progettata per stampare il risultato del prodotto matrice per matrice.

Il prototipo della funzione è il seguente:

```
void** printResult(int matrixSize, int gridSize, double **  
                    partialResult, MPI_Comm mpi_comm_grid);
```

Descriviamo i parametri presi in input dalla funzione :

- matrixSize : rappresenta la dimensione originale della matrice (numero totale di righe e colonne)
- gridSize : dimensione della griglia di processori utilizzata per la computazione parallela
- partialResult : il risultato parziale del prodotto delle matrici ottenuto dal chiamante
- mpi\_comm\_grid : il comunicatore MPI che rappresenta la griglia virtuale dei processori

Funzionamento : La funzione inizia ottenendo l’ID del processo corrente nella grigli MPI (‘processId’) ,utilizzando la funzione ‘MPI\_Com\_rank’. Successivamente, ogni processore, inclusi quelli non appartenenti alla griglia (il cui risultato parziale è vuoto), invia i propri risultati parziali al processo MASTER\_ID utilizzando ‘MPI\_Send’. Questa operazione avviene attraverso un doppio loop che itera sulle righe e colonne della porzione di risultato parziale di ciascun processore.

Il processo con ‘MASTER\_ID’ riceve i risultati parziali da tutti i processori, organizzandoli nella matrice ‘resultMatrix’ in basa alle coordinate nella griglia e alle dimensioni della matrice parziale. Ciò avviene attraverso un doppio loop innestato che itera sulle coordinate della griglia e sulla porzione di risultato di ciascun processore.

Infine, il processo con ‘Master\_ID’ stampa la matrice risultante utilizzando la funzione ausiliaria ‘printSquareMatrix’.

Nota : La funzione assume che la matrice ‘partialResult’ sia stata allocata e inizializzata correttamente dal chiamante, inoltre la funzione è progettata per essere utilizzata all’interno di un programma MPI parallelo per la moltiplicazione di matrici distribuita tra diversi processori.

Per il codice della finzione si veda il capitolo dal nome “Codice” con sottoparagrafo dal nome “Utils.c”.

#### **4.1.11 Subroutine – validateInput**

La funzione “ validateInput “ è progettata per stampare il risultato del prodotto matrice per matrice.

Il prototipo della funzione è il seguente:

```
int validateInput(int numberOfProcessor, int** matrixSize,
                  int argc, char** argv, int processId);
```

Descriviamo i parametri presi in input dalla funzione :

- `numberOfProcessor` : un intero che rappresenta il numero totale di processori coinvolti nel calcolo
- `matrixSize` : un puntatore a un intero, che rappresenta la dimensione della matrice. Il valore puntato sarà modificato dalla funzione in base all'argomento passato
- `argc` : un intero che rappresenta il numero di argomenti passati dalla riga di comando
- `argv` : un array di stringhe che contiene gli argomenti passata dalla riga di comando
- `processId` : un intero che rappresenta l'identificatore del processo

Funzionamento : La funzione verifica se il numeri di argomenti passati in input. Se ciò non è vero, stampa un messaggio di errore (nel processo con ‘MASTER\_ID’) e restituisce 1 per indicare un errore.

Verifica se il numero di processori fornito (‘numberOfProcessors’) è una radice quadrata di un numero interi. Se ciò non è vero, stampa un messaggio di errore (nel processo con ‘MASTER\_ID’) e restituisce 1 per indicare un errore

Utilizza la funzione ‘parseInt’ per convertire la stringa che rappresenta la dimensione della matrice in un valore intero. Se il parsing non riesce stampa un messaggio di errore(nel processo con ‘MASTER\_ID’) e restituisce 1.

Verifica che la dimensione della matrice ‘matrixSize’ sia maggiore o uguale al numero totale di processori ‘numberOfProcessors’ e che sia un multiplo del numero di processori. Se queste condizioni non sono soddisfatte, stampa un messaggio di errore (nel processo con ‘MASTER\_ID’) e restituisce 1.

Risultato :Se tutti i controlli hanno esito positivo, la funzione restituisce 0 per indicare che gli input sono validi.

Per il codice della finzione si veda il capitolo dal nome “Codice” con sottoparagrafo dal nome “Utils.c”.

### 3.1 Subroutine MPI

Il software descritto, si avvale dell’ausilio di alcune routine della libreria MPI per il calcolo parallelo.

Esse sono documentate differentemente da quelle personalizzate in quanto la documentazione ufficiale è disponibile su <https://www.open-mpi.org/doc/>, viene però fornita una breve descrizione.

#### 3.1.1 Routine – MPI\_Init

La routine *MPI\_Init* si occupa di inizializzare l’ambiente di esecuzione MPI.

Il prototipo della funzione è:

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

Tale routine prende in input :

- argc : un puntatore al numero di argomenti del programma ;
- size : l'array degli argomenti del programma.

### 3.1.2 Routine – MPI\_Comm\_rank

La routine *MPI\_Comm\_rank* si occupa di assegnare un identificativo (chiamato rank) al processo appartenente ad un communicator.

Il prototipo della funzione è :

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

Tale routine prende in input :

- comm : il communicator da cui si vuole ottenere il rank;

mentre in output :

- rank (int) : il rank del processo chiamante nel gruppo comm.

### 3.1.3 Routine – MPI\_Comm\_size

La routine *MPI\_Comm\_size* si occupa di ritornare la dimensione del gruppo di processi appartenenti ad un communicator.

Il prototipo della funzione è :

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

Tale routine prende in input :

- comm : il communicator da cui si vuole ottenere la dimensione;

mentre in output :

- size (int) : il numero di processori del gruppo comm.

### 3.1.4 Routine – MPI\_Bcast

La routine *MPI\_Bcast* permette al processore con identificativo root di spedire a tutti i processori del communicator *comm* lo stesso dato memorizzato in \*buffer. *Count*, *datatype*, *comm* devono essere uguali per ogni processore del communicator *comm*.

Il prototipo della funzione è :

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

Tale routine prende in input:

- count: Massimo numero di elementi da ricevere nel buffer
- datatype: Tipo di ogni dato da ricevere nel buffer
- comm: Communicator
- root: Rank del processo che invia i dati
- buffer: Indirizzo iniziale del buffer di ricezione (parametro di output)

mentre in output :

- buffer : il valore che sarà condiviso tra i processori.

### 3.1.5 Routine – MPI\_Wtime

La routine *MPI\_Wtime* ritorna un valore temporale in secondi passato dall'avvio di un processore.

Il prototipo della funzione è :

```
double MPI_Wtime(void)
```

Tale routine restituisce in output :

- time (double ) : il tempo trascorso dall'ultima volta in cui la funzione è stata chiamata.

### 3.1.6 Routine – MPI\_Reduce

La routine *MPI\_Reduce* permette al processore root di ottenere il risultato dell'operazione op degli elementi memorizzati in \*sendbuf. Tale risultato viene memorizzato in recvbuf.

Il prototipo della funzione è :

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

Tale routine prende in input :

- sendbuf : il valore su cui applicare l'operazione di reduce
- count : il numero di valori
- datatype : il tipo di dato del valore, tra MPI\_INT, MPI\_FLOAT,....
- op : l'operazione da effettuare, tra MPI\_SUM, MPI\_PROD,...

- root : l'identificativo del processo che riceverà il risultato
- comm : il communicator

Mentre restituisce in output :

- recvbuf : il risultato dell'operazione.

### 3.1.7 Routine – MPI\_Send

La routine *MPI\_Send* effettua un invio di dati in modo sincrono (bloccante).

Il prototipo della funzione è :

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

Tale routine prende in input :

- buf : l'indirizzo del valore da inviare
- count : il numero di valori
- datatype : il tipo di dato del valore, tra MPI\_INT, MPI\_FLOAT,....
- dest : l'identificativo del processo di destinazione
- tag : un tag associato all'invio
- comm : il communicator

### 3.1.8 Routine – MPI\_Recv

La routine *MPI\_Recv* effettua una ricezione di dati in modo sincrono (bloccante).

Il prototipo della funzione è :

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

Tale routine prende in input :

- count : il numero di valori
- datatype : il tipo di dato del valore, tra MPI\_INT, MPI\_FLOAT,....
- Source : l'identificativo del processo mittente
- tag : un tag associato all'invio
- comm : il communicator

Mentre restituisce in output :

- buf : l'indirizzo dove verrà memorizzato il valore ricevuto
- status : informazioni aggiuntive sulla ricezione

### 3.1.9 Routine – MPI\_Barrier

La routine *MPI\_Barrier* fornisce un meccanismo sincronizzante per tutti i processori del communicator comm.

Il prototipo della funzione è :

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

Tale routine prende in input :

- comm : il communicator

### 3.1.10 Routine – MPI\_Finalize

La routine *MPI\_Finalize* termina l'ambiente di esecuzione MPI

Il prototipo della funzione è :

```
int MPI_Finalize();
```

### 3.1.11 Routine – MPI\_Cart\_create

La routine *MPI\_Cart\_create* è un’operazione collettiva che restituisce un nuovo comunicator *new\_comm* in cui i processi sono organizzati in una griglia di dimensioni *dim* comunicator *comm*.

Il prototipo della funzione è :

```
int MPI_Cart_create(MPI_Comm comm, int dim, int* ndim,
int* periods, int reorder, MPI_Comm* comm_cart);
```

Tale routine prende in input :

- *old\_comm* : il comunicator precedentemente utilizzato
- *dim* : numero di dimensione della griglia
- *ndim* : vettore di dimensione *dim* contenente le lunghezze di ciascuna dimensione
- *periods* : vettore di dimensione *dim* contenente la periodicità di ciascuna dimensione
- *reorder* : il comunicator associato alla griglia

Mentre restituisce in output :

- *comm\_cart* : il comunicator associato alla griglia

### 3.1.12 Routine – MPI\_Cart\_coords

La routine *MPI\_Cart\_create* è un’operazione collettiva che restituisce a ciascun processo di *comm\_grid* con identificativo *menu\_grid*, le sue coordinate all’interno della griglia predefinita. Inoltre “**coordinate**” è un vettore di dimensione *dim*, i cui elementi rappresentano le coordinate del processo all’interno della griglia.

Il prototipo della funzione è :

```
int MPI_Cart_coords(MPI_Comm comm_grid, int menum, int
                     dim, int coordinate[]);
```

Tale routine prende in input :

- comm\_grid : il communicator precedentemente utilizzato
- menum : identificativo del processo
- dim : dimensione del vettore coordinate

Mentre restituisce in output :

- coordinate[] : le coordinate del processo con identificativo menum.

### 3.1.13 Routine – MPI\_Cart\_rank

La routine *MPI\_Cart\_rank* è una funzione che, date delle coordinate, ritorna il menum associato a tali coordinate.

Il prototipo della funzione è :

```
int MPI_Cart_rank(MPI_Comm comm_grid, int* coords, int*
                     menum);
```

Tale routine prende in input :

- comm\_grid : il communicator precedentemente utilizzato
- coords : coordinate del processo di cui si vuole conoscere il rank

Mentre restituisce in output :

- menum : identificativo del processo

## 5. Esempi d'uso

In questo capitolo, viene fornita una dettagliata descrizione dello script PBS, accompagnata da esempi specifici che illustrano l'utilizzo, pratico, dell'algoritmo.

### 5.1 Script PBS utilizzato per l'esecuzione dell'algoritmo

```
1.#!/bin/bash
2.
3. # Imposta le direttive per l'ambiente PBS
4. #PBS -q studenti
5. #PBS -l nodes=4:ppn=4
6. #PBS -N matrix
7. #PBS -o matrix.out
8. #PBS -e matrix.err
9.
10. sort -u $PBS_NODEFILE > hostlist
11.
12. NCPU=$(wc -l < hostlist)
13. echo "[Job-Script] Starting with \"$NCPU\" CPUs..."
14.
15. echo "[Job-Script] Compiling..."
16. PBS_O_WORKDIR=$PBS_O_HOME/Elaborato4
17. /usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS_O_WORKDIR/matrix
18. $PBS_O_WORKDIR/main.c $PBS_O_WORKDIR/utils.c
19.
20. echo "[Job-Script] Checking input the values..."
21. #####
22. ## CUSTOM VALUES ##
23. #####
24. # La dimensione della matrice, deve essere uguale o un multiplo del numero
25. di processori
26. MATRIX_SIZE=4
27. #####
28.
29. echo "[Job-Script] Running the process..."
30. /usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile hostlist --np $NCPU
31. $PBS_O_WORKDIR/matrix $MATRIX_SIZE
```

Questo file denominato “*job-script.pbs*” ci permette di compilare ed eseguire il programma C denominato “*main.c*” che si occupa di calcolare il prodotto scalare tra due matrici. Le uscite dell'esecuzione vengono reindirizzate ai file “*matrix.out*” ed “*matrix.err*”.

Descriviamo brevemente i vari parametri presenti nel file pbs :

1. Definizione delle direttive PBS :

- ‘`#!/bin/bash`’ : dichiarazione del tipo di shell utilizzata
- ‘`#PBS -q studenti`’ : specifica la coda su cui verrà eseguito il job
- ‘`#PBS -l nodes=4 : ppn=4`’ : richiede 4 nodo con 4 processori
- ‘`#PBS -N matrix`’ : assegna un nome al job, in questo caso “matxvet”.
- ‘`#PBS -o output.out`’ e ‘`#PBS -e error.err`’ : specifica i file in cui verranno indirizzati i messaggi di output e gli errori

2. Creazione del file di hostlist :

- ‘`sort -u $PBS_NODEFILE > hostlist`’ : crea un elenco univoco di nodi disponibili e li scrive nel file “hostlist”

3. Inizializzazione del numero di CPU :

- ‘`NCPU=$(wc -l < hostlist)`’ : conta il numero di righe nel file “hostlist” per determinare il numero totale di CPU disponibili.

4. Compilazione del programma parallelo :

- ‘`PBS_O_WORKDIR = $PBS_O_HOME/Elaborato4`’ : imposta la directory di lavoro in base alla variabile PBS\_O\_HOME
- ‘`/usr/lib64/openmpi/1.4-gcc/bin/mpicc -o %PBS_O_WORKDIR/matrix $PBS_O_WORKDIR/main.c $PBS_O_WORKDIR/utils.c`’ : compila il programma parallelo usando il compilatore MPI

5. Verifica dei valori di input personalizzati :

- ‘`MATRIX_SIZE =4`’ : imposta la dimensione della matrice

6. Esecuzione del programma parallelo :

- ‘`/usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile hostlist -np $NCPU $PBS_O_WORKDIR/MATRIX $MATRIX_SIZE`’ : esegue il programma parallelo utilizzando MPI, specificando il

file hostlist, il numero totale di processori e la dimensione della matrice come argomento del programma.

Il primo passo per eseguire e compilare il programma “*main.c*” è eseguire il file pbs, ciò viene fatto collegandoci al cluster tramite credenziali, posizionandoci nella cartella contenente il file “*main.c*” e eseguendo sul terminale il comando “*qsub job-script.pbs*”.

All’ esecuzione del comando “*qsub job-script.pbs*” si generano diversi file, in particolare:

1. L’esecutore relativo al programma scritto in linguaggio C;
2. Un file “matrix.out” contenente l’output del programma.
3. Un file “*matrix.err*” contenente eventuali errori del programma, dove in caso di esecuzione senza errore, tale file rimarrà vuoto.

## 5.1 Esempi d’uso

Vengono riportati alcuni esempi d’uso, partendo dall’esecuzione del file PBS fino a mostrare l’output del programma.

### 5.1.1 Esempi 1 – ( $A \in R^{(4 \times 4)}$ e $B \in R^{(4 \times 4)}$ )

Tenendo conto del seguente file “job-script.pbs” :

```

#!/bin/bash

# Imposta le direttive per l'ambiente PBS
#PBS -q studenti
#PBS -l nodes=1:ppn=4
#PBS -N matrix
#PBS -o matrix.out
#PBS -e matrix.err

sort -u $PBS_NODEFILE > hostlist
$NCPU=$(wc -l < hostlist)
echo "[Job-Script] Starting with \"$NCPU\" CPUs..."

echo "[Job-Script] Compiling..."
PBS_O_WORKDIR=$PBS_O_HOME/Elaborato4
/usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS_O_WORKDIR/matrix $PBS_O_WORKDIR/main.c $PBS_O_WORKDIR/utils.c

echo "[Job-Script] Checking input the values..."
#####
## CUSTOM VALUES ##
#####
# Nota: La dimensione della matrice, deve essere uguale o un multiplo del numero di processori
MATRIX_SIZE=4
#####

echo "[Job-Script] Running the process..."
/usr/lib64/openmpi/1.4-gcc/bin/mpicxx --machinefile hostlist --np $NCPU $PBS_O_WORKDIR/matrix $MATRIX_SIZE

```

Come possiamo vedere, viene dato l'input al programma passando i parametri come argomenti. Il programma verrà lanciato con un numero di processori pari a 4 e matrici di dimensioni 4x4.

Il prossimo passo è eseguire il comando “`qsub <nome_file>.pbs`” ed attendere la fine dell'esecuzione attraverso il comando “`qstat`”.

Una volta terminata l'esecuzione possiamo mostrare il contenuto del file “`matrix.out`” che sarà vuoto nel caso sia andato tutto bene (come in questo caso). Per vedere l'output del programma eseguiamo il comando “`cat matrix.out`”.

Nell'esempio che stiamo analizzando possiamo vedere l'output del programma :

```

[SLMFBL99X@ui-studenti Elaborato4]$ cat matrix.out
[Job-Script] Starting with 4 CPUs...
[Job-Script] Compiling...
[Job-Script] Checking input the values...
[Job-Script] Running the process...
Error: Matrix size must be >= of the grid and a multiple of number of processors.
matrixSize=2
numberofProcessors:4
[SLMFBL99X@ui-studenti Elaborato4]$ nano job-script.pbs
[SLMFBL99X@ui-studenti Elaborato4]$ qsub job-script.pbs
[Job-Script] Starting...
[SLMFBL99X@ui-studenti Elaborato4]$ nano job-script.pbs
[SLMFBL99X@ui-studenti Elaborato4]$ cat matrix.out
[Job-Script] Starting with 4 CPUs...
[Job-Script] Compiling...
[Job-Script] Checking input the values...
[Job-Script] Running the process...
First random matrix:
13.576897 13.884083 7.612881 12.957953
2.78855 12.388838 6.387823 6.026791
0.072143 12.882964 9.874974 2.228789
0.094759 14.379514 6.917288 10.137202

Second random matrix:
11.688401 11.212667 14.483336 3.251404
14.686461 1.292935 5.710284 12.361529
7.024677 11.325658 13.941376 13.583656
1.798164 12.708515 1.671366 0.375061

RESULT:
389.523990 410.088887 354.265889 234.581891
225.639548 281.984693 277.747138 232.988383
116.100231 52.559224 130.778631 98.555556
370.404481 377.213579 301.267142 220.553962

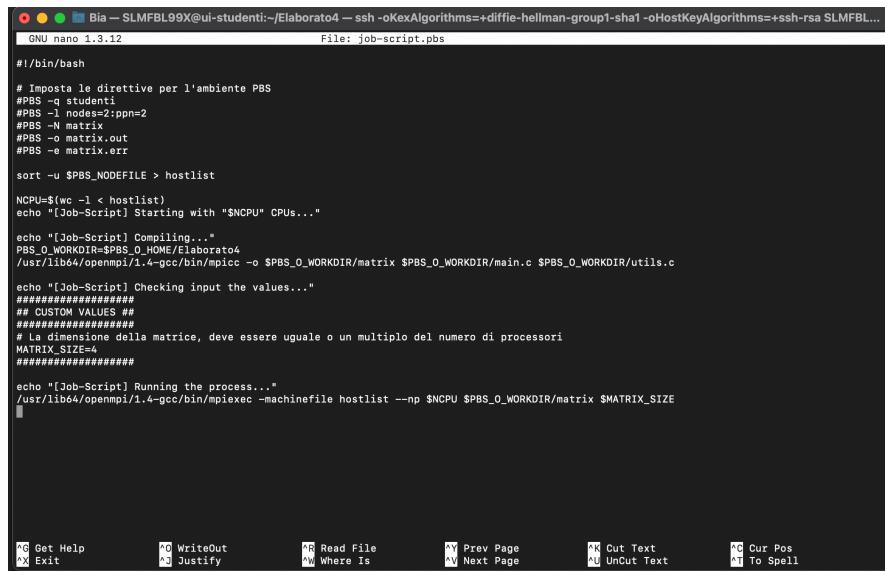
Time elapsed: 0.080422 seconds
[SLMFBL99X@ui-studenti Elaborato4]$

```

Viene, inoltre, visualizzato anche il tempo calcolato in quanto è stato usato l'algoritmo che preleva anche i tempi di esecuzione.

### 5.1.2 Esempi 2 – ERROR - Number of processors provided cannot be distributed in a NN grid. number of processors:N

Tenendo conto del seguente file “job-script.pbs” :



```
#!/bin/bash

# Imposta le direttive per l'ambiente PBS
#PBS -q studenti
#PBS -l nodes=2:ppn=2
#PBS -N matrix
#PBS -o matrix.out
#PBS -e matrix.err

sort -u $PBS_NODEFILE > hostlist
NCPU=$(wc -l < hostlist)
echo "[Job-Script] Starting with \"$NCPU\" CPUs..."

echo "[Job-Script] Compiling..."
PBS_O_WORKDIR=$PBS_O_HOME/Elaborato4
/usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS_O_WORKDIR/matrix $PBS_O_WORKDIR/main.c $PBS_O_WORKDIR/utils.c

echo "[Job-Script] Checking input the values...
#####
## CUSTOM VALUES ##
#####
# La dimensione della matrice, deve essere uguale o un multiplo del numero di processori
MATRIX_SIZE=8
#####

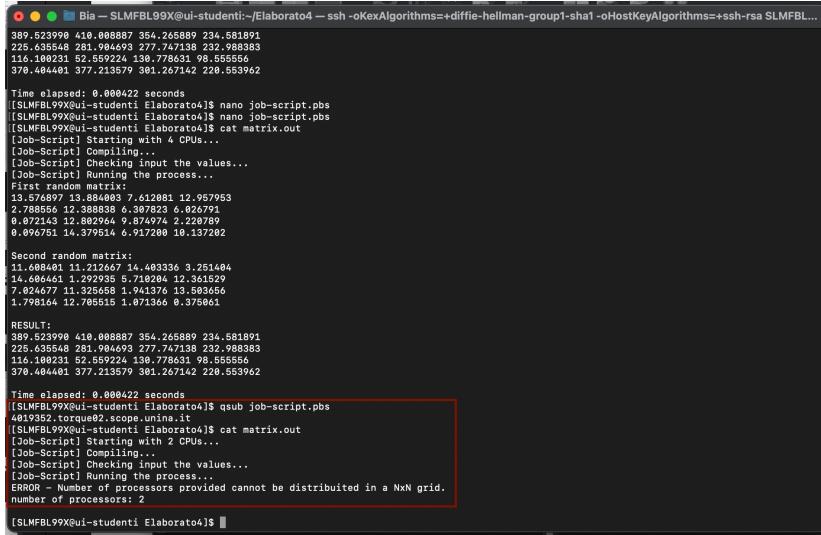
echo "[Job-Script] Running the process..."
/usr/lib64/openmpi/1.4-gcc/bin/mpixexec --np $NCPU $PBS_O_WORKDIR/matrix $MATRIX_SIZE
```

Come possiamo vedere, viene dato l'input al programma passando i parametri come argomenti. Il programma verrà lanciato con un numero di processori pari a 2 e matrici di dimensioni 4x4.

Il prossimo passo è eseguire il comando “*qsub <nome\_file>.pbs*” ed attendere la fine dell'esecuzione attraverso il comando “*qstat*”.

Una volta terminata l'esecuzione possiamo mostrare il contenuto del file “*matrix.err*” che sarà vuoto nel caso sia andato tutto bene (come in questo caso). Per vedere l'output del programma dobbiamo eseguire il comando “*cat matrix.out*”.

Nell'esempio che stiamo analizzando possiamo vedere l'output del programma :



```
389.523998 410.000887 354.265889 234.581894
225.635548 281.984693 277.747138 232.988383
116.100731 52.559224 138.778631 98.555556
378.404401 377.213579 301.267142 220.553962

Time elapsed: 0.000422 seconds
[SLMFBL99X@ui-studenti Elaborato4]$ nano job-script.pbs
[SLMFBL99X@ui-studenti Elaborato4]$ nano job-script.pbs
[SLMFBL99X@ui-studenti Elaborato4]$ cat matrix.out
[Job-Script] Starting with 4 CPUs...
[Job-Script] Compiling...
[Job-Script] Checking input the values...
[Job-Script] Running the process...
First random matrix:
13.576897 13.884883 7.612081 12.957953
2.788556 12.388838 6.387823 6.026791
0.072143 12.882964 9.874974 2.220789
0.086751 14.379514 6.917288 18.137282

Second random matrix:
11.608401 11.212667 14.493336 3.251464
14.686441 1.292938 5.710284 13.361529
7.024677 11.325688 1.941576 13.583656
1.798164 12.785515 1.071366 0.375661

RESULT:
389.523998 410.000887 354.265889 234.581894
225.635548 281.984693 277.747138 232.988383
116.100731 52.559224 138.778631 98.555556
378.404401 377.213579 301.267142 220.553962

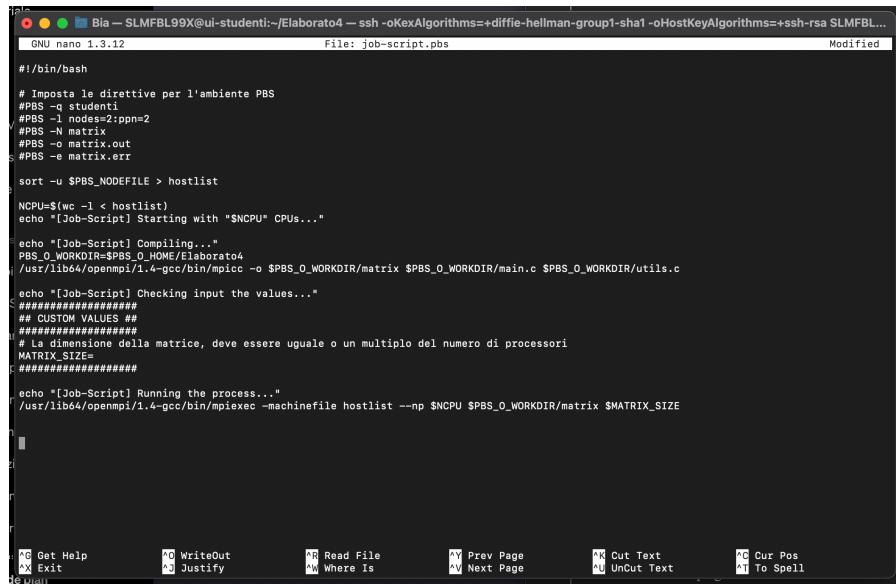
Time elapsed: 0.000422 seconds
[SLMFBL99X@ui-studenti Elaborato4]$ qsub job-script.pbs
4019362.torque@2.scope.unina.it
[SLMFBL99X@ui-studenti Elaborato4]$ cat matrix.out
[Job-Script] Starting with 2 CPUs...
[Job-Script] Compiling...
[Job-Script] Checking input the values...
[Job-Script] Running the process...
[Job-Script] Running the process...
ERROR - Number of processors provided cannot be distributed in a NxN grid.
number of processors: 2

[SLMFBL99X@ui-studenti Elaborato4]$
```

Si presenta tale errore in quanto il numero di processori fornito non è compatibile con la creazione di una griglia quadrata, ciò dovuto in particolare al vincolo imposto sulla griglia che deve essere quadrata.

### 5.1.3 Esempi 3 – Provide only one argument: the number of rows (and columns) of the matrices

Tenendo conto del seguente file “job-script.pbs” :



```

GNU nano 1.3.12                               File: job-script.pbs                         Modified
#!/bin/bash

# Imposta le direttive per l'ambiente PBS
#PBS -q studenti
#PBS -N matrix
#PBS -o matrix.out
#PBS -e matrix.err

sort -u $PBS_NODEFILE > hostlist
NCPU=$(wc -l < hostlist)
echo "[Job-Script] Starting with \"$NCPU\" CPUs..."

echo "[Job-Script] Compiling..."
PBS_O_WORKDIR=$PBS_O_HOME/Elaborato4
/usr/lib64/openmpi/1.4-gcc/mpicc -o $PBS_O_WORKDIR/matrix $PBS_O_WORKDIR/main.c $PBS_O_WORKDIR/utils.c

echo "[Job-Script] Checking input the values..."
## CUSTOM VALUES ##
#####
# La dimensione della matrice, deve essere uguale o un multiplo del numero di processori
MATRIX_SIZE=
#####

echo "[Job-Script] Running the process..."
/usr/lib64/openmpi/1.4-gcc/bin/mpexec --np $NCPU $PBS_O_WORKDIR/matrix $MATRIX_SIZE

n
z
r
t
d

^G Get Help      ^Q WriteOut     ^R Read File      ^Y Prev Page      ^K Cut Text      ^C Cur Pos
^X Exit          ^J Justify      ^W Where Is       ^V Next Page      ^U Uncut Text    ^I To Spell
^U Undo          ^P Print        ^O Open          ^L Select All    ^A Select None

```

Come possiamo vedere, non viene indicata la dimensione delle matrici.

Il prossimo passo è eseguire il comando “*qsub <nome\_file>.pbs*” ed attendere la fine dell'esecuzione attraverso il comando “*qstat*”.

Una volta terminata l'esecuzione possiamo mostrare il contenuto del file “*matrix.err*” che sarà vuoto nel caso sia andato tutto bene (come in questo caso). Per vedere l'output del programma dobbiamo eseguire il comando “*cat matrix.out*”.

Nell'esempio che stiamo analizzando possiamo vedere l'output del programma :

```

Second random matrix:
11.212667 11.212667 3.251336
14.698461 1.292935 5.710204 12.361529
7.024677 11.325658 1.941376 13.583656
1.798164 12.705515 1.071366 0.375061

RESULT:
389.523998 410.008887 354.265889 234.581891
225.635548 281.904693 277.747138 232.988383
116.180231 52.559224 138.778631 98.555556
370.484401 377.213579 381.267142 220.553962

Time elapsed: 0.000422 seconds
[SLMFBL99X@ui-studenti Elaborato4]$ qsub job-script.pbs
4019352.torque02.scope.unina.it
[SLMFBL99X@ui-studenti Elaborato4]$ cat matrix.out
[Job-Script] Starting with 2 CPUs...
[Job-Script] Compiling...
[Job-Script] Checking input the values...
[Job-Script] Running the process...
[Job-Script] ERROR - Number of processors provided cannot be distributed in a NxN grid.
number of processors: 2

[SLMFBL99X@ui-studenti Elaborato4]$ nano job-script.pbs
[SLMFBL99X@ui-studenti Elaborato4]$ qsub job-script.pbs
4019353.torque02.scope.unina.it
[SLMFBL99X@ui-studenti Elaborato4]$ cat matrix.out
[Job-Script] Starting with 2 CPUs...
[Job-Script] Compiling...
[Job-Script] Checking input the values...
[Job-Script] Running the process...
INSERIRE UN SOLO ARGOMENTO: IL NUMERO DI RIGHE (E COLONNE) DELLE MATRICI
[SLMFBL99X@ui-studenti Elaborato4]$ nano main.c
[SLMFBL99X@ui-studenti Elaborato4]$ qsub job-script.pbs
4019354.torque02.scope.unina.it
[SLMFBL99X@ui-studenti Elaborato4]$ cat matrix.out
[Job-Script] Starting with 2 CPUs...
[Job-Script] Compiling...
[Job-Script] Checking input the values...
[Job-Script] Running the process...
[Job-Script] Provide only one argument: the number of rows (and columns) of the matrices.
[SLMFBL99X@ui-studenti Elaborato4]$ 

```

Tale errore si verifica in quanto lo script richiede solo un argomento, ovvero il numero di righe (e colonne) della matrice, ma non lo si fornisce in input.

### 5.1.4 Esempi 4 – Errore - Matrix size must be >= of the grid and a multiple of number of processor matrixsize : N numberofProcessors : M

Tenendo conto del seguente file “job-script.pbs” :

```

GNU nano 1.3.12
File: job-script.pbs
#!/bin/bash

# Imposta le direttive per l'ambiente PBS
#PBS -q studenti
#PBS -N matrix
#PBS -o matrix.out
#PBS -e matrix.err

sort -u $PBS_NODEFILE > hostlist
NCPU=$(wc -l < hostlist)
echo "[Job-Script] Starting with \"$NCPU\" CPUs..." 

echo "[Job-Script] Compiling..."
PBS_O_WORKDIR=$PBS_O_HOME/elaborato4
/usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS_O_WORKDIR/matrix $PBS_O_WORKDIR/main.c $PBS_O_WORKDIR/utils.c

echo "[Job-Script] Checking input the values..." 
#####
#####
#####
# La dimensione della matrice, deve essere uguale o un multiplo del numero di processori
MATRIX_SIZE=10
#####

echo "[Job-Script] Running the process..." 
/usr/lib64/openmpi/1.4-gcc/bin/mpicexec --np $NCPU $PBS_O_WORKDIR/matrix $MATRIX_SIZE

```

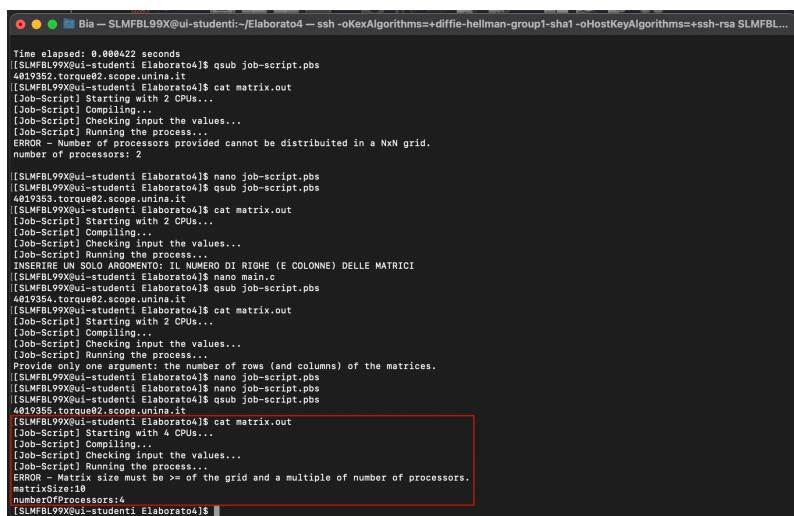
Come possiamo vedere, viene dato l'input al programma passando i parametri come argomenti. Il programma verrà lanciato con un numero di processori pari a 4 e matrici di dimensioni 10x10.

Il prossimo passo è eseguire il comando “*qsub <nome\_file>.pbs*” ed attendere la fine dell'esecuzione attraverso il comando “*qstat*”.

Una volta terminata l'esecuzione possiamo mostrare il contenuto del file “*matrix.err*” che sarà vuoto nel caso sia andato tutto bene (come in questo caso).

Per vedere l'output del programma dobbiamo eseguire il comando “*cat matrix.out*”.

Nell'esempio che stiamo analizzando possiamo vedere l'output del programma :

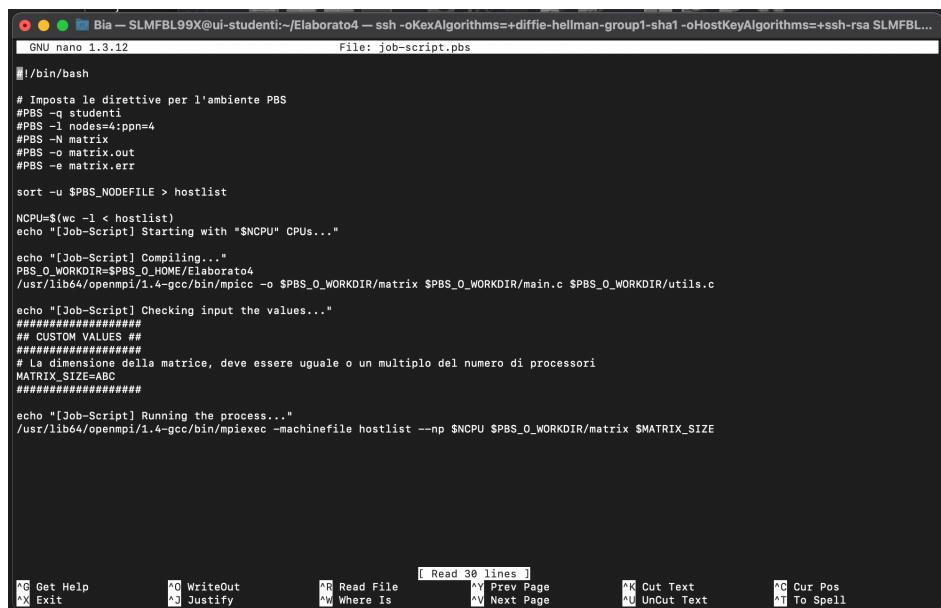


```
Time elapsed: 0.000422 seconds
[SLMFBL99X@Elaborato4 ~]$ qsub job-script.pbs
4019353 torque@2.scope.unina.it
[SLMFBL99X@Elaborato4 ~]$ cat matrix.out
[Job-Script] Starting with 2 CPUs...
[Job-Script] Compiling...
[Job-Script] Checking input the values...
[Job-Script] Running the process...
[Job-Script] ERROR Number of processors provided cannot be distributed in a NxN grid.
number of processors: 2
[SLMFBL99X@Elaborato4 ~]$ nano job-script.pbs
[SLMFBL99X@Elaborato4 ~]$ qsub job-script.pbs
4019353 torque@2.scope.unina.it
[SLMFBL99X@Elaborato4 ~]$ cat matrix.out
[Job-Script] Starting with 2 CPUs...
[Job-Script] Compiling...
[Job-Script] Checking input the values...
[Job-Script] Running the process...
INSERIRE UN SOLO ARGOMENTO: IL NUMERO DI RIGHE (E COLONNE) DELLE MATRICI
[SLMFBL99X@Elaborato4 ~]$ nano main.c
[SLMFBL99X@Elaborato4 ~]$ qsub job-script.pbs
4019353 torque@2.scope.unina.it
[SLMFBL99X@Elaborato4 ~]$ cat matrix.out
[Job-Script] Starting with 2 CPUs...
[Job-Script] Compiling...
[Job-Script] Checking input the values...
[Job-Script] Running the process...
[Job-Script] ERROR - Matrix size must be >= of the grid and a multiple of number of processors.
matrixSize:10
numberofProcessors:4
[SLMFBL99X@Elaborato4 ~]$
```

Come notiamo vi è un'errore che in quanto la dimensione della matrice fornita deve essere maggiore o uguale alla dimensione della griglia (quadrata) e deve essere un multiplo del numero dei processori specificato.

### 5.1.5 Esempi 5 – Errore - Cannot parse the matrixSize with value provided. matrixSize : ABC

Tenendo conto del seguente file “job-script.pbs” :



```
GNU nano 1.3.12          File: job-script.pbs
#!/bin/bash

# Imposta le direttive per l'ambiente PBS
#PBS -q studenti
#PBS -l nodes=4:ppn=4
#PBS -N matrix
#PBS -o matrix.out
#PBS -e matrix.err

sort -u $PBS_NODEFILE > hostlist
NCPU=$(wc -l < hostlist)
echo "[Job-Script] Starting with \"$NCPU\" CPUs..."

echo "[Job-Script] Compiling..."
PBS_O_WORKDIR=$PBS_O_HOME/Elaborato4
/usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS_O_WORKDIR/matrix $PBS_O_WORKDIR/main.c $PBS_O_WORKDIR/utils.c

echo "[Job-Script] Checking input the values...
#####
## CUSTOM VALUES ##
#####
# La dimensione della matrice, deve essere uguale a un multiplo del numero di processori
MATRIX_SIZE=ABC
#####

echo "[Job-Script] Running the process..."
/usr/lib64/openmpi/1.4-gcc/bin/mpieexec -machinefile hostlist --np $NCPU $PBS_O_WORKDIR/matrix $MATRIX_SIZE

[ Read 38 lines ]
[G Get Help      ^O WriteOut      ^R Read File      ^W Where Is      ^Y Prev Page      ^X Cut Text      ^C Cur Pos
^X Exit          ^J Justify       ^W Where Is      ^Y Next Page      ^U UnCut Text     ^T To Spell
```

Come possiamo vedere, come dimensione corrispondenti alla matrice abbiamo una stringa.

Il prossimo passo è eseguire il comando “*qsub <nome\_file>.pbs*” ed attendere la fine dell'esecuzione attraverso il comando “*qstat*”.

Una volta terminata l'esecuzione possiamo mostrare il contenuto del file “*matrix.err*” che sarà vuoto nel caso sia andato tutto bene (come in questo caso). Per vedere l'output del programma dobbiamo eseguire il comando “*cat matrix.out*”.

Nell'esempio che stiamo analizzando possiamo vedere l'output del programma :

```

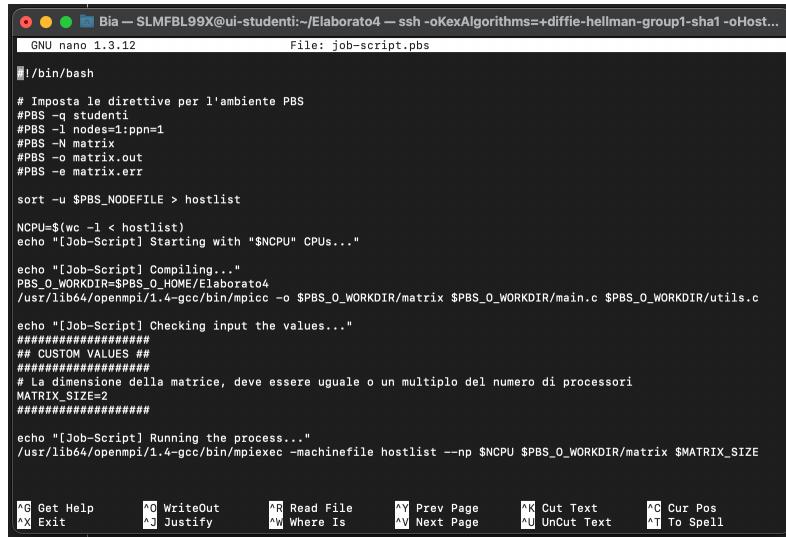
[● ○ ●] Bia - SLMFBL99X@ui-studenti:~/Elaborato4 -- ssh -oKexAlgorithms=+diffie-hellman-group1-sha1 -oHostKeyAlgorithms=+ssh-rsa SLMFBL...
[SLMFBL99X@ui-studenti Elaborato4]$ nano job-script.pbs
[SLMFBL99X@ui-studenti Elaborato4]$ qsub job-script.pbs
4019353.torque@02.scope.unina.it
[SLMFBL99X@ui-studenti Elaborato4]$ cat matrix.out
[Job-Script] Starting with 2 CPUs...
[Job-Script] Compiling...
[Job-Script] Checking input the values...
[Job-Script] Running the process...
INSERIRE UN SOLO ARGOMENTO: IL NUMERO DI RIGHE (E COLONNE) DELLE MATRICI
[SLMFBL99X@ui-studenti Elaborato4]$ nano main.c
[SLMFBL99X@ui-studenti Elaborato4]$ qsub job-script.pbs
4019354.torque@02.scope.unina.it
[SLMFBL99X@ui-studenti Elaborato4]$ cat matrix.out
[Job-Script] Starting with 2 CPUs...
[Job-Script] Compiling...
[Job-Script] Checking input the values...
[Job-Script] Running the process...
Provide only one argument: the number of rows (and columns) of the matrices.
[SLMFBL99X@ui-studenti Elaborato4]$ nano job-script.pbs
[SLMFBL99X@ui-studenti Elaborato4]$ qsub job-script.pbs
4019355.torque@02.scope.unina.it
[SLMFBL99X@ui-studenti Elaborato4]$ cat matrix.out
[Job-Script] Starting with 4 CPUs...
[Job-Script] Compiling...
[Job-Script] Checking input the values...
[Job-Script] Running the process...
ERROR - Matrix size must be >= of the grid and a multiple of number of processors.
matrixSize:16
numberOfProcessors:4
[SLMFBL99X@ui-studenti Elaborato4]$ nano job-script.pbs
[SLMFBL99X@ui-studenti Elaborato4]$ qsub job-script.pbs
4019356.torque@02.scope.unina.it
[SLMFBL99X@ui-studenti Elaborato4]$ cat matrix.out
[Job-Script] Starting with 4 CPUs...
[Job-Script] Compiling...
[Job-Script] Checking input the values...
[Job-Script] Running the process...
ERROR - Cannot parse the matrixSize with value provided.
matrixSize:ABC
[SLMFBL99X@ui-studenti Elaborato4]$ 

```

Notiamo che in questo caso come dimensione delle matrici abbiamo utilizzato una stringa “ABC” che non è un valore numerico valido per la matrice. Il sistema, infatti, restituisce l’errore “*Cannot parte the matrixSize with value provided*” perché non può interpretare correttamente il valore fornito come una dimensione numerica della matrice.

### 5.1.6 Esempi 6 – ( $A \in \mathbb{R}^{(2x2)}$ e $B \in \mathbb{R}^{(2x2)}$ )

Tenendo conto del seguente file “job-script.pbs” :



```

Bia — SLMFBL99X@ui-studenti:~/Elaborato4 — ssh -oKexAlgorithms=+diffie-hellman-group1-sha1 -oHost...
File: job-script.pbs

GNU nano 1.3.12

#!/bin/bash

# Imposta le direttive per l'ambiente PBS
#PBS -q studenti
#PBS -l nodes=1:ppn=1
#PBS -N matrix
#PBS -o matrix.out
#PBS -e matrix.err

sort -u $PBS_NODEFILE > hostlist
NCPU=$(wc -l < hostlist)
echo "[Job-Script] Starting with \"$NCPU\" CPUs..."

echo "[Job-Script] Compiling..."
PBS_O_WORKDIR=$PBS_O_HOME/Elaborato4
/usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS_O_WORKDIR/matrix $PBS_O_WORKDIR/main.c $PBS_O_WORKDIR/utils.c

echo "[Job-Script] Checking input the values..."
#####
## CUSTOM VALUES ##
#####
# La dimensione della matrice, deve essere uguale o un multiplo del numero di processori
MATRIX_SIZE=2
#####

echo "[Job-Script] Running the process..."
/usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile hostlist --np $NCPU $PBS_O_WORKDIR/matrix $MATRIX_SIZE


```

^G Get Help    ^Q WriteOut    ^R Read File    ^Y Prev Page    ^K Cut Text    ^C Cur Pos  
 ^X Exit    ^J Justify    ^W Where Is    ^V Next Page    ^U UnCut Text    ^T To Spell

Come possiamo vedere, viene dato l'input al programma passando i parametri come argomenti. Il programma verrà lanciato con un numero di processori pari a 1 e matrici di dimensioni 2x2.

Il prossimo passo è eseguire il comando “*qsub <nome\_file>.pbs*” ed attendere la fine dell'esecuzione attraverso il comando “*qstat*”.

Una volta terminata l'esecuzione possiamo mostrare il contenuto del file “*matrix.err*” che sarà vuoto nel caso sia andato tutto bene (come in questo caso). Per vedere l'output del programma dobbiamo eseguire il comando “*cat matrix.out*”.

Nell'esempio che stiamo analizzando possiamo vedere l'output del programma :

```
● ○ ● Bia -- SLMFBL99X@ui-studenti:~/Elaborato4 -- ssh -oKexAlgorithms=+diffie-hellman-group1-sha1 -oHost...  
2.817401 4.279201 5.418332 9.598791  
1.998124 14.307665 11.923873 6.750082  
5.897497 11.215696 11.140066 11.339782  
7.396957 2.529333 13.629365 2.612695  
  
RESULT:  
202.629963 169.890483 228.461149 168.107024  
182.807081 229.776244 203.361968 121.152449  
282.337728 346.838086 322.257776 281.128647  
410.760765 324.949136 453.011598 332.946362  
  
Time elapsed: 0.000003 seconds  
[SLMFBL99X@ui-studenti Elaborato4]$ nano job-script.pbs  
[SLMFBL99X@ui-studenti Elaborato4]$ qsub job-script.pbs  
40193608.torque02.scope.unina.it  
[SLMFBL99X@ui-studenti Elaborato4]$ cat matrix.out  
[Job-Script] Starting with 1 CPUs...  
[Job-Script] Compiling...  
[Job-Script] Checking input the values...  
[Job-Script] Running the process...  
First random matrix:  
12.818973 14.584521  
9.076973 9.427228  
  
Second random matrix:  
2.643865 8.867996  
9.986696 11.947191  
  
RESULT:  
134.819952 150.655078  
215.981186 250.438086  
  
Time elapsed: 0.000022 seconds  
[SLMFBL99X@ui-studenti Elaborato4]$ nano job-script.pbs  
[SLMFBL99X@ui-studenti Elaborato4]$
```

Viene, inoltre, visualizzato anche il tempo calcolato in quanto è stato usato l'algoritmo che preleva anche i tempi di esecuzione.

## 6 Analisi delle performance

In questo capitolo andremo a valutare le prestazioni dell'algoritmo, in questione: che si occupa del prodotto scalare tra due matrici, mediante parametri che vengono impiegati per valutare un software parallelo. Questi parametri sono : *tempo di esecuzione*; *Speed Up* e *Efficienza*.

Di seguito vengono riportate le formule relative :

- $T(t)$  : Tempo di esecuzione
  - Questo rappresenta il tempo totale che il programma impiega per eseguire con t processori.
- $S(t) = \frac{T(1)}{T(t)}$  : Speed Up
  - Rappresenta il guadagno di prestazioni ottenuto dalla parallelizzazione. Indica quanto velocemente il programma può essere eseguito utilizzando t processori rispetto a quando viene eseguito con un singolo processore.
- $E(t) = \frac{S(t)}{t}$  : Efficienza
  - Misura quanto “bene” viene sfruttata la parallelizzazione. Rappresenta la frazione del potenziale guadagno di velocità che è stata effettivamente ottenuta per ogni processore aggiuntivo. Un’efficienza del 100% indica che ogni processore sta contribuendo in modo ottimale al miglioramento delle prestazioni.

Nello specifico, sono state effettuate alcune analisi sui tempi di esecuzione dell'algoritmo in base al numero di processori utilizzati e alla dimensione delle matrici.

Inoltre sono stati testati prodotti effettuati con matrici di dimensione 4x4, 8x8, 16x16, 32x32, 64x64, 128x128, 256x256, 512x512, 1024x1024.

Infine sono stati raccolti i tempi con 1 e 4 processori.

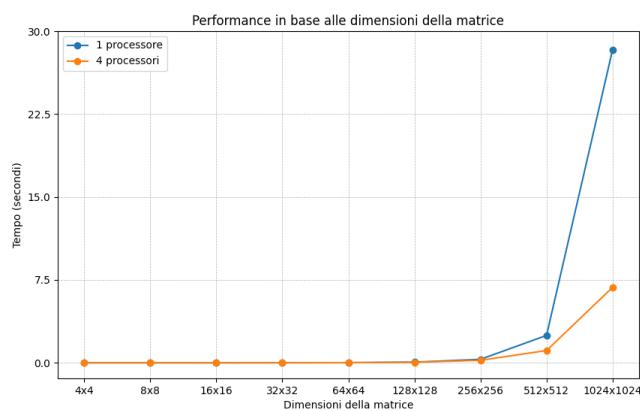
#### 6.4.1 Analisi sul numero di processori

Seguono alcune analisi basate sul numero di processori.

I primi tempi sono stati raccolti utilizzando un solo processore mentre per i secondi sono stati utilizzati quattro processori.

Processori	[4x4]	[8x8]	[16x16]	[32x32]	[64x64]	[128x128]	[256x256]	[512x512]	[1024x1024]
1	0.000045	0.000146	0.000551	0.002304	0.009639	0.052281	0.310700	2.470626	28.315679
4	0.000411	0.000768	0.001396	0.003722	0.011894	0.046020	0.222504	1.103747	6.819813

Graficamente :



Si può osservare, dal grafico, che :

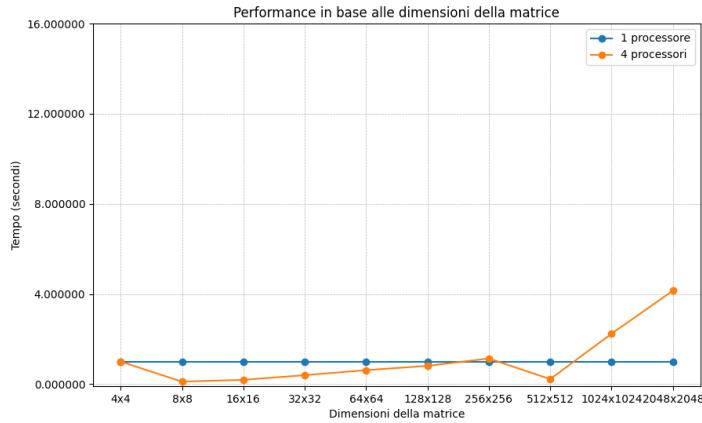
1. All'aumentare del numero di processori, i tempi di esecuzione tendono a diminuire. Questo è un comportamento comune quando si utilizzano parallelismi, poiché più processori possono lavorare contemporaneamente per completare un'attività più velocemente.
2. La scalabilità sembra essere efficace fino ad un certo punto. Ad esempio, passando da 1 a 4 processori, si osserva un miglioramento significativo nei tempi di esecuzione. Tuttavia, con l'aumentare ulteriore dei processori, il beneficio in termini di tempo sembra diminuire.
3. All'aumentare delle dimensioni della matrice (ad esempio, da [4x4] a [1024x1024]), i tempi di esecuzione aumentano. Questo potrebbe essere dovuto al fatto che gestire matrici più grandi richiede più tempo di elaborazione, anche se si utilizzano più processori.

#### 6.4.2 Calcolo dello speed-up

Segue la tabella contenente i valori relativi al calcolo dello speed-up :

Processori	[4x4]	[8x8]	[16x16]	[32x32]	[64x64]	[128x128]	[256x256]	[512x512]	[1024x1024]
1	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000
4	0.109290	0.189845	0.394985	0.619229	0.809661	1.136018	1.394539	2.237045	4.150046

Graficamente :



Si può osservare, dal grafico, che :

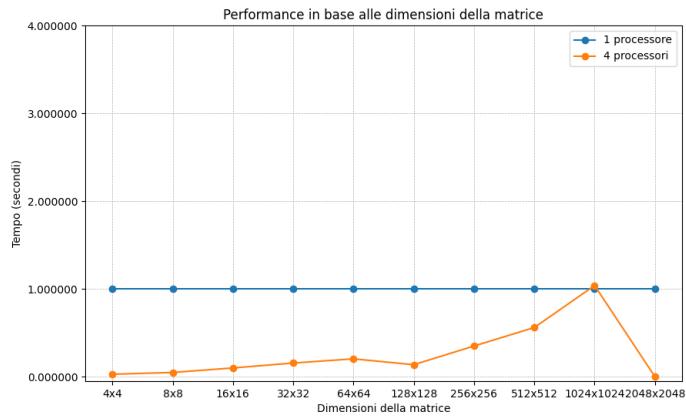
1. Con un solo processore il tempo di esecuzione per un singolo processore è considerato come fase di riferimento, quindi lo speedUp è sempre 1, indicando nessun miglioramento rispetto all'esecuzione sequenziale.
2. Con più processori (quattro), l'efficienza dell'implementazione parallela aumenta con la dimensione della matrice, raggiungendo lo speedUp massimo di circa 4.15 per la matrice di dimensioni [1024x1024]. Ciò suggerisce che il problema è altamente parallelizzabile e l'uso di più processori porta a un significativo miglioramento delle prestazioni.

#### 6.4.3 Calcolo dell'efficienza

Segue la tabella contenente i valori relativi al calcolo dell'efficienza :

Processori	[4x4]	[8x8]	[16x16]	[32x32]	[64x64]	[128x128]	[256x256]	[512x512]	[1024x1024]
1	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000
4	0.027323	0.047461	0.098746	0.154807	0.202425	0.136018	0.348635	0.559261	1.037512

Graficamente :



Si può osservare, dal grafico, che :

1. Con un solo processore l'efficienza è sempre 1, il che indica che il sistema sfrutta completamente il potenziale di parallelismo quando si utilizza un singolo processore.
2. Con più processori (quattro), l'efficienza rappresenta quanto “bene” il sistema sfrutta il parallelismo rispetto all'utilizzo di un singolo processore. I valori sono inferiori a 1 per ogni configurazione, indicando che l'efficienza diminuisce con l'aumentare del numero di processi. Questo può essere dovuto a overhead di comunicazione, sincronizzazione o altri fattori che influenzano la parallelizzazione.

#### 6.4.4 Conclusioni

Dall'analisi del tempo di esecuzione, SpeedUp e efficienza si è notato che :

**Tempo di Esecuzione :**

Abbiamo notato un aumento significativo del tempo di esecuzione all'aumentare delle dimensioni della matrice, come ci si aspetta. In particolare, con un singolo processore abbiamo osservato un notevole incremento del tempo di esecuzione al crescere delle dimensioni della matrice. Questo fenomeno riflette la complessità computazionale associata alle matrici di grandi dimensioni e la necessità di considerare approcci paralleli per affrontare queste sfide.

### **SpeedUp :**

Lo SpeedUp, che misura l'efficienza dell'esecuzione parallela rispetto a quella sequenziale, ci ha offerto interessanti spunti di riflessione. Sebbene generalmente l'aumento del numero di processori dovrebbe portare a uno SpeedUp maggiore, abbiamo notato alcuni casi, come [64x64] e [128x128], in cui lo speedUp diminuisce con l'aumentare dei processori. Questo suggerisce la presenza di overhead di comunicazione o altri fattori che potrebbero compromettere l'efficienza della parallelizzazione in queste configurazioni specifiche.

### **Efficienza :**

L'efficienza, che misura quanto bene il sistema sfrutta il parallelismo rispetto al numero di processori utilizzati, ci ha rilevato che alcuni valori sono inferiori a 1. Questo indica che, all'aumentare del numero di processori, l'efficienza diminuisce. Tale fenomeno potrebbe essere attribuito a problemi di overhead o inefficienza nell'implementazione parallela.

In sintesi, i risultati suggeriscono che l'implementazione parallela potrebbe trarre vantaggio da ulteriori ottimizzazioni, soprattutto quando si lavora con matrici di dimensioni maggiori. Inoltre, non è possibile effettuare un'analisi

approfondita in quanto sul cluster è possibile utilizzare solo 1 o 4 processi, poiché il numero massimo di processi utilizzabili è 8 e la griglia dei processi deve essere quadrata.

# 7 Codice Sorgente

## 7.1 main.c

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <mpi.h>
4. #include <time.h>
5. #include "utils.h"
6.
7. #define MIN_RANDOM_NUMBER 0
8. #define MAX_RANDOM_NUMBER 15
9. #define MASTER_ID 0
10.
11. #define TAG_MATRIX_FIRST 1
12. #define TAG_MATRIX_SECOND 1
13.
14. void distributeMatrixes(int matrixSize, int gridSize, MPI_Comm mpi_comm);
15. double** receiveMatrix(int matrixSize, int tag, MPI_Comm mpi_comm);
16. double** getPartialMatrixResult(int numberOfValues, int gridSize, double*
startTime, MPI_Comm mpi_comm_grid);
17. void printResult(int matrixSize, int gridSize, double** partialResult,
MPI_Comm mpi_comm);
18. int validateInput(int numberOfProcessors, int *matrixSize, int argc,
char** argv, int processId);
19.
20. /**
21. * Gli argomenti vengono passati nella forma: matrixSize
22. * @param matrixSize la dimensione della matrice MxM
23. */
24. int main(int argc, char** argv) {
25.     int matrixSize, processId, numberOfProcessors;
26.
27.     MPI_Init(&argc, &argv);
28.     MPI_Comm_rank(MPI_COMM_WORLD, &processId);
29.     MPI_Comm_size(MPI_COMM_WORLD, &numberOfProcessors);
30.
31.     /* I tempi dei singoli processori, ottenuti tramite differenza,
mentre il tempo totale come il
32.      * massimo valore tra i processori */
33.     double startTime, endTime, processorTime, totalTime = 0.0;
34.
35.     // ****
36.     // ***** Controllo input
37.     // ****
38.
39.     // Validazione dei parametri di input
40.     if (validateInput(numberOfProcessors, &matrixSize, argc, argv,
processId)) {
41.         // Errore nei parametri di input, esce dal programma
42.         MPI_Finalize();
43.         return 1;
44.     }
45.
46.     // ****
47.     // ***** Creazione della griglia di processori
48.     // ****
49.
50.     int gridSize;
```

```

52.     MPI_Bcast(&gridSize, 1, MPI_DOUBLE, MASTER_ID, MPI_COMM_WORLD);
53.
54.     gridSize = (int) sqrt(numberOfProcessors);
55.     int numberOfValues = matrixSize/gridSize;
56.
57.     // Siccome è una matrice quadrata le due dimensioni sono uguali
58.     int* sizesOfGrid = calloc(2, sizeof(int));
59.     sizesOfGrid[0] = sizesOfGrid[1] = gridSize;
60.
61.     int* gridPeriod = calloc(2, sizeof(int));
62.     gridPeriod[0] = gridPeriod[1] = 0;
63.
64.     MPI_Comm mpi_comm_grid;
65.     MPI_Cart_create(MPI_COMM_WORLD, 2, sizesOfGrid, gridPeriod, 0,
&mpi_comm_grid);
66.     MPI_Comm_rank(mpi_comm_grid, &processId);
67.
68.     free(sizesOfGrid);
69.     free(gridPeriod);
70.
71.     // *****
72.     // ***** Calcolo prodotto matrice x matrice
73.     // *****
74.
75.     // Il processore con id MASTER_ID invia le matrici generate
casualmente
76.     if(processId == MASTER_ID) {
77.         distributeMatrixes(matrixSize, gridSize, mpi_comm_grid);
78.     }
79.
80.     // Effettuo il prodotto parziale della matrice
81.     double** partialResult = getPartialMatrixResult(numberOfValues,
gridSize, &startTime, mpi_comm_grid);
82.
83.     // Effettua la stampa del risultato finale
84.     printResult(matrixSize, gridSize, partialResult, mpi_comm_grid);
85.
86.     // Salvo l'istante di tempo finale di esecuzione dalla somma
parziale, faccio stampare al MASTER_ID
87.     endTime = MPI_Wtime();
88.     processorTime = endTime - startTime;
89.
90.     // Passa al masterId il tempo maggiore impiegato
91.     MPI_Reduce(&processorTime, &totalTime, 1, MPI_DOUBLE, MPI_MAX,
MASTER_ID, MPI_COMM_WORLD);
92.
93.     // Il MASTER_ID fa la stampa del tempo impiegato
94.     if(processId == MASTER_ID) {
95.         printf("\nTime elapsed: %f seconds\n", totalTime);
96.     }
97.
98.     // Libero la memoria dinamica utilizzata
99.     int i;
100.    for(i = 0; i < numberOfValues; ++i) {
101.        free(partialResult[i]);
102.    }
103.    free(partialResult);
104.
105.    MPI_Finalize();
106.    return 0;
107. }
108.

```

```

109. /**
110. * Distribuisce le matrici generate casualmente tra tutti i processori,
111. * compreso il chiamante
112. * @param matrixSize la dimensione della matrice da generare casualmente
113. * @param gridSize la dimensione della matrice di processori
114. * @param mpi_comm il Communicator, deve essere una griglia di dimensione
115. * gridSize
116. */
117. void distributeMatrixes(int matrixSize, int gridSize, MPI_Comm mpi_comm)
118. {
119.     int i, j, k, l;
120.     int numberofValues = matrixSize/gridSize;
121.
122.     // Inizializzazione del seed dei valori casuali
123.     srand(time(NULL));
124.
125.     // Generazione e stampa delle due matrici casuali
126.     printf("First random matrix:\n");
127.     double** firstMatrix = getMatrixOfRandomNumbersOfSize(matrixSize,
matrixSize, MIN_RANDOM_NUMBER, MAX_RANDOM_NUMBER);
128.     printSquareMatrix(firstMatrix, matrixSize);
129.
130.     printf("\nSecond random matrix:\n");
131.     double** secondMatrix = getMatrixOfRandomNumbersOfSize(matrixSize,
matrixSize, MIN_RANDOM_NUMBER, MAX_RANDOM_NUMBER);
132.     printSquareMatrix(secondMatrix, matrixSize);
133.
134.     // Distribuzione di porzioni delle due matrici firstMatrix e
135.     // secondMatrix a tutti i processori
136.     for(i = 0; i < gridSize; ++i) {
137.         for(j = 0; j < gridSize; ++j) {
138.             int receiverCoordinate[2] = { i, j };
139.             int receiverProcessor;
140.             MPI_Cart_rank(mpi_comm, receiverCoordinate,
&receiverProcessor);
141.
142.             MPI_Send(&firstMatrix[k][l], 1, MPI_DOUBLE,
receiverProcessor, TAG_MATRIX_FIRST, mpi_comm);
143.             MPI_Send(&secondMatrix[k][l], 1, MPI_DOUBLE,
receiverProcessor, TAG_MATRIX_SECOND, mpi_comm);
144.         }
145.     }
146. }
147.
148. free(firstMatrix);
149. free(secondMatrix);
150. }
151.
152. /**
153. * Riceve una matrice inviata con un determinato tag
154. * @param matrixSize il numero M di righe e colonne della matrice MxM
155. * @param tag il tag da cui ricevere
156. * @param mpi_comm il Communicator
157. * @return la matrice ricevuta
158. */

```

```

159. double** receiveMatrix(int matrixSize, int tag, MPI_Comm mpi_comm) {
160.     int i, j;
161.     MPI_Status status;
162.
163.     double** matrix = malloc(sizeof(double*) * matrixSize);
164.     for(i = 0; i < matrixSize; ++i) {
165.         matrix[i] = malloc(sizeof(double*) * matrixSize);
166.
167.         for(j = 0; j < matrixSize; ++j) {
168.             MPI_Recv(&matrix[i][j], 1, MPI_DOUBLE, MASTER_ID, tag,
169.                      &status);
170.         }
171.
172.     return matrix;
173. }
174.
175. /**
176. * Effettua un prodotto parziale matrice x matrice
177. * @param numberofValues la dimensione parziale della matrice
178. * @param gridSize la dimensione della griglia dei processori
179. * @param startTime il valore in output dell'istante di inizio dei
calcoli
180. * @param mpi_comm_grid il Communicator, deve essere una griglia di
dimensione gridSize
181. * @return il prodotto parziale, una matrice di dimensione numberofValues
x numberofValues
182. */
183. double** getPartialMatrixResult(int numberofValues, int gridSize, double*
startTime, MPI_Comm mpi_comm_grid) {
184.     int i, j, k;
185.     MPI_Status status;
186.
187.     int processId;
188.     MPI_Comm_rank(mpi_comm_grid, &processId);
189.
190.     int* coordinates = calloc(2, sizeof(int));
191.     MPI_Cart_coords(mpi_comm_grid, processId, 2, coordinates);
192.
193.     // Tutti i processori ricevono una porzione di matrice
194.     double** matrixOne = receiveMatrix(numberofValues, TAG_MATRIX_FIRST,
195.                                         mpi_comm_grid);
196.     double** matrixTwo = receiveMatrix(numberofValues, TAG_MATRIX_SECOND,
197.                                         mpi_comm_grid);
198.
199.     // Alloco la matrice risultato parziale
200.     double** result = calloc(numberofValues, sizeof(double*));
201.     for(i = 0; i < numberofValues; ++i) {
202.         result[i] = calloc(numberofValues, sizeof(double*));
203.
204.         // Mi sincronizzo con tutti i processori che hanno ricevuto le loro
porzioni e salvo l'istante di tempo iniziale
205.         MPI_Barrier(MPI_COMM_WORLD);
206.         *startTime = MPI_Wtime();
207.
208.         // Se è sulla diagonale invia alle varie righe i suoi valori,
altrimenti riceve dalla diagonale
209.         int iteration;
210.         for(iteration = 0; iteration < gridSize; ++iteration) {
211.             double** receivedMatrix = NULL;

```

```

212.         if(coordinates[0] == mod(coordinates[1] - iteration, gridSize)) {
213.
214.             int rowProcessor;
215.             for(rowProcessor = 0; rowProcessor < gridSize;
216.                 ++rowProcessor) {
216.                 int receiverCoordinate[2] = { coordinates[0],
217.                     rowProcessor };
217.                     int receiverId;
218.                     MPI_Cart_rank(mpi_comm_grid, receiverCoordinate,
219. &receiverId);
219.
220.                     // Non invia a se stesso
221.                     if(processId != receiverId) {
222.                         //printf("Sono %d, devo inviare a %d che SULLA
222. GRIGLIA sta a (%d, %d)\n\n", processId, receiverId, coordinates[0],
223. rowProcessor);
223.                         for(i = 0; i < numberOfRowsValues; ++i) {
224.                             for(j = 0; j < numberOfRowsValues; ++j) {
225.                                 MPI_Send(&matrixOne[i][j], 1, MPI_DOUBLE,
226. receiverId, coordinates[0], mpi_comm_grid);
226.                             }
227.                         }
228.                     }
229.                 }
230.             } else {
231.
232.                 // Calcolo da chi devo ricevere
233.                 int diagonalCoordinates[2] = {coordinates[0],
233. mod(coordinates[0] + iteration, gridSize)};
234.                 int diagonalProcessor;
235.                 MPI_Cart_rank(mpi_comm_grid, diagonalCoordinates,
235. &diagonalProcessor);
236.
237.                 receivedMatrix = malloc(sizeof(double*) * numberOfRowsValues);
238.
239.                 for(i = 0; i < numberOfRowsValues; ++i) {
240.                     receivedMatrix[i] = malloc(sizeof(double*) *
240. numberOfRowsValues);
241.
242.                     for(j = 0; j < numberOfRowsValues; ++j) {
243.                         MPI_Recv(&receivedMatrix[i][j], 1, MPI_DOUBLE,
243. diagonalProcessor, coordinates[0], mpi_comm_grid, &status);
244.                     }
245.                 }
246.             }
247.
248.             // Non è necessario eseguirlo la prima volta
249.             if(iteration > 0) {
250.                 // Invio la mia matrice B al processore (i - 1, j), nella
250. riga precedente
251.                 int sendTo;
252.                 int sendCoordinate[2] = { mod(coordinates[0] - 1, gridSize),
252. coordinates[1] };
253.                 MPI_Cart_rank(mpi_comm_grid, sendCoordinate, &sendTo);
254.
255.                 for(i = 0; i < numberOfRowsValues; ++i) {
256.                     for(j = 0; j < numberOfRowsValues; ++j) {
257.                         MPI_Send(&matrixTwo[i][j], 1, MPI_DOUBLE, sendTo,
257. coordinates[1], mpi_comm_grid);
258.                     }
259.                 }
260.

```

```

261.          // Ricevo la matrice B dal processore (i + 1, j), nella riga
successiva
262.          int receiveFrom;
263.          int receiveCoordinate[2] = { mod(coordinates[0] + 1,
gridSize), coordinates[1] };
264.          MPI_Cart_rank(MPI_COMM_GRID, receiveCoordinate,
&receiveFrom);
265.
266.          for(i = 0; i < numberofValues; ++i) {
267.              for(j = 0; j < numberofValues; ++j) {
268.                  MPI_Recv(&matrixTwo[i][j], 1, MPI_DOUBLE,
receiveFrom, coordinates[1], MPI_COMM_GRID, &status);
269.              }
270.          }
271.      }
272.
273.      // Creo un puntatore alla matrice che voglio usare, per i
processori diagonali è MatrixOne altrimenti quella ricevuta
274.      double** matrixToUse = (receivedMatrix == NULL) ? matrixOne :
receivedMatrix;
275.
276.      // Calcolo del prodotto parziale
277.      for(i = 0; i < numberofValues; ++i) {
278.          for(j = 0; j < numberofValues; ++j) {
279.              for(k = 0; k < numberofValues; ++k) {
280.                  result[i][j] += (matrixToUse[i][k] *
matrixTwo[k][j]);
281.              }
282.          }
283.      }
284.
285.      // Pulisco le matrici che allo step successivo non servono più
286.      if(receivedMatrix != NULL) {
287.          for(i = 0; i < numberofValues; ++i) {
288.              free(receivedMatrix[i]);
289.          }
290.          free(receivedMatrix);
291.      }
292.  }
293.
294.  for(i = 0; i < numberofValues; ++i) {
295.      free(matrixOne[i]);
296.      free(matrixTwo[i]);
297.  }
298.  free(matrixOne);
299.  free(matrixTwo);
300.  free(coordinates);
301.
302.  return result;
303. }
304.
305. /**
306. * Verifica la correttezza dei parametri di input
307. * @param numberofProcessors il numero di processori
308. * @param matrixSize la dimensione della matrice (output)
309. * @param argc il numero di argomenti passati al programma
310. * @param argv gli argomenti del programma
311. * @param processId l'ID del processo
312. * @return 0 se i parametri sono validi, 1 altrimenti
313. */
314. int validateInput(int numberofProcessors, int *matrixSize, int argc,
char** argv, int processId) {

```

```

315.     // Check if there are enough arguments
316.     if (argc != 2) {
317.         if (processId == MASTER_ID) {
318.             fprintf(stdout, "Provide only one argument: the number of
rows (and columns) of the matrices\n", argv[0]);
319.         }
320.         return 1;
321.     }
322.
323.
324.     // Tests whether the number of processors is a square root of
integers
325.     if (!isPerfectSquare(numberOfProcessors)) {
326.         if (processId == MASTER_ID) {
327.             fprintf(stdout, "ERROR - Number of processors provided cannot
be distributed in a NxN grid.\n"
328.                     "number of processors: %d\n\n",
numberOfProcessors);
329.         }
330.         return 1;
331.     }
332.
333.     // Check the correctness of the size of the matrix passed as input
334.     if (!parseInt(argv[1], matrixSize)) {
335.         if (processId == MASTER_ID) {
336.             fprintf(stdout, "ERROR - Cannot parse the matrixSize with
value provided.\n"
337.                     "matrixSize:%s\n\n", argv[1]);
338.         }
339.         return 1;
340.     }
341.
342.     //Check that matrixSize is equal to numberOfProcessors or a multiple
thereof
343.     if (*matrixSize < numberOfProcessors || *matrixSize %
numberOfProcessors != 0) {
344.         if (processId == MASTER_ID) {
345.             fprintf(stdout, "ERROR - Matrix size must be >= of the grid
and a multiple of number of processors.\n"
346.                     "matrixSize:%d\nnumberOfProcessors:%d\n",
*matrixSize, numberOfProcessors);
347.         }
348.         return 1;
349.     }
350.
351.     return 0;
352. }
353.
354. /**
355. * Stampa il risultato del prodotto matrice per matrice
356. * @param matrixSize la dimensione originale della matrice
357. * @param gridSize la dimensione della griglia di processori
358. * @param partialResult il risultato parziale del chiamante
359. * @param mpi_comm il Communicator, deve essere una griglia di dimensione
gridSize
360. */
361. void printResult(int matrixSize, int gridSize, double** partialResult,
MPI_Comm mpi_comm_grid) {
362.     int i, j, k, l;
363.     MPI_Status status;
364.     int numberofValues = matrixSize/gridSize;
365.     int processId;

```

```

366.     MPI_Comm_rank(MPI_COMM_WORLD, &processId);
367.
368.     // Tutti i processori inviano al MASTER_ID il proprio prodotto
369.     for(i = 0; i < numberOfRows; i++) {
370.         for(j = 0; j < numberOfRows; j++) {
371.             MPI_Send(&partialResult[i][j], 1, MPI_DOUBLE, MASTER_ID,
372.                      MPI_COMM_WORLD, mpi_comm_grid);
373.         }
374.
375.     // Il MASTER_ID riceve tutta la matrice e la stampa ordinata
376.     if(processId == MASTER_ID) {
377.
378.         double** resultMatrix = malloc(sizeof(double*) * matrixSize);
379.         for (i = 0; i < matrixSize; ++i) {
380.             resultMatrix[i] = malloc(sizeof(double*) * matrixSize);
381.         }
382.
383.         for (i = 0; i < gridSize; ++i) {
384.             for (j = 0; j < gridSize; ++j) {
385.                 int receiverCoordinate[2] = {i, j};
386.                 int receiverProcessor;
387.                 MPI_Cart_rank(MPI_COMM_WORLD, receiverCoordinate,
388.                               &receiverProcessor);
389.                 for (k = (i * numberOfRows); k < (i * numberOfRows) +
390.                     numberOfRows; ++k) {
391.                     for (l = (j * numberOfRows); l < (j *
392.                         numberOfRows) + numberOfRows; ++l) {
393.                             MPI_Recv(&resultMatrix[k][l], 1, MPI_DOUBLE,
394.                                     receiverProcessor, MASTER_ID, MPI_COMM_WORLD, &status);
395.                         }
396.                     }
397.                 printf("\nRESULT: \n");
398.                 printSquareMatrix(resultMatrix, matrixSize);
399.
400.                 for (i = 0; i < matrixSize; ++i) {
401.                     free(resultMatrix[i]);
402.                 }
403.                 free(resultMatrix);
404.             }
405.         }
406.     }
407.

```

## 7.2 utils.h

```

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <errno.h>
4. #include <stdbool.h>
5. #include <limits.h>
6. #include <math.h>
7.
8. #ifndef Elaborato4_UTILS_H
9. #define Elaborato4_UTILS_H
10.

```

```

11. /* ****
12. * Allocazioni di matrici e vettori casuali con generatore casuale *
13. * **** */
14.
15. /**
16. * Ritorna un numero casuale nel range definito
17. * @param min il numero minimo, incluso
18. * @param max il numero massimo, incluso
19. * @return float il numero casuale
20. */
21. double getRandomDoubleNumberInRange(int min, int max);
22.
23. /**
24. * Alloca una matrice di dimensione column*row con valori casuali
25. * @param column il numero di colonne
26. * @param row il numero di righe
27. * @return la matrice allocata e riempita casualmente
28. */
29. double** getMatrixOfRandomNumbersOfSize(int column, int row, int min, int
max);
30.
31. /**
32. * Stampa la matrice in ingresso in standard output
33. * @param matrix la matrice da stampare
34. */
35. void printSquareMatrix(double** matrix, int dimension);
36.
37.
38.
39. /* ****
40. * Funzioni per il parsing di argomenti ottenuti da riga di comando *
41. * **** */
42.
43. /**
44. * Converte una stringa in input in int
45. * @param str la stringa da convertire
46. * @param val dove viene salvato il risultato della conversione
47. * @return true se la conversione termina con successo, falso altrimenti
48. */
49. bool parseInt(char* arg, int* output);
50.
51.
52.
53. /* ****
54. * Funzioni matematiche
55. * **** */
56.
57. /**
58. * Ritorna se il numero in ingresso ha una radice quadrata intera
59. * @param number il valore da verificare
60. * @return true se ha radice intera, false altrimenti
61. */
62. bool isPerfectSquare(int number);
63.
64. /**
65. * Ritorna il modulo b di a
66. * @param a il valore da cui ottenere il modulo b
67. * @param b il modulo da utilizzare
68. * @return a modulo b
69. */
70. int mod(int a, int b);
71.

```

```
72. #endif //Elaborato4_UTILS_H  
73.
```

### 7.3 utils.c

```
1. #include "utils.h"  
2.  
3. bool parseInt(char* str, int* val) {  
4.     char *temp;  
5.     bool result = true;  
6.     errno = 0;  
7.     long ret = strtol(str, &temp, 0);  
8.  
9.     if (temp == str || *temp != '\0' || ((ret == LONG_MIN || ret ==  
LONG_MAX) && errno == ERANGE))  
10.         result = false;  
11.  
12.     *val = (int) ret;  
13.     return result;  
14. }  
15.  
16. double getRandomDoubleNumberInRange(int min, int max) {  
17.     return (double) min + rand() / (double) RAND_MAX * max - min;  
18. }  
19.  
20. bool isPerfectSquare(int number) {  
21.     int s = sqrt(number);  
22.     return (s * s) == number;  
23. }  
24.  
25. int mod(int a, int b) {  
26.     int r = a % b;  
27.     return r < 0 ? r + b : r;  
28. }  
29.  
30. double** getMatrixOfRandomNumbersOfSize(int column, int row, int min, int  
max) {  
31.     int i, j;  
32.     double** matrix = malloc(sizeof(double*) * column);  
33.     for (i = 0; i < column; ++i) {  
34.         matrix[i] = malloc(sizeof(double*) * row);  
35.         for (j = 0; j < row; ++j) {  
36.             matrix[i][j] = getRandomDoubleNumberInRange(min, max);  
37.         }  
38.     }  
39.     return matrix;  
40. }  
41.  
42. void printSquareMatrix(double** matrix, int size) {  
43.     int i, j;  
44.  
45.     for(i = 0; i < size; ++i) {  
46.         for(j = 0; j < size; ++j) {  
47.             printf("%f ", matrix[i][j]);  
48.         }  
49.         printf("\n");  
50.     }  
51. }  
52.  
53.
```

## 7.4 job-script.pbs

```
1.#!/bin/bash
2.
3. # Imposta le direttive per l'ambiente PBS
4. #PBS -q studenti
5. #PBS -l nodes=4:ppn=4
6. #PBS -N matrix
7. #PBS -o matrix.out
8. #PBS -e matrix.err
9.
10.sort -u $PBS_NODEFILE > hostlist
11.
12.NCPU=$(wc -l < hostlist)
13.echo "[Job-Script] Starting with \"$NCPU\" CPUs..."
14.
15.echo "[Job-Script] Compiling..."
16.PBS_O_WORKDIR=$PBS_O_HOME/Elaborato4
17./usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS_O_WORKDIR/matrix
$PBS_O_WORKDIR/main.c $PBS_O_WORKDIR/utils.c
18.
19.echo "[Job-Script] Checking input the values..."
20.#####
21.## CUSTOM VALUES ##
22.#####
23.# La dimensione della matrice, deve essere uguale o un multiplo del numero
di processori
24.MATRIX_SIZE=4
25.#####
26.
27.echo "[Job-Script] Running the process..."
28./usr/lib64/openmpi/1.4-gcc/bin/mpirun -machinefile hostlist --np $NCPU
$PBS_O_WORKDIR/matrix $MATRIX_SIZE
```