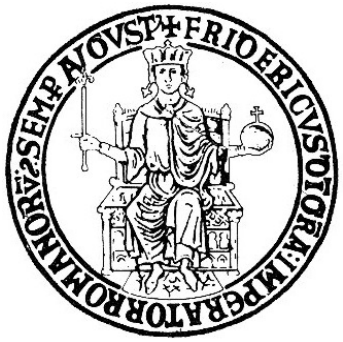
Università degli Studi di Napoli Federico II
Scuola Politecnica e delle Scienze di Base
Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione



Corso di Laurea Magistrale in Informatica
Insegnamento di Parallel and Distributed Computing
Anno accademico 2020/2021

Sviluppo di un algoritmo per il calcolo del prodotto matrice-matrice, su architettura MIMD a memoria distribuita, che utilizzi la libreria MPI con strategia di comunicazione BMR.

Autori:
Salvatore Bazzicalupo, N97000345
Annarita Della Rocca, N97000341

Docenti:
Prof. Giuliano Laccetti
Dott.sa Valeria Mele

Questa pagina è stata lasciata intenzionalmente bianca.

Indice

Capitolo 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, utilizzando la libreria MPI.

L'algoritmo è composto da due parti principali: nella prima, formata da uno script PBS, 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, prevede l'implementazione di una topologia e la generazione di una matrice di dimensione presa in input, infine 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 *pxp*. Nel capitolo successivo verrà analizzata nel dettaglio la sua implementazione.

Capitolo 2

Descrizione algoritmo

L'algoritmo riceve un solo parametro in input:

- il numero N che sarà la grandezza della matrice NxN. Tale dimensione deve necessariamente essere un multiplo del numero di processori utilizzati per l'esecuzione in ambiente parallelo.

2.1 Job-Script PBS

L'algoritmo utilizza uno script PBS per ricevere il parametro in input specificato nella sezione precedente, nello script è presente la seguente porzione di codice:

```
1 ######################
2 ## CUSTOM VALUES ##
3 ######################
4 # La dimensione della matrice, deve essere uguale o un multiplo del
      numero di processori
5 MATRIX_SIZE=16
6 ######################
```

2.2 Algoritmo in C

L'algoritmo prevede, oltre alle operazioni standard della libreria MPI, una fase di controlli di consistenza sul parametro passato in input.
Nel dettaglio viene controllato che:

- il numero di processori con il quale viene lanciato lo script ha una radice quadrata intera. Questo permette la generazione di una topologia a *griglia bidimensionale* di dimensione *pxp*;
- il valore passato in ingresso sia un intero;
- il valore passato in ingresso `matrixSize` sia un multiplo del numero di processori `numberOfProcessors` utilizzati.

Dopo aver appurato che i valori in ingresso siano validi, viene inviato il valore `gridSize` a tutti i processori coinvolti, mediante la funzione `MPI_Bcast(...)`.

2.3 Creazione della griglia bidimensionale

Di seguito viene illustrata l'implementazione della griglia bidimensionale mediante le opportune funzioni messe a disposizione dalla libreria MPI.

```
1  // ******************************************
2  // ****** Creazione della griglia di processori
3  // ******************************************
4
5  MPI_Bcast(&gridSize, 1, MPI_DOUBLE, MASTER_ID, MPI_COMM_WORLD);
6
7  gridSize = (int) sqrt(numberOfProcessors);
8  int numberOfValues = matrixSize/gridSize;
9
10 // Siccome è una matrice quadrata le due dimensioni sono uguali
11 int* sizesOfGrid = calloc(2, sizeof(int));
12 sizesOfGrid[0] = sizesOfGrid[1] = gridSize;
13
14 int* gridPeriod = calloc(2, sizeof(int));
15 gridPeriod[0] = gridPeriod[1] = 0;
16
17 MPI_Comm mpi_comm_grid;
18 MPI_Cart_create(MPI_COMM_WORLD, 2, sizesOfGrid, gridPeriod, 0, &
       mpi_comm_grid);
19 MPI_Comm_rank(mpi_comm_grid, &processId);
```

Nel dettaglio:

- `numberOfValues` indica quanti valori dovranno essere partizionati per ogni processore;
- `gridPeriod` permette di impostare se la griglia termina con un valore periodico, in questo caso tale valore è settato a 0 poiché la grandezza della matrice è un multiplo della grandezza della griglia;
- `MPI_Cart_create` permette la creazione della griglia con i parametri specificati;
- `MPI_Comm_rank` restituisce l'id del processore nella griglia appena creata.

2.4 Calcolo del prodotto matrice-matrice

```
1  // ****************************************
2  // ****** Calcolo prodotto matrice x matrice
3  // ****************************************
4
5  // Il processore con id MASTER_ID invia le matrici generate
       casualmente
6  if(processId == MASTER_ID) {
7      distributeMatrixes(matrixSize, gridSize, mpi_comm_grid);
8  }
9
10 // Effettuo il prodotto parziale della matrice
11 double** partialResult = getPartialMatrixResult(numberOfValues,
       gridSize, &startTime, mpi_comm_grid);
12
13 // Effettua la stampa del risultato finale
14 printResult(matrixSize, gridSize, partialResult, mpi_comm_grid);
15
16 // Salvo l'istante di tempo finale di esecuzione dalla somma
       parziale, faccio stampare al MASTER_ID
17 endTime = MPI_Wtime();
18 processorTime = endTime - startTime;
19
20 // Passa al masterId il tempo maggiore impiegato
```

```
21 MPI_Reduce(&processorTime, &totalTime, 1, MPI_DOUBLE, MPI_MAX,
       MASTER_ID, MPI_COMM_WORLD);
22
23 // Il MASTER_ID fa la stampa del tempo impiegato
24 if(processId == MASTER_ID) {
25     printf("\nTime elapsed: %e seconds\n", totalTime);
26 }
```

Nel dettaglio:

- nella funzione `distributeMatrixes`, eseguita soltanto da un processore, vengono generate in maniera casuale le matrici di dimensione `matrixSize`, vengono poi distribuite porzioni di matrici ai processori coinvolti nel calcolo;
- nella funzione `getPartialMatrixResult`, ogni processore riceve la propria porzione di matrice ed esegue su di essa il calcolo parziale. La funzione ritorna il risultato di tale calcolo.

L'agoritmo prevede un ciclo for esterno che viene eseguito sulla grandezza `gridSize` della griglia.

Innanzitutto viene verificata la condizione in cui, se il processore si trova sulla diagonale, deve inviare agli altri processori della riga la propria matrice A altrimenti deve riceverla.

```
1 if(coordinates[0] == mod(coordinates[1] - iteration, gridSize))
```

Si noti che la condizione di diagonalità viene "shiftata" a destra ad ogni iterazione per coprire tutta la matrice.

Una volta ottenuta la matrice A, se l'iterazione non è la prima, è necessario inviare la propria porzione di matrice B al processore nella riga precedente e ricevere quella del processore nella riga successiva. Questo viene fatto nel seguente modo:

```
1 int sendTo, receiveFrom
2 int sendCoordinate[2] = { mod(coordinates[0] - 1, gridSize),
      coordinates[1] };
3 int receiveCoordinate[2] = { mod(coordinates[0] + 1, gridSize),
      coordinates[1] };
4
```

```
// Invio la mia matrice B al processore (i - 1, j), nella riga precedente
MPI_Cart_rank(mpi_comm_grid, sendCoordinate, &sendTo);
for(i = 0; i < numberOfValues; ++i) {
    for(j = 0; j < numberOfValues; ++j) {
        MPI_Send(&matrixTwo[i][j], 1, MPI_DOUBLE, sendTo, coordinates[1], mpi_comm_grid);
    }
}

// Ricevo la matrice B dal processore (i + 1, j), nella riga successiva
MPI_Cart_rank(mpi_comm_grid, receiveCoordinate, &receiveFrom);
for(i = 0; i < numberOfValues; ++i) {
    for(j = 0; j < numberOfValues; ++j) {
        MPI_Recv(&matrixTwo[i][j], 1, MPI_DOUBLE, receiveFrom, coordinates[1], mpi_comm_grid, &status);
    }
}
```

Viene poi effettuato il calcolo parziale come segue:

```
// Creo un puntatore alla matrice che voglio usare, per i processori diagonali è MatrixOne altrimenti quella ricevuta
double** matrixToUse;
if(receivedMatrix == NULL) {
    matrixToUse = matrixOne
} else {
    matrixToUse = receivedMatrix;
}

for(i = 0; i < numberOfValues; ++i) {
    for(j = 0; j < numberOfValues; ++j) {
        for(k = 0; k < numberOfValues; ++k) {
            result[i][j] += (matrixToUse[i][k] * matrixTwo[k][j]);
        }
    }
}
```

Capitolo 3

Input ed Output

L'algoritmo opera su numeri reali; richiede il seguente valore da riga di comando:

- il numero N che sarà la grandezza della matrice NxN. È un multiplo del numero di processori utilizzati per l'esecuzione in ambiente parallelo.

L'output dell'algoritmo, è della seguente forma:

```
First random matrix:
<valori-prima-matrice>

Second random matrix:
<valori-seconda-matrice>

RESULT:
<valori-matrice-risultato>

Time elapsed: <ammontare-secondi> seconds
```

3.1 Input/Output per valori casuali

Per poter eseguire il programma, va specificato il numero di processori da utilizzare e la grandezza della matrice come di seguito:

```
~/opt/usr/local/bin/mpiexec -np 4 ./main 4
```

Si riceverà il seguente ouput:

```
First random matrix:
5.318026 10.059348 2.462740 6.272614
```

```
3.818302 4.197258 13.321763 8.864349
3.115805 2.328532 0.645060 11.527440
1.686453 9.218279 11.622541 10.050312

Second random matrix:
0.588249 1.697868 6.072526 0.941970
6.687669 4.658447 9.521524 8.248315
14.431044 7.555654 12.874255 2.601360
11.063038 11.478053 11.631317 7.541606

RESULT:
82.906395 135.251682 165.713996 129.929276
64.797829 155.389166 231.429546 189.847596
192.234245 242.971428 306.317335 253.294458
84.024625 208.157127 258.118741 238.190332

Time elapsed: 7.510000e-04 seconds
```

Si noti che il tempo impiegato per il calcolo del prodotto, può differire in base a vari fattori, tra cui il numero di processori, la potenza di calcolo dei processori, e così via. Inoltre il calcolo della matrice risultante potrà differire in base ai valori generati casualmente.

3.2 Input/Output attraverso lo script PBS

È possibile personalizzare il valore in input attraverso lo script PBS.
Al suo interno è contenuta una variabile da modificare opportunamente, come descritto in precedenza.

Utilizzando la seguente configurazione:

```
1 ######################
2 ## CUSTOM VALUES ##
3 ######################
4 MATRIX_SIZE=4
5 ######################
```

Avviando l'algoritmo, utilizzando:

```
qsub job-script.pbs
```

Si otterrà il seguente output:

```
First random matrix:
5.482614 1.299033 7.842666 6.684888
2.913305 3.908947 12.670042 5.389345
8.719219 8.920134 10.693750 14.853745
1.887682 1.269764 10.920726 4.645378

Second random matrix:
14.859796 13.590724 14.303661 11.634225
11.417712 2.488588 5.700323 0.327799
4.320751 3.867846 11.895907 14.517137
14.529512 12.516254 0.678382 1.566380

RESULT:
107.025408 392.557190 301.643029 63.848998
84.301557 291.398773 216.753097 34.618462
77.582720 414.573277 305.803387 77.006668
87.578755 508.990848 363.514892 84.931393

Time elapsed: 8.910000e-04 seconds
```

Anche in questo caso, come i precedenti, il tempo impiegato potrà variare.

Capitolo 4

Indicatori di errore

Tutti i messaggi di errore dell'applicativo seguono la seguente forma:
"ERROR - <descrizione-errore>: <parametri>"

Con i valori tra parentesi angolari:

- `<descrizione-errore>`: una sintetica descrizione dell'errore verificatosi;
- `<parametri>`: una lista della forma chiave:valore dei parametri che hanno causato l'errore.

In `main`:

```
1  // Se il numero di processori non è una radice quadrata di interi
       allora non è rappresentabile in una griglia pxp
2  if(!isPerfectSquare(numberOfProcessors)) {
3      if(processId == MASTER_ID) {
4          fprintf(stderr, "ERROR - Number of processors provided
       cannot be distribuited in a NxN grid.\nnumber of processors: %d\n
       \n", numberOfProcessors);
5      }
6      MPI_Finalize();
7      return 1;
8  }
9
10 // Verifico la correttezza della dimensione della matrice passata in
       input
11 if(!parseInt(argv[1], &matrixSize)) {
12     fprintf(stderr, "ERROR - Cannot parse the matrixSize with value
       provided.\nmatrixSize:%s\n\n", argv[1]);
```

```
13      MPI_Finalize();
14      return 1;
15  }
16
17  // MatrixSize dev'essere uguale a numberOfProcessors o un suo
        multiplo
18  if(matrixSize < numberOfProcessors || matrixSize %
        numberOfProcessors != 0) {
19      fprintf(stderr, "ERROR - Matrix size must be >= of the grid and
        a multiple of number of processors.\nmatrixSize:%d\
        nnumberOfProcessors:%d\n", matrixSize, numberOfProcessors);
20      MPI_Finalize();
21      return 1;
22  }
```

Capitolo 5

Subroutine

Nell'algoritmo sono presenti varie subroutine utilizzate, queste sono documentate a livello interno del codice tramite commenti che hanno la seguente forma:

- breve descrizione del metodo;
- 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 riportate le firme dei metodi, la loro documentazione interna ed eventualmente una descrizione aggiuntiva.

5.1 Subroutines personalizzate

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

```
1  //  Allocazioni di matrici e vettori casuali con generatore casuale
2
3  /**
4   * Ritorna un numero casuale nel range definito
5   * @param min il numero minimo, incluso
6   * @param max il numero massimo, incluso
7   * @return float il numero casuale
8   */
9  double getRandomDoubleNumberInRange(int min, int max);
10
11 /**
```

```
12  * Alloca una matrice di dimensione column*row con valori casuali
13  * @param column il numero di colonne
14  * @param row il numero di righe
15  * @return la matrice allocata e riempita casualmente
16  */
17 double** getMatrixOfRandomNumbersOfSize(int column, int row, int min
        , int max);
18
19 /**
20  * Stampa la matrice in ingresso in standard output
21  * @param matrix la matrice da stampare
22  */
23 void printSquareMatrix(double** matrix, int dimension);
24
25 // Funzioni per il parsing di argomenti ottenuti da riga di comando
26
27 /**
28  * Converte una stringa in input in int
29  * @param str la stringa da convertire
30  * @param val dove viene salvato il risultato della conversione
31  * @return true se la conversione termina con successo, falso
        altrimenti
32  */
33 bool parseInt(char* arg, int* output);
34
35 // Funzioni matematiche
36
37 /**
38  * Ritorna se il numero in ingresso ha una radice quadrata intera
39  * @param number il valore da verificare
40  * @return true se ha radice intera, false altrimenti
41  */
42 bool isPerfectSquare(int number);
43
44 /**
45  * Ritorna il modulo b di a
46  * @param a il valore da cui ottenere il modulo b
47  * @param b il modulo da utilizzare
48  * @return a modulo b
```

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

Di seguito vediamo alcune subroutine che riguardano parti principali del programma.

```
/**
 * Distribuisce le matrici generate casualmente tra tutti i
   processori, compreso il chiamante
 * @param matrixSize la dimensione della matrice da generare
   casualmente
 * @param gridSize la dimensione della matrice di processori
 * @param mpi_comm il Communicator, deve essere una griglia di
   dimensione gridSize
 */
void distributeMatrixes(int matrixSize, int gridSize, MPI_Comm
   mpi_comm)

/**
 * Riceve una matrice inviata con un determinato tag
 * @param matrixSize il numero M di righe e colonne della matrice
   MxM
 * @param tag il tag da cui ricevere
 * @param mpi_comm il Communicator
 * @return la matrice ricevuta
 */
double** receiveMatrix(int matrixSize, int tag, MPI_Comm mpi_comm)

/**
 * Effettua un prodotto parziale matrice x matrice
 * @param numberOfValues la dimensione parziale della matrice
 * @param gridSize la dimensione della griglia dei processori
 * @param startTime il valore in output dell'istante di inizio dei
   calcoli
 * @param mpi_comm_grid il Communicator, deve essere una griglia di
   dimensione gridSize
 * @return il prodotto parziale, una matrice di dimensione
   numberOfValues x numberOfValues
 */
double** getPartialMatrixResult(int numberOfValues, int gridSize,
```

```
    double* startTime, MPI_Comm mpi_comm_grid)

/**
 * Stampa il risultato del prodotto matrice per matrice
 * @param matrixSize la dimensione originale della matrice
 * @param gridSize la dimensione della griglia di processori
 * @param partialResult il risultato parziale del chiamante
 * @param mpi_comm il Communicator, deve essere una griglia di
    dimensione gridSize
 */
void printResult(int matrixSize, int gridSize, double**
    partialResult, MPI_Comm mpi_comm_grid)
```

5.2 Subroutine MPI

Seguono le subroutines MPI utilizzate nel progetto.
Esse sono documentate differentemente da quelle personalizzate in quanto la documentazione ufficiale è disponibile su open-mpi.org [1], viene però fornita una breve descrizione.

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

Descrizione: inizializza l'ambiente di esecuzione MPI.
Parametri in input:

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

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

Descrizione: assegna un identificativo (chiamato rank) al processo appartenente ad un communicator.
Parametri in input:

- **comm**: il communicator da cui si vuole ottenere il rank.

Parametri in output:

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

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

Descrizione: ritorna la dimensione del gruppo di processi appartenenti ad un communicator.
Parametri in input:

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

Parametri in output:

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

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

Descrizione: permette al processore con identificativo root di spedire a tutti i processori del comunicator comm lo stesso dato memorizzato in *buffer. Count, datatype, comm devono essere uguali per ogni processore di comm.
Parametri in output:

- **buffer** il valore che sarà condiviso tra i processori

```
double MPI_Wtime(void)
```

Descrizione: ritorna un valore temporale in secondi passato dall'avvio di un processo.
Parametri in output:

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

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

Descrizione: permette al processore root di ottenere il risultato dell'operazione op degli elementi memorizzati in *sendbuf. Tale risultato viene memorizzato in recvbuf.
Parametri 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

Parametri in output:

- **recvbuf**: il risultato dell'operazione.

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

Descrizione: effettua un invio di dati in modo sincrono (bloccante).
Parametri 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.

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

Descrizione: effettua una ricezione di dati in modo sincrono (bloccante).
Parametri 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

Parametri di output:

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

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

Descrizione: fornisce un meccanismo sincronizzante per tutti i processori del communicator comm.
Parametri in input:

- **comm**: il communicator

```
int MPI_Finalize()
```

Descrizione: Termina l'ambiente di esecuzione MPI.

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

Descrizione: operazione collettiva che restituisce un nuovo communicator new_comm in cui i processi sono organizzati in una griglia di dimensioni dim communicator comm.
Parametri in input:

- **old_comm**: il communicator precedentemente utilizzato
- **dim**: numero di dimensioni 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**: permesso di riordinare i menum (1=si; 0=no).

Parametri di output:

- **comm_cart**: il communicator associato alla griglia.

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

Descrizione: operazione collettiva che restituisce a ciascun processo di comm_grid con identificativo menum_grid, le sue coordinate all'interno della griglia predefinita. **coordinate** è un vettore di dimensione dim, i cui elementi rappresentano le coordinate del processo all'interno della griglia.
Parametri in input:

- **comm_grid**: il communicator precedentemente utilizzato
- **menum**: identificativo del processo;
- **dim**: dimensione del vettore **coordinate**.

Parametri di output:

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

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

Descrizione: funzione che, date delle coordinate, ritorna il menum associato a tali coordinate.
Parametri in input:

- **comm_grid**: il communicator precedentemente utilizzato
- **coords**: coordinate del processo di cui si vuole conoscere il **rank**.

Parametri di output:

- **menum**: identificativo del processo;

Capitolo 6

Analisi dei tempi

Sono state effettuate alcune analisi dei tempi di esecuzione dell'algoritmo in base al numero di processori utilizzati e alla grandezza della matrice.

I tempi che seguono sono una media di 10 esecuzioni per ogni strategia.

Sono stati testati prodotti effettuati con matrici di dimensioni 4x4, 8x8, 16x16, 32x32, 64x64, 128x128, 256x256, 512x512 e 1024x1024, di questi prodotti è stata calcolata la media per avere risultati più coerenti.

Sono stati raccolti tempi con 1, e 4 processori.

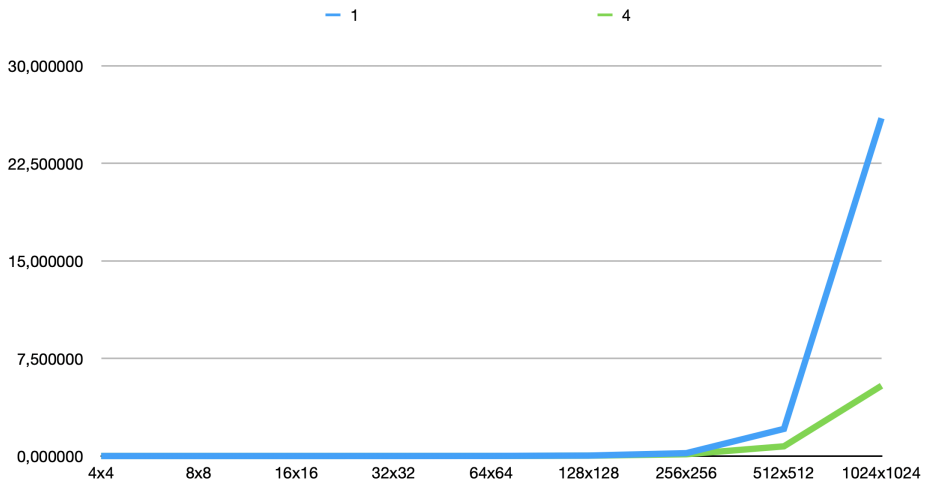
6.1 Analisi sul numero di processori

Seguono alcune analisi basate sul numero di processori.

I primi tempi sono stati raccolti utilizzando un solo processore, questo è essenziale anche per i calcoli successivi.

Tempi per processore

	4x4	8x8	16x16	32x32	64x64	128x128	256x256	512x512	1024x1024
1	0,000025	0,000046	0,000155	0,000747	0,005359	0,032438	0,220933	2,084789	25,966911
4	0,000381	0,000667	0,001044	0,002355	0,006112	0,025505	0,135436	0,744717	5,400459



Come si evince dal grafico, con la linea blu rappresentante l'esecuzione con un processore e quella verde con 4, fino a 256 non si nota nessuna differenza, mentre con 512x512 si inizia a notare e con 1024x1024 si impiega 3 volte il tempo necessario per calcolare il prodotto.

6.2 Speed-up ed efficienza

Conseguentemente alle precedenti analisi è possibile calcolare lo speed-up e l'efficienza del prodotto matriciale in rapporto al numero di processori.

6.2.1 Calcolo dello speed-up

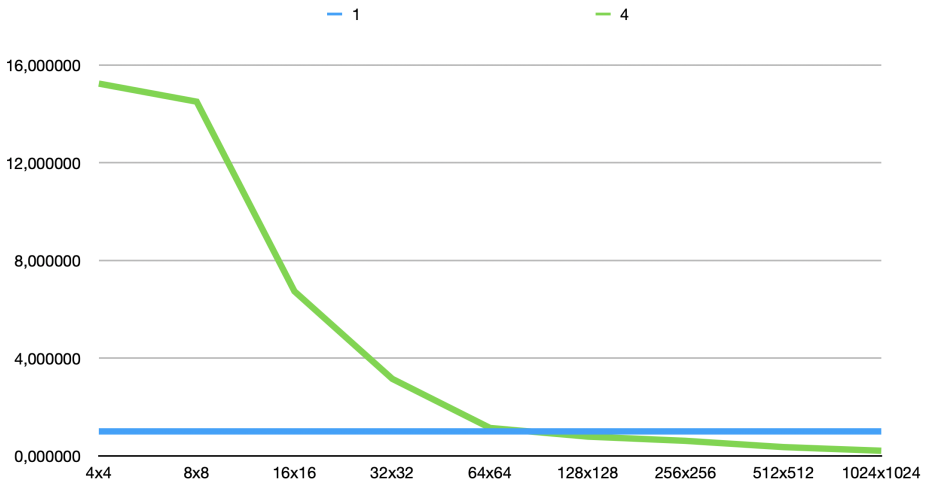
Lo speed-up misura la riduzione del tempo di esecuzione rispetto all'algoritmo su di un solo processore.

Definiamo lo speed-up come: $S(p) = \frac{T(1)}{T(p)}$, e cioè il rapporto tra il tempo impiegato dall'algoritmo per calcolare il risultato utilizzando un solo processore e il tempo impiegato per calcolare il risultato utilizzando p processori.

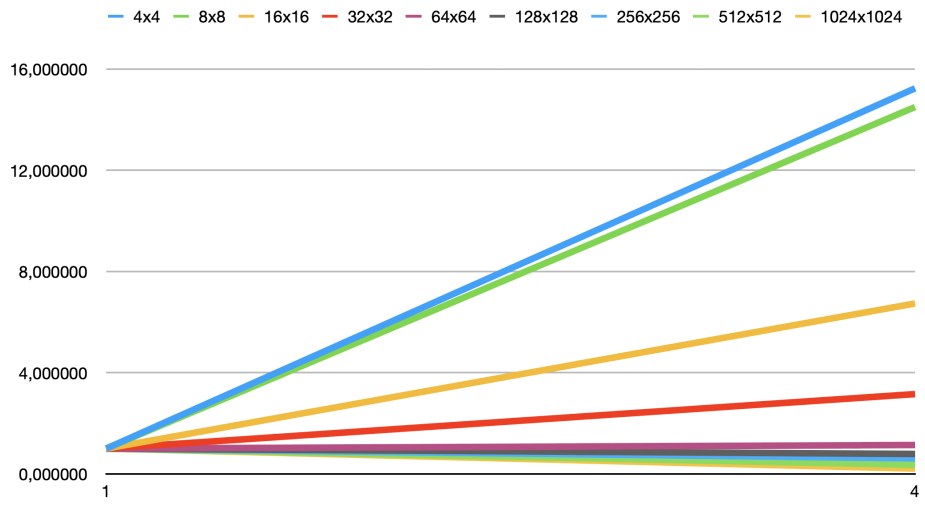
Segue la tabella contenente i valori relativi al calcolo dello speed-up:

Speed-up

	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	15,240000	14,500000	6,735484	3,152610	1,140511	0,786269	0,613018	0,357215	0,207975



Lo speed-up con 4 processori è inizialmente estremamente alto, ben 16 volte maggiore rispetto alla fase finale, per poi ridursi fino ad un quarto per una matrice 1024x1024.



In questo grafico, avente il numero di processori sulle ascisse, si può notare come le varie dimensioni della matrice siano differenti, la 4x4 con 1 processore impiega molto meno rispetto alla stessa con 4 processori.

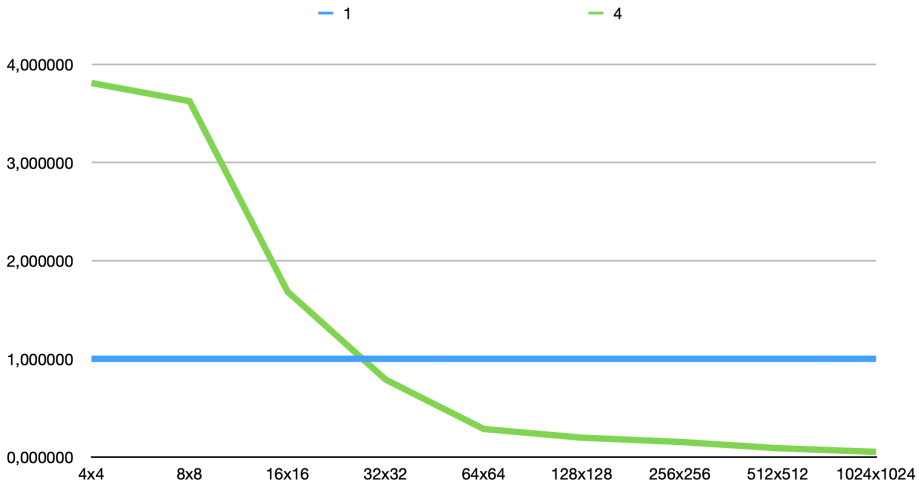
6.2.2 Calcolo dell'efficienza

L'efficienza misura quanto l'algoritmo sfrutta il parallelismo del calcolatore.
Definiamo l'efficienza come: $E(p) = \frac{S(p)}{p}$, e cioè il rapporto tra lo speed-up su p processori e il numero di processori

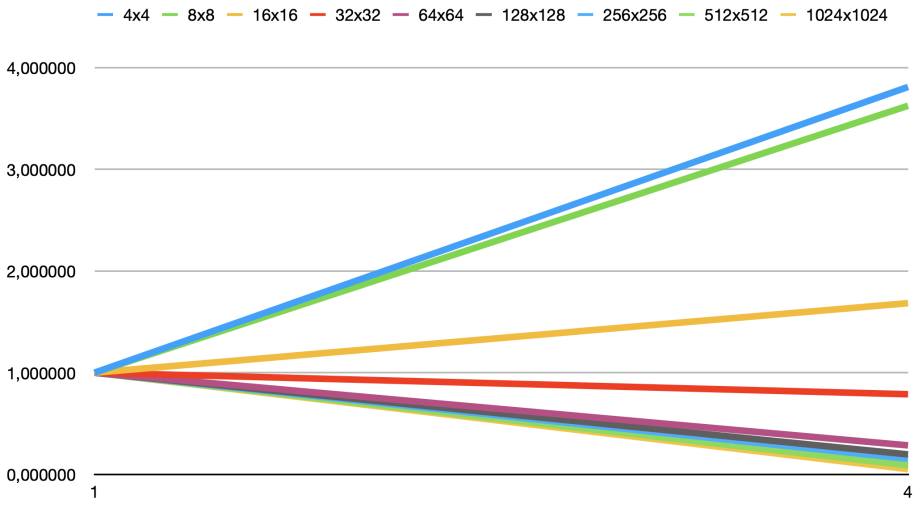
Segue la tabella contenente i valori relativi al calcolo dell'efficienza.

Efficienza

	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	3,810000	3,625000	1,683871	0,788153	0,285128	0,196567	0,153255	0,089304	0,051994



Si può notare che l'efficienza segue un andamento simile a quello dello speed-up precedentemente mostrato, qui però si raggiunge una migliore efficienza anche con una matrice 32x32.



Anche in questo caso, come per lo speed-up, si può notare come le matrici più piccole abbiano poca efficienza su matrici relativamente piccole.

Capitolo 7

Esempi d'uso

L'utilizzo dell'algoritmo può avvenire tramite due modi, o utilizzandolo in maniera diretta, quindi invocando in locale le librerie MPI, oppure utilizzando uno script PBS. Seguono esempi d'uso per entrambi i casi.

7.1 Uso diretto

Ipotizzando di avere open-mpi installato nel path /opt/usr/local/bin/, ed avere nella stessa cartella tutti i file richiesti in appendice, contenente il codice sorgente, è necessario compilarlo eseguendo:

```
~/opt/usr/local/bin/mpicc -o main main.c utils.c
```

Dopo aver compilato il file, ipotizzando di voler fare il prodotto tra due matrici 4x4 è possibile utilizzare 1 o 4 processori, si ipotizza di voler utilizzare 4 processori.

Sarà sufficiente digitare:

```
~/opt/usr/local/bin/mpiexec -np 4 ./main 4
```

Verrà stampato il seguente risultato:

```
First random matrix:
5.035924 8.779774 6.668902 4.228609
0.223229 1.813387 12.598571 4.185630
12.876856 1.323098 7.312677 9.164309
4.540553 8.066555 4.597906 12.006843

Second random matrix:
4.008700 9.216746 0.855394 6.613519
```

```
3.405720 14.936030 4.858432 10.664863
9.345476 4.407866 12.999767 12.092364
1.366220 12.063076 4.116982 14.113265

RESULT:
45.574074 96.454884 121.157952 106.747082
91.221965 253.119641 248.040722 251.698714
157.888397 350.247181 391.055858 375.804867
147.443275 228.733690 222.446591 261.829503

Time elapsed: 1.021000e-03 seconds
```

Si noti che il risultato può variare in quanto i numeri sono generati casualmente, così come il tempo di esecuzione può variare in base all'hardware del dispositivo.

7.2 Uso tramite script PBS

Utilizzando l'algoritmo su un cluster è necessario utilizzare uno script PBS, in appendice ne è fornito uno che verrà utilizzato in questo caso d'uso ed è chiamato `job-script.pbs`.

Si ipotizzi di voler effettuare il prodotto con le stesse condizioni precedenti, ovvero una matrice 4x4 con 4 processori.

È necessario modificare la seguente parte dello script PBS:

```
1  ...
2  #PBS -l nodes=4:ppn=4
3  #PBS -o matrix.out
4  ...
5  ######################
6  ## CUSTOM VALUES ##
7  ######################
8  # La dimensione della matrice, deve essere uguale o un multiplo del
       numero di processori
9  MATRIX_SIZE=4
10 ######################
```

Ed invocare lo script come segue:

```
qsub job-script.pbs
```

Attendere il completamento e, supponendo che il risultato sia memorizzato nel file matrix.out (come scritto nel file pbs) digitare:

```
cat matrix.out
```

Verrà stampato il seguente risultato:

```
First random matrix:
5.075604 0.676360 12.587885 4.584777
1.353123 1.936462 11.114775 11.029744
6.909634 0.216081 1.668210 2.605315
2.534857 3.347156 5.648396 12.586823

Second random matrix:
1.736077 3.246550 9.772744 0.503663
5.059611 1.873682 5.970011 2.977891
9.411610 5.936768 14.261764 12.461503
10.482959 12.099390 14.443592 8.457326

RESULT:
74.849152 279.135959 269.671836 299.936826
20.509614 63.661169 75.562364 84.186047
79.539829 242.640834 369.539690 213.988676
60.103405 190.304897 239.449309 161.990164

Time elapsed: 9.300000e-04 seconds
```

Si noti che il risultato può variare in quanto i numeri sono generati casualmente, così come il tempo di esecuzione può variare in base all'hardware del dispositivo.

Bibliografia

[1] Open MPI Documentation,
https://www.open-mpi.org/doc/

Capitolo A

Codice

Segue il codice dell'algoritmo, quest'ultimo è composto dai seguenti file:

- `main.c` il file principale contenente la logica della griglia e del prodotto matrice per matrice;
- `utils.c` un file contenente varie funzioni accessorie;
- `utils.h` un file header contenente import, costanti, dichiarazioni e documentazione delle funzioni contenute in utils.c;
- `job-script.pbs` file necessario per il lancio dell'algoritmo su cluster.

A.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
19 /**
20  * Gli argomenti vengono passati nella forma: matrixSize
21  * @param matrixSize la dimensione della matrice MxM
22  */
23 int main(int argc, char** argv) {
24     int matrixSize, processId, numberOfProcessors;
25
26     MPI_Init(&argc, &argv);
27     MPI_Comm_rank(MPI_COMM_WORLD, &processId);
28     MPI_Comm_size(MPI_COMM_WORLD, &numberOfProcessors);
29
30     /* I tempi dei singoli processori, ottenuti tramite differenza,
       mentre il tempo totale come il
31      * massimo valore tra i processori */
32     double startTime, endTime, processorTime, totalTime = 0.0;
33
34     // ****************************************
35     // ****** Verifica input utente
36     // ****************************************
37
38     // Se il numero di processori non è una radice quadrata di
       interi allora non è rappresentabile in una griglia pxp
39     if(!isPerfectSquare(numberOfProcessors)) {
40         if(processId == MASTER_ID) {
41             fprintf(stderr, "ERROR - Number of processors provided
       cannot be distribuited in a NxN grid.\n"
42                             "number of processors: %d\n\n",
       numberOfProcessors);
43         }
44         MPI_Finalize();
45         return 1;
```

```
46      }
47
48      // Verifico la correttezza della dimensione della matrice
        passata in input
49      if(!parseInt(argv[1], &matrixSize)) {
50          fprintf(stderr, "ERROR - Cannot parse the matrixSize with
        value provided.\n"
51                          "matrixSize:%s\n\n", argv[1]);
52          MPI_Finalize();
53          return 1;
54      }
55
56      // MatrixSize dev'essere uguale a numberOfProcessors o un suo
        multiplo
57      if(matrixSize < numberOfProcessors || matrixSize %
        numberOfProcessors != 0) {
58          fprintf(stderr, "ERROR - Matrix size must be >= of the grid
        and a multiple of number of processors.\n"
59                          "matrixSize:%d\nnumberOfProcessors:%d\n",
        matrixSize, numberOfProcessors);
60          MPI_Finalize();
61          return 1;
62      }
63
64
65      // ****************************************
66      // ****** Creazione della griglia di processori
67      // ****************************************
68
69      int gridSize;
70
71      MPI_Bcast(&gridSize, 1, MPI_DOUBLE, MASTER_ID, MPI_COMM_WORLD);
72
73      gridSize = (int) sqrt(numberOfProcessors);
74      int numberOfValues = matrixSize/gridSize;
75
76      // Siccome è una matrice quadrata le due dimensioni sono uguali
77      int* sizesOfGrid = calloc(2, sizeof(int));
78      sizesOfGrid[0] = sizesOfGrid[1] = gridSize;
```

```
79
80      int* gridPeriod = calloc(2, sizeof(int));
81      gridPeriod[0] = gridPeriod[1] = 0;
82
83      MPI_Comm mpi_comm_grid;
84      MPI_Cart_create(MPI_COMM_WORLD, 2, sizesOfGrid, gridPeriod, 0, &
        mpi_comm_grid);
85      MPI_Comm_rank(mpi_comm_grid, &processId);
86
87      free(sizesOfGrid);
88      free(gridPeriod);
89
90      // ****************************************
91      // ****** Calcolo prodotto matrice x matrice
92      // ****************************************
93
94      // Il processore con id MASTER_ID invia le matrici generate
        casualmente
95      if(processId == MASTER_ID) {
96          distributeMatrixes(matrixSize, gridSize, mpi_comm_grid);
97      }
98
99      // Effettuo il prodotto parziale della matrice
100     double** partialResult = getPartialMatrixResult(numberOfValues,
        gridSize, &startTime, mpi_comm_grid);
101
102     // Effettua la stampa del risultato finale
103     printResult(matrixSize, gridSize, partialResult, mpi_comm_grid);
104
105     // Salvo l'istante di tempo finale di esecuzione dalla somma
        parziale, faccio stampare al MASTER_ID
106     endTime = MPI_Wtime();
107     processorTime = endTime - startTime;
108
109     // Passa al masterId il tempo maggiore impiegato
110     MPI_Reduce(&processorTime, &totalTime, 1, MPI_DOUBLE, MPI_MAX,
        MASTER_ID, MPI_COMM_WORLD);
111
112     // Il MASTER_ID fa la stampa del tempo impiegato
```

```
113     if(processId == MASTER_ID) {
114         printf("\nTime elapsed: %e seconds\n", totalTime);
115     }
116 
117     // Libero la memoria dinamica utilizzata
118     int i;
119     for(i = 0; i < numberOfValues; ++i) {
120         free(partialResult[i]);
121     }
122     free(partialResult);
123 
124     MPI_Finalize();
125     return 0;
126 }
127 
128 /**
129  * Distribuisce le matrici generate casualmente tra tutti i
       processori, compreso il chiamante
130  * @param matrixSize la dimensione della matrice da generare
       casualmente
131  * @param gridSize la dimensione della matrice di processori
132  * @param mpi_comm il Communicator, deve essere una griglia di
       dimensione gridSize
133  */
134 void distributeMatrixes(int matrixSize, int gridSize, MPI_Comm
       mpi_comm) {
135     int i, j, k, l;
136     int numberOfValues = matrixSize/gridSize;
137 
138     // Inizializzazione del seed dei valori casuali
139     srand(time(NULL));
140 
141     // Generazione e stampa delle due matrici casuali
142     printf("First random matrix:\n");
143     double** firstMatrix = getMatrixOfRandomNumbersOfSize(matrixSize
       , matrixSize, MIN_RANDOM_NUMBER, MAX_RANDOM_NUMBER);
144     printSquareMatrix(firstMatrix, matrixSize);
145 
146     printf("\nSecond random matrix:\n");
```

```
147     double** secondMatrix = getMatrixOfRandomNumbersOfSize(
    matrixSize, matrixSize, MIN_RANDOM_NUMBER, MAX_RANDOM_NUMBER);
148     printSquareMatrix(secondMatrix, matrixSize);
149 
150     // Distribuzione di porzioni delle due matrici firstMatrix e
    secondMatrix a tutti i processori
151     for(i = 0; i < gridSize; ++i) {
152         for(j = 0; j < gridSize; ++j) {
153             int receiverCoordinate[2] = { i, j };
154             int receiverProcessor;
155             MPI_Cart_rank(mpi_comm, receiverCoordinate, &
    receiverProcessor);
156 
157             for(k = (i * numberOfValues); k < (i * numberOfValues) +
     numberOfValues; ++k) {
158                 for(l = (j * numberOfValues); l < (j *
    numberOfValues) + numberOfValues; ++l) {
159                     // Per l'invio vengono utilizzati due tag
    differenti per identificare i valori
160                     MPI_Send(&firstMatrix[k][l], 1, MPI_DOUBLE,
    receiverProcessor, TAG_MATRIX_FIRST, mpi_comm);
161                     MPI_Send(&secondMatrix[k][l], 1, MPI_DOUBLE,
    receiverProcessor, TAG_MATRIX_SECOND, mpi_comm);
162                 }
163             }
164         }
165     }
166 
167     free(firstMatrix);
168     free(secondMatrix);
169 }
170 
171 /**
172  * Riceve una matrice inviata con un determinato tag
173  * @param matrixSize il numero M di righe e colonne della matrice
    MxM
174  * @param tag il tag da cui ricevere
175  * @param mpi_comm il Communicator
176  * @return la matrice ricevuta
```

```
177  */
178 double** receiveMatrix(int matrixSize, int tag, MPI_Comm mpi_comm) {
179     int i, j;
180     MPI_Status status;
181 
182     double** matrix = malloc(sizeof(double*) * matrixSize);
183     for(i = 0; i < matrixSize; ++i) {
184         matrix[i] = malloc(sizeof(double*) * matrixSize);
185 
186         for(j = 0; j < matrixSize; ++j) {
187             MPI_Recv(&matrix[i][j], 1, MPI_DOUBLE, MASTER_ID, tag,
    mpi_comm, &status);
188         }
189     }
190 
191     return matrix;
192 }
193 
194 /**
195  * Effettua un prodotto parziale matrice x matrice
196  * @param numberOfValues la dimensione parziale della matrice
197  * @param gridSize la dimensione della griglia dei processori
198  * @param startTime il valore in output dell'istante di inizio dei
     calcoli
199  * @param mpi_comm_grid il Communicator, deve essere una griglia di
     dimensione gridSize
200  * @return il prodotto parziale, una matrice di dimensione
     numberOfValues x numberOfValues
201  */
202 double** getPartialMatrixResult(int numberOfValues, int gridSize,
    double* startTime, MPI_Comm mpi_comm_grid) {
203     int i, j, k;
204     MPI_Status status;
205 
206     int processId;
207     MPI_Comm_rank(mpi_comm_grid, &processId);
208 
209     int* coordinates = calloc(2, sizeof(int));
210     MPI_Cart_coords(mpi_comm_grid, processId, 2, coordinates);
```

```
211
212     // Tutti i processori ricevono una porzione di matrice
213     double** matrixOne = receiveMatrix(numberOfValues,
    TAG_MATRIX_FIRST, mpi_comm_grid);
214     double** matrixTwo = receiveMatrix(numberOfValues,
    TAG_MATRIX_SECOND, mpi_comm_grid);
215
216     // Alloco la matrice risultato parziale
217     double** result = calloc(numberOfValues, sizeof(double*));
218     for(i = 0; i < numberOfValues; ++i) {
219         result[i] = calloc(numberOfValues, sizeof(double*));
220     }
221
222     // Mi sincronizzo con tutti i processori che hanno ricevuto le
    loro porzioni e salvo l'istante di tempo iniziale
223     MPI_Barrier(MPI_COMM_WORLD);
224     *startTime = MPI_Wtime();
225
226     // Se è sulla diagonale invia alle varie righe i suoi valori,
    altrimenti riceve dalla diagonale
227     int iteration;
228     for(iteration = 0; iteration < gridSize; ++iteration) {
229
230         double** receivedMatrix = NULL;
231         if(coordinates[0] == mod(coordinates[1] - iteration,
    gridSize)) {
232
233             int rowProcessor;
234             for(rowProcessor = 0; rowProcessor < gridSize; ++
    rowProcessor) {
235                 int receiverCoordinate[2] = { coordinates[0],
    rowProcessor };
236                 int receiverId;
237                 MPI_Cart_rank(mpi_comm_grid, receiverCoordinate, &
    receiverId);
238
239                 // Non invia a se stesso
240                 if(processId != receiverId) {
```

```
241                     //printf("Sono %d, devo inviare a %d che SULLA
    GRIGLIA sta a (%d, %d)\n\n", processId, receiverId, coordinates
    [0], rowProcessor);
242                     for(i = 0; i < numberOfValues; ++i) {
243                         for(j = 0; j < numberOfValues; ++j) {
244                             MPI_Send(&matrixOne[i][j], 1, MPI_DOUBLE
    , receiverId, coordinates[0], mpi_comm_grid);
245                         }
246                     }
247                 }
248             }
249         } else {
250 
251             // Calcolo da chi devo ricevere
252             int diagonalCoordinates[2] = {coordinates[0], mod(
    coordinates[0] + iteration, gridSize)};
253             int diagonalProcessor;
254             MPI_Cart_rank(mpi_comm_grid, diagonalCoordinates, &
    diagonalProcessor);
255 
256             receivedMatrix = malloc(sizeof(double*) * numberOfValues
    );
257 
258             for(i = 0; i < numberOfValues; ++i) {
259                 receivedMatrix[i] = malloc(sizeof(double*) *
    numberOfValues);
260 
261                 for(j = 0; j < numberOfValues; ++j) {
262                     MPI_Recv(&receivedMatrix[i][j], 1, MPI_DOUBLE,
    diagonalProcessor, coordinates[0], mpi_comm_grid, &status);
263                 }
264             }
265         }
266 
267         // Non è necessario eseguirlo la prima volta
268         if(iteration > 0) {
269             // Invio la mia matrice B al processore (i - 1, j),
    nella riga precedente
270             int sendTo;
```

```
271             int sendCoordinate[2] = { mod(coordinates[0] - 1,
    gridSize), coordinates[1] };
272             MPI_Cart_rank(mpi_comm_grid, sendCoordinate, &sendTo);
273 
274             for(i = 0; i < numberOfValues; ++i) {
275                 for(j = 0; j < numberOfValues; ++j) {
276                     MPI_Send(&matrixTwo[i][j], 1, MPI_DOUBLE, sendTo
    , coordinates[1], mpi_comm_grid);
277                 }
278             }
279 
280             // Ricevo la matrice B dal processore (i + 1, j), nella
    riga successiva
281             int receiveFrom;
282             int receiveCoordinate[2] = { mod(coordinates[0] + 1,
    gridSize), coordinates[1] };
283             MPI_Cart_rank(mpi_comm_grid, receiveCoordinate, &
    receiveFrom);
284 
285             for(i = 0; i < numberOfValues; ++i) {
286                 for(j = 0; j < numberOfValues; ++j) {
287                     MPI_Recv(&matrixTwo[i][j], 1, MPI_DOUBLE,
    receiveFrom, coordinates[1], mpi_comm_grid, &status);
288                 }
289             }
290         }
291 
292         // Creo un puntatore alla matrice che voglio usare, per i
    processori diagonali è MatrixOne altrimenti quella ricevuta
293         double** matrixToUse = (receivedMatrix == NULL) ? matrixOne
    : receivedMatrix;
294 
295         // Calcolo del prodotto parziale
296         for(i = 0; i < numberOfValues; ++i) {
297             for(j = 0; j < numberOfValues; ++j) {
298                 for(k = 0; k < numberOfValues; ++k) {
299                     result[i][j] += (matrixToUse[i][k] * matrixTwo[k
    ][j]);
300                 }
```

```
301             }
302         }
303 
304         // Pulisco le matrici che allo step successivo non servono
    più
305         if(receivedMatrix != NULL) {
306             for(i = 0; i < numberOfValues; ++i) {
307                 free(receivedMatrix[i]);
308             }
309             free(receivedMatrix);
310         }
311     }
312 
313     for(i = 0; i < numberOfValues; ++i) {
314         free(matrixOne[i]);
315         free(matrixTwo[i]);
316     }
317     free(matrixOne);
318     free(matrixTwo);
319     free(coordinates);
320 
321     return result;
322 }
323 
324 /**
325  * Stampa il risultato del prodotto matrice per matrice
326  * @param matrixSize la dimensione originale della matrice
327  * @param gridSize la dimensione della griglia di processori
328  * @param partialResult il risultato parziale del chiamante
329  * @param mpi_comm il Communicator, deve essere una griglia di
    dimensione gridSize
330  */
331 void printResult(int matrixSize, int gridSize, double**
    partialResult, MPI_Comm mpi_comm_grid) {
332     int i, j, k, l;
333     MPI_Status status;
334     int numberOfValues = matrixSize/gridSize;
335     int processId;
336     MPI_Comm_rank(mpi_comm_grid, &processId);
```

```
337
338     // Tutti i processori inviano al MASTER_ID il proprio prodotto
    parziale
339     for(i = 0; i < numberOfValues; i++) {
340         for(j = 0; j < numberOfValues; j++) {
341             MPI_Send(&partialResult[i][j], 1, MPI_DOUBLE, MASTER_ID,
     MASTER_ID, mpi_comm_grid);
342         }
343     }
344
345     // Il MASTER_ID riceve tutta la matrice e la stampa ordinata
346     if(processId == MASTER_ID) {
347
348         double** resultMatrix = malloc(sizeof(double*) * matrixSize)
    ;
349         for (i = 0; i < matrixSize; ++i) {
350             resultMatrix[i] = malloc(sizeof(double*) * matrixSize);
351         }
352
353         for (i = 0; i < gridSize; ++i) {
354             for (j = 0; j < gridSize; ++j) {
355                 int receiverCoordinate[2] = {i, j};
356                 int receiverProcessor;
357                 MPI_Cart_rank(mpi_comm_grid, receiverCoordinate, &
    receiverProcessor);
358
359                 for (k = (i * numberOfValues); k < (i *
    numberOfValues) + numberOfValues; ++k) {
360                     for (l = (j * numberOfValues); l < (j *
    numberOfValues) + numberOfValues; ++l) {
361                         MPI_Recv(&resultMatrix[k][l], 1, MPI_DOUBLE,
     receiverProcessor, MASTER_ID, mpi_comm_grid, &status);
362                     }
363                 }
364             }
365         }
366
367         printf("\nRESULT: \n");
368         printSquareMatrix(resultMatrix, matrixSize);
```

```
369
370         for (i = 0; i < matrixSize; ++i) {
371             free(resultMatrix[i]);
372         }
373         free(resultMatrix);
374     }
375 }
```

A.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 PDC_4_JANUARY2_UTILS_H
9  #define PDC_4_JANUARY2_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 //PDC_4_JANUARY2_UTILS_H
```

A.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  }
```

A.4 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/4-january
/usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS_O_WORKDIR/matrix
    $PBS_O_WORKDIR/matrix.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/mpiexec -machinefile hostlist --np
    $NCPU $PBS_O_WORKDIR/matrix $MATRIX_SIZE
```