UNIVERSITÀ DEGLI STUDI DI NAPOLI
FEDERICO II

PARALLEL AND DISTRIBUITED COMPUTING

Corso di Laurea in Informatica

Progetto 3

PRODOTTO MATRICE PER MATRICE IN AMBIENTE MIMD-DM

Studente:

Matteo Richard Gaudino

N86003226

ANNO ACCADEMICO 2021/2022

Indice

Capitolo 1

Presentazione e Analisi del Problema

Sommario

1.1 Il Problema

L'obiettivo è sviluppare un algoritmo per il prodotto di Matrici su un calcolatore MIMD a memoria condivisa utilizzando C e openMPI per lo sviluppo del programma. Le matrici devono essere quadrate di dimensione $m \times m$ e distribuite su una griglia di processori di dimensione $p \times p$ tale che $p|m$.

1.2 Prodotto Matrice per Matrice

Per prodotto Matrice-Matrice si intende il prodotto righe per colonne tra due matrici quadrate $A, B \in \mathcal{M}_{m,m}$:

$$\cdot : A \in \mathcal{M}_{m,m} \times B \in \mathcal{M}_{m,m} \longrightarrow C \in \mathcal{M}_{m,m}$$

Il prodotto righe per colonne si effettua utilizzando il prodotto scalare tra le righe di A e le colonne di B:

$$C_{i,j} = A_i \bullet B^j$$

$$C_{i,j} = \sum_{k=1}^{m} A_{i,k} \cdot B_k^j$$

1.3 Generazione della Matrice

La matrice è allocata dinamicamente come singolo puntatore. La notazione vettoriale rende più facile la comunicazione dell'intera matrice tra i vari processori. La funzione randomMatrix alloca una matrice $m \times n$ e ne restituisce il puntatore. I vaori della matrice sono settati randomicamente con valore massimo 1000.

```
float* randomMatrix(int m, int n){
   float* M = malloc(m*n * sizeof(float));
   for(int i = 0; i < m*n; i++) {
       M[i] = ((float) rand())/((float) RAND_MAX/1000.0);
   }
   return M;
}
```

1.4 Creazione della griglia di processori

La funzione crea_griglia crea una griglia di processori periodica di dimensione $dim \times dim$. Inoltre crea un comunicatore per processori sulla stessa riga tramite MPI_Cart_create per effettuare un broadcast efficiente.

```
void crea_griglia(int dim, MPI_Comm *comm_grid, MPI_Comm *comm_row){

   int dims[] = {dim, dim}; // Dimensione della griglia
   int period[] = {1, 1}; // La griglia deve essere periodica per ogni lato

   MPI_Cart_create(MPI_COMM_WORLD, 2, dims, period, 0, comm_grid); // Crrea una griglia di due

   int remain[] = {0, 1};
   MPI_Cart_sub(*comm_grid, remain, comm_row); // Crea un comunicatore per broadcast su righe s
}
```

1.5 Comunicazione dell'input e dell'output

Ogni processore riceverà una sottomatrice quadrata delle matrici di input A e B di dimensione $\frac{m}{p}$ dove p è la radice quadrata del numero di processori.

```
int colOffset, rowOffset;
      for(int i = 1; i < worldSize; i++){
         rowOffset = (mSub) * (i/p); // Riga da cui partire
         colOffset = (mSub) * (i%p); // Colonna da cui partire
         for(int j = 0; j < mSub; j++){
            MPI_Send(A + (rowOffset * m) + colOffset, mSub, MPI_FLOAT, i, 0, MPI_COMM_WORLD);
            MPI_Send(B + (rowOffset * m) + colOffset, mSub, MPI_FLOAT, i, 0, MPI_COMM_WORLD);
            rowOffset++; // Prossima riga
         }
      }
```

Il processore 0 spedice per righe le sottomatrici di A e B.

```
for(int j = 0; j < mSub; j++){
         MPI_Recv(subA + (mSub*j), mSub, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, &status);
         MPI_Recv(subB + (mSub*j), mSub, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, &status);
      }
```

La comunicazione dell'output avviene nello stesso identico modo ma con le operazioni di Recv e Send invertite e la matrice C al posto di A e B.

1.6 Strategia di Parallelizzazione

La strategia utilizzata per la parallelizzazione è la BMR (Broadcast-Multiply-Rolling). La strategia è divisa in 3 fasi da ripetere per p volte.

Il processori sulla diagonale principale inviano la propra porzione di A ai processori situati sulla stessa riga tramite il comunicatore *comm_row*. Tutti i processori effettuano il prodotto matrice-matrice tra la porzione appena ricevuta e la propia porzione di B.

```
int bSenderCoords[2] = {gridCoords[0], gridCoords[0]}; // processore che
↪  effettua il broadcast sulla riga. Inizialmente sulla diagonale principale
   int bSenderRank; // Rank nel comunicator comm_grid
   MPI_Cart_rank(comm_row, bSenderCoords, &bSenderRank);


    if(gridCoords[0] == gridCoords[1]){ // Se si trova sulla diagonale si
    ↪  prepara ad inviare
      memcpy(Temp, subA, subMatrixSize * sizeof(float));
    }
```

```
MPI_Bcast(Temp, subMatrixSize, MPI_FLOAT, bSenderRank, comm_row);

t += matrXmatr(Temp, mSub, mSub, subB, mSub, subC);
```

Nei seguenti $p-1$ passi i processori che dovranno effettuare il broadcast si trovano a destra (nella riga della griglia) di quelli che hanno effettuato il broadcast nel passo precedente. In seguito la sottomatrice di B viene inviata dal processore sulla riga superiore e stessa colonna e viene sovrascritta dalla sotto matrice di B inviata dal processore sulla riga inferiore e stessa colonna.

Dopo la comunicazione viene effettuato il prodotto tra le 2 sottomatrici ricevute.

```
for(int i = 1; i < p; i++){
      bSenderCoords[1] += 1; // prossima diagonale
      MPI_Cart_rank(comm_row, bSenderCoords, &bSenderRank);

      if(gridCoords[1] == (bSenderCoords[1])%p){ // Se deve inviare
         memcpy(Temp, subA, mSub*mSub*sizeof(float));
      }

      MPI_Bcast(Temp, subMatrixSize, MPI_FLOAT, bSenderRank, comm_row); //
      ↪ Broadcast sulla stessa riga

        MPI_Isend(subB, subMatrixSize, MPI_FLOAT, upperRowRank, 0, comm_grid,
        ↪ &rqst); // Spedizione alla riga superiore

        MPI_Recv(subB, subMatrixSize, MPI_FLOAT, lowerRowRank, 0, comm_grid,
        ↪ &status); // Ricezione dalla riga inferiore


        t += matrXmatr(Temp, mSub, mSub, subB, mSub, subC); // prodotto
    }
```

Capitolo 2

Esecuzione e Testing

Sommario

2.1 Compilazione

Per compilare il programma eseguire il comando

mpicc -std=c99 matriceXmatrice.c -o matrXmatr -lm

I flag -lm serve al compilatore per linkare la libreria math.h. Lo standard c99 o superiore è necessario per l'utilizzo di dichiarazioni di variabili nel for.

2.2 Esecuzione

Per eseguire il comando lanciare

mpiexec -machinefile $< hostlist >$ **-np <nProc> matrXmatr** $< m >$

Il programma creerà due matrici casuali di $m \times m$ e effettuerà il prodotto righe per colonne distribuito su $nProc$ processori. Terminato il calcolo il processore 0 stamperà sulla stessa riga prima il tempo impiegato per la computazione senza overhead e dopo il tempo comprensivo delle istruzioni di comunicazione.
Per la corretta realizzazione della griglia di processori $nProc$ deve essere un quadrato perfetto e la radice di $nProc$ deve dividere m.

2.3 Testing

Il testing è stato effettuato sui calcolatori dell'infrastruttura SCoPE della Federico II. La lista dei componenti è reperibile dal sito ufficiale del datacenter. Il programma è stato testato eseguendo il seguente script scritto in bash.

```
for p in 1 4
do
   head -$p hostlist > host$p
    echo "-------------- test $p processor --------------"
    for m in 10 100 300 500 800 1000
    do
        echo Test: $m
        for i in {1..5}
        do
            res=$(/usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile host$p
            ↪  -np $p $PBS_O_WORKDIR/matrXmatr $m)
            echo $res >> $PBS_O_WORKDIR/results/p$p_$m.txt
            echo $res
        done
    done
done
```

Lo script lancia il programma per 5 volte con input matrici quadrate di dimensione 10, 100, 300, 500, 800 e 1000 con numero di processori di 1 e 4. L'output è poi salvato nella cartella results in file con il seguente formato:

$$p < p > \_ < m > .txt$$

Avendo a disposizione solo 8 calcolatori non sarebbe stato utile testare anche l'esecuzione con 9 processori.

Capitolo 3

Analisi delle prestazioni

Sommario

3.1 Algoritmo non parallelo

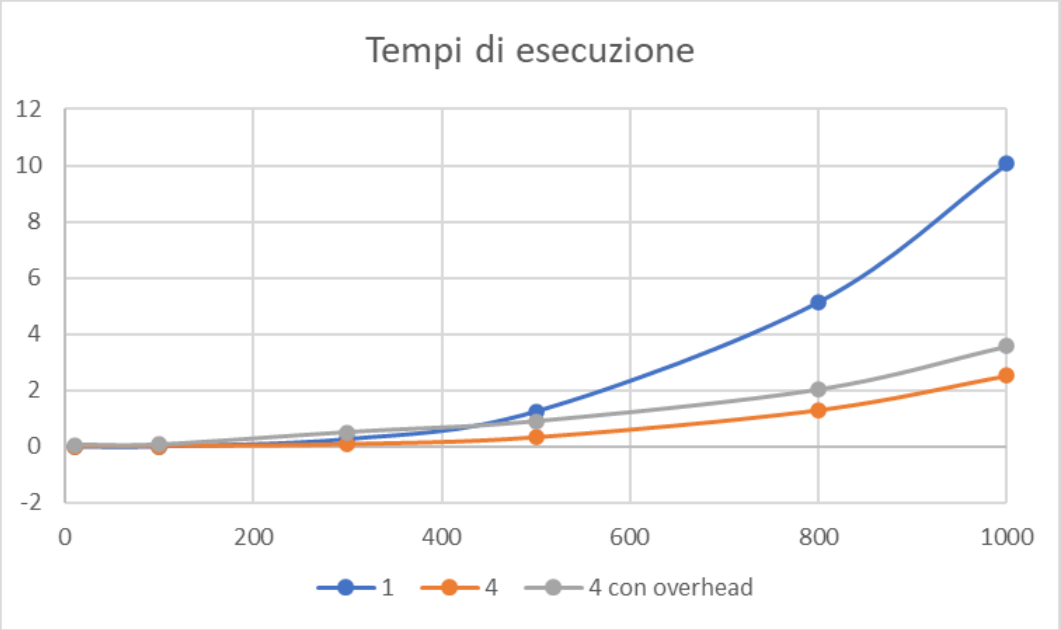
L'algoritmo non parallelo è stato testato settando $nProc = 1$. Settando il numero di processori a 1 MPI ignora le istruzioni di comunicazione che quindi non influiscono negativamente sul tempo di esecuzione finale.

3.2 Tempi di Esecuzione

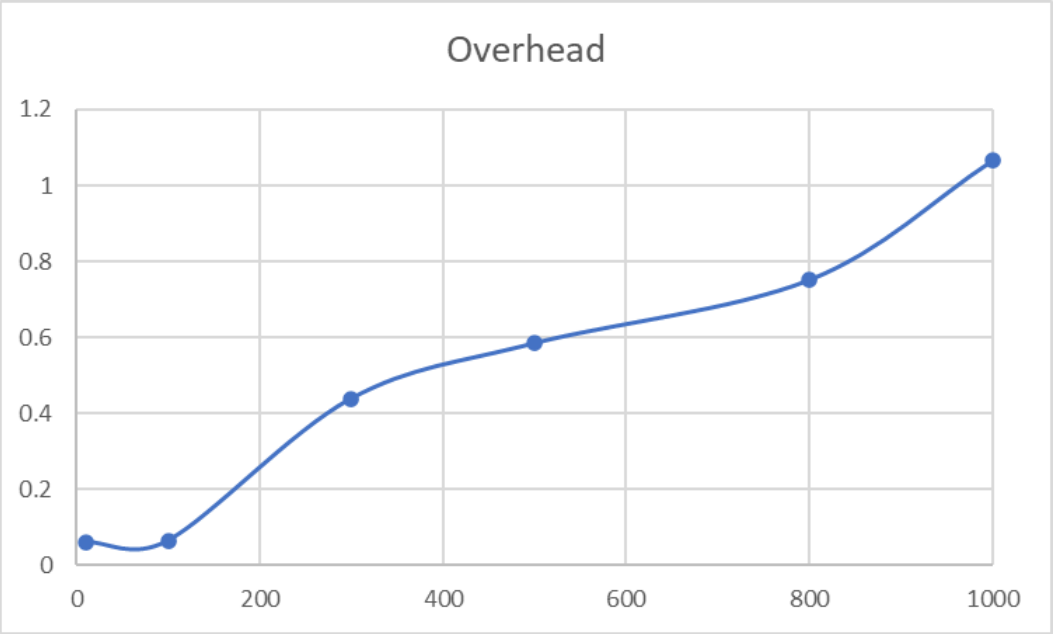
Nella seguente tabella sono riportati i tempi relativi all'esecuzione del test del capitolo precedente. Sulla prima colonna è riportata la dimensione dell'input. Sulla seconda i tempi di esecuzione con un processore e sulla terza e la quarta i tempi su 4 processori con e senza overhead. L'overhead dell'esecuzione con un processore non è riportato perchè i valori coincidono con uno scarto di pochi millisecondi.

input	p1	p4	p4 con overhead
10	0.000013	0.000005	0.061282
100	0.011499	0.002555	0.067211
300	0.272797	0.068333	0.507618
500	1.256976	0.314933	0.900388
800	5.145574	1.286253	2.038007
1000	10.055237	2.513156	3.579447

3.3 Considerazioni



Come da aspettative l'esecuzione con 4 processori richiede molto meno tempo di quella a singolo processore. L'overhead di comunicazione non influisce molto sul tempo di esecuzione.



Con l'aumentare delle dimensioni delle matrici aumentano anche i tempi necessari alla comunicazione e quindi l'overhead.

3.4 Speedup ed Efficienza

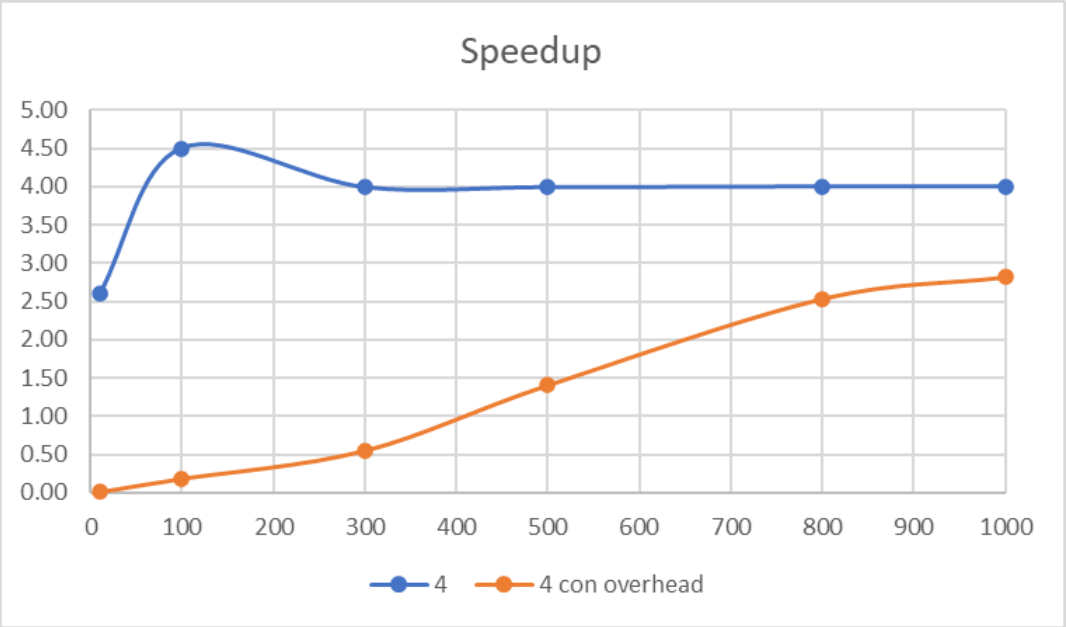
Siano Speedup ed Efficienza definiti come segue

$$S(p) = \frac{T(1)}{T(p)} \qquad E(p) = \frac{S(p)}{p}$$

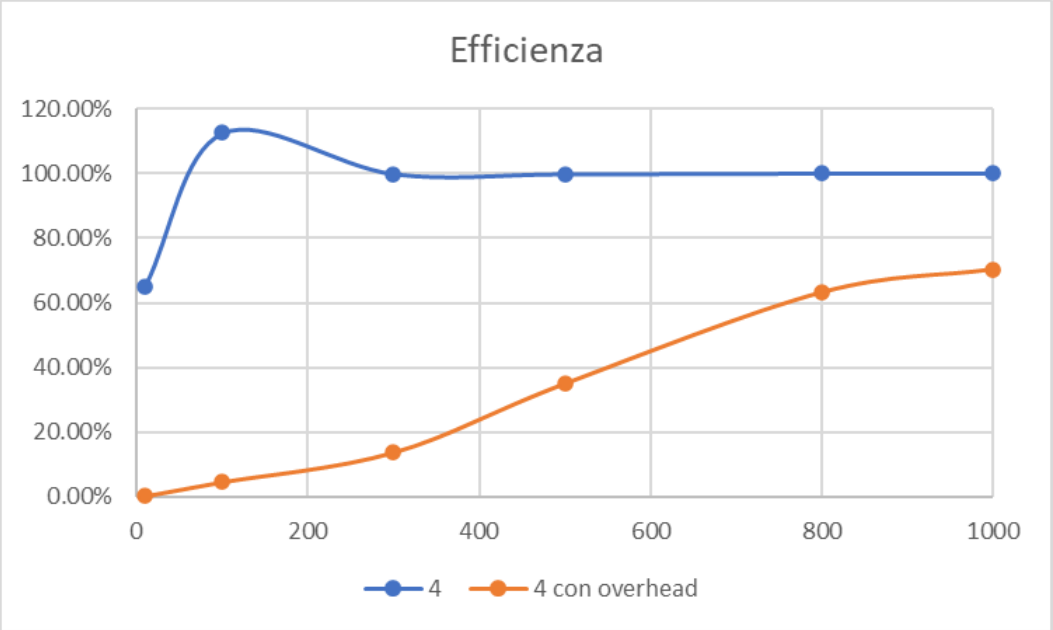
Applicando le equazioni ai dati riportati nelle tabelle si ottengono gli Speedup e le Efficienze delle varie esecuzioni. Sulle ultime due righe sono riportati speedup ed efficienza considerando anche l'overhead

input	10	100	300	500	800	1000
S(4)	2.60	4.50	3.99	3.99	4.00	4.00
E(4)	65.00%	112.51%	99.80%	99.78%	100.01%	100.03%
$S_h(4)$	0.01	0.18	0.55	1.40	2.53	2.82
$E_h(4)$	0.19%	4.55%	13.71%	35.08%	63.32%	70.40%

Di seguito i grafici con i dati delle tabelle:



Non considerando l'overhead lo speedup raggiunge il valore ideale. La seconda misurazione supera il valore massimo teorico, probabilmente per questioni relative al sistema operativo, e quindi non dovrebbe essere considerato.



Anche l'efficienza raggiunge il 100% se non si considera l'overhead. Con overhead, efficienza e speedup indicano che c'è un aumento delle prestazioni solo con matrici di dimensioni superiori a 300×300

Appendice A

Codice completo con Documentazione interna

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

float* randomMatrix(int m, int n);
double matrXmatr(const float *A, int m, int n, const float *B, int k, float
    *C);
void crea_griglia(int dim, MPI_Comm *comm_grid, MPI_Comm *comm_row);

// ----------------------------------------

int main(int argc, char *argv[]){
    int worldRank, worldSize; // rank all'interno del COMM_WORLD e dimensione
        del comunicatore
    int gridRank, gridCoords[2]; // rank e coordinate all'interno del
        comunicatore griglia

    int m; // Righe e Colonne delle matrici
    int mSub; // Righe e colonne delle sottomatrici
    int matrixSize; // m*m
```

```
21      int subMatrixSize; // m*m/worldSize
22
23      float *A, *B; // Matrici di input
24      float *C; // Matrice risultato
25      float *subA, *subB, *subC; // Sottomatrici locali
26      float *Temp; // Matrice di appoggio. Necessaria per non sovrascrivere subA
        ↪  durante la comunicazione in BMR
27
28      double t = 0.F, start, end; // Tempo per impiegato per il calcolo e tempo
        ↪  totale con overhead
29
30      MPI_Status status; // per send
31      MPI_Request rqst; // per isend
32      MPI_Comm comm_grid, comm_row; // Comunicatore di griglia e di riga
33
34
35
36      MPI_Init(&argc, &argv);
37      MPI_Comm_rank(MPI_COMM_WORLD, &worldRank);
38      MPI_Comm_size(MPI_COMM_WORLD, &worldSize);
39
40
41  //------------------------ Controlli sull'input
    ↪  ---------------------------------
42      int p = (int) sqrt(worldSize); // Lato della griglia quadrata di processori
43      // Il numero di processori deve essere un quadrato perfetto per la BMR
44      if (p*p != worldSize){
45          perror("ERROR Processors must be perfect square.\n");
46          MPI_Abort(MPI_COMM_WORLD, 1);
47          return 1;
48      }
49
50      if(argc <= 1){
51          perror("ERROR Argument required. usage: ./program <matrixDim>\n");
52          MPI_Abort(MPI_COMM_WORLD, 1);
53          return 1;
54      }
```

```
55
56      if(worldRank==0){
57          m = atoi(argv[1]);  // Righe e colonne delle matrici
58          // Il numero di righe/colonne delle matrici deve essere multiplo di p
59          if(m % p != 0){
60              perror("ERROR rows/cols of the matrices must be multiple of sqrt
                ↪  nProc.\n");
61              MPI_Abort(MPI_COMM_WORLD, 1);
62              return 1;
63          }
64      }
65  //-----------------------Generazione Matrici ----------------------------
66
67       if (worldRank == 0){
68          srand(time(NULL));
69          // Generazione delle matrici random
70          A = randomMatrix(m, m);
71          B = randomMatrix(m, m);
72          // Alloca spazio per la matrice risultato
73          C = malloc(m*m*sizeof(float));
74      }
75
76  // ---------------------- Comunicazione input ---------------------------
77
78      start = MPI_Wtime();
79
80      MPI_Bcast(&m, 1, MPI_INT, 0, MPI_COMM_WORLD);
81      MPI_Bcast(&p, 1, MPI_INT, 0, MPI_COMM_WORLD);
82
83       matrixSize = m*m;
84       subMatrixSize = matrixSize/worldSize;
85       mSub = m/p;
86
87
88       crea_griglia(p, &comm_grid, &comm_row); // Crea la griglia di processori
89
90      MPI_Comm_rank(comm_grid, &gridRank);
```

```
91      MPI_Cart_coords(comm_grid, gridRank, 2, gridCoords);
92
93
94      subA = malloc(subMatrixSize * sizeof(float)); // Sottomatrice di A
95      Temp = malloc(subMatrixSize * sizeof(float)); // Blocco di appoggio
96      subB = malloc(subMatrixSize * sizeof(float)); // Sottomatrice di B
97      subC = malloc(subMatrixSize * sizeof(float)); // Sottomatrice di C
98      memset(subC, 0, subMatrixSize); // Riempie C di zeri
99
100     if(worldRank==0){
101        int colOffset, rowOffset;
102        for(int i = 1; i < worldSize; i++){
103           rowOffset = (mSub) * (i/p); // Riga da cui partire
104           colOffset = (mSub) * (i%p); // Colonna da cui partire
105           for(int j = 0; j < mSub; j++){
106              MPI_Send(A + (rowOffset * m) + colOffset, mSub, MPI_FLOAT, i, 0,
                  ↪ MPI_COMM_WORLD);
107              MPI_Send(B + (rowOffset * m) + colOffset, mSub, MPI_FLOAT, i, 0,
                  ↪ MPI_COMM_WORLD);
108              rowOffset++; // Prossima riga
109           }
110        }
111
112        // Copia di subA
113        for(int j = 0; j < mSub; j++){
114           for(int z = 0; z < mSub; z++)
115              *(subA + (mSub*j) + z) = *(A + (m*j) + z);
116        }
117        // Copia di subB
118        for(int j = 0; j < mSub; j++){
119           for(int z = 0; z < mSub; z++)
120              *(subB + (mSub*j) + z) = *(B + (m*j) + z);
121        }
122     } else{
123        // Ricezione delle proprie sottomatrici
124        for(int j = 0; j < mSub; j++){
```

```
125             MPI_Recv(subA + (mSub*j), mSub, MPI_FLOAT, 0, 0, MPI_COMM_WORLD,
                ↪  &status);
126             MPI_Recv(subB + (mSub*j), mSub, MPI_FLOAT, 0, 0, MPI_COMM_WORLD,
                ↪  &status);
127         }
128     }
129 // ------------------------------ CALCOLO con BMR
    ↪  ----------------------------------------
130
131     // SubB va inviato al processore sulla riga superiore e ricevuto dalla riga
        ↪  inferiore
132     int upperRowRank, lowerRowRank;
133     { // coords non è necessaro al di fuori di questo blocco
134         int coords[2];
135         coords[0] = (gridCoords[0] - 1); // Riga superiore
136         coords[1] = gridCoords[1]; // Stessa colonna
137         MPI_Cart_rank(comm_grid, coords, &upperRowRank);
138
139         // SubB viene ricevuto dal processore sulla riga inferiore
140         coords[0] = (gridCoords[0] + 1); // Riga inferiore, stessa colonna
141         MPI_Cart_rank(comm_grid, coords, &lowerRowRank);
142     }
143
144     int bSenderCoords[2] = {gridCoords[0], gridCoords[0]}; // processore che
        ↪  effettua il broadcast sulla riga. Inizialmente sulla diagonale
        ↪  principale
145     int bSenderRank; // Rank nel comunicator comm_grid
146     MPI_Cart_rank(comm_row, bSenderCoords, &bSenderRank);
147
148
149      if(gridCoords[0] == gridCoords[1]){ // Se si trova sulla diagonale si
         ↪  prepara ad inviare
150        memcpy(Temp, subA, subMatrixSize * sizeof(float));
151      }
152
153      MPI_Bcast(Temp, subMatrixSize, MPI_FLOAT, bSenderRank, comm_row);
154
```

```
155     t += matrXmatr(Temp, mSub, mSub, subB, mSub, subC);
156
157      for(int i = 1; i < p; i++){
158        bSenderCoords[1] += 1; // prossima diagonale
159        MPI_Cart_rank(comm_row, bSenderCoords, &bSenderRank);
160
161        if(gridCoords[1] == (bSenderCoords[1])%p){ // Se deve inviare
162           memcpy(Temp, subA, mSub*mSub*sizeof(float));
163        }
164
165        MPI_Bcast(Temp, subMatrixSize, MPI_FLOAT, bSenderRank, comm_row); //
           ↪ Broadcast sulla stessa riga
166
167          MPI_Isend(subB, subMatrixSize, MPI_FLOAT, upperRowRank, 0, comm_grid,
           ↪ &rqst); // Spedizione alla riga superiore
168
169          MPI_Recv(subB, subMatrixSize, MPI_FLOAT, lowerRowRank, 0, comm_grid,
           ↪ &status); // Ricezione dalla riga inferiore
170
171
172          t += matrXmatr(Temp, mSub, mSub, subB, mSub, subC); // prodotto
173      }
174
175  // ------------------------------- FINE CALCOLO
     ↪ ------------------------------------
176
177  // ------------------------------- Comunicazione del risultato
     ↪ ------------------------------------
178
179     if(worldRank==0){
180         // Copia in C
181         for (int i = 0; i < mSub; ++i) {
182             memcpy(C + (i*m), subC + (i*mSub), mSub); // Copia ogni riga di
                ↪ subC in C
183         }
184
185         // Riceve dagli altri processori le loro parti di C
```

```
186         int colOffset, rowOffset;
187         for(int i = 1; i < worldSize; i++){
188             rowOffset = (mSub) * (i/p); // Riga da cui partire
189             colOffset = (mSub) * (i%p); // Colonna da cui partire
190             for(int j = 0; j < mSub; j++){ // Ricezione per righe
191                 MPI_Recv(C + (rowOffset * m) + colOffset, mSub, MPI_FLOAT, i,
                    ↪  0, MPI_COMM_WORLD, &status);
192                 rowOffset++; // passa alla prossima riga
193             }
194         }
195 
196     } else {
197         for(int j = 0; j < mSub; j++){ // Invio per righe
198             MPI_Send(subC + (mSub*j), mSub, MPI_FLOAT, 0, 0, MPI_COMM_WORLD);
199         }
200     }
201 
202     end = MPI_Wtime();
203 
204     double tMax; // Tempo massimo per il calcolo
205 
206     // Ricava il tempo massimo per il prodotto
207     MPI_Reduce(&t, &tMax, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
208 
209     if (worldRank == 0) { // Stapa del risultato
210         for (int i = 0; i < m; ++i) {
211             for (int j = 0; j < m; ++j) {
212                 printf("%.2f\t", C[i*m + j]);
213             }
214             printf("\n");
215         }
216 
217         printf("%f\t%f\n", tMax, end-start); // Tempo con e senza overhead
218     }
219     MPI_Finalize();
220     return 0;
221 }
```

```
222
223 //
    ↪ ----------------------------------------------------------------------------
224
225 // Alloca una matrice random di dimensione mxn
226 float* randomMatrix(int m, int n){
227     float* M = malloc(m*n * sizeof(float));
228     for(int i = 0; i < m*n; i++) {
229         M[i] = ((float) rand())/((float) RAND_MAX/1000.0);
230     }
231     return M;
232 }
233
234 //crea una griglia bidimensionale periodica
235 void crea_griglia(int dim, MPI_Comm *comm_grid, MPI_Comm *comm_row){
236
237     int dims[] = {dim, dim}; // Dimensione della griglia
238     int period[] = {1, 1}; // La griglia deve essere periodica per ogni lato
239
240     MPI_Cart_create(MPI_COMM_WORLD, 2, dims, period, 0, comm_grid); // Crrea
        ↪ una griglia di due dimensioni pxp periodica
241
242     int remain[] = {0, 1};
243     MPI_Cart_sub(*comm_grid, remain, comm_row); // Crea un comunicatore per
        ↪ broadcast su righe separate
244 }
245
246
247 // Prodotto righe per colonne
248 double matrXmatr(const float *A, int m, int n, const float *B, int k, float
    ↪ *C){
249     double t_start, t_end;
250      t_start = MPI_Wtime();
251
252     for(int i = 0; i < m; i++){
253         for(int z = 0; z < n; z++){
254             for(int j = 0; j < k; j++){ // Prodotto scalare
```

```
255                 C[i*k+j] += A[i*n+z] * B[k*z+j];
256             }
257         }
258     }
259
260     t_end = MPI_Wtime();
261
262     return t_end - t_start;
263 }
```

Appendice B

File PBS con Test

```
#!/bin/bash
############################
#
#
# The PBS directives #
#
#
############################
#PBS -q studenti
#PBS -l nodes=4:ppn=8
#PBS -N progetto3
#PBS -o progetto3.out
#PBS -e progetto3.err
########################################
# -q coda su cui va eseguito il job #
# -l numero di nodi richiesti #
# -N nome job(stesso del file pbs) #
# -o, -e nome files contenente l'output #
########################################
#
#
# qualche informazione sul job #
#
#
########################################
echo -----------------------------------------------------
```

```
27 echo PBS: qsub is running on $PBS_O_HOST
28 echo PBS: originating queue is $PBS_O_QUEUE
29 echo PBS: executing queue is $PBS_QUEUE
30 echo PBS: working directory is $PBS_O_WORKDIR
31 echo PBS: execution mode is $PBS_ENVIRONMENT
32 echo PBS: job identifier is $PBS_JOBID
33 echo PBS: job name is $PBS_JOBNAME
34 echo PBS: node file is $PBS_NODEFILE
35 echo PBS: number of nodes is $NNODES
36 echo PBS: current home directory is $PBS_O_HOME
37 echo PBS: PATH = $PBS_O_PATH
38 echo ------------------------------------------------------
39 echo
40 echo Job reserved nodes:
41 cat $PBS_NODEFILE
42 echo
43 echo ------------------------------------------------------
44 echo
45 sort -u $PBS_NODEFILE > hostlist
46 cat hostlist
47 echo
48 echo ------------------------------------------------------
49 echo
50 PBS_O_WORKDIR=$PBS_O_HOME/matriceXmatrice
51
52 echo starting compilation
53 /usr/lib64/openmpi/1.4-gcc/bin/mpicc -std=c99 $PBS_O_WORKDIR/matriceXmatrice.c
   ↪  -o $PBS_O_WORKDIR/matrXmatr -lm
54
55 echo compilation terminated
56 echo ------------------------------------------------------
57
58 echo Start testing
59
60 for p in 1 4
61 do
62    head -$p hostlist > host$p
```

```
63      echo "-------------- test $p processor --------------"
64      for m in 10 100 300 500 800 1000
65      do
66          echo Test: $m
67          for i in {1..5}
68          do
69              res=$(/usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile host$p
                ↪  -np $p $PBS_O_WORKDIR/matrXmatr $m)
70              echo $res >> $PBS_O_WORKDIR/results/p$p_$m.txt
71              echo $res
72          done
73      done
74  done
```