UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

PARALLEL AND DISTRIBUITED COMPUTING

Corso di Laurea in Informatica

Progetto 2

PRODOTTO MATRICE PER VETTORE IN AMBIENTE MIMD-SM

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 Matrice-Vettore su un calcolatore MIMD a memoria condivisa utilizzando C e openMP per lo sviluppo del programma.

1.2 Prodotto Matrice per Vettore

Per prodotto Matrice-Vettore si intende il prodotto righe per colonne tra una matrice $M \in \mathcal{M}_{m,n}$ e un vettore colonna $V \in \mathcal{M}_{n,1}$:

$$\cdot : M \in \mathcal{M}_{m,n} \times V \in \mathcal{M}_{n,1} \longrightarrow R \in \mathcal{M}_{m,1}$$

La matrice risultante sarà un vettore colonna con stesse righe della matrice M. Il prodotto righe per colonne si effettua utilizzando il prodotto scalare tra le righe di M e il vettore V:

$$R_i = M_i \bullet V$$

$$R_i = \sum_{j=1}^{n} M_{i,j} \cdot V_j$$

1.3 Strategia di Parallelizzazione

La strategia consiste nel suddividere i j prodotti scalari sui vari thread.

```
void matricePerVettore(long long **Matrix, unsigned int row, unsigned int col,
↪  long long *Vector, long long *Result){
  unsigned int i, j;

  #pragma omp parallel for shared(Matrix,row,col,Vector,Result) private(i,j)
  for(i = 0; i < row; i++){
    for(j = 0; j < col; j++)
      Result[i] += Matrix[i][j] * Vector[j];
  }
}
```

La direttiva omp parallel for comunica a openMP di suddividere le varie iterazioni del for (e quindi i vari prodotti scalari) sui thread a disposizione. La matrice e i vettori sono condivisi, mentre gli indici sono privati per evitare problemi di concorrenza. Il risultati vengono salvati nel vettore Result.

1.4 Generazione della Matrice

La matrice è allocata dinamicamente come puntatore di puntatori. Successivamente vengono iseriti valori generati casualmente da 0 a MAX, dove MAX è una costante con valore 2^{16}

```
long long** randomMatrix(unsigned int row, unsigned int col){
  long long **Matrix;

  Matrix = malloc(row * (sizeof(long long*)));
  for (unsigned int i = 0; i < row; i++){
    Matrix[i] = malloc(col * sizeof(long long));
  }

  for(unsigned int i = 0; i < row; i++)
    for(unsigned int j = 0; j < col; j++)
      Matrix[i][j]= rand()%MAX;

  return Matrix;
}
```

1.5 Generazione del Vettore

Anche il vettore è allocato dinamicamente e riempito con valori minori di 2^{16}

```
long long*  randomVector(unsigned int size){
  long long *Vector;
  Vector = malloc(size * sizeof(long long));
  for(unsigned int i = 0; i < size; i++){
    Vector[i]= rand()%MAX;
  }
  return Vector;
}
```

Capitolo 2

Esecuzione e Testing

Sommario

2.1 Compilazione

Per compilare il programma eseguire il comando

gcc -std=c99 -fopenmp -lgomp -o matricePerVettore matricePerVettore.c

I flag -fopenmp -lgomp servono al compilatore per linkare la libreria di openMP. Lo standard c99 o superiore è necessario per l'utilizzo di dichiarazioni di variabili nel for e il tipo long long.

2.2 Esecuzione

Per eseguire il comando lanciare

matricePerVettore $< row > < col >$

Per cambiare il numero massimo di thread va settata la variabile di sistema:

OMP_NUM_THREADS=$< threads >$

Il programma creerà una matrice casuale di $< row >$ per $< col >$ e la moltiplicherà con un vettore generato casualmente di $< col >$ elementi. Terminato il calcolo il main thread stamperà il tempo impiegato per effettuare la computazione (in secondi) e sulla seguente linea i $< row >$ elementi di Result separati da spazi.

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. I file di output sono reperibili alla pagina GitHub del progetto.

```
echo "Compiling matricePerVettore.c with gcc"
gcc -std=c99 -fopenmp -lgomp -o $PBS_O_WORKDIR/matricePerVettore
↪  $PBS_O_WORKDIR/matricePerVettore.c

echo "Start Testing"
for i in {1..5}
do
    for t in {1..8}
    do
        export OMP_NUM_THREADS=$t
        echo Run test on $OMP_NUM_THREADS threads
        $PBS_O_WORKDIR/matricePerVettore 100 100 >>
        ↪  $PBS_O_WORKDIR/results/${t}t_100x100.txt
        $PBS_O_WORKDIR/matricePerVettore 1000 1000 >>
        ↪  $PBS_O_WORKDIR/results/${t}t_1000x1000.txt
        $PBS_O_WORKDIR/matricePerVettore 10000 10000 >>
        ↪  $PBS_O_WORKDIR/results/${t}t_10000x10000.txt
    done
done
```

Lo script lancia il programma per 5 volte con input matrici quadrate di dimensione 100, 1000, 10000 con numero di trhead da 1 a 8. L'output è poi salvato nella cartella results in file con il seguente formato:

$$< threads > t\_ < size > x < size >$$

2.4 Esempio di esecuzione

Il seguente esempio mostra come compilare, eseguire il programma e impostare il numero massimo di threads.

Il primo comando compila il programma tramite gcc come spiegato nella sezione Compilazione. Il flag -std non è necessario dato che lo standard di default su questo specifico compilatore è superiore al c99.

Il terzo comando setta il numero di thread massimi a 1. Successivamente viene lanciato il programma con input le dimensioni per creare una matricce 10×10. Il programma viene rilanciato con $OMP\_NUM\_THREADS = 4$ e stesso input. In questo caso l'esecuzione con 4 thread è stata circa 4.7 volte più veloce di quella single-thread (Probabilmente per motivi di caching).

Capitolo 3

Analisi delle prestazioni

Sommario

3.1 Algoritmo non parallelo

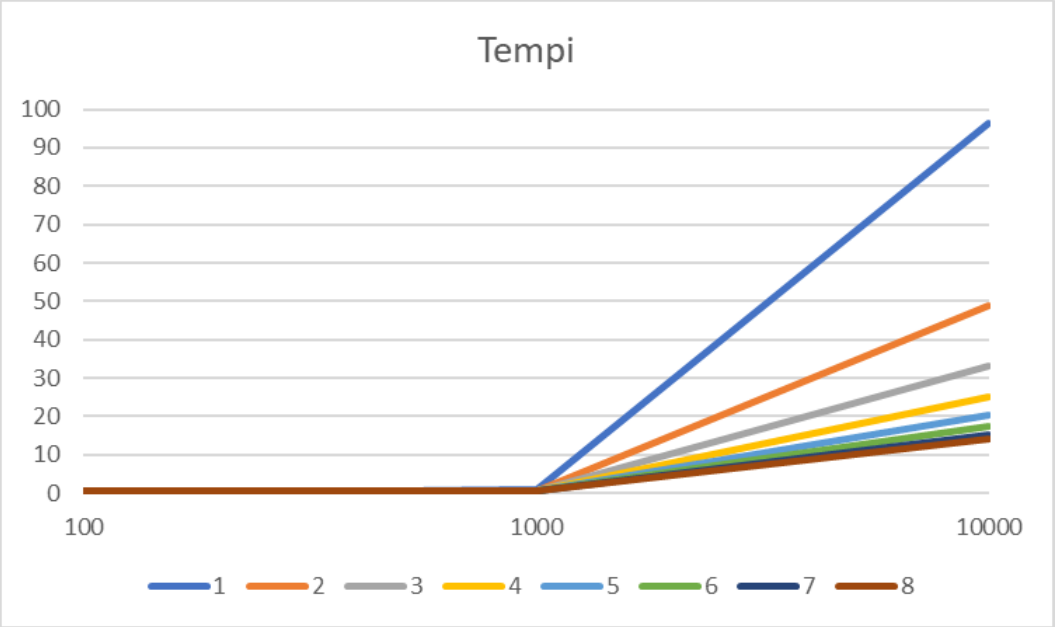
L'algoritmo non parallelo è stato testato settando $OMP\_NUM\_THREADS$ 1. Settando il numero di thread a 1 openMP ignora le direttive per il parallelismo che quindi non influiscono negativamente sul tempo di esecuzione finale.

3.2 Tempi di Esecuzione

Nella seguente tabella sono riportati i tempi relativi all'esecuzione del test del capitolo precedente. Il test è stato lanciato 2 volte per un totale di 10 misurazioni per ogni input e numero di thread da 1 a 8. La colonna di sinistra indica la dimensione della matrice quadrata in input ($100 \rightarrow \mathcal{M}_{100,100} \rightarrow 10000\, elementi$). I tempi sono riportati in $s \cdot 10^2$

	1	2	3	4	5	6	7	8
100	0.01208	0.15194	0.3107	0.34699	0.40396	0.39906	0.53422	0.65246
1000	1.02741	0.64456	0.67614	0.58675	0.56699	0.64601	0.58302	0.79617
10000	96.18422	48.82804	33.07041	25.04235	20.30654	17.29039	15.19124	14.12433

3.3 Considerazioni



L'esecuzione single-thread è la più veloce per la matrice 100×100. All'aumentare dei thread aumenta anche il tempo di esecuzione. Ciò perchè i cambi di contesto per la creazione dei thread sono molto dispendiosi e il calcolatore impiega meno a effettuare le somme sequenzialmente piuttosto che creare nuovi thread e parallelizzare il calcolo.

Con la matrice 1000×1000 si inizia ad otterene un guadagno dal parallelismo. L'esecuzione con 3 thread è più lenta di quella con 2, quella da 7 è più lenta di 6 e 5, e quella da 8 è la più lenta in assoluto tranne che di 1. Ciò è dovuto sia dal fatto che essendoci più thread ci sono più cambi di contesto sia perchè 3 e 7 non dividono perfettamente 1000.

Con matrici di 1000×1000 o superiori l'aumento delle prestazioni è significativo. All'aumentare dei thread diminuisce il tempo di esecuzione.

3.4 Speedup ed Efficienza

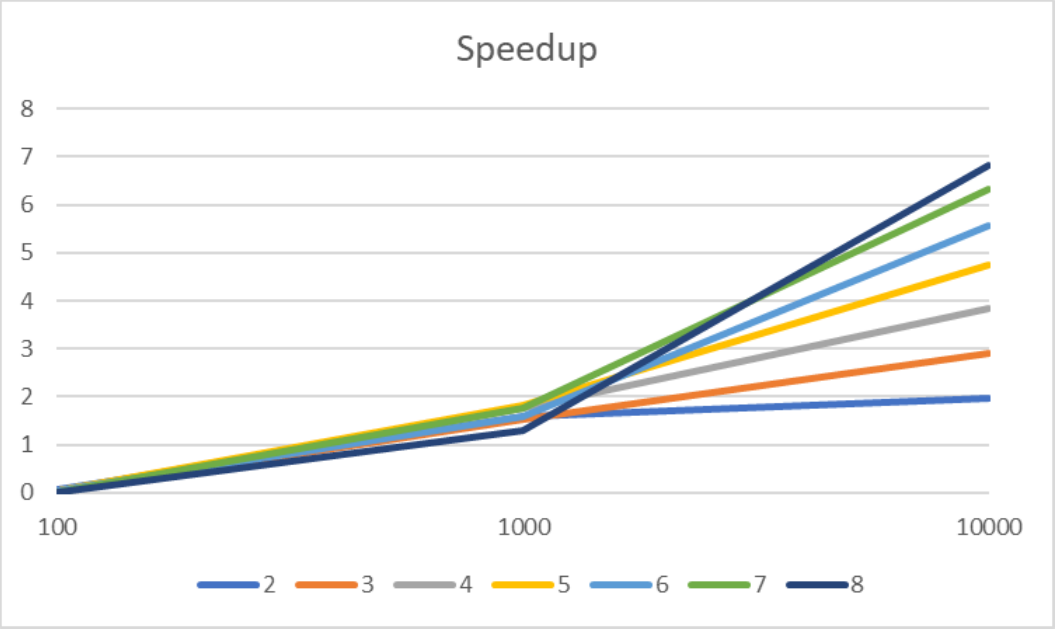
Siano Speedup ed Efficienza definiti come segue

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

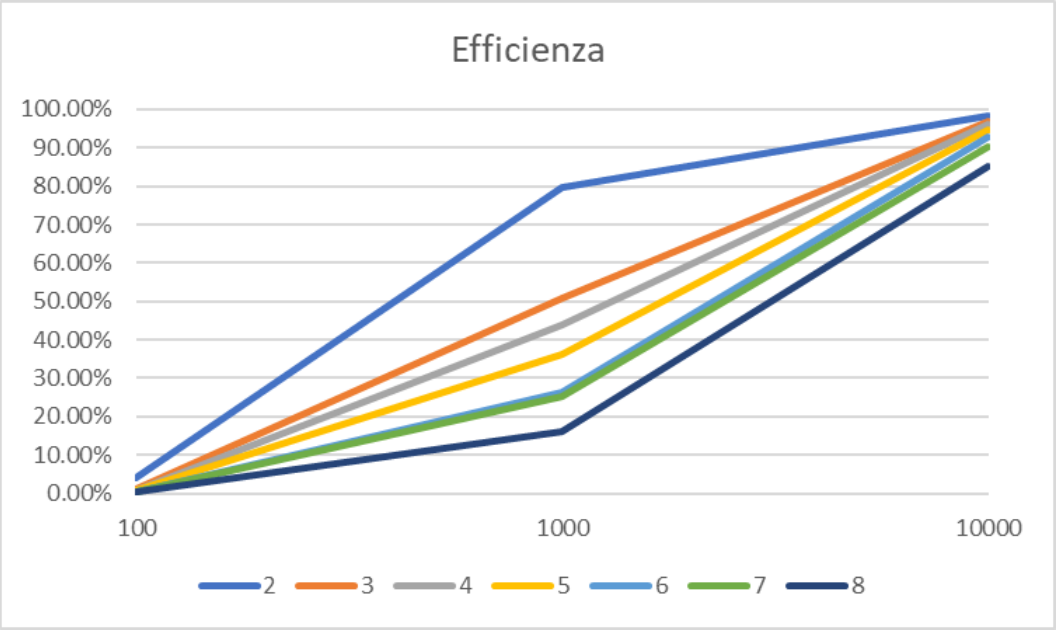
Applicando le equazioni ai dati riportati nelle tabelle si ottengono gli Speedup e le Efficienze delle varie esecuzioni.

input	100	1000	10000
$S(2)$	0.07951	1.59397	1.96986
$E(2)$	3.98%	79.70%	98.49%
$S(3)$	0.03888	1.51952	2.90847
$E(3)$	1.30%	50.65%	96.95%
$S(4)$	0.03481	1.75102	3.84086
$E(4)$	0.87%	43.78%	96.02%
$S(5)$	0.02990	1.81204	4.73661
$E(5)$	0.60%	36.24%	94.73%
$S(6)$	0.03027	1.59039	5.56287
$E(6)$	0.50%	26.51%	92.71%
$S(7)$	0.02261	1.76222	6.33155
$E(7)$	0.32%	25.17%	90.45%
$S(8)$	0.01851	1.29044	6.80982
$E(8)$	0.23%	16.13%	85.12%

Di seguito i grafici con i dati delle tabelle:



Come già discusso sopra c'è un aumento delle prestazioni solo con input grandi. Con l'aumentare della dimensione le esecuzioni si avvicinano allo speedup ideale.



Come da aspettative più è basso il numero di thread più è alta l'efficienza. Con la matrice di 10000 tutte le esecuzioni superano l'80% di efficienza.

Appendice A

Codice completo con Documentazione interna

```
/**
 *  Autore: Matteo Richard Gaudino
 *  Matricola: N86003226
 *
 **/

#include <omp.h> // omp_get_wtime, omp parallel, ...
#include <stdio.h> // printf
#include <stdlib.h> // malloc, rand, srand, atoi, ...
#include <string.h> // memset
#include <time.h> // time

const unsigned long long MAX = 1UL << 16;

// Genera una matrice con numeri casuali di dimensione row X col
long long** randomMatrix(unsigned int row, unsigned int col);
// Genera un vettore con numeri casuali di dimensione size
long long*  randomVector(unsigned int size);
// Esegue il prodotto Righe per colonne della matrice Matrix per il vettore
↪  colonna Vector
// Il vettore risultante viene scritto in Result
void matricePerVettore(long long **Matrix, unsigned int row, unsigned int col,
↪  long long *Vector, long long *Result);
```

```
22
23 int main(int argc, char *argv[]) {
24   long long **Matrix; // Matrice
25   long long *Vector; // Vettore
26   long long *Result; // Vettore risultante
27
28   unsigned int ROW = atoi(argv[1]); // = Result_size
29   unsigned int COL = atoi(argv[2]); // Righe e colonne della matrice. COL =
     ↪  Vector_size, ROW = Result_size
30
31   double t1, t2; // tempo di inizio e di fine
32
33   srand(time(NULL)); // Genera numeri casuali
34
35   // Creazione della matrice e del vettore
36   Matrix = randomMatrix(ROW, COL);
37   Vector = randomVector(COL);
38
39   // Allocazione e inizializzazione del vettore risultante
40   Result = malloc(ROW * sizeof(long long));
41   Result = memset(Result, 0, ROW);
42
43
44   t1 = omp_get_wtime(); // Tempo inizio
45
46   matricePerVettore(Matrix, ROW, COL, Vector, Result); // Calcolo del prodotto
47
48   t2 = omp_get_wtime(); // Tempo fine
49
50   printf("%f\n", t2-t1); // Stampa il tempo impiegato
51
52   // Stampa il Vettore Result
53 /*for (unsigned int i = 0; i < ROW; i++){ // Stampa il risultato
54     printf("%lld ", Result[i]);
55   }
56   printf("\n");*/
57   return 0;
```

```
58 }
59
60 // ------- Funzioni -------
61
62 long long** randomMatrix(unsigned int row, unsigned int col){
63
64   long long **Matrix;
65
66   Matrix = malloc(row * (sizeof(long long*))); // Matrice allocata come
     ↪  vettore di puntatori
67   for (unsigned int i = 0; i < row; i++){
68     Matrix[i] = malloc(col * sizeof(long long));
69   }
70
71   // Riempimento con elementi casuali < MAX
72   for(unsigned int i = 0; i < row; i++)
73     for(unsigned int j = 0; j < col; j++)
74       Matrix[i][j]= rand()%MAX;
75
76   return Matrix;
77 }
78
79 long long*  randomVector(unsigned int size){
80   long long *Vector;
81   Vector = malloc(size * sizeof(long long));
82   for(unsigned int i = 0; i < size; i++){ // Riempimento con elementi casuali
     ↪  < MAX
83     Vector[i]= rand()%MAX;
84   }
85   return Vector;
86 }
87
88 void matricePerVettore(long long **Matrix, unsigned int row, unsigned int col,
   ↪  long long *Vector, long long *Result){
89   unsigned int i, j;
90   // Sezone parallela
```

```
91    #pragma omp parallel for shared(Matrix, row, col, Vector, Result) private(i,
      ↪  j)
92    for(i = 0; i < row; i++){ // Le righe di Matrix sono suddivise sui vari
      ↪  thread
93      for(j = 0; j < col; j++)
94        Result[i] += Matrix[i][j] * Vector[j];
95    }
96  }
```

Appendice B

File PBS con Test

```
#!/bin/bash
############################
#
#
# The PBS directives #
#
#
############################
#PBS -q studenti
#PBS -l nodes=1:ppn=8
#PBS -N matricePerVettore
#PBS -o matricePerVettore.out
#PBS -e matricePerVettore.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 #
#
#
########################################
NCPU=`wc -l < $PBS_NODEFILE`
```

```
27 echo ------------------------------------------------------
28 echo ' This job is allocated on '${NCPU}' cpu(s)'
29 echo 'Job is running on node(s): '
30 cat $PBS_NODEFILE
31
32
33 PBS_O_WORKDIR=$PBS_O_HOME/matricePerVettore
34 echo ------------------------------------------------------
35 echo PBS: qsub is running on $PBS_O_HOST
36 echo PBS: originating queue is $PBS_O_QUEUE
37 echo PBS: executing queue is $PBS_QUEUE
38 echo PBS: working directory is $PBS_O_WORKDIR
39 echo PBS: execution mode is $PBS_ENVIRONMENT
40 echo PBS: job identifier is $PBS_JOBID
41 echo PBS: job name is $PBS_JOBNAME
42 echo PBS: node file is $PBS_NODEFILE
43 echo PBS: current home directory is $PBS_O_HOME
44 echo PBS: PATH = $PBS_O_PATH
45 echo ------------------------------------------------------
46
47 echo "Compiling matricePerVettore.c with gcc"
48 gcc -std=c99 -fopenmp -lgomp -o $PBS_O_WORKDIR/matricePerVettore
   ↪ $PBS_O_WORKDIR/matricePerVettore.c
49
50 echo "Start Testing"
51 for i in {1..5}
52 do
53     for t in {1..8}
54     do
55         export OMP_NUM_THREADS=$t
56         echo Run test on $OMP_NUM_THREADS threads
57         $PBS_O_WORKDIR/matricePerVettore 100 100 >>
           ↪ $PBS_O_WORKDIR/results/${t}t_100x100.txt
58         $PBS_O_WORKDIR/matricePerVettore 1000 1000 >>
           ↪ $PBS_O_WORKDIR/results/${t}t_1000x1000.txt
59         $PBS_O_WORKDIR/matricePerVettore 10000 10000 >>
           ↪ $PBS_O_WORKDIR/results/${t}t_10000x10000.txt
```

```
60          done
61      done
```