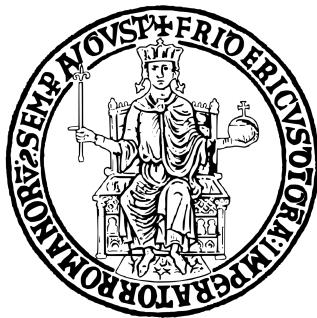


TO UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

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-vettore, in ambiente di calcolo parallelo
su architettura MIMD a memoria condivisa**

Docenti

Prof. Giuliano Laccetti
Prof. ssa Valeria Mele

Studenti

Francesco Jr. Iaccarino – N97000440
Fabiola Salomone – N97000457

Questa pagina è stata lasciata intenzionalmente bianca.

INDICE

1.	Definizione ed Analisi del problema	5
2.	Descrizione dell'algoritmo	6
2.1	Descrizione dell'algoritmo	6
3	Input, Output ed errori	9
3.1	Input	9
3.2	Restrizioni	10
3.3	Output.....	10
4	Descrizione delle Subroutine	13
4.1	Subroutine personalizzate	13
4.1.1	Subroutine - check_input	13
4.1.2	Subroutine -matxvet	14
4.2	Direttive OpenMp	15
4.3	Subroutine Standar del linguaggio C	15
4.3.1	Funzione - calloc.....	15
4.3.2	Funzione - malloc.....	16
4.3.3	Funzione - atoi.....	16
4.3.4	Funzione - exit.....	16
5.	Esempi d'uso	18
5.1	Script PBS utilizzato per l'esecuzione dell'algoritmo	18
5.1	Esempi d'uso	20
5.1.1	Esempi 1 - matrice 8x8 e vettore di 8 elementi	20
5.1.2	Esempi 2 – matrice 14x10 e vettore di 10 elementi	22
5.1.3	Esempi 3 – Dimensione delle righe matrice non valida	24
5.1.4	Esempi 4 – Dimensione delle colonne della matrice e del vettore non valida	26
6	Analisi delle performance	28
6.4.1	Analisi con N = 1.000 e M = {1.000, 5.000, 10.000}.....	28
6.4.2	Analisi con N = 5.000 e M = {1.000, 5.000, 10.000}.....	31
6.4.3	Analisi con N = 10.000 e M = {1.000, 5.000, 10.000}.....	33
6.4.4	Conclusioni analisi dei tempi	36
7	Codice Sorgente.....	37

1. Definizione ed Analisi del problema

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

L'obiettivo dell'algoritmo è quello di eseguire il prodotto scalare tra una matrice $A \in R^{(N \times M)}$ ed un vettore $X \in R^M$. Il risultato di tale prodotto è un vettore $Y \in R^N$, ovvero un vettore con dimensione pari al numero di righe n della matrice.

Condizione necessaria affinchè l'algoritmo possa essere eseguito è che la dimensione del vettore X sia uguale al numero di colonne della matrice A.

L'algoritmo produrrà poi in output un vettore $Y \in R^N$, le cui componenti sono ottenute come segue:

$$y_i = \sum_{j=1}^m a_{ij} x_j = a_{i_1} x_1 + a_{i_2} x_2, \dots, a_{i_m} x_m$$

L'algoritmo dovrà ricevere in input dati come :

- La dimensione N delle righe della matrice.
- La dimensione M delle colonne della matrice

Per il secondo input, si assume che la dimensione delle M colonne della matrice dove essere anche la dimensione del vettore Y.

Successivamente si analizzano come variano i tempi al variare del numero di threads utilizzati e al variare della dimensione del problema.

2. Descrizione dell'algoritmo

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

2.1 Descrizione dell'algoritmo

L'algoritmo è strutturato in sei fasi distinte:

1. **Fase 1:** Input e Verifica;
2. **Fase 2:** Allocazione di memoria;
3. **Fase 3:** Inizializzazione della Matrice e del Vettore
4. **Fase 4:** Calcolo del Prodotto Matrice-Vettore.
5. **Fase 5:** Output
6. **Fase 6:** Deallocazione della Memoria

Nella prima fase l'algoritmo inizia prendendo in input due valori fondamentali n , che rappresenta il numero di righe della matrice, e m , la dimensione delle colonne della matrice e del vettore. Successivamente, una verifica accurata segue questa fase, assicurando che n e m siano entrambi valori maggiori di zero. Qualora uno di essi non rispetta questa condizione l'algoritmo interrompe la sua esecuzione, restituendo un messaggio di errore. Questa attenzione ai dettagli garantisce che il calcolo sia eseguito in un contesto significativo e senza ambiguità.

Avanziamo quindi nell'arena delle allocazioni dinamiche, la seconda fase, dove le strutture dati prendono forma. La matrice A , il vettore x , e il vettore risultato ris vengono creati e allocati per accogliere i dati necessari al calcolo. Questa fase di preparazione è cruciale per garantire che l' algoritmo possa operare su dati validi e ben strutturati.

Con le strutture dati pronte per l'algoritmo, si procede con la terza fase in cui l'algoritmo dà vita alla matrice A e al vettore x . La matrice viene popolata con valori crescenti da 1 a $n \times m$, creando così un contesto ricco di dati significativi. Allo stesso modo, il vettore x assume valori crescenti da 1 a m , preparandosi così a svolgere il suo ruolo nel calcolo.

A tal punto l'algoritmo, quarta fase, attiva un “cronometro”.

Attraverso l'impiego delle direttive della libreria OpenMp ossia “**#pragma omp parallel for**”, l'algoritmo suddivide il lavoro tra i thread disponibili. Ogni thread ha il compito di calcolare una porzione del prodotto matrice-vettore, contribuendo così a un aumento significativo delle prestazioni grazie al parallelismo. Con l'abilità di OpenMp il risultato parziale di ogni thread viene aggregato nel vettore risultato *ris*. Al termine della regione parallela, il cronometro si ferma, e il tempo totale impiegato per il calcolo viene rivelato permettendoci di apprezzare l'efficienza dell'algoritmo.

L'algoritmo presenta infine il suo prodotto ,quinta fase, cioè il vettore risultato *ris*. Con un semplice comando di stampa, il risultato del prodotto matrice-vettore si presenta sulla console del terminale, consegnando il risultato del calcolo all'utente nella forma più appropriata.

Con il completatamento del calcolo, l'algoritmo effettua un'ultima operazione, sesta fase, cioè la deallocazione della memoria precedentemente allocata. Le porzioni di memoria centrale occupata dalle strutture dati che hanno svolto il loro ruolo nel processo, vengono restituite al sistema, garantendo un uso efficiente delle risorse.

Si evidenzia che il codice implementato è un connubio tra programmazione sequenziale e programmazione parallela di preciso viene fatta in sequenziale la

parte relativa all’allocazione di memoria; all’inizializzazione della matrice e del vettore; la chiamata alla funzione “*matxvet*” e la stampa del risultato. Queste operazioni vengono eseguite in sequenziale da un singolo thread. Mentre è stata implementata in parallelo la parte di codice che si occupa del calcolo del prodotto scalare tra la matrice e il vettore attraverso il quale diversi threads lavorano su righe differenti della matrice in modo simultaneo, migliorando le prestazioni complessive dell’algoritmo.

3 Input, Output ed errori

In questo capitolo, saranno esaminati gli aspetti concernenti 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 capitolo relativo all'introduzione, i parametri richiesti dall'algoritmo sono i seguenti:

- La dimensione N delle righe della matrice.
- La dimensione M delle colonne della matrice.

Tutti i parametri devono essere passati come argomento all'interno del file PBS, alla riga corrispondente all'esecuzione del programma, nel seguente ordine:

1. Numero di colonne;
2. Numero di righe;

Per comprendere meglio dove inserire tali parametri, si mostra (figura) di seguito il punto esatto in cui vanno inseriti i parametri (numero di righe e colonne della matrice) nel file PBS :

```
#!/bin/bash
# Impostare le variabili dell'ambiente
#PBS -l nodes=1:ppn=8
#PBS -l matvet
#PBS -o matvet.out
#PBS -e matvet.err
PBS_O_WORKDIR=$PBS_O_HOME/Elaborato2
gcc -fopenmp -fomp -o $PBS_O_WORKDIR/matvet $PBS_O_WORKDIR/prodotto_mat_vet.c
#####
## CUSTOM VALUES ##
#####
COLS=7
ROWS=18
THREADS=(1 2 3 4 5 6 7)
#####
for THREAD in ${THREADS[@]}; do
    echo "Job-Script) Starting with $THREAD Threads..."
    export OMP_NUM_THREADS=$THREAD
    export PSC_OMP_AFFINITY=TRUE
    echo -e "\n==== Threads $THREAD - Matrix $COLUMN x $ROWS ====\n"
    $PBS_O_WORKDIR/matvet $COLUMN $ROWS
done
```

Cioe:

```
1. #####
2. ## CUSTOM VALUES ##
3. #####
4. COLUMN=10
5. ROWS=10
6. THREADS=(1 2 3 4 5 6 7 )
7. #####
8.
9. for THREAD in "${THREADS[@]}"; do
10.   echo "[Job-Script] Starting with $THREAD Threads..."
11.
12.   export OMP_NUM_THREADS=$THREAD
13.   export PSC_OMP_AFFINITY=TRUE
14.
15.   echo -e "\n==== Threads $THREAD - Matrix $COLUMN x $ROWS ===\n"
16.   $PBS_O_WORKDIR/matxvet $COLUMN $ROWS
17. done
18.
```

3.2 Restrizioni

I parametri devono rispettare determinate restrizioni/costrizioni:

- I valori che rappresentano il numero di righe e colonne della matrice devono essere entrambi maggiori di zero.

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

3.3 Output

Vediamo ora alcuni esempi di output che ci possiamo aspettare in base a alcuni input di esempio. Nell'algoritmo, la matrice è inizializzata da l a $n \times m$, mentre il vettore è inizializzato da l a m .

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

tempo impiegato : <tempo in secondi>
<risultato>

I valori tra parentesi angolari corrispondono a :

- <risultato> : il risultato del prodotto tra la matrice e il vettore;
- <tempo in secondi> : il tempo totale impiegato dai threads per calcolare il prodotto richiesto.

Vediamo alcuni output relativi agli input nella forma (<n>, <m>):

- INPUT: 8 8
- OUTPUT:

```
[Job-Script] Starting with 8 Threads...
```

```
==== Threads 8 - Matrix 8 x 8 ===
```

```
tempo impiegato: 0.009112
x[0]: 204
x[1]: 492
x[2]: 780
x[3]: 1068
x[4]: 1356
x[5]: 1644
x[6]: 1932
x[7]: 2220
```

- INPUT: 6 3
- OUTPUT:

```
==== Threads 6 - Matrix 6 x 3 ===
```

```
tempo impiegato: 0.005122
x[0]: 14
x[1]: 32
x[2]: 50
x[3]: 68
x[4]: 86
x[5]: 104
```

- INPUT: 5 0
- OUTPUT:

[Job-Script] Starting with 3 Threads...

==== Threads 3 - Matrix 5 x 0 ===

Il numero di colonne della matrice (e quindi dimensione vettore) deve essere un valore maggiore di 0.

- INPUT: 0 7
- OUTPUT:

[Job-Script] Starting with 7 Threads...

==== Threads 7 - Matrix 0 x 7 ===

Il numero di righe della matrice deve essere un valore maggiore di 0

- INPUT: 4 3
- OUTPUT:

[Job-Script] Starting with 5 Threads...

==== Threads 5 - Matrix 4 x 3 ===

```
tempo impiegato: 0.003099
x[0]: 14
x[1]: 32
x[2]: 50
x[3]: 68
```

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

4 Descrizione delle Subroutine

In questo capitolo vengono descritte le subroutine utilizzate nell'algoritmo, queste sono documentate all'interno del codice tramite 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

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

4.1.1 Subroutine – `check_input`

La funzione “`check_input`” verifica la correttezza degli argomenti forniti da riga di comando.

Possiede la seguente firma:

```
void check_input(int argc, char **argv)
```

Descriviamo i parametri presi in input dalla funzione :

- `argc` : rappresenta il numero totale di argomenti passati da riga di comando.
- `argv` : rappresenta un array di stringhe contenente gli argomenti passati da riga di comando.

La funzione accetta il numero totale di argomenti (`argc`) e un array di stringhe (`argv`) e controlla che siano presenti esattamente tre argomenti. Inoltre,

verifica che i valori numerici associati al numero di righe e colonne siano entrambi maggiori di zero. In caso di violazione di queste condizioni, stampa messaggi di errori appropriati e termina il programma con un codice di errore.

4.1.2 Subroutine –matxvet

La funzione “matxvet” calcola il prodotto matrice-vettore in modo parallelo utilizzando le direttive della libreria OpenMp.

Possiede la seguente firma :

```
int* matxvet(int n, int m, int *x, int **A)
```

Descriviamo i parametri presi in input dalla funzione :

- n : rappresenta la dimensione delle righe della matrice
- m : rappresenta la dimensione delle colonne della matrice e del vettore
- x : rappresenta il vettore di input
- A : rappresenta la matrice di input

Inoltre, la funzione restituisce il vettore risultato contenente il prodotto matrice-vettore.

Precisamente la funzione prende in input la dimensione delle righe (**n**), delle colonne (**m**), un vettore (**x**), e una matrice (**A**). Successivamente alloca dinamicamente un vettore risultato (**ris**) e inizializza i suoi elememnti a 0. Utilizzando la libreria OpenMp per parallelizzare il calcolo del prodotto matrice-vettore, disritbuisce il lavoro tra i thread disponibili. Infine, restituisce il vettore risultante.

La funzione stampa, inoltre, anche il tempo impiegato per l'esecuzione del calcolo in modalità parallela.

4.2 Direttive OpenMp

La direttiva utilizzata nell'algoritmo in questione è la seguente :

```
#pragma omp parallel for default(none) shared(m,n,A,x,b)
    private(i,j)
```

Dove :

- pragma omp : è un costrutto utilizzato per specificare direttive di compilazione a OpenMp (Open Multi-Processing);
- parallel: forma un team di thread ed avvia così un'esecuzione parallela;
- for : specifica che le iterazioni del ciclo contenuto devono essere distribuite tra i team;
- default(none) : controlla gli attributi di data-sharing delle variabili in un costrutto
 - shared : tutte le variabili saranno considerate condivise;
 - none : deciderà tutto il programmatore.
- shared(list) : gli argomenti contenuti in list sono condivisi tra i thread del team;
- private(list) : gli argomenti contenuti in list sono privati per ogni thread che li utilizza.

4.3 Subroutine Standard del linguaggio C

4.3.1 Funzione - calloc

La funzione *calloc* si occupa di allocare memoria dinamica per un certo numero di variabili di una determinata dimensione, e inizializza la memoria allocata a zero. La firma è :

```
void *calloc(size_t nitems, size_t size)
```

Utilizzata per allocare i blocchi di memoria per le strutture dati utilizzate come : vettore e matrice, inizializzandoli a zero.

Tale funzione prende in input :

- nitems : numero di elementi da allocare ;
- size : dimensione (grandezza) di ciascun elemento

4.3.2 Funzione - malloc

La funzione *malloc* si occupa di allocare memoria dinamica per un certo numero specifica. La firma è :

```
void* malloc( size_t size)
```

Utilizzata per l'allocazione di blocchi di memoria di dimensione arbitrarie.

Tale funzione prende in input :

- size : numero di byte da allocare

4.3.3 Funzione - atoi

La funzione *atoi* si occupa di convertire una stringa (array di caratteri) che rappresenta un numero intero in una variabilr di tipo *int*. La firma è :

```
int atoi(const char *str)
```

Tale funzione prende in input :

- str : stringa che rappresenta un numero intero

4.3.4 Funzione - exit

La funzione *exit* si occupa di terminare il programma, chiudendo il flusso di esecuzione principale. La firma è :

```
void exit(int status)
```

Viene utilizzata per uscire dal programma in modo controllato, eseguendo operazioni di pulizia e di terminazione .

Tale funzione prende in input :

- status : valore dello stato che ritorna al processo padre.

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. # Impostare le variabili dell'ambiente
4. #PBS -q studenti
5. #PBS -l nodes=1:ppn=8
6. #PBS -N matxvet
7. #PBS -o matxvet.out
8. #PBS -e matxvet.err
9.
10. PBS_O_WORKDIR=$PBS_O_HOME/Elaborato3
11.
12. gcc -fopenmp -lgomp -o $PBS_O_WORKDIR/matxvet
$PBS_O_WORKDIR/prodotto_mat_vet.c
13.
14. #####
15. ## CUSTOM VALUES ##
16. #####
17. COLUMN=4
18. ROWS=3
19. THREADS=(1 2 3 4 5 6 7 8 )
20. #####
21.
22. for THREAD in "${THREADS[@]}"; do
23.     echo "[Job-Script] Starting with $THREAD Threads..."
24.
25.     export OMP_NUM_THREADS=$THREAD
26.     export PSC_OMP_AFFINITY=TRUE
27.
28.     echo -e "\n==== Threads $THREAD - Matrix $COLUMN x $ROWS ===\n"
29.     $PBS_O_WORKDIR/matxvet $COLUMN $ROWS
30. done
```

Questo file denominato “*elaborato3.pbs*” ci permette di compilare ed eseguire il programma C denominato “*prodotto_mat_vet.c*” che si occupa di calcolare il prodotto scalare tra una matrice di dimensione nxm e un vettore di dimensione m su un cluster di 8 nodi. Le uscite dell'esecuzione vengono reindirizzate ai file “*output.out*” ed “*error.err*”.

Descriviamo brevemente i vari parametri presenti nel file pbs :

1. Definizione delle direttive PBS :

- ‘`#PBS -q studenti`’ : specifica la coda su cui verrà eseguito il job
- ‘`#PBS -l nodes=1 : ppn=8`’ : richiede 1 nodo con 8 processori
- ‘`#PBS -N elaborato3`’ : 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. Definizione della directory di lavoro :

- ‘`PBS_O_WORKDIR=$PBS_O_HOME/Elaborato3`’ : imposta la directory di lavoro a ‘`=$PBS_O_HOME/Elaborato3`’

3. Compilazione del codice sorgente C :

- ‘`gcc -fopenmp .lgomp -o $PBS_O_WORKDIR/matxvet $PBS_O_WORKDIR/prodotto_mat_vet.c`’ : compila il programma C “`prodotto_mat_vet.c`” con il supporto per OpenMp e crea l’eseguibile “`matxvet`” nella directory di lavoro specificata

4. Personalizzazione dei valori :

- ‘`COLUMN =4`’ e ‘`ROWS=3`’ : definiscono il numero di colonne e righe per la matrice.
- ‘`THREADS = (1 2 3 4 5 6 7 8)`’ : specifica un array di valori per il numero di thread da utilizzare in ogni interazione del ciclo successivo

5. Ciclo per eseguire il programma con diversi numeri di thread

- ‘`for THREAD in “$(THREADS[@]}”`; do : inizializza un ciclo che attraversa gli elementi dell’array “`THREADS`”.
- ‘`export OMP_UM_THREADS = $THREAD`’ : imposta il numero di thread OpenMP da utilizzare.
- ‘`export PSC_OMP_AFFINITY=TRUE`’ : abilita l’affinità del processore per i thread OpenMp

- ‘\$PBS_O_WORKDIR/matxvet \$COLUMN \$ROWS’ : esegue il programma ‘matxvet’ con i parametri specificati (numero di colonne e righe della matrice) e utilizzando il numero corrente di thread.

Il primo passo per eseguire e compilare il programma “*prodotto_mat_vet.c*” è eseguire il file pbs, ciò viene fatto collegandoci al cluster, posizionandoci nella cartella contenente il file “*prodotto_mat_vet.c*” e eseguendo sul terminale il comando “*qsub elaborato3.pbs*”.

All’ esecuzione del comando “*qsub elaborato3.pbs*” si generano diversi file, in particolare:

1. L’ esecutore relativo al programma scritto in linguaggio C;
2. Un file “*output.out*” contenente l’ output del programma. Questo output è generato dai valori dati in input nell’ ultima riga del file “*elaborato3.pbs*” tra cui abbiamo il numero di threads, la dimensione delle righe e delle colonne della matrice;
3. Un file “*error.err*” contenente eventuali errori del programma. In caso di esecuzione senza errore, questo file rimarrà vuoto, come in questo caso.

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 - matrice 8x8 e vettore di 8 elementi

Tenendo conto del seguente file “*elaborato3.pbs*” :

```

#!/bin/bash

# Impostare le variabili dell'ambiente
#PBS -N elaborato2
#PBS -l nodes=1:ppn=8
#PBS -N matxvet
#PBS -o output.out
#PBS -e error.err

PBS_O_WORKDIR=$PBS_O_HOME/Elaborato2
gco -fopenmp -lgomp -o $PBS_O_WORKDIR/matrixvet $PBS_O_WORKDIR/prodotto_mat_vet.c

#####
## CUSTOM VALUES ##
#####
COL=8
ROWS=8
THREADS=(1 2 3 4 5 6 7 8 )
#####

for THREAD in "${THREADS[@]}"; do
    echo "[Job-Script] Starting with $THREAD Threads..."
    export OMP_NUM_THREADS=$THREAD
    export PSC_OMP_AFFINITY=TRUE
    echo -e "n==== Threads $THREAD - Matrix $COLUMN x $ROWS ====\n"
    $PBS_O_WORKDIR/matrixvet $COLUMN $ROWS
done

```

Come possiamo vedere, viene dato l'input al programma passando i parametri come argomenti. Il programma verrà lanciato con un numero di threads che va da 1 a 8 come indicato nella variabile “*export OMP_NUM_THREADS*”, nel ciclo *for*.

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 “*error.err*” che sarà vuoto nel caso sia andato tutto bene (come in questo caso). Per vedere l'output del programma eseguiamo il comando “*cat output.out*”.

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

```

[Job-Script] Starting with 1 Threads...
*** Threads 1 - Matrix 8 x 8 ***
tempo eseguito: 0.000004
x[0]: 284
x[1]: 492
x[2]: 988
x[3]: 1868
x[4]: 3356
x[5]: 6144
x[6]: 1932
x[7]: 3824
[Job-Script] Starting with 2 Threads...
*** Threads 2 - Matrix 8 x 8 ***
tempo eseguito: 0.000165
x[0]: 284
x[1]: 492
x[2]: 988
x[3]: 1868
x[4]: 3356
x[5]: 6144
x[6]: 1932
x[7]: 3824
[Job-Script] Starting with 3 Threads...
*** Threads 3 - Matrix 8 x 8 ***
tempo eseguito: 0.000216
x[0]: 284
x[1]: 492
x[2]: 988
x[3]: 1868
x[4]: 3356
x[5]: 6144
x[6]: 1932
x[7]: 3824
[Job-Script] Starting with 4 Threads...
*** Threads 4 - Matrix 8 x 8 ***
tempo eseguito: 0.000218
x[0]: 284
x[1]: 492
x[2]: 988
x[3]: 1868
x[4]: 3356
x[5]: 6144
x[6]: 1932
x[7]: 3824
[Job-Script] Starting with 5 Threads...
*** Threads 5 - Matrix 8 x 8 ***

```

```

● ● ● Bia — SLMFBL99X@ui-studenti:~/Elaborato2 — ssh -oKexAlgorithms+=diffie-hellman-group1-sha1 -oHostKeyAlgorithms+=ssh-rsa SLMFBL99X@ui-studenti.scope.unina.it...
x[0]: 1932
x[7]: 2228
[Job-Script] Starting with 5 Threads...
*** Threads 5 - Matrix 8 x 8 ***
tempo impiegato: 0.003112
x[0]: 284
x[1]: 492
x[2]: 780
x[3]: 1068
x[4]: 1356
x[5]: 1644
x[6]: 1932
x[7]: 2228
[Job-Script] Starting with 6 Threads...
*** Threads 6 - Matrix 8 x 8 ***
tempo impiegato: 0.004228
x[0]: 284
x[1]: 492
x[2]: 780
x[3]: 1068
x[4]: 1356
x[5]: 1644
x[6]: 1932
x[7]: 2228
[Job-Script] Starting with 7 Threads...
*** Threads 7 - Matrix 8 x 8 ***
tempo impiegato: 0.005961
x[0]: 284
x[1]: 492
x[2]: 780
x[3]: 1068
x[4]: 1356
x[5]: 1644
x[6]: 1932
x[7]: 2228
[Job-Script] Starting with 8 Threads...
*** Threads 8 - Matrix 8 x 8 ***
tempo impiegato: 0.008071
x[0]: 284
x[1]: 492
x[2]: 780
x[3]: 1068
x[4]: 1356
x[5]: 1644
x[6]: 1932
x[7]: 2228
[SLMFBL99X@ui-studenti Elaborato2]$ 

```

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 – matrice 14x10 e vettore di 10 elementi

Tenendo del seguente file “*elaborato3.pbs*” :

```

● ● ● Bia — SLMFBL99X@ui-studenti:~/Elaborato2 — ssh -oKexAlgorithms+=diffie-hellman-group1-sha1 -oHostKeyAlgorithms+=ssh-rsa SLMFBL99X@ui-studenti.scope.unina.it...
GNU nano 1.3.12 File: elaborato2.pbs
#!/bin/bash

# Impostare le variabili dell'ambiente
#PBS -q studenti
#PBS -l nodes=1:ppn=8
#PBS -N matvet
#PBS -o error.out
#PBS -e error.err

PBS_O_WORKDIR=$PBS_O_HOME/Elaborato2
gcc -fopenmp -lgomp -o $PBS_O_WORKDIR/matvet $PBS_O_WORKDIR/prodotto_mat_vet.c

#####
## CUSTOM VALUES #
#####
COLUMN=14
ROWS=18
THREADS=(1 2 3 4 5 6 7 8 )
#####

for THREAD in ${THREADS[@]}; do
    echo "[Job-Script] Starting with $THREAD Threads..."
    export OMP_NUM_THREADS=$THREAD
    export PSC_OMP_AFFINITY=TRUE
    echo "Number of threads $THREAD - Matrix $COLUMN x $ROWS ===\n"
done

```

Come possiamo vedere, viene dato l'input al programma passando i parametri come argomenti. Il programma verrà lanciato con un numero di threads che va da 1 a 8 come indicato nella variabile “*export OMP_NUM_THREADS*” nel ciclo *for*.

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 “*error.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 output.out*”.

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

```
[@ ~] Bio - SLMPBL99X@ui-studenti:~/Elaborato2 - ssh -oKexAlgorithms=diffie-hellman-group1-sha1 -oHostKeyAlgorithms=ssh-rsa SLMPBL99X@ui-studenti.scope.unina.it...
[Job-Script] Starting with 1 Threads...
*** Threads 1 - Matrix 14 x 10 ***
tempo impiegato: 0.000014
x[0]: 385
x[1]: 935
x[2]: 1485
x[3]: 2885
x[4]: 2685
x[5]: 3185
x[6]: 3685
x[7]: 4285
x[8]: 4785
x[9]: 5385
x[10]: 5985
x[11]: 6485
x[12]: 6985
x[13]: 7585
[Job-Script] Starting with 2 Threads...
*** Threads 2 - Matrix 14 x 10 ***
tempo impiegato: 0.002238
x[0]: 385
x[1]: 935
x[2]: 1485
x[3]: 2885
x[4]: 2685
x[5]: 3185
x[6]: 3685
x[7]: 4285
x[8]: 4785
x[9]: 5385
x[10]: 5985
x[11]: 6485
x[12]: 6985
x[13]: 7585
[Job-Script] Starting with 3 Threads...
*** Threads 3 - Matrix 14 x 10 ***
tempo impiegato: 0.004172
x[0]: 385
x[1]: 935
x[2]: 1485
x[3]: 2885
x[4]: 2685
x[5]: 3185
x[6]: 3685
x[7]: 4285
x[8]: 4785
x[9]: 5385
x[10]: 5985
x[11]: 6485
x[12]: 6985
x[13]: 7585
[Job-Script] Starting with 4 Threads...
*** Threads 4 - Matrix 14 x 10 ***
tempo impiegato: 0.008331
x[0]: 385
x[1]: 935
x[2]: 1485
x[3]: 2885
x[4]: 2685
x[5]: 3185
x[6]: 3685
x[7]: 4285
x[8]: 4785
x[9]: 5385
x[10]: 5985
x[11]: 6485
x[12]: 6985
x[13]: 7585
[Job-Script] Starting with 5 Threads...
*** Threads 5 - Matrix 14 x 10 ***
tempo impiegato: 0.0090010
x[0]: 385
x[1]: 935
x[2]: 1485
x[3]: 2885
x[4]: 2685
x[5]: 3185
x[6]: 3685
x[7]: 4285
x[8]: 4785
x[9]: 5385
x[10]: 5985
x[11]: 6485
x[12]: 6985
x[13]: 7585
[Job-Script] Starting with 6 Threads...
*** Threads 6 - Matrix 14 x 10 ***
tempo impiegato: 0.003143
x[0]: 385
x[1]: 935
x[2]: 1485
x[3]: 2885
x[4]: 2685
x[5]: 3185
x[6]: 3685
x[7]: 4285
x[8]: 4785
x[9]: 5385
x[10]: 5985
x[11]: 6485
x[12]: 6985
x[13]: 7585
```

```
[@ ~] Bio - SLMPBL99X@ui-studenti:~/Elaborato2 - ssh -oKexAlgorithms=diffie-hellman-group1-sha1 -oHostKeyAlgorithms=ssh-rsa SLMPBL99X@ui-studenti.scope.unina.it...
[Job-Script] Starting with 7 Threads...
*** Threads 7 - Matrix 14 x 10 ***
tempo impiegato: 0.003143
x[0]: 385
x[1]: 935
x[2]: 1485
x[3]: 2885
x[4]: 2685
x[5]: 3185
x[6]: 3685
x[7]: 4285
x[8]: 4785
x[9]: 5385
x[10]: 5985
x[11]: 6485
x[12]: 6985
x[13]: 7585
[Job-Script] Starting with 8 Threads...
*** Threads 8 - Matrix 14 x 10 ***
tempo impiegato: 0.003143
x[0]: 385
x[1]: 935
x[2]: 1485
x[3]: 2885
x[4]: 2685
x[5]: 3185
x[6]: 3685
x[7]: 4285
x[8]: 4785
x[9]: 5385
x[10]: 5985
x[11]: 6485
x[12]: 6985
x[13]: 7585
```

```

○ Bia - SLMFBL99X@ui-studenti:~/Elaborato2 - ssh -oKexAlgorithms=difile-hellman-group1-sha1 -oHostKeyAlgorithms=ssh-rsa SLMFBL99X@ui-studenti.scope.unina.it...
*** Threads 6 - Matrix 14 x 10 ***
tempo impiegato: 0.003143
{x0}: 385
{x1}: 935
{x2}: 1485
{x3}: 2835
{x4}: 2885
{x5}: 1235
{x6}: 3485
{x7}: 4285
{x8}: 770
{x9}: 5335
{x10}: 5885
{x11}: 6435
{x12}: 6985
{x13}: 7535
[Job-Script] Starting with 7 Threads...
*** Threads 7 - Matrix 14 x 10 ***
tempo impiegato: 0.005197
{x0}: 385
{x1}: 935
{x2}: 1485
{x3}: 2835
{x4}: 2885
{x5}: 1235
{x6}: 3485
{x7}: 4285
{x8}: 770
{x9}: 5335
{x10}: 5885
{x11}: 6435
{x12}: 6985
{x13}: 7535
[Job-Script] Starting with 8 Threads...
*** Threads 8 - Matrix 14 x 10 ***
tempo impiegato: 0.005222
{x0}: 385
{x1}: 935
{x2}: 1485
{x3}: 2835
{x4}: 2885
{x5}: 1235
{x6}: 3485
{x7}: 4285
{x8}: 770
{x9}: 5335
{x10}: 5885
{x11}: 6435
{x12}: 6985
{x13}: 7535

```

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

5.1.3 Esempi 3 – Dimensione delle righe matrice non valida

Tenendo del seguente file “elaborato3.pbs” :

```

○ Bia - SLMFBL99X@ui-studenti:~/Elaborato2 - ssh -oKexAlgorithms=difile-hellman-group1-sha1 -oHostKeyAlgorithms=ssh-rsa SLMFBL99X@ui-studenti.scope.unina.it...
GNU nano 1.3.12
File: elaborato2.pbs
#!/bin/bash
# Impostare le variabili dell'ambiente
#PBS -q studenti
#PBS -l nodes=1:ppn=8
#PBS -N matvel
#PBS -o output.out
#PBS -e error.err
PBS_O_WORKDIR=$PBS_O_HOME/Elaborato2
gcc -fopenmp -I/home/a/PBS_O_HOME/Matvel/matvvt $PBS_O_WORKDIR/prodotto_mat_vvt.c
#####
## CUSTOM VALUES ##
#####
ROWS=3
ROWS=3
THREADS=(1 2 3 4 5 6 7 8 )
#####

for THREAD in "${THREADS[@]}"; do
    echo "[Job-Script] Starting with $THREAD Threads..."
    export OMP_NUM_THREADS=$THREAD
    export PSC_OMP_AFFINITY=TRUE
    echo -e '\n==== Thread $THREAD - Matrix $COLUMN x $ROWS ===\n'
    $PBS_O_WORKDIR/matvvt $COLUMN $ROWS
done

[ Read 33 lines ] [ Prev Page ] [ Next Page ] [ Cut Text ] [ Uncut Text ] [ Cur Pos ] [ To Spell ]

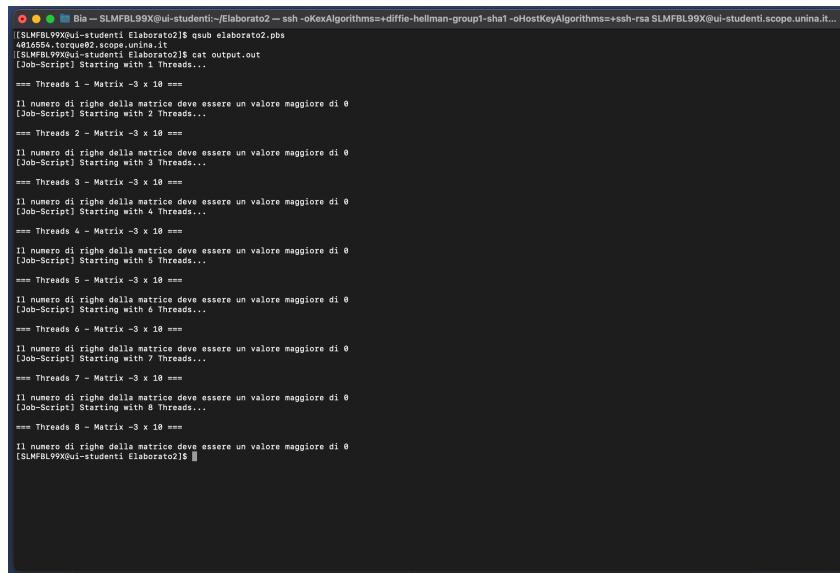
```

Come possiamo vedere, viene dato l'input al programma passando i parametri come argomenti. Il programma verrà lanciato con un numero di threads che va da 1 a 8 come indicato nella variabile “*export OMP_NUM_THREADS*” nel ciclo *for*.

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 “*error.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 output.out*”.

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



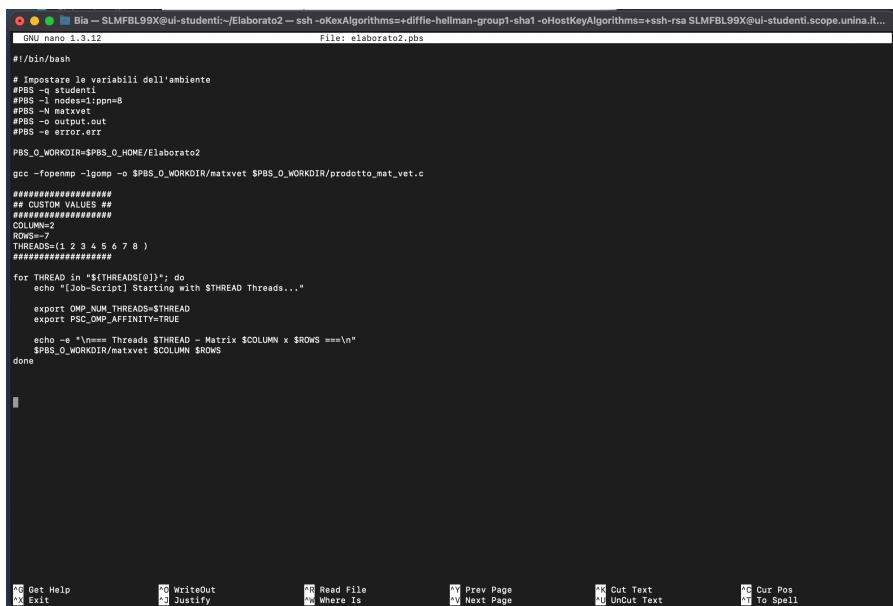
The image shows a terminal window with a black background and white text. At the top, there is a small icon bar with several colored dots (red, yellow, green, blue) followed by the text "Bia -- SLMFBL99X@ui-studenti:~/[Elaborato2 -- ssh -oKexAlgorithms=+diffie-hellman-group1-sha1 -oHostKeyAlgorithms=+ssh-rsa SLMFBL99X@ui-studenti.scope.unina.it..". Below this, there is a series of command-line interactions:

```
[SLMFBL99X@ui-studenti Elaborato2]$ qsub elaborato2.pbs
4916554.torque02.scope.unina.it
[SLMFBL99X@ui-studenti Elaborato2]$ cat output.out
[Job-Script] Starting with 1 threads...
==== Threads 1 - Matrix -3 x 10 ====
Il numero di righe della matrice deve essere un valore maggiore di 0
[Job-Script] Starting with 2 threads...
==== Threads 2 - Matrix -3 x 10 ====
Il numero di righe della matrice deve essere un valore maggiore di 0
[Job-Script] Starting with 3 threads...
==== Threads 3 - Matrix -3 x 10 ====
Il numero di righe della matrice deve essere un valore maggiore di 0
[Job-Script] Starting with 4 threads...
==== Threads 4 - Matrix -3 x 10 ====
Il numero di righe della matrice deve essere un valore maggiore di 0
[Job-Script] Starting with 5 threads...
==== Threads 5 - Matrix -3 x 10 ====
Il numero di righe della matrice deve essere un valore maggiore di 0
[Job-Script] Starting with 6 threads...
==== Threads 6 - Matrix -3 x 10 ====
Il numero di righe della matrice deve essere un valore maggiore di 0
[Job-Script] Starting with 7 threads...
==== Threads 7 - Matrix -3 x 10 ====
Il numero di righe della matrice deve essere un valore maggiore di 0
[Job-Script] Starting with 8 threads...
==== Threads 8 - Matrix -3 x 10 ====
Il numero di righe della matrice deve essere un valore maggiore di 0
[SLMFBL99X@ui-studenti Elaborato2]$
```

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

5.1.4 Esempi 4 – Dimensione delle colonne della matrice e del vettore non valida

Tenendo del seguente file “elaborato3.pbs” :



```
GNU nano 1.3.12
File: elaborato2.pbs

#!/bin/bash

# Impostare le variabili dell'ambiente
#PBS -q studenti
#PBS -l nodes=1:ppn=8
#PBS -N matvet
#PBS -o output.out
#PBS -e error.err

PBS_O_WORKDIR=$PBS_O_HOME/Elaborato2
gcc -fopenmp -lgomp -o $PBS_O_WORKDIR/matxvet $PBS_O_WORKDIR/prodotto_mat_vet.c

#####
## CUSTOM VALUES ##
#####
COLUMN=2
ROWS=8
THREADS=(1 2 3 4 5 6 7 8)
#####

for THREAD in ${THREADS[@]}; do
    echo "[Job-Script] Starting with $THREAD Threads..."
    export OMP_NUM_THREADS=$THREAD
    export PSC_OMP_AFFINITY=TRUE
    echo -e "\n==== Threads $THREAD - Matrix $COLUMN x $ROWS ====\n"
    $PBS_O_WORKDIR/matxvet $COLUMN $ROWS
done

|
```

Come possiamo vedere, viene dato l’input al programma passando i parametri come argomenti. Il programma verrà lanciato con un numero di threads che va da 1 a 8 come indicato nella variabile “*export OMP_NUM_THREADS*” nel ciclo *for*.

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 “*error.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 output.out*”.

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

```
Bia - SLMFBL99X@ui-studenti:~/Elaborato2 - ssh -oKexAlgorithms=+diffie-hellman-group1-sha1 -oHostKeyAlgorithms=+ssh-rsa SLMFBL...
[SLMFBL99X@ui-studenti Elaborato2]$ qsub elaborato2.pbs
[4016576.torque02.scope.unina.it]
[SLMFBL99X@ui-studenti Elaborato2]$ cat output.out
[Job-Script] Starting with 1 Threads...
==== Threads 1 - Matrix 2 x -7 ====
Il numero di colonne della matrice (e quindi dimensione vettore) deve essere un valore maggiore di 0
[Job-Script] Starting with 2 Threads...
==== Threads 2 - Matrix 2 x -7 ====
Il numero di colonne della matrice (e quindi dimensione vettore) deve essere un valore maggiore di 0
[Job-Script] Starting with 3 Threads...
==== Threads 3 - Matrix 2 x -7 ====
Il numero di colonne della matrice (e quindi dimensione vettore) deve essere un valore maggiore di 0
[Job-Script] Starting with 4 Threads...
==== Threads 4 - Matrix 2 x -7 ====
Il numero di colonne della matrice (e quindi dimensione vettore) deve essere un valore maggiore di 0
[Job-Script] Starting with 5 Threads...
==== Threads 5 - Matrix 2 x -7 ====
Il numero di colonne della matrice (e quindi dimensione vettore) deve essere un valore maggiore di 0
[Job-Script] Starting with 6 Threads...
==== Threads 6 - Matrix 2 x -7 ====
Il numero di colonne della matrice (e quindi dimensione vettore) deve essere un valore maggiore di 0
[Job-Script] Starting with 7 Threads...
==== Threads 7 - Matrix 2 x -7 ====
Il numero di colonne della matrice (e quindi dimensione vettore) deve essere un valore maggiore di 0
[Job-Script] Starting with 8 Threads...
==== Threads 8 - Matrix 2 x -7 ====
Il numero di colonne della matrice (e quindi dimensione vettore) deve essere un valore maggiore di 0
```

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

6 Analisi delle performance

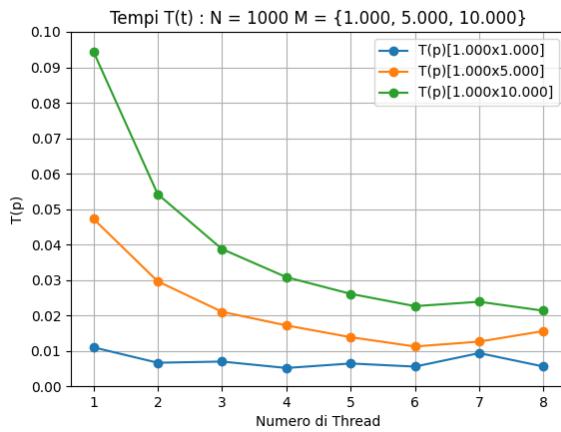
In questo capitolo andremo a valutare le prestazioni di tale algoritmo che si occupa del prodotto scalare tra matrice e vettore mediante parametri utilizzati per valutare un software parallelo che sono : *tempo di esecuzione* con “t” thread; *Speed Up* e *Efficienza*.

Di seguito vengono riportate le formule relative :

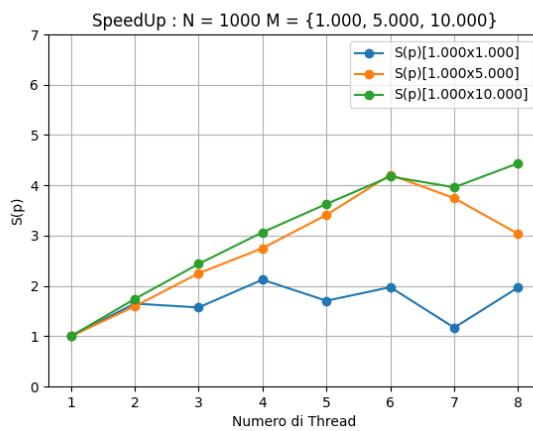
- $T(t)$: Tempo di esecuzione con “t” thread
 - Questo rappresenta il tempo totale che il programma impiega per eseguire con t thread.
- $S(t) = \frac{T(1)}{T(t)}$: Speed Up
 - Rappresenta il guadagno di prestazioni ottenuto dalla parallelizzazione. Indica quanto velocemente il programma può essere eseguito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

6.4.1 Analisi con $N = 1.000$ e $M = \{1.000, 5.000, 10.000\}$

Thread/s	T(p) [1.000x1.000]	T(p)[1.000x5.000]	T(p)[1.000x10.000]
1	0.011001	0.047260	0.094460
2	0.006648	0.029617	0.054213
3	0.006980	0.021018	0.038719
4	0.005169	0.017181	0.030795
5	0.006437	0.013858	0.026086
6	0.005559	0.011231	0.022616
7	0.009369	0.012610	0.023853
8	0.005574	0.015595	0.021319

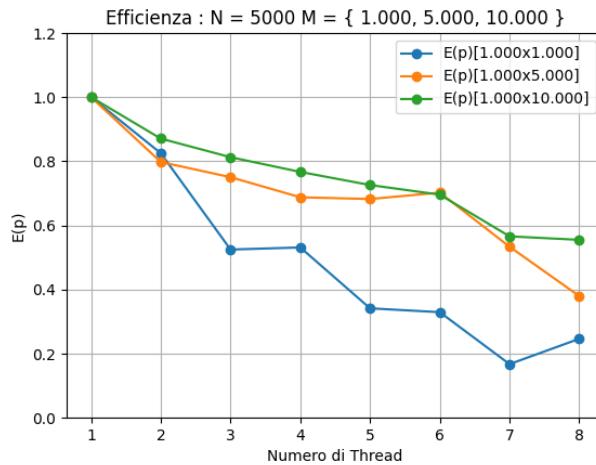


Thread/s	S(p) [1.000x1.000]	S(p)[1.000x5.000]	S(p)[1.000x10.000]
1	1	1	1
2	1.650240	1.595396	1.741425
3	1.574075	2.251083	2.438209
4	2.125062	2.751010	3.067088
5	1.707162	3.411013	3.629502
6	1.976331	4.2113769	4.175614
7	1.171818	3.745720	3.962102
8	1.971037	3.036390	4.439551



Thread/s	E(p) [1.000x1.000]	E(p)[1.000x5.000]	E(p)[1.000x10.000]
1	1	1	1
2	0.825120	0.797698	0.870712
3	0.524692	0.750361	0.812736

4	0.531266	0.687753	0.766772
5	0.341432	0.682203	0.725900
6	0.329388	0.702295	0.695936
7	0.167402	0.5335103	0.565872
8	0.246380	0.379549	0.554944



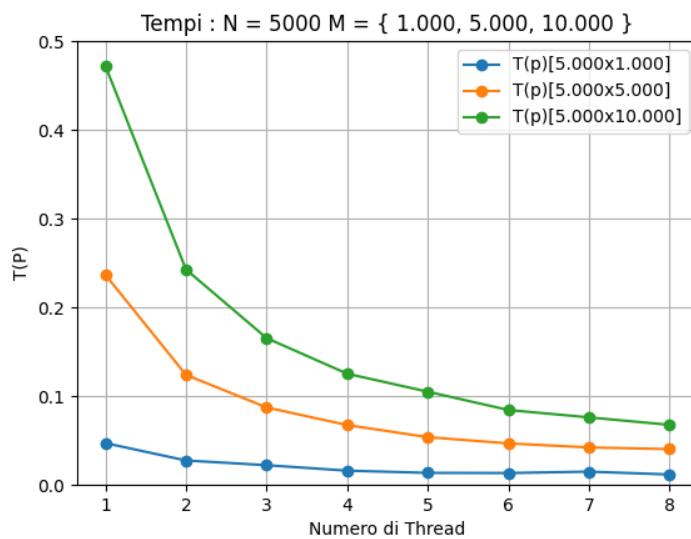
Esaminando i risultati dello SpeedUp e Efficienza per le diverse configurazioni, possiamo trarre alcune osservazioni :

- **Dimensioni 1.000 x 1.000** : aumentando il numero di thread, lo SpeedUp aumenta, indicando un miglioramento delle prestazioni mentre l'Efficienza è inferiore a 1 per la maggior parte delle configurazioni, il che suggerisce che il parallelismo non è completamente sfruttato e ci potrebbe essere un certo overhead nell'aggiunta di thread. Inoltre per alcune configurazioni, l'Efficienza diminuisce all'aumentare del numero di thread, indicando che l'aumento del parallelismo potrebbe non essere completamente sfruttato.
- **Dimensioni 1.000 x 5.000** : le stesse tendenze osservate nelle dimensioni 1.000 x 1.000 si ripetono anche in tal dimensione, dove aumentando il numero di thread lo SpeedUp aumenta ma l'Efficienza è inferiore a 1 per molte configurazioni.
- **Dimensioni 1.000 x 10.000** : ancora una volta, l'aumento del numero di thread porta a un aumento di SpeedUp. Tuttavia, l'Efficienza può diminuire all'aumentare del numero di thread per alcune configurazioni.

In generale, l'efficienza diminuisce quando il numero di thread aumenta, suggerendo un certo overhead o limite di scalabilità ed il parallelismo sembra beneficiale di più nelle dimensioni della matrice più grandi (1.000×5.000 e 1.000×10.000), dove veiè un aumento più significativo di SpeedUp.

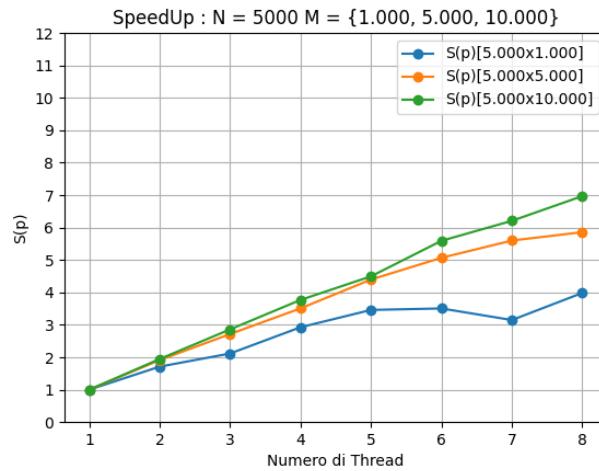
6.4.2 Analisi con $N = 5.000$ e $M = \{1.000, 5.000, 10.000\}$

Thread/s	$T(p) [5.000 \times 1.000]$	$T(p)[5.000 \times 5.000]$	$T(p)[5.000 \times 10.000]$
1	0.047192	0.236804	0.472145
2	0.027614	0.124029	0.242729
3	0.022315	0.087365	0.165448
4	0.016143	0.067486	0.125327
5	0.013661	0.053886	0.105057
6	0.013473	0.046848	0.084452
7	0.015028	0.042307	0.076132
8	0.011846	0.040428	0.067750

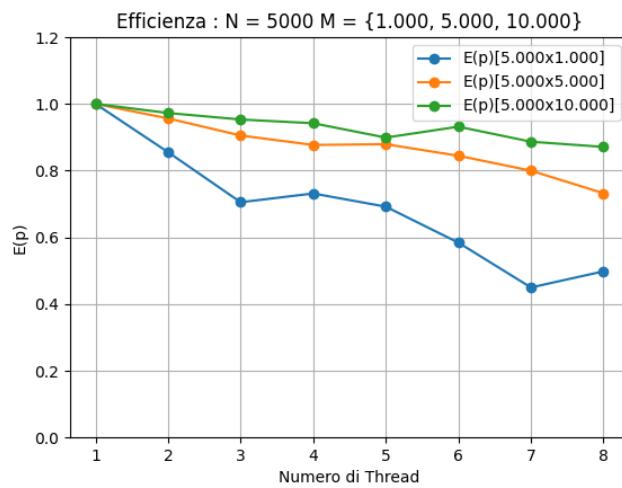


Thread/s	$S(p) [5.000 \times 1.000]$	$S(p)[5.000 \times 5.000]$	$S(p)[5.000 \times 10.000]$
1	1	1	1
2	1.710380	1.912875	1.945221
3	2.115516	2.715609	2.859511
4	2.925813	3.507404	3.767669
5	3.459226	4.395703	4.494724
6	3.504725	5.066704	5.589768

7	3.147809	5.598595	6.206380
8	3.983997	5.856664	6.968264



Thread/s	E(p) [5.000x5.000]	E(p)[5.000x10.000]	E(p)[5.000x10.000]
1	1	1	1
2	0.855190	0.956437	0.972611
3	0.705172	0.905203	0.953170
4	0.731453	0.876851	0.941917
5	0.691845	0.879141	0.898945
6	0.584121	0.844451	0.931628
7	0.449687	0.799799	0.886626
8	0.497999	0.732083	0.8710033



Esaminando i risultati dello SpeedUp e Efficienza per le diverse configurazioni, possiamo trarre alcune osservazioni :

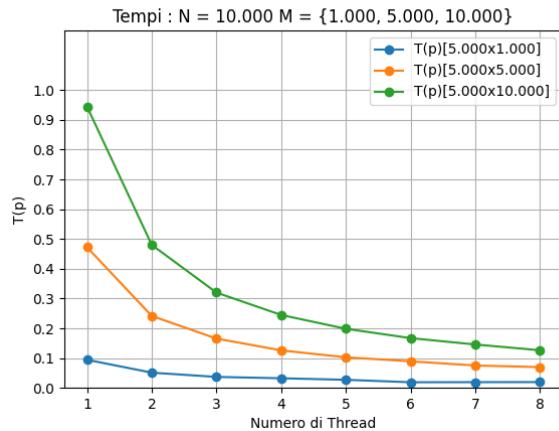
- **Dimensioni 5.000 x 1.000** : lo SpeedUp aumenta con il numero di thread, il che indica un miglioramento delle prestazioni.
L'Efficienza diminuisce all'aumentare del numero di thread, il che indica che l'overhead dell'utilizzo di più thread potrebbe superare i benefici dell'accelerazione (per accelerazione si intende la capacità di eseguire un compito più velocemente utilizzando più risorse di elaborazione in parallelo). Potrebbe esserci una soglia oltre la quale l'efficienza diminuisce a causa dell'aumento del coordinamento tra i thread e degli eventuali blocchi dovuti alla concorrenza.
- **Dimensioni 5.000 x 5.000** : lo SpeedUp aumenta in modo più pronunciato rispetto alle dimensioni più piccole, ma l'Efficienza inizia a diminuire dopo un certo numero di thread. Potrebbe esserci una saturazione delle risorse o un limite nell'Efficacia della parallelizzazione. Tale fenomeno indica che, anche se l'accelerazione iniziale è significativa, raggiungo un certo punto, l'aumento del numero di thread non fornisce un miglioramento proporzionale.
- **Dimensioni 5.000 x 10.000** : l'aumento dello SpeedUp è meno marcato rispetto alle dimensioni 5.000 x 5.000, ma l'Efficienza mostra una tendenza più evidente al calo. Ciò potrebbe indicare che, per problemi di dimensioni maggiori, l'efficienza della parallelizzazione può essere più difficile da mantenere per tal motivo potrebbero emergere limitazioni dovute alle dimensioni del problema, alla capacità del sistema o a un'eventuale saturazione delle risorse.

In generale, l'efficienza tende a diminuirsi con un aumento del numero di thread, indicando che l'overhead introdotto dall'utilizzo di più thread potrebbe influire negativamente sull'accelerazione complessiva. È importante, dunque trovare un equilibrio ottimale tra parallelismo e overhead per ottenere prestazioni ottimali.

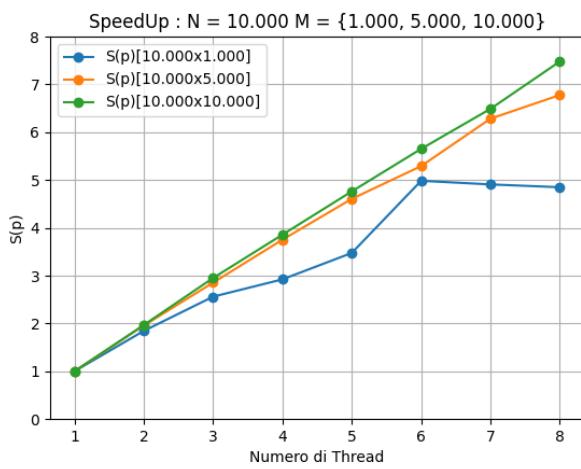
6.4.3 Analisi con N = 10.000 e M = {1.000, 5.000, 10.000}

Thread/s	T(p) [10.000x1.000]	T(p)[10.000x5.000]	T(p)[10.000x10.000]
1	0.094301	0.472107	0.944373
2	0.051135	0.241507	0.479955
3	0.036826	0.165534	0.319955
4	0.032270	0.125888	0.244792
5	0.027129	0.102655	0.198390

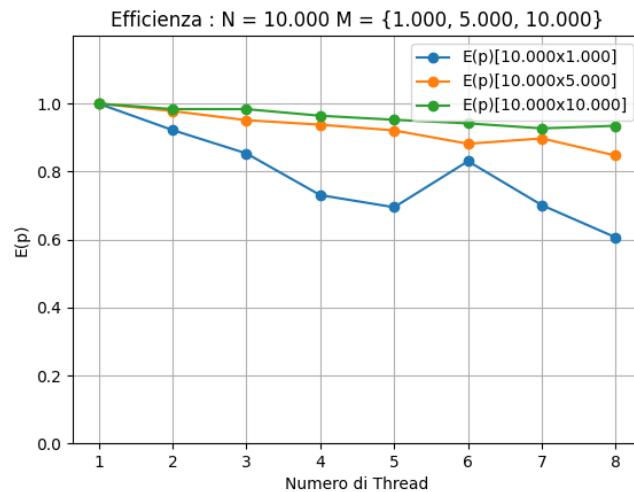
6	0.018904	0.089137	0.167037
7	0.019218	0.075141	0.145634
8	0.019470	0.069690	0.126289



Thread/s	S(p) [10.000x1.000]	S(p)[10.000x5.000]	S(p)[10.000x10.000]
1	1	1	1
2	1.844442	1.956077	1.967743
3	2.560588	2.853897	2.951221
4	2.922738	3.751897	3.857201
5	3.474783	4.605949	4.762035
6	4.983423	5.291172	5.651268
7	4.908046	6.283212	6.488667
8	4.849282	6.776991	7.478168



Thread/s	E(p) [10.000x1.000]	E(p)[10.000x5.000]	E(p)[10.000x10.000]
1	1	1	1
2	0.922221	0.978039	0.983872
3	0.853529	0.951299	0.983740
4	0.730685	0.937974	0.964300
5	0.694957	0.921190	0.952407
6	0.830571	0.881862	0.941878
7	0.701149	0.897602	0.927095
8	0.606160	0.847124	0.934771



Esaminando i risultati dello SpeedUp e Efficienza per le diverse configurazioni, possiamo trarre alcune osservazioni :

- **Dimensioni 10.000 x 1.000** : lo SpeedUp aumenta con il numero di thread indicando un miglioramento delle prestazioni. Tuttavia, il guadagno non è proporzionale al numero di thread, il che suggerisce la presenza di overhead o limitazioni nell'efficacia della parallelizzazione. Mentre l'Efficienza è inferiore a 1 per la maggior parte delle configurazioni, il che indica che l'utilizzo di più thread potrebbe non essere completamente efficiente.
- **Dimensioni 10.000 x 5.000** : lo SpeedUp mostra un aumento significativo con il numero di thread. Tuttavia, si osserva una leggera diminuzione dell'incremento oltre un certo numero di thread. Potrebbe indicare una saturazione delle risorse o una limitazione nell'efficacia della parallelizzazione. Mentre l'Efficienza è più alta rispetto alle dimensioni più piccole, ma

diminuisce leggermente con il numero di thread, indicando una possibile saturazione delle risorse.

- **Dimensioni 10.000 x 5.000** : l'incremento dello SpeedUp è meno marcato rispetto alle dimensioni 10.000 x 5.000, e si osserva una tendenza più evidente al calo. Questo suggerisce che per problemi di dimensioni maggiori, mantenere un alto livello di efficienza nella parallelizzazione può essere più difficile. Mentre l'Efficienza mostra una diminuzione più significativa, indicando che per problemi di dimensioni maggiori, la parallelizzazione può essere meno efficiente.

In generale, all'aumentare delle dimensioni della matrice l'incremento dello SpeedUp è meno marcato e l'Efficienza mostra una tendenza al calo più evidente. Ciò suggerisce che la parallelizzazione potrebbe diventare più difficile da gestire ed efficiente per problemi di dimensioni maggiori. Inoltre per alcune configurazioni, specialmente per dimensioni 10.000 x 1.000 e 10.000 x 5.000, l'Efficienza diminuisce con l'aumentare del numero di thread.

6.4.4 Conclusione analisi dei tempi

I dati e i grafici precedenti evidenziano che, in presenza di dimensioni relativamente piccole, il variare del numero di threads non contribuisce ad un miglioramento significativo delle prestazioni; anzi, in alcuni casi si potrebbe osservare un peggioramento. Questo fenomeno potrebbe essere attribuito all'effetto dello scheduling standard della parallelizzazione. Introducendo uno scheduling dynamic, potrebbero manifestarsi leggeri miglioramenti. Tuttavia, quando ci troviamo di fronte a dimensioni del problema più elevate, si nota chiaramente un aumento deciso dello SpeedUp, accompagnato da un'efficienza che risulta essere quasi costante e molto elevata. Questa tendenza suggerisce che, in scenari con dimensioni maggiori, l'approccio di parallelizzazione risulti più efficace e efficiente, mentre per dimensioni minori potrebbe essere necessarie strategie di scheduling più avanzate per sfruttare appieno le potenzialità della parallelizzazione.

7 Codice Sorgente

```
1. #include <omp.h>
2. #include <stdio.h>
3. #include <stdlib.h>
4. #include <sys/time.h>
5.
6.
7. // Dichiarazione Funzioni
8. int* matxvet(int,int,int*,int**);
9. void check_input(int,char**);
10.
11. int main(int argc, char** argv){
12.     int **A, *x, *ris;
13.     int n, m, i, j;
14.
15.     check_input(argc,argv); //Chiamata a funzione che verifica se i dati
passati in input rispettano le restrizioni indicate nell'intestazione
16.
17.     n = atoi(argv[1]); //Prendo in input il numero di righe della matrice
18.     m = atoi(argv[2]); //Prendo in input il numero di colonne della
matrice e la dimensione del vettore
19.
20.     //----- INIZIO ALLOCAZIONI
21.
22.     A = (int**) malloc (n*sizeof(int*)); //Alloco le righe della matrice
23.     for(i=0;i<n;i++)
24.         A[i] = (int*) malloc (m*sizeof(int)); //Alloco le colonne della
matrice
25.
26.     x = (int*) malloc (m*sizeof(int)); //Alloco il vettore
27.
28.     //----- FINE ALLOCAZIONI
29.
30.
31.     //----- INIZIO INIZIALIZZAZIONE MATRICE E VETTORE
32.
33.     for(i=0;i<n;i++)
34.         for(j=0;j<m;j++)
35.             A[i][j] = (i*m) + j + 1; //Inizializzo la matrice da 1 a NxM
36.
37.     for(i=0; i<m; i++)
38.         x[i] = i+1; //Inizializzo il vettore da 1 a M
39.
40.     //----- FINE INIZIALIZZAZIONE MATRICE E VETTORE
41.
42.     ris = matxvet(n,m,x,A); //Chiamo la funzione che effettua il prodotto
matrice-vettore Ax
43.
44.     for(i=0;i<n;i++)
45.         printf("x[%d]: %d\n",i,ris[i]); //Stampo il risultato
46.
47.     return 0;
48. }
49.
50.
51. /**
52. * Calcola il prodotto matrice-vettore in modo parallelo utilizzando
| OpenMP.
```

```

53. *
54. * @param n La dimensione delle righe della matrice.
55. * @param m La dimensione delle colonne della matrice e del vettore.
56. * @param x Il vettore di input.
57. * @param A La matrice di input.
58. * @return int* Il vettore risultato del prodotto matrice-vettore.
59. *
60. * Questa funzione prende in input la dimensione delle righe (n), delle
61. * colonne (m), un vettore (x)
62. * e una matrice (A). Alloca dinamicamente un vettore risultato (ris) e
63. * inizializza i suoi elementi a 0.
64. * Utilizza la libreria OpenMP per parallelizzare il calcolo del prodotto
65. * matrice-vettore, distribuendo
66. * il lavoro tra i thread disponibili. Infine, restituisce il vettore
67. * risultato.
68. */
69. int* matxvet(int n, int m, int *x, int **A){
70.     int i,j;
71.     int *ris;
72.     ris = (int*) calloc (n,sizeof(int)); //Allocò il vettore risultato e
73.     inizializzo i suoi elementi a 0
74.     struct timeval time;
75.     double inizio, fine;
76.     gettimeofday(&time, NULL);
77.     inizio=time.tv_sec+(time.tv_usec/1000000.0); //Prendo il tempo
iniziale
78.     //----- INIZIO REGIONE PARALLELA
79.     #pragma omp parallel for default(none) shared(A,x,m,n,ris)
private(i,j)
80.         for(i=0; i<n; i++)
81.             for(j=0; j<m; j++)
82.                 ris[i] += A[i][j]*x[j]; //Effettuo il prodotto matrice-
vettore in parallelo
83.     //----- FINE REGIONE PARALLELA
84.     gettimeofday(&time, NULL);
85.     fine=time.tv_sec+(time.tv_usec/1000000.0); //Prendo il tempo finale
86.     printf("tempo impiegato: %f\n", fine-inizio); //Stampo il tempo di
87.     esecuzione del prodotto matrice-vettore
88.     return ris; //Ritorno il vettore risultato
89. }
90. /**
91. * Verifica la correttezza degli argomenti forniti da riga di comando.
92. *
93. * @param argc Il numero totale di argomenti passati da riga di comando.
94. * @param argv Un array di stringhe contenente gli argomenti passati da
95. * riga di comando.
96. *
97. * Questa funzione accetta il numero totale di argomenti (argc) e un
98. * array di stringhe (argv)
99. * e controlla che siano presenti esattamente tre argomenti. Inoltre,
100. * verifica che i valori numerici

```

```

101. * associati al numero di righe e colonne siano entrambi maggiori di
zero. In caso di violazioni di
102. * queste condizioni, stampa messaggi di errore appropriati e termina il
programma con un codice di errore.
103. *
104. */
105. void check_input(int argc, char **argv){
106.     if(argc!=3){
107.         printf("Inserire i 2 argomenti richiesti: \n1. Numero di righe
della matrice\n2. Numero di colonne della matrice (e quindi dimensione
vettore)\n");
108.         exit(1);
109.     }
110.     if(atoi(argv[1]) <= 0){
111.         printf("Il numero di righe della matrice deve essere un valore
maggiore di 0\n");
112.         exit(2);
113.     }
114.     if(atoi(argv[2]) <= 0){
115.         printf("Il numero di colonne della matrice (e quindi dimensione
vettore) deve essere un valore maggiore di 0\n");
116.         exit(3);
117.     }
118. }
119.

```