UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II
SCUOLA POLITECNICA E DELLE SCIENZE DI BASE
DIPARTIMENTO DI INGEGNERIA ELETTRICA E TECNOLOGIE DELL'INFORMAZIONE

CORSO DI LAUREA IN INFORMATICA

Anno Accademico 2021/2022

Algoritmo per il prodotto matrice vettore in OpenMP

Autore

Carmine Grimaldi

Matr. N97000394

Sommario

1. Definizione ed analisi del problema 3

2. Descrizione dell'algoritmo 3

3. Guida alla compilazione e manuale d'uso 4

Passaggio dei parametri 4

Input, output e condizioni di errore 5

Esempio di funzionamento 5

Esempi di errore 6

Funzioni 7

4. Analisi delle prestazioni 8

Tempi di esecuzione T(p) 9

SpeedUp S(p) 11

Efficienza E(p) 13

Conclusioni 15

5. Codice sorgente 16

1. Definizione ed analisi del problema

Si vuole progettare un algoritmo in **OpenMP** per risolvere il prodotto vettoriale tra una matrice di dimensioni M*N, e un vettore di numeri reali di dimensione N, su p thread.
In particolare si utilizza l'infrastruttura **S.C.o.P.E.** per permettere l'esecuzione del software in un ambiente parallelo. Tuttavia, a differenza degli elaborati MPI, il software è eseguibile anche su una macchina multicore locale.

2. Descrizione dell'algoritmo

In particolare, l'algoritmo implementato nel file elaborato2.c allegato a questa documentazione, consiste nel calcolo del prodotto matrice vettore in parallelo utilizzando le direttive e le clausole fornite da **OpenMP**. Si è scelto di misurare i tempi di esecuzione nel thread 0 (master thread) usando la primitiva `omp_get_wtime()` prima e dopo il calcolo parallelo, scegliendo il minimo tra 3 misurazioni ripetute. Infine, i controlli di robustezza del software sono stati interamente delegati al master thread.

3. Guida alla compilazione e manuale d'uso

Di seguito è illustrato il modo con cui è possibile invocare l'eseguibile allegato a questa relazione.

```
qsub libreria2.pbs -v M=[rows],N=[cols]
```

All'interno dello script "`libreria2.pbs`" è presente l'invocazione vera e propria dell'algoritmo (scritto in linguaggio C).

Passaggio dei parametri

M: indica il numero di righe della matrice che si desidera utilizzare. Tale numero deve essere necessariamente positivo.

N: indica il numero di colonne della matrice che si desidera utilizzare. Tale numero deve essere necessariamente positivo, inoltre, questo parametro corrisponde anche alla dimensione del vettore con il quale la matrice verrà moltiplicata.

In output sarà fornito un messaggio che stamperà la matrice, il vettore e il vettore risultato del prodotto scalare, inoltre viene riportato quanto tempo è stato impiegato per ottenere il risultato mostrato a schermo.

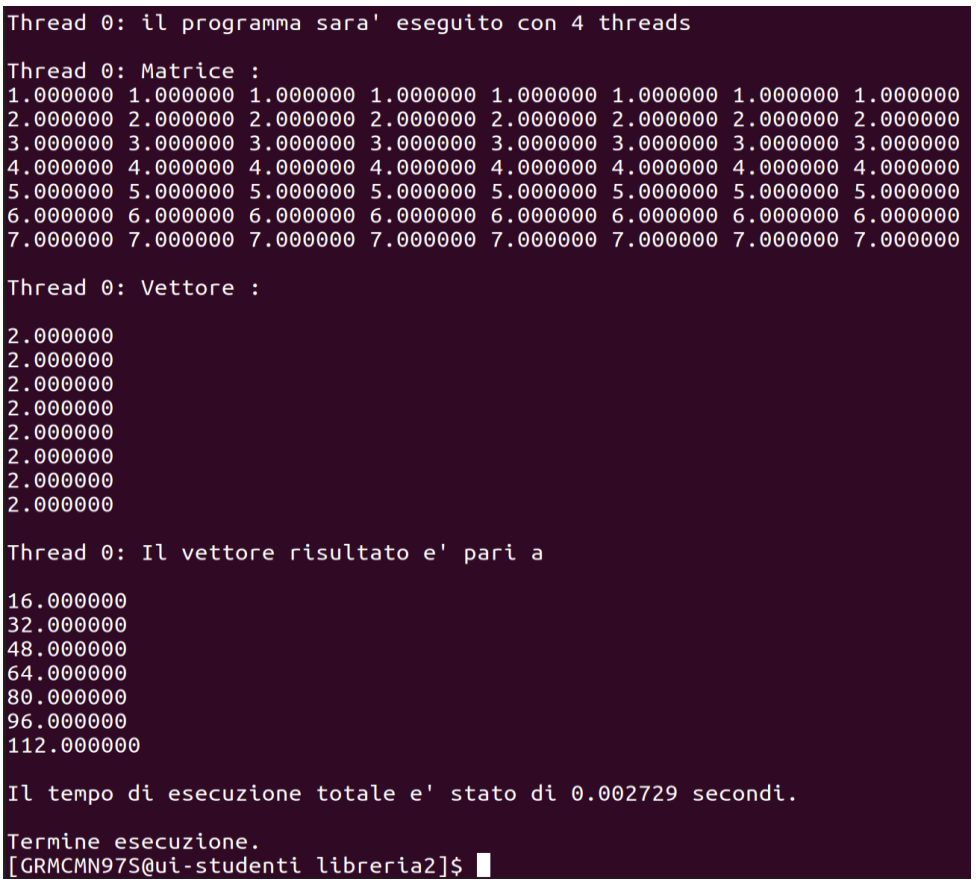
In allegato a questa documentazione è presente la cartella "Output", in cui sono contenuti alcuni file di output di esempio.

Input, output e condizioni di errore

- **Input**: la matrice e il vettore di cui effettuare il prodotto vettoriale e le dimensioni della matrice M, N; la dimensione del vettore corrisponde automaticamente a N.
- **Output**: il vettore risultato del prodotto vettoriale tra la matrice e il vettore.
- **Condizioni di errore**: la dimensione delle colonne della matrice deve essere uguale al numero di righe del vettore, inoltre gli input, così come il numero di thread, devono essere interi positivi.

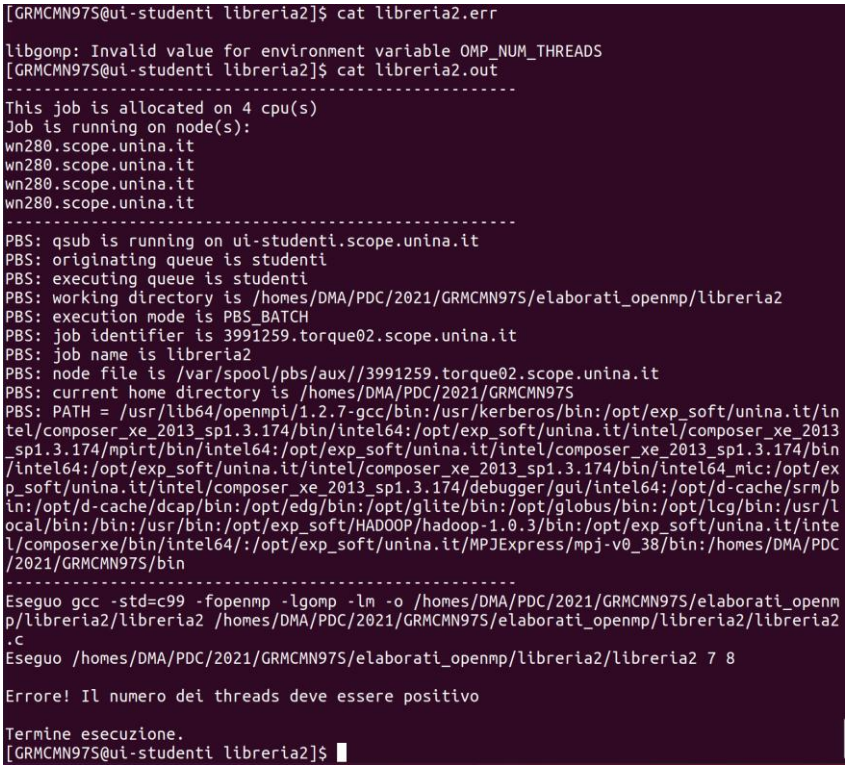
Esempio di funzionamento

Nell'immagine seguente vi è un esempio di funzionamento, con 4 thread, matrice di dimensione 7x8 e dimensione del vettore pari a 8.

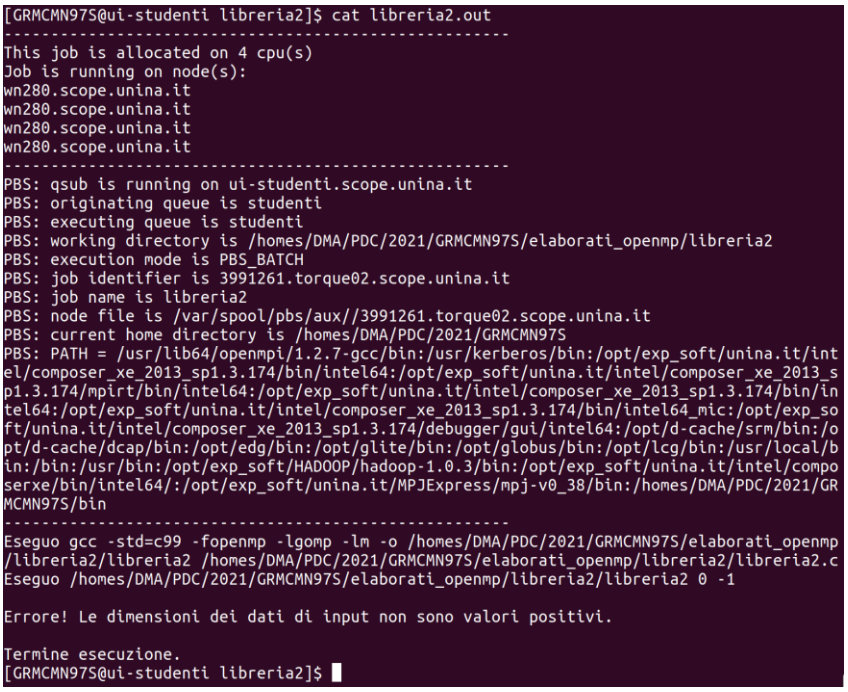


Esempi di errore

Nelle successivi immagini, invece, sono mostrati i messaggi di errore al verificarsi delle condizioni precedentemente citate.



Errore: numero di thread non positivo



Errore: i valori di input non sono positivi

Funzioni

void printVett(double *array, int dim):

Semplice funzione per la stampa dei valori di tipo double contenuti nell'array ricevuto in input, richiede il passaggio in input dell'array e della sua dimensione.

double * matxvet(int m, int n, double *time, double * restrict vettore, double **matrice):

Funzione che prende in input il numero di righe della matrice "m", il numero di colonne della matrice "n" (il quale corrisponde inoltre alla dimensione del vettore), un puntatore a una variabile che memorizza il tempo totale di esecuzione, un vettore e una matrice di tipo double, con i quali la funzione andrà a eseguire il prodotto.

Il vettore viene definito con la keyword "**restrict**", usandola si informa il compilatore che è possibile effettuare delle ottimizzazioni, in modo da migliorare le performance.

Tale funzione si occupa quindi di effettuare il prodotto matrice vettore, e lo fa utilizzando la libreria OpenMP. In output restituisce il vettore risultato del prodotto tra la matrice e il vettore.

4. Analisi delle prestazioni

Si è scelto di misurare i tempi di esecuzione usando la primitiva `omp_get_wtime()` prima e dopo il calcolo parallelo.

Si è scelto inoltre di considerare tre casi diversi che rappresentano le tre possibilità per le dimensioni della matrice: il caso in cui il numero di righe è maggiore numero di colonne, il caso in cui il numero di righe è minore del numero di colonne ed infine, il caso in cui la matrice è quadrata.

Per ciascuno di questi tre casi è stata considerata la media tra 6 esecuzioni ripetute, in particolare:

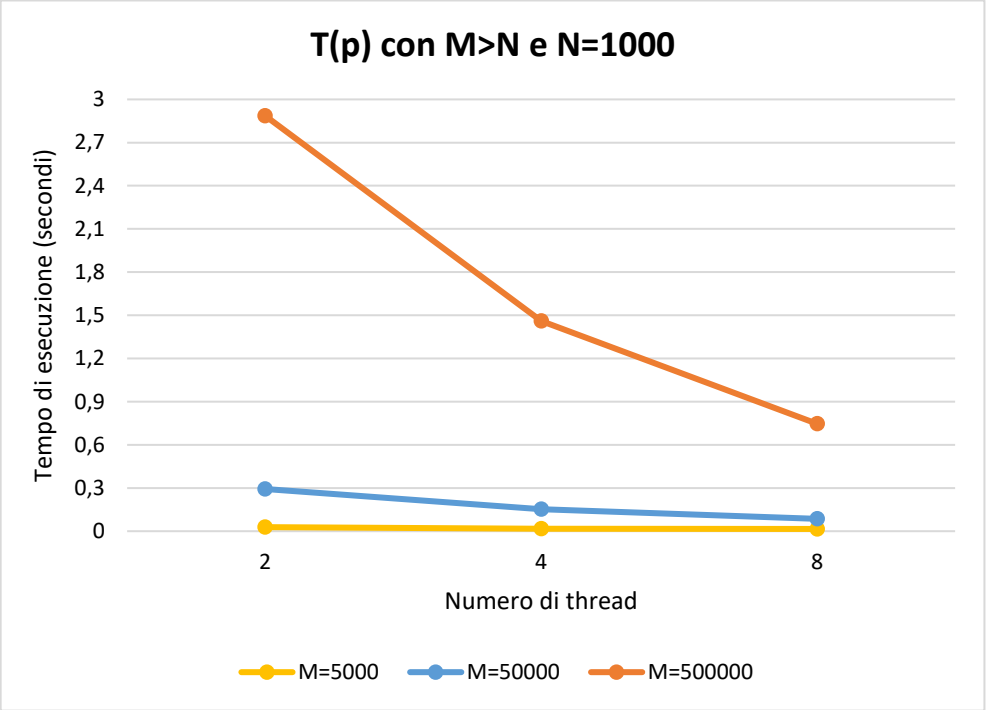
- quando il numero di righe M è maggiore del numero di colonne N **si fa variare M** da 5.000 a 500.000 con N fissato a 1.000;
- quando il numero di righe M è minore del numero di colonne N **si fa variare N** da 5.000 a 500.000 con M fissato a 1.000;
- quando la matrice è quadrata si **fa variare M=N** da 100 a 10.000.

Si noti che nel caso di matrici quadrate si è scelta una diversa configurazione rispetto al caso di matrici sbilanciate data l'impossibilità di eseguire sul cluster il programma con dimensioni 50000 x 50000. Pertanto, solo nel caso di matrici quadrate, è stata ridotta la dimensione massima. I grafici e le tabelle illustrati nelle pagine seguenti riassumono i risultati ottenuti per tutti i possibili valori di M, N e il numero di thread.

Tempi di esecuzione T(p)

Tabella dei tempi di esecuzione con M > N, varia M e N=1000:

Numero thread	Valore di M		
	5.000	50.000	500.000
2	0,028155	0,293232	2,886498
4	0,016936	0,151866	1,460349
8	0,015594	0,085883	0,746371



Per ulteriori considerazioni su questi risultati si rimanda alla sezione *"Conclusioni"* di questo capitolo.

Tabella dei tempi di esecuzione con M < N, varia N e M=1000:

Numero thread	Valore di N		
	5.000	50.000	500.000
2	0,029545	0,245962	2,452856
4	0,016722	0,128719	1,264680
8	0,014135	0,074042	0,956825

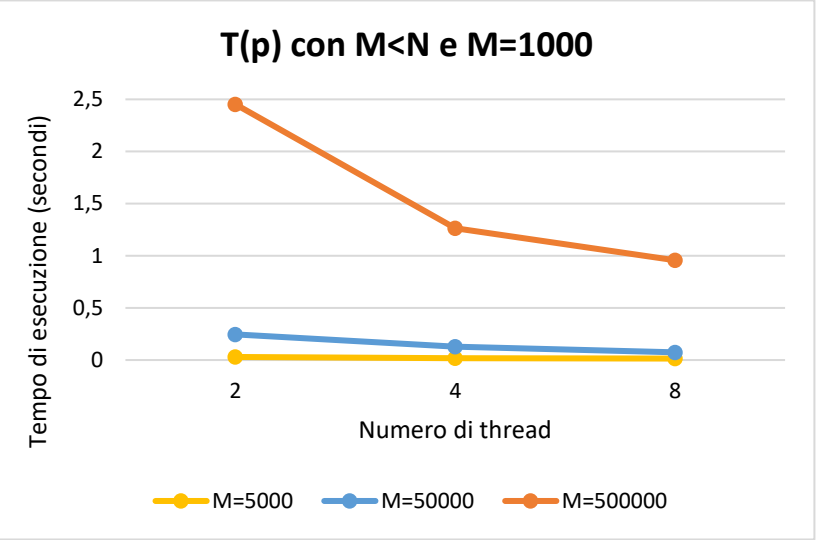
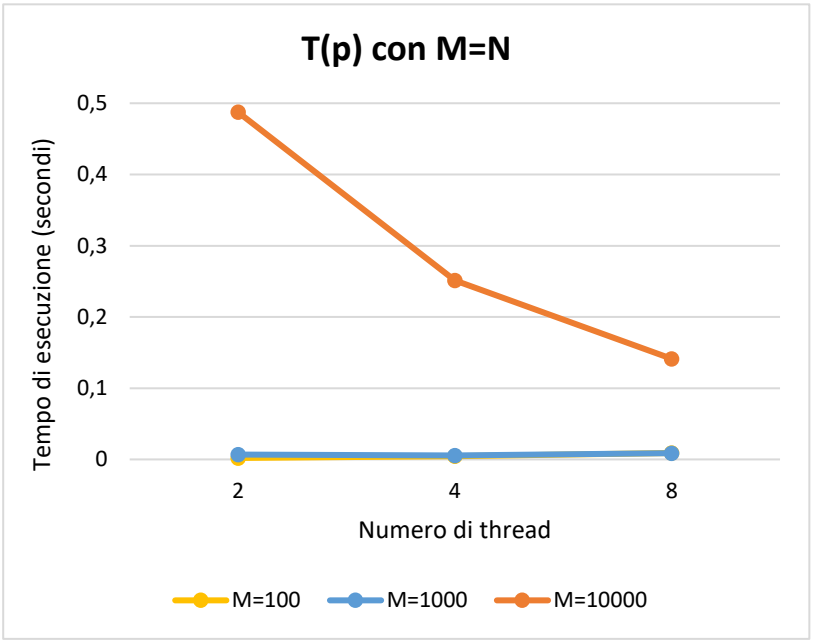


Tabella dei tempi di esecuzione con M = N, variano M e N:

Numero thread	Valore di M (= N)		
	100	1.000	10.000
2	0,002038	0,006612	0,487450
4	0,004655	0,005593	0,251115
8	0,009003	0,008781	0,141320



SpeedUp S(p)

Si è calcolato, inoltre, il tempo di riferimento T(1) che corrisponde al tempo di esecuzione su un processo single-thread. A partire dai tempi misurati nella sezione precedente e da T(1) è stato calcolato lo speed-up al variare di M, N e il numero di thread.

Tabella dei tempi di esecuzione T(1) con un processo single-thread:

Tipologia	Valore di M		
	5.000	50.000	500.000
M > N	0,047326	0,574243	5,737013
Tipologia	**Valore di N**		
	5.000	50.000	500.000
M < N	0,048077	0,479593	4,856553
Tipologia	**Valore di M (=N)**		
	100	1.000	10.000
M = N	0,000127	0,010994	0,956950

Tabella SpeedUp con M > N, varia M e N=1000:

Numero thread	Valore di M		
	5.000	50.000	500.000
2	1,680896	1,958319	1,987534
4	2,794321	3,781240	3,928521
8	3,034827	6,686329	7,686535

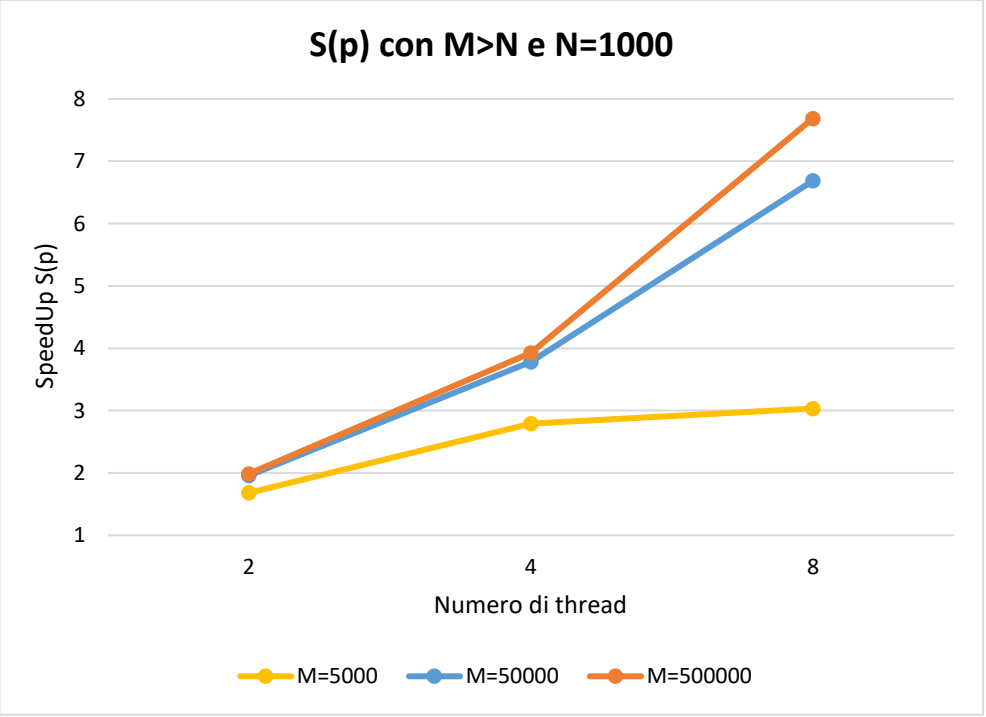


Tabella SpeedUp con M < N, varia N e M=1000:

Numero thread	Valore di N		
	5.000	50.000	500.000
2	1,627259	1,949867	1,979958
4	2,875082	3,725891	3,840142
8	3,401275	6,477244	5,075694

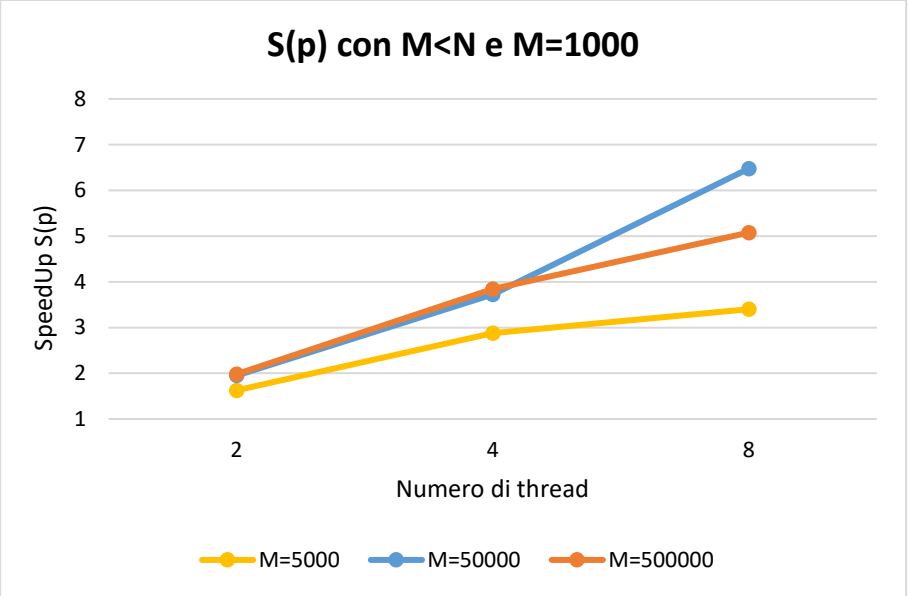
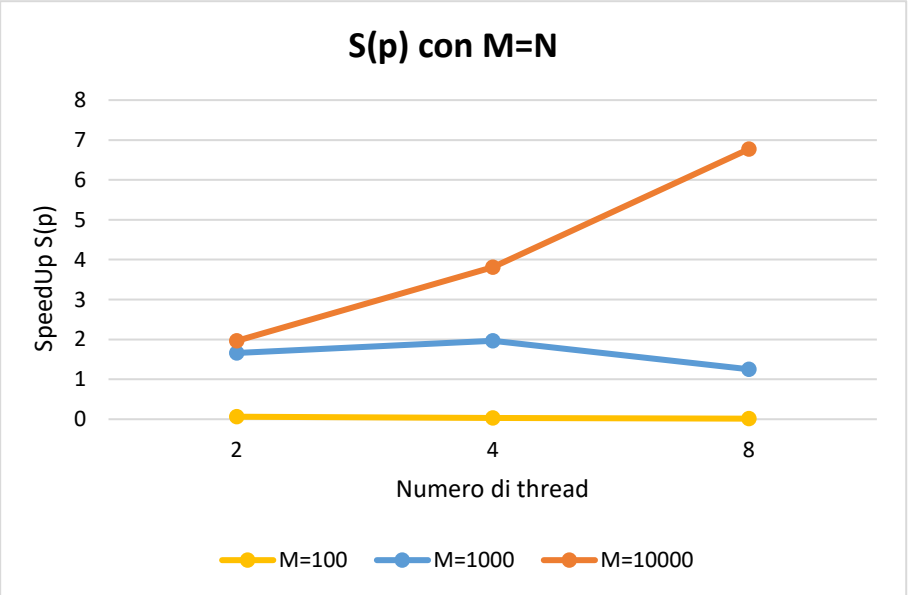


Tabella SpeedUp con M = N, variano M e N:

Numero thread	Valore di M (= N)		
	100	1.000	10.000
2	0,062402	1,662679	1,963175
4	0,027321	1,965514	3,810799
8	0,014128	1,252027	6,771503



Efficienza E(p)

Tabella efficienza con M > N, varia M e N=1000:

Numero thread	Valore di M		
	5.000	50.000	500.000
2	0,840448	0,979159	0,993767
4	0,698580	0,945310	0,982130
8	0,379353	0,835791	0,960817

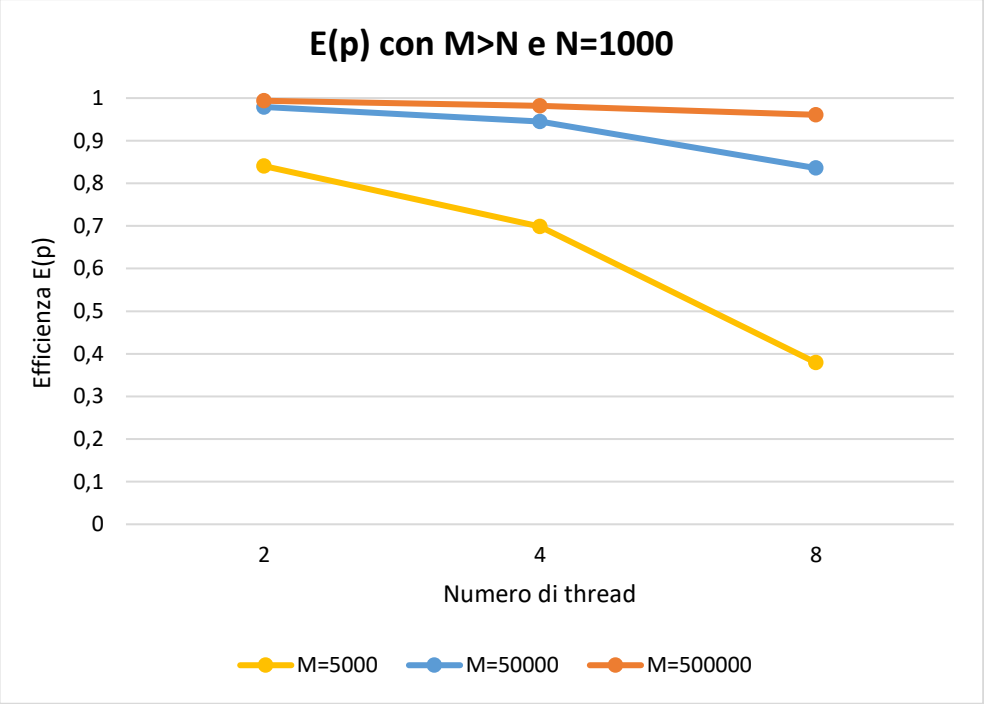


Tabella efficienza con M < N, varia N e M=1000:

Numero thread	Valore di N		
	5.000	50.000	500.000
2	0,813630	0,974933	0,989979
4	0,718771	0,931473	0,960035
8	0,425159	0,809656	0,634462

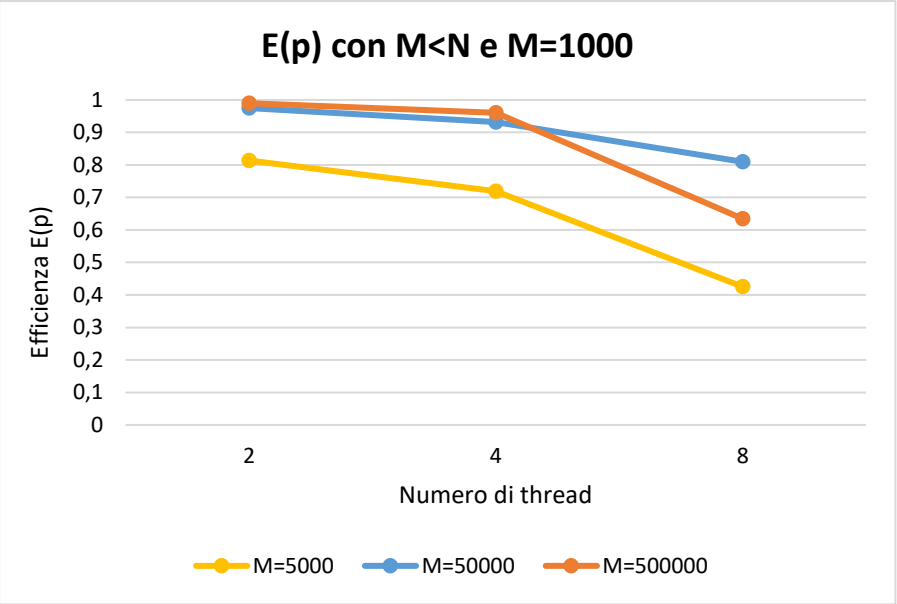
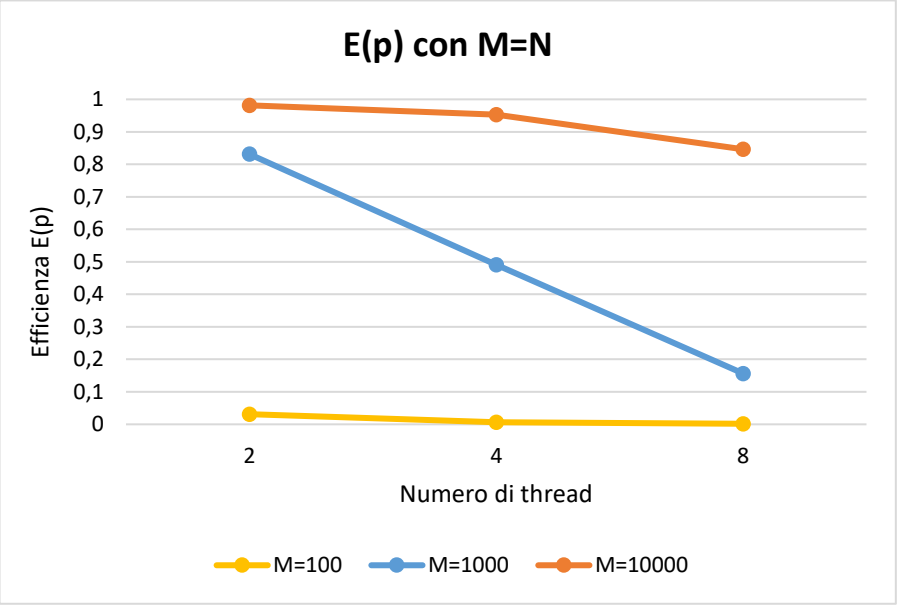


Tabella efficienza con M = N, variano M e N:

Numero thread	Valore di M (=N)		
	100	1.000	10.000
2	0,031201	0,831340	0,981587
4	0,006830	0,491379	0,952700
8	0,001766	0,156503	0,846438



Conclusioni

Dai grafici e dalle tabelle appena presentate si possono trarre le seguenti considerazioni:

- in tutte le configurazioni l'efficienza migliore si ha per 2 thread;
- l'efficienza e lo SpeedUp risultano complessivamente maggiori nella configurazione in cui M > N;
- l'efficienza degrada rapidamente all'aumentare del numero di thread, in particolare quando M = N; nel caso in cui M < N, l'efficienza peggiora con 8 thread quando aumentiamo la dimensione di N da 50.000 a 500.000.

Si possono fare ulteriori considerazioni notando che, per una dimensione fissata del problema, l'efficienza peggiora dopo un certo valore di p (ciò verifica sperimentalmente la legge di Amdahl), e che, in generale, all'aumentare sia della dimensione del problema che di p, l'efficienza migliora (verificando la legge di Gustafson).

Valgono analoghe considerazioni anche per i tempi e lo SpeedUp.

5. Codice sorgente

Il codice sorgente di seguito illustrato implementa l'algoritmo descritto nel capitolo 2, implementato attraverso l'ausilio del linguaggio di programmazione C.

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

/*************************Prototipi funzioni*************************/

void printVett(double *array, int dim);
double * matxvet(int m, int n, double * restrict vettore, double **matrice);

/********************************************************************/

int main(int argc, char *argv[]) {
    int i, j, M, N; //M: Righe matrice, N: Colonne matrice
    double **matrice;
    double *risultato, *vettore;
    double t1, t2;

    if(argc != 3) {
        printf("\nErrore! Occorre passare in input al PBS il numero"
        " di righe M e il numero di colonne N della matrice.\n\n");
        exit(1);
    }

    M = atoi(argv[1]);
    N = atoi(argv[2]);

    //Controlli di robustezza
    if (atoi(getenv("OMP_NUM_THREADS")) <= 0) {
        printf("\nErrore! Il numero dei threads deve essere positivo\n\n");
        exit(1);
    }
    if (M <= 0 || N <= 0) {
        printf("\nErrore! Le dimensioni dei dati di input non sono "
        "valori positivi.\n\n");
        exit(1);
    }

    printf("\n\nThread %d: il programma sara' eseguito con %d threads\n",
          omp_get_thread_num(), atoi(getenv("OMP_NUM_THREADS")));

    //Allocazione spazio per matrice e vettore
    matrice = (double**) malloc(M * sizeof(double*));
    for (i = 0; i < M; i++)
        matrice[i] = (double*) malloc(N * sizeof(double));

    vettore = (double*) malloc(N * sizeof(double));

    printf("\nThread %d: Matrice :\n", omp_get_thread_num());

    //Inizializzazione e stampa matrice
    for (i = 0; i < M; i++) {
        for (j = 0; j < N; j++) {
            matrice[i][j] = (double)i + 1.;
            printf("%lf ", matrice[i][j]);
        }

        printf("\n");
    }

    printf("\n");
```

```
62      //Inizializzazione vettore
63      for (i = 0; i < N; i++) {
64          vettore[i] = 2.;
65      }
66
67      printf("Thread %d: Vettore :\n", omp_get_thread_num());
68      printVett(vettore, N);
69      printf("\n");
70
71      //Memorizzo il tempo di inizio
72      t1 = omp_get_wtime();
73
74      risultato = matxvet(M, N, vettore, matrice);
75
76      //Memorizzo il tempo di fine
77      t2 = omp_get_wtime();
78
79      //Stampa del risultato
80      printf("Thread %d: Il vettore risultato"
81          " e' pari a \n", omp_get_thread_num());
82      printVett(risultato, M);
83
84      printf("\nIl tempo di esecuzione totale e' "
85          "stato di %lf secondi.\n\n", (t2 - t1));
86
87      /* Libero la memoria */
88
89      for (i = 0; i < M; i++)
90          free(matrice[i]);
91
92      free(matrice);
93      free(vettore);
94      free(risultato);
95
96      return 0;
97  }
98
99  /*****************************************************************************/
100
101 void printVett(double *array, int dim) {
102     int i;
103
104     for (i = 0; i < dim; i++) {
105         printf("\n%lf ", array[i]);
106     }
107     printf("\n");
108 }
109
110 double * matxvet(int m, int n, double * restrict vettore, double **matrice) {
111     int i, j;
112     double *risultato;
113
114     //Allocazione spazio per vettore risultato
115     risultato = (double*) malloc(m * sizeof(double));
116
117     //Calcolo in parallelo del prodotto tra matrice e vettore
118     #pragma omp parallel for private(i,j) shared (matrice, vettore, risultato)
119     for (i = 0; i < m; i++) {
120         for (j = 0; j < n; j++) {
121             risultato[i] += matrice[i][j] * vettore[j];
122         }
123     }
124
125     return risultato;
126 }
```