
Università degli Studi di Napoli "Parthenope"

# Relazione Calcolo Parallelo e Distribuito

*a.a 2021/2022*

**Proponenti**

Gianfranco Terrazzano - *0124002052*

Giuseppe Orlando - *0124002053*

Ilaria Scuotto - *0124002123*

# Sommario

# Definizione ed Analisi del problema

Si vuole realizzare un software che effettui **l'implementazione di un programma parallelo per l'ambiente multicore con np unità processanti che impieghi la libreria OpenMP. Il programma deve essere organizzato come segue: il core master deve generare una matrice A di dimensione N×M. Quindi, ogni core deve estrarre M/p colonne ed effettuare localmente la somma degli elementi delle sottomatrici estratte, conservando il risultato in un vettore b di dimensione M.**

Dunque, le operazioni da fare sono, la creazione di una matrice di dimensioni N×M, con N e M scelti dall'utente, successivamente visualizzare questa matrice e suddividerla in sottomatrici, i cui elementi saranno sommati dunque la loro somma andrà ad occupare una posizione all'interno di un vettore di dimensione M.

Per comprendere meglio il problema, è utile analizzare il comportamento dell'algoritmo sequenziale, ovvero la sua esecuzione con 1 thread. Dopo la dichiarazione delle variabili, la funzione controllo_threads diventa superflua, dato che è una funzione utile all'approccio parallelo per definire la clausola num_threads. Si procede quindi con l'allocazione dinamica della matrice e del vettore e vengono richiamate le funzioni crea_matrice e stampa_matrice. La definizione di nloc anche qui sembra essere superflua in quanto nloc=M/p, ove p=1 perché vi è solo un thread in esecuzione. Anche l'implementazione della perfetta divisibilità perde valore, in quanto abbiamo un solo thread. Il risultato, dunque, è che non si creerà alcuna sottomatrice, le variabili start ed end, assumeranno gli indici di inizio e fine della matrice. Dunque, l'algoritmo procederà a fare la somma di tutti gli elementi della matrice NxM ed essa verrà messa nella posizione 0 dell'array b[M].
L'operazione principale dell'algoritmo sono le somme della matrice. La complessità è data dalla presenza di due cicli for innestati ed è $T_{1(NM)}$=NM-1.

# Descrizione dell'approccio parallelo

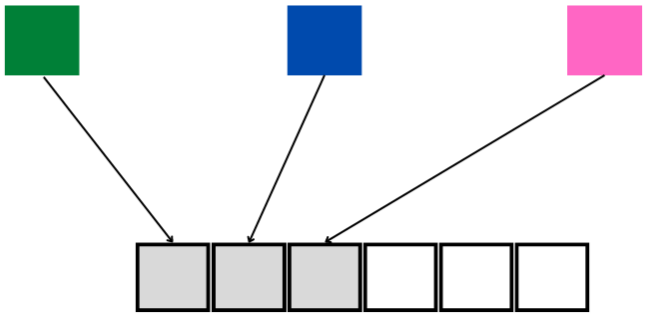
La somma di ogni elemento di una sottomatrice della matrice di partenza è indipendente dalle somme degli altri elementi appartenenti ad un'altra sottomatrice.
Questo problema si presta all'esecuzione in parallelo, assegnando ad ogni unità processante un numero M/p colonne, la quale, in parallelo, svolgerà le operazioni di somma degli elementi della sottomatrice di dimensione N×(M/p), questa somma sarà una somma locale e verrà inserita all'interno di un vettore b[M], non vi sarà la necessità di collezionare queste somme locali. Non essendovi questo passaggio non saranno presenti parti da svolgere in sequenziale, giovandone per quanto riguarda le prestazioni e l'efficienza del nostro algoritmo.

Ipotizziamo di avere N=5, M=6 e p=3, ogni unità processante andrà a lavorare in una sottomatrice di dimensioni 5×(6/P) → 5×2, ogni unità processante è indicata con un colore diverso: **verde**, **blu** e **rosa**.

Dello stesso colore è indicata la sottomatrice di riferimento, le somme locali andranno poi inserite in un vettore di dimensione M = 6, le posizioni occupate però saranno minori rispetto alla sua dimensione, successivamente verrà richiamata la funzione che stamperà il vettore risultante.

[1][3][5][7][8][7]
[6][7][5][5][4][2]
[1][4][4][5][1][2]
[3][5][4][1][3][2]
[1][4][4][5][1][2]

[1][3][5][7][8][7]
[6][7][5][5][4][2]
[1][4][4][5][1][2]
[3][5][4][1][3][2]
[1][4][4][5][1][2]



Considerando le metriche standard valutiamo l'efficienza del nostro approccio parallelo, valutando parametri come SpeedUp, Overhead, Efficienza, SpeedUp con Ware-Amdahl e Isoefficienza.

## SpeedUp

Lo SpeedUp misura la riduzione del tempo di esecuzione rispetto all'algoritmo su un processore ed esso viene calcolato come il rapporto $T_1$ su $T_p$

$$Sp = \frac{T_1}{T_p} = \frac{NM - 1}{N\left(\frac{M}{p}\right) - 1} = p \rightarrow speed\ up\ ideale$$

Il nostro algoritmo al crescere della dimensione del problema migliora le sue prestazioni (vedere tabella tempi di esecuzione), per tanto consideriamo che lo speed up trattando delle complessità a grandi dimensioni, assorba la costante -1, alla fine dei conti avremo p, ovvero, lo speed up ideale.

## Overhead

L'Overhead totale misura quanto lo speed up differisce da quello ideale ed esso viene calcolato nel seguente modo:

$$O_h = \left(pT_p - T_1\right) t_{calc}$$

$$T_1 = NM - 1 \text{ e } T_p = N\left(\frac{M}{p}\right) - 1$$

Per ragionamento analogo a quello dello SpeedUp fatto in precedenza, le costanti non vengono considerate, dunque il nostro overhead è zero.

$$O_h = \left(pN\left(\frac{M}{p}\right) - NM - 1\right) t_{calc} \rightarrow O_h = 0$$

## Efficienza

L'Efficienza misura quanto l'algoritmo sfrutta il parallelismo del calcolatore e si calcola nel seguente modo:

$$E_p = \frac{S_p}{p} = \frac{p}{p} = 1$$

Poiché noi abbiamo uno SpeedUp ideale l'efficienza ci verrà pari a 1, coincidendo così con l'efficienza ideale.

### Speed-up con la caratterizzazione di Ware-Amdahl

La legge di Ware consente di predire l'aumento dello speed up nel caso in cui aumentino il numero di processori e/o la dimensione del problema.

In generale per **MIMD-SM**

Il tempo di esecuzione di un algoritmo parallelo, distribuito su p processori, comprende 2 componenti:

- Ts tempo per eseguire la parte seriale
- Tc /p tempo per eseguire la parte parallela

Ove $\boldsymbol{Ts} = \boldsymbol{a}$ e $\boldsymbol{Tc} = \mathbf{1} - \boldsymbol{a}$, quindi $\mathbf{Tp} = \boldsymbol{Ts} + \frac{\boldsymbol{Tc}}{\boldsymbol{p}}$

Alla luce di queste considerazioni definiamo la legge di Ware come:

$$Sp = \frac{1}{a + \frac{1-a}{p}}$$

Per quanto riguarda il nostro algoritmo $\boldsymbol{a} = \mathbf{0}$ per il "semplice" motivo che non viene effettuata alcuna operazione in sequenziale, dato che la traccia non richiede alcuna collezione dei dati , ma solo l'inserimento delle somme in un vettore. Quindi:

$$S_p = \frac{1}{\alpha + \frac{(1-\alpha)}{p}} \xrightarrow{\alpha \to 0} p$$

$\alpha \to 0$, $p \to 0$

$S_p \to p$

$(S^{ideale}(p) = p)$

Ipotizzando di avere $\boldsymbol{p = 8}$:

$$Sp = \frac{1}{0 + \frac{1-0}{8}} = 8 \rightarrow Speed\ up\ ideale$$

### Isoefficienza

L'Isoefficienza è la funzione I che esprime la legge secondo cui si deve scegliere la nuova dimensione n1 affinché l'efficienza resti costante.

Un algoritmo si dice scalabile se l'efficienza rimane costante al crescere del numero dei processori e della dimensione del problema, in un rapporto costante pari a:

$$I = \frac{O_h(n_1,p_1)}{O_h(n_0,p_0)} = \frac{0}{0} = Forma\ Indeterminata$$

Per convenzione l'isoefficienza viene posta uguale ad infinito, ovvero posso usare qualunque costante moltiplicativa per calcolare n1 e quindi controllare la scalabilità dell'algoritmo.

# Descrizione dell'algoritmo parallelo

L'algoritmo parallelo è stato implementato in questo modo**:**

- **Definizione dei prototipi delle funzioni**: abbiamo definito quattro funzioni, ovvero: crea_matrice, stampa_matrice, vettore_ris e controllo_threads.

- **Dichiarazione delle variabili:** Definiamo diverse variabili.
  - **Strutture dati:** Per implementare l'algoritmo abbiamo avuto bisogno di una matrice **A** e un vettore **b** entrambi integer.
  - **Contatori:** Per scorrere gli elementi di una matrice utilizziamo due contatori **i** e **j.**

- **Dimensione del problema:** Definiamo **N** righe e **M** colonne che sono le dimensioni del problema, da cui ricaviamo la dimensione del sotto problema **nloc**, e dell'array risultante, **b[M]** come stabilito dalla traccia.
- **Altre variabili:**
  - **p:** numero di threads nella sezione parallela.
  - **sum**: Somma locale fatta da ogni thread.
  - **r**: Resto della divisone M/p.
  - **id**: Id del thread in esecuzione.
  - **step**: utilizzata per la perfetta divisibilità.
  - **start**: colonna di partenza gestita da un thread.
  - **end**: ultima colonna gestita da un thread.
  - **tmp**: variabile per stabilire il numero corretto di thread da utilizzare.
  - **max**: numero massimo di thread utilizzabili.

## Spiegazione del main:

All'inizio del codice incontriamo **srand**(time(NULL)), essa è una funzione che ci permette di inizializzare a valori diversi, per ogni esecuzione il seme della **rand** (È stato necessario includere la libreria time.h per poterla utilizzare).

Da terminale prendiamo le dimensioni della nostra matrice, ovvero **N** e **M**, avvalendoci di **printf e scanf**.
Assegniamo alla variabile **max** il massimo numero di thread a disposizione tramite la funzione **omp_get_max_threads()** (Per utilizzare le funzioni riguardanti OMP abbiamo incluso la libreria omp.h), questa assegnazione è utile per la funzione **controllo_threads** alla quale passiamo **max e M,** il valore di ritorno di quest'ultima viene salvato in **tmp** che tramite la clausola **num_threads**, del costrutto omp parallel, ci permetterà di utilizzare il giusto numero di threads.

Vengono allocate dinamicamente la matrice **A** e il vettore **b** utilizzando la **malloc**, dopodiché richiamiamo le funzioni **crea_matrice** e **stampa_matrice**. A questo punto del codice entriamo nel costrutto **#pragma omp parallel** e definiamo le **clausole num_threads**(tmp) **private**(i, j, id, start, end, step, sum) **shared**(A, b, r). Questo costrutto ad alto livello ci permetterà di gestire i threads parallelamente senza troppe difficoltà.

Quindi definiamo **p=omp_get_num_threads()**, ovvero il numero di threads attivi nel costrutto parallel, definiti da **num_threads.**
Assegniamo la dimensione al sotto problema, ovvero il numero di colonne delle sottomatrici da sommare **nloc=M/p**. Fatto ciò prima di procedere, è importante implementare la **perfetta divisibilità** del problema, ovvero far funzione l'algoritmo anche quando i threads non sono divisibili per la dimensione del problema.

```
//Implementazione perfetta divisibilità
r=M%p;
id=omp_get_thread_num();
if (id < r)
{
        nloc++;
        step=0;
}
else
        step=r;
```

Quindi confrontiamo il resto della visione M/p con l'id del processo, se l'id è minore allora incrementeremo **nloc** e daremo **step**=0, altrimenti avremo **step**=r.

Una volta fatto ciò procediamo al cuore del codice; ricaviamo le posizioni di inizio e fine della sottomatrice **start=nloc*id+step** , **end=nloc*id+step+nloc-1,** successivamente tramite due cicli for innestati sommiamo gli elementi della sottomatrice e li mettiamo nella somma locale **sum**, una volta fatto ciò possiamo inserire la somma locale nella giusta posizione dell'array b ottenuta facendo **b[nloc*id]=sum**

```
//Dichiaro inizio e fine sottomatrice
start=nloc*id+step;
end=nloc*id+step+nloc-1;

//Effettuo le somme locali per ogni thread
for(i=0; i<N; i++){
        for(j=start; j<=end; j++){
                sum=A[i][j]+sum;
        }
}
```

```
b[nloc*id]=sum;
```

Dopo la fine del codice parallelo stampiamo il vettore risultante tramite la funzione **vettore_ris** e chiudiamo il main con **return** 0;

- **Implementazione delle funzioni:** Vengono implementate quattro funzioni
  - **Crea_matrice:** Gli passiamo **M,N** e **A**. Nasce allo scopo di riempire la matrice con numeri pseudocasuali tramite la funzione **rand**.
  - **Stampa_matrice**: Gli passiamo **M,N** e **A**. Classica funzione per la stampa di una matrice.
  - **Vettore_ris:** Gli passiamo **M** e **b**. Stampa il vettore risultante.
  - **Controllo_threads**: Gli passiamo **max** e **M**. Nasce allo scopo di assegnare un giusto numero di thread, un eccesso di thread sarebbe inutile in quanto al più un thread può gestire una colonna.

# Input e Output

Inizialmente il nostro algoritmo prende come input l'intero N e l'intero M che andranno ad indicare le righe e le colonne della nostra matrice.

Successivamente l'algoritmo prevede l'allocazione della matrice A in modo dinamico, mediante l'utilizzo della funzione malloc; tale matrice verrà poi riempita in modo randomico dalla funzione rand.

Effettuiamo il medesimo processo anche per allocare il vettore risultante b, il quale ospiterà le somme degli elementi delle sottomatrici effettuate dai thread, questi ultimi impostati con il comando export OMP_NUM_THREADS, qualora non vengano impostati il programma partirà con i threads di default della macchina.

Una volta terminato quanto segue, e dopo aver calcolato le somme locali per ogni thread, l'algoritmo stampa a video come output, la matrice di partenza, le singole somme delle sottomatrici, specificando anche i thread che gestiscono le sottomatrici stesse e alla fine il vettore b risultante di dimensione M.

# Routine implementate

Il software è composto da diverse routine, alcune implementate da noi programmatori, altre appartenenti alla libreria OpenMP.
Le routine implementate dai programmatori sono:

- crea_matrice(int, int, int **)
- stampa_matrice(int, int, int **)
- vettore_ris(int, int *)
- controllo_threads(int, int)

Rispettivamente svolgono la funzione di "**creazione di una matrice**", ovvero la matrice viene riempita di numeri casuali da 1 a 10, svolgono la funzione di "**stampa matrice**", due indici scorrono le righe e le colonne e visualizzano a video il contenuto della matrice, inoltre abbiamo la routine "**vettore risultante**" che stampa a video il contenuto del vettore risultante contenente le varie somme degli elementi della sottomatrice, infine abbiamo la routine controllo_threads la quale controlla che le unità processanti impostate non siano maggiori del numero di colonne nella nostra matrice, in tal caso il numero massimo di thread utilizzati sarà al più il numero di colonne altrimenti resterà invariato.

Le routine usate appartenenti alla libreria time.h :

- srand(time(NULL))

La funzione **srand()** serve a inizializzare la funzione per la generazione dei numeri casuali: senza di essa allo stesso seed (seme) il programma estrarrebbe sempre gli stessi numeri casuali.

Le routine usate appratenti alle librerie standard sono:

- rand()% numero
- malloc(size)
- printf
- scanf

Quando si invoca la funzione **rand**() questa restituisce un valore casuale (o random) compreso in un certo intervallo. Se questo intervallo non è definito, si usa un intervallo di default che va da 0 a 32766 , per cui **rand**() restituisce in modo casuale numeri interi che vanno da 0 a 32766.

La funzione **malloc** serve a chiedere una nuova zona di memoria da usare. Quello che succede quando viene chiamata la funzione è che il calcolatore individua una zona di memoria libera, ossia una zona che non è attualmente occupata da variabili o usata in altro modo, e ne restituisce l'indirizzo iniziale.

Sono state utilizzate anche printf e scanf, rispettivamente **printf** è una funzione per visualizzare sullo standard output una stringa costruita in base ad un formato specificato, mentre **scanf** ha lo scopo di rilevare quanto immesso dall'utente tramite la tastiera, convertirlo in un numero intero e memorizzarne il risultato in una variabile.

Le routine usate appartenenti alle libreria omp.h:

- omp_get_num_threads()
- omp_get_thread_num()
- omp_get_wtime()
- omp_get_max_threads()

La routine **omp_get_num_threads()** restituisce il numero di thread attualmente presenti nel team che esegue l'area parallela da cui viene chiamata.
La **omp_get_thread_num()** funzione restituisce il numero del thread, all'interno del team, del thread che esegue la funzione. Il numero di thread è compreso tra 0 e omp_get_num_threads()-1 inclusi. Il thread master del team è il thread 0.
La routine **omp_get_wtime()** restituisce un valore in secondi del tempo trascorso da un certo punto.
La routine **omp_get_max_threads()** restituisce un intero uguale o maggiore del numero di thread che sarebbero disponibili se un'area parallela senza num_threads fosse stata definita in quel punto del codice.

Le direttive utilizzate per implementare le sezioni parallele sono:

- #pragma omp parallel

La direttiva **omp parallel** indica esplicitamente al compilatore di parallelizzare il blocco di codice scelto.

# Analisi delle performance del Software

Riportiamo i tempi di esecuzione del nostro algoritmo, in delle tabelle. Sono stati riportati variando il numero di processori impiegati e le dimensioni dell'input. Sotto riportati troviamo anche grafici per i tempi, speed-up ed efficienza.
La macchina utilizzata per le esecuzioni possiede le seguenti caratteristiche:

- Processore: Intel core i7 9th gen
- RAM: 16 GB
- Scheda video: Nvidia GTX 1060
- Memoria: 256 GB SSD, 1TB HDD
- 6 core – 12 Threads

## Tempi di esecuzione

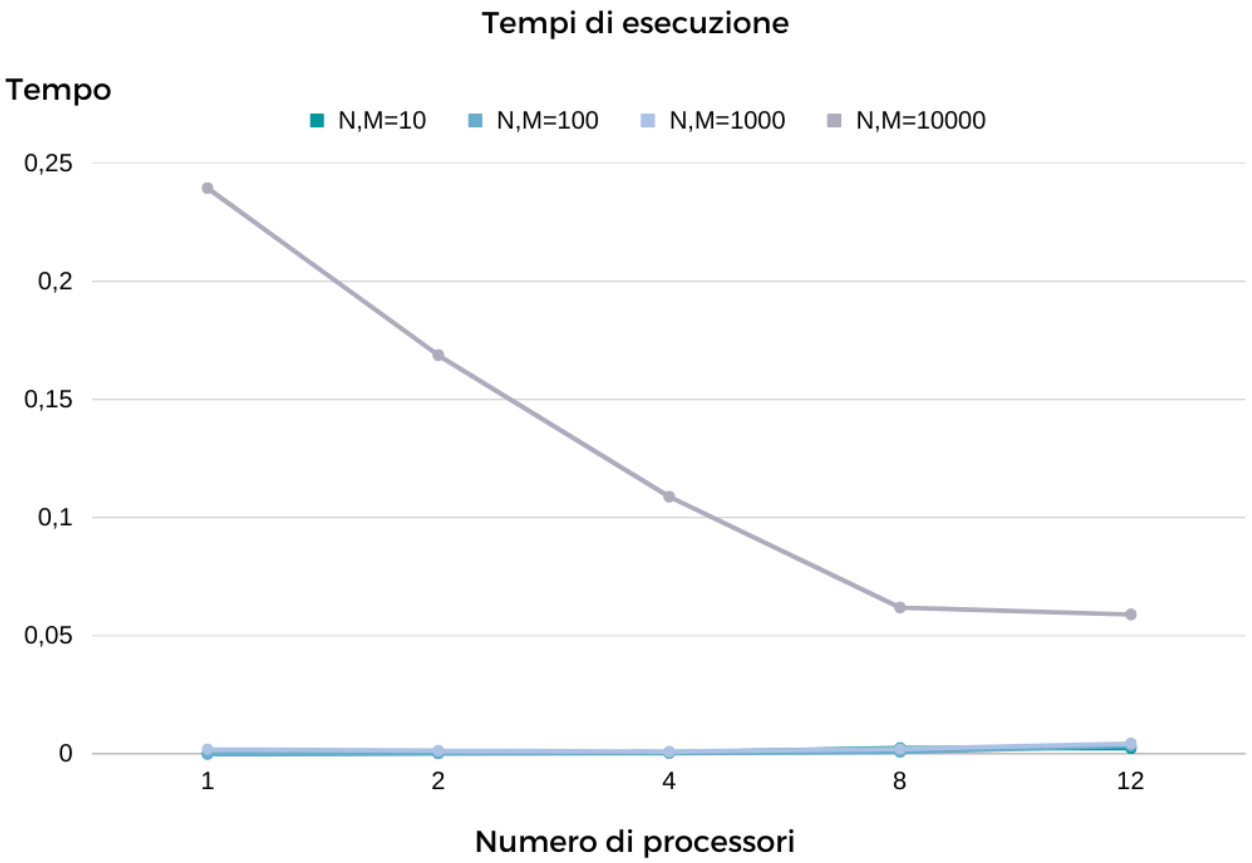
| N° PROC | N=10 M=10 | N=100 M=100 | N=1000 M=1000 | N=10000 M=10000 | N=100000 M=100000 |
|---|---|---|---|---|---|
| **1** | 0.000055 | 0.000067 | 0.002082 | 0.239490 | Killed |
| **2** | 0.000551 | 0.000315 | 0.001525 | 0.168759 | Killed |
| **4** | 0.000692 | 0.000410 | 0.001114 | 0.108941 | Killed |
| **8** | 0.002587 | 0.000977 | 0.002193 | 0.062001 | Killed |
| **12** | 0.002573 | 0.003590 | 0.004492 | 0.059085 | Killed |

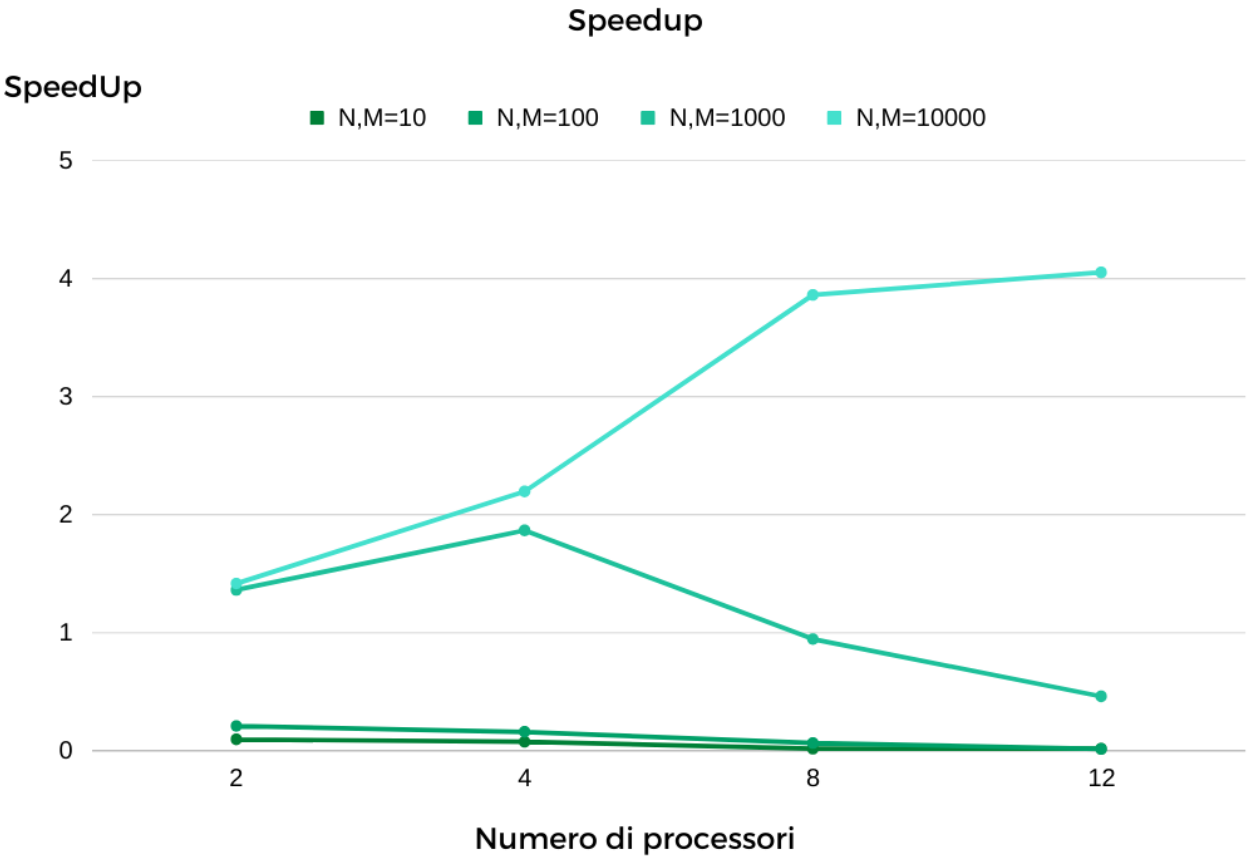
## SpeedUp

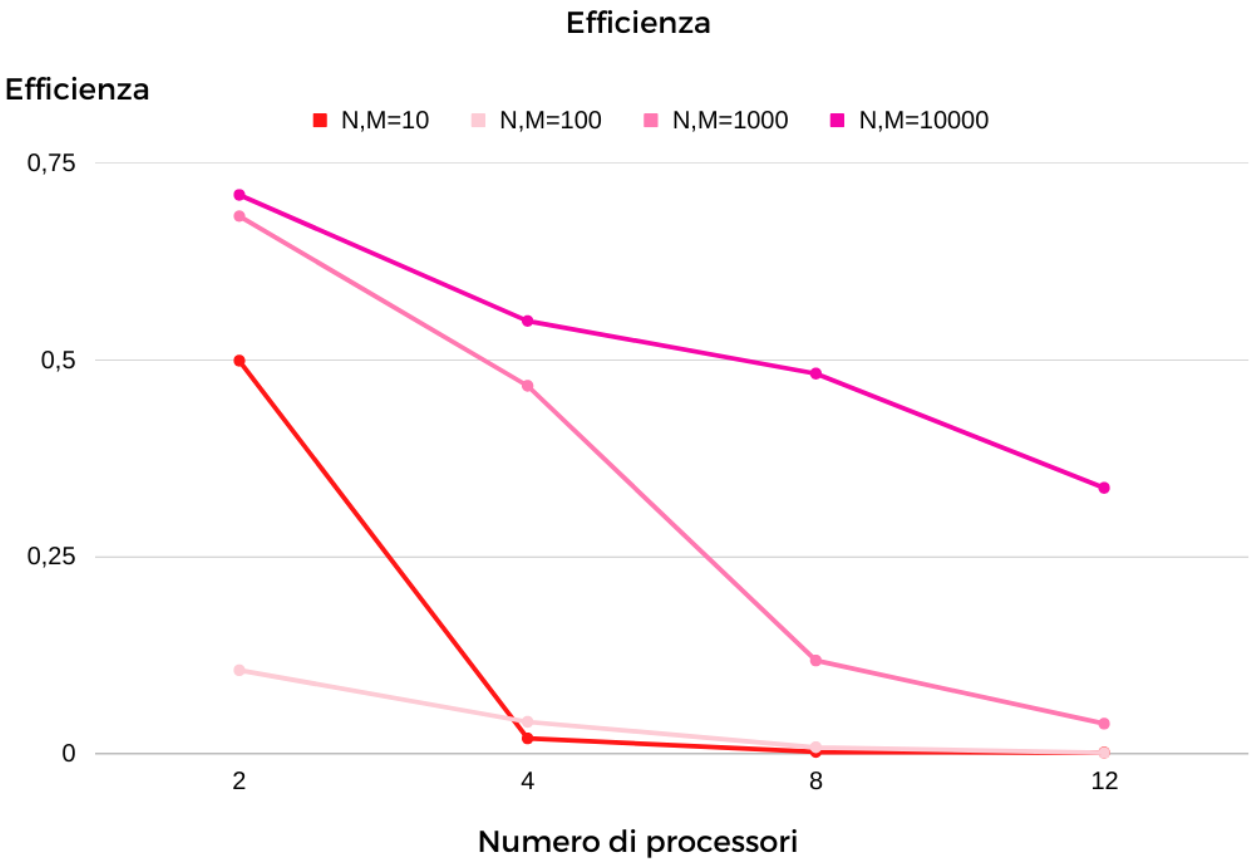
| N° PROC | N=10 M=10 | N=100 M=100 | N=1000 M=1000 | N=10000 M=10000 | N=100000 M=100000 |
|---|---|---|---|---|---|
| **2** | 0.099818 | 0.212698 | 1.365245 | 1.419124 | Killed |
| **4** | 0.079479 | 0.163414 | 1.868940 | 2.198345 | Killed |
| **8** | 0.021260 | 0.068577 | 0.949384 | 3.862679 | Killed |
| **12** | 0.021375 | 0.0186629 | 0.463490 | 4.053313 | Killed |

## Efficienza

| N° PROC | N=10 M=10 | N=100 M=100 | N=1000 M=1000 | N=10000 M=10000 | N=100000 M=100000 |
|---|---|---|---|---|---|
| **2** | 0.49909 | 0.106349 | 0.6826225 | 0.709562 | Killed |
| **4** | 0.01986975 | 0.0408535 | 0.467235 | 0. 54958625 | Killed |
| **8** | 0.0026575 | 0.008572125 | 0. 118673 | 0. 48283487 | Killed |
| **12** | 0.0017812 | 0.0015552417 | 0.03862416 | 0.33777608 | Killed |


Tempi di esecuzione
Tempo
N,M=10
N,M=100
N,M=1000
N,M=10000
0,25
0,2
0,15
0,1
0,05
0
1
2
4
8
12
Numero di processori


Speedup
SpeedUp
N,M=10
N,M=100
N,M=1000
N,M=10000
5
4
3
2
1
0
2
4
8
12
Numero di processori


Efficienza
Efficienza
N,M=10
N,M=100
N,M=1000
N,M=10000
0,75
0,5
0,25
0
2
4
8
12
Numero di processori

# Esempi d'uso

Riportiamo di seguito degli esempi d'uso del nostro algoritmo.
Il primo ha una matrice di N=9 e M=6, gestita con sei threads, dunque, ogni thread si occuperà di una sottomatrice composta da nove righe e una sola colonna.

```
Inserisci numero di righe
9
Inserisci numero di colonne
6
10  9  4  9  9  3
5  4  10  4  2  4
7  3  1  7  10  9
4  7  4  10  8  6
7  2  9  3  4  8
4  5  6  9  3  7
3  9  10  3  2  3
6  8  6  9  7  7
9  10  3  2  2  3
La sottomatrice che va da colonna: 0 a colonna: 0 è gestita dal thread: 0
 La somma degli elementi della sottomatrice e': 61
La sottomatrice che va da colonna: 5 a colonna: 5 è gestita dal thread: 5
 La somma degli elementi della sottomatrice e': 50
La sottomatrice che va da colonna: 2 a colonna: 2 è gestita dal thread: 2
 La somma degli elementi della sottomatrice e': 53
La sottomatrice che va da colonna: 1 a colonna: 1 è gestita dal thread: 1
 La somma degli elementi della sottomatrice e': 57
La sottomatrice che va da colonna: 4 a colonna: 4 è gestita dal thread: 4
 La somma degli elementi della sottomatrice e': 47
La sottomatrice che va da colonna: 3 a colonna: 3 è gestita dal thread: 3
 La somma degli elementi della sottomatrice e': 56
Il vettore alla posizione 0 risultante e': 61
Il vettore alla posizione 1 risultante e': 57
Il vettore alla posizione 2 risultante e': 53
Il vettore alla posizione 3 risultante e': 56
Il vettore alla posizione 4 risultante e': 47
Il vettore alla posizione 5 risultante e': 50
```

Il secondo ha una matrice di N=4 e M=7, gestita con cinque threads, dunque, vi saranno due threads che prenderanno due colonne e i restanti tre threads ne gestiranno una sola. Le somme riportate saranno chiaramente cinque, non essendovi una perfetta divisibilità, dove ogni thread gestisce una colonna, il vettore b di dimensione M, ricordiamo essere uguale a 7, avrà due posizioni vuote in quanto il numero di somme è minore al numero di colonne.

```
Inserisci numero di righe
4
Inserisci numero di colonne
7
9  8  2  5  2  5  10
2  3  2  6  2  1  8
1  8  10  1  4  6  9
5  10  6  3  6  4  2
La sottomatrice che va da colonna: 2 a colonna: 3 è gestita dal thread: 0
 La somma degli elementi della sottomatrice e': 40
La sottomatrice che va da colonna: 4 a colonna: 5 è gestita dal thread: 1
 La somma degli elementi della sottomatrice e': 30
La sottomatrice che va da colonna: 5 a colonna: 5 è gestita dal thread: 3
 La somma degli elementi della sottomatrice e': 16
La sottomatrice che va da colonna: 6 a colonna: 6 è gestita dal thread: 4
 La somma degli elementi della sottomatrice e': 29
La sottomatrice che va da colonna: 4 a colonna: 4 è gestita dal thread: 2
 La somma degli elementi della sottomatrice e': 14
Il vettore alla posizione 0 risultante e': 40
Il vettore alla posizione 1 risultante e': 30
Il vettore alla posizione 2 risultante e': 14
Il vettore alla posizione 3 risultante e': 16
Il vettore alla posizione 4 risultante e': 29
Il vettore alla posizione 5 risultante e': 0
Il vettore alla posizione 6 risultante e': 0
```

# Riferimenti bibliografici

Il materiale utilizzato per la realizzazione della seguente relazione, e stesura del codice è il seguente:


- Appunti personali delle lezioni di Calcolo Parallelo e Distribuito (Prof.ssa Livia Marcellino)
- Slide del corso Calcolo Parallelo e Distribuito presenti su E-Learning
- https://stackoverflow.com/
- https://docs.microsoft.com


# Appendice

Di seguito viene riportato il codice del nostro algoritmo in parallelo, opportunamente commentato per facilitarne la comprensione.

```
#include <omp.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
//Dichiarazione dei prototipi delle funzioni
void crea_matrice(int  ,int , int ** );
void stampa_matrice(int  ,int  ,int ** );
void vettore_ris(int, int *);
int controllo_threads(int, int);

int main(){
int N, M,i,j;
int **A, *b;
int nloc, p,sum,r,id,step,start,end,tmp,max;
double t1,t0,tp;
srand(time(NULL)); //Inizializzo il seme ad ogni run
printf("Inserisci numero di righe \n");
scanf("%d", &N);
printf("Inserisci numero di colonne \n");
scanf("%d", &M);

//prendo il numero massimo di thread assegnati con export
max=omp_get_max_threads();

//Funzione che ritorna il numero di thread da usare in omp parallel
tmp=controllo_threads(max,M);

//Allocazione dinamica della matrice
A = (int**)malloc(N * sizeof(int*));
        for (i = 0; i < N; i++)
            A[i] = (int*)malloc(M * sizeof(int));

//Allocazione dinamica del vettore
b = (int*)malloc(M * sizeof(int));
```

```
//funzione che riempie la matrice
crea_matrice(N,M,A);
//funzione per stampare la matrice
stampa_matrice(N,M,A);

t0=omp_get_wtime();
#pragma omp parallel num_threads(tmp) private(i,j,id,start,end,step,sum) shared(A,b,r) //Inizio codice parallelo
{
        p=omp_get_num_threads();
        nloc=M/p;
        //Implementazione perfetta divisibilità
        r=M%p;
        id=omp_get_thread_num();
      if (id < r)
        {
                nloc++;
                step=0;
        }
        else
                step=r;
        //Dichiaro inizio e fine sottomatrice
        start=nloc*id+step;
        end=nloc*id+step+nloc-1;

        //Effettuo le somme locali per ogni thread
        for(i=0; i<N; i++){
                for(j=start; j<=end; j++){
                        sum=A[i][j]+sum;
                }
        }
        printf("La sottomatrice che va da colonna: %d a colonna: %d è gestita dal thread:"
                "%d \n La somma degli elementi della sottomatrice e': %d \n",start, end, id, sum);
        b[nloc*id]=sum;


} //Fine codice parallelo
t1=omp_get_wtime();
tp=t1-t0;
        vettore_ris(M,b); //Funzione che stampa il vettore risultante
        printf("il tempo trascorso e' : %f \n",tp);
return 0;
}

void crea_matrice(int N, int M, int **A) {
int i,j;
        //riempio la matrice con numeri casuali da 1 a 10
        for (i=0; i<N; i++) {
                for(j=0; j<M; j++){
                        A[i][j]=rand()%10+1;

                }
           }
}

void stampa_matrice(int N, int M, int **A) {
int i,j;
        for (i=0; i<N; i++){
                for(j=0; j<M; j++){
                        printf("%d  ", A[i][j]);
                }
                printf("\n");
        }
 }
```

```
void vettore_ris(int M, int *b){
int j;
        for(j=0; j<M; j++){
                printf("Il vettore alla posizione %d risultante e': %d \n", j , b[j]);
        }
}
//Questa funzione setta il giusto numero di threads, se il numero di threads è > colonne, vuol dire che questi sono in eccesso,
//al più 1 thread gestisce 1 colonna.
int controllo_threads(int max, int M) {
int tmpf;
        if(max>M)
                tmpf=M;
        else
                tmpf=max;
return tmpf;
}
```