

# CALCOLO PARALLELO E DISTRIBUITO

## Progetto d'esame

DOCENTE: Livia Marcellino

A.A. 2021/2022

**Studente:** Gennaro Panico  
**Matricola:** 0124002115



# Indice

<b>1</b>	<b>Definizione ed Analisi del problema</b>	<b>3</b>
<b>2</b>	<b>Descrizione dell'approccio parallelo utilizzato</b>	<b>4</b>
2.1	Descrizione dell'algoritmo parallelo . . . . .	5
2.2	Input Output . . . . .	7
2.3	Routine implementate . . . . .	11
<b>3</b>	<b>Analisi delle performance del software</b>	<b>12</b>
<b>4</b>	<b>Riferimenti bibliografici</b>	<b>14</b>
<b>5</b>	<b>Appendice</b>	<b>15</b>
5.1	MAIN.c . . . . .	15
5.2	FUN.c . . . . .	16
5.3	SEQ_FUN.c . . . . .	17
5.4	LIB.H . . . . .	18

## 1 Definizione ed Analisi del problema

Si vuole implementare un programma in 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 leggere una matrice di dimensione  $N \times N$ , quindi i core devono collaborare per ricopiare in parallelo gli elementi della diagonale principale, in un vettore di lunghezza  $N$ . Infine, i core devono effettuare la somma degli elementi di tale vettore in parallelo. Segue un'immagine d'esempio in cui viene mostrata la matrice  $A$  con  $m = n$ , in quanto matrice quadrata:

diagonale principale

$$A(n) = \begin{pmatrix} \boxed{a_{1,1}} & a_{1,2} & a_{1,3} & a_{1,n} \\ a_{2,1} & \boxed{a_{2,2}} & a_{2,3} & a_{2,n} \\ a_{3,1} & a_{3,2} & \boxed{a_{3,3}} & a_{3,n} \\ a_{m,1} & a_{m,2} & a_{m,3} & \boxed{a_{m,n}} \end{pmatrix}$$

I valori presenti sulla diagonale principale devono poi essere ricopiati in parallelo in un vettore  $B$ , idealmente vogliamo:  $B = (a_{1,1} ; a_{2,2} ; a_{3,3} ; a_{m,n})$

e successivamente bisogna effettuare la somma di tali valori contenuti nel vettore  $B$ ,

in parallelo. Le operazioni da effettuare fondamentalmente quindi sono due, ovvero:

- **Copia in B:** la copia in parallelo degli elementi presenti sulla diagonale principale;
- **Somma:** la loro somma in parallelo (risolviamo il problema della somma).

L'implementazione di tale algoritmo è fattibile, c'è bisogno di adottare naturalmente tecniche di buona programmazione e soprattutto bisogna rispettare l'ambiente di lavoro in cui siamo. Si è deciso di implementare l'idea usando dei valori in virgola mobile con doppia precisione per la matrice, seguono maggiori dettagli su linguaggio di programmazione, ambiente, strategia e idea di sviluppo.

## 2 Descrizione dell'approccio parallelo utilizzato

Il problema posto in analisi richiede l'utilizzo della libreria 'openMP' del linguaggio C esplicitando indirettamente che siamo in ambiente MIMD-SM (Shared-Memory). Per quest'ambiente possiamo lavorare seguendo due Strategie, la III che è stata studiata, infatti, non è implementabile in ambiente MIMD-SM in quanto la memoria è condivisa. L'implementazione che segue è basata sulla II strategia perchè è risultata essere quella più adatta allo scopo, considerando anche che la macchina dove gira tale algoritmo ha a disposizione solo 4 Threads (architettura Intel Core i5-500 Mobile Series basata su 64bit). L'idea di base per la II Strategia è quella di far calcolare ad ogni core la propria somma parziale, ad ogni passo poi, la metà dei core (rispetto al core precedente) calcola un contributo della somma parziale. Il valore finale si trova nell'unica memoria condivisa. Da notare inoltre che parliamo di core in quanto siamo in Shared-Memory. Ci soffermiamo in particolare sull'analisi dell'algoritmo per il calcolo della somma sia in sequenziale che in parallelo, per cui in primo luogo analizzo la complessità di tempo dell'algoritmo sequenziale, che risulta essere  $T(N) = N-1$  somme, in parallelo invece, adottando la II Strategia in ambiente MIMD-SM con 'p' core, vale:  $T_p(N) = (N/p - 1 + \log_2 p)t_{calc}$

da cui possiamo scrivere la formula per il calcolo dello SpeedUp,  $Sp = \frac{T(N)}{T(p)}$

quindi sostituendo nella formula avremo per il nostro caso  $Sp = \frac{T(N-1)}{T_p(N)=(N/p-1+\log_2 p)t_{calc}}$

Successivamente, eseguiamo una valutazione dell'overhead per capire quanto lo SpeedUp differisce da quello ideale, infatti, ricordiamo che  $Sp \leq p$  e quindi  $Sp_{ideale} = p$  per cui andiamo a considerare la formula per l'OverHead  $Oh = pT_p - T_1 = Oh(p \log_2 p)$

Al crescere di p, l'Overhead aumenta, infatti, aumentando il numero di core si riduce il tempo impiegato per eseguire le operazioni richieste, ma bisogna trovare lo SpeedUp più vicino allo SpeedUp ideale.

La complessità di tempo però non è adatta a misurare l'efficienza di un algoritmo parallelo, difatti c'è bisogno di effettuare un'ulteriore analisi per l'efficienza data dalla formula  $Ep = \frac{S(p)}{p}$

che ci permette di capire quanto l'algoritmo sfrutta il parallelismo del calcolatore.

Quindi per il caso in analisi,  $Ep = \frac{\frac{T(N-1)}{T_p(N)=(N/p-1+\log_2 p)t_{calc}}}{p}$

In seguito, concentro l'analisi sull'operazione della somma nel dettaglio, infatti, l'algoritmo si divide in due parti, una parte relativa alle operazioni che devono essere eseguite esclusivamente in sequenziale e una parte relativa alle operazioni che potrebbero essere eseguite concorrentemente, ottenendo quindi  $T_s$ (tempo seriale) e  $T_c$ (tempo parallelo), che nel caso di due core, e 4 numeri, avremo 2 addizioni eseguite concorrentemente con tempo  $T_c$  e 1 addizione eseguita in sequenziale con tempo  $T_s$ . Possiamo quindi dedurre che  $T_s=a$  e  $T_c=1-a$ , con queste supposizioni definiamo la legge di Ware-Amdhal  $Sp = \frac{T_1}{T_p} = \frac{1}{a+a(1-a)/p}$

Effettuando dei calcoli con la legge di Ware-Amdhal osserviamo che fissato il size n del problema e aumentando il numero p di CP, esiste Pmax ovvero miglior numero di processori per risolvere il problema con l'algoritmo in esame, superato questo valore le prestazioni peggiorano. Infine valuto l'isoefficienza, una legge che ci permette di capire la nuova dimensione  $n_1$

affinché l'efficienza resti costante, parliamo quindi di scalabilità della dimensione del problema. Per analizzare ciò abbiamo bisogno di esprimere lo SpeedUp in funzione dell'overhead, quindi  $Sp = \frac{p}{\frac{p}{T_1} + 1}$

Dopo aver sostituito Sp in funzione dell'overhead nella formula  $Ep0(n0) = \frac{Sp0(n0)}{p0}$

e analogamente per Ep1(n1), otteniamo  $I = \frac{Oh(n1, p1)}{Oh(n0, p0)}$

che nel nostro specifico caso seguendo la II strategia, si evolve in:

$$n_1 - 1 = \frac{(p_1 \log_2 p_1)}{(p_0 \log_2 p_0)} (n_0 - 1)$$

Dove:

$$I = \frac{(p_1 \log_2 p_1)}{(p_0 \log_2 p_0)}$$

Per verificare se la somma è un algoritmo scalabile, ci serviamo della seguente formula:

$$T_p(n) = \left(\frac{n}{p} - 1 + \log_2 p\right) t_{calc}$$

Con questa formula otteniamo nuovi valori di 'Tp' che andremo a sostituire nel nuovo calcolo dell'efficienza, e successivamente facciamo le nostre valutazioni su quei calcoli. Possiamo concludere che l'operazione della somma è scalabile perchè l'efficienza rimane costante al crescere di p e di n con la legge di isoefficienza introdotta pocanzi.

## 2.1 Descrizione dell'algoritmo parallelo

Si è quindi deciso di lavorare con questa libreria rispettando le norme di buona programmazione. L'algoritmo è organizzato in 4 file, è stato ideato un file main.c che richiama due function, fun.c e seq\_fun.c, con annessa libreria chiamata lib.h. Il file seq\_fun.c è la versione sequenziale dell'algoritmo, che mi permette di verificare in che modo il singolo thread lavora sull'operazione di copia e l'operazione di somma, mentre il file fun.c contiene la versione parallela dell'algoritmo, e utilizzando le stesse modalità di stampa di seq\_fun.c ci permette di verificare come il lavoro viene distribuito tra i thread durante l'operazione di copia e l'operazione di somma. Il file lib.h semplicemente è la libreria ove risiedono i protoitipi delle due funzioni precedentemente descritte. Il main richiama le due funzioni in maniera diretta ovvero le funzioni sono entrambe di tipo 'void' per cui quando richiamate, mostrano direttamente l'output. Nel main.c viene generata una matrice randomica di tipo double(doppia precisione), con valori compresi tra 10 e 99 unicamente per questioni di visibilità ottimale della matrice, per fare ciò utilizzo la funzione 'srand', inoltre, la matrice stessa è allocata dinamicamente attraverso l'utilizzo di una 'calloc' e dopo aver lanciato le due funzioni fun.c e seq\_fun.c, tale memoria è deallocata con 'free(mat)'. Nel main viene inserito un controllo sulle dimensioni della matrice, infatti non è possibile inserire un valore maggiore di 2000, e inoltre, è possibile vedere il funzionamento dei thread e la stampa della matrice solo se le dimensioni sono minori o uguale a 20, perchè per dimensioni elevate la stampa risulterà illeggibile.

La funzione fun.c prevede l'allocazione dinamica del vettore 'vector' in cui andranno copiati i valori sulla diagonale principale in parallelo, e tale memoria viene deallocata al termine delle operazioni. Vengono utilizzate due direttive 'pragma omp parallel for' molto differenti tra loro, la prima prevede la clausola 'default(none)' con la quale il programmatore si assicura il pieno

controllo sulle variabili, seguono poi le clausole `'private(i)'` `'shared(vector,matrix,dim)'`, in questo modo la variabile di ciclo è privata mentre `vector`, `matrix` e `dim` sono condivise tra i core. Successivamente abbiamo la clausola `'schedule(dynamic, CHUNK)'` con cui definisco uno schedule dinamico che rispetta il `CHUNK`, in questo modo ad ogni thread sarà associato a un numero prestabilito di iterazioni, quindi quando avrà terminato, avrà assegnato un nuovo `CHUNK`. Infine abbiamo `'num_threads(num_proc)'` che mi permette di stabilire il numero di thread con cui deve lavorare la direttiva, tale numero risiede nella variabile intera `num_proc`. All'interno della direttiva ci sono due cicli `for`, il primo che cicla sulla dimensione della matrice (essendo quadrata, ne basta una) e assegna alla posizione `i`-sima l'elemento di posizione `i+i` della matrice, ovvero l'elemento sulla diagonale principale. Il secondo, invece, cicla sempre in base alla dimensione ma riporta gli elementi contenuti nel vettore, ovvero quelli estratti dalla matrice. La seconda direttiva prevede anch'essa una clausola `'schedule(dynamic, CHUNK)'` con la quale evito di ottenere tempi troppo discordanti perchè usando il `CHUNK` assicuro un lavoro controllato su tutti i thread con cui lavoro. In seguito, v'è la clausola `'num_threads'` a cui passo sempre la variabile `'num_proc'` e, infine, ciò che rende differenti di molto le due direttive, ovvero la clausola `'reduction(+:sum)'` a cui passo la variabile `'sum'`. Tale clausola mi permette di implementare la II Strategia per il calcolo della somma in ambiente MIMD-SM con openMP. All'interno della direttiva ciclo con il `'for'` sulla dimensione del vettore e sommo i valori contenuti in esso nelle posizioni `i`-sime. Entrambe le direttive vengono monitorate con il calcolo del tempo eseguendo successivamente la differenza tra gli intervalli, permettondoci di sapere quanto tempo impiega ogni direttiva a compiersi. Infine specifico che nell'algoritmo che segue, la non esatta divisibilità NON viene gestita, per cui alcuni core faranno 1 somma in più nella fase puramente parallela.

## 2.2 Input Output

L'algoritmo prevede che gli venga dato in input il valore di una delle dimensioni (un valore di tipo intero), poichè essendo una matrice quadrata, si è scelto di prenderne in input una soltanto per generare la matrice. Inoltre, tale valore deve essere minore o uguale di 2000 altrimenti il programma termina lanciando un messaggio di errore con annessa motivazione. In output si ottengono informazioni riguardanti lo scopo dell'algoritmo e le sue limitazioni, precisamente, se si digita un valore minore o uguale a 20 vengono stampate le iterazioni di ogni thread per la parte sequenziale e per la parte parallela per cui è possibile constatare in che modo il lavoro viene effettuato. Ovviamente, vengono effettuate poi stampe per i valori estratti dalla diagonale principale nel vettore 'A' e la somma di tali valori. Se si digitano valori maggiori di 20, tali stampe NON sono previste in quanto l'output risulterebbe illeggibile, ma Vi sono, a prescindere, anche stampe che riportano i tempi sia in sequenziale che in parallelo per permetterci di effettuare un confronto, anche se la parte interessante riguarda il lavoro tra i thread per la somma. Seguono print-screen che mostrano alcuni degli output più importanti dell'algoritmo:

```
gennaro@pop-os:~/Scrivania/Progetto_CPDS$ ./main.o

      BENVENUTO!
Data la dimensione, questo programma genera una matrice randomica con valori compresi tra 10 e 99, questa scelta è stata presa unicamente per visualizzare al meglio la matrice di double
successivamente, ricopia in parallelo i valori sulla diagonale principale e li somma (in parallelo).

-----| Start |-----

ATTENZIONE, con valori <=20 è possibile vedere come lavorano i thread, anche se si suggerisce di inserire valori da 500 in poi per accertare i test.
Inserisci il valore delle dimensioni della matrice quadrata (nxn) che sia <= 2000:      2500

-----| Stai generando i valori in una matrice quadrata 2500x2500. --> Ordine: 2500

ERRORE: dimensione errata della matrice, problemi con la visualizzazione della matrice.
```

Figura 1: Output-Errore

```

gennaro@pop-os:~/Scrivania/Progetto_CPO$ ./main.o

      BENVENUTO!
Data la dimensione, questo programma genera una matrice randomica con valori compresi tra 10 e 99, questa scelta è stata presa unicamente per visualizzare al meglio la matrice di double
successivamente, ricopia in parallelo i valori sulla diagonale principale e li somma (in parallelo).

-----| Start |-----

ATTENZIONE, con valori <=20 è possibile vedere come lavorano i thread, anche se si suggerisce di inserire valori da 500 in poi per accertare i test.
Inserisci il valore delle dimensioni della matrice quadrata (nxn) che sia <= 2000:      1000

-----| Stai generando i valori in una matrice quadrata 1000x1000. --> Ordine: 1000

-----| Matrice 1000x1000, generata!

-----| Start Algoritmo Sequenziale |-----

-----| Thread View: somma in SEQUENZIALE degli elementi nel vettore 'A'.

||||> La somma dei valori sulla diagonale principale vale: 54421.509633

-----| Tempi prest.

Start: 28049.808046   End: 28049.808096

||||> Tempo di calcolo Copia: 0.000050

Start: 28049.808111   End: 28049.808132

||||> Tempo di calcolo Somma: 0.000021

-----| Start Algoritmo Parallelo |-----

-----| Thread View: somma in PARALLELO degli elementi nel vettore 'A'.

||||> La somma dei valori sulla diagonale principale vale: 54421.509633

-----| Tempi prest.

Start: 28049.808265   End: 28049.808460

||||> Tempo di calcolo Copia: 0.000195

Start: 28049.808473   End: 28049.808480

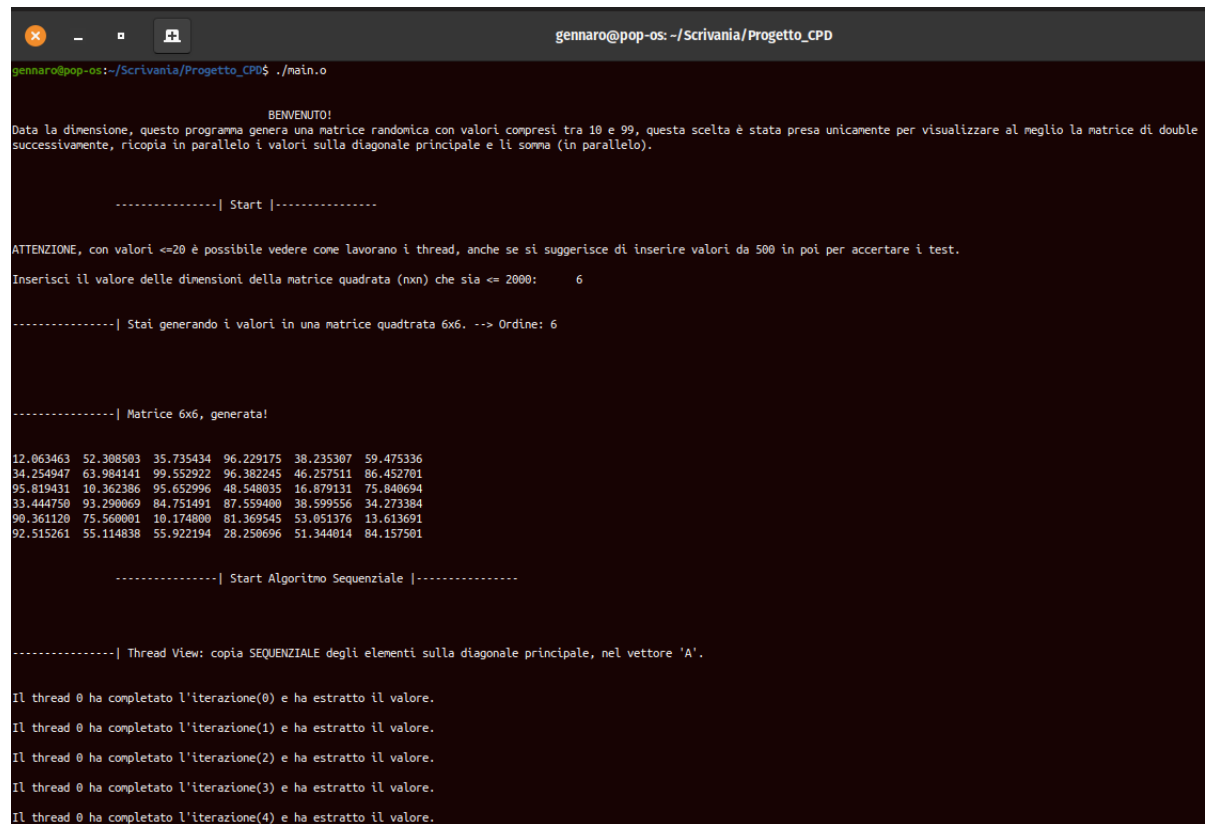
||||> Tempo di calcolo Somma: 0.000007

```

Figura 2: Output-Privo di stampe indicative



Per gli output che seguono, è stato impostato il CHUNK a 2, ed è stata data in input una matrice 6x6, in questo modo è possibile constatare come lavorano le stampe.



```
gennaro@pop-os: ~/Scrivania/Progetto_CPD$ ./main.o
BENVENUTO!
Data la dimensione, questo programma genera una matrice randomica con valori compresi tra 10 e 99, questa scelta è stata presa unicamente per visualizzare al meglio la matrice di double
successivamente, ricopia in parallelo i valori sulla diagonale principale e li somma (in parallelo).

-----| Start |-----

ATTENZIONE, con valori <=20 è possibile vedere come lavorano i thread, anche se si suggerisce di inserire valori da 500 in poi per accertare i test.
Inserisci il valore delle dimensioni della matrice quadrata (nxn) che sia <= 2000:      6

-----| Stai generando i valori in una matrice quadrata 6x6. --> Ordine: 6

-----| Matrice 6x6, generata!

12.063463  52.308503  35.735434  96.229175  38.235307  59.475336
34.254947  63.904141  99.552922  96.382245  46.257511  86.452701
95.819431  10.362386  95.652996  48.548035  16.879131  75.840694
33.444750  93.290069  84.751491  87.559400  38.599556  34.273384
90.361120  75.560001  10.174800  81.369545  53.051376  13.613691
92.515261  55.114838  55.922194  28.250696  51.344014  84.157501

-----| Start Algoritmo Sequenziale |-----

-----| Thread View: copia SEQUENZIALE degli elementi sulla diagonale principale, nel vettore 'A'.

Il thread 0 ha completato l'iterazione(0) e ha estratto il valore.
Il thread 0 ha completato l'iterazione(1) e ha estratto il valore.
Il thread 0 ha completato l'iterazione(2) e ha estratto il valore.
Il thread 0 ha completato l'iterazione(3) e ha estratto il valore.
Il thread 0 ha completato l'iterazione(4) e ha estratto il valore.
```

Figura 3: Output-Indicativo

```
gennaro@pop-os: ~/Scrivania/Progetto_CPD

Il thread 0 ha completato l'iterazione(0) e ha estratto il valore.
Il thread 0 ha completato l'iterazione(1) e ha estratto il valore.
Il thread 0 ha completato l'iterazione(2) e ha estratto il valore.
Il thread 0 ha completato l'iterazione(3) e ha estratto il valore.
Il thread 0 ha completato l'iterazione(4) e ha estratto il valore.
Il thread 0 ha completato l'iterazione(5) e ha estratto il valore.
||||> Vettore A (valori estratti sulla diagonale principale): 12.063463 63.984141 95.652996 87.559400 53.051376 84.157501
-----| Thread View: somma in SEQUENZIALE degli elementi nel vettore 'A'.

Il thread 0 ha completato l'iterazione(0)
Il thread 0 ha completato l'iterazione(1)
Il thread 0 ha completato l'iterazione(2)
Il thread 0 ha completato l'iterazione(3)
Il thread 0 ha completato l'iterazione(4)
Il thread 0 ha completato l'iterazione(5)

||||> La somma dei valori sulla diagonale principale vale: 396.468876

-----| Tempi presi.

Start: 1485.420991 End: 1485.421074

||||> Tempo di calcolo Copia: 0.000083

Start: 1485.421120 End: 1485.421205

||||> Tempo di calcolo Somma: 0.000085
```

Figura 4: Output-Indicativo

```
gennaro@pop-os: ~/Scrivania/Progetto_CPD
Start: 1485.421120 End: 1485.421205
||||> Tempo di calcolo Somma: 0.000885

-----| Start Algoritmo Parallelo |-----

-----| Thread View: copia in PARALLELO degli elementi sulla diagonale principale, nel vettore 'A'.

Il thread 0 ha completato l'iterazione(2) e ha estratto il valore.
Il thread 0 ha completato l'iterazione(3) e ha estratto il valore.
Il thread 1 ha completato l'iterazione(0) e ha estratto il valore.
Il thread 1 ha completato l'iterazione(1) e ha estratto il valore.
Il thread 3 ha completato l'iterazione(4) e ha estratto il valore.
Il thread 3 ha completato l'iterazione(5) e ha estratto il valore.
||||> Vettore A (valori estratti sulla diagonale principale): 12.063463 63.984141 95.652996 87.559408 53.051376 84.157581
-----| Thread View: somma in PARALLELO degli elementi nel vettore 'A'.

Il thread 2, ha completato l'iterazione(0)
Il thread 2, ha completato l'iterazione(1)
Il thread 1, ha completato l'iterazione(2)
Il thread 1, ha completato l'iterazione(3)
Il thread 3, ha completato l'iterazione(4)
Il thread 3, ha completato l'iterazione(5)

||||> La somma dei valori sulla diagonale principale vale: 396.468876

-----| Tempi presi.

Start: 1485.421343 End: 1485.421858
||||> Tempo di calcolo Copia: 0.000516

Start: 1485.421878 End: 1485.421945
||||> Tempo di calcolo Somma: 0.000867
gennaro@pop-os: ~/Scrivania/Progetto_CPD$
```

Figura 5: Output-Indicativo

## 2.3 Routine implementate

L'algoritmo prevede l'utilizzo della libreria `omp.h` per la parte in parallelo, `stdio.h` per effettuare le stampe e leggere l'input, `stdlib.h` e infine `"lib.h"` ideata dal programmatore e contenente i prototipi delle funzioni `fun.c` e `seq_fun.c`.

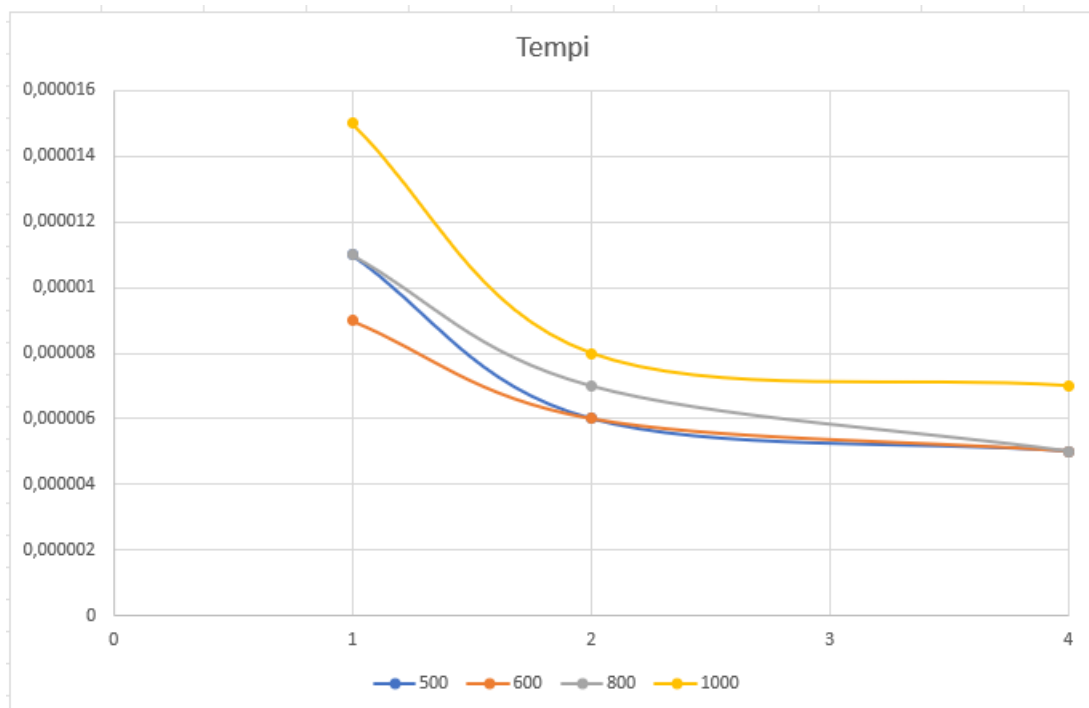
### 3 Analisi delle performance del software

In questa sezione, per restare in linea con l'analisi fatta fino ad ora, si è deciso di soffermarsi unicamente sul calcolo parallelo della somma, per cui tutte le valutazioni che seguono riguardano precisamente quella parte dell'algoritmo, sebbene vi siano due parti che vengono implementate in parallelo.

Nella tabella che segue, vengono inseriti i tempi risultanti dai test eseguiti, che per praticità sono stati fatti tenendo conto di matrici quadrate con dimensioni pari a 500, 600, 800, 1000.

Somma

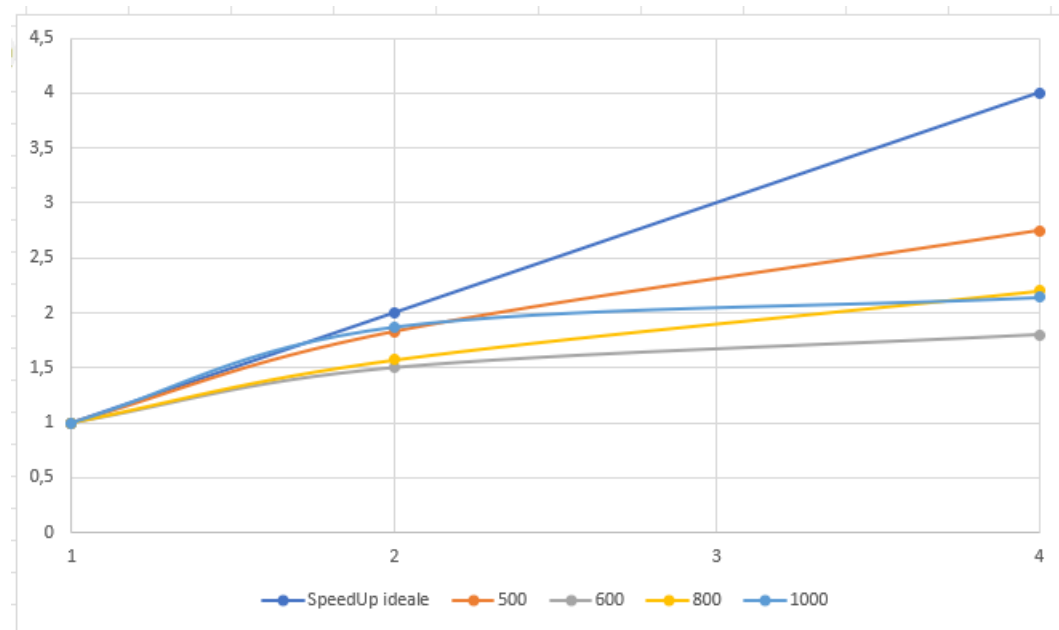
P \ N	500	600	800	1000
1	0.000011	0.000009	0.000011	0.000015
2	0.000006	0.000006	0.000007	0.000008
4	0.000005	0.000005	0.000005	0.000007



Successivamente proseguo con il calcolo dello Speed-Up tenendo conto, come già scritto precedentemente, della formula  $Sp = \frac{T_1}{T_p}$

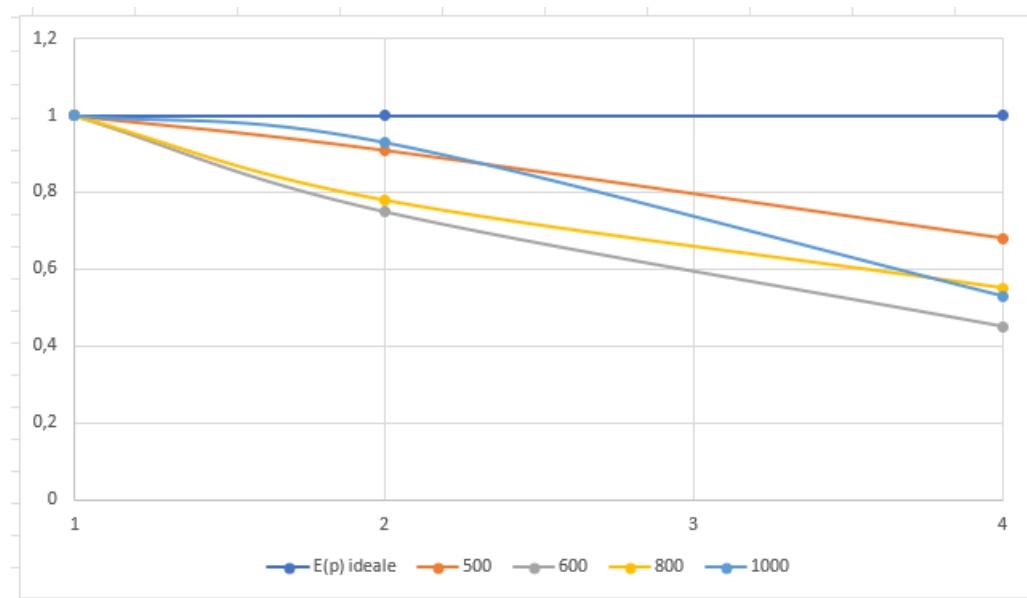
Speed-Up S(p)

p	S(p) con n=500	S(p) con n=600	S(p) con n=800	S(p) con n=1000
2	1,83	1,50	1,57	1,87
4	2,75	1,80	2,20	2,14



Noto che lo Speed-Up su due core è più vicino allo Speed-Up ideale, che ricordiamo essere uguale al numero di core, ma in una situazione più reale esso è minore del numero dei core. Procedo successivamente al calcolo dell'efficienza confermando che l'algoritmo girando su due core è più efficiente rispetto allo stesso algoritmo che gira su 4 core.

p	E(p) con n=500	E(p) con n=600	E(p) con n=800	E(p) con n=1000
1	1	1	1	1
2	0,91	0,75	0,78	0,93
4	0,68	0,45	0,55	0,53



#### 4 Riferimenti bibliografici

Per lo sviluppo e l'analisi di questo algoritmo sono state utilizzate le slides e la dispensa, messe a disposizione agli studenti dalla Gent.ma Professoressa Marcellino Livia, presenti sulla piattaforma E-learning (<https://elearning.uniparthenope.it/course/view.php?id=111>) per il corso di Calcolo Parallelo e Distribuito CFU9.

## 5 Appendice

### 5.1 MAIN.c

---

```
1  /* Progetto di Calcolo Parallelo e Distribuito 2021/2022
2  *
3  * CFU: 9
4  *
5  * Autore: Panico Gennaro - '0124002115'
6  *
7  * Il codice che segue è costituito da commenti di riga,
8  * ove espongo per l'appunto, riga per riga, ciò che faccio,
9  * ma per avere informazioni riguardo l'idea di sviluppo e
10 * i risultati dei test, consultare l'annessa documentazione
11 * presente su giitHub.
12 */
13
14 #include <stdio.h>
15 #include <omp.h>
16 #include <stdlib.h>
17 #include "lib.h"
18 #include <time.h>
19
20 int main() {
21     //Sezione eseguita dal Master Thread.
22     //Dichiarazione delle variabili.
23     int i, j, n;
24     double *mat;
25
26
27     //Ricorda di valutare un possibile controllo sui pari e sul max della dimensione.
28     printf("\n\n\t\t\t\t\tBENVENUTO!\nData la dimensione, questo programma genera una matrice randomica con valori compresi tra 10 e 99, questa scelta è stata presa unicamente per visualizzare\n\n\n");
29     printf("\n\n\t\t\t\t\t-----| Start |-----\n\n\n");
30     printf("ATTENZIONE, con valori <=20 è possibile vedere come lavorano i thread, anche se si suggerisce di inserire valori da 500 in poi per accertare i test.\n\n");
31     printf("Inserisci il valore delle dimensioni della matrice quadrata (nxn) che sia <= 2000:\t");
32     scanf("%d", &n);
33     printf("\n\n\t\t\t\t\t-----| Stai generando i valori in una matrice quadrata %dx%d. --> Ordine: %d \n\n", n, n, n);
34
35     if(n<=2000){
36         //Allocazione dinamica della matrice.
37         mat = calloc(n*n,sizeof(double*));
38
39         //Riempimento della matrice.
40         srand(time(NULL));
41         for(i=0; i<n; i++){
42             for(j=0; j<n; j++){
43                 //genero la matrice randomica con numeri compresi tra 10 e 99 (10<numeri<99).
44                 mat[j+i*n] = ( (double)rand() * ( 100 - 10 ) ) / (double)RAND_MAX + 10;
45             }
46         }
47     } else {
48         printf("ERRORE: dimensione errata della matrice, problemi con la visualizzazione della matrice.\n\n\n");
49         return 0;
50     }
51
52     printf("\n\n\n\t\t\t\t\t-----| Matrice %dx%d, generata!\n\n\n", n, n);
53
54     if(n<=20){
55         //Sezione di stampa a video della matrice.
56         for(i=0; i<n; i++){
57             for(j=0; j<n; j++){
58                 printf("%f ", mat[j+i*n]);
59             }
60             printf("\n");
61         }
62     }
63 }
64
65 printf("\n\n\t\t\t\t\t-----| Start Algoritmo Sequenziale |-----\n\n\n");
66 seq_sum(mat,n);
67 printf("\n\n\t\t\t\t\t-----| Start Algoritmo Parallelo |-----\n\n\n");
68 parallel_sum(mat,n);
69 free(mat);
70 }
```

---

## 5.2 FUN.c

```
1  /* Progetto di Calcolo Parallelo e Distribuito 2021/2022
2  *
3  * CFU: 9
4  *
5  * Autore: Panico Gennaro - '0124002115'
6  *
7  * Il codice che segue è costituito da commenti
8  * di riga, ove espongono per l'appunto, riga per riga ciò
9  * che faccio, ma per avere informazioni riguardo l'idea
10 * di sviluppo e i risultati dei test, consultare l'annessa
11 * documentazione presente su gitHub.
12 * La macchina su cui gira possiede maz: 4 thread e 2 core.
13 */
14
15 #include <stdio.h>
16 #include <stdlib.h>
17 #include <omp.h>
18 #include "lib.h"
19
20 #define CHUNK 2 //aumentando questo valore lo schedule tende ad essere static mentre diminuendolo tende ad essere di tipo dynamic.
21
22 //FUNZIONE PER ESTRARRE GLI ELEMENTI PRESENTI SULLA DIAGONALE PRINCIPALE E SUCCESSIVAMENTE SOMMARLI.
23
24 void parallel_sum(double *matrix, int dim){
25
26     //Sezione eseguita dal Master Thread.
27     int i, num_proc=4;
28     double *vector;
29     double t0, t1, ts0, ts1, final_t=0.0, final_s=0.0, sum=0.0;
30
31
32     //Allocazione dinamica del vettore.
33     vector = (double *)calloc(dim,sizeof(double));
34
35     if(dim<=20){
36         printf("\n\n-----| Thread View: copia in PARALLELO degli elementi sulla diagonale principale, nel vettore 'A'.\n\n");
37     }
38
39     //Sezione eseguita in Parallelo
40     //Genero un team di thread con la direttiva #pragma omp parallel.
41     t0 = omp_get_wtime();
42     #pragma omp parallel for default(none) private(i) shared(vector,matrix,dim) schedule(dynamic, CHUNK) num_threads(num_proc)
43     for(i=0; i<dim; i++){
44         if(dim<=20){
45             printf("\nIl thread %d ha completato l'iterazione(%d) e ha estratto il valore.\n", omp_get_thread_num(), i);
46         }
47         vector[i] = matrix[i+i*dim];
48     }
49     t1 = omp_get_wtime();
50     final_t = t1-t0;
51
52     if(dim<=20){
53         printf("\n\n||||> Vettore A (valori estratti sulla diagonale principale): ");
54         for(i=0; i<dim; i++){
55             printf("%f ", vector[i]);
56         }
57     }
58
59
60     printf("\n\n-----| Thread View: somma in PARALLELO degli elementi nel vettore 'A'.\n\n");
61     ts0 = omp_get_wtime();
62     #pragma omp parallel for reduction(+:sum) schedule(dynamic, CHUNK) num_threads(num_proc) //optare per guided o dynamic
63     for(i=0; i<dim; i++){
64         if(dim<=20){
65             printf("\nIl thread %d, ha completato l'iterazione(%d)\n", omp_get_thread_num(), i);
66         }
67         sum = sum + vector[i]; //valutare in caso non ci sia un numero divisibile per t.
68     }
69     ts1 = omp_get_wtime();
70     final_s = ts1-ts0;
71
72
73     printf("\n\n||||> La somma dei valori sulla diagonale principale vale: %f", sum);
74
75     //Sezione di stampa dei tempi.
76     printf("\n\n\n\n-----| Tempi presi.\n\n\n\n");
77     printf("\n\nStart: %lf End: %lf\n\n", t0, t1);
78     printf("\n\n||||> Tempo di calcolo Copia: %lf\n\n", final_t);
79     printf("\n\nStart: %lf End: %lf\n\n", ts0, ts1);
80     printf("\n\n||||> Tempo di calcolo Somma: %lf\n\n", final_s);
81
82     free(vector);
83
84 }
85 }
```



## 5.3 SEQ\_FUN.c

```
1  /* Progetto di Calcolo Parallelo e Distribuito 2021/2022
2  *
3  * CFU: 9
4  *
5  * Autore: Panico Gennaro - '0124002115'
6  *
7  * Il codice che segue è costituito da commenti
8  * di riga, ove espongono per l'appunto, riga per riga ciò
9  * che faccio, ma per avere informazioni riguardo l'idea
10 * di sviluppo e i risultati dei test, consultare l'annessa
11 * documentazione presente su GitHub.
12 * La macchina su cui gira possiede maz: 4 thread e 2 core.
13 */
14
15 #include <stdio.h>
16 #include <stdlib.h>
17 #include <omp.h>
18 #include "lib.h"
19
20 #define CHUNK 10 //aumentando questo valore lo schedule tende ad essere static mentre diminuendolo tende ad essere di tipo dynamic.
21
22 //FUNZIONE PER ESTRARRE GLI ELEMENTI PRESENTI SULLA DIAGONALE PRINCIPALE E SUCCESSIVAMENTE SOMMARLI.
23
24 void seq_sum(double *matrix, int dim){
25
26     //Dichiarazione variabili.
27     int i;
28     double *vector;
29     double t0, t1, ts0, ts1, final_t, final_s, sum=0.0;
30
31
32     //Allocazione dinamica del vettore.
33     vector = calloc(dim,sizeof(double*));
34
35     if(dim<=20){
36         printf("\n\n-----| Thread View: copia SEQUENZIALE degli elementi sulla diagonale principale, nel vettore 'A'.\n\n");
37     }
38
39     //Genero un team di thread con la direttiva #pragma omp parallel.
40     t0 = omp_get_wtime();
41     for(i=0; i<dim; i++){
42         if(dim<=20){
43             printf("\n\nThread %d ha completato l'iterazione(%d) e ha estratto il valore.\n", omp_get_thread_num(), i);
44         }
45         vector[i] = matrix[i+i*dim];
46     }
47     t1 = omp_get_wtime();
48     final_t = t1-t0;
49
50     if(dim<=20){
51         printf("\n\n|||> Vettore A (valori estratti sulla diagonale principale): ");
52         for(i=0; i<dim; i++){
53             printf("%f ", vector[i]);
54         }
55     }
56
57     printf("\n\n-----| Thread View: somma in SEQUENZIALE degli elementi nel vettore 'A'.\n\n");
58     ts0 = omp_get_wtime();
59     for(i=0; i<dim; i++){
60         if(dim<=20){
61             printf("\n\nThread %d ha completato l'iterazione(%d)\n", omp_get_thread_num(), i);
62         }
63         sum += matrix[i+i*dim]; //valutare in caso non ci sia un numero divisibile per t.
64     }
65     ts1 = omp_get_wtime();
66     final_s = ts1-ts0;
67
68     printf("\n\n|||> La somma dei valori sulla diagonale principale vale: %f", sum);
69
70     //Sezione di stampa dei tempi.
71     printf("\n\n\n\n-----| Tempi presi.\n\n\n\n");
72     printf("\n\nStart: %lf End: %lf\n\n", t0, t1);
73     printf("\n\n|||> Tempo di calcolo Copia: %lf\n\n", final_t);
74     printf("\n\nStart: %lf End: %lf\n\n", ts0, ts1);
75     printf("\n\n|||> Tempo di calcolo Somma: %lf\n\n", final_s);
76
77     free(vector);
78 }
79 }
```

## 5.4 LIB.H

---

```
1  /* Progetto di Calcolo Parallelo e Distribuito 2021/2022
2  *
3  * CFU: 9
4  *
5  * Autore: Panico Gennaro - '0124002115'
6  *
7  * Libreria del progetto, consultare la documentazione su gitHub.
8  */
9
10 #ifndef LIBRERIA_H
11 void parallel_sum(double *matrix, int dim);
12 void seq_sum(double *matrix, int dim);
13 #endif
```

---