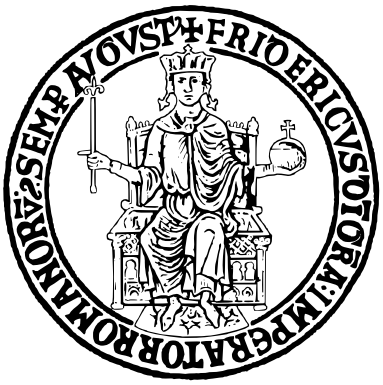


Università degli studi di Napoli Federico II

Scuola Politecnica e delle Scienze di base

Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione

Corso di laurea magistrale in Informatica

Parallel and Distributed Computing

Documentazione relativa al calcolo del prodotto matrice-vettore su architettura MIMD a memoria condivisa

Studente
Francesco Scarfato

Matricola
N86003769

2023-2024

Indice

Capitolo 1

Descrizione del problema

Nel campo dell'informatica, viene definito *parallelismo* la decomposizioni di un problema in sotto problemi risolti contemporaneamente da più processori. L'introduzione del calcolo parallelo ha permesso di superare molti limiti tecnologici e di ottenere prestazioni di alto livello. L'architettura MIMD (Multiple Instruction Multiple Data) è un paradigma che consente di eseguire diverse operazioni su diversi processori simultaneamente. Nel caso di studio, tutte le unità di elaborazione condividono la stessa memoria, quindi a differenza di un'architettura MIMD a memoria distribuita, non è più un problema la gestione della comunicazione tra i processori ma si sposta il focus sulla gestione dell'accesso ai dati condivisi.

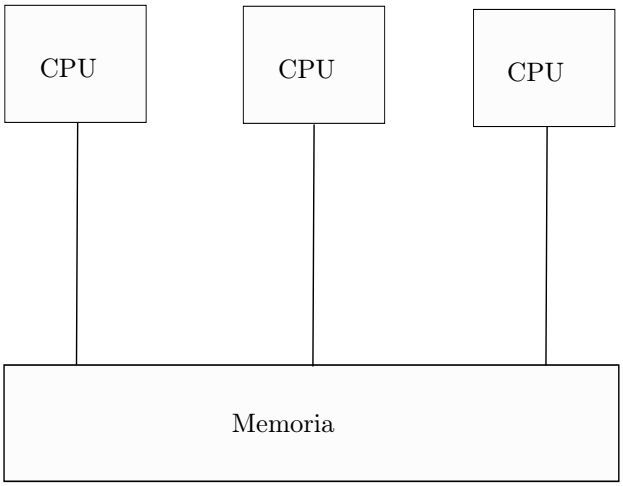


Figura 1.1: Architettura MIMD a memoria condivisa

Il calcolo del prodotto matrice-vettore è un'operazione tra una matrice e un vettore che produce un altro vettore.

Sia $A \in M_{r,c}$ matrice con r righe, c colonne e $X \in M_{r_1,1}$ vettore con r_1 numero di componenti, allora è possibile eseguire il prodotto matrice-vettore se e solo se $v_1 = c$ e il vettore risultante $B \in M_{r_2,1}$ avrà r_2 componenti con $v_2 = r$ quindi

$$A \in M_{r,c} \times X \in M_{c,1} \rightarrow B \in M_{r,1}$$

Nello specifico il prodotto viene calcolato moltiplicando ogni colonna della matrice per ogni componente

del vettore e poi sommando.

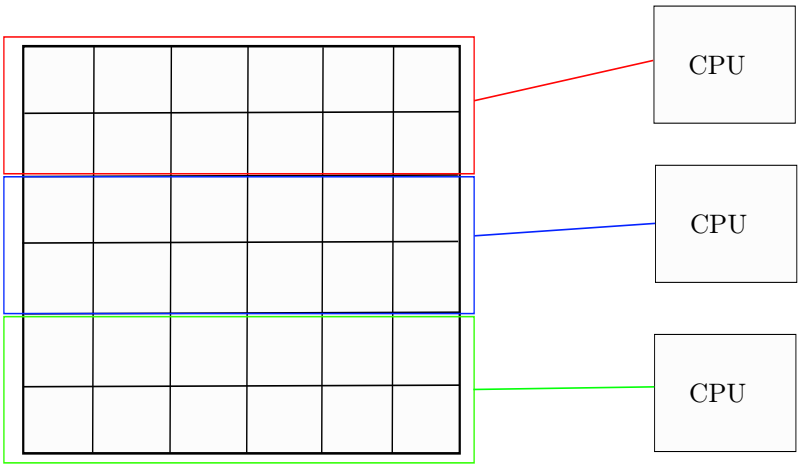
$$\begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,c} \\ a_{2,1} & a_{2,2} & \dots & a_{2,c} \\ \dots & \dots & \dots & \dots \\ a_{r,1} & a_{r,2} & \dots & a_{r,c} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_c \end{bmatrix} = \begin{bmatrix} a_{1,1} \cdot x_1 + a_{1,2} \cdot x_2 + \dots + a_{1,c} \cdot x_c \\ a_{2,1} \cdot x_1 + a_{2,2} \cdot x_2 + \dots + a_{2,c} \cdot x_c \\ \dots \\ a_{r,1} \cdot x_1 + a_{r,2} \cdot x_2 + \dots + a_{r,c} \cdot x_c \end{bmatrix}$$

Capitolo 2

Descrizione dell'algoritmo

L'idea è di dividere il problema in sotto problemi risolvibili nello stesso momento da diversi processori. La risoluzione del problema del prodotto tra una matrice e vettore deve tenere conto dell'architettura MIMD a memoria condivisa. La matrice e il vettore sono condivisi tra i processori. In questo caso l'algoritmo è sviluppato in modo che ogni processore calcoli un prodotto matrice-vettore più piccolo rispetto alla matrice originale, nello specifico viene calcolato un prodotto tra un blocco di righe e l'intero vettore

Tutte le righe vengono divise tra i processori. Nel caso in cui il numero di righe non sia divisibile con il numero di processori allora ci saranno delle CPU a cui verranno "assegnate" righe in più.



Subito dopo ogni processore calcola il prodotto tra le righe assegnate e il vettore.

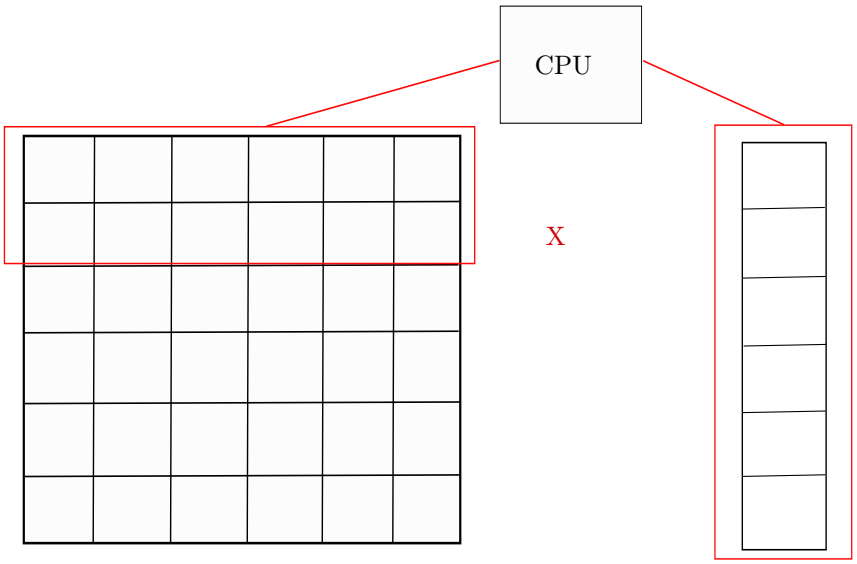


Figura 2.1: Prodotto matrice-vettore per un singolo processore

Parallelamente ogni processore calcola delle componenti del vettore risultante.

Listing 2.1: Codice per il prodotto matrice-vettore

```
1    for(i=0,i<r_p,i++) {
2        for(j=0,j<c,j++) {
3            resultantVector[i] += mainMatrix[i][j] * vector[j]
4        }
5    }
```

Il numero di componenti risultanti è uguale al numero di righe assegnate poiché vale sempre

$$A \in M_{r,c} \cdot X \in M_{c,1} \rightarrow B \in M_{r,1}$$

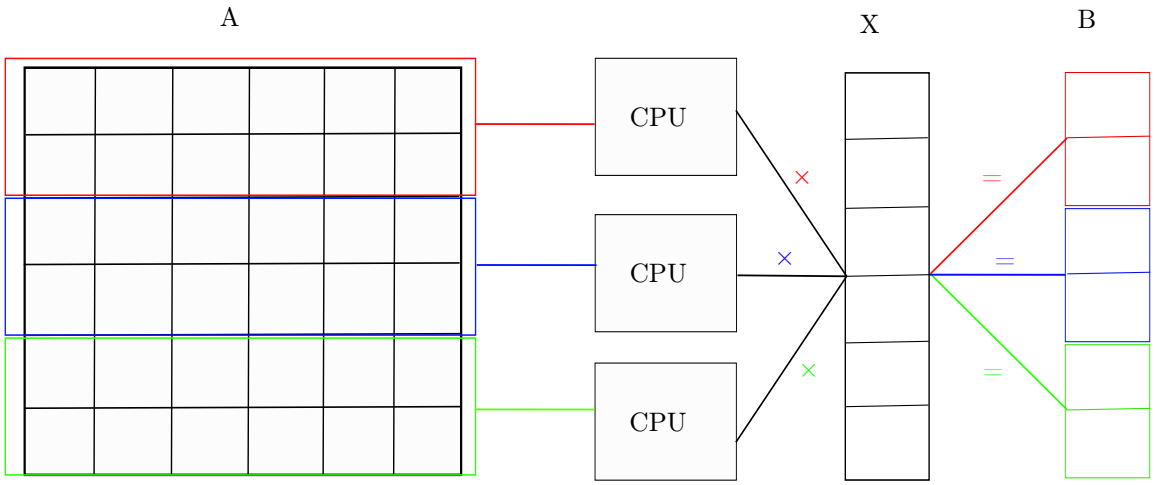


Figura 2.2: Prodotto matrice-vettore

Alla fine, raggruppando tutti i risultati dei processori, si ottiene il vettore risultante.

Capitolo 3

Descrizione delle routine

Prima di descrivere le routine, vengono presentate le librerie utilizzate:

- `omp.h`, libreria che contiene funzionalità per programmare un ambiente multiprocessore a memoria condivisa. Contiene un set di funzioni e variabili di ambiente per creare thread e sincronizzarli
- `stdio.h`, libreria standard di C per la gestione dell'input/output
- `stdlib.h`, libreria standard di C che comprende alcune utility per allocare memoria (`malloc()`), per la conversione (`atoi()`) e per la generazione di numeri pseudocasuali
- `time.h`, libreria standard di C per gestire tipi di dato temporali (usata per la generazione di numeri casuali dandogli come seed l'orario attuale con `time()`)
- `getopt.h`, libreria che contiene variabili e funzioni per gestire i parametri passati tramite linea di comando
- `ctype.h`, libreria che dichiara funzioni per la gestione dei singoli caratteri (usata per controllare se un carattere è effettivamente un numero con `isdigit()`)

Listing 3.1: Librerie incluse nel codice

```
1  #include <omp.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <sys/time.h>
5  #include <time.h>
6  #include <getopt.h>
7  #include <ctype.h>
```

Il primo step è la configurazione del problema tramite la lettura dei parametri da linea di comando. Il programma, per essere chiamato, necessita di alcune opzioni. Per chiarezza, la sinossi del comando è

```
1  ./product-matrix-vector -t <number of processor>
2            -r <number of rows> -c <number of columns>
3            -v <vector size>
```

in cui `-t` è un intero e indica il numero di processori, `-r` è un intero e indica il numero di righe della matrice, `-c` è un intero e indica il numero di colonne della matrice e `-v` è un intero e indica il numero di elementi nel vettore. Tutte le opzioni sono necessarie.

La configurazione inizia con `interpretCommandLine()` che legge le opzioni elencate e le memorizza con i dovuti controlli

```
1  void interpretCommandLine(/*args*/) {
2  int opt, tmp;
3  //controllo opzioni necessarie
4  int checkT = -1, checkV = -1, checkR = -1, checkC = -1;
```

```
5
6       //lettura opzioni
7       while ((opt = getopt(argc, argv, ":t:r:c:v:")) != -1) {
8           switch (opt) {
9               //Opzione per il numero di threads
10              case 't':
11                  checkT = 0;
12                  if((tmp = atoi(optarg)) == 0)
13                      //codice gestione errore
14                  *nThreads = tmp;
15                  break;
16              //Opzione per il numero di righe della matrice
17              case 'r':
18                  checkR = 0;
19                  if((tmp = atoi(optarg)) == 0)
20                      //codice gestione errore
21                  dimsMatrix[0] = tmp;
22                  break;
23              //Opzione per il numero di colonne della matrice
24              case 'c':
25                  checkC = 0;
26                  if((tmp = atoi(optarg)) == 0)
27                      //codice gestione errore;
28                  dimsMatrix[1] = tmp;
29                  break;
30              //Opzione per il numero di elementi del vettore
31              case 'v':
32                  checkV = 0;
33                  if ((tmp = atoi(optarg)) == 0)
34                      //codice gestione errore
35                  *dimVector = tmp;
36                  break;
37              case '?':
38                  //codice gestione errore
39          }
40      }
41      //Controlla se tutte le opzioni sono state date a riga di comando
42      if (checkR != 0 || checkC != 0 || checkV != 0 || checkT != 0) {
43          //codice gestione errore
44      }
45      //controlla se n.colonne == n.di elementi del vettore
46      if (dimsMatrix[1] != (*dimVector)){
47          //codice gestione errore
48      }
49
50  }
```

Una volta inizializzati correttamente tutti i parametri allora viene inizializzato il numero di thread con la funzione `omp_set_num_threads(nThreads)` della libreria OpenMP. In seguito vengono allocati e inizializzati la matrice con numeri casuali e il vettore tramite le funzioni `mallocMatrix()` e `generateRandomNumbers()`.

Listing 3.2: Funzione che alloca la matrice

```
1       void mallocMatrix(int ***matrix, int r, int c) {
2           int i;
3
4           int *cell = (int *)malloc(r*c*sizeof(int));
5           if (!cell){
```

```
6             exit(-1);
7         };
8
9         (*matrix) = (int **)malloc(r*sizeof(int*));
10        if (!(*matrix)) {
11            free(cell);
12            exit(-1);
13        }
14
15        for (i=0; i<r; i++){
16            (*matrix)[i] = &(cell[i*c]);
17        }
18    }
```

Listing 3.3: Funzione che inzializza la matrice e il vettore con numeri casuali

```
1   void generateRandomNumbers(/* args */){
2       int i, j;
3
4       srand(time(0));
5       for (i=0; i<dimsMatrix[0]; i++){
6           for (j=0; j<dimsMatrix[1]; j++){
7               matrix[i][j] = rand()%10;
8           }
9       }
10
11      for (i = 0; i<dimVector; i++){
12          vector[i] = rand()%10;
13      }
14  }
```

Dopo aver inizializzato la matrice e il vettore, viene calcolato il prodotto matrice-vettore tramite la funzione `int* computeMatrixPerVector()` che restituisce il puntatore al vettore risultante.

```
1   int* computeMatrixPerVector(int **matrix, int *vector, int dimsMatrix[]){
2   int i,j;
3   int *result;
4
5   //n.righe uguale alla dimensione del vettore risultante
6   result = (int *)malloc(dimsMatrix[0]*sizeof(int));
7
8   #pragma omp parallel for default(none)
9           shared(dimsMatrix, matrix, vector, result) private(i,j)
10  for (i = 0; i<dimsMatrix[0]; i++){
11      for (j = 0; j<dimsMatrix[1]; j++){
12          result[i] += matrix[i][j]*vector[j];
13      }
14
15  }
16  return result;
17 }
```

La parallelizzazione è gestita da OpenMP tramite la direttiva

```
1   #pragma omp parallel for
```

che indica al compilatore che il ciclo `for` dovrebbe essere parallelizzato. Ogni iterazione del ciclo è assegnata a diversi thread[1]. La direttiva agisce sul primo ciclo for che scorre le righe, quindi è in questo punto che vari blocchi di righe vengono assegnati a diversi thread. Una volta assegnate, ogni thread calcola il prodotto matrice vettore.

Alla fine, tutto lo spazio allocato per la matrice e il vettore viene deallocato tramite la funzione `free()` di C per i vettori e `freeMatrix()` per la matrice.

```
1   int freeMatrix(int ***array) {
2       free(&((*array)[0][0]));
3       free(*array);
4
5       return 0;
6   }
```

[1] in questo caso con thread indichiamo i processori

Capitolo 4

Descrizione dei test

I test sono stati effettuati sullo *SCoPE Datacenter*, infrastruttura con risorse di calcolo e storage accessibili mediante paradigmi di calcolo distribuito. Il datacenter ha messo a disposizione 8 nodi DELL blade M600, ognuno dotato di 2 processori Intel Xeon quadcore. Le varie esecuzioni sono state lanciate in batch tramite *Portable Batch System* (PBS). Tra i test rientra anche un caso d'uso, nel quale è stato utilizzato un PBS più semplice e dei log più verbosi per rendere chiaro il più possibile il funzionamento del programma. Il PBS tipo utilizzato per l'esempio d'uso è questo:

Listing 4.1: Script PBS per l'esempio d'uso

```
1  #!/bin/bash
2
3  #PBS -q studenti
4  #PBS -l nodes=1:ppn=8
5  #PBS -o $PROJECT.out
6  #PBS -e $PROJECT.err
7
8  echo "Job is running on node(s):"
9  cat $PBS_NODEFILE
10 PBS_O_WORKDIR=$PBS_O_HOME/$PROJECT
11 echo PBS: qsub is running on $PBS_O_HOST
12 echo PBS: originating queue is $PBS_O_QUEUE
13 echo PBS: executing queue is $PBS_QUEUE
14 echo PBS: working directory is $PBS_O_WORKDIR
15 echo PBS: execution mode is $PBS_ENVIRONMENT
16 echo PBS: job identifier is $PBS_JOBID
17 echo PBS: job name is $PBS_JOBNAME
18 echo PBS: node file is $PBS_NODEFILE
19
20 export PSC_OMP_AFFINITY=TRUE
21 gcc -fopenmp -lgomp -o $PBS_O_WORKDIR/$PROJECT $PBS_O_WORKDIR/$PROJECT.c
22
23 $PBS_O_WORKDIR/$PROJECT -t $T -r $R -c $C -v $V
```

e viene chiamato da questo comando

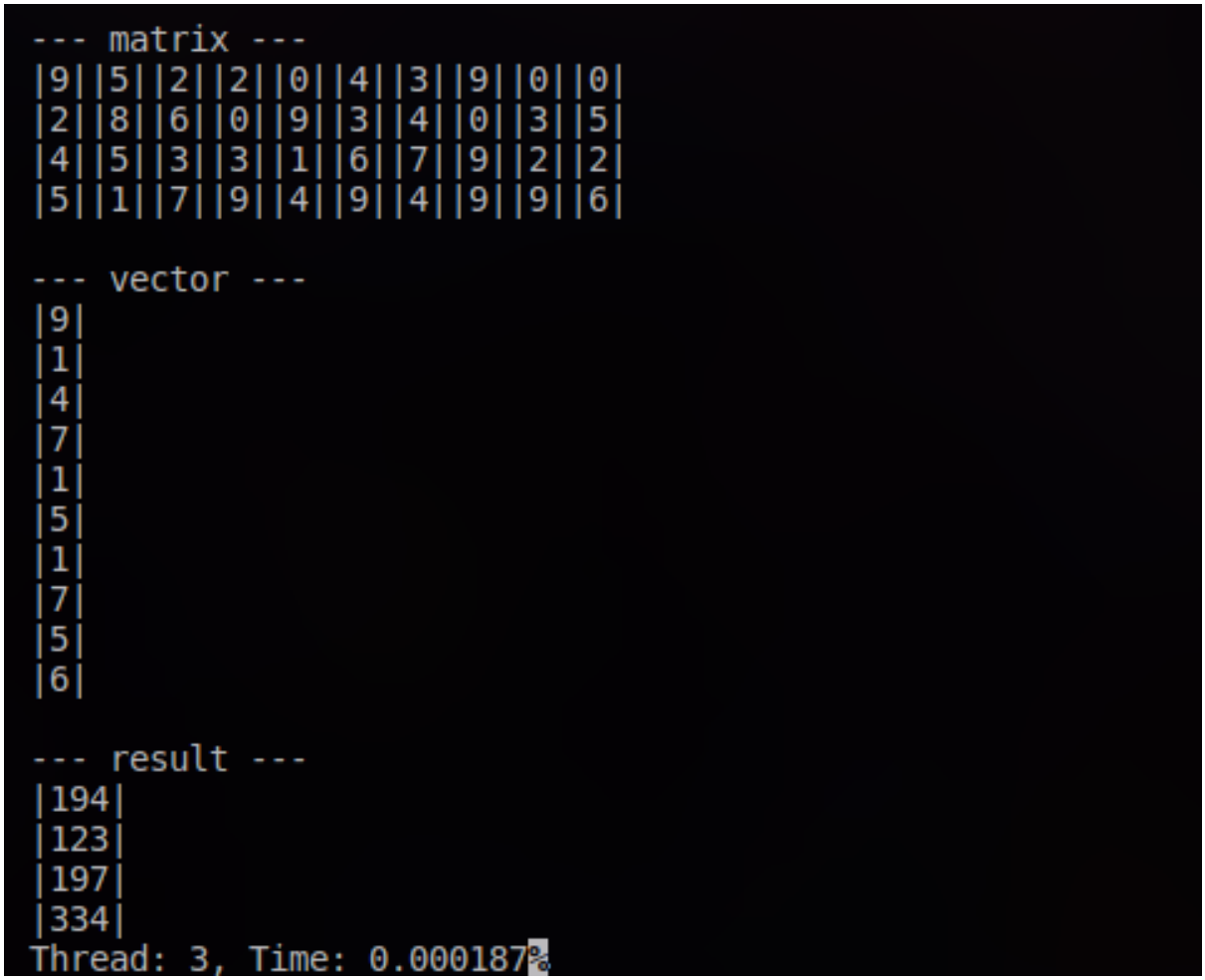
```
1  qsub -v PROJECT="<nome>",T=<threads number>,V=<vector size>,
2         R=<rows number>,C=<columns number> script.pbs
```

in cui `PROJECT` è una variabile che prende il nome del progetto, `T` è il numero di thread da inizializzare nella parte parallela, `V` è la dimensione del vettore da moltiplicare, `R` e `C` sono le dimensioni della matrice,

Un esempio d'uso per verificare la correttezza dell'algoritmo è descritto sotto. Sia numero di thread uguale a 3, un vettore di dimensione 10 e una matrice 4x10 allora il comando `qsub` con i precedenti parametri

```
qsub -v PROJECT="esempio",T=3,V=10,R=4,C=10 script.pbs
```

restituisce un file `esempio.out` in cui l'output è



Capitolo 5

Analisi delle performance

Per eseguire l'analisi, sono stati presi dieci tempi d'esecuzione dell'algoritmo di risoluzione e ne è stata fatta la media, in modo da avere un risultato con un margine d'errore più basso. Questo non significa che i dati presi sono corretti al 100%. I test sono stati eseguiti per studiare il tempo impiegato dall'algoritmo al variare dei processori con in aggiunta il calcolo e l'analisi dello speed-up e dell'efficienza. Per rendere più robusto lo studio, l'algoritmo è stato eseguito con diverse dimensioni. Per rendere il tutto automatizzato, è stato implementato uno script PBS che, oltre a definire le direttive PBS come quello precedentemente presentato, calcola automaticamente tutti i dati necessari e li fornisce in output tramite una tabella.

Listing 5.1: Script PBS usato per il calcolo delle performance

```
#Direttive PBS e compilazione uguali

#Incrementa il numero processori
for (( p=1; p<=8; p++ ))
do
    sum=0 #Somma di 10 tempi
    for (( i=0; i<10; i++ ))
    do
        # Prende il valore del tempo totale di un esecuzione
        # Il programma stampa il tempo
        tmp=$($PBS_O_WORKDIR/$PROJECT -t $p -v $V -r $R -c $C)
        #echo $tmp
        #Somma dei 10 valori
        sum=$(echo "$sum $tmp" | awk '{printf("%.5f", $1+$2)}')
    done
    average=$(echo "$sum" | awk '{printf("%.6f", $1/10)}')

    if [ $p -eq 1 ]; then
        #T(1) settato quando ottengo il tempo di 1 prcoessore
        t1=$average
        speedup=1
        efficiency=1
    else
        tp=$average
        #Calcolo speed-up T(1)/T(P)
        speedup=$(echo "$t1 $tp" | awk '{printf("%.6f", $1/$2)}')
        #Calcolo efficienza S(P)/P
        efficiency=$(echo "$speedup $p" | awk '{printf("%.6f", $1/$2)}')
    fi
done
```

Lo speed-up misura la riduzione del tempo di esecuzione di un algoritmo paral- lelo rispetto all'esecuzione dello stesso algoritmo su un solo processore (sequen- ziale) ed è calcolabile come

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

Mentre l'efficienza misura quanto l'algoritmo sfrutta il parallelismo del calcolatore e si calcola con

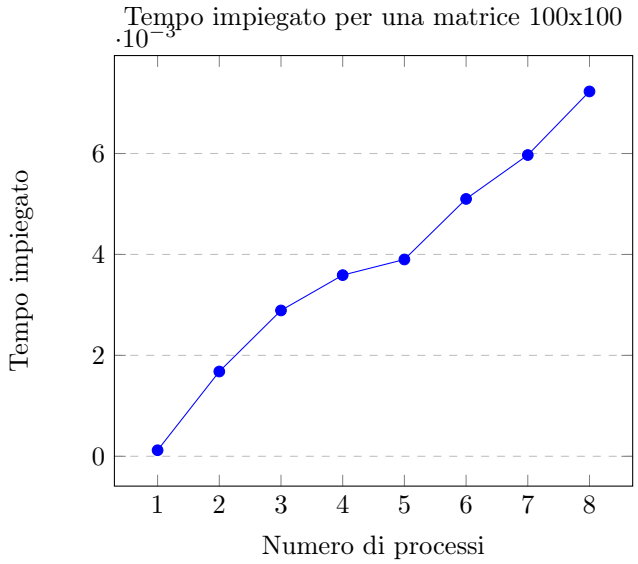
$$E(p) = \frac{S(p)}{p}$$

5.1 Analisi su matrice 100x100

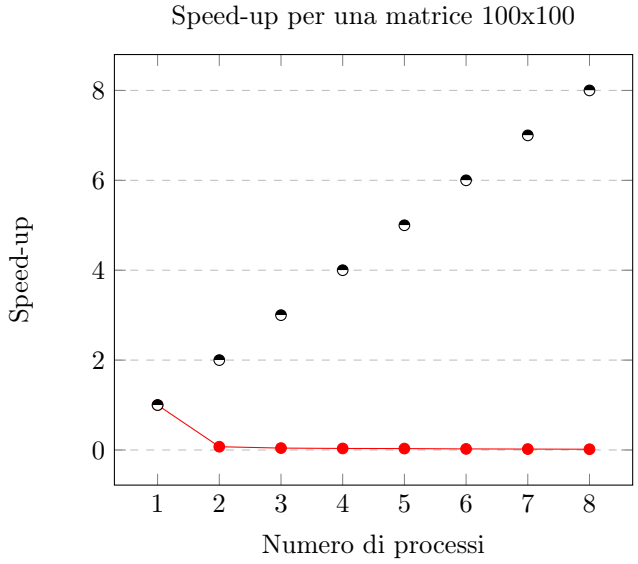
Questa è la tabella da cui sono stati generati i grafici per l'analisi su una matrice 100x100 e un vettore di dimensione 100 al variare del numero di processori.

p	T(p)	S(p)	E(p)
1	0.00012	1.00000	1.00000
2	0.00168	0.07266	0.03633
3	0.00289	0.04224	0.01408
4	0.00359	0.03402	0.00851
5	0.00390	0.03129	0.00626
6	0.00510	0.02391	0.00398
7	0.00597	0.02042	0.00292
8	0.00723	0.01687	0.00211

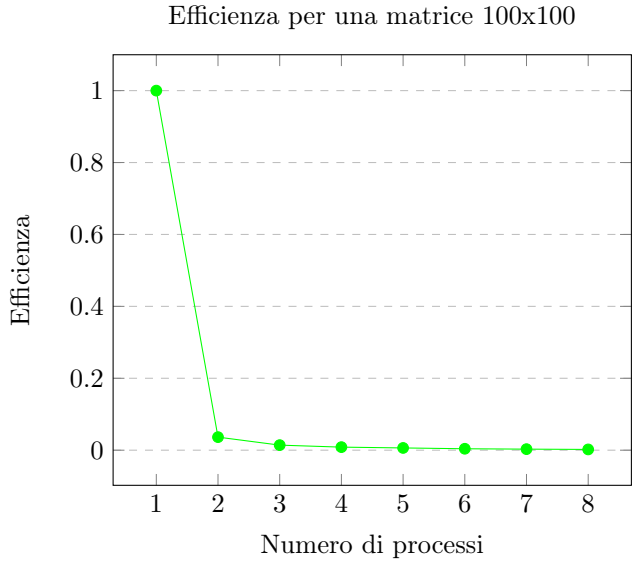
Il grafico sottostante ha un andamento anomalo in quanto all'aumentare dei processori il tempo impiegato dall'algoritmo per risolvere il problema aumenta. Gli elementi sono troppo pochi per avere un miglioramento nel caso dell'algoritmo parallelizzato, tanto è che l'algoritmo sequenziale è quello che impiega meno tempo. Per così pochi elementi non conviene parallelizzare l'algoritmo perchè in questo modo si spreca più tempo per le operazioni che servono a parallelizzare che per le operazioni che servono effetivamente a risolvere il problema. In questo grafico i tempi sono rappresentati come $s \cdot 10^{-3}$.



Quanto visto sopra, si riflette anche nello speed-up che diminuisce vertiginosamente invece di aumentare. Già solo con 2 processori si può notare il calo drastico dello speed-up. Aumentando i processori, lo speed-up tende a 0. Inoltre si può notare ancora di più il risultato molto scarso di questo algoritmo, confrontando lo speed-up con lo speed-up ideale (cerchietti neri e bianchi). Questa differenza si rifletterà soprattutto nell'efficienza.



Come spiegato precedentemente, anche l'efficienza segue il trend visto fino ad adesso. Seppure l'aspettativa fosse che l'efficienza diminuisse, di sicuro non era preventivato che ci fosse un calo così drastico.

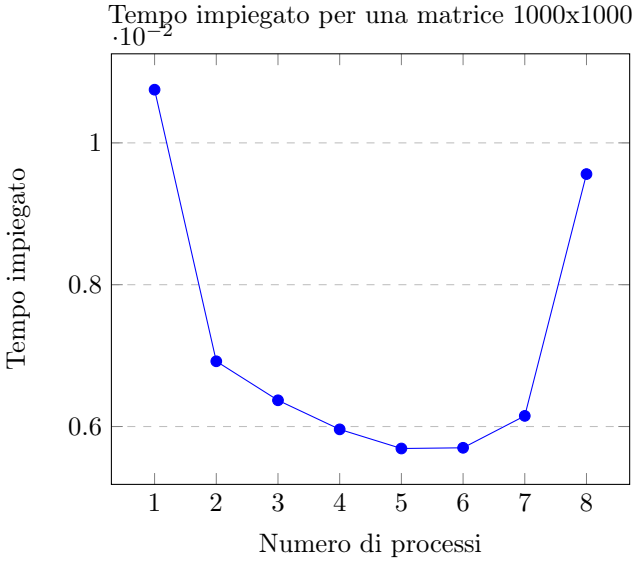


5.2 Analisi su matrice 1000x1000

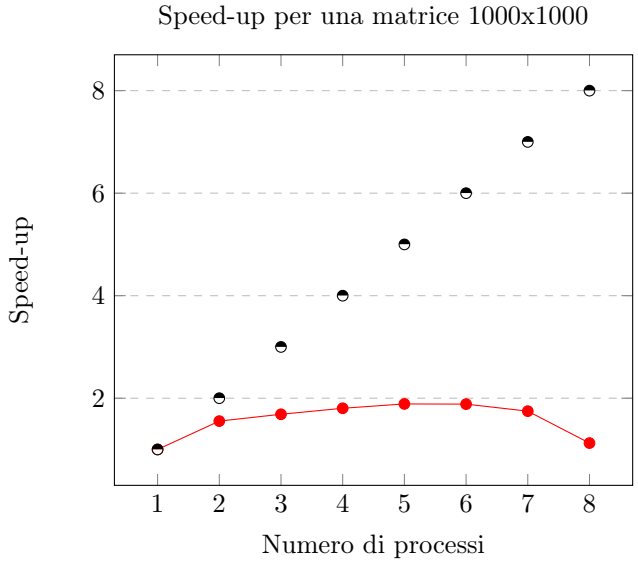
Questa è la tabella da cui sono stati generati i grafici per l'analisi su una matrice 1000x1000 e un vettore di dimensione 1000 al variare del numero di processori.

p	T(p)	S(p)	E(p)
1	0.01075	1.00000	1.00000
2	0.00692	1.55332	0.77666
3	0.00637	1.68665	0.56222
4	0.00655	1.64007	0.41002
5	0.00569	1.88811	0.37762
6	0.00570	1.88447	0.31408
7	0.00615	1.74724	0.24960
8	0.00956	1.12449	0.14056

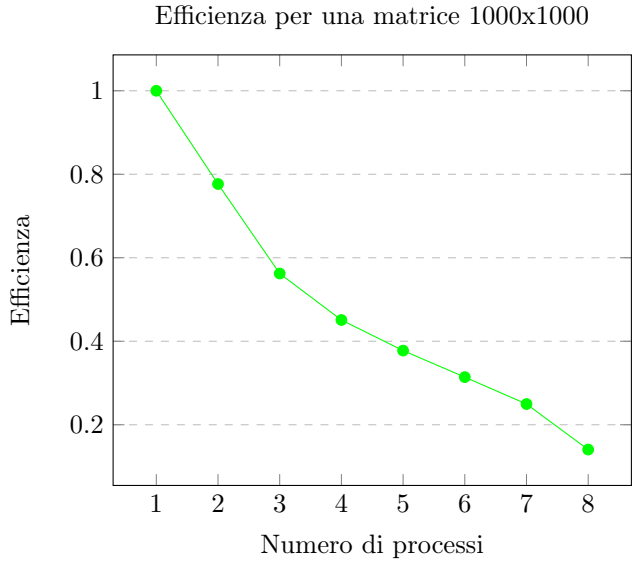
Il grafico sottostante è dividibile in 2 parti: da 1 a 5 processori il tempo impiegato diminuisce, mentre da 6 a 8 il tempo impiegato aumenta. Il problema è lo stesso descritto precedentemente ovvero la scarsa quantità di dati. Fino a 5 processori i dati sono abbastanza per sfruttare la paralelizzazione e avere un guadagno di tempo. Invece da 6 processori fino a 8, le operazioni per rendere l'algoritmo parallelo impattano molto di più poichè i processori sono di più. I tempi sono rappresentati come $s \cdot 10^{-2}$.



Nel grafico sottostante è possibile osservare che lo speed-up aumenta ma è molto distante dallo speed-up ideale. Ma, come da aspettative dal grafico precedente, lo speed-up inizia a diminuire dopo che vengono impiegati più di 5 processori. Una differenza così marcata tra speed-up ideale e speed-up effettivo è già un indicatore della scarsa efficienza dell'algoritmo, poichè, di solito, la differenza è molto minore.



Anche in questo caso l'efficienza cala molto. A differenza del caso precedente, non cala drasticamente in quanto all'aumentare del numero di processori comunque il tempo impiegato è minore rispetto a un algoritmo sequenziale. Di certo questo è un risultato pessimo in quanto l'algoritmo diminuisce molto d'efficienza all'aumentare dei processori e non si mantiene "costante"[1].



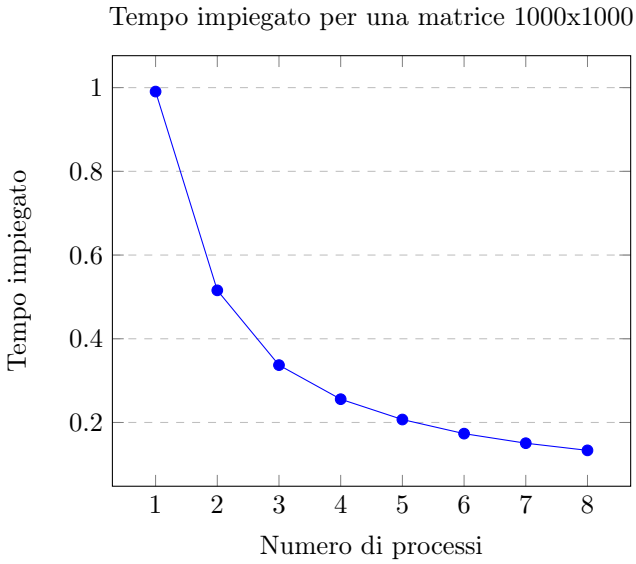
[1] inteso come stabile poichè è impossibile ottenere un andamento costante

5.3 Analisi su matrice 10000x10000

Questa è la tabella da cui sono stati generati i grafici per l'analisi su una matrice 10000x10000 e un vettore di dimensione 10000 al variare del numero di processori.

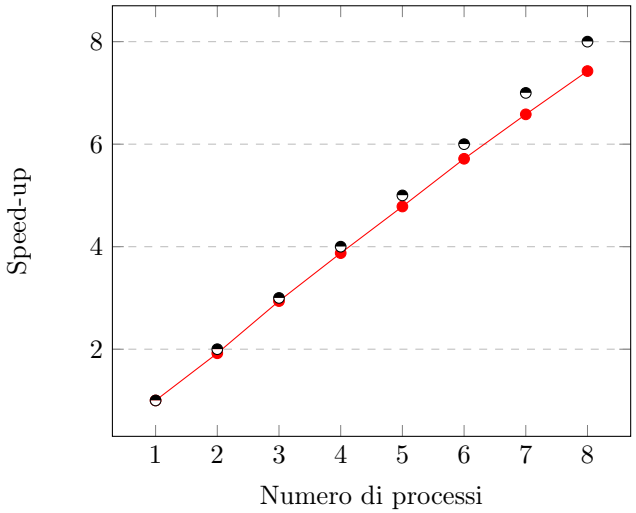
p	T(p)	S(p)	E(p)
1	0.99057	1.00000	1.00000
2	0.51568	1.92088	0.96044
3	0.33705	2.93896	0.97965
4	0.25565	3.87464	0.96866
5	0.20706	4.78394	0.95679
6	0.17333	5.71499	0.95250
7	0.15050	6.58189	0.94027
8	0.13339	7.42622	0.92828

Nel grafico sottostante, è possibile notare come in questo caso con una matrice da 10000x10000, l'algoritmo parallelo si esprima molto meglio. I tempi sono rappresentati come s. Da notare soprattutto il passaggio dall'algoritmo eseguito da un processore (quindi sequenziale) e lo stesso eseguito da due processori. La pendenza di questo passaggio evidenzia il netto miglioramento dell' algoritmo se eseguito in parallelo.



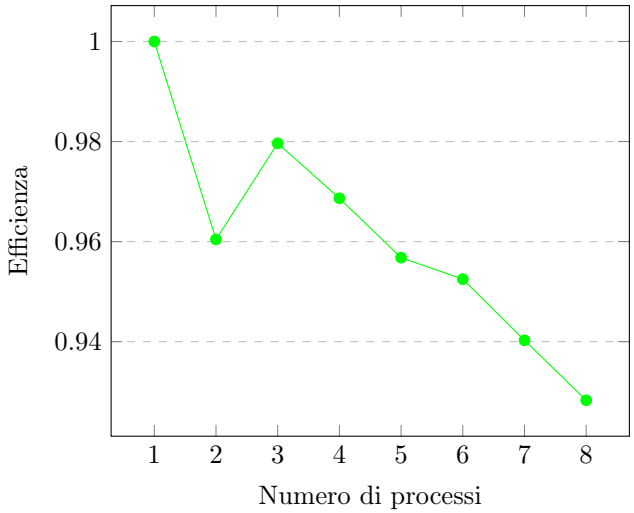
In questo caso è bene osservare come l'aumento dello speed-up sembra quasi lineare. Come descritto precedentemente, la differenza tra speed-up ideale e speed-up effettivo è minima ed è molto minore rispetto ai casi già presi in esame. Questo indica che rendere parallelo questo algoritmo porterà un miglioramento massiccio sia nel tempo impiegato sia nell'efficienza.

Speed-up per una matrice 10000x10000



Come da aspettative, all'aumentare dei processori l'efficienza diminuisce poiché la parte parallela diminuisce. Quindi il grafico segue la legge di Amdahl. Inaspettato è il comportamento dell'algoritmo con 2 processori poiché diminuisce di molto l'efficienza rispetto a 3 e 4 processori. Come detto all'inizio, i dati non sono corretti al 100% e, probabilmente, questa misurazione è stata falsata da un valore che ha invalidato il calcolo corretto della media.

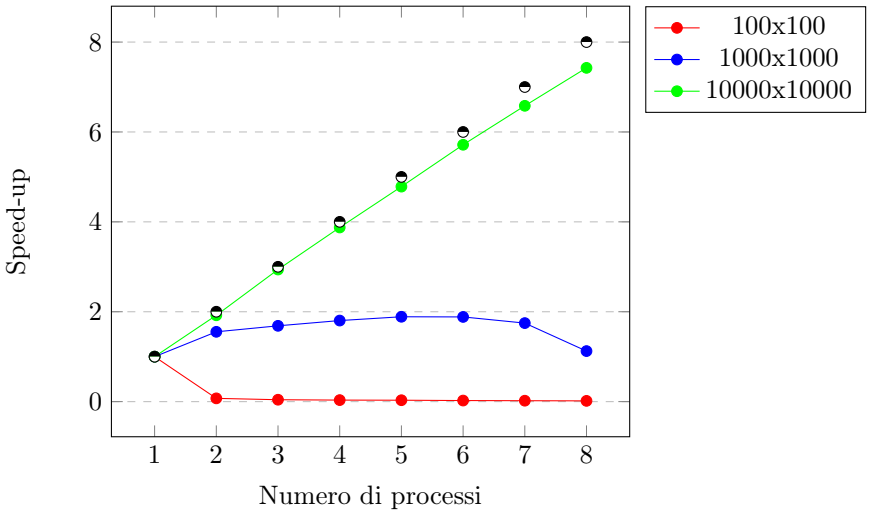
Efficienza per una matrice 10000x10000



5.4 Confronto speed-up e efficienza

Quest'ultima sezione è usata per rappresentare i grafici di confronto tra i 3 casi e rendere ancora più chiare le differenza descritte in questo capitolo.

Confronto speed-up tra i 3 casi



Confronto efficienza tra i 3 casi