

Calcolo Parallelo della Somma di Numeri Reali con MPI in Ambiente MIMD Distribuito

Professors:

prof. Giuliano Laccetti
prof. Valeria Mele

Author:

Oleksandr Sosovskyy N86003573

October 24, 2023

Contents

1	Introduzione	2
2	Descrizione del Problema	2
3	Implementazione	2
3.1	Definizione dei tipi personalizzati:	3
3.2	Funzione ‘initializeNumbers’	3
3.3	Funzione ‘saveResultsToCSV’	4
3.4	Input e Output	4
4	Strategie di Parallelizzazione	7
4.1	Strategia di Comunicazione 0	7
4.2	Strategia di Comunicazione 1	7
4.3	Strategia di Comunicazione 3	8
5	Risultati e Analisi	9
5.1	Descrizione dei test svolti	9
5.2	Analisi delle prestazioni	11
5.3	Strategia 1	11
5.4	Strategia 2	14
5.5	Strategia 3	16

1 Introduzione

Questo lavoro presenta lo sviluppo di un algoritmo di calcolo parallelo utilizzando la libreria MPI per calcolare la somma di numeri reali. L'algoritmo è progettato per operare in un ambiente MIMD a memoria distribuita. L'implementazione consente di gestire l'input dei numeri da sommare, generare numeri casuali quando necessario e applicare strategie di comunicazione specifiche. Sono analizzati i risultati relativi alle prestazioni, con una discussione sui tempi di esecuzione, il speed-up e l'efficienza ottenuti con diverse strategie. L'elaborato fornisce una visione completa dell'implementazione, con documentazione dettagliata, codice sorgente e esempi di utilizzo significativi per ciascun tipo di input previsto.

Il calcolo parallelo è stato condotto su un cluster composto da 8 nodi, ciascuno dei quali dispone di 8 core di elaborazione. Ogni nodo è un server Dell PowerEdge M600, equipaggiato con le seguenti specifiche:

- Due processori quad-core Intel Xeon E5410 a 2.33 GHz, con architettura a 64 bit.
- 8 GB di memoria RAM.
- Due dischi SATA da 80 GB configurati in modalità RAID1.
- Due schede Gigabit Ethernet configurate in bonding.

2 Descrizione del Problema

Il contesto del presente report riguarda il calcolo della somma di un insieme di numeri reali. Questo problema verrà affrontato studiando tre strategie di parallelizzazione.

Il problema può essere definito come segue:

- Input:
 - Un insieme di N numeri x_1, x_2, \dots, x_N .
 - Il tipo dei numeri passati (interi, reali, double, short, ecc.).
 - Un numero P di processori utilizzati.
 - Una strategia di comunicazione specifica da utilizzare per la parallelizzazione.
- Output: La somma dei numeri reali nell'insieme, contenuto in uno o più processori contemporaneamente.

Per affrontare il problema in un ambiente di calcolo parallelo, è necessario suddividere il carico di lavoro tra più processori. L'obiettivo è parallelizzare l'operazione di somma in modo da ottenere un miglioramento delle prestazioni rispetto a un calcolo sequenziale.

3 Implementazione

L'implementazione del programma è scritta in linguaggio C e fa uso della libreria MPI (Message Passing Interface) per la programmazione parallela. Il programma è progettato per calcolare la somma di un vettore di dati, diviso tra diversi processori, utilizzando diverse strategie di comunicazione.

3.1 Definizione dei tipi personalizzati:

Sono definiti tipi personalizzati utilizzando le macro 'MIO_TIPO' e 'MIO_TIPO_MPI'. Questi tipi vengono utilizzati per specificare il tipo di dati con cui si lavora nel programma. È possibile modificarli in base alle esigenze, ad esempio, definendo 'MIO_TIPO' come 'int' o 'float' e 'MIO_TIPO_MPI' come 'MPI_INT' o 'MPI_FLOAT' a seconda del tipo di dati desiderato.

```

1 // Definizione dei tipi personalizzati tramite macro, fare attenzione a
  sostituire entrambi con tipi sensati come:
2 // - Per il tipo short: #define MIO_TIPO short e #define MIO_TIPO_MPI
  MPI_SHORT
3 // - Per il tipo int: #define MIO_TIPO int e #define MIO_TIPO_MPI
  MPI_INT
4 // - Per il tipo float: #define MIO_TIPO float e #define MIO_TIPO_MPI
  MPI_FLOAT
5 // - Per il tipo double: #define MIO_TIPO double e #define MIO_TIPO_MPI
  MPI_DOUBLE
6 #define MIO_TIPO double
7 #define MIO_TIPO_MPI MPI_DOUBLE

```

Listing 1: Definizione dei tipi personalizzati per generalizzare la comunicazione tramite MPI senza modificare il codice

3.2 Funzione 'initializeNumbers'

Questa funzione è stata modificata per inizializzare il vettore di dati in base al valore di 'n' fornito in input. Se 'n' è minore o uguale a 20, il programma legge i dati da un file chiamato "numbers.txt". Altrimenti, genera numeri casuali nel range specificato.

```

1 // Questa funzione inizializza l'array di numeri x in base a vari scenari
2 void initializeNumbers(int n, MIO_TIPO *x, int menum) {
3     int i;
4     if (menum == 0) {
5         if (n <= 20) {
6             // Se n è minore o uguale a 20, leggi i dati da un file "numbers
              .txt"
7             FILE *file = fopen("numbers.txt", "r");
8             if (file == NULL) {
9                 // Se si verifica un errore nell'apertura del file, stampa
              un messaggio di errore e termina il programma
10                fprintf(stderr, "Errore nell'apertura del file 'numbers.txt
                  '\n");
11                MPI_Abort(MPI_COMM_WORLD, 1);
12                return;
13            }
14            for (i = 0; i < n; i++) {
15                if (fscanf(file, "%lf", &x[i]) != 1) {
16                    // Se si verifica un errore nella lettura dei dati dal
              file, stampa un messaggio di errore, chiudi il file e termina il
              programma
17                    fprintf(stderr, "Errore nella lettura dei numeri dal
                  file 'numbers.txt'\n");
18                    fclose(file);
19                    MPI_Abort(MPI_COMM_WORLD, 1);
20                    return;
21                }

```

```

22     }
23     fclose(file);
24 } else {
25     // Genera numeri casuali per n maggiore di 20
26     srand((unsigned int)time(NULL));
27     for (i = 0; i < n; i++) {
28         x[i] = (MIO_TIPO)(rand() % 100); // Genera numeri casuali
        nel range da 0 a 99 (regolabile)
29     }
30 }
31 }
32 }

```

Listing 2: Funzione initializeNumbers

3.3 Funzione 'saveResultsToCSV'

Questa funzione è stata aggiunta per salvare i dati dei tempi di esecuzione, speed-up ed efficienza in un file CSV chiamato "results.csv". I dati vengono scritti in formato CSV per consentire analisi e valutazioni delle prestazioni in modo agevole tramite tool quali Python ed Excel, ad esempio.

La funzione accetta i parametri 'n', 'p', 'strategy', e 'time' e scrive i dati nel file CSV.

```

1 // Questa funzione salva i risultati delle prestazioni in un file CSV
2 void saveResultsToCSV(int n, int p, int strategy, double time) {
3     // Apri il file CSV in modalita append
4     FILE *file = fopen("results.csv", "a");
5     if (file == NULL) {
6         // Se si verifica un errore nell'apertura del file, stampa un
        messaggio di errore e termina il programma
7         printf("Errore nell'apertura del file 'results.csv'.\n");
8         exit(1);
9     }
10
11     // Scrivi i dati dei risultati nel file CSV in formato "n, p, strategy,
    time"
12     fprintf(file, "%d, %d, %d, %e\n", n, p, strategy, time);
13
14     // Chiudi il file dopo aver scritto i dati
15     fclose(file);
16 }

```

Listing 3: funzione saveResultsToCSV

3.4 Input e Output

Il programma accetta due argomenti in input: **n** e **strategy**. **n** rappresenta la dimensione del vettore da sommare, mentre **strategy** specifica la strategia di comunicazione da utilizzare (0, 1, 2, o 3). in particolare:

- argc:
 - argc < 2, errore:
 - argc = 2, esecuzione con strategia 0 (ovvero sequenziale):

```
C:\Users\alless\source\repos\MPI11\Debug>mpiexec -n 8 MPI11.exe
Usage: MPI11.exe n [strategy]
Please provide the value of 'n'. You can also specify an optional 'strategy' argument (0, 1, 2, or 3).
```

Figure 1: $argc=1$

```
C:\Users\alless\source\repos\MPI11\Debug>mpiexec -n 8 MPI11.exe 100
Sono il processo 0: Somma totale=100.000000 in tempo 6.000046e-07 secondi
```

Figure 2: $argc=2$

```
C:\Users\alless\source\repos\MPI11\Debug>mpiexec -n 8 MPI11.exe 100 2
Sono il processo 0: Somma totale=100.000000 in tempo 3.528000e-04 secondi
```

Figure 3: $argc=3$

- $argc = 3$, esecuzione corretta:
- $argc > 3$, ignora gli argomenti successivi al terzo (la strategia):

```
C:\Users\alless\source\repos\MPI11\Debug>mpiexec -n 8 MPI11.exe 100 2 3 5 5 gda geja ejhs
Sono il processo 0: Somma totale=100.000000 in tempo 4.488000e-04 secondi
```

Figure 4: $argc > 3$

- n :

- $n < 1$, errore:

```
C:\Users\alless\source\repos\MPI11\Debug>mpiexec -n 8 MPI11.exe -34
Invalid value of 'n'. Please provide a positive integer within the range of representable integers.
```

Figure 5: $n=-34$

- $1 \leq n \leq 20$, esecuzione corretta, con lettura dal file (poiché i numeri sono scritti nel file da 1 in ordine crescente, si può verificare la validità della somma con la formula di Gauss):
 - $20 \leq n \leq \text{INT_MAX}$, esecuzione corretta, con generazione di numeri casuali:
 - $n > \text{INT_MAX}$, errore:
- strategy:

```
C:\Users\alless\source\repos\MPI11\Debug>mpiexec -n 8 MPI11.exe 5 0
```

Sono il processo 0: Somma totale=15.000000 in tempo 2.000015e-07 secondi

Figure 6: $n=5$

```
C:\Users\alless\source\repos\MPI11\Debug>mpiexec -n 8 MPI11.exe 1000 3
```

Sono il processo 0: Somma totale=48255.000000 in tempo 5.014000e-04 secondi

Figure 7: $n=1000$

```
C:\Users\alexs\source\repos\MPI11\Debug>mpiexec -n 8 MPI11.exe 30000000000000000000000000301034315135135135135 3
Memory allocation of array x failed.
```

Figure 8: *n molto grande*

- strategy < 0, errore:

```
C:\Users\alless\source\repos\MPI11\Debug>mpiexec -n 8 MPI11.exe 100 -3
Invalid strategy: -3
```

Figure 9: $strategy=-3$

- $0 \leq \text{strategy} \leq 3$, esecuzione corretta:

```
C:\Users\alless\source\repos\MPI11\Debug>mpiexec -n 8 MPI11.exe 1000 3
```

Sono il processo 0: Somma totale=1000.000000 in tempo 3.709000e-04 secondi

Figure 10: $strategy=3$

- strategy > 3, errore:

```
C:\Users\alless\source\repos\MPI11\Debug>mpiexec -n 8 MPI11.exe 1000 5
Invalid strategy: 5
```

Figure 11: *strategy=5*

Inoltre, se il numero di processori indicati non è 1 o una potenza di 2 e la strategia scelta è 2 o 3, allora il programma fa uno switch automatico alla strategia 1.

```
C:\Users\alless\source\repos\MPI11\Debug>mpiexec -n 7 MPI11.exe 100000000 3
Strategy 3 requires P to be a power of 2 or 1. Automatically switching to strategy 1.
Sono il processo 0: Somma totale=4946916211.000000 in tempo 3.845580e-02 secondi
```

Figure 12: *cambio dalla strategia 3 alla strategia 1, poiché 7 processori indicati*

4 Strategie di Parallelizzazione

In questa sezione, esamineremo le strategie di parallelizzazione implementate nell'algoritmo per il calcolo parallelo della somma di numeri reali. Abbiamo progettato e implementato tre diverse strategie di comunicazione, denominate I, II e III, ognuna delle quali è stata applicata in base a specifiche condizioni. Discuteremo quando e come ognuna di queste strategie è stata utilizzata.

4.1 Strategia di Comunicazione 0

La "Strategia di Comunicazione 0" rappresenta l'approccio sequenziale in cui il software esegue il calcolo utilizzando un unico processore. Questa strategia è utilizzata come riferimento per valutare il miglioramento delle prestazioni ottenuto con le strategie di parallelizzazione.

```
1 // utilizza la strategia 0, ovvero il calcolo della somma in modo
  sequenziale
2   if (strategy == 0) {
3       if (menum == 0) {
4           for (i = 0; i < n; i++) {
5               sum += x[i];
6           }
7       }
8   }
```

Listing 4: *Codice della strategia 0*

4.2 Strategia di Comunicazione 1

Strategia di Comunicazione 1

La "Strategia di Comunicazione 1" coinvolge l'uso di somme parziali per ottenere il risultato finale. Ogni processore diverso da zero calcola una somma parziale dei dati locali e invia questo valore al processo radice (Processore 0). Il processo radice aggrega le somme parziali ricevute per ottenere la somma totale dei dati.

```
1 // Utilizza la strategia 1
2   else if (strategy == 1) {
3       if (menum == 0) {
4           // Processore radice (0) calcola la somma cumulativa delle somme
          parziali
5           for (i = 0; i < nloc; i++) {
6               sum += x[i];
7           }
8           // Processore radice riceve le somme parziali dai processori non
          zero
9           for (i = 1; i < nproc; i++) {
10              tag = 300 + i;
11              MPI_Recv(&sum_parz, 1, MIO_TIPO_MPI, i, tag, MPI_COMM_WORLD,
                  &status);
12              sum += sum_parz;
```

```

13     }
14     } else {
15         // Processori non zero inviano le somme locali al processore
radice
16         for (i = 0; i < nloc; i++) {
17             sum += xloc[i];
18         }
19         tag = menum + 300;
20         MPI_Send(&sum, 1, MIO_TIPO_MPI, 0, tag, MPI_COMM_WORLD);
21     }

```

Listing 5: Codice della strategia 1**Strategia di Comunicazione 2**

La "Strategia di Comunicazione 2" prevede che ad ogni passo di iterazione successivo, il numero di processori coinvolti nel calcolo si dimezza secondo un andamento logaritmico. Questo significa che inizialmente, ogni processore calcola somme parziali locali. In seguito, i risultati parziali vengono combinati in coppie in uno dei due processori, procedendo così via via finché le somme parziali non arrivano tutte al processore 0 che calcola quindi la somma totale. Il risultato principale è una riduzione notevole del numero di comunicazioni necessarie rispetto alla "Strategia di Comunicazione 1".

```

1 // Utilizza la strategia 2
2 } else if (strategy == 2) {
3     // Ogni processore somma i dati locali
4     for (i = 0; i < nloc; i++) {
5         sum += xloc[i];
6     }
7     for (i = 0; i < log2nproc; i++) {
8         if ((menum % (1 << i)) == 0) {
9             if ((menum % (1 << (i + 1))) == 0) {
10                // Chi riceve
11                partner = menum + (1 << i);
12                if (partner < nproc) {
13                    recv_tag = 300 + partner;
14                    MPI_Recv(&sum_parz, 1, MIO_TIPO_MPI, partner,
recv_tag, MPI_COMM_WORLD, &status);
15                    sum += sum_parz;
16                }
17            } else {
18                // Chi spedisce
19                partner = menum - (1 << i);
20                send_tag = 300 + menum;
21                MPI_Send(&sum, 1, MIO_TIPO_MPI, partner, send_tag,
MPI_COMM_WORLD);
22            }
23        }
24    }
}

```

Listing 6: Codice della strategia 2**4.3 Strategia di Comunicazione 3**

La "Strategia di Comunicazione 3" è un'approccio simile alla "Strategia 2", ma con una differenza chiave. In entrambe le strategie, i dati vengono suddivisi tra i processori in modo equo, e ciascun processore calcola somme parziali dei dati assegnati. Tuttavia, la "Strategia 3" si distingue per la gestione dei risultati intermedi.

In questa strategia, dopo aver calcolato le somme parziali, i processori si scambiano tra loro i dati intermedi, invece di avere un processore che semplicemente riceve e l'altro che manda i propri dati locali. Il risultato finale è che ogni processore possiede la somma totale dei numeri passati in input.

```

1 // Utilizza la strategia 3
2 } else if(strategy == 3) {
3     // Ogni processore somma i dati locali
4     for (i = 0; i < nloc; i++) {
5         sum += xloc[i];
6     }
7
8     for (i = 0; i < log2nproc; i++) {
9         if ((menum % (1 << i + 1)) < (1 << i)) {
10             partner = menum + (1 << i);
11             // Spedisci a menum + 2^i
12             send_tag = 300 + menum;
13             MPI_Send(&sum, 1, MIO_TIPO_MPI, partner, send_tag,
14 MPI_COMM_WORLD);
15             // Ricevi da menum + 2^i
16             recv_tag = 300 + partner;
17             MPI_Recv(&sum_parz, 1, MIO_TIPO_MPI, partner, recv_tag,
18 MPI_COMM_WORLD, &status);
19             sum += sum_parz;
20         } else {
21             partner = menum - (1 << i);
22             sum_tmp = sum;
23             // Ricevi da menum - 2^i
24             recv_tag = 300 + partner;
25             MPI_Recv(&sum_parz, 1, MIO_TIPO_MPI, partner, recv_tag,
26 MPI_COMM_WORLD, &status);
27             sum += sum_parz;
28             // Spedisci a menum - 2^i
29             send_tag = 300 + menum;
30             MPI_Send(&sum_tmp, 1, MIO_TIPO_MPI, partner, send_tag,
31 MPI_COMM_WORLD);
32         }
33     }
34 }

```

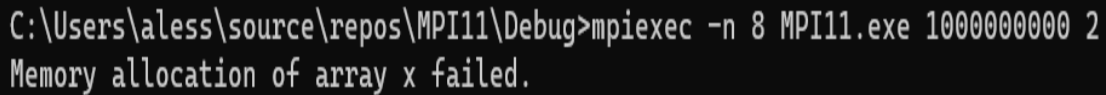
Listing 7: Codice della strategia 3

5 Risultati e Analisi

In questa sezione, sono presentati i risultati per $N=10000000$ dell'implementazione delle diverse strategie di comunicazione nell'algoritmo di calcolo parallelo per la somma di numeri reali. Si è scelto questo valore perché sufficientemente grande da rendere poco significativi i ritardi dovuti ai tempi di comunicazione, ma sufficientemente piccolo da ottenere abbastanza test senza sovraccaricare il cluster e la coda dei job pbs.

5.1 Descrizione dei test svolti

Sono stati condotti diversi test per valutare le prestazioni del programma in diverse condizioni. Tuttavia, è importante notare che il cluster SCOPE dispone di una quantità limitata di memoria RAM



```
C:\Users\alless\source\repos\MPI11\Debug>mpirun -n 8 MPI11.exe 1000000000 2
Memory allocation of array x failed.
```

Figure 13: per n pari a un miliardo la memoria non viene allocata dinamicamente

per ogni core, pari a 8 GB. Ciò implica che l'allocazione di risorse per problemi di dimensioni molto elevate, ad esempio con $N=10^9$, può portare a problemi di esaurimento della memoria.

Di conseguenza, il numero di test eseguiti è stato limitato in base alle dimensioni del problema e alle risorse disponibili. I test sono stati progettati in modo da essere gestibili entro i limiti di memoria del cluster, assicurando al contempo risultati accurati e tempi di esecuzione ragionevoli.

In particolare questo è il ciclo di operazioni svolte, che genera 5200 test:

```
1 # Elenco dei valori da testare per P e STRATEGY
2 P_VALUES=(1 2 3 4 5 6 7 8)
3 STRATEGY_VALUES=(0 1 2 3)
4
5 # Inizializza il valore di N
6
7 # Compila il programma C
8 echo "Compilazione di $PBS_O_WORKDIR/progetto1.c"
9 /usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS_O_WORKDIR/progetto1
   $PBS_O_WORKDIR/progetto1.c
10
11 # Loop su STRATEGY e valori di P
12 for STRATEGY in "${STRATEGY_VALUES[@]"; do
13   for P in "${P_VALUES[@]"; do
14     # Controlla se P è uguale a 1 o una potenza di 2 quando STRATEGY è 2 o 3
15     if [[ "$STRATEGY" == "2" || "$STRATEGY" == "3" ]]; then
16       if [[ $P -ne 1 && $((P & (P - 1))) -ne 0 ]]; then
17         continue
18       fi
19     fi
20     # Controlla se P > 1 e STRATEGY è 0, quindi interrompi il ciclo P
21     if [[ "$STRATEGY" == "0" && $P -gt 1 ]]; then
22       break # Esci dal ciclo P
23     fi
24     N=100
25     # Ciclo fino a raggiungere il numero massimo di interi
26     while [ $N -lt 1000000000 ]; do
27       N_tmp=$N
28       N_tmp_max=$((N * 16))
29       while [ $N_tmp -le $N_tmp_max ]; do
30         echo "Esecuzione 10 volte di $PBS_O_WORKDIR/progetto1 con STRATEGY=
$STRATEGY, P=$P, N=$N_tmp"
31         for ((i = 1; i <= 10; i++)); do
32           /usr/lib64/openmpi/1.4-gcc/bin/mpirun -machinefile hostlist -np
$P $PBS_O_WORKDIR/progetto1 $N_tmp $STRATEGY
33         done
34         N_tmp=$((N_tmp * 2)) # Moltiplica N per 2 per la prossima
iterazione
35       done
36       N=$((N * 10)) # Moltiplica N per 10 per la prossima iterazione
37     done
```

```

38 done
39 done

```

Listing 8: parte dello script pbs che compila e avvia l'eseguibile per diversi valori di P e $STRATEGY$

5.2 Analisi delle prestazioni

Per valutare le prestazioni delle strategie di comunicazione, abbiamo misurato i tempi di esecuzione dell'algoritmo su un cluster di calcolo parallelo. I tempi di esecuzione sono stati registrati per diverse dimensioni del problema e per un numero variabile di processori. I risultati mostreranno come ciascuna strategia si comporta in relazione alla dimensione del problema e al numero di processori coinvolti.

5.3 Strategia 1

p	$T(p)$	$S(p)$	$Oh(p)$	$E(p)$
1	41736,102	1	0	1
2	21400,349	1,950253335	1064,596	0,975126667
3	14307,19	2,917141801	1185,468	0,9723806
4	10731,699	3,889048882	1190,694	0,972262221
5	8663,1776	4,817643586	1579,786	0,963528717
6	7237,5061	5,766641357	1688,9346	0,961106893
7	24285,437	1,718564998	128261,957	0,245509285
8	5460,8822	7,642739849	1950,9556	0,955342481

Figure 14: Strategia 1: speed-up, overhead ed efficienza

Analisi della strategia 1:

- **T(P) (Tempo di Esecuzione):** Il tempo di esecuzione cala significativamente con l'incremento del numero di processori fino al caso con 7 processori. Si è notato sul cluster un brusco calo di prestazioni specificatamente nel caso di 7 processori. Riprovando i test su una macchina locale questo calo di prestazioni non è stato percepito. Dunque si attribuisce il problema a qualche configurazione di comunicazione tra i nodi del cluster.

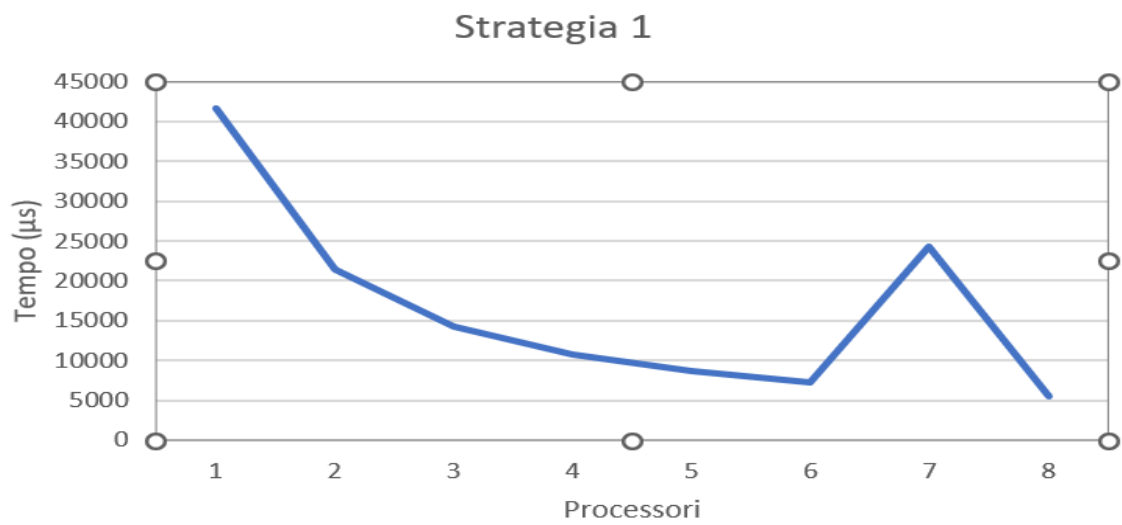


Figure 15: *Strategia 1: tempo di esecuzione*

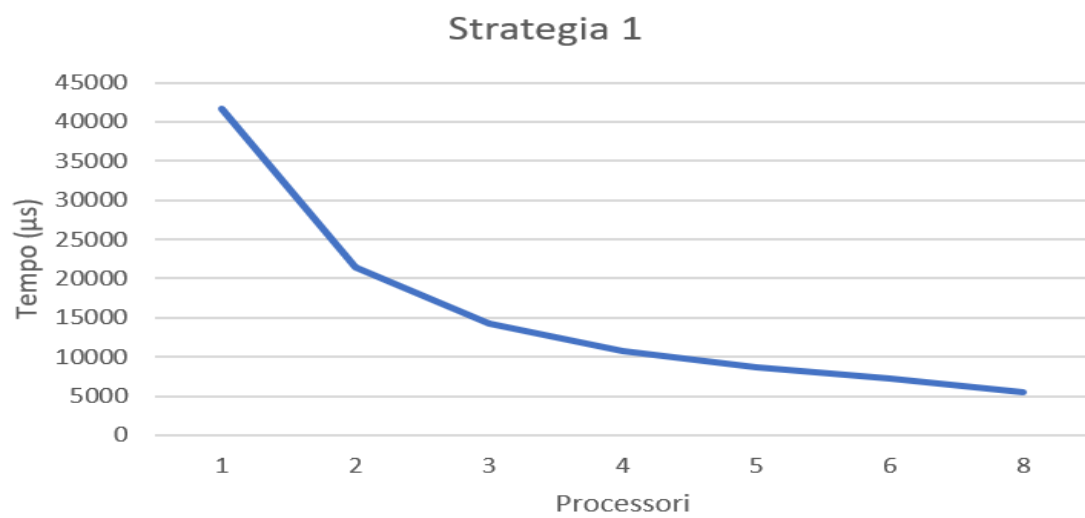
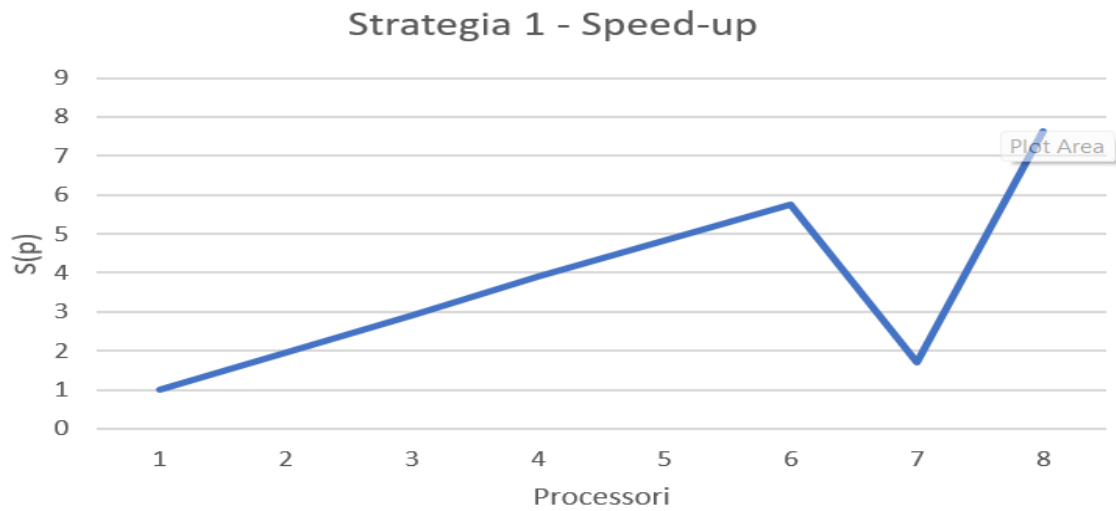
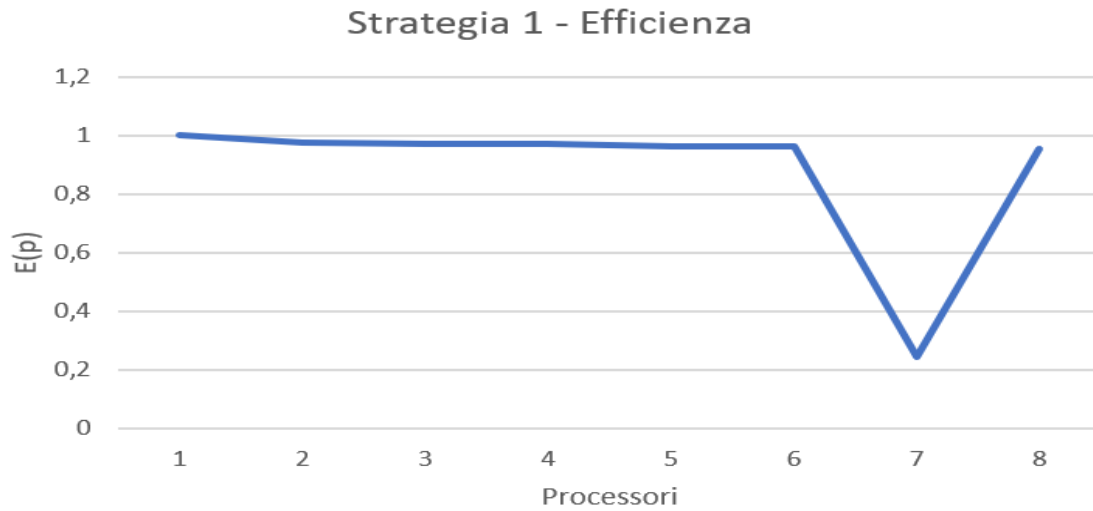


Figure 16: *Strategia 1: tempo di esecuzione senza considerare il caso dei 7 processori*

- **S(P) (Speed-Up):** Inizialmente, lo speed-up cresce in maniera costante, a meno del caso dei 7 processori.

Figure 17: *Strategia 1: speed-up*

- **E(P) (Efficienza):** L'efficienza cala in maniera poco significativa, a meno del caso di 7 processori.

Figure 18: *Strategia 1: efficienza*

I valori scalati sono ottimi, e si mantengono su valori pressoché costanti, dato il numero massimo di processori utilizzati (ovvero 8):

n	p	T(n,p)	S(n,p)	Oh(n,p)	E(n,p)
10000000	1	43735,291	1	0	1
20000000	2	42822,782	1,958271273	41910,273	0,979135637
40000000	4	41833,859	3,826096321	123600,145	0,95652408
80000000	8	40830,885	7,468730001	282911,789	0,93359125

Figure 19: *Strategia 1: speed-up scalato, overhead scalato ed efficienza scalata*

5.4 Strategia 2

p	T(p)	S(p)	Oh(p)	E(p)
1	42746,52	1	0	1
2	21426,773	1,995005034	107,026	0,997502517
4	10752,821	3,975377252	264,764	0,993844313
8	5420,1602	7,886578703	614,7616	0,985822338

Figure 20: Strategia 2: speed-up, overhead ed efficienza

Analisi della strategia 2:

- **T(P) (Tempo di Esecuzione)**: Il grafico è molto simile al tempo di esecuzione della strategia 1 nel caso in cui si tolgano i valori per il caso con 7 processori.

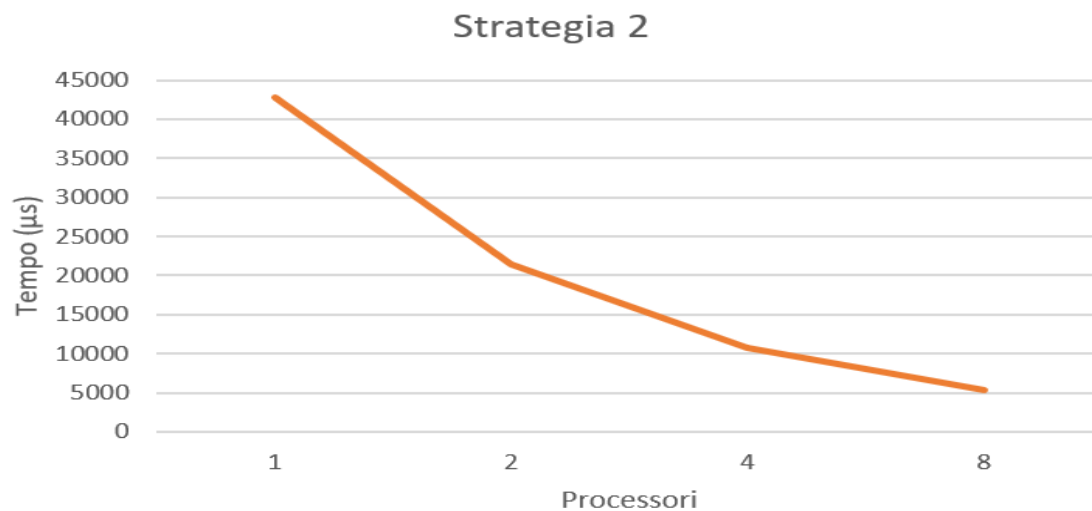
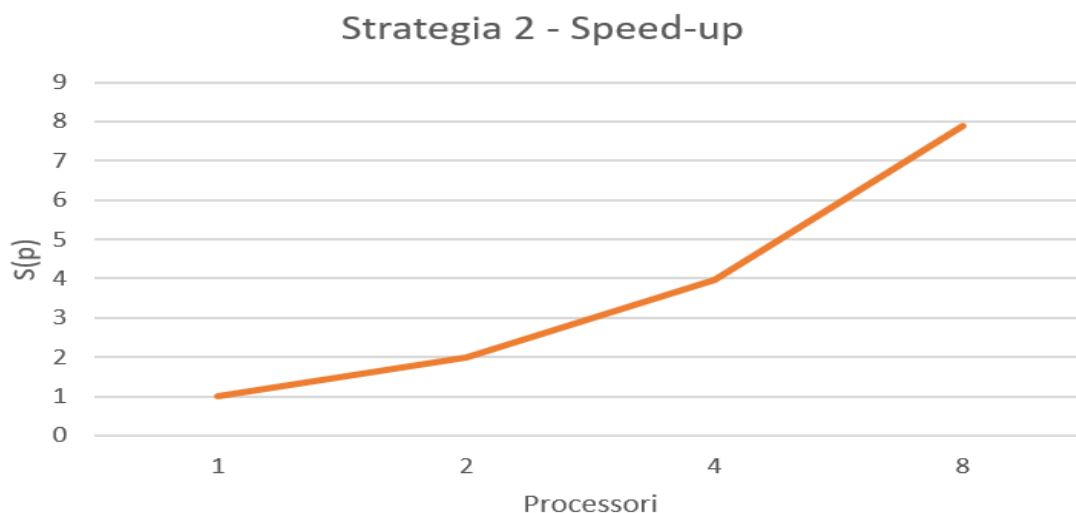
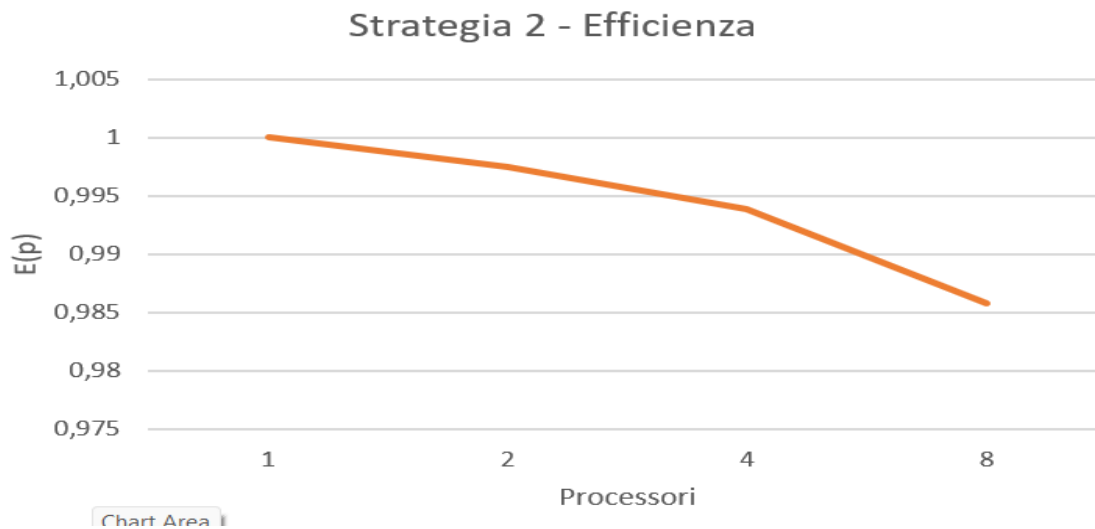


Figure 21: Strategia 2: tempo di esecuzione

- **S(P) (Speed-Up)**: Lo speed-up migliora considerevolmente all'aumentare di P.

**Figure 22:** *Strategia 2: speed-up*

- **E(P) (Efficienza):** L'efficienza dell'algoritmo è notevolmente alta con l'aumento di P. Ciò suggerisce che la Strategia 2 sfrutta in modo efficiente le risorse disponibili all'aumentare del numero di processori.

**Figure 23:** *Strategia 2: efficienza*

I valori scalati corrispondenti sono altrettanto molto positivi in quanto riflettono l'andamento dei parametri non scalati:

n	p	T(n,p)	S(n,p)	Oh(n,p)	E(n,p)
10000000	1	43746,52	1	0	1
20000000	2	42854,428	1,959215407	41962,336	0,979607704
40000000	4	41814,208	3,82331742	123510,312	0,955829355
80000000	8	40796,08333	7,460448663	282622,1467	0,932556083

Figure 24: *Strategia 2: speed-up scalato, overhead scalato ed efficienza scalata*

5.5 Strategia 3

Strategia 3	p	T(p)	S(p)	Oh(p)	E(p)
10000000	1	42616,296	1	0	1
	2	21422,171	1,989354674	228,046	0,994677337
	4	10770,178	3,95687945	464,416	0,989219862
	8	5438,4709	7,836080542	891,4712	0,979510068

Figure 25: *Strategia 3: speed-up, overhead ed efficienza*

Analisi della strategia 3:

- **T(P) (Tempo di Esecuzione):** I tempi sono molto simili a quelli della strategia 2.

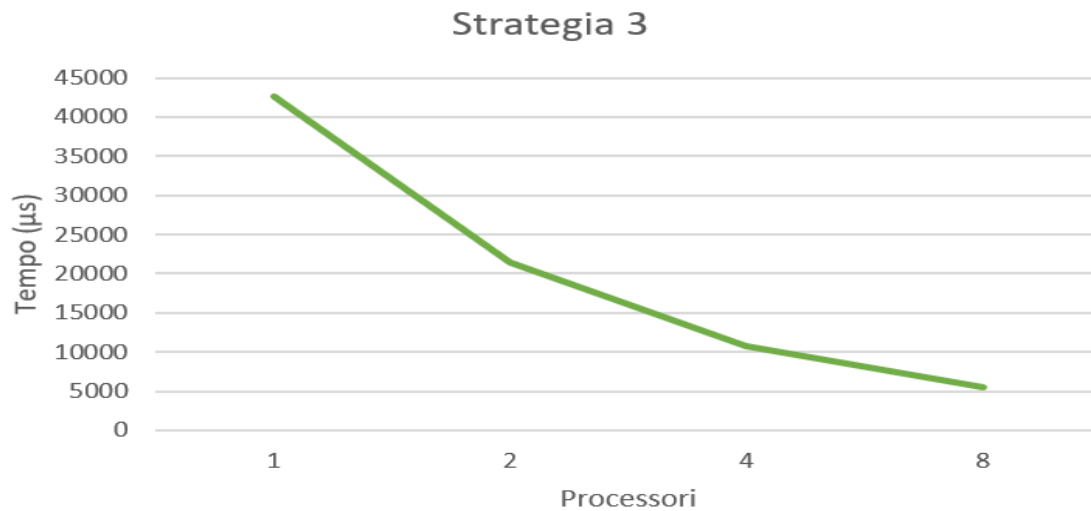
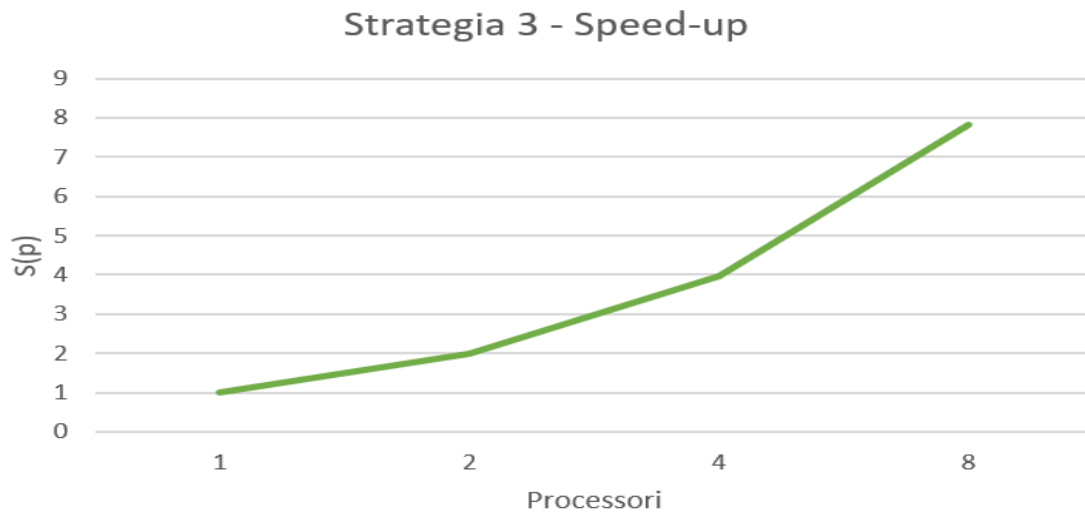
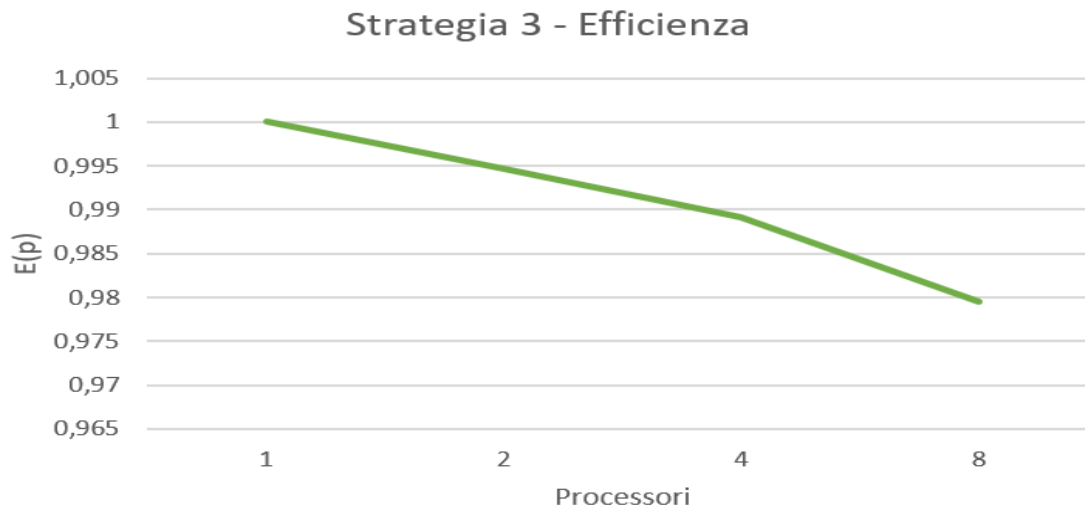


Figure 26: *Strategia 3: tempo di esecuzione*

- **S(P) (Speed-Up):** Lo speed-up aumenta cresce linearmente.

Figure 27: *Strategia 3: speed-up*

- **E(P) (Efficienza):** L'efficienza è notevolmente alta.

Figure 28: *Strategia 3: efficienza*

Di seguito i parametri scalati, che, come per gli altri parametri, sono molto simili a quelli ricavati per la strategia 2:

n	p	$T(n,p)$	$S(n,p)$	$Oh(n,p)$	$E(n,p)$
10000000	1	43616,296	1	0	1
20000000	2	42770,648	1,961223301	41925	0,98061165
40000000	4	41788,982	3,832419149	123539,632	0,958104787
80000000	8	40971,49375	7,514896496	284155,654	0,939362062

Figure 29: *Strategia 3: speed-up scalato, overhead scalato ed efficienza scalata*