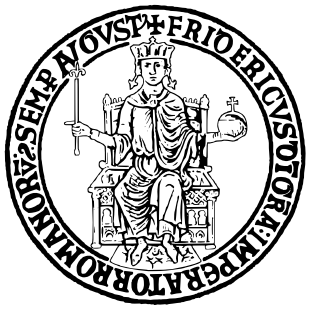
Università degli Studi di Napoli Federico II



Scuola Politecnica e delle Scienze di Base

Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione

Corso di Laurea Magistrale in Informatica

Corso di Parallel and Distributed Computing

Calcolo della somma di N numeri

Relatore
Prof. Giuliano Laccetti
Prof.ssa Valeria Mele

Candidato
Fabrizio Vitale N97/0449
Giovanni Falcone N97/0451
Luigi Mangiacapra N97/0454

Anno Accademico 2023-2024

Indice

Capitolo 1

Descrizione del problema

Il goal del problema è calcolare la somma di N numeri in ambiente *parallelo* su architettura **MIMD** (**Multiple Instruction Multiple Data**) a memoria distribuita, utilizzando la libreria `MPI` in linguaggio C. In particolare, verranno adoperate 3 strategie differenti per effettuare tale somma e tramite dei grafici verranno mostrate le differenze tra le 3 strategie in termini di **tempo di esecuzione**, **speed up** ed **efficienza**.

L'algoritmo prende in input (da terminale) gli N numeri da sommare:

- se $N \leq 20$, allora verranno presi in input gli N valori forniti al momento del lancio del programma.
- se $N > 20$, gli elementi da sommare verranno generati randomicamente. Ovviamente se $N > 20$, i valori inseriti da terminali non verranno presi in considerazione.

Capitolo 2

Descrizione algoritmo

In questo capitolo descriveremo la struttura del programma, ovvero come abbiamo organizzato i vari file *.c*, *.h* e *.pbs*, le strategie che questo applicherà e le relative problematiche.

2.1 Struttura del programma

Innanzitutto descriviamo come abbiamo suddiviso i vari file e cosa ciascuno di essi contiene. La struttura del programma è così suddivisa:

```
/
├── Main.c
├── Strategy.c
├── Strategy.h
├── Utils.h
├── Utils.c
├── job-script.pbs
└── sum.pbs
```

dove

- `Main.c` è file principale che contiene il main del programma: richiama le funzioni della libreria `MPI` per le inizializzazioni e le opportune funzioni dai file header per applicare le diverse strategie.
- `Strategy.c` è il file che contiene le funzioni relative alle strategie da usare.
- `Strategy.h` è il file che contiene i diversi prototipi.
- `Utils.c` è il file che contiene le funzioni necessarie al controllo dei parametri e altre funzioni di utilità come il riempimento dell'array, della distribuzioni degli operandi, ecc
- `Utils.h` è il file che contiene i prototipi.
- `sum.pbs`: il pbs utilizzato per testare le funzionalità del programma (non "blocca" il cluster)

- `job-script.pbs`: il pbs utilizzato per raccogliere i tempi da analizzare (usato per "bloccare" il cluster)

2.2 Strategie

Definiamo, dunque, le strategie che vogliamo applicare.

2.2.1 Strategia I

Tramite questa strategia ciascun processore calcola la propria somma parziale e invia tale somma ad un processore prestabilito (nel nostro caso il processore P_0), il quale alla fine conterrà la somma totale.

Al passo 0 ciascun processore P_i effettua la propria somma locale S_i. Al passo 1 il processore P_1 invia la propria somma parziale al processore P_0 che provvede a fare la somma tra la propria somma parziale e quella inviata da P_1. E così via.

In generale all'$i-$esimo passo, il processore P_i invia la propria somma parziale al processore P_0 che provvede a fare la somma:

$$S_{0,i-1} = S_{0,i-1} + S_i$$

Uno schema di questa strategia viene mostrato in Figura 2.1 con $i = 4$.

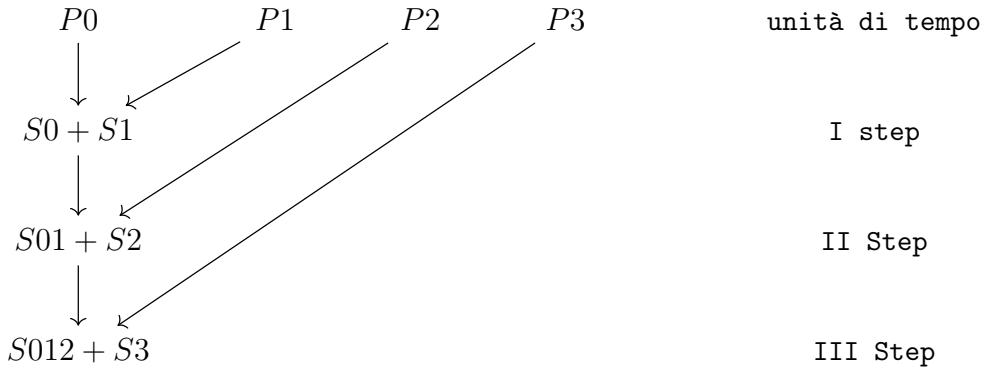


Figura 2.1: Schema funzionamento I strategia

L'algoritmo viene mostrato nel Listing 2.1. Il suo prototipo è descritto nella Sezione 3.2.1.

```
1 int first_strategy(int menum, int nproc, int sum){
2     int sum_parz = 0;
3     int tag;
4     MPI_Status status;
5
6     if(menum == 0){
7         for(int i = 1; i < nproc; i++){
8             tag = 80 + i;
9             MPI_Recv(&sum_parz, 1, MPI_INT, i, tag, MPI_COMM_WORLD,
```

```
10              &status);
11              sum += sum_parz;
12          }
13      }else{
14          tag = menum + 80;
15          MPI_Send(&sum, 1, MPI_INT, 0, tag, MPI_COMM_WORLD);
16      }
17
18      return sum;
19  }
```

Listing 2.1: Algoritmo strategia I

2.2.2 Strategia II

Con questa strategia ciascuna coppia di processori comunica tra loro la propria somma parziale. Anche in questo caso, la somma totale si trova in unico processore prestabilito (che nel nostro caso è sempre P_0). Inoltre, poichè le comunicazioni avvengono a coppie è necessario che il numero di processori sia una potenza di 2.

Uno schema di questa strategia viene mostrato in Figura 2.2 con $i = 4$.

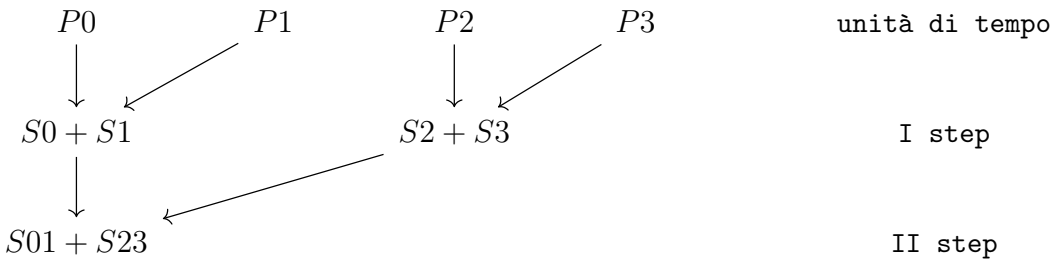


Figura 2.2: Schema funzionamento II strategia

L'algoritmo viene mostrato nel Listing 2.2. Il suo prototipo è descritto nella Sezione 3.2.2.

```
1  int second_strategy(int menum, int logNproc, int *array, int sum){
2      int sum_parz = 0;
3      int tag;
4      int partner;
5
6      int power_for_partecipation;
7      int does_processor_partecipate;
8
9      int power_for_communication;
10     int does_processor_receive;
11
12     MPI_Status status;
13
14     for(int i = 0; i < logNproc; i++){
15         power_for_partecipation =  array[i];
```

```
16          does_processor_partecipate = (menum % power_for_partecipation) ↩
        == 0;
17
18          if(does_processor_partecipate){
19              power_for_communication = array[i + 1];
20              does_processor_receive = (menum % power_for_communication) ↩
        == 0;
21
22              if (does_processor_receive){
23                  partner = menum + power_for_partecipation;
24                  tag = 60 + i;
25                  MPI_Recv(&sum_parz, 1, MPI_INT, partner, tag,
26                  MPI_COMM_WORLD, &status);
27                  sum += sum_parz;
28              } else{
29                  partner = menum - power_for_partecipation;
30                  tag = 60 + i;
31                  MPI_Send(&sum, 1, MPI_INT, partner, tag,
32                  MPI_COMM_WORLD);
33              }
34          }
35      }
36
37      return sum;
38      }
39  }
```

Listing 2.2: Algoritmo strategia II

2.2.3 Strategia III

La strategia III è identica alla II eccetto che tutte le coppie inviano e ricevano la propria somma parziale in modo che alla fine tutti i processori abbiano la somma totale.

Uno schema di questa strategia viene mostrato in Figura 2.3 con $i = 4$.

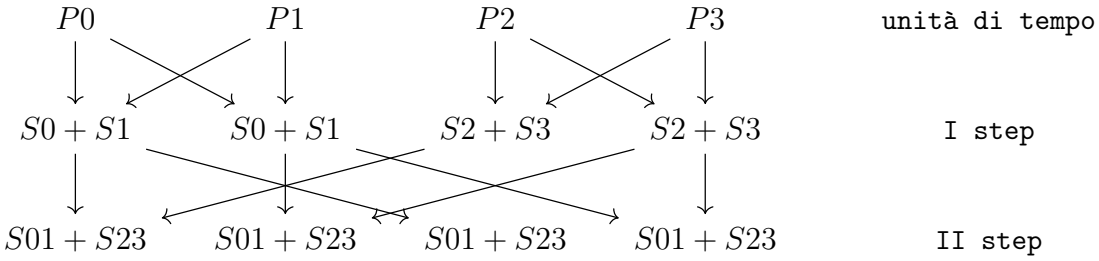


Figura 2.3: Schema funzionamento III strategia

L'algoritmo viene mostrato nel Listing 2.3. Il suo prototipo è descritto nella Sezione 3.2.3.

```
1  int third_strategy(int menum, int logNproc, int *array, int sum){
2      int partner;
```

```
    int send_tag;
    int recv_tag;
    int sum_parz;
    MPI_Status status;

    sum_parz = 0;
    for(int i = 0; i < logNproc; i++){
        if ((menum % array[i + 1]) < array[i]) {
            partner = menum + array[i];
            send_tag = 40 + i;
            recv_tag = 40 + i;

            // Invia la somma locale al processo partner
            MPI_Send(&sum, 1, MPI_INT, partner, send_tag,
            MPI_COMM_WORLD);

            // Ricevi la somma del processo partner
            MPI_Recv(&sum_parz, 1, MPI_INT, partner, recv_tag,
            MPI_COMM_WORLD, &status);

            // Aggiorna la variabile 'sum' con la somma ricevuta
            sum += sum_parz;
        } else {
            partner = menum - array[i];
            send_tag = 40 + i;
            recv_tag = 40 + i;

            // Ricevi la somma dal processo partner
            MPI_Recv(&sum_parz, 1, MPI_INT, partner, recv_tag,
            MPI_COMM_WORLD, &status);

            // Invia la somma locale al processo partner
            MPI_Send(&sum, 1, MPI_INT, partner, send_tag,
            MPI_COMM_WORLD);
            sum += sum_parz;
        }
    }

    return sum;
}
```

Listing 2.3: Algoritmo strategia III

Capitolo 3

Descrizione routine

In questo capitolo vengono trattate le funzioni utilizzate per lo scopo del problema: nella sezione 3.1 discuteremo delle funzioni della libreria `MPI` utilizzate, mentre nella sezione 3.2 discuteremo delle funzioni di supporto utilizzate per risolvere il nostro problema.

3.1 Funzioni *MPI* utilizzate

3.1.1 *MPI_Init*

```
int MPI_Init(int *argc, char ***argv)
```

Descrizione: Inizializza l'ambiente di esecuzione MPI.

Parametri di input:

- `argc`: puntatore al numero di parametri
- `argv`: vettore di argomenti

Errors: Restituisce il codice `MPI_SUCCESS` in caso di successo, `MPI_ERR_OTHER` altrimenti.

3.1.2 *MPI_Comm_rank*

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

Descrizione: Determina l'identificativo del processo chiamante nel comunicatore.

Parametri di input:

- `comm`: comunicatore

Parametri di output:

- `rank`: id del processo chiamante nel gruppo `comm`

Errors: Restituisce `MPI_SUCCESS` se la routine termina con successo, `MPI_ERR_COMM` altrimenti (comunicatiore invalido, e.g comunicatore NULL).

3.1.3 *MPI_Comm_size*

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

Descrizione: Restituisce la dimensione del gruppo associato al comunicatore.

Parametri di input:

- `comm`: comunicatore

Parametri di output:

- `size`: numero di processori nel gruppo `comm`.

Errors: Restituisce `MPI_SUCCESS` se la routine termina con successo, `MPI_ERR_COMM` in caso di comunicatore non valido o `MPI_ERR_ARG` se un argomento non è valido e non è identificato da una classe di errore specificata.

3.1.4 *MPI_Bcast*

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype,
    int root, MPI_Comm comm)
```

Descrizione: Invia un messaggio in broadcast dal processo con id `root` a tutti gli altri processi del gruppo.

Parametri di input:

- `buffer`: puntatore al buffer
- `count`: numero di elementi del buffer
- `datatype`: tipo di dato del buffer
- `root`: id del processo da cui ricevere
- `comm`: communicator

Errors: Restituisce `MPI_SUCCESS` se la routine termina con successo o un codice di errore altrimenti (`MPI_ERR_COMM`, `MPI_ERR_COUNT`, `MPI_ERR_TYPE`, `MPI_ERR_BUFFER`, `MPI_ERR_ROOT`).

3.1.5 *MPI_send*

```
int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int ↩
    dest, int tag, MPI_Comm comm)
```

Descrizione: Invia un messaggio in modalità standard e bloccante.

Parametri di input:

- `buf`: puntatore al buffer
- `count`: numero di elementi del buffer
- `datatype`: tipo di dato del buffer
- `dest`: identificativo del processo destinatario
- `tag`: identificativo del messaggio
- `comm`: comunicatore

Errors: Restituisce `MPI_SUCCESS` se la routine termina con successo o un codice di errore altrimenti (`MPI_ERR_COMM`, `MPI_ERR_COUNT`, `MPI_ERR_TYPE`, `MPI_ERR_BUFFER`, `MPI_ERR_RANK`).

3.1.6 *MPI_Recv*

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
    int source, int tag, MPI_Comm comm, MPI_Status *status)
```

Descrizione: Funzione di ricezione, bloccante: ritorna solo dopo che il buffer di ricezione contiene il nuovo messaggio ricevuto.

Parametri di input:

- `count`: numero di elementi del buffer
- `datatype`: tipo di dato del buffer
- `source`: identificativo del processo mittente
- `tag`: identificativo del messaggio
- `comm`: comunicatore

Parametri di output:

- `buf`: puntatore al buffer di ricezione
- `status`: racchiude informazioni sulla ricezione del messaggio

Errors: Restituisce `MPI_SUCCESS` se la routine termina con successo o un codice di errore altrimenti (`MPI_ERR_COMM`, `MPI_ERR_COUNT`, `MPI_ERR_TYPE`, `MPI_ERR_BUFFER`, `MPI_ERR_RANK`).

3.1.7 *MPI_Wtime*

```
double MPI_Wtime()
```

Descrizione: Restituisce un tempo in secondi.

Output: tempo in secondi (double).

3.1.8 *MPI_Reduce*

```
int MPI_Reduce(const void *sendbuf, void *recvbuf, int count,
               MPI_Datatype datatype, MPI_Op op, int root,
               MPI_Comm comm)
```

Descrizione: esegue un'operazione di riduzione globale (come somma, massimo, AND logico, ecc.) su tutti i membri di un gruppo.

Parametri di input:

- `sendbuf`: puntatore al buffer
- `count`: numero di elementi del buffer
- `datatype`: tipo di dato del buffer
- `op`: operazione di riduzione (`MPI_MAX`, `MPI_MIN`, `MPI_SUM`, ...)
- `root`: identificativo del processo che visualizzerà il risultato
- `comm`: comunicatore

Parametri di output:

- `recvbuff`: puntatore al buffer di ricezione

Errors: Restituisce `MPI_SUCCESS` se la routine termina con successo o un codice di errore altrimenti (`MPI_ERR_COMM`, `MPI_ERR_COUNT`, `MPI_ERR_TYPE`, `MPI_ERR_BUFFER`).

3.1.9 *MPI_Barrier*

```
int MPI_Barrier(MPI_Comm comm)
```

Descrizione: Fornisce un meccanismo sincronizzare per tutti i processi del gruppo: ogni processo si blocca finchè tutti gli altri processi del gruppo non hanno eseguito anch'essi tale routine.

Parametri di input:

- `comm`: communicator

Errors: Restituisce `MPI_SUCCESS` se la routine termina con successo, `MPI_ERR_COMM` altrimenti.

3.1.10 *MPI_Finalize*

```
int MPI_Finalize()
```

Descrizione: Termina l'ambiente di esecuzione MPI. Tutti i processi devono chiamare questa routine prima di uscire. Il numero di processi in esecuzione dopo la chiamata di questa routine non è definito.

Errors: Restituisce solo `MPI_SUCCESS`

3.2 Funzioni *ausiliare* utilizzate

3.2.1 La funzione *first_strategy*

```
int first_strategy(int menum, int nproc, int sum)
```

Descrizione: Esegue la somma applicando la prima strategia.

Parametri di input:

- `menum`: l'identificativo del processo
- `nproc`: il numero di processori da utilizzare
- `sum`: la somma parziale fatta precedentemente da ciascun processore

Output: La somma totale.

3.2.2 La funzione *second_strategy*

```
int second_strategy(int menum, int nproc, int *array, int sum)
```

Descrizione: Esegue la somma applicando la seconda strategia.

Parametri di input:

- `menum`: l'identificativo del processo
- `nproc`: il numero di processori da utilizzare
- `array`: il vettore contenente le potenze di due per verificare chi deve partecipare alla comunicazione, e chi deve inviare/ricevere.
- `sum`: la somma parziale fatta precedentemente da ciascun processore

Output: La somma totale.

3.2.3 La funzione *third_strategy*

```
int second_strategy(int menum, int nproc, int *array, int sum)
```

Descrizione: Esegue la somma applicando la terza strategia.

Parametri di input:

- `menum`: l'identificativo del processo
- `nproc`: il numero di processori da utilizzare
- `array`: il vettore contenente le potenze di due per verificare chi deve partecipare alla comunicazione, e chi deve inviare/ricevere.
- `sum`: la somma parziale fatta precedentemente da ciascun processore

Output: La somma totale.

3.2.4 La funzione *check_if_inputs_are_valid*

```
int check_if_inputs_are_valid(int argc, int N, int strategy)
```

Descrizione: Verifica se i parametri passati in ingresso al programma sono quelli corretti. Più precisamente è richiesto che N, ossia il numero di valori nel caso in cui $N \leq 20$, sia uguale a $argc - 3$ (cioè solo i valori da sommare, in quanto vanno esclusi il nome del programma, N stesso e la strategia), che la strategia sia un numero compreso fra 1 e 3 e, infine, che N non sia minore o uguale a 0.

Parametri di input:

- `argc`: il numero di parametri passati in ingresso
- `N`: il numero di valori da sommare
- `strategy`: la strategia da applicare

Output: Restituisce 0 se i parametri sono corretti, `EXIT_FAILURE` altrimenti.

3.2.5 La funzione *fill_array*

```
void fill_array(int *elements, int N, char *argv[])
```

Descrizione: Riempe l'array in modo randomico nel caso in cui $N > 20$, altrimenti viene riempito utilizzando i valori di $argv$ (quelli dal terzo in poi).

Parametri di input:

- `N`: il numero di valori che si vogliono sommare
- `argv`: il vettore di argomenti

Parametri di Output:

- `elements` il vettore di interi contenente i valori da sommare

3.2.6 La funzione *strategy_2_OR_3_are_applicable*

```
int strategy_2_OR_3_are_applicable(int strategy, int nproc)
```

Descrizione: Verifica se le strategie 2 o 3 sono applicabili, ovvero se numero di processori è una potenza di 2.

Parametri di input:

- `strategy`: la strategia da applicare
- `nproc`: il numero di processori che si vuole utilizzare

Output: Restituisce 0 se il numero dei processori è potenza di 2, 1 altrimenti.

3.2.7 La funzione *sequential_sum*

```
int sequential_sum(int *array, int n)
```

Descrizione: Esegue la somma degli elementi del vettore. Usata da ciascun processore per eseguire la propria somma locale (parziale) prima di applicare la strategia desiderata.

Parametri di input:

- `array`: l'array di interi
- `nproc`: la dimensione dell'array

Output: La somma degli elementi dell'array.

3.2.8 La funzione *operand_distribution*

```
void operand_distribution(int menum, int *elements, int *elements_loc, ↩
    int nloc, int nproc, int rest)
```

Descrizione: Il processo con identificativo 0 distribuisce i diversi operandi da sommare a ciascun processo.

Parametri di input:

- `element`: il vettore di interi da distribuire
- `menum`: l'identificativo del processo
- `nloc`: il numero di elementi che ciascun processore dovrebbe fornire "di partenza"
- `nproc`: il numero di processi
- `rest`: il resto della divisione tra N e $nloc$. In base a questo intero capiamo se altri processi devono sommare elementi in più (evitando che quelli "extra" vengano sommati solo dal processo con ID 0).

Parametri di output:

- `elements_loc`: l'array di interi che ciascun processo dovrà sommare inizialmente

3.2.9 La funzione *compute_power_of_two*

```
void compute_power_of_two(int logNproc, int *array)
```

Descrizione: Calcola le potenze di due in base al numero di step da fare per le strategie 2 e 3.

Parametri di input:

- `logNproc`: il numero di step

Parametri di output:

- `array`: l'array di potenze di due

3.2.10 La funzione *print_result*

```
void print_result(int menum, int strategy, int sum, double timetot)
```

Descrizione: Stampa l'ouput: la somma parziale di ciascun processore per le strategie 2 e 3 e i rispettivi tempi, la somma totale e il tempo impiegato altrimenti (strategia 1).

Parametri di input:

- `menum`: l'identificativo del processo
- `strategy`: la strategia applicata
- `sum`: la somma totale o parziale a seconda della strategia
- `timetot`: il tempo impiegato

Capitolo 4

Testing

Capitolo 5

Analisi performance

In questo capitolo analizzeremo il tempo medio impiegato, lo speed up e l'efficienza al variare del numero dei processori con N fissato. Calcoleremo, inoltre, lo speed up e l'efficienza scalata per ciascun processore al variare di N. Quindi:

- $N = 10.000.000$ nel primo caso, ossia, quello in cui N è fisso e varia il numero dei processori
- Numero dei processori P_i con $i \in 1, ..., 8$
- Per calcolare il tempo medio, per ciascun caso il programma è stato eseguito esattamente 10 volte per considerare, appunto, la media aritmetica
- Una volta ricavato il tempo medio è stato possibile calcolare lo speed e l'efficienza mediante le seguenti formule:
  - Speed up $S = \frac{T(1)}{T(P)}$
  - Efficienza $E = \frac{S(p)}{p}$
- Infine, abbiamo deciso poi di calcolare Speed up ed efficienza scalati. Per capire quale fosse la giusta quantità N da testare a seconda della quantità dei processori è stata utilizzata la seguente formula:

$$K = \frac{P_1 * log(P_1)}{P_0 * log(P_0)}$$

Naturalmente, al fine di avere un algoritmo ottimizzato tutte le inizializzazioni e operazioni costose come logaritmi e potenze, come si evince dal dal Listing 6.1, sono state effettuate prima di utilizzare la funzione `MPI_Wtime`.

Tutti i test sono stati effettuati con un vettore (di dimensione N, naturalmente) contenenente solo il valore 1.

5.1 Analisi con *dieci milioni*

5.1.1 Analisi I strategia

N. processori	P1	P2	P4	P8
	3.959417e-02	2.048206e-02	9.963989e-03	5.017996e-03
	3.959513e-02	2.089810e-02	9.962082e-03	5.033016e-03
	3.962588e-02	1.985812e-02	9.954214e-03	5.146027e-03
	3.960299e-02	1.984787e-02	9.952068e-03	5.839825e-03
	3.962207e-02	1.988387e-02	9.954929e-03	5.017996e-03
	3.963208e-02	1.985788e-02	9.959936e-03	5.131006e-03
	3.962302e-02	1.984787e-02	9.985924e-03	5.027056e-03
	3.994894e-02	1.988387e-02	9.990931e-03	5.017042e-03
	3.959608e-02	1.985502e-02	9.963036e-03	5.046129e-03
	3.953505e-02	1.984811e-02	9.944916e-03	5.015135e-03
Tempo medio	3.9637541e-02	2.0026277e-02	9.9632025e-03	5.1291228e-03
Speed up	1	1.97927657747	3.97839359383	7.72793761148
Efficienza	1	0.98963828873	0.99459839845	0.96599220143

Tabella 5.1: Strategia I con $N = 10^7$

Figura 5.1: Plot *I strategia* del tempo al variare del numero dei processori per $N = 10^7$

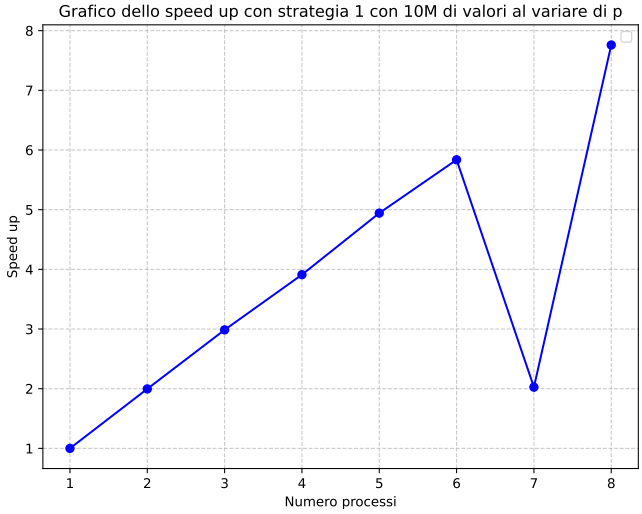


Figura 5.2: Plot *I strategia* dello speed up al variare del numero dei processori per $N = 10^7$

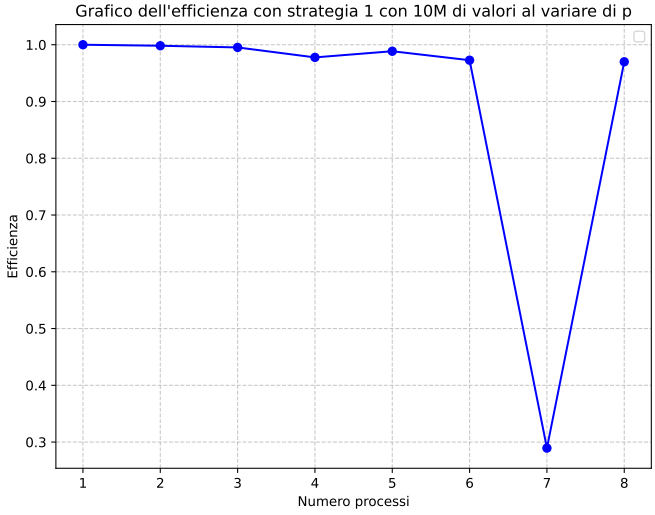


Figura 5.3: Plot *I strategia* dell'efficienza al variare del numero dei processori per $N = 10^7$

5.1.2 Analisi II strategia

N. processori	P1	P2	P4	P8
	3.959417e-02	2.048206e-02	9.963989e-03	5.017996e-03
	3.959513e-02	2.089810e-02	9.962082e-03	5.033016e-03
	3.962588e-02	1.985812e-02	9.954214e-03	5.146027e-03
	3.960299e-02	1.984787e-02	9.952068e-03	5.839825e-03
	3.962207e-02	1.988387e-02	9.954929e-03	5.017996e-03
	3.963208e-02	1.985788e-02	9.959936e-03	5.131006e-03
	3.962302e-02	1.984787e-02	9.985924e-03	5.027056e-03
	3.994894e-02	1.988387e-02	9.990931e-03	5.017042e-03
	3.959608e-02	1.985502e-02	9.963036e-03	5.046129e-03
	3.953505e-02	1.984811e-02	9.944916e-03	5.015135e-03
Tempo medio	3.9637541e-02	2.0026277e-02	9.9632025e-03	5.1291228e-03
Speed up	1	1.97927657747	3.97839359383	7.72793761148
Efficienza	1	0.98963828873	0.99459839845	0.96599220143

Tabella 5.2: Strategia II con $N = 10^7$

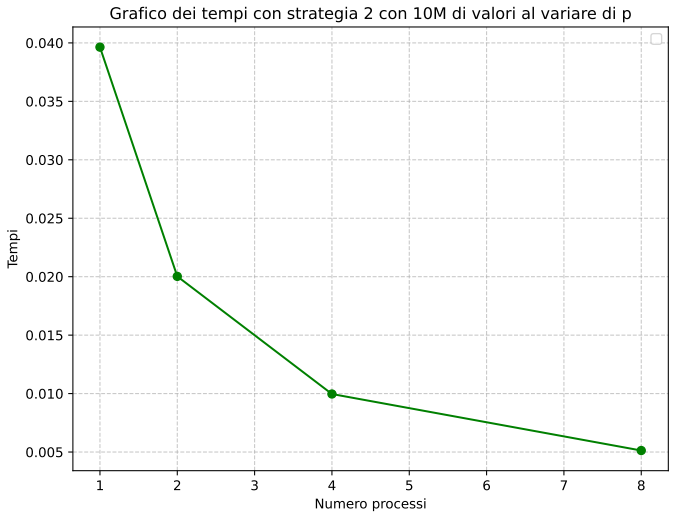


Figura 5.4: Plot *II strategia* del tempo al variare del numero dei processori per $N = 10^7$

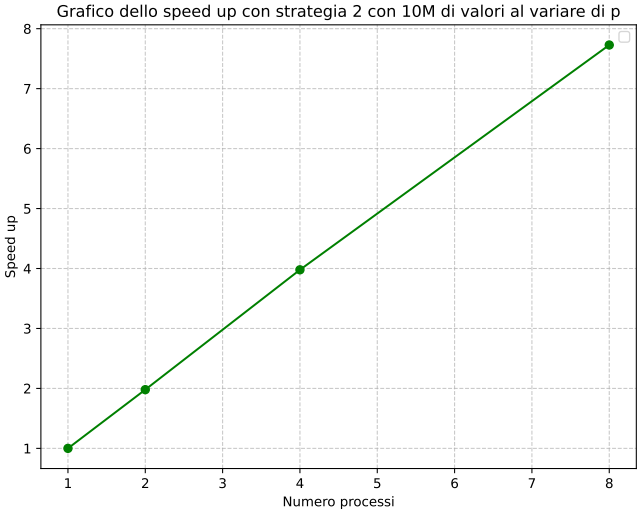


Figura 5.5: Plot *II strategia* dello speed up al variare del numero dei processori per $N = 10^7$

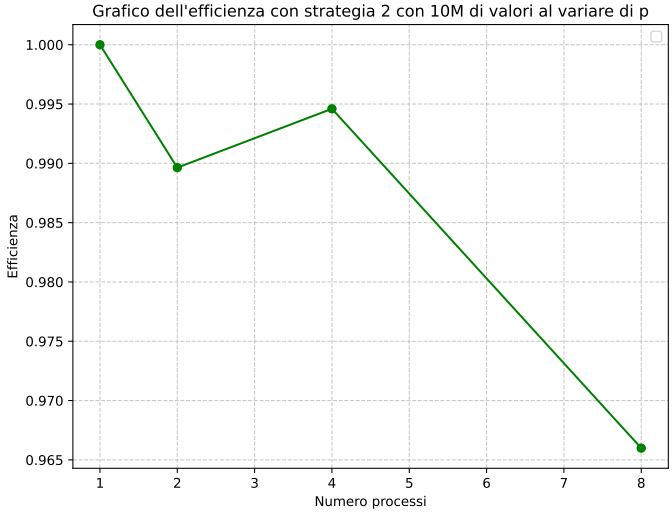


Figura 5.6: Plot *II strategia* dell'efficienza al variare del numero dei processori per $N = 10^7$

5.1.3 Analisi III strategia

N. processori	P1	P2	P4	P8
	3.959417e-02	1.996303e-02	9.971142e-03	5.018950e-03
	3.959513e-02	1.987791e-02	9.980917e-03	5.026102e-03
	3.962588e-02	1.988316e-02	9.977818e-03	5.029917e-03
	3.960299e-02	1.989198e-02	9.979963e-03	5.047083e-03
	3.962207e-02	1.987410e-02	9.982109e-03	5.023003e-03
	3.963208e-02	1.986694e-02	10.01406e-03	5.033016e-03
	3.962302e-02	1.987600e-02	9.973049e-03	5.028009e-03
	3.994894e-02	1.989913e-02	9.979963e-03	5.763054e-03
	3.959608e-02	1.990294e-02	9.986162e-03	5.147934e-03
	3.953505e-02	1.986909e-02	9.968996e-03	5.150080e-03
Tempo medio	3.9637541e-02	1.9890428e-02	9.9814179e-03	5.1267148e-03
Speed up	1	1.9927947754	3.97113329961	7.7315673967
Efficienza	1	0.9963973877	0.9927833249	0.9664459245

Tabella 5.3: Strategia III con $N = 10^7$

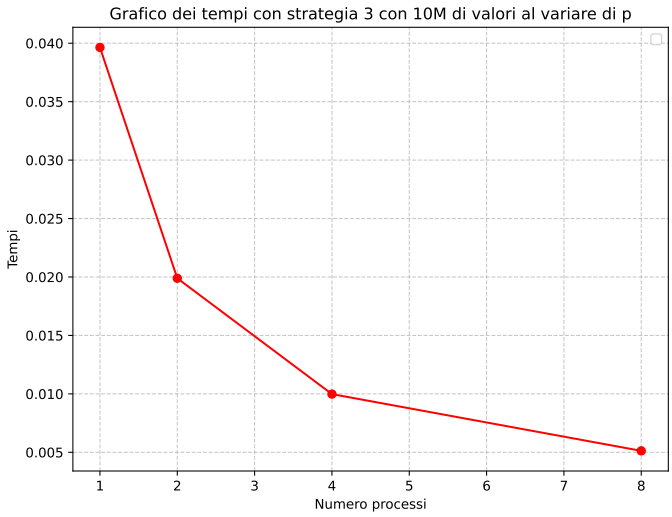


Figura 5.7: Plot *III strategia* del tempo al variare del numero dei processori per $N = 10^7$

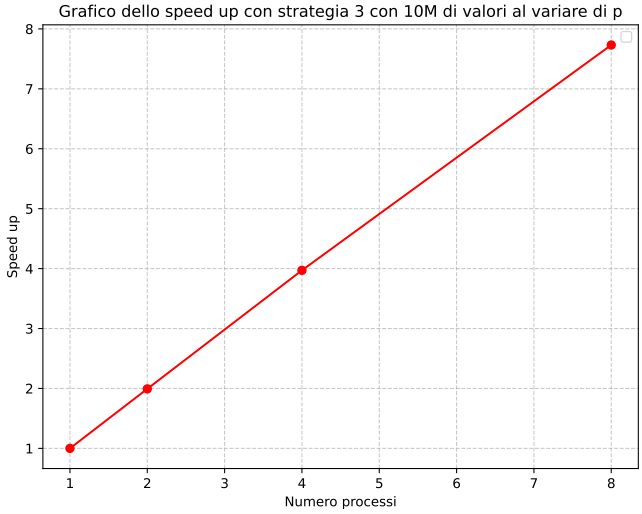


Figura 5.8: Plot *III strategia* dello speed up al variare del numero dei processori per $N = 10^7$

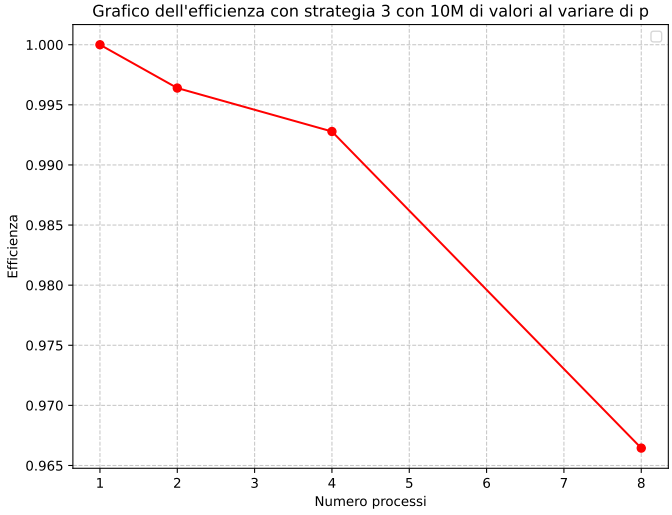


Figura 5.9: Plot *III strategia* dell'efficienza al variare del numero dei processori per $N = 10^7$

5.1.4 Confronto fra strategie

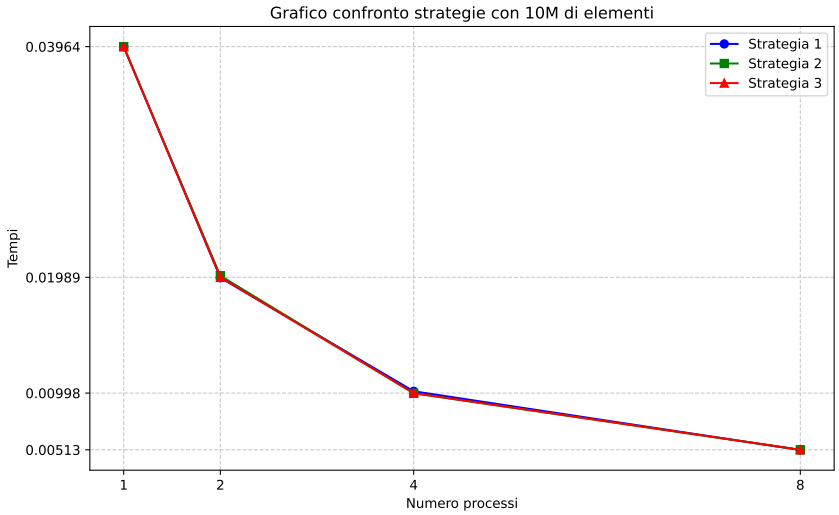


Figura 5.10: Plot della differenza fra strategie in termini di tempo al variare del numero dei processori per $N = 10^7$

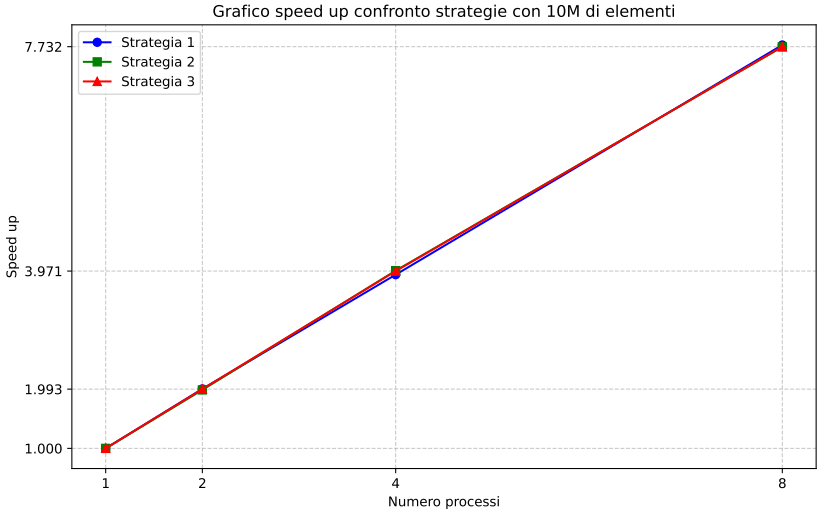


Figura 5.11: Plot della differenza fra strategie in termini di speed up al variare del numero dei processori per $N = 10^7$

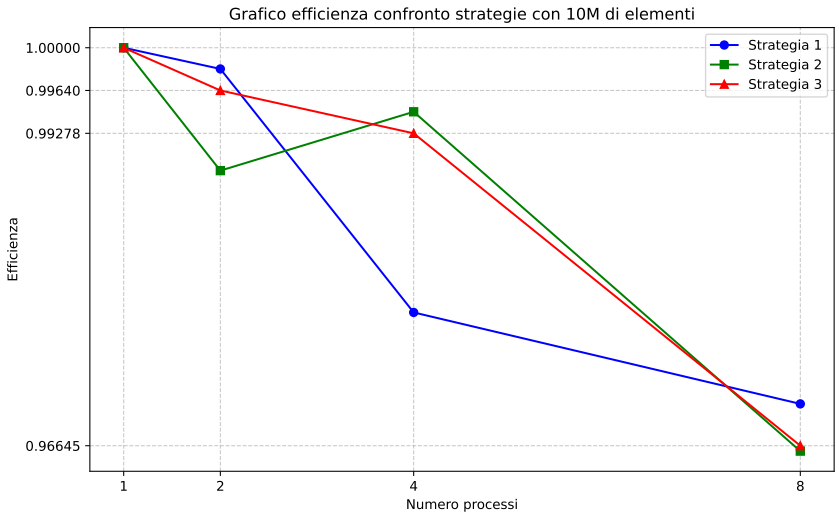


Figura 5.12: Plot della differenza fra strategie in termini di efficienza al variare del numero dei processori per $N = 10^7$

5.2 Analisi *N variabile*

5.2.1 Analisi I strategia

N. processori	**P2** $N_0 = 10^5$	**P4** $N_1 = 4 * 10^5$	**P8** $N_1 = 1.2 * 10^6$
	2.241135e-04	4.251003e-04	6.508827e-04
	2.119541e-04	4.239082e-04	6.132126e-04
	2.100468e-04	4.119873e-04	6.539822e-04
	2.140999e-04	4.761219e-04	7.081032e-04
	2.431870e-04	4.758835e-04	6.539822e-04
	2.140999e-04	4.148483e-04	6.539822e-04
	2.138615e-04	4.789829e-04	6.580353e-04
	2.100468e-04	4.179478e-04	6.160736e-04
	2.110004e-04	4.131794e-04	6.539822e-04
	2.331734e-04	4.110336e-04	6.141663e-04
Tempo medio	2.1855833e-04	4.3489932e-04	6.4764025e-04
Speed up	1	2.01019702675	4.04962471063
Efficienza	0.5	0.50254925668	0.50620308882

Tabella 5.4: Strategia I con N Scalato

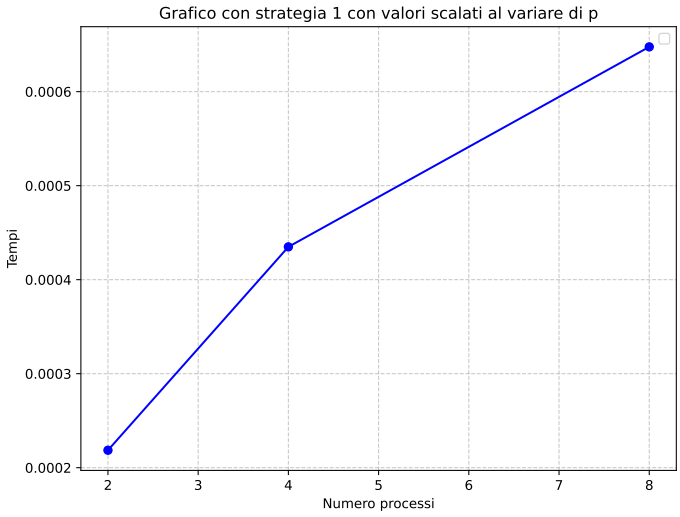


Figura 5.13: Plot *I strategia* del tempo scalato

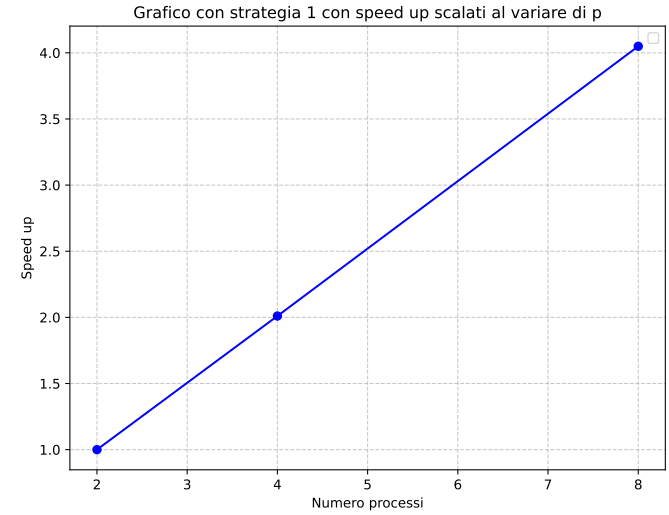


Figura 5.14: Plot *I strategia* dello speed up scalato

Figura 5.15: Plot *I strategia* dell'efficienza scalata

5.2.2 Analisi II strategia

N. processori	**P2** $N_0 = 5 * 10^5$	**P4** $N_1 = 2 * 10^6$	**P8** $N_1 = 6 * 10^6$
	1.007080e-03	2.012968e-03	3.014088e-03
	1.013041e-03	2.003908e-03	3.017902e-03
	1.282930e-03	2.005100e-03	3.011942e-03
	1.021147e-03	2.004147e-03	3.012896e-03
	1.004934e-03	2.014875e-03	3.015995e-03
	1.011133e-03	2.004147e-03	3.010988e-03
	1.003981e-03	2.017021e-03	3.010035e-03
	1.003027e-03	2.011061e-03	3.015995e-03
	1.004934e-03	2.007008e-03	3.808975e-03
	1.003981e-03	2.011061e-03	3.010988e-03
Tempo medio	1.0356188e-03	2.0095349e-03	3.0929804e-03
Speed up	1	2.06182577769	4.0179451509
Efficienza	0.5	0.51545644442	0.50224314386

Tabella 5.5: Strategia II con N Scalato

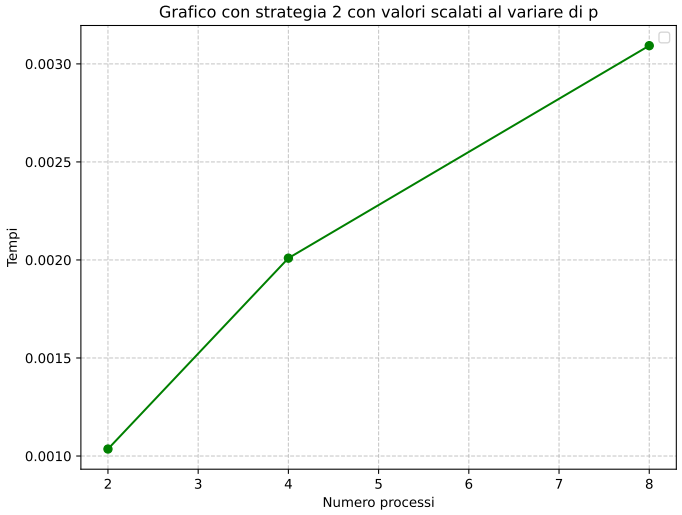


Figura 5.16: Plot *II strategia* del tempo scalato

Figura 5.17: Plot *II strategia* dello speed up scalato

Figura 5.18: Plot *II strategia* dell'efficienza scalata

5.2.3 Analisi III strategia

N. processori	**P2** $N_0 = 10^6$	**P4** $N_1 = 4 * 10^6$	**P8** $N_1 = 12 * 10^6$
	1.996040e-03	4.653931e-03	6.064177e-03
	2.006054e-03	4.215002e-03	6.029129e-03
	2.037048e-03	4.836798e-03	6.025076e-03
	2.161980e-03	5.012035e-03	6.026983e-03
	2.004147e-03	4.010916e-03	6.025076e-03
	2.214909e-03	4.011154e-03	6.021023e-03
	2.218008e-03	4.009008e-03	6.032944e-03
	2.038956e-03	4.009962e-03	6.030083e-03
	2.336979e-03	4.038095e-03	6.048918e-03
	2.634048e-03	4.035950e-03	6.032944e-03
Tempo medio	2.1648169e-03	4.2832851e-03	6.0336353e-03
Speed up	1	2.02164166004	4.3054976823
Efficienza	0.5	0.50541041501	0.53818721028

Tabella 5.6: Strategia III con N Scalato

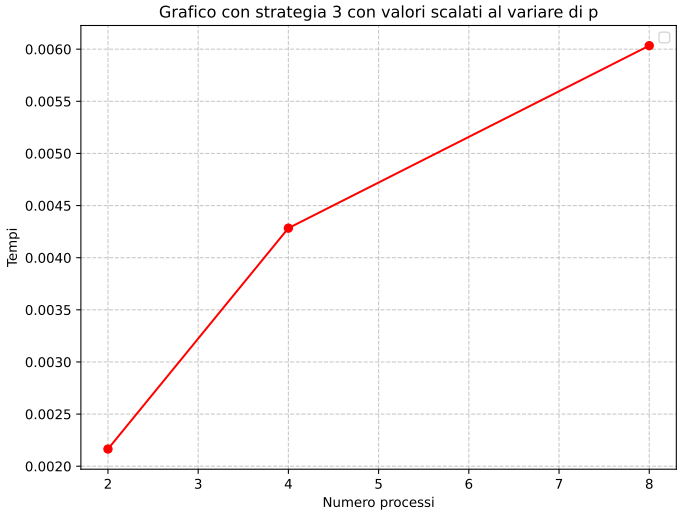


Figura 5.19: Plot *III strategia* del tempo scalato

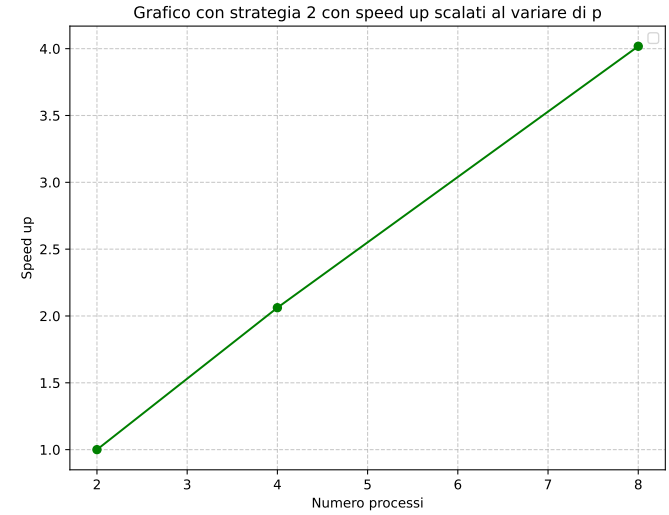


Figura 5.20: Plot *III strategia* dello speed up scalato

Figura 5.21: Plot *III strategia* dell'efficienza scalata

5.2.4 Confronto fra strategie

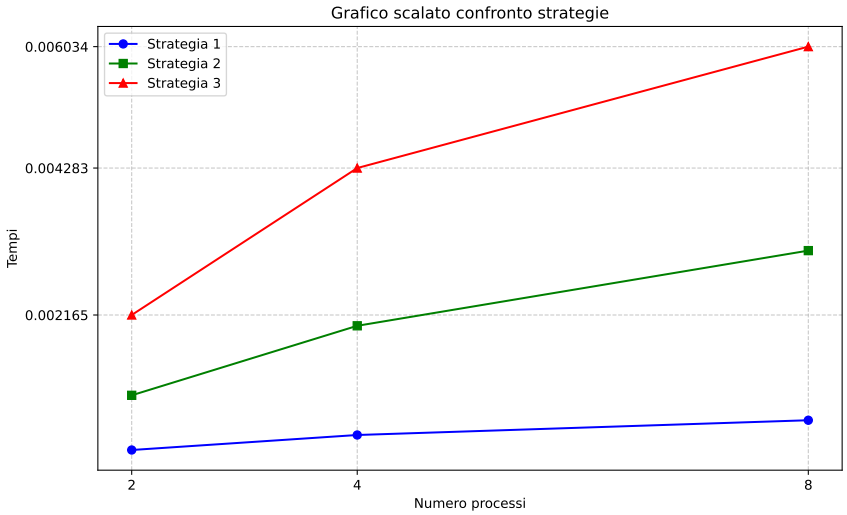


Figura 5.22: Plot della differenza fra strategie in termini di tempo scalato

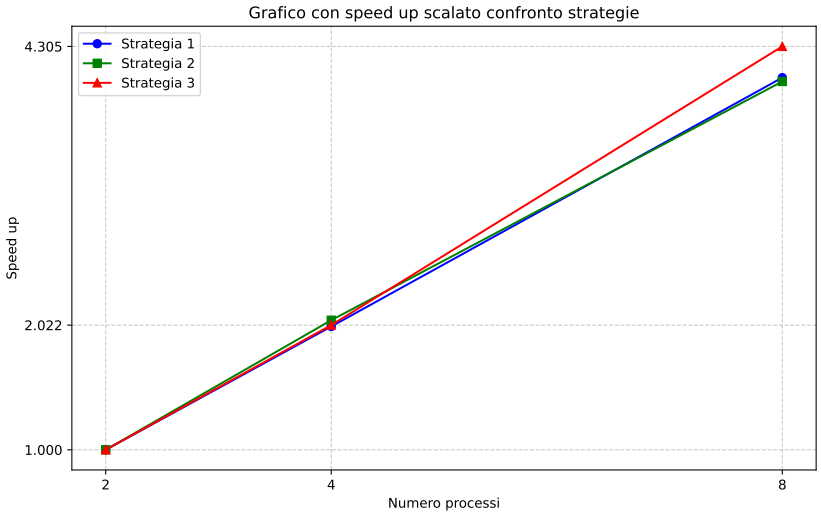


Figura 5.23: Plot della differenza fra strategie in termini di speed up scalato

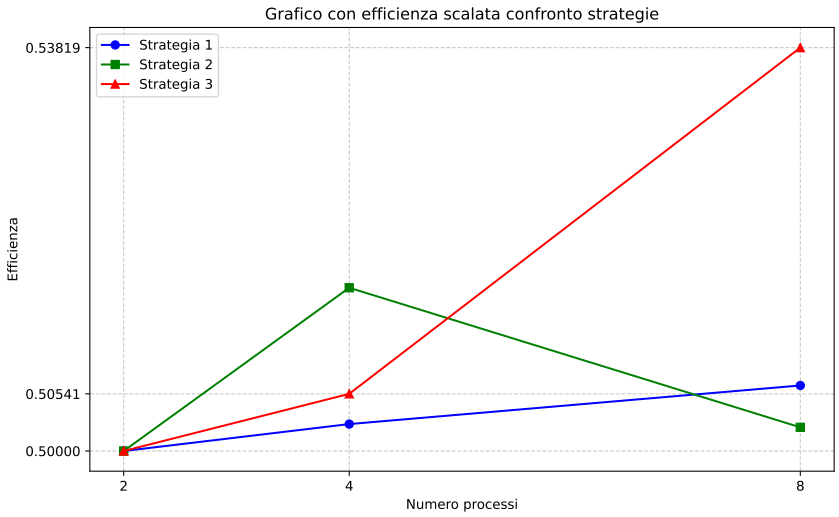


Figura 5.24: Plot della differenza fra strategie in termini di efficienza scalata

Capitolo 6

Source code

6.1 *Main.c*

```c
/**
 * @author Fabrizio Vitale
 * @author Giovanni Falcone
 * @author Luigi Mangiacapra
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "mpi.h"
#include "Strategy.h"
#include "Utils.h"

# define STRATEGY_1 1
# define STRATEGY_2 2
# define STRATEGY_3 3

int main(int argc, char *argv[]){
    int menum;                      // id del processore
    int nproc;                      // numero processori
    int N;                          // numero di elementi da sommare
    int sum;                        // somma totale da stampare
    int logNproc;                   // numero di passi da effettuare ↩
    per la II, III strategia
    int strategy;                   // strategia con cui sommare
    int nloc;                       // numero di elementi che ciascun ↩
    processore deve sommare
    int rest;                       // resto della divisione
    int *elements;                  // array completo
    int *elements_loc;              // vettore di elementi locale
    int *array_of_powers_of_two;    // vettore di potenze di 2
    double end_time;
    double start_time;
    double timetot = 0;

```

```
35      if(argc < 3){
36          fprintf(stderr, "Utilizzo: <numeri da sommare> <tipo di strategia> <numeri da sommare se N>\n");
37          return EXIT_FAILURE;
38      }
39
40      // MPI initialization
41      MPI_Init(&argc, &argv);
42      MPI_Comm_rank(MPI_COMM_WORLD, &menum);
43      MPI_Comm_size(MPI_COMM_WORLD, &nproc);
44
45      if(menum == 0){
46          // convert to integer the number to sum and strategy to apply
47          N = atoi(argv[1]);
48          strategy = atoi(argv[2]);
49
50          if(check_if_inputs_are_valid(argc, N, strategy) != 0){
51              MPI_Finalize();
52              return EXIT_FAILURE;
53          }
54
55          // array malloc
56          elements = (int *)malloc(sizeof(int) * N);
57          if(elements == NULL){
58              fprintf(stderr, "Errore nell'allocazione della memoria per l'array 'elements'!\n");
59              return EXIT_FAILURE;
60          }
61
62          // fill array with N elements
63          fill_array(elements, N, argv);
64
65          // Verifica se la strategia 2 (o 3) e' applicabile: se il numero dei processori non e' potenza di 2 applica la strategia 1
66          if(!strategy_2_OR_3_are_applicable(strategy, nproc)){
67              strategy = STRATEGY_1;
68              printf("Applico la prima strategia.\n");
69          }
70      }
71
72      // send data to all other processors
73      MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);
74      MPI_Bcast(&strategy, 1, MPI_INT, 0, MPI_COMM_WORLD);
75
76      // compute the logarithm and powers of two for second and third strategy
77      if(strategy == STRATEGY_2 || strategy == STRATEGY_3) {
78          // get number of steps
79          logNproc = log2(nproc);
80          // allocation of array of powers of 2
81          array_of_powers_of_two = (int *)malloc(sizeof(int) * logNproc);
82          if(array_of_powers_of_two == NULL){
83              fprintf(stderr, "Errore nell'allocazione della memoria per l'array 'array_of_powers_of_two'!\n");
```

```
84              return EXIT_FAILURE;
85          }
86          // fill the array with powers of 2
87          compute_power_of_two(logNproc, array_of_powers_of_two);
88      }
89
90
91      // in order to check how many elements each processor must sum
92      nloc = N / nproc;
93      rest = N % nproc;
94
95      if(menum < rest){
96          nloc = nloc + 1;
97      }
98
99      // allocation of local array for each processor
100     elements_loc = (int *)malloc(sizeof(int) * nloc);
101     if(elements_loc == NULL){
102         fprintf(stderr, "Errore nell'allocazione della memoria per l'↩
        array 'elements'!\n");
103         return EXIT_FAILURE;
104     }
105
106     // invia elementi da sommare agli altri processori
107     operand_distribution(menum, elements, elements_loc, nloc, nproc, ↩
        rest);
108
109     // attendiamo che i processi si sincronizzino
110     MPI_Barrier(MPI_COMM_WORLD);
111     start_time = MPI_Wtime();
112
113     sum = 0;
114     // first step: each processor performs the first partial sum
115     sum = sequential_sum(elements_loc, nloc);
116
117     // check the strategy to apply
118     if(strategy == STRATEGY_1){
119         sum = first_strategy(menum, nproc, sum);
120     }else if(strategy == STRATEGY_2){
121         sum = second_strategy(menum, logNproc, array_of_powers_of_two, ↩
        sum);
122     } else{ // third_strategy
123         sum = third_strategy(menum, logNproc, array_of_powers_of_two, ↩
        sum);
124     }
125
126     end_time = MPI_Wtime();
127
128     double timeP = end_time - start_time;
129     printf("Il tempo impiegato da %d e' di %e s\n", menum, timeP);
130
131     // compute total time
132     MPI_Reduce(&timeP, &timetot, 1, MPI_DOUBLE, MPI_MAX, 0, ↩
        MPI_COMM_WORLD);
```

```
133
134     // print sum total and partial sum of each processor and the time
135     print_result(menum, strategy, sum, timetot);
136
137     // freeing memory before program termination
138     if(menum == 0){
139         free(elements);
140         free(elements_loc);
141         free(array_of_powers_of_two);
142     }else{
143         free(elements_loc);
144         free(array_of_powers_of_two);
145     }
146
147     MPI_Finalize();
148     return 0;
149 }
```

6.2 *Strategy.c*

```
1  /**
2   * @author Fabrizio Vitale
3   * @author Giovanni Falcone
4   * @author Luigi Mangiacapra
5   */
6
7  #include <stdio.h>
8  #include <stdlib.h>
9  #include <time.h>
10 #include <math.h>
11
12 #include "mpi.h"
13
14
15 int first_strategy(int menum, int nproc, int sum){
16     int sum_parz = 0;
17     int tag;
18     MPI_Status status;
19
20     if(menum == 0){
21         for(int i = 1; i < nproc; i++){
22             tag = 80 + i;
23             MPI_Recv(&sum_parz, 1, MPI_INT, i, tag, MPI_COMM_WORLD, &↩
       status);
24             sum += sum_parz;
25         }
26     }else{
27         tag = menum + 80;
28         MPI_Send(&sum, 1, MPI_INT, 0, tag, MPI_COMM_WORLD);
29     }
30
31     return sum;
```

```
32 }
33
34 int second_strategy(int menum, int logNproc, int *array, int sum){
35     int sum_parz = 0;
36     int tag;
37     int partner;
38
39     int power_for_partecipation;
40     int does_processor_partecipate;
41
42     int power_for_communication;
43     int does_processor_receive;
44
45     MPI_Status status;
46
47     for(int i = 0; i < logNproc; i++){
48         power_for_partecipation =  array[i];
49         does_processor_partecipate = (menum % power_for_partecipation) ↩
    == 0;
50
51         if(does_processor_partecipate){
52             power_for_communication = array[i + 1];
53             does_processor_receive = (menum % power_for_communication) ↩
    == 0;
54
55             if (does_processor_receive){
56                 partner = menum + power_for_partecipation;
57                 tag = 60 + i;
58                 MPI_Recv(&sum_parz, 1, MPI_INT, partner, tag, ↩
    MPI_COMM_WORLD, &status);
59                 sum += sum_parz;
60             }
61             else{
62                 partner = menum - power_for_partecipation;
63                 tag = 60 + i;
64                 MPI_Send(&sum, 1, MPI_INT, partner, tag, MPI_COMM_WORLD↩
    );
65             }
66         }
67     }
68
69     return sum;
70 }
71
72 int third_strategy(int menum, int logNproc, int *array, int sum){
73     int partner;
74     int send_tag;
75     int recv_tag;
76     int sum_parz;
77     MPI_Status status;
78
79     sum_parz = 0;
80     for(int i = 0; i < logNproc; i++){
81         if ((menum % array[i + 1]) < array[i]) {
```

```
82              partner = menum + array[i];
83              send_tag = 40 + i;
84              recv_tag = 40 + i;
85
86              // Invia la somma locale al processo partner
87              MPI_Send(&sum, 1, MPI_INT, partner, send_tag, ↩
    MPI_COMM_WORLD);
88
89              // Ricevi la somma del processo partner
90              MPI_Recv(&sum_parz, 1, MPI_INT, partner, recv_tag, ↩
    MPI_COMM_WORLD, &status);
91
92              // Aggiorna la variabile 'sum' con la somma ricevuta
93              sum += sum_parz;
94          } else {
95              partner = menum - array[i];
96              send_tag = 40 + i;
97              recv_tag = 40 + i;
98
99              // Ricevi la somma dal processo partner
100             MPI_Recv(&sum_parz, 1, MPI_INT, partner, recv_tag, ↩
    MPI_COMM_WORLD, &status);
101
102             // Invia la somma locale al processo partner
103             MPI_Send(&sum, 1, MPI_INT, partner, send_tag, ↩
    MPI_COMM_WORLD);
104             sum += sum_parz;
105         }
106     }
107
108     return sum;
109 }
```

6.3 *Strategy.h*

```
1  #ifndef STRATEGY_H
2  #define STRATEGY_H
3
4  /**
5   * @brief apply the first strategy
6   *
7   * @param menum id of the processor
8   * @param nproc number of processor
9   * @param sum partial sum performed at the first step
10  * @return int total sum
11  */
12 int first_strategy(int menum, int nproc, int sum);
13
14 /**
15  * @brief apply the second strategy
16  *
17  * @param menum id of the processor
```

```
 * @param logNproc number of steps
 * @param array the array of powers of two
 * @param sum  partial sum performed at the first step
 * @return int total sum
 */
int second_strategy(int menum, int logNproc, int *array, int sum);

/**
 * @brief apply the third strategy
 *
 * @param menum id of the processor
 * @param logNproc number of steps
 * @param array the array of powers of two
 * @param sum  partial sum performed at the first step
 * @return int total sum
 */
int third_strategy(int menum, int logNproc, int *array, int sum);

#endif
```

6.4 *Utils.c*

```
/**
 * @author Fabrizio Vitale
 * @author Giovanni Falcone
 * @author Luigi Mangiacapra
*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

# include "mpi.h"

/* ************************************************************** */
/*                        SUPPORT FUNCTION                        */
/* ************************************************************** */

static void fill_array_randomly(int *elements, int N);

static void fill_array_by_argv(int *elements, int N, char *argv[]);

/**
 * @brief check if strategy is a number between 1 and 3
 *
 * @param strategy the integer
 *
 * @return 0 if it's valid, 1 otherwise
 *
*/
static int strategy_is_valid(int strategy);
```

```
31
32 /* ************************************************************** */
33 /* ************************************************************** */
34 /* ************************************************************** */
35
36 int check_if_inputs_are_valid(int argc, int N, int strategy){
37     if(N <= 20 && argc - 3 != N && strategy_is_valid(strategy) == 0){
38         fprintf(stderr, "Il numero di elementi inserito non corrisponde↩
       ad N!\n");
39         return EXIT_FAILURE;
40     }
41
42     if(N <= 0){
43         fprintf(stderr, "Inserire un numero maggiore di 0!\n");
44         return EXIT_FAILURE;
45     }
46
47     if(strategy_is_valid(strategy) != 0){
48         fprintf(stderr, "La strategia deve essere un valore compreso ↩
      tra 1 e 3!\n");
49         return EXIT_FAILURE;
50     }
51
52     return 0; // valid input
53 }
54
55 int strategy_is_valid(int strategy){
56     if(strategy < 1 && strategy > 3)
57         return EXIT_FAILURE;
58
59     // it's valid
60     return 0;
61 }
62
63 void fill_array(int *elements, int N, char *argv[]){
64     if(N > 20)
65         fill_array_randomly(elements, N);
66     else
67         fill_array_by_argv(elements, N, argv);
68 }
69
70 void fill_array_randomly(int *elements, int N){
71     srand(time(NULL));
72
73     printf("Generazione numeri randomici...\n");
74
75     for(int i = 0; i < N; i++){
76         elements[i] = rand() % 100;
77     }
78 }
79
80 void fill_array_by_argv(int *elements, int N, char *argv[]){
81     printf("Inserimento dei numeri forniti da terminale...\n");
82
```

```
83      for(int i = 0; i < N; i++){
84          elements[i] = atoi(argv[i + 3]);    // 0: name src; 1: N; 2: ↩
    strategy; starting from 3 we have all numbers
85      }
86  }
87
88  int strategy_2_OR_3_are_applicable(int strategy, int nproc){
89      return !(((strategy == 2 || strategy == 3) && ((nproc & (nproc - 1)↩
    ) != 0)) || (nproc == 1)) ? 1 : 0;
90  }
91
92  int sequential_sum(int *array, int n){
93      int sum = 0;
94
95      for(int i = 0; i < n; i++){
96          sum += array[i];
97      }
98
99      return sum;
100 }
101
102 void operand_distribution(int menum, int *elements, int *elements_loc, ↩
    int nloc, int nproc, int rest){
103     int tag;
104     MPI_Status status;
105
106     if (menum == 0){
107         for (int i = 0; i < nloc; i++){
108             elements_loc[i] = elements[i];
109         }
110
111         int tmp = nloc;
112         int start = 0;
113         for (int i = 1; i < nproc; i++){
114             start += tmp;
115             tag = 22 + i;
116             if (i == rest)
117                 tmp -= 1;
118
119             MPI_Send(&elements[start], tmp, MPI_INT, i, tag, ↩
    MPI_COMM_WORLD);
120         }
121     } else {
122         tag = 22 + menum;
123         MPI_Recv(elements_loc, nloc, MPI_INT, 0, tag, MPI_COMM_WORLD, &↩
    status);
124     }
125 }
126
127 void print_result(int menum, int strategy, int sum, double timetot){
128     if(strategy == 1){
129         if(menum == 0)
130             printf("La somma totale e' %d e l'algoritmo, per calcolarla↩
    , ha impiegato %e.\n", sum, timetot);
```

```
131     } else {
132         printf("\n Sono il processo %d e la somma totale e' %d\n", ↩
    menum, sum);
133         if(menum == 0)
134             printf("Tempo totale impiegato per l'algoritmo: %e\n", ↩
    timetot);
135     }
136 }
137 
138 void compute_power_of_two(int logNproc, int *array){
139     for(int i = 0; i < logNproc + 1; i++){
140         array[i] = (int) pow(2, i);
141     }
142 }
```

6.5 *Utils.h*

```
1  #ifndef UTILS_H
2  #define UTILS_H
3  
4  /**
5   * @brief check if the inpurs are correct in order to sum
6   *
7   * @param argc number of parameters
8   * @param N number of elements to sum
9   * @param strategy the strategy to apply
10  */
11 int check_if_inputs_are_valid(int argc, int N, int strategy);
12 
13 /**
14  * @brief fill the array randomly if N is greater than 20, from argv ↩
     otherwise
15  *
16  * @param elements the array of integers
17  * @param N capacity of array
18  * @param argv the elements to insert into the array
19  */
20 void fill_array(int *elements, int N, char *argv[]);
21 
22 /**
23  * @brief check if the strategy 2 (or 3) is applicable: the number of ↩
     processor must be a power of 2
24  *
25  * @param strategy the strategy to apply (2 or 3)
26  * @param nproc the number of processor
27  * @return int 1 if it's applicable, 0 otherwise
28  */
29 int strategy_2_OR_3_are_applicable(int strategy, int nproc);
30 
31 /**
32  * @brief performs the sum of each array value and returns it
33  *
```

```
34  * @param array the array of integer
35  * @param n size of array
36  * @return int the sum
37  */
38 int sequential_sum(int *array, int n);
39
40 /**
41  * @brief the processor with id 0 send the elements to sum to the other↩
        processor
42  *
43  * @param menum id of the processor
44  * @param elements array of all integers
45  * @param elements_loc local array for each processor
46  * @param nloc number of elements to sum for each processor
47  * @param nproc number of processor
48  * @param rest the rest of division between the all numbers to sum and ↩
       number of processor
49  */
50 void operand_distribution(int menum, int *elements, int *elements_loc, ↩
       int nloc, int nproc, int rest);
51
52 /**
53  * @brief print results: print the partial sum for each processor, the ↩
       time spent for each partial sum,
54  * the total sum and total time spent for the total sum
55  *
56  * @param menum id of processor
57  * @param strategy the strategy applied
58  * @param sum the result to print
59  * @param timetot time taken
60  */
61 void print_result(int menum, int strategy, int sum, double timetot);
62
63 /**
64  * @brief compute the powers of 2 for second and third strategy
65  *
66  * @param logNproc number of steps
67  * @param array array to fill
68  */
69 void compute_power_of_two(int logNproc, int *array);
70
71 #endif
```

6.6 *job-script.pbs*

Il seguente pbs considera il caso in cui $N = 10000$ e $strategy = 1$.

```
1 #!/bin/bash
2
3 #PBS -q studenti
4 #PBS -l nodes=1:ppn=1
5 #PBS -N Main
6 #PBS -o Main.out
```

```
#PBS -e Main.err


cat $PBS_NODEFILE
echo ----------------------------
sort -u $PBS_NODEFILE > hostlist

NCPU=$(wc -l < hostlist)
echo ----------------------------
echo 'This job is allocated on '${NCPU}' cpu(s)'' on host:'
cat hostlist
echo ----------------------------

PBS_O_WORKDIR=$PBS_O_HOME/Progetto_Sum


echo ------------------------
echo "Eseguo: /usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS_O_WORKDIR/↩
    Main $PBS_O_WORKDIR/Main.c $PBS_O_WORKDIR/Strategy.c $PBS_O_WORKDIR/↩
    Utils.c"
/usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS_O_WORKDIR/Main ↩
    $PBS_O_WORKDIR/Main.c $PBS_O_WORKDIR/Strategy.c $PBS_O_WORKDIR/Utils↩
    .c -lm -std=c99

echo "Eseguo: /usr/lib64/openmpi/1.4-gcc/bin/-machinefile hotlist -np ↩
    $NCPU $PBS_O_WORKDIR/Main"
/usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile hostlist -np $NCPU ↩
    $PBS_O_WORKDIR/Main 10000 1
```