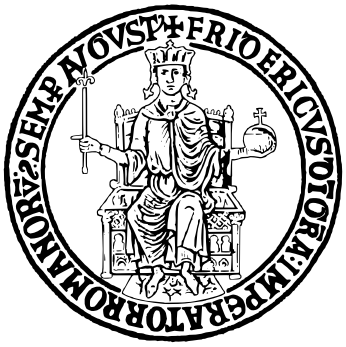
Università degli Studi di Napoli Federico II



Scuola Politecnica e delle Scienze di Base
Dipartimento di Ingegneria Elettrica e
Tecnologie dell'Informazione

Corso di Laurea Magistrale in Informatica

Parallel and Distributed Computing

Progettazione di un algoritmo per il calcolo della somma di n numeri reali in ambiente di calcolo parallelo su architettura MIMD a memoria distribuita

Docenti
Prof. Giuliano Laccetti
Prof.ssa Valeria Mele

Candidati
Marco Romano N97000395
Gianluca L'arco N97000393

Anno Accademico 2021 - 2022

Indice

Elenco delle figure

Elenco delle tabelle

1 Definizione e analisi del problema

Lo scopo del software è calcolare la somma di n numeri in parallelo su p processori. Il numero di operandi e il numero di processori da adoperare nel calcolo vengono forniti in input tramite riga di comando, mentre gli addendi vengono generati randomicamente all'interno del software se $n > 20$, altrimenti vengono forniti anch'essi tramite riga di comando.

L'infrastruttura di calcolo parallelo utilizzata per gli esperimenti è un'architettura di tipo MIMD a memoria distribuita. La gestione dei processori avviene mediante la libreria *Message Passing Interface* (MPI), che consente agli utenti di creare programmi che possono essere eseguiti in modo efficiente sulla maggior parte delle architetture parallele (cluster).

Il linguaggio di programmazione adoperato per sviluppare il software è il C.

2 Input e Output

Il software riceve in input i seguenti parametri:

1. *operands*: indica il numero di addendi nell'operazione di somma. Tale parametro deve essere necessariamente un intero maggiore di 0. Se *operands* < 20, vanno forniti in input anche gli addendi stessi, altrimenti vengono generati randomicamente all'interno del software;

2. *strategy*: specifica il tipo di strategia da adottare. Il suo valore deve essere compreso tra 1 e 3;

3. *printer*: identifica il processore che deve effettuare la stampa del risultato, il suo valore deve essere compreso tra -1 e il numero dei processori sottratto uno.

Per facilitare la comprensione circa i parametri di input del software, è stata implementata la funzionalità *help*, la quale può essere invocata direttamente da linea di

comando mediante il comando --help.

L'output è contenuto nel file *SommaNumeri.out* ed è strutturato nel seguente modo:

▷ [P0] Result of the sum: 3.000000

3 Indicatori di errore

Gli indicatori di errore, illustrati anche nella funzione *help*, sono descritti nella Tabella 1:

Codice	Errore	Descrizione
101	ERR_ARGC	Invalid number of arguments
102	ERR_NO_OPERANDS	Mandatory argument [-o --operands] not provided
103	ERR_NO_STRATEGY	Mandatory argument [-s --strategy] not provided
104	ERR_NO_PRINTER	Mandatory argument [-p --printer] not provided
105	ERR_TOT_OPERANDS	Invalid amount of operands provided
106	ERR_OPERAND	Invalid operand provided
107	ERR_STRATEGY	Invalid strategy provided
108	ERR_PRINTER	Invalid ID printer provided
109	ERR_MEMORY	Unable to allocate memory

Tabella 1: Indicatori di errori

4 Subroutine

In questa sezione vengono elencate e descritte tutte le funzioni adoperate della libreria *mpi.h* (implementazione di MPI, un protocollo di comunicazione utilizzato nelle applicazioni per sistemi a memoria distribuita per il calcolo parallelo) oltre alle diverse funzioni codificate.

4.1 Funzioni MPI

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

Descrizione: Inizializzare l'ambiente di esecuzione MPI consentendo la comunicazione e la sincronizzazione tra processori.

Input: *argc* puntatore al numero di argomenti; *argv* puntatore all'array degli argomenti.

Errori: Restituisce un codice di errore tra (MPI_SUCCESS, MPI_ERR_OTHER).

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

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

Input: *comm* communicator di riferimento.

Output: *rank* identificativo del processo chiamante nel gruppo *comm*.

Errori: Restituisce un codice di errore tra (MPI_SUCCESS, MPI_ERR_OTHER).

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

Descrizione: Determina la dimensione del gruppo associato a un comunicatore.

Input: *comm* communicator di riferimento.

Output: *size* numero di processori nel gruppo *comm*.

Errori: Restituisce un codice di errore tra (MPI_SUCCESS, MPI_ERR_OTHER).

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

Descrizione: Trasmette un messaggio dal processo con identificativo *root* a tutti gli altri processi del comunicatore.

Input: $buffer$ puntatore al buffer di lettura; *count* numero di elementi del buffer; *root* identificativo del processo root; *comm* communicator di riferimento.

Output: $buffer$ puntatore al buffer di scrittura.

Errori: Restituisce un codice di errore tra (MPI_SUCCESS, MPI_ERR_COMM, MPI_ERR_COUNT, MPI_ERR_TYPE, MPI_ERR_BUFFER, MPI_ERR_ROOT).

```
int MPI_Finalize(void)
```

Descrizione: Termina l'ambiente di esecuzione MPI. Tutti i processi devono chiamare questa routine prima di uscire.

Errori: Restituisce un codice di errore tra (MPI_SUCCESS, MPI_ERR_OTHER).

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

Descrizione: Si blocca finché tutti i processi nel comunicatore non hanno raggiunto questa routine.

Input: *comm* communicator di riferimento.

Errori: Restituisce un codice di errore tra (MPI_SUCCESS, MPI_ERR_OTHER).

```
double MPI_Wtime(void)
```

Descrizione: Restituisce il tempo trascorso sul processore chiamante.

Output: Tempo trascorso sul processore chiamante.

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

Descrizione: Riduce i valori su tutti i processi a un unico valore. É possibile effettuare la riduzione con diverse operazioni (MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, MPI_LAND, MPI_LOR, MPI_BAND, MPI_BOR, MPI_MAXLOC, MPI_MINLOC).

Input: *sendbuf* puntatore al buffer di trasmissione; *count* numero di elementi del buffer; *datatype* tipo degli elementi del buffer; *op* operazione di riduzione; *root* identificativo del processo root che riceverà il risultato; *comm* comunicatore di riferimento.

Output: *recvbuf* puntatore al buffer di ricezione.

Errori: Restituisce un codice di errore tra (MPI_SUCCESS, MPI_ERR_COMM, MPI_ERR_COUNT, MPI_ERR_TYPE, MPI_ERR_BUFFER).

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

Descrizione: Funzione di trasmissione dati bloccante.

Input: *buf* puntatore al buffer di trasmissione; *count* numero di elementi del buffer; *datatype* tipo degli elementi del buffer; *dest* idenbtificativo del processo destinatario; *tag* identificativo della comunicazione; *comm* comunicatore di riferimento.

Errori: Restituisce un codice di errore tra (MPI_SUCCESS, MPI_ERR_COMM, MPI_ERR_COUNT, MPI_ERR_TYPE, MPI_ERR_TAG, MPI_ERR_RANK).

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

Descrizione: Funzione di ricezione dati bloccante.

Input: *count* numero di elementi del buffer; *datatype* tipo degli elementi del buffer; *dest* idenbtificativo del processo destinatario; *tag* identificativo della comunicazione; *comm* comunicatore di riferimento.

Output: *buf* puntatore al buffer di ricezione; *status* informazioni sulla comunicazione.

Errori: Restituisce un codice di errore tra (MPI_SUCCESS, MPI_ERR_COMM, MPI_ERR_COUNT, MPI_ERR_TYPE, MPI_ERR_TAG, MPI_ERR_RANK).

```
int MPI_Abort(MPI_Comm comm, int errorcode)
```

Descrizione: Termina l'ambiente di esecuzione MPI, nello specifico tutti i processi MPI associati alla comunicazione del comunicatore *comm*.

Input: *comm* communicator di riferimento; *errorcode* codice di errore per tornare all'ambiente chiamante.

Errori: Restituisce un codice di errore tra (MPI_SUCCESS, MPI_ERR_OTHER).

4.2 Funzioni codificate

```
void help(char* program_name)
```

Descrizione: Stampa a video l'help del programma.

Input: *program_name* nome del programma.

```
double* generate_random_operands(int amount, double lower, double upper)
```

Descrizione: Genera operandi pseudo-randomici.

Input: *amount* numero di operandi pseudo-randomici da generare; *lower* limite inferiore del valore degli operandi; *upper* limite superiore del valore degli operandi.

Output: Array dinamico contenente gli operandi generati.

```
double sequential_sum(double* operands, int amount)
```

Descrizione: Effettua la somma degli operandi contenuti nell'array in input.

Input: *operands* array contente gli operandi da sommare; *amount* numero di operandi da sommare.

Output: Risultato della somma.

```
double* generate_operands(char* argv[])
```

Descrizione: Genera gli operandi in relazione agli argomenti passati in ingresso. Nello specifico, genera operandi random se il numero di operandi da generare è maggiore di 20, altrimenti legge da riga di comando.

Input: *argv* puntatore all'array degli argomenti.

Output: Array dinamico contenente gli operandi generati.

```
int check_args(int argc, char* argv[], int mpi_size)
```

Descrizione: Verifica l'integrità degli argomenti passati in ingresso al programma.

Input: *argc* puntatore al numero di argomenti; *argv* puntatore all'array degli argomenti; *mpi_size* numero di processori in uso.

Output: Un codice tra i diversi possibili (SCC_ARGS, SCC_HELP, ERR_ARGC, ERR_NO_OPERANDS, ERR_NO_STRATEGY, ERR_NO_PRINTER, ERR_TOT_OPERANDS, ERR_OPERAND, ERR_STRATEGY, ERR_PRINTER, ERR_MEMORY).

```
int* create_lookup_table_pow2(int size)
```

Descrizione: Crea un array contenente le potenze di due fino a 2^{size}.

Input: *size* dimensione dell'array.

Output: Array dinamico contenente le potenze di due fino a 2^{size}.

```
void distribute_operands(int total_operands, int total_subproblem_operands, int
    mpi_size, double* operands)
```

Descrizione: Distribuisce gli operandi dal processore P0 ai restanti. Questa funzione deve essere necessariamente richiamata dal processore P0.

Input: *total_operands* numero totale degli operandi; *total_subproblem_operands* numero degli operandi da distribuire; *mpi_size* numero di processori in uso; *operands* array contenente gli operandi da distribuire.

```
void apply_strategy_1(int mpi_rank, int mpi_size, int printer, double* sum)
```

Descrizione: Esegue la strategia per il calcolo della somma parallela con strategia I.

Input: *mpi_rank* identificativo del processore chiamante; *mpi_size* numero di processori in uso; *printer* identificativo del processore che deve stampare il risultato.

Output: *sum* risultato della somma.

```
void apply_strategy_2(int mpi_rank, int mpi_size, int printer, int log2_mpi_size,
    int* lookup_table_pow2, double* sum)
```

Descrizione: Esegue la strategia per il calcolo della somma parallela con strategia II.

Input: *mpi_rank* identificativo del processore chiamante; *mpi_size* numero di processori in uso; *printer* identificativo del processore che deve stampare il risultato; *log2_mpi_size* valore del logaritmo in base 2 di mpi_size (passi di comunicazione); *lookup_table_pow2* array contenente le potenze di due fino a $2^{log2\_mpi\_size+1}$.

Output: *sum* risultato della somma.

```
void apply_strategy_3(int mpi_rank, int log2_mpi_size, int* lookup_table_pow2,
    double* sum)
```

Descrizione: Esegue la strategia per il calcolo della somma parallela con strategia III.

Input: *mpi_rank* identificativo del processore chiamante; *log2_mpi_size* valore del logaritmo in base 2 di mpi_size (passi di comunicazione); *lookup_table_pow2* array contenente le potenze di due fino a $2^{log2\_mpi\_size+1}$.

Output: *sum* risultato della somma.

5 Descrizione dell'algoritmo

In questa sezione viene fornita una panoramica generale del funzionamento dell'algoritmo di somma di n numeri in parallelo.

5.1 Inizializzazione

Nella fase iniziale, viene inizializzato l'ambiente di lavoro del calcolatore ad architettura MIMD a memoria distribuita attraverso la libreria MPI.
Al termine di tale fase, tutti i processori sono in grado di comunicare tra loro e di effettuare operazioni di sicronizzazione.

5.2 Controllo dell'input

Il processore P_0 si occupa di effettuare specifici controlli sull'input al fine di verificare che i valori di input siano corretti.
Nel caso di errore, il software termina e viene fornito in output il codice e la descrizione dell'errore (vedere sezione 3).

5.3 Generazione operandi

Allo stato attuale, il software può ricevere gli operandi in due modi:

- se $n \leq 20$, gli operandi vengono forniti in input tramite riga di comando;
- se $n > 20$, gli operandi vengono generati randomicamente all'interno del software attraverso la funzione $srand()$.

In entrambi i casi, gli addendi sono memorizzati in un array allocato dinamicamente in base al numero degli operandi n.

5.4 Distribuzione degli operandi

Il processore 0 si occupa di distribuire gli addendi generati a tutti gli altri processori. La distribuzione dipende dal numero di processori utilizzato all'interno del software (specificato tramite il file .pbs).

```
total_locations = total_operands / total_processors;
remainder = total_operands % total_processors;
total_locations += (remainder > id_processor) ? 1 : 0;
operands = "Allocazione dinamica del vettore"
if (!id_processor)
    "Riceve addendi"
else
    "Invia addendi"
```

Codice 1: Pseudocodice della distribuzione degli addendi

Ad ogni processore è assegnato un numero di operandi pari almeno alla divisione intera tra il totale degli operandi e il numero di processori. Se il resto di tale divisione risulta diverso da 0, allora i processori per cui

$$id\_processor < total\_operands \ \% \ total\_processors \tag{1}$$

ottengono un operando in più da sommare.

La comunicazione tra processori sono implementate mediante le funzioni bloccanti della libreria MPI. Nello specifico, il processore 0 invia gli operandi contenuti in un array mediante la funzione *MPI_Send*. Gli altri processori ricevono tale array tramite la funzione *MPI_Recv*.

```
void distribute_operands(int total_operands, int total_subproblem_operands, int
    mpi_size, double* operands) {

int initial_operand_index = total_subproblem_operands;

    for(int processor = 1; processor < mpi_size; processor++) {
      total_subproblem_operands-=((total_operands % mpi_size)==processor) ? 1 : 0;
      MPI_Send(
        &operands[initial_operand_index],
        total_subproblem_operands,
        MPI_DOUBLE,
        processor,
        DIST_TAG + processor,
        MPI_COMM_WORLD
      );
      initial_operand_index += total_subproblem_operands;
    }
}
```

Codice 2: Funzione della distribuzione degli addendi

6 Strategie della somma parallela

In questo paragrafo, sono illustrati i diversi algoritmi, denominati *strategie*, adoperati per il calcolo della somma su architettura parallela. Ogni strategia ha l'obiettivo di ottenere la somma su p processori.

6.1 Strategia 1

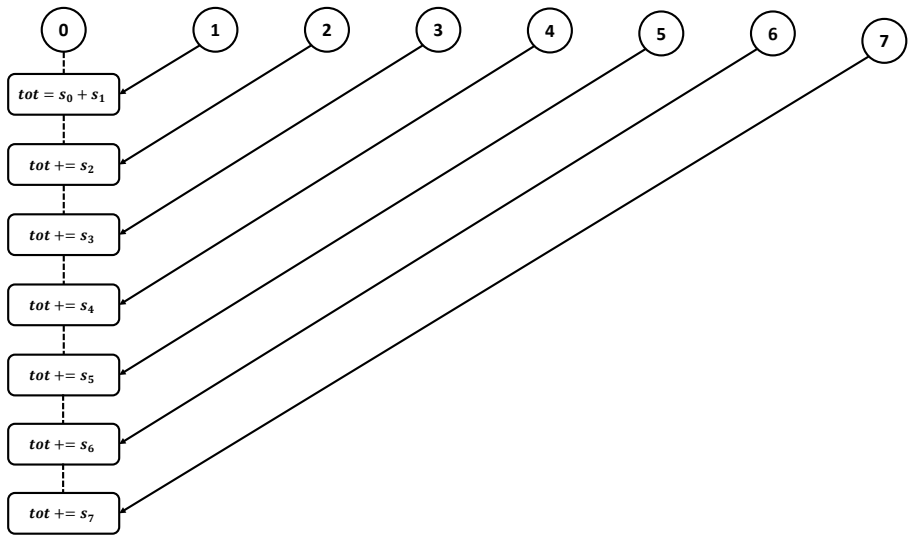


Figura 1: Strategia 1: schema con 8 processi

Tutti i processori eseguono le seguenti operazioni:

- sommare localmente gli addendi ricevuti da p_0,
- inviare il risultato della somma parziale al processo p_0

Successivamente, p_0 si occupa di sommare tutte le somme parziali e di inviare il risultato al printer, il quale stampa il risultato in output.

```
void apply_strategy_1(int mpi_rank, int mpi_size, int printer, double* sum) {

  MPI_Status status;
  double partial_sum;
  printer = (printer == -1) ? 0 : printer;
```

```
  if(mpi_rank == printer) {
    for(int processor = 0; processor < mpi_size; processor++) {
      if(processor != printer) {
        MPI_Recv(&partial_sum, 1, MPI_DOUBLE, processor, SUM_TAG + processor,
    MPI_COMM_WORLD, &status);
        *sum += partial_sum;
      }
    }
  } else
    MPI_Send(sum, 1, MPI_DOUBLE, printer, SUM_TAG + mpi_rank, MPI_COMM_WORLD);
}
```

Codice 3: Funzione della strategia 1

6.2 Strategia 2

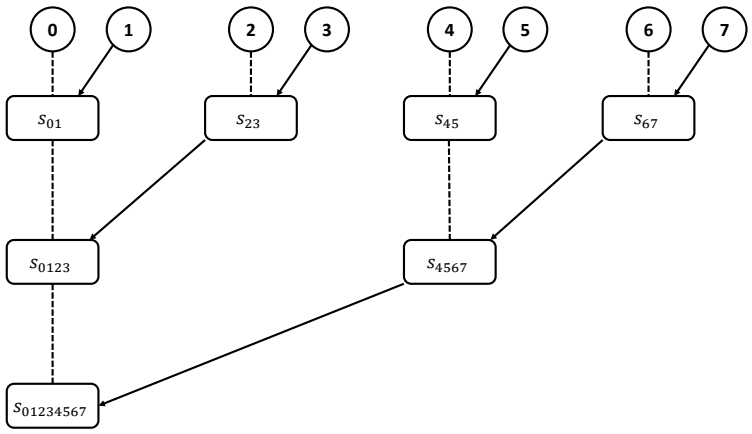


Figura 2: Strategia 2: schema con 8 processi

Come illustrato in Figura 2, con la strategia 2 si ottiene uno schema ad albero che ha il printer come radice e in cui ogni livello rappresenta un diverso passo di comunicazione. Nello specifico, si effettua un ciclo da 0 fino al $log_2(n\_processors)$ iterando, in questo modo, sui vari passi di comunicazione. Ad ogni step, si discriminano i processori che devono inviare la somma parziale da quelli che devono riceverla.
Si osservi che, differentemente dalla strategia 1, vi sono più processori che ottengono una somma parziale. Il risultato finale, successivamente, viene calcolato da un

singolo processore in maniera analoga alla strategia 1.

```
void apply_strategy_2(int mpi_rank, int mpi_size, int printer, int log2_mpi_size,
    int* lookup_table_pow2, double* sum) {

  MPI_Status status;
  double partial_sum;
  printer = (printer == -1) ? 0 : printer;
  int comm_processor;
  int alt_mpi_rank = (mpi_rank + (mpi_size - printer)) % mpi_size;

  for(int comm_step = 0; comm_step < log2_mpi_size; comm_step++) {
    if((alt_mpi_rank % lookup_table_pow2[comm_step]) == 0) {
      if((alt_mpi_rank % lookup_table_pow2[comm_step+1]) == 0) {
        comm_processor = mpi_rank + lookup_table_pow2[comm_step];
        comm_processor = (comm_processor >= mpi_size) ? (comm_processor % mpi_size)
     : comm_processor;
        MPI_Recv(&partial_sum, 1, MPI_DOUBLE, comm_processor, SUM_TAG + mpi_rank,
    MPI_COMM_WORLD, &status);
        *sum += partial_sum;
      } else {
        comm_processor = mpi_rank - lookup_table_pow2[comm_step];
        comm_processor = (comm_processor < 0) ? (comm_processor + mpi_size) :
    comm_processor;
        MPI_Send(sum, 1, MPI_DOUBLE, comm_processor, SUM_TAG + comm_processor,
    MPI_COMM_WORLD);
      }
    }
  }
}
```

Codice 4: Funzione della strategia 2

6.3 Strategia 3

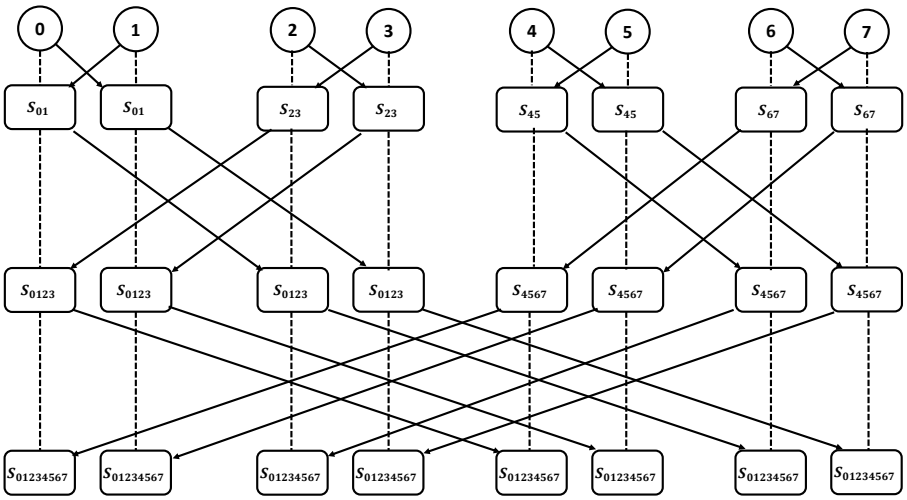


Figura 3: Strategia 3: schema con 8 processi

Nella strategia 3, ad ogni passo di comunicazione, tutti i processori partecipano ad uno scambio dati. In maniera analoga alla strategia 2, si effettua un ciclo di $log_2(n\_processors)$ iterazioni che corrispondono ai passi di comunicazione. Ad ogni iterazione, ogni processore invia e riceve somme parziali. All'ultimo step di comunicazione, differentemente da strategia 1 e strategia 2, tutti i processori dispongono della somma totale.

La stampa viene effettuata dal pid che corrisponde a quello del printer inserito in input.

```
void apply_strategy_3(int mpi_rank, int log2_mpi_size, int* lookup_table_pow2,
    double* sum) {

  MPI_Status status;
  double partial_sum;

  for(int comm_step = 0; comm_step < log2_mpi_size; comm_step++) {
    if((mpi_rank % lookup_table_pow2[comm_step+1]) < lookup_table_pow2[comm_step])
    {
      int comm_processor = mpi_rank + lookup_table_pow2[comm_step];
      MPI_Recv(&partial_sum, 1, MPI_DOUBLE, comm_processor, SUM_TAG + mpi_rank,
    MPI_COMM_WORLD, &status);
```

```
      MPI_Send(sum, 1, MPI_DOUBLE, comm_processor, SUM_TAG + comm_processor,
    MPI_COMM_WORLD);
    } else {
      int comm_processor = mpi_rank - lookup_table_pow2[comm_step];
      MPI_Send(sum, 1, MPI_DOUBLE, comm_processor, SUM_TAG + comm_processor,
    MPI_COMM_WORLD);
      MPI_Recv(&partial_sum, 1, MPI_DOUBLE, comm_processor, SUM_TAG + mpi_rank,
    MPI_COMM_WORLD, &status);
    }
    *sum += partial_sum;
  }
}
```

Codice 5: Funzione della strategia 3

7 Analisi dei tempi e delle prestazioni

La valutazione delle performance del software avvengono mediante i seguenti parametri:

1. Tempo medio impiegato: per ogni esperimento sono state effettuate 5 prove ed è stata, successivamente, considerata la media aritmetica dei tempi di ciascuna prova;

2. Speed Up: misura la riduzione del tempo di esecuzione rispetto all'algoritmo su 1 processore, $S(p) = \frac{T(1)}{T(p)}$ dove $T(1)$ rappresenta il tempo impiegato dall'algoritmo con singolo processore, mentre $T(p)$ rappresenta il tempo impiegato dall'algoritmo con p processori. Si osservi che $S(p)_{ideale} = p$;

3. Efficienza: misura quanto l'algoritmo sfrutta il parallelismo del calcolatore, $E(p) = \frac{S(p)}{p}$. Si osservi che $E(p)_{ideale} = 1$

Per misurare il tempo impiegato dal software ad effettuare le somme, è stata adoperata la funzione di libreria *MPI_Wtime*. Per calcolare il tempo massimo impiegato da tutti i processori si utilizza la funzione di libreria *MPI_Reduce*.

7.1 Dati raccolti

Di seguito vengono riportati, in formato tabellare, i tempi ottenuti dai vari test effettuati. Il numero di test totali per ogni tupla (n, s, p) è di 5, dove n rappresenta la dimensione del problema, s la strategia impiegata e p il numero di processori utilizzati. Per ogni tupla (n, s, p) è riportato il tempo medio impiegato.

	Strategia I			
Input	**P1**	**P2**	**P4**	**P8**
1,00E+03	0,000033	0,000026	0,000023	0,000078
1,00E+03	0,000031	0,000020	0,000026	0,000076
1,00E+03	0,000035	0,000038	0,000025	0,000076
1,00E+03	0,000037	0,000026	0,000028	0,000072
1,00E+03	0,000031	0,000023	0,000023	0,000079
Media	**0,0000334**	**0,0000266**	**0,0000250**	**0,0000762**

Tabella 2: Test con 10^3 operandi e strategia I

	Strategia I			
Input	**P1**	**P2**	**P4**	**P8**
1,00E+04	0,000067	0,000047	0,000031	0,000041
1,00E+04	0,000081	0,000052	0,000037	0,000077
1,00E+04	0,000072	0,000047	0,000034	0,000078
1,00E+04	0,000071	0,000052	0,000034	0,000045
1,00E+04	0,000071	0,000047	0,000034	0,000041
Media	**0,0000724**	**0,0000490**	**0,0000340**	**0,0000564**

Tabella 3: Test con 10^4 operandi e strategia I

	Strategia I			
Input	**P1**	**P2**	**P4**	**P8**
1,00E+05	0,000412	0,000230	0,000131	0,000122
1,00E+05	0,000410	0,000234	0,000123	0,000124
1,00E+05	0,000465	0,000234	0,000129	0,000117
1,00E+05	0,000448	0,000234	0,000123	0,000122
1,00E+05	0,000394	0,000258	0,000128	0,000125
Media	**0,0004258**	**0,0002380**	**0,0001270**	**0,0001220**

Tabella 4: Test con 10^5 operandi e strategia I

	Strategia I			
Input	**P1**	**P2**	**P4**	**P8**
1,00E+06	0,003821	0,002153	0,001094	0,000596
1,00E+06	0,004233	0,002153	0,001090	0,000598
1,00E+06	0,004143	0,002154	0,001091	0,000595
1,00E+06	0,004070	0,002159	0,001092	0,000594
1,00E+06	0,004015	0,002161	0,001092	0,000558
Media	**0,0040564**	**0,0021560**	**0,0010918**	**0,0005882**

Tabella 5: Test con 10^6 operandi e strategia I

	Strategia I			
Input	**P1**	**P2**	**P4**	**P8**
1,00E+07	0,041058	0,021389	0,011040	0,00547
1,00E+07	0,040668	0,021373	0,010696	0,006067
1,00E+07	0,040275	0,021369	0,010707	0,005473
1,00E+07	0,041882	0,021411	0,010694	0,005445
1,00E+07	0,042340	0,021370	0,010699	0,006518
Media	**0,0412446**	**0,0213824**	**0,0107670**	**0,0057946**

Tabella 6: Test con 10^7 operandi e strategia I

	Strategia I			
Input	**P1**	**P2**	**P4**	**P8**
1,00E+08	0,413574	0,213229	0,106777	0,05357
1,00E+08	0,412241	0,213449	0,106644	0,053361
1,00E+08	0,437119	0,213163	0,106522	0,053403
1,00E+08	0,403671	0,213404	0,106681	0,053360
1,00E+08	0,416429	0,213331	0,106668	0,053393
Media	**0,4166068**	**0,2133152**	**0,1066580**	**0,0534174**

Tabella 7: Test con 10^8 operandi e strategia I

	Strategia II			
Input	**P1**	**P2**	**P4**	**P8**
1,00E+03	0,000033	0,000038	0,000027	0,000051
1,00E+03	0,000031	0,000024	0,000029	0,000043
1,00E+03	0,000035	0,000020	0,000028	0,000047
1,00E+03	0,000037	0,000014	0,000041	0,000044
1,00E+03	0,000031	0,000024	0,000023	0,000033
Media	**0,0000334**	**0,0000240**	**0,0000296**	**0,0000436**

Tabella 8: Test con 10^3 operandi e strategia II

	Strategia II			
Input	**P1**	**P2**	**P4**	**P8**
1,00E+04	0,000067	0,000048	0,000035	0,000045
1,00E+04	0,000081	0,000048	0,000044	0,000051
1,00E+04	0,000072	0,000054	0,000037	0,000038
1,00E+04	0,000071	0,000048	0,000037	0,000046
1,00E+04	0,000071	0,000048	0,000037	0,000045
Media	**0,0000724**	**0,0000492**	**0,0000380**	**0,0000450**

Tabella 9: Test con 10^4 operandi e strategia II

	Strategia II			
Input	**P1**	**P2**	**P4**	**P8**
1,00E+05	0,000412	0,000230	0,000133	0,000084
1,00E+05	0,000410	0,000229	0,000125	0,000086
1,00E+05	0,000465	0,000235	0,000124	0,000094
1,00E+05	0,000448	0,000230	0,000133	0,000096
1,00E+05	0,000394	0,000236	0,000130	0,000095
Media	**0,0004258**	**0,0002320**	**0,0001290**	**0,0000910**

Tabella 10: Test con 10^5 operandi e strategia II

	Strategia II			
Input	**P1**	**P2**	**P4**	**P8**
1,00E+06	0,003821	0,002153	0,001098	0,000566
1,00E+06	0,004233	0,002161	0,001097	0,000563
1,00E+06	0,004143	0,002151	0,001100	0,000565
1,00E+06	0,004070	0,002155	0,001097	0,000565
1,00E+06	0,004015	0,002154	0,001096	0,000566
Media	**0,0040564**	**0,0021548**	**0,0010976**	**0,0005650**

Tabella 11: Test con 10^6 operandi e strategia II

	Strategia II			
Input	**P1**	**P2**	**P4**	**P8**
1,00E+07	0,041058	0,021388	0,010732	0,005418
1,00E+07	0,040668	0,021376	0,010747	0,005417
1,00E+07	0,040275	0,021373	0,010744	0,005417
1,00E+07	0,041882	0,021391	0,010749	0,005411
1,00E+07	0,042340	0,021399	0,010746	0,005419
Media	**0,0412446**	**0,0213854**	**0,0107436**	**0,0054164**

Tabella 12: Test con 10^7 operandi e strategia II

	Strategia II			
Input	**P1**	**P2**	**P4**	**P8**
1,00E+08	0,413574	0,213379	0,106905	0,053436
1,00E+08	0,412241	0,213157	0,106776	0,053551
1,00E+08	0,437119	0,213293	0,106660	0,053475
1,00E+08	0,403671	0,213178	0,106756	0,053465
1,00E+08	0,416429	0,213439	0,106713	0,053454
Media	**0,4166068**	**0,2132892**	**0,1067620**	**0,0534762**

Tabella 13: Test con 10^8 operandi e strategia II

	Strategia III			
Input	**P1**	**P2**	**P4**	**P8**
1,00E+03	0,000033	0,000028	0,000034	0,000063
1,00E+03	0,000031	0,000019	0,000062	0,000059
1,00E+03	0,000035	0,000046	0,000049	0,000062
1,00E+03	0,000037	0,000029	0,000033	0,000060
1,00E+03	0,000031	0,000018	0,000034	0,000064
Media	**0,0000334**	**0,0000280**	**0,0000424**	**0,0000616**

Tabella 14: Test con 10^3 operandi e strategia III

	Strategia III			
Input	**P1**	**P2**	**P4**	**P8**
1,00E+04	0,000067	0,000047	0,000045	0,000064
1,00E+04	0,000081	0,000049	0,000047	0,000057
1,00E+04	0,000072	0,000049	0,000047	0,000063
1,00E+04	0,000071	0,000048	0,000047	0,000067
1,00E+04	0,000071	0,000047	0,000047	0,000063
Media	**0,0000724**	**0,0000480**	**0,0000466**	**0,0000628**

Tabella 15: Test con 10^4 operandi e strategia III

	Strategia III			
Input	**P1**	**P2**	**P4**	**P8**
1,00E+05	0,000412	0,000230	0,000142	0,0000970
1,00E+05	0,000410	0,000236	0,000138	0,0001050
1,00E+05	0,000465	0,000232	0,000143	0,0001100
1,00E+05	0,000448	0,000264	0,000142	0,0000980
1,00E+05	0,000394	0,000242	0,000137	0,0000970
Media	**0,0004258**	**0,0002410**	**0,0001404**	**0,0001014**

Tabella 16: Test con 10^5 operandi e strategia III

	Strategia III			
Input	**P1**	**P2**	**P4**	**P8**
1,00E+06	0,003821	0,002161	0,001101	0,000573
1,00E+06	0,004233	0,002161	0,001107	0,000578
1,00E+06	0,004143	0,002154	0,001102	0,000575
1,00E+06	0,004070	0,002158	0,001102	0,000576
1,00E+06	0,004015	0,002157	0,001102	0,000577
Media	**0,0040564**	**0,0021580**	**0,0011028**	**0,0005758**

Tabella 17: Test con 10^6 operandi e strategia III

	Strategia III			
Input	**P1**	**P2**	**P4**	**P8**
1,00E+07	0,041058	0,021388	0,010762	0,005583
1,00E+07	0,040668	0,021373	0,010759	0,005422
1,00E+07	0,040275	0,021380	0,01080	0,005429
1,00E+07	0,041882	0,021385	0,010744	0,005416
1,00E+07	0,042340	0,021387	0,010730	0,005426
Media	**0,0412446**	**0,0213830**	**0,0107590**	**0,0054552**

Tabella 18: Test con 10^7 operandi e strategia III

	Strategia III			
Input	**P1**	**P2**	**P4**	**P8**
1,00E+08	0,413574	0,213721	0,106836	0,053589
1,00E+08	0,412241	0,213368	0,106620	0,053489
1,00E+08	0,437119	0,213344	0,106713	0,053485
1,00E+08	0,403671	0,213417	0,106702	0,053624
1,00E+08	0,416429	0,213382	0,106657	0,053477
Media	**0,4166068**	**0,2134460**	**0,1067056**	**0,0535328**

Tabella 19: Test con 10^8 operandi e strategia III

7.2 Analisi dei tempi con 10^3 operandi

	Strategia I		
Processori	**Tempo medio**	**Speed Up**	**Efficienza**
1	0,0000334	1	1
2	0,0000266	1,255	0,627
4	0,0000250	1,336	0,334
8	0,0000762	0,438	0,054

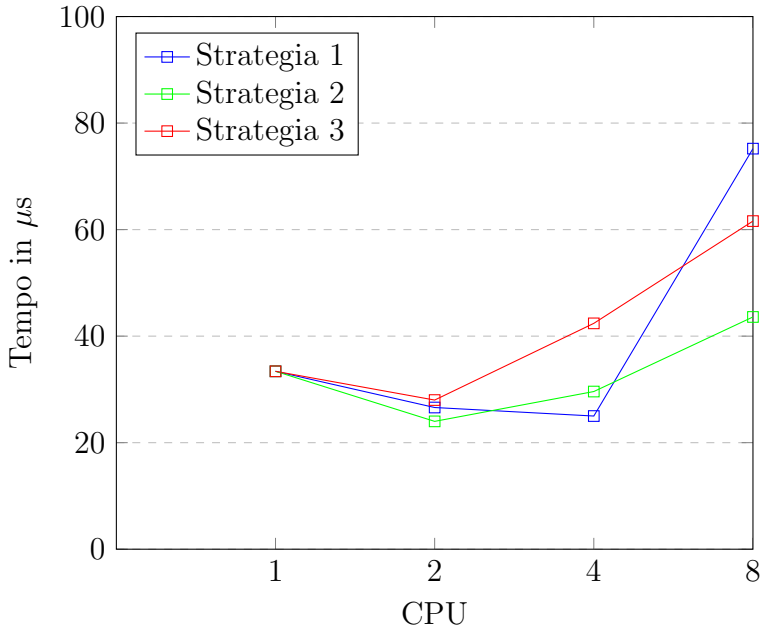
Tabella 20: Dati ottenuti con 10^3 operandi e strategia I

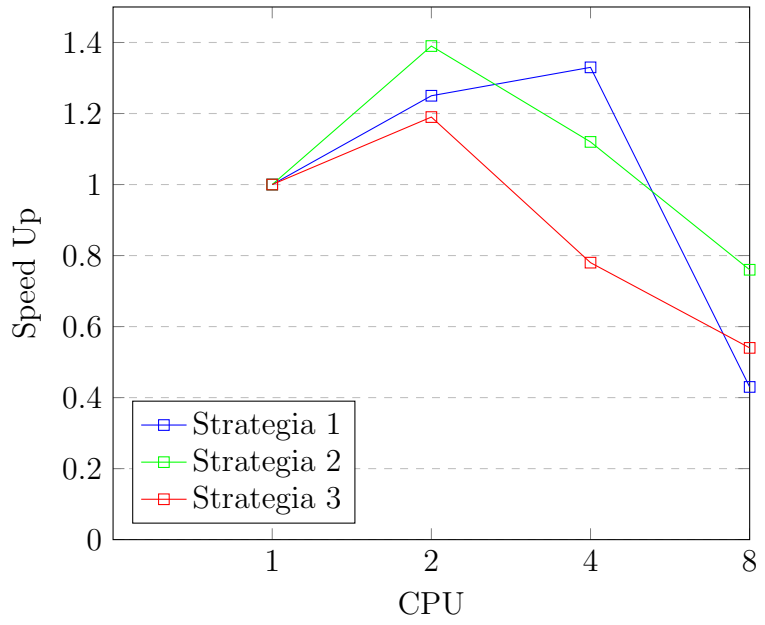
	Strategia II		
Processori	**Tempo medio**	**Speed Up**	**Efficienza**
1	0,0000334	1	1
2	0,0000240	1,391	0,695
4	0,0000296	1,128	0,282
8	0,0000436	0,766	0,095

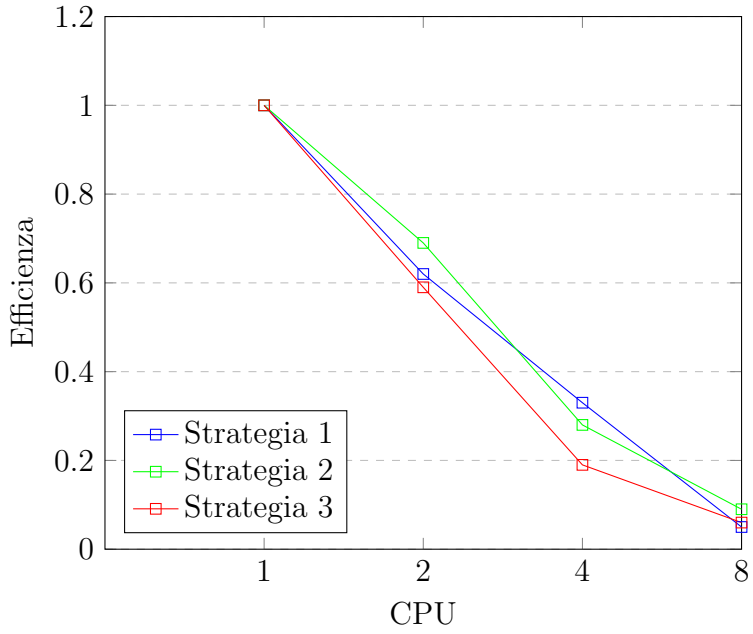
Tabella 21: Dati ottenuti con 10^3 operandi e strategia II

	Strategia III		
Processori	**Tempo medio**	**Speed Up**	**Efficienza**
1	0,0000334	1	1
2	0,0000280	1,192	0,596
4	0,0000424	0,787	0,196
8	0,0000616	0,542	0,067

Tabella 22: Dati ottenuti con 10^3 operandi e strategia III


Tempo in μs
100
80
60
40
20
0
1
2
4
8
CPU
Strategia 1
Strategia 2
Strategia 3


Speed Up
1.4
1.2
1
0.8
0.6
0.4
0.2
0
1
2
4
8
CPU
Strategia 1
Strategia 2
Strategia 3



7.3 Analisi dei tempi con 10^4 operandi

	Strategia I		
Processori	**Tempo medio**	**Speed Up**	**Efficienza**
1	0,0000724	1	1
2	0,0000490	1,477	0,738
4	0,0000340	2,129	0,532
8	0,0000564	1,283	0,160

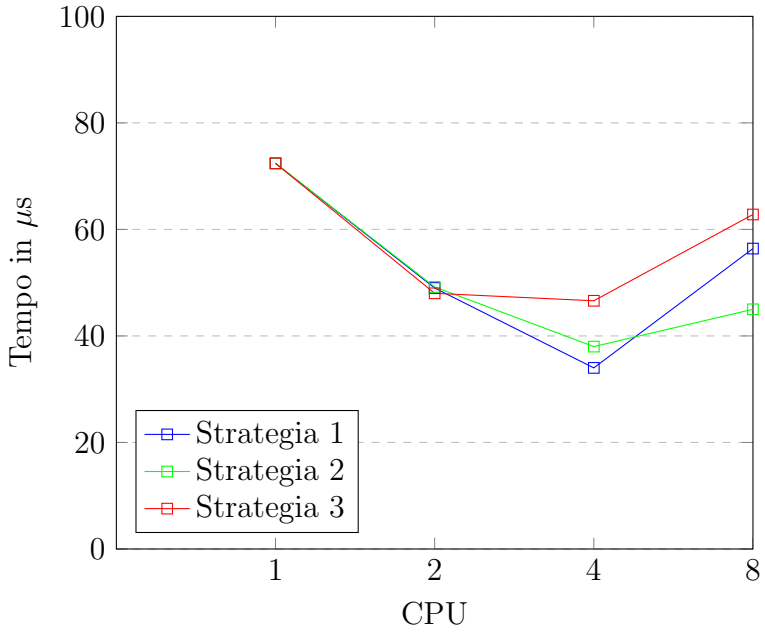
Tabella 23: Dati ottenuti con 10^4 operandi e strategia I

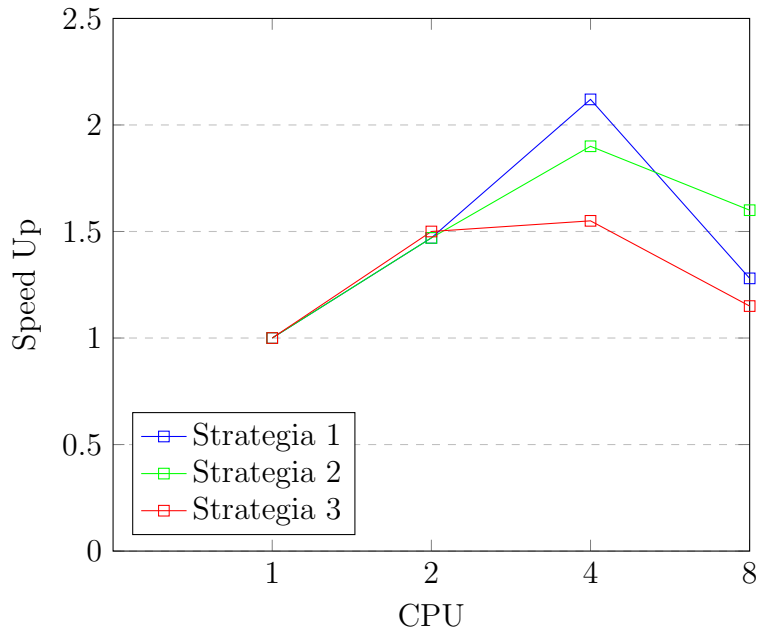
	Strategia II		
Processori	**Tempo medio**	**Speed Up**	**Efficienza**
1	0,0000724	1	1
2	0,0000492	1,471	0,735
4	0,0000380	1,905	0,476
8	0,0000450	1,608	0,201

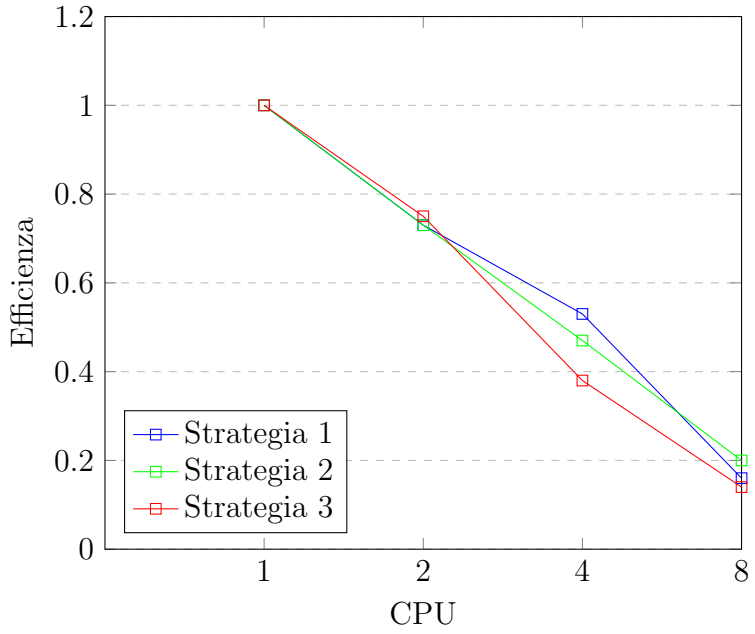
Tabella 24: Dati ottenuti con 10^4 operandi e strategia II

	Strategia III		
Processori	**Tempo medio**	**Speed Up**	**Efficienza**
1	0,0000724	1	1
2	0,0000480	1,508	0,754
4	0,0000466	1,553	0,388
8	0,0000628	1,152	0,144

Tabella 25: Dati ottenuti con 10^4 operandi e strategia III







7.4 Analisi dei tempi con 10^5 operandi

	Strategia I		
Processori	**Tempo medio**	**Speed Up**	**Efficienza**
1	0,0004258	1	1
2	0,0002380	1,789	0,894
4	0,0001270	3,358	0,839
8	0,0001220	3,490	0,436

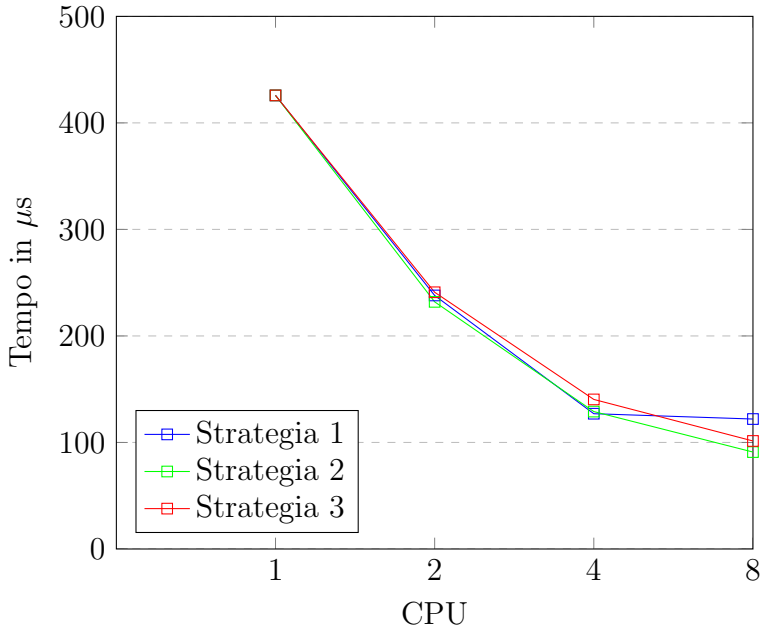
Tabella 26: Dati ottenuti con 10^5 operandi e strategia I

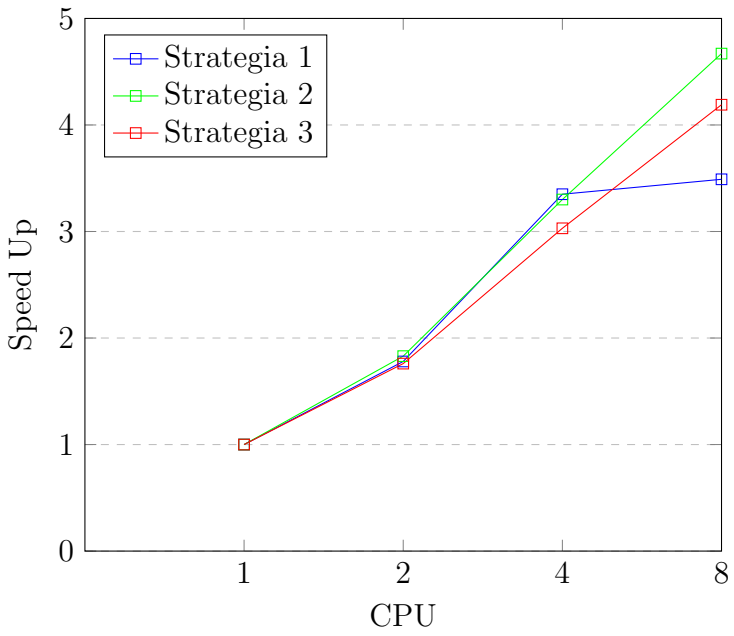
	Strategia II		
Processori	**Tempo medio**	**Speed Up**	**Efficienza**
1	0,0004258	1	1
2	0,0002320	1,835	0,917
4	0,0001290	3,300	0,825
8	0,0000910	4,679	0,584

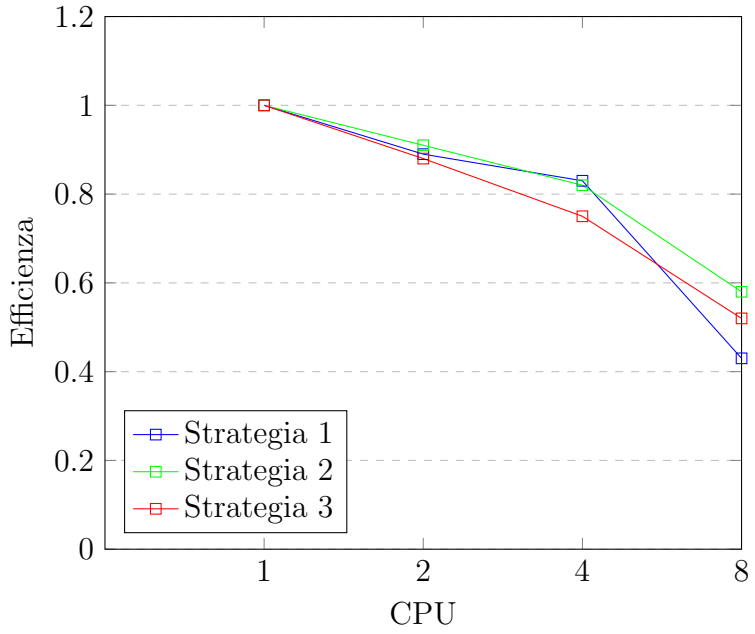
Tabella 27: Dati ottenuti con 10^5 operandi e strategia II

	Strategia III		
Processori	**Tempo medio**	**Speed Up**	**Efficienza**
1	0,0004258	1	1
2	0,0002410	1,766	0,883
4	0,0001404	3,032	0,758
8	0,0001014	4,199	0,524

Tabella 28: Dati ottenuti con 10^5 operandi e strategia III







7.5 Analisi dei tempi con 10^6 operandi

	Strategia I		
Processori	**Tempo medio**	**Speed Up**	**Efficienza**
1	0,0040564	1	1
2	0,0021560	1,881	0,940
4	0,0010918	3,715	0,928
8	0,0005882	6,896	0,862

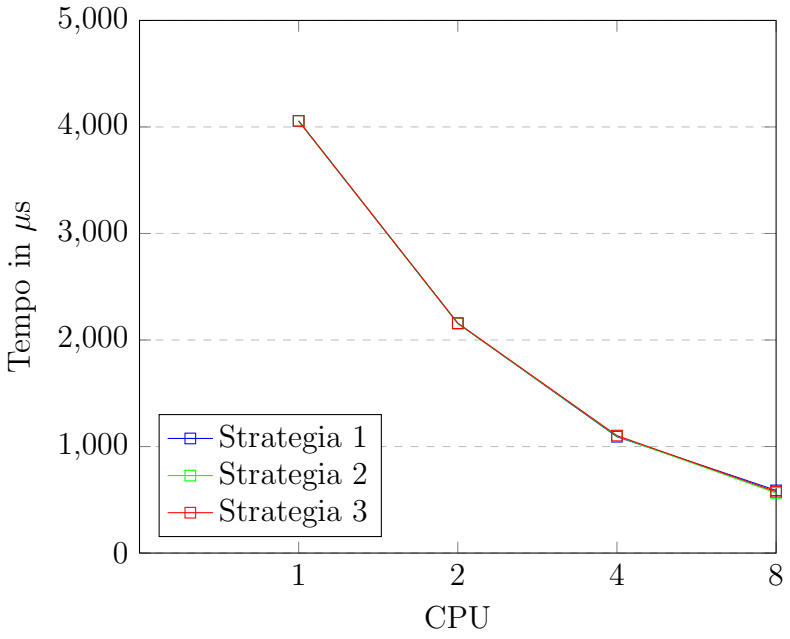
Tabella 29: Dati ottenuti con 10^6 operandi e strategia I

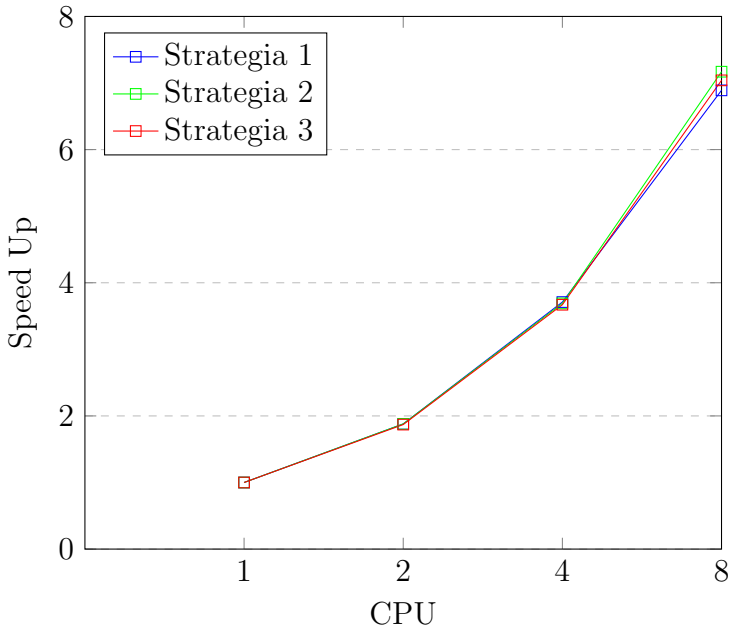
	Strategia II		
Processori	**Tempo medio**	**Speed Up**	**Efficienza**
1	0,0040564	1	1
2	0,0021548	1,882	0,941
4	0,0010976	3,695	0,923
8	0,0005650	7,179	0,897

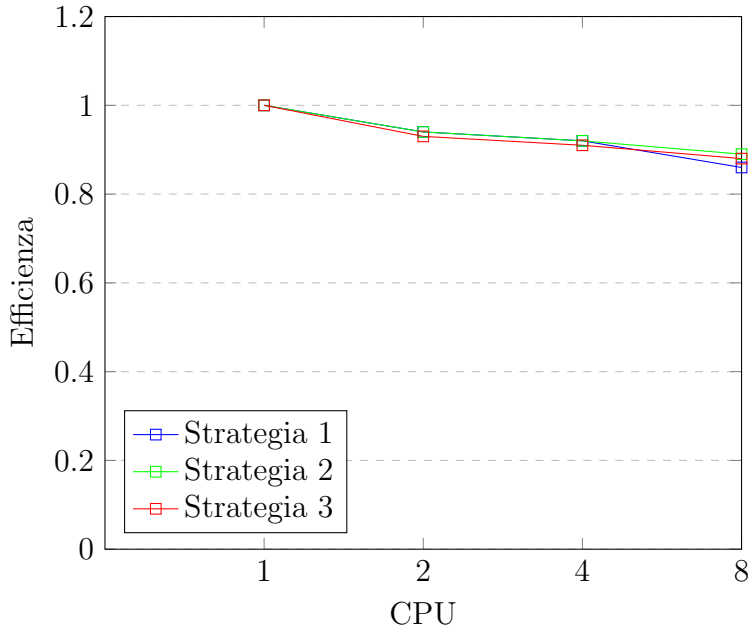
Tabella 30: Dati ottenuti con 10^6 operandi e strategia II

	Strategia III		
Processori	**Tempo medio**	**Speed Up**	**Efficienza**
1	0,0040564	1	1
2	0,0021582	1,879	0,939
4	0,0011028	3,678	0,919
8	0,0005758	7,044	0,880

Tabella 31: Dati ottenuti con 10^6 operandi e strategia III







7.6 Analisi dei tempi con 10^7 operandi

	Strategia I		
Processori	**Tempo medio**	**Speed Up**	**Efficienza**
1	0,0412446	1	1
2	0,0213824	1,928	0,964
4	0,0107670	3,830	0,957
8	0,0057946	7,117	0,889

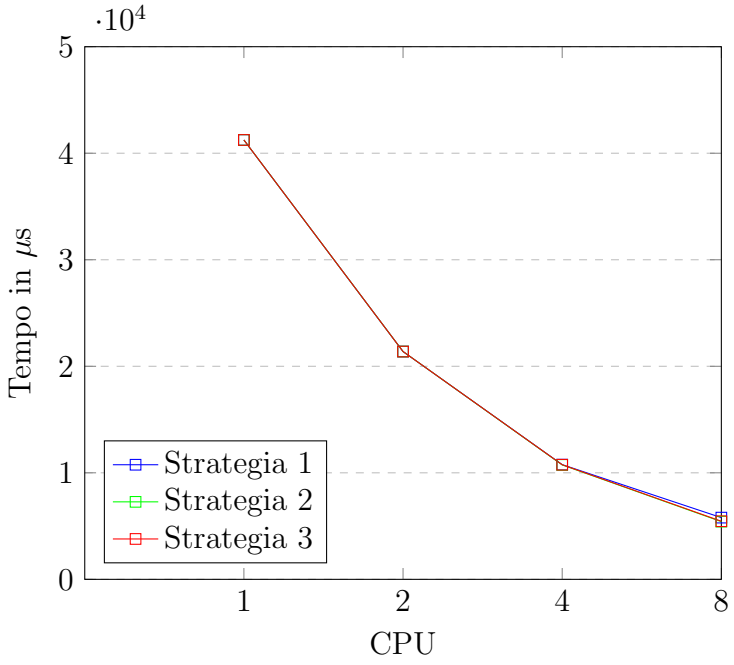
Tabella 32: Dati ottenuti con 10^7 operandi e strategia I

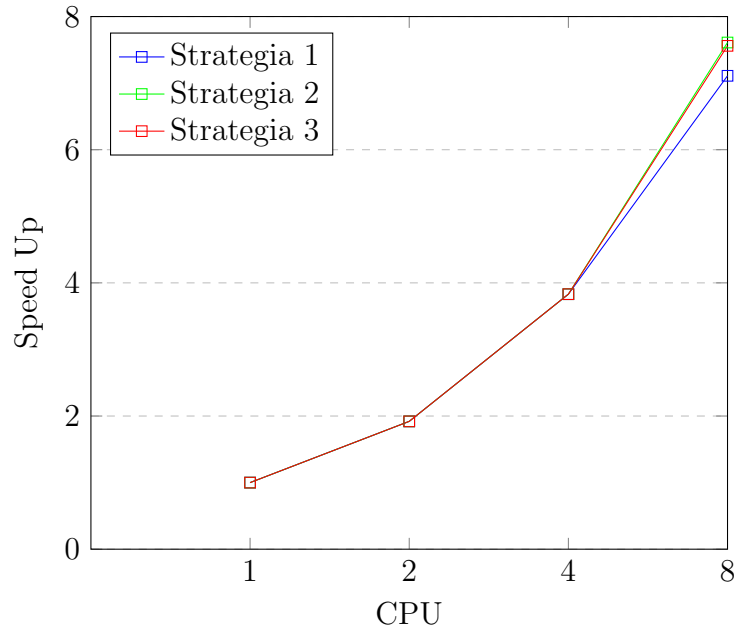
	Strategia II		
Processori	**Tempo medio**	**Speed Up**	**Efficienza**
1	0,0412446	1	1
2	0,0213854	1,928	0,964
4	0,0107436	3,838	0,959
8	0,0054164	7,614	0,951

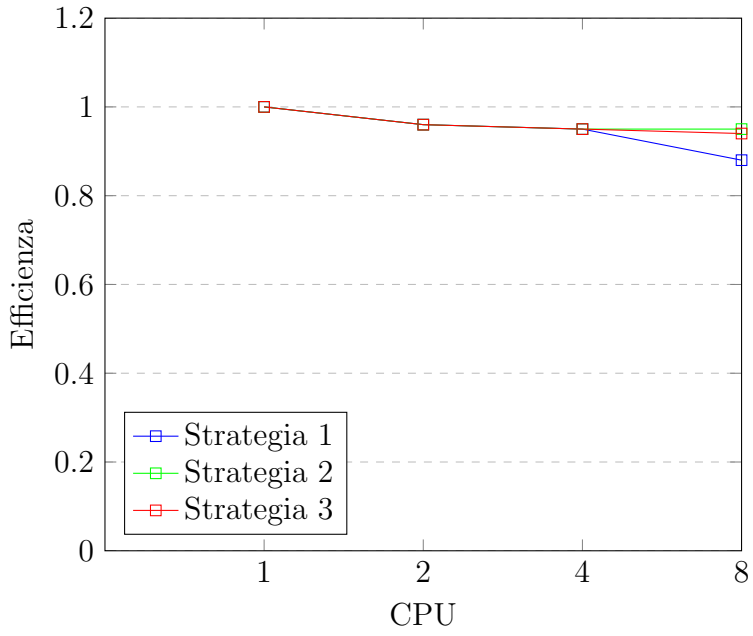
Tabella 33: Dati ottenuti con 10^7 operandi e strategia II

	Strategia III		
Processori	**Tempo medio**	**Speed Up**	**Efficienza**
1	0,0412446	1	1
2	0,0213830	1,928	0,964
4	0,0107590	3,833	0,958
8	0,0054552	7,560	0,945

Tabella 34: Dati ottenuti con 10^7 operandi e strategia III




Strategia 1
Strategia 2
Strategia 3
Speed Up
CPU
0
2
4
6
8
1
2
4
8


Strategia 1
Strategia 2
Strategia 3
Efficienza
CPU
0
0.2
0.4
0.6
0.8
1
1.2
1
2
4
8

7.7 Analisi dei tempi con 10^8 operandi

	Strategia I		
Processori	**Tempo medio**	**Speed Up**	**Efficienza**
1	0,4166068	1	1
2	0,2133152	1,953	0,976
4	0,1066580	3,905	0,976
8	0,0534174	7,799	0,974

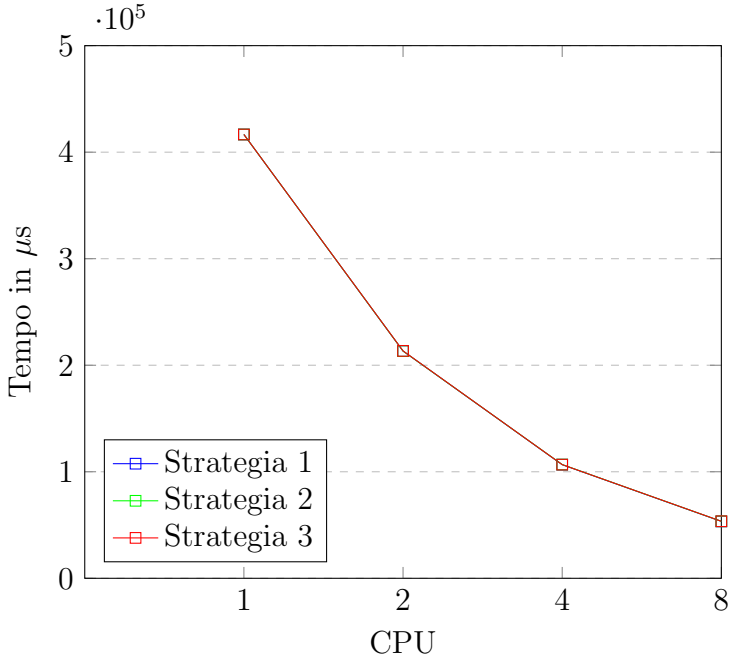
Tabella 35: Dati ottenuti con 10^8 operandi e strategia I

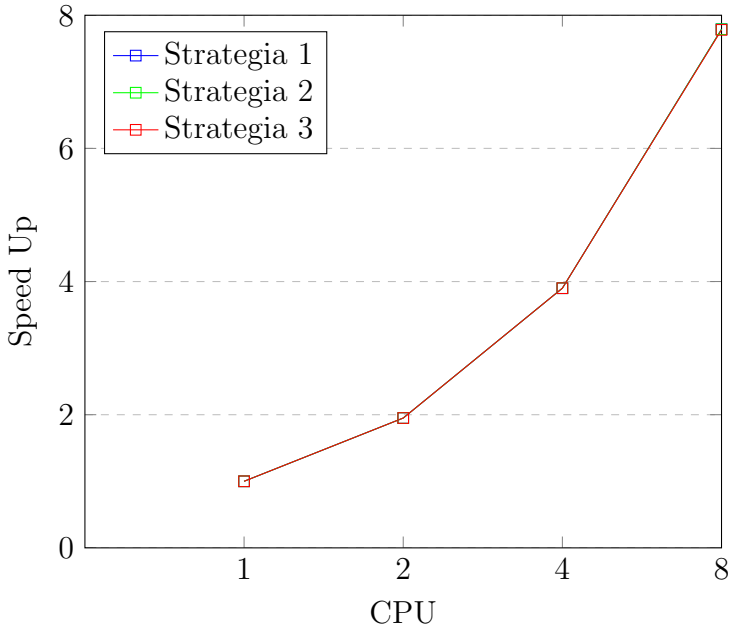
	Strategia II		
Processori	**Tempo medio**	**Speed Up**	**Efficienza**
1	0,4166068	1	1
2	0,2132892	1,953	0,976
4	0,106762	3,902	0,975
8	0,0534762	7,790	0,973

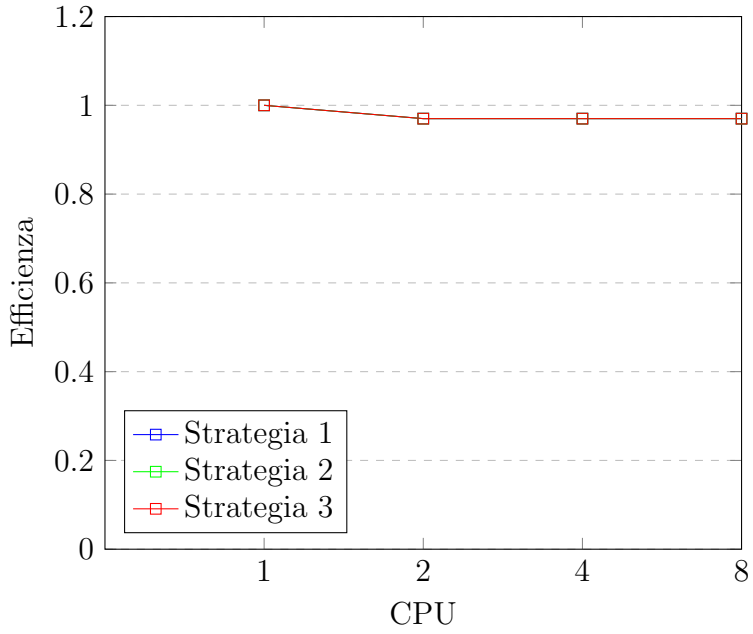
Tabella 36: Dati ottenuti con 10^8 operandi e strategia II

	Strategia III		
Processori	**Tempo medio**	**Speed Up**	**Efficienza**
1	0,4166068	1	1
2	0,2134460	1,951	0,975
4	0,1067056	3,904	0,976
8	0,0535328	7,782	0,972

Tabella 37: Dati ottenuti con 10^8 operandi e strategia III


$\cdot 10^5$
5
4
3
2
1
0
Tempo in μs
Strategia 1
Strategia 2
Strategia 3
1
2
4
8
CPU


8
6
4
2
0
Speed Up
Strategia 1
Strategia 2
Strategia 3
1
2
4
8
CPU



7.8 Considerazioni sui risultati ottenuti

In questo studio, sono stati condotti diversi test tenendo in considerazione i seguenti parametri:

- n: numero totale di addendi reali da sommare. Tale valore assume valori in $N = \{10^e : e \in [3..8]\}$;
- p: numero di processori. Tale parametro può assumere valori in $P = \{2, 4, 6, 8\}$. Si sono scelte potenze di due al fine di poter impiegare tutte le strategie possibili;
- s: indica il tipo di strategia adottato per la somma. Tale parametro può assumere valori in $S = \{1, 2, 3\}$.

Ogni test è identificato dalla tupla (n, s, p, i) dove i rappresenta il numero del test ($i \in [1..5]$). Pertanto, essendo $(n, s, p, i) \in \{(n, s, p, i) : n \in N \land\ s \in S \land p \in P \land i \in [1..5]\} := N \times S \times P \times [1..5] = T$, il numero di test totali effettuati è $|T| = 360$. Innanzitutto osserviamo che tutte le strategie per input alti ($> 10^5$) si comportano in modo analogo: tempi medi, speed up ed efficienza convergono pressoché allo stesso

valore. Ciò suggerisce che per input alti la strategia adottata è ininfluente.

Con $n = 10^3$ e $n = 10^4$ speed up ed efficienza risultano molto bassi con qualsiasi strategia e qualsiasi numero di CPU utilizzate (esclusa 1 CPU); pertanto l'overhead, per tali valori di input, è oltremodo elevato.

Sulla base di quanto evidenziato dai grafici, utilizzare un numero di processori elevato per input piccoli non risulta efficiente.

Per $n = 10^5$, overhead minore e maggiore efficienza si ottengono con $p = 4$ registrando una notevole riduzione del tempo di esecuzione.

Per $n > 10^6$ lo speed up con p maggiori raggiunge pressoché lo speed up ideale, l'efficienza sfiora il valore ideale 1 e il tempo di esecuzione è oltremodo ridotto; pertanto, è proprio per questi valori di input che i vantaggi del calcolo parallelo appaiono evidenti.

8 Esempi d'uso

8.1 Esecuzione dei test

L'esecuzione dei test sul cluster avviene mediante l'esecuzione del seguente script PBS. Per variare il numero di processori da impiegare è necessario modificare la variabile NCPU, la quale, nel seguente PBS, è impostata ad 1 (singolo processore). Si noti che per semplicità di verifica i test sono stati effettuati generando esclusivamente operandi pari a 1:

```
1  #!/bin/bash
2
3  ############################
4  #                          #
5  #    The PBS Directive     #
6  #                          #
7  ############################
8
9  #PBS -q studenti
10 #PBS -N SommaNumeri
```

```
11 #PBS -o SommaNumeri.out
12 #PBS -e SommaNumeri.err
13 #PBS -l nodes=8:ppn=8
14
15 # -q cosa sui va eseguito il job
16 # -l numero di nodi richiesti
17 # -N nome job (stesso del file pbs)
18 # -o, -e nome file contente l'output o error
19
20
21 ############################
22 #                          #
23 #   Informazioni sul Job   #
24 #                          #
25 ############################
26
27 sort -u $PBS_NODEFILE > hostlist
28
29 NCPU=1
30 echo ----------------------------
31 echo 'This Job is allocated on '${NCPU}' cpu(s)'
32 echo 'Job is running on node(s):'
33 cat hostlist
34
35 PBS_O_WORKDIR=$PBS_O_HOME/SommaNumeri
36 echo ----------------------------------
37 echo PBS: qsub is running on $PBS_O_HOST
38 echo PBS: originating queue is $PBS_O_QUEUE
39 echo PBS: executing queue is $PBS_QUEUE
40 echo PBS: working directory is $PBS_O_WORKDIR
41 echo PBS: execution mode is $PBS_ENVIRONMENT
42 echo PBS: job identifier is $PBS_JOBID
43 echo PBS: job name is $PBS_JOBNAME
44 echo PBS: node file is $PBS_NODEFILE
45 echo PBS: current home directory is $PBS_O_HOME
46 echo PBS: PATH = $PBS_O_PATH
47 echo ----------------------------------
48
49
50 ############################
51 #                          #
52 #       Compilazione       #
53 #                          #
```

```
############################

echo "Compilazione con /usr/lib64/openmpi/1.4-gcc/bin/mpicc -lm -std=c99 -o
    $PBS_O_WORKDIR/SommaNumeri $PBS_O_WORKDIR/SommaNumeri.c"
/usr/lib64/openmpi/1.4-gcc/bin/mpicc -lm -std=c99 -o $PBS_O_WORKDIR/SommaNumeri
    $PBS_O_WORKDIR/SommaNumeri.c


############################
#                          #
#   Esecuzione dei test    #
#                          #
############################

for operands in 1000 10000 100000 1000000 10000000 100000000
do
  for test in {1..5}
  do
    for strategy in {1..3}
    do
            echo -e "\n\n\n### INI TEST N$TEST - [Operandi: $operands] [Strategia:
    $strategy] ###\n"
      /usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile $PBS_NODEFILE -np $NCPU
    $PBS_O_WORKDIR/SommaNumeri -o $operands -s $strategy -p 0
            echo "### END TEST N$TEST - [Operandi: $operands] [Strategia: $strategy
    ] ###"
    done
  done
done
```

Il risultato ottenuto dall'esecuzione dello script PBS è contenuto nel file SommaNumeri.out mentre eventuali errori nel file SommaNumeri.err:

```
----------------------------
This Job is allocated on 8 cpu(s)
Job is running on node(s):
wn273.scope.unina.it
wn274.scope.unina.it
wn275.scope.unina.it
wn276.scope.unina.it
wn277.scope.unina.it
wn278.scope.unina.it
wn279.scope.unina.it
```

```
wn280.scope.unina.it
----------------------------------
PBS: qsub is running on ui-studenti.scope.unina.it
PBS: originating queue is studenti
PBS: executing queue is studenti
PBS: working directory is /homes/DMA/PDC/2021/RMNMRC98R/SommaNumeri
PBS: execution mode is PBS_BATCH
PBS: job identifier is 3988087.torque02.scope.unina.it
PBS: job name is SommaNumeri
PBS: node file is /var/spool/pbs/aux//3988087.torque02.scope.unina.it
PBS: current home directory is /homes/DMA/PDC/2021/RMNMRC98R
PBS: PATH = /usr/lib64/openmpi/1.2.7-gcc/bin:/usr/kerberos/bin:/opt/exp_soft/unina.
    it/intel/composer_xe_2013_sp1.3.174/bin/intel64:/opt/exp_soft/unina.it/intel/
    composer_xe_2013_sp1.3.174/mpirt/bin/intel64:/opt/exp_soft/unina.it/intel/
    composer_xe_2013_sp1.3.174/bin/intel64:/opt/exp_soft/unina.it/intel/
    composer_xe_2013_sp1.3.174/bin/intel64_mic:/opt/exp_soft/unina.it/intel/
    composer_xe_2013_sp1.3.174/debugger/gui/intel64:/opt/d-cache/srm/bin:/opt/d-
    cache/dcap/bin:/opt/edg/bin:/opt/glite/bin:/opt/globus/bin:/opt/lcg/bin:/usr/
    local/bin:/bin:/usr/bin:/opt/exp_soft/HADOOP/hadoop-1.0.3/bin:/opt/exp_soft/
    unina.it/intel/composerxe/bin/intel64/:/opt/exp_soft/unina.it/MPJExpress/mpj-
    v0_38/bin:/homes/DMA/PDC/2021/RMNMRC98R/bin
----------------------------------
Compilazione con /usr/lib64/openmpi/1.4-gcc/bin/mpicc -lm -std=c99 -o /homes/DMA/
    PDC/2021/RMNMRC98R/SommaNumeri/SommaNumeri /homes/DMA/PDC/2021/RMNMRC98R/
    SommaNumeri/SommaNumeri.c



### INI TEST N - [Operandi: 1000] [Strategia: 1] ###

 >> [P0] Result of the sum: 1000.000000.

 >> Maximum time detected: 0.000078.

### END TEST N - [Operandi: 1000] [Strategia: 1] ###


# ...
# ...
# ...


### INI TEST N - [Operandi: 100000000] [Strategia: 3] ###
```

```
 >> [P0] Result of the sum: 100000000.000000.

 >> Maximum time detected: 0.053624.

### END TEST N - [Operandi: 100000000] [Strategia: 3] ###
```

8.2 Esecuzione utente

L'esecuzione del programma sul cluster con argomenti variabili a discrezione dell'utente avviene mediante l'esecuzione del seguente script PBS. Proprio come in 8.1, per variare il numero di processori da impiegare è necessario modificare la variabile NCPU:

```
#!/bin/bash

#############################
#                           #
#    The PBS Directive      #
#                           #
#############################

#PBS -q studenti
#PBS -N SommaNumeri
#PBS -o SommaNumeri.out
#PBS -e SommaNumeri.err
#PBS -l nodes=8:ppn=8

# -q cosa sui va eseguito il job
# -l numero di nodi richiesti
# -N nome job (stesso del file pbs)
# -o, -e nome file contente l'output o error


#############################
#                           #
#   Informazioni sul Job    #
#                           #
#############################

sort -u $PBS_NODEFILE > hostlist
```

```
28
29 NCPU=‘wc -l < hostlist‘
30 echo ------------------------------
31 echo 'This Job is allocated on '${NCPU}' cpu(s)'
32 echo 'Job is running on node(s):'
33 cat hostlist
34
35 PBS_O_WORKDIR=$PBS_O_HOME/SommaNumeri
36 echo ----------------------------------
37 echo PBS: qsub is running on $PBS_O_HOST
38 echo PBS: originating queue is $PBS_O_QUEUE
39 echo PBS: executing queue is $PBS_QUEUE
40 echo PBS: working directory is $PBS_O_WORKDIR
41 echo PBS: execution mode is $PBS_ENVIRONMENT
42 echo PBS: job identifier is $PBS_JOBID
43 echo PBS: job name is $PBS_JOBNAME
44 echo PBS: node file is $PBS_NODEFILE
45 echo PBS: current home directory is $PBS_O_HOME
46 echo PBS: PATH = $PBS_O_PATH
47 echo ----------------------------------
48
49
50 ############################
51 #                          #
52 #       Compilazione       #
53 #                          #
54 ############################
55
56 echo "Compilazione con /usr/lib64/openmpi/1.4-gcc/bin/mpicc -lm -std=c99 -o
       $PBS_O_WORKDIR/SommaNumeri $PBS_O_WORKDIR/SommaNumeri.c"
57 /usr/lib64/openmpi/1.4-gcc/bin/mpicc -lm -std=c99 -o $PBS_O_WORKDIR/SommaNumeri
       $PBS_O_WORKDIR/SommaNumeri.c
58
59
60 ############################
61 #                          #
62 #        Esecuzione        #
63 #                          #
64 ############################
65
66 echo -e "\n\n\n################### INI EXEC ###################\n"
67
```

```
/usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile hostlist -np $NCPU
    $PBS_O_WORKDIR/SommaNumeri --operands 1000 --strategy 1 --printer 3

echo "################## END EXEC ##################"
```

Proprio come nel caso dell'esecuzione dei test, il risultato ottenuto dall'esecuzione dello script PBS è contenuto nel file SommaNumeri.out mentre eventuali errori nel file SommaNumeri.err:

```
------------------------------
This Job is allocated on 8 cpu(s)
Job is running on node(s):
wn273.scope.unina.it
wn274.scope.unina.it
wn275.scope.unina.it
wn276.scope.unina.it
wn277.scope.unina.it
wn278.scope.unina.it
wn279.scope.unina.it
wn280.scope.unina.it
---------------------------------
PBS: qsub is running on ui-studenti.scope.unina.it
PBS: originating queue is studenti
PBS: executing queue is studenti
PBS: working directory is /homes/DMA/PDC/2021/RMNMRC98R/SommaNumeri
PBS: execution mode is PBS_BATCH
PBS: job identifier is 3988091.torque02.scope.unina.it
PBS: job name is SommaNumeri
PBS: node file is /var/spool/pbs/aux//3988091.torque02.scope.unina.it
PBS: current home directory is /homes/DMA/PDC/2021/RMNMRC98R
PBS: PATH = /usr/lib64/openmpi/1.2.7-gcc/bin:/usr/kerberos/bin:/opt/exp_soft/unina.
    it/intel/composer_xe_2013_sp1.3.174/bin/intel64:/opt/exp_soft/unina.it/intel/
    composer_xe_2013_sp1.3.174/mpirt/bin/intel64:/opt/exp_soft/unina.it/intel/
    composer_xe_2013_sp1.3.174/bin/intel64:/opt/exp_soft/unina.it/intel/
    composer_xe_2013_sp1.3.174/bin/intel64_mic:/opt/exp_soft/unina.it/intel/
    composer_xe_2013_sp1.3.174/debugger/gui/intel64:/opt/d-cache/srm/bin:/opt/d-
    cache/dcap/bin:/opt/edg/bin:/opt/glite/bin:/opt/globus/bin:/opt/lcg/bin:/usr/
    local/bin:/bin:/usr/bin:/opt/exp_soft/HADOOP/hadoop-1.0.3/bin:/opt/exp_soft/
    unina.it/intel/composerxe/bin/intel64/:/opt/exp_soft/unina.it/MPJExpress/mpj-
    v0_38/bin:/homes/DMA/PDC/2021/RMNMRC98R/bin
---------------------------------
Compilazione con /usr/lib64/openmpi/1.4-gcc/bin/mpicc -lm -std=c99 -o /homes/DMA/
    PDC/2021/RMNMRC98R/SommaNumeri/SommaNumeri /homes/DMA/PDC/2021/RMNMRC98R/
```

```
    SommaNumeri/SommaNumeri.c
25
26
27
28 ################### INI EXEC ###################
29
30  >> Maximum time detected: 0.0000762.
31
32  >> [P3] Result of the sum: 34753.45865.
33
34 ################### END EXEC ###################
```

8.2.1 Composizione argomenti

Il programma prende in ingresso i seguenti argomenti (necessariamente in ordine):

- **-o** o equivalentemente –**operands**

```
1 # Operandi totali: 100000
2
3 /usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile hostlist -np $NCPU
      $PBS_O_WORKDIR/SommaNumeri --operands 100000
4
5 # Oppure, in modo equivalente
6 /usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile hostlist -np $NCPU
      $PBS_O_WORKDIR/SommaNumeri -o 100000
```

- **-s** o equivalentemente –**strategy**

```
1 # Strategia da adottare: 2
2
3 /usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile hostlist -np $NCPU
      $PBS_O_WORKDIR/SommaNumeri --operands 100000 --strategy 2
4
5 # Oppure, in modo equivalente
6 /usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile hostlist -np $NCPU
      $PBS_O_WORKDIR/SommaNumeri --operands 100000 -s 2
```

- **-p** o equivalentemente –**printer**

```
1 # Identificativo del processore stampante: 3
2
```

```
3 /usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile hostlist -np $NCPU
      $PBS_O_WORKDIR/SommaNumeri --operands 100000 --strategy 2 --printer 3
4
5 # Oppure, in modo equivalente
6 /usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile hostlist -np $NCPU
      $PBS_O_WORKDIR/SommaNumeri --operands 100000 --strategy 2 -p 3
```

8.2.2 Help

Il programma offre anche la possibilità di stampare (sul file SommaNumeri.out) l'help, ossia le informazioni sugli argomenti necessari per avviare il programmare correttamente:

```
1 # Per stampare l'help e' necessario passare al programma esclusivamente l'argomento
      --help
2
3 echo -e "\n\n\n################### INI EXEC ###################\n"
4
5 /usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile hostlist -np $NCPU
    $PBS_O_WORKDIR/SommaNumeri --help
6
7 echo "################### END EXEC ###################"
```

Successivamente all'esecuzione dello script PBS riportato precedentemente, sul file SommaNumeri.out troveremo:

```
1 ################### INI EXEC ###################
2
3  >> Usage: /homes/DMA/PDC/2021/RMNMRC98R/SommaNumeri/SommaNumeri [-o --operands] <
    value> [<values...>] [-s --strategy] <value> [-p --printer] <value>
4
5   Mandatory arguments:
6      -o  --operands           Amount of operands (followed by the actual operands
    if less than 20)
7      -s  --strategy           Strategy to be applied in order to calculate the sum
    [1 2 3]
8      -p  --printer            ID of the process that will print the result (-1 ->
    all processes)
9
10  Optional arguments:
11         --help               Display this help and exit
12
```

```
  Error codes:
     101 ERR_ARGC              Invalid number of arguments
     102 ERR_NO_OPERANDS       Mandatory argument [-o --operands] not provided
     103 ERR_NO_STRATEGY       Mandatory argument [-s --strategy] not provided
     104 ERR_NO_PRINTER        Mandatory argument [-p --printer] not provided
     105 ERR_TOT_OPERANDS      Invalid amount of operands provided
     106 ERR_OPERAND           Invalid operand provided
     107 ERR_STRATEGY          Invalid strategy provided
     108 ERR_PRINTER           Invalid ID printer provided
     109 ERR_MEMORY            Unable to allocate memory

################## END EXEC ##################
```

8.2.3 Esempio di esecuzione con totale operandi > 20

```
echo -e "\n\n\n################## INI EXEC ##################\n"

/usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile hostlist -np $NCPU
    $PBS_O_WORKDIR/SommaNumeri --operands 100000 --strategy 3 --printer 0

echo "################## END EXEC ##################"
```

- Operandi totali: 100000;
- Strategia utilizzata: 3;
- Identificativo del processore stampate: 0.

Un possibile output del programma è il seguente:

```
################## INI EXEC ##################

 >> Maximum time detected: 0.000105.

 >> [P0] Result of the sum: 57753.458665.

################## END EXEC ##################
```

8.2.4 Esempio di esecuzione con totale operandi ≤ 20

```
echo -e "\n\n\n################### INI EXEC ##################\n"

/usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile hostlist -np $NCPU
    $PBS_O_WORKDIR/SommaNumeri --operands 10 1 2 3 4 5 6 7 8 9 10 --strategy 1 --
    printer 2

echo "################### END EXEC ##################"
```

- Operandi totali: 10 - $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$;
- Strategia utilizzata: 1;
- Identificativo del processore stampate: 2.

Un possibile output del programma è il seguente:

```
################### INI EXEC ##################

 >> Maximum time detected: 0.000156.

 >> [P2] Result of the sum: 55.000000.

################### END EXEC ##################
```

9 Codice Sorgente

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <ctype.h>
#include <limits.h>
#include <mpi.h>


#define SD_ARG_OPERANDS     "-o"
#define SD_ARG_STRATEGY     "-s"
#define SD_ARG_PRINTER      "-p"
```

```
13
14 #define DD_ARG_OPERANDS     "--operands"
15 #define DD_ARG_STRATEGY     "--strategy"
16 #define DD_ARG_PRINTER      "--printer"
17 #define DD_ARG_HELP         "--help"
18
19 #define MAX_OPERANDS_CMD    20
20
21 #define STRATEGY_1          1
22 #define STRATEGY_2          2
23 #define STRATEGY_3          3
24
25 #define SCC_ARGS            0
26 #define SCC_HELP            1
27
28 #define ERR_ARGC            101
29 #define ERR_NO_OPERANDS     102
30 #define ERR_NO_STRATEGY     103
31 #define ERR_NO_PRINTER      104
32 #define ERR_TOT_OPERANDS    105
33 #define ERR_OPERAND         106
34 #define ERR_STRATEGY        107
35 #define ERR_PRINTER         108
36 #define ERR_MEMORY          109
37
38 #define DIST_TAG            222
39 #define SUM_TAG             333
40
41 #define MISSING_ARG         "\n <!> ERROR: Expected [%s %s] argument! For
       additional info type %s.\n"
42 #define INVALID_ARG         "\n <!> ERROR: Invalid value for argument [%s %s]! For
       additional info type %s.\n"
43
44
45 void help(char*);
46 double* generate_random_operands(int, double, double);
47 double* generate_operands(char* []);
48 double sequential_sum(double*, int);
49 int check_args(int, char* [], int);
50 int* create_lookup_table_pow2(int);
51 void distribute_operands(int, int, int, double*);
52 void apply_strategy_1(int, int, int, double*);
53 void apply_strategy_2(int, int, int, int, int*, double*);
```

```
54 void apply_strategy_3(int, int, int*, double*);
55
56
57 int main(int argc, char* argv[]) {
58
59   /*
60     mpi_rank: Identificativo MPI del processore;
61     mpi_size: Numero di processori del communicator;
62
63     args_error: Codice errore ottenuto dalla verifica degli argomenti;
64
65     log2_mpi_size: Valore del logaritmo in base 2 di mpi_size;
66     lookup_table_pow2: Vettore contenente le potenze di 2 fino a 2^(log2_mpi_size
       +1);
67
68     total_operands: Numero totale di operandi da sommare (dim. problema);
69     total_subproblem_operands: Numero totale di operandi che ogni processore
70                                deve sommare (dim. sotto-problema);
71     operands: Vettore degli operandi;
72
73     strategy: Strategia da adottare;
74     printer: Identificativo del processore che deve stampare;
75
76     start_time: Tempo inizio somma;
77     end_time: Tempo fine somma;
78     delta_time: Differenza temporale tra inizio e fine somma;
79     max_time: Tempo di somma massimo;
80
81     sum: Somma calcolata.
82   */
83
84   int mpi_rank;
85   int mpi_size;
86
87   int args_error;
88
89   int log2_mpi_size;
90   int* lookup_table_pow2;
91
92   int total_operands;
93   int total_subproblem_operands;
94   double* operands;
95
```

```
96  int strategy;
97  int printer;
98
99  double start_time;
100 double end_time;
101 double delta_time;
102 double max_time;
103
104 double sum = 0;
105
106 MPI_Status status;
107
108 MPI_Init(&argc, &argv);
109 MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
110 MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);
111
112 // Controllo argomenti
113
114 if(!mpi_rank) {
115
116   args_error = check_args(argc, argv, mpi_size);
117
118   switch(args_error) {
119
120     case SCC_HELP:
121       help(argv[0]);
122       break;
123
124     case ERR_ARGC:
125       printf("\n <!> ERROR: Invalid number of arguments! For additional info type
      %s.\n", DD_ARG_HELP);
126       break;
127
128     case ERR_NO_OPERANDS:
129       printf(MISSING_ARG, SD_ARG_OPERANDS, DD_ARG_OPERANDS, DD_ARG_HELP);
130       break;
131
132     case ERR_NO_STRATEGY:
133       printf(MISSING_ARG, SD_ARG_STRATEGY, DD_ARG_STRATEGY, DD_ARG_HELP);
134       break;
135
136     case ERR_NO_PRINTER:
137       printf(MISSING_ARG, SD_ARG_PRINTER, DD_ARG_PRINTER, DD_ARG_HELP);
```

```
138         break;
139
140       case ERR_TOT_OPERANDS:
141       case ERR_OPERAND:
142         printf(INVALID_ARG, SD_ARG_OPERANDS, DD_ARG_OPERANDS, DD_ARG_HELP);
143         break;
144
145       case ERR_STRATEGY:
146         printf(INVALID_ARG, SD_ARG_STRATEGY, DD_ARG_STRATEGY, DD_ARG_HELP);
147         break;
148
149       case ERR_PRINTER:
150         printf(INVALID_ARG, SD_ARG_PRINTER, DD_ARG_PRINTER, DD_ARG_HELP);
151         break;
152
153     }
154
155     if(args_error != SCC_ARGS && args_error != SCC_HELP)
156       MPI_Abort(MPI_COMM_WORLD, args_error);
157
158   }
159
160   // Propagazione codice SCC_HELP
161
162   if(mpi_size != 1)
163     MPI_Bcast(&args_error, 1, MPI_INT, 0, MPI_COMM_WORLD);
164   if(args_error == SCC_HELP) {
165     MPI_Finalize();
166     return 0;
167   }
168
169   // Lettura argomenti
170
171   if (!mpi_rank) {
172
173     operands = generate_operands(argv);
174
175     total_operands = atoi(argv[2]);
176     strategy = (total_operands <= MAX_OPERANDS_CMD) ? atoi(argv[total_operands +
      4]) : atoi(argv[4]);
177     printer = (total_operands <= MAX_OPERANDS_CMD) ? atoi(argv[total_operands + 6])
       : atoi(argv[6]);
178
```

```
179     if(strategy != STRATEGY_1) {
180       if(total_operands < mpi_size) {
181         strategy = STRATEGY_1;
182         if(!mpi_rank)
183           printf("\n <!> WARNING: Strategy forced to %d -> not enough operands.",
    STRATEGY_1);
184       } else if((mpi_size & (mpi_size - 1))) {
185           strategy = STRATEGY_1;
186           if(!mpi_rank)
187           printf(
188             "\n <!> WARNING: Strategy forced to %d -> number of processors must be
    power of two, current [%d].",
189             STRATEGY_1, mpi_size
190           );
191       }
192     }
193
194   }
195
196   // Distribuzione operandi
197
198   MPI_Bcast(&total_operands, 1, MPI_INT, 0, MPI_COMM_WORLD);
199   MPI_Bcast(&strategy, 1, MPI_INT, 0, MPI_COMM_WORLD);
200   MPI_Bcast(&printer, 1, MPI_INT, 0, MPI_COMM_WORLD);
201
202     // Calcolo della somma
203
204     if (mpi_size == 1) {
205
206       // Somma sequenziale
207
208       printf("\n <!> WARNING: Single processor detected, sequential sum will be
    performed!\n");
209       start_time = MPI_Wtime();
210     sum = sequential_sum(operands, total_operands);
211     end_time = MPI_Wtime();
212     max_time = end_time - start_time;
213
214   } else {
215
216     // Calcolo log2_mpi_size e creazione lookup table delle potenze di 2 fino a
    log2_mpi_size
217
```

```
218     log2_mpi_size = (int) log2f(mpi_size);
219     lookup_table_pow2 = create_lookup_table_pow2(log2_mpi_size+1);
220     if(!lookup_table_pow2)
221       MPI_Abort(MPI_COMM_WORLD, ERR_MEMORY);
222
223     // Calcolo dimensione sotto-problema
224
225     total_subproblem_operands = total_operands / mpi_size;
226     total_subproblem_operands += ((total_operands % mpi_size) > mpi_rank) ? 1 : 0;
227
228     // Distribuzione operandi
229
230     if(!mpi_rank)
231       distribute_operands(total_operands, total_subproblem_operands, mpi_size,
    operands);
232
233     else {
234
235       operands = (double*) calloc(total_subproblem_operands, sizeof(double));
236       if(!operands)
237         MPI_Abort(MPI_COMM_WORLD, ERR_MEMORY);
238       MPI_Recv(operands, total_subproblem_operands, MPI_DOUBLE, 0, DIST_TAG +
    mpi_rank, MPI_COMM_WORLD, &status);
239
240     }
241
242
243     // Sincronizzazione e inizializzazione time start
244
245     MPI_Barrier(MPI_COMM_WORLD);
246       start_time = MPI_Wtime();
247
248       // Calcolo sotto - problema
249
250       sum = sequential_sum(operands, total_subproblem_operands);
251
252     // Applicazione delle strategie
253
254     switch(strategy) {
255
256       case STRATEGY_1:
257         apply_strategy_1(mpi_rank, mpi_size, printer, &sum);
258         break;
```

```
259
260       case STRATEGY_2:
261         apply_strategy_2(mpi_rank, mpi_size, printer, log2_mpi_size,
      lookup_table_pow2, &sum);
262         break;
263
264       case STRATEGY_3:
265         apply_strategy_3(mpi_rank, log2_mpi_size, lookup_table_pow2, &sum);
266         break;
267     }
268
269     // Calcolo tempo
270
271     end_time = MPI_Wtime();
272     delta_time = end_time - start_time;
273     MPI_Reduce(&delta_time, &max_time, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
274
275   }
276
277   // Stampa del risultato
278
279   if(mpi_rank == printer)
280     printf("\n >> [P%d] Result of the sum: %lf.\n", mpi_rank, sum);
281   else if(printer == -1) {
282     if(strategy == STRATEGY_3) {
283       printf("\n >> [P%d] Result of the sum: %lf.\n", mpi_rank, sum);
284     } else {
285       MPI_Bcast(&sum, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
286       printf("\n >> [P%d] Result of the sum: %lf.\n", mpi_rank, sum);
287     }
288   }
289
290   // Stampa del tempo
291
292   if(!mpi_rank)
293     printf("\n >> Maximum time detected: %f.\n\n", max_time);
294
295   MPI_Finalize();
296   return 0;
297
298 }
299
300
```

```
301
302 /*
303
304   Stampa a video l'help del programma.
305
306   @params:
307     char* program_name: Nome del programma
308
309   @return:
310     void
311
312 */
313
314 void help(char* program_name) {
315
316
317   printf(
318     "\n >> Usage: %s [%s %s] <value> [<values...>] [%s %s] <value> [%s %s] <value>"
      ,
319     program_name,
320     SD_ARG_OPERANDS, DD_ARG_OPERANDS,
321     SD_ARG_STRATEGY, DD_ARG_STRATEGY,
322     SD_ARG_PRINTER, DD_ARG_PRINTER
323   );
324
325   printf("\n\n\tMandatory arguments:");
326   printf("\n\t   %s  %-20s Amount of operands (followed by the actual operands if
      less than 20)", SD_ARG_OPERANDS, DD_ARG_OPERANDS);
327   printf(
328     "\n\t   %s  %-20s Strategy to be applied in order to calculate the sum [%d %d %
      d]",
329     SD_ARG_STRATEGY, DD_ARG_STRATEGY,
330     STRATEGY_1, STRATEGY_2, STRATEGY_3
331   );
332   printf(
333     "\n\t   %s  %-20s ID of the process that will print the result (-1 -> all
      processes)",
334     SD_ARG_PRINTER, DD_ARG_PRINTER
335   );
336
337
338   printf("\n\n\tOptional arguments:");
339   printf("\n\t       %-20s Display this help and exit", DD_ARG_HELP);
```

```
340
341   printf("\n\n\tError codes:");
342   printf("\n\t   %d %-20s Invalid number of arguments", ERR_ARGC, "ERR_ARGC");
343   printf(
344     "\n\t   %d %-20s Mandatory argument [%s %s] not provided",
345     ERR_NO_OPERANDS, "ERR_NO_OPERANDS",
346     SD_ARG_OPERANDS, DD_ARG_OPERANDS
347   );
348   printf(
349     "\n\t   %d %-20s Mandatory argument [%s %s] not provided",
350     ERR_NO_STRATEGY, "ERR_NO_STRATEGY",
351     SD_ARG_STRATEGY, DD_ARG_STRATEGY
352   );
353   printf(
354     "\n\t   %d %-20s Mandatory argument [%s %s] not provided",
355     ERR_NO_PRINTER, "ERR_NO_PRINTER",
356     SD_ARG_PRINTER, DD_ARG_PRINTER
357   );
358   printf("\n\t   %d %-20s Invalid amount of operands provided", ERR_TOT_OPERANDS, "
       ERR_TOT_OPERANDS");
359   printf("\n\t   %d %-20s Invalid operand provided", ERR_OPERAND, "ERR_OPERAND");
360   printf("\n\t   %d %-20s Invalid strategy provided", ERR_STRATEGY, "ERR_STRATEGY")
       ;
361   printf("\n\t   %d %-20s Invalid ID printer provided", ERR_PRINTER, "ERR_PRINTER")
       ;
362   printf("\n\t   %d %-20s Unable to allocate memory", ERR_MEMORY, "ERR_MEMORY");
363
364 }
365
366
367
368 /*
369
370   Genera operandi pseudo-randomici.
371
372   @params:
373     int amount: Numero di operandi pseudo-randomici da generare
374     double lower: Limite inferiore del valore degli operandi
375     double upper: Limite superiore del valore degli operandi
376
377   @return:
378     double*: Array dinamico contenente gli operandi generati
379
```

```
380 */
381
382 double* generate_random_operands(int amount, double lower, double upper) {
383
384   double* operands = (double*) calloc(amount, sizeof(double));
385
386     if(operands) {
387     for(int i = 0; i < amount; i++) {
388       operands[i] = ((double) rand() * (upper - lower)) / (double) RAND_MAX + lower
      ;;
389     }
390   }
391
392   return operands;
393
394 }
395
396
397
398 /*
399
400   Effettua la somma degli operandi contenuti nell'array in input.
401
402   @params:
403     double* operands: Array contente gli operandi da sommare
404     int amount: Numero di operandi da sommare
405
406   @return:
407     double: Risultato della somma
408
409 */
410
411 double sequential_sum(double* operands, int amount) {
412
413   double sum = 0;
414
415   for(int i = 0; i < amount; i++)
416     sum += operands[i];
417
418   return sum;
419 }
420
421
```

```
422
423 /*
424
425   Genera gli operandi in relazione agli argomenti passati in ingresso.
426   Nello specifico, genera operandi random se il numero di operandi da
427   generare e' maggiore di 20, altrimenti legge da riga di comando.
428
429   @params:
430     char* argv[]: Argomenti passati in ingresso al programma
431
432   @return:
433     double*: Array dinamico contenente gli operandi generati
434
435 */
436
437 double* generate_operands(char* argv[]) {
438
439   double* operands;
440   int total_operands = atoi(argv[2]);
441
442   if(total_operands <= MAX_OPERANDS_CMD) {
443       operands = (double*) calloc(total_operands, sizeof(double));
444       if(operands) {
445         for(int i = 0; i < total_operands; i++) {
446           operands[i] = atof(argv[i+3]);
447       }
448     }
449   } else {
450     srand(MPI_Wtime());
451     operands = generate_random_operands(total_operands, INT_MIN, INT_MAX);
452   }
453
454   return operands;
455
456 }
457
458
459
460 /*
461
462   Verifica l'integrita' degli argomenti passati in ingresso al programma.
463
464   @params:
```

```
465     int argc: Numero di argomenti passati in ingresso al programma
466     char* argv[]: Argomenti passati in ingresso al programma
467     int mpi_size: Numero di processori utilizzati
468
469   @return:
470     int: Risultato della verifica (codice compreso tra 101 e 109)
471
472 */
473
474 int check_args(int argc, char* argv[], int mpi_size) {
475
476   if(argc == 2 && !strcmp(argv[1], DD_ARG_HELP))
477     return SCC_HELP;
478
479   if(argc >= 7) {
480
481     if(strcmp(argv[1], SD_ARG_OPERANDS) && strcmp(argv[1], DD_ARG_OPERANDS))
482       return ERR_NO_OPERANDS;
483
484     int total_operands = atoi(argv[2]);
485     if(total_operands <= 0)
486       return ERR_TOT_OPERANDS;
487
488     if((total_operands <= MAX_OPERANDS_CMD && total_operands+7 != argc) || (
      total_operands > MAX_OPERANDS_CMD && argc != 7))
489       return ERR_ARGC;
490
491     if(total_operands <= 20) {
492       double operand;
493       for(int i = 0; i < total_operands; i++) {
494         if(!sscanf(argv[i+3], "%lf", &operand))
495           return ERR_OPERAND;
496       }
497     }
498
499     int succ_arg_pos = (total_operands <= MAX_OPERANDS_CMD) ? total_operands + 3 :
      3;
500
501     if(strcmp(argv[succ_arg_pos], SD_ARG_STRATEGY) && strcmp(argv[succ_arg_pos],
      DD_ARG_STRATEGY))
502       return ERR_NO_STRATEGY;
503
504     int strategy = atoi(argv[succ_arg_pos+1]);
```

```
505     if((strategy== 0 && argv[succ_arg_pos+1][0] != '0') || (strategy < STRATEGY_1
        || strategy > STRATEGY_3))
506       return ERR_STRATEGY;
507
508     if(strcmp(argv[succ_arg_pos+2], SD_ARG_PRINTER) && strcmp(argv[succ_arg_pos+2],
         DD_ARG_PRINTER))
509       return ERR_NO_PRINTER;
510
511     int printer = atoi(argv[succ_arg_pos+3]);
512     if((printer == 0 && argv[succ_arg_pos+3][0] != '0') || (printer < -1 || printer
         >= mpi_size))
513       return ERR_PRINTER;
514
515     return SCC_ARGS;
516
517   }
518
519   return ERR_ARGC;
520
521 }
522
523
524
525 /*
526
527   Crea un array delle potenze di due.
528
529   @params:
530     int size: Grandezza della tabella
531
532   @return:
533     int: Array dinamico contenente le potenze di due fino alla 2^size-esima
534
535 */
536
537 int* create_lookup_table_pow2(int size) {
538
539   int* lookup_table_pow2 = (int*) calloc(size, sizeof(int));
540
541   if(lookup_table_pow2) {
542     lookup_table_pow2[0] = 1;
543     for(int i = 1; i < size; i++) {
544       lookup_table_pow2[i] = lookup_table_pow2[i-1] << 1;
```

```
545     }
546   }
547
548   return lookup_table_pow2;
549
550 }
551
552
553
554 /*
555
556   Distribuisce gli operandi dal processore P0 ai restanti.
557   Questa funzione deve essere necessariamente richiamata dal processore P0.
558
559   @params:
560     int total_operands: Numero totale degli operandi
561     int total_subproblem_operands: Numero degli operandi da distribuire
562     int mpi_size: Numero di processori utilizzati
563     double* operands: Array contenente gli operandi da distribuire
564
565   @return:
566     void
567
568 */
569
570 void distribute_operands(int total_operands, int total_subproblem_operands, int
      mpi_size, double* operands) {
571
572   int initial_operand_index = total_subproblem_operands;
573
574   for(int processor = 1; processor < mpi_size; processor++) {
575     total_subproblem_operands -= ((total_operands % mpi_size) == processor) ? 1 :
      0;
576     MPI_Send(
577       &operands[initial_operand_index],
578       total_subproblem_operands,
579       MPI_DOUBLE,
580       processor,
581       DIST_TAG + processor,
582       MPI_COMM_WORLD
583     );
584     initial_operand_index += total_subproblem_operands;
585   }
```

```
586
587 }
588
589
590
591 /*
592
593   Esegue la strategia per il calcolo della somma parallela con strategia I.
594
595   @params:
596     int mpi_rank: ID del processore chiamante
597     int mpi_size: Numero di processori utilizzati
598     int printer: ID del processore che deve stampare il risultato
599     double* sum: Riferimento alla variabile utilizzata per salvare la somma
600
601   @return:
602     void
603
604 */
605
606 void apply_strategy_1(int mpi_rank, int mpi_size, int printer, double* sum) {
607
608   MPI_Status status;
609   double partial_sum;
610   printer = (printer == -1) ? 0 : printer;
611
612   if(mpi_rank == printer) {
613     for(int processor = 0; processor < mpi_size; processor++) {
614       if(processor != printer) {
615         MPI_Recv(&partial_sum, 1, MPI_DOUBLE, processor, SUM_TAG + processor,
    MPI_COMM_WORLD, &status);
616         *sum += partial_sum;
617       }
618     }
619   } else
620     MPI_Send(sum, 1, MPI_DOUBLE, printer, SUM_TAG + mpi_rank, MPI_COMM_WORLD);
621
622 }
623
624
625
626 /*
627
```

```
628   Esegue la strategia per il calcolo della somma parallela con strategia II.
629
630   @params:
631     int mpi_rank: ID del processore chiamante
632     int mpi_size: Numero di processori utilizzati
633     int printer: ID del processore che deve stampare il risultato
634     int log2_mpi_size: Valore del logaritmo in base 2 di mpi_size (passi di
      comunicazione)
635     int* lookup_table_pow2: Array contenente le potenze di due
636     double* sum: Riferimento alla variabile utilizzata per salvare la somma
637
638   @return:
639     void
640
641 */
642
643 void apply_strategy_2(int mpi_rank, int mpi_size, int printer, int log2_mpi_size,
      int* lookup_table_pow2, double* sum) {
644
645   MPI_Status status;
646   double partial_sum;
647   printer = (printer == -1) ? 0 : printer;
648   int comm_processor;
649   int alt_mpi_rank = (mpi_rank + (mpi_size - printer)) % mpi_size;
650
651   for(int comm_step = 0; comm_step < log2_mpi_size; comm_step++) {
652     if((alt_mpi_rank % lookup_table_pow2[comm_step]) == 0) {
653       if((alt_mpi_rank % lookup_table_pow2[comm_step+1]) == 0) {
654         comm_processor = mpi_rank + lookup_table_pow2[comm_step];
655         comm_processor = (comm_processor >= mpi_size) ? (comm_processor % mpi_size)
       : comm_processor;
656         MPI_Recv(&partial_sum, 1, MPI_DOUBLE, comm_processor, SUM_TAG + mpi_rank,
      MPI_COMM_WORLD, &status);
657         *sum += partial_sum;
658       } else {
659         comm_processor = mpi_rank - lookup_table_pow2[comm_step];
660         comm_processor = (comm_processor < 0) ? (comm_processor + mpi_size) :
      comm_processor;
661         MPI_Send(sum, 1, MPI_DOUBLE, comm_processor, SUM_TAG + comm_processor,
      MPI_COMM_WORLD);
662       }
663     }
664   }
```

```
665
666 }
667
668
669 /*
670
671   Esegue la strategia per il calcolo della somma parallela con strategia III.
672
673   @params:
674     int mpi_rank: ID del processore chiamante
675     int log2_mpi_size: Valore del logaritmo in base 2 di mpi_size (passi di
      comunicazione)
676     int* lookup_table_pow2: Array contenente le potenze di due
677     double* sum: Riferimento alla variabile utilizzata per salvare la somma
678
679   @return:
680     void
681
682 */
683
684 void apply_strategy_3(int mpi_rank, int log2_mpi_size, int* lookup_table_pow2,
      double* sum) {
685
686   MPI_Status status;
687   double partial_sum;
688
689   for(int comm_step = 0; comm_step < log2_mpi_size; comm_step++) {
690     if((mpi_rank % lookup_table_pow2[comm_step+1]) < lookup_table_pow2[comm_step])
      {
691       int comm_processor = mpi_rank + lookup_table_pow2[comm_step];
692       MPI_Recv(&partial_sum, 1, MPI_DOUBLE, comm_processor, SUM_TAG + mpi_rank,
      MPI_COMM_WORLD, &status);
693       MPI_Send(sum, 1, MPI_DOUBLE, comm_processor, SUM_TAG + comm_processor,
      MPI_COMM_WORLD);
694     } else {
695       int comm_processor = mpi_rank - lookup_table_pow2[comm_step];
696       MPI_Send(sum, 1, MPI_DOUBLE, comm_processor, SUM_TAG + comm_processor,
      MPI_COMM_WORLD);
697       MPI_Recv(&partial_sum, 1, MPI_DOUBLE, comm_processor, SUM_TAG + mpi_rank,
      MPI_COMM_WORLD, &status);
698     }
699     *sum += partial_sum;
700   }
```

```
701
702 }
```