

Università degli Studi di Napoli Federico II

Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione

# Corso di Laurea Magistrale in Informatica

## Corso di Parallel and Distributed Computing

*Prof. Giuliano Laccetti – Prof.<sup>ssa</sup> Valeria Mele*

# Sviluppo di un algoritmo per il calcolo della somma di $n$ numeri reali in ambiente di calcolo parallelo

## CANDIDATI:

CAROFANO Mario Gabriele - N97000437

NOVIELLO Francesco - N97000436

Anno Accademico 2023-2024

# Indice

<b>1</b>	<b>Definizione e analisi del problema</b>	<b>1</b>
1.1	Strategia 1 . . . . .	2
1.2	Strategia 2 . . . . .	3
1.3	Strategia 3 . . . . .	4
<b>2</b>	<b>Errori, costanti e librerie</b>	<b>5</b>
2.1	Errori . . . . .	5
2.2	Costanti . . . . .	8
2.3	Librerie . . . . .	13
2.3.1	Libreria mpi.h . . . . .	13
2.3.2	Libreria auxfunc.h . . . . .	17
2.3.3	Libreria menufunc.h . . . . .	18
2.3.4	Altre librerie . . . . .	19
<b>3</b>	<b>Creazione del file di esecuzione .pbs</b>	<b>23</b>
3.1	Inizializzazione dell'ambiente di lavoro . . . . .	24
3.2	Scelta della strategia da applicare . . . . .	25
3.3	Scelta del numero di operandi da sommare . . . . .	26
3.4	Applicazione della suite di testing . . . . .	27
3.5	Scrittura del file .pbs . . . . .	28
<b>4</b>	<b>Implementazione dell'algoritmo</b>	<b>29</b>
4.1	Inizializzazione dell'ambiente di lavoro . . . . .	30
4.2	Inizializzazione dell'ambiente MPI . . . . .	34
4.3	Lettura dei dati . . . . .	34
4.4	Distribuzione degli operandi . . . . .	36
4.4.1	Calcolo della quantità locale di operandi . . . . .	36
4.4.2	Allocazione del vettore degli operandi . . . . .	36
4.4.3	Lettura e/o generazione del valore degli operandi . . . . .	37

4.4.4	Assegnazione dei valori agli operandi	39
4.5	Applicazione della strategia	40
4.6	Calcolo dei tempi di esecuzione	40
4.7	Stampa dell'output e terminazione	41
<b>5</b>	<b>Analisi dei tempi e delle prestazioni</b>	<b>43</b>
5.1	Analisi con $10^4$ operandi	45
5.2	Analisi con $10^5$ operandi	47
5.3	Analisi con $10^6$ operandi	49
<b>6</b>	<b>Conclusioni</b>	<b>53</b>
6.1	Futuri sviluppi	54
<b>A</b>	<b>Elenco dei listati</b>	<b>55</b>
A.1	auxfunc.c	55
A.2	menufunc.c	58
A.3	menu.c	65
A.4	proval.c	68
<b>Sitografia</b>		<b>81</b>

# Capitolo 1

## Definizione e analisi del problema

L'algoritmo che viene presentato in questa relazione ha l'obiettivo di effettuare il calcolo della somma di  $N$  numeri reali, in ambiente di calcolo parallelo con  $p$  processori su architettura MIMD a memoria distribuita.

In particolare, l'algoritmo è stato implementato in C e fa uso della libreria *Message Passing Interface (MPI)* per la gestione dei processori dell'infrastruttura di calcolo parallelo di tipo *Multiple Instruction, Multiple Data (MIMD)* a memoria distribuita che è stata utilizzata.

Nelle prossime sezioni 1.1 a pagina 2, 1.2 a pagina 3 e 1.3 a pagina 4 sono illustrate le diverse strategie adoperate per il calcolo della somma su architettura parallela. Ogni strategia ha l'obiettivo di ottenere la somma su  $p$  processori.

## 1.1 Strategia 1

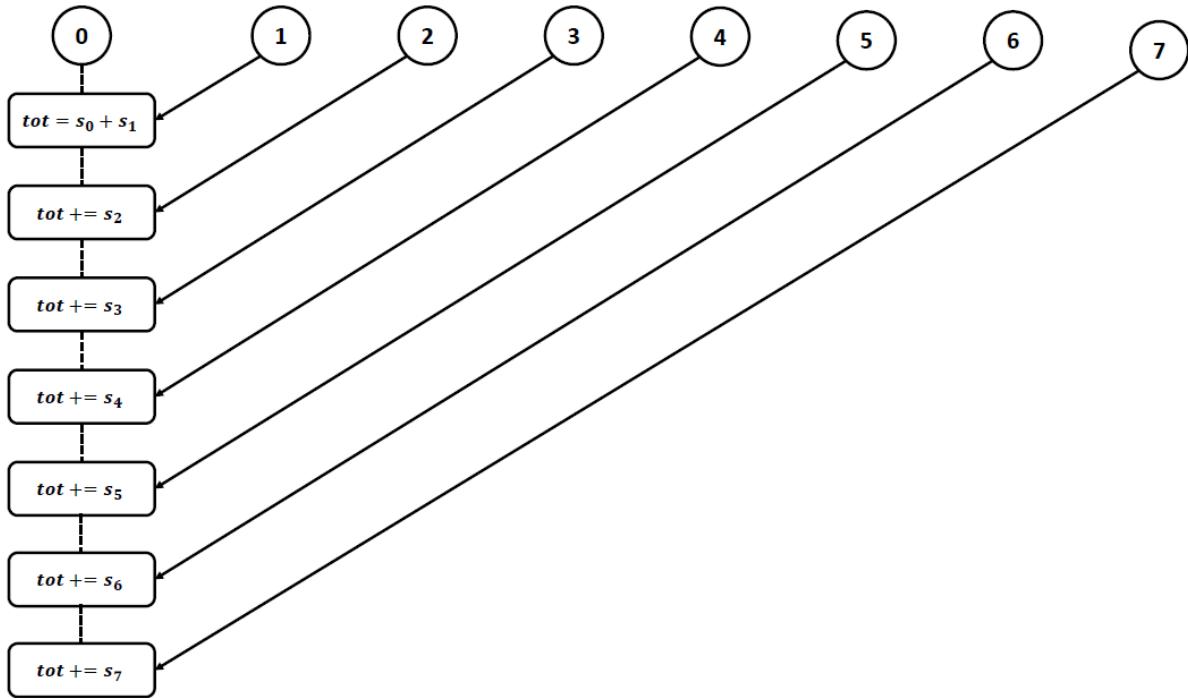


Figura 1.1: Strategia 1 con  $p = 8$ .

Tutti i processori eseguono le seguenti operazioni:

1. Sommare localmente gli addendi ricevuti dalla lettura e distribuzione degli operandi eseguita dal solo processore  $p_0$ .
2. Inviare il risultato della somma parziale al processore  $p_0$ , in modo tale che possa sommarla alla propria somma parziale.

L'algoritmo termina quando il processore  $p_0$  ha ricevuto tutte le somme parziali e siamo in presenza, quindi, della somma totale che sarà opportunamente visualizzata in output.

Si può concludere che si hanno tanti passi di comunicazione quanti sono i processori coinvolti nel calcolo della somma.

## 1.2 Strategia 2

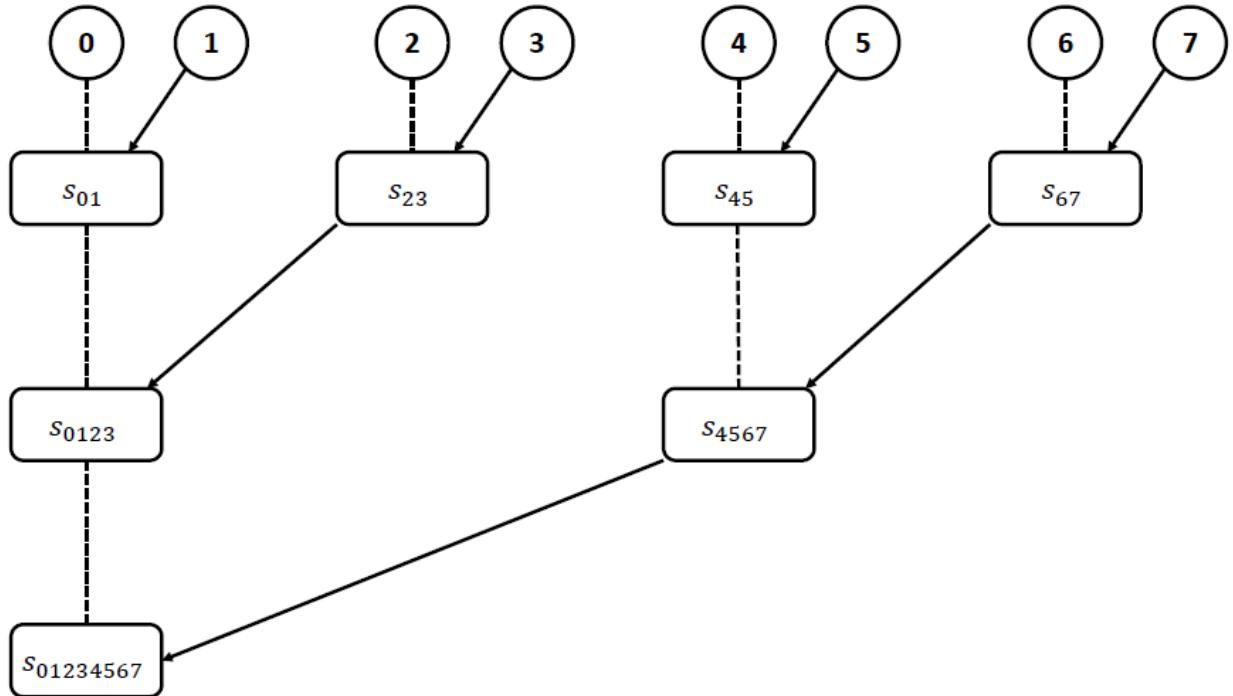


Figura 1.2: Strategia 2 con  $p = 8$ .

Con la strategia 2 si ottiene uno schema ad albero, mostrato nella figura 1.2, che ha il processore  $p_0$  come radice e in cui ogni livello rappresenta un diverso passo di comunicazione. Cioè, si hanno  $\log_2(\text{numero processori})$  passi di comunicazione.

In particolare, si può notare che se il numero di processori non restituisce un valore intero per il calcolo del  $\log_2$ , allora la strategia 2 non è applicabile.

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.

La somma totale, infine, è comunque calcolata e visualizzata in output da un singolo processore, in maniera analoga alla strategia 1.

### 1.3 Strategia 3

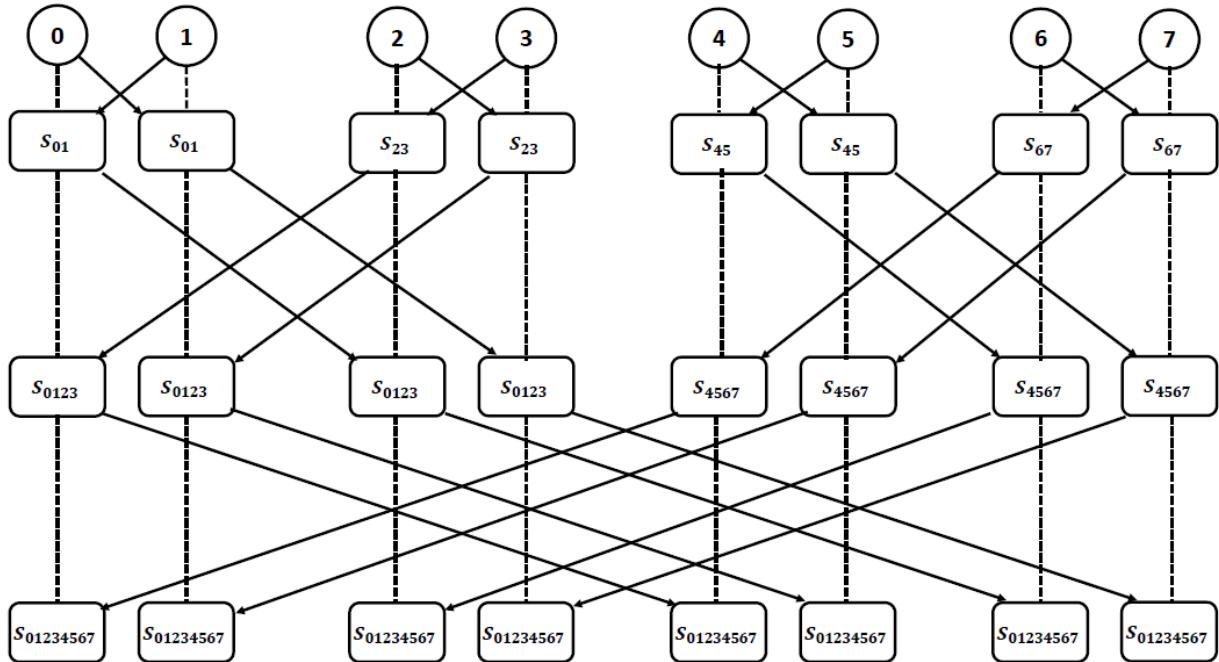


Figura 1.3: Strategia 3 con  $p = 8$ .

Nella strategia 3, proprio come accade nella strategia 2, si hanno  $\log_2(\text{numero processori})$  passi di comunicazione. La principale differenza, però, è che ad ogni passo di comunicazione, tutti i processori partecipano allo scambio delle somme parziali.

All’ultimo step di comunicazione, differentemente da quanto visto nella strategia 1 e nella strategia 2, tutti i processori ottengono la somma totale e, quindi, tutti i processori possono visualizzarla in output.

# Capitolo 2

## Errori, costanti e librerie

Si inizia la trattazione dell'algoritmo implementato introducendo alcuni file aggiuntivi realizzati dal team di sviluppo contenenti costanti, codici di errore e funzioni accessorie.

### 2.1 Errori

Nel file *constants.c* sono definiti i codici di errori, utili a rappresentare le cause di terminazione del programma.

In particolare, per la loro definizione si utilizza la direttiva del preprocessore `#define` (principalmente, per una miglior leggibilità del codice). In questo modo, prima dell'inizio della compilazione del programma, il preprocessore sostituisce nel codice tali definizioni con i valori ad esse associate.

- `#define NOT_ENOUGH_ARGS_ERROR 1`

Si utilizza per segnalare all'utente che l'esecuzione del programma è stata terminata per mancanza di un numero sufficiente di argomenti.

- ***#define EMPTY\_ARG\_ERROR 2***

Si utilizza per segnalare all’utente che l’esecuzione del programma è stata terminata siccome un argomento dato in input è vuoto.

---

- ***#define INPUT\_ARG\_ERROR 3***

Si utilizza per segnalare all’utente che l’esecuzione del programma è stata terminata a causa di un’errata lettura di un argomento dato in input.

---

- ***#define NOT\_INT\_ARG\_ERROR 4***

Si utilizza per segnalare all’utente che l’esecuzione del programma è stata terminata poichè un argomento dato in input non è rappresentabile come intero.

---

- ***#define INPUT\_LINE\_ERROR 5***

Si utilizza per segnalare all’utente che l’esecuzione del programma è stata terminata a causa di un’errata lettura dello ’standard input stream’ (stdin).

---

- ***#define NOT\_REAL\_NUMBER\_ERROR 6***

Si utilizza per segnalare all’utente che l’esecuzione del programma è stata terminata siccome non ha inserito un numero reale correttamente.

---

- ***#define NOT\_NATURAL\_NUMBER\_ERROR 7***

Si utilizza per segnalare all’utente che l’esecuzione del programma è stata terminata siccome non ha inserito un numero naturale correttamente.

- ***#define NOT\_IN\_RANGE\_ERROR 8***

Si utilizza per segnalare all'utente che l'esecuzione del programma è stata terminata siccome non ha inserito un numero compreso in un certo intervallo.

---

- ***#define NOT\_ENOUGH\_OPERANDS 9***

Si utilizza per segnalare all'utente che l'esecuzione del programma è stata terminata siccome non ha inserito un numero sufficiente di operandi per la somma.

---

- ***#define FILE\_OPENING\_ERROR 10***

Si utilizza per segnalare all'utente che l'esecuzione del programma è stata terminata a causa di un errore durante l'apertura di un determinato file.

## 2.2 Costanti

Nel file *constants.c* sono definite anche le costanti, utili per la definizione di alcuni parametri nel codice. La loro definizione è identica a quanto spiegato nella sezione 2.1 a pagina 5.

- `#define NO_STRATEGY 0`

Si utilizza per applicare l'algoritmo del calcolo della somma in sequenziale quando si esegue un caso di test su un solo processore.

---

- `#define FIRST_STRATEGY 1`
- `#define SECOND_STRATEGY 2`
- `#define THIRD_STRATEGY 3`
- `#define TESTING_SUITE 4`
- `#define EXIT_APPLICATION 5`

Queste 5 costanti sono valori interi che rappresentano le possibili scelte che può inserire l'utente nel menù principale dell'applicazione.

---

- `#define NO_TEST 0`

Si utilizza per eseguire l'algoritmo del calcolo della somma utilizzando i valori degli operandi scelti dall'utente oppure generati in modo casuale.

- `#define SUM_ONE_TEST 1`
- `#define SUM_SINGLE_NUMBER_TEST 2`
- `#define SUM_OPPOSITE_NUMBER_TEST 3`
- `#define GAUSS_TEST 4`
- `#define EXIT_TEST 5`

Queste 5 costanti sono valori interi che rappresentano le possibili scelte che può inserire l'utente nel menù della suite di test dell'applicazione.

---

- `#define OP_MIN_EXP_TEST 4`

Si utilizza come esponente minimo per determinare la quantita' di operandi nella creazione dei casi di test.

---

- `#define OP_MAX_EXP_TEST 6`

Si utilizza come esponente massimo per determinare la quantita' di operandi nella creazione dei casi di test.

---

- `#define OP_MAX_QUANTITY 20`

Come richiesto dalle specifiche dell'algoritmo, il massimo numero di operandi che l'utente può inserire manualmente è 20.

- **`#define NO_TIME_CALC 0`**

- **`#define OK_TIME_CALC 1`**

Si utilizza per personalizzare il flusso di esecuzione del programma in quei controlli dove si sceglie se effettuare (OK\_TIME\_CALC) o meno (NO\_TIME\_CALC) il calcolo dei tempi di esecuzione.

---

- **`#define NOME_PROVA "prova1"`**

Si utilizza nella funzione createPBS() per personalizzare il file .pbs con il nome della prova corrente.

---

- **`#define NODE_NUMBER "8"`**

Si utilizza nella funzione createPBS() per personalizzare il file .pbs con il numero di nodi con i quali si vuole eseguire il programma.

---

- **`#define NODE_PROCESS "8"`**

Si utilizza nella funzione createPBS() per personalizzare il file .pbs con il numero di processori occupati da ogni nodo per l'esecuzione del programma.

---

- **`#define MKDIR_PATH "/bin/mkdir"`**

Si utilizza per indicare alla funzione system() il percorso sul cluster dell'eseguibile 'mkdir'.

---

- **`#define RM_PATH "/bin/rm"`**

Si utilizza per indicare alla funzione system() il percorso sul cluster dell'eseguibile 'rm'.

- `#define CSV_TIME_PATH "../output"`

È il path del file .csv nella quale sono memorizzate tutte le informazioni necessarie allo studio dei casi di test. In particolare, la sua struttura è la seguente:

1. *test* : *int* (indica il caso di test eseguito)
  2. *strategia* : *int* (indica la strategia scelta per l'esecuzione)
  3. *n\_proc* : *int* (indica il numero di processori)
  4. *q\_num* : *int* (indica il numero di operandi sommati)
  5. *t\_tot* : *double* (indica il tempo di esecuzione impiegato)
- 

- `#define OP_MAX_VALUE 100`

Si utilizza come limite massimo del valore di un operando quando si sceglie di sommare numeri reali casuali.

---

- `#define DISTRIBUTION_TAG 25`

Si utilizza per assegnare un identificativo unico alle comunicazioni di MPI durante la fase di distribuzione dei dati.

---

- `#define FIRST_STRATEGY_TAG 100`

Si utilizza per assegnare un identificativo unico alle comunicazioni di MPI durante l'applicazione della 1a strategia.

---

- `#define SECOND_STRATEGY_TAG 200`

Si utilizza per assegnare un identificativo unico alle comunicazioni di MPI durante l'applicazione della 2a strategia.

- `#define THIRD_STRATEGY_TAG 300`

Si utilizza per assegnare un identificativo unico alle comunicazioni di MPI durante l'applicazione della 3a strategia.

## 2.3 Librerie

In questa sezione sono elencate e descritte le funzioni adoperate dalla libreria *mpi.h* [1], dalle librerie sviluppate dal team di sviluppo e dalle altre librerie disponibili per il linguaggio C.

### 2.3.1 Libreria mpi.h



Figura 2.1: Il logo di "Open MPI Project"

Il progetto "Open MPI" è un'implementazione open source delle specifiche Message Passing Interface (MPI), sviluppata e mantenuta da un consorzio di partner accademici, di ricerca e industriali. Open MPI combina le competenze, le tecnologie e le risorse di tutta la comunità dell'High Performance Computing per costruire la migliore libreria MPI disponibile.

In particolare, le funzioni utilizzate dal team di sviluppo sono elencate di seguito:

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

Definisce l'insieme dei processori attivati (contesto), assegnandovi un identificativo.

---

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

Assegna ad ogni processore del communicator l'identificativo 'id\_proc' (sempre associato al contesto).

In particolare, viene passato MPI\_COMM\_WORLD alla variabile *comm* e che indica il communicator a cui appartengono tutti i processori attivati e che non può essere alterato.

---

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

Restituisce ad ogni processore di MPI\_COMM\_WORLD il numero di processori nel contesto.

---

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

Si utilizza questa funzione per inviare gli argomenti letti dal processore 0 (root) a tutti i processori del contesto, incluso se stesso.

Cioè, i valori di *scelta*, *q\_num* e *time\_calc* (interi di dimensione 1) saranno copiati in tutti i processori del communicator MPI\_COMM\_WORLD.

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

Il processo che esegue questa funzione spedisce i primi *count* elementi di *buf* di tipo *datatype* al processo con identificativo *dest*.

In particolare, l'identificativo *tag* individua univocamente l'invio nel contesto *comm*.

---

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

Il processo che esegue questa funzione riceve i primi *count* elementi di **buf** di tipo *datatype* dal processo con identificativo *source*.

In particolare, l'identificativo *tag* individua univocamente la ricezione nel contesto *comm*, mentre *status* ne racchiude alcune informazioni.

---

- `int MPI_Barrier(MPI_Comm comm)`

Si utilizza questa funzione per restituire il controllo al chiamante solo dopo che tutti i processori del contesto 'comm' hanno effettuato la chiamata.

Per calcolare correttamente i tempi di esecuzione in sicurezza, si aspetta che tutti i processori siano sincronizzati.

---

- `double MPI_Wtime()`

Si utilizza per ottenere un valore di tempo in secondi rispetto ad un tempo arbitrario nel passato.

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

Si utilizza questa funzione per eseguire un'operazione collettiva su tutti i processori del contesto. In questo caso, si utilizza questa funzione per calcolare il massimo tempo tra tutti i tempi di esecuzione calcolati localmente.

---

- `int MPI_Finalize()`

Determina la fine del programma MPI.

Da questo punto in poi non è possibile richiamare altre funzioni MPI.

### 2.3.2 Libreria auxfunc.h

Questa è una delle due librerie implementate dal team di sviluppo e contiene alcune funzionalità aggiuntive per l'esecuzione del programma della somma. Il codice è disponibile nella sezione A.1 dell'appendice A a pagina 55.

- `int argToInt(char *arg)`

La funzione riceve in input una stringa, verifica che sia non vuota e che sia diversa dal null-termination character, e la converte in un intero.

---

- `double argToDouble(char *arg)`

La funzione riceve in input una stringa, verifica che sia non vuota e che sia diversa dal null-termination character, e la converte in un double.

---

- `void writeTimeCSV(int test, int strategia, int n_proc, int q_num, double t_tot)`

La funzione riceve in input tutte le informazioni necessarie allo studio dei tempi impiegati dai casi di test. Si utilizza per creare un file .csv, in modo da studiare l'output dell'esecuzione del programma in modo più semplice.

### 2.3.3 Libreria menufunc.h

Questa è l'altra delle due librerie implementate dal team di sviluppo e contiene alcune funzionalità aggiuntive per la gestione e presentazione del menù, nonchè per la creazione dei file *.pbs*. Il codice è disponibile nella sezione A.2 dell'appendice A a pagina 58.

- `double getNumberFromInput()`

La funzione effettua l'analisi del valore inserito nello 'standard input stream' e ne restituisce la rappresentazione in un double.

---

- `int getIntegerFromInput()`

La funzione effettua l'analisi del valore inserito nello 'standard input stream' e ne restituisce la rappresentazione in un intero.

---

- `void checkScelta(int scelta, int lim_inf, int lim_sup)`

La funzione riceve in input il numero scelto, il limite inferiore e quello superiore del range di definizione e verifica se il numero vi appartiene.

---

- `void createPBS(int n_proc, int scelta, int q_num, int test, int time_calc, int pbs_count)`

La funzione riceve in input l'operazione da effettuare (calcolo della somma con la strategia scelta o esecuzione della suite di testing), il numero di operandi e l'opzione di calcolo dei tempi di esecuzione per creare in maniera dinamica il file pbs necessario alla compilazione e all'esecuzione del programma sul cluster.

## 2.3.4 Altre librerie

- ***stdio.h***[2]

È la libreria utilizzata in C per gestire i flussi di dati in input / output grazie alla definizione di altri tipi di dato, come **FILE**, **size\_t** e **ssize\_t**, e altre funzioni, come

```
int fclose ( FILE* stream),  
FILE* fopen (const char* filename, const char* mode),  
int fprintf ( FILE * stream, const char * format, ... ), la funzione  
ssize_t getline(char **lineptr, size_t *n, FILE *stream) [3] per  
inizializzare un buffer con caratteri estratti dal flusso di input specificato,  
int printf ( const char * format, ... ) e  
int sprintf (char * str, const char * format, ...).
```

---

- ***stdlib.h***[4]

Questa libreria definisce alcune funzioni utili per diversi scopi, tra cui:

1. Gestione dinamica della memoria con

```
void* calloc (size_t num, size_t size) e void free (void* ptr).
```

2. Generazione di numeri pseudo-casuali con **int rand (void)** e

```
void srand (unsigned int seed).
```

3. Comunicazione con l'ambiente shell sottostante utilizzando

```
int system (const char* command) e void exit (int status).
```

4. Conversione degli argomenti passati in input al programma con

```
long int strtol (const char* str, char** endptr, int base) e  
double strtod (const char* str, char** endptr).
```

- ***errno.h***[5]

All'interno di questa libreria è definita la macro `int errno`, inizializzata a 0, utilizzabile in lettura e scrittura per gestire le casistiche di errore, assegnandole valori diversi da 0.

---

- ***limits.h***[6]

All'interno di questa libreria sono definite alcune costanti utili alla corretta rappresentazione dei valori interi nel programma, tra cui `int INT_MIN` e `int INT_MAX`.

---

- ***string.h***[7]

È una libreria che offre funzionalità per la gestione delle stringhe e, in generale, degli array. Nel codice si utilizza la funzione `size_t strlen (const char *str)` per controllare che la lunghezza degli argomenti in input non sia nulla.

---

- ***tgmath.h***[8]

Si utilizza per compilare correttamente il programma quando si utilizzano funzioni della libreria ***math.h*** con tipi **`double`**.

In particolare, questa già include tale libreria. In essa sono definite funzioni che implementano le più comuni operazioni e trasformazioni matematiche, tra cui `double log2 (double x)`, `double pow (double base, double exponent)`, `double ceil (double x)` e `double floor (double x)`.

- ***time.h***[9]

È una libreria che offre funzionalità per la gestione del tempo. Si utilizza il tipo di dato `time_t` e la funzione `time_t time (time_t* timer)` per ottenere l'istante di tempo corrente (da utilizzare poi come seme per la generazione di numeri pseudo-casuali).

---

- ***ctype.h***[10]

È una libreria che offre funzionalità per la gestione e trasformazione dei caratteri. Si utilizza la funzione `int isdigit (int c)` per verificare che un carattere inserito nello stream di input sia una cifra valida.



# Capitolo 3

## Creazione del file di esecuzione .pbs

Questo capitolo è dedicato alla descrizione del menù di creazione dei file *.pbs*. Si esamineranno le scelte che può eseguire l'utente che, alla fine dell'esecuzione, diventeranno gli argomenti in input passati al programma del calcolo della somma. Il codice è disponibile nella sezione A.3 dell'appendice A a pagina 65.

### 3.1 Inizializzazione dell'ambiente di lavoro

All'avvio del programma, si definiscono e inizializzano le variabili da utilizzare:

- `int strategia = NO_STRATEGY`

È una variabile di tipo intero per memorizzare la strategia scelta.

---

- `int q_num = 0`

È una variabile di tipo intero per memorizzare la quantità di operandi da sommare.

---

- `int test = NO_TEST`

È una variabile di tipo intero per memorizzare il caso di test da eseguire.

---

- `int time_calc = NO_TIME_CALC`

È una variabile di tipo intero per memorizzare la scelta di calcolare o meno il tempo di esecuzione impiegato.

---

- `int i = 0`

È una variabile di tipo intero utilizzata genericamente nei cicli *for*.

---

- `int pbs_count = 1`

È una variabile di tipo intero per memorizzare la numerazione dei file *.pbs* da creare.

---

- `FILE *pbs_file`

È una variabile di tipo puntatore a **FILE** che si riferisce al file *.pbs* da creare.

### 3.2 Scelta della strategia da applicare

Appena il programma si mostra all’utente, si chiede di scegliere se intende effettuare una somma con una delle tre strategie disponibili, se intende eseguire un caso di test dalla suite di testing implementata oppure se intende uscire dall’applicazione.

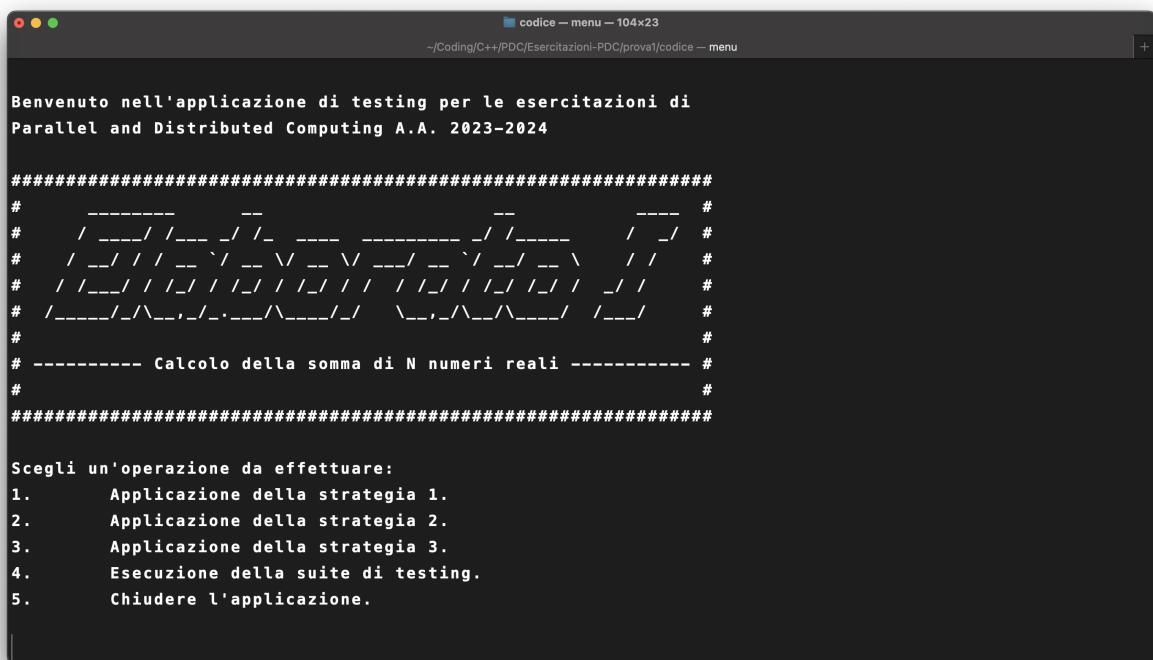


Figura 3.1: Avvio del menù per la creazione del file *.pbs*.

### 3.3 Scelta del numero di operandi da sommare

Nel caso in cui l'utente sceglie di eseguire la somma con una delle tre strategie proposte, gli viene chiesto la quantità di operandi che desidera sommare (che deve essere almeno pari a 2).

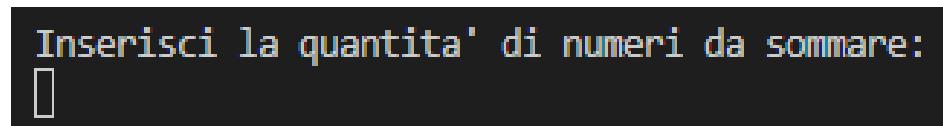


Figura 3.2: Inserimento della quantità di operandi.

La funzione *createPBS*, poi, permette l'inserimento dei valori degli operandi se tale quantità è minore o uguale a 20, come mostrato nella figura 3.3, altrimenti sono assegnati dei valori pseudo-casuali.

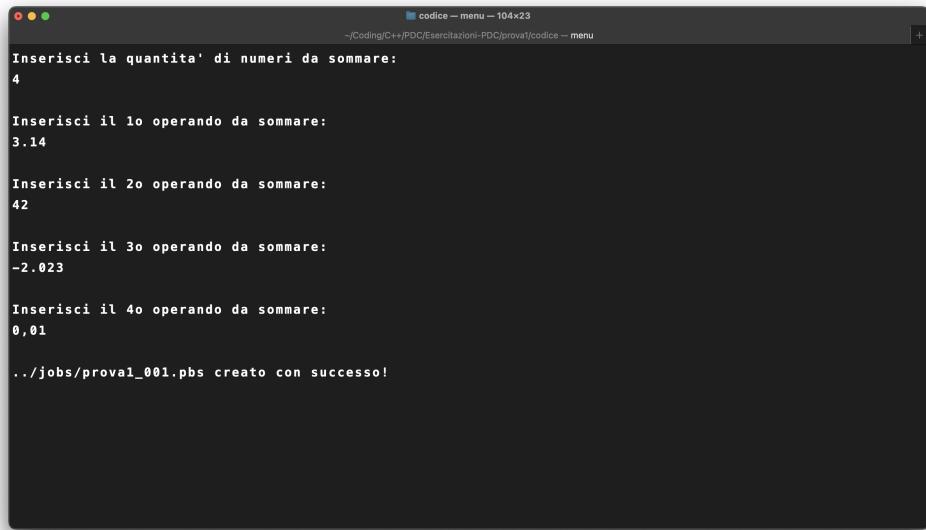


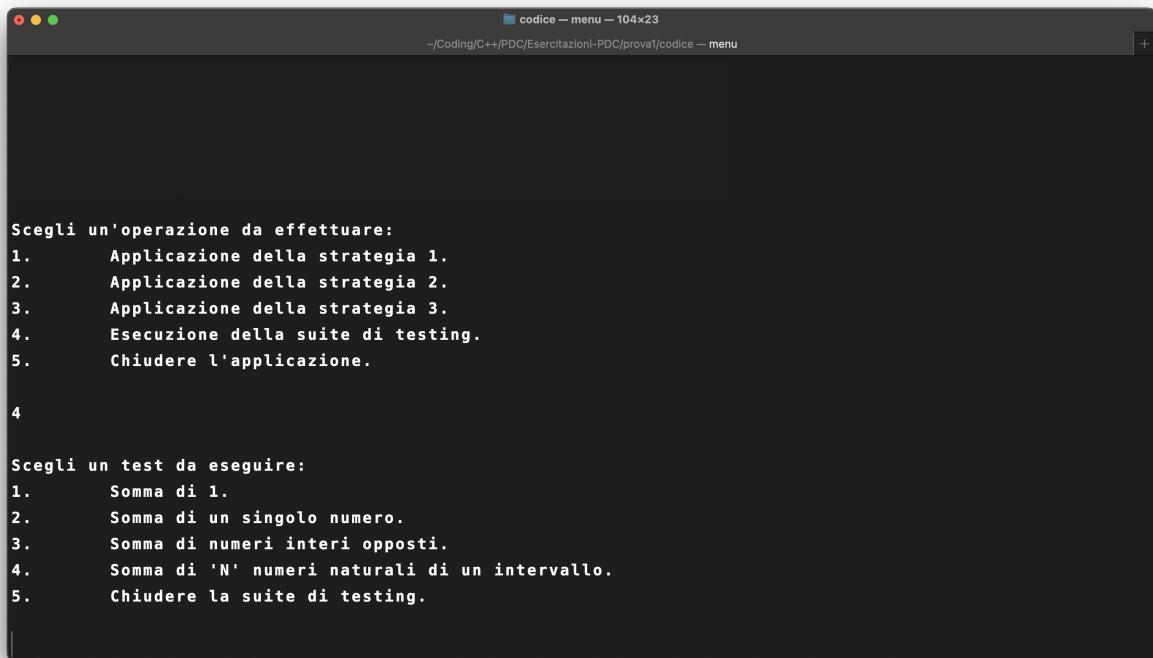
Figura 3.3: Inserimento degli operandi, se la quantità è minore o uguale a 20.

In particolare, come si evince dalla figura 3.3 se il processo di creazione del file *.pbs* termina con successo, allora si mostra il percorso relativo dove è collocato il file appena creato.

## 3.4 Applicazione della suite di testing

Come si vede nella figura 3.1 a pagina 25, dal menù l'utente può procedere alla creazione del file *.pbs* selezionando l'esecuzione di un caso di test. Nella suite di testing implementata sono disponibili:

- Somma di 1.
- Somma di un singolo numero.
- Somma di numeri interi opposti.
- Somma di 'N' numeri naturali di un intervallo.



```
codice — menu — 104x23
~/Coding/C++/PDC/Esercitazioni-PDC/prova/codice — menu

Scegli un'operazione da effettuare:
1. Applicazione della strategia 1.
2. Applicazione della strategia 2.
3. Applicazione della strategia 3.
4. Esecuzione della suite di testing.
5. Chiudere l'applicazione.

4

Scegli un test da eseguire:
1. Somma di 1.
2. Somma di un singolo numero.
3. Somma di numeri interi opposti.
4. Somma di 'N' numeri naturali di un intervallo.
5. Chiudere la suite di testing.
```

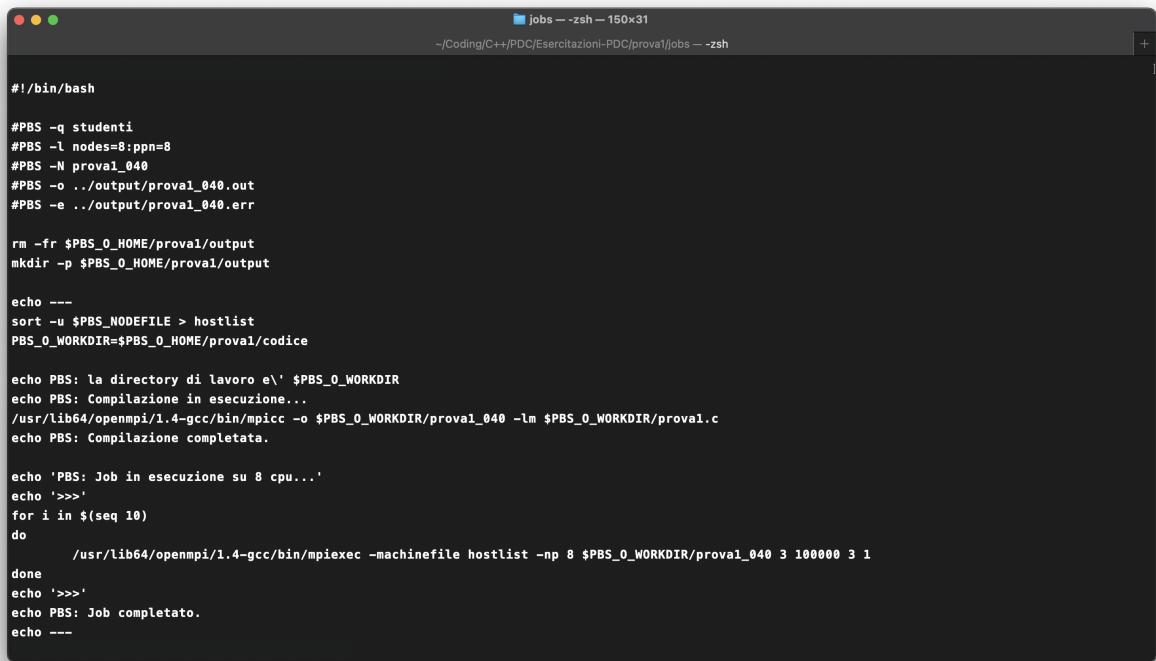
Figura 3.4: Suite di testing.

## 3.5 Scrrittura del file .pbs

Per evitare un eccessivo tempo di risposta da parte del cluster che comporterebbe un blocco delle esecuzioni in coda, attraverso il menù si generano in modo automatizzato differenti file *.pbs* per i differenti test da eseguire.

In particolare, ogni file *.pbs* generato è personalizzato in base alle scelte effettuate dall'utente come descritto nelle sezioni precedenti di questo capitolo.

La figura 3.5 mostra un esempio di file *.pbs* per la somma di 100000 numeri opposti (in riferimento al test SUM \_ OPPOSITE \_ NUMBER \_ TEST) applicando la strategia 3 su 8 processori calcolando, infine, il tempo di esecuzione impiegato da 10 esecuzioni in sequenza (si veda il ciclo for).



```
#!/bin/bash

#PBS -q studenti
#PBS -l nodes=8:ppn=8
#PBS -N prova1_040
#PBS -o ..../output/prova1_040.out
#PBS -e ..../output/prova1_040.err

rm -fr $PBS_O_HOME/prova1/output
mkdir -p $PBS_O_HOME/prova1/output

echo ---
sort -u $PBS_NODEFILE > hostlist
PBS_O_WORKDIR=$PBS_O_HOME/prova1/codice

echo PBS: la directory di lavoro e' $PBS_O_WORKDIR
echo PBS: Compilazione in esecuzione...
/usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS_O_WORKDIR/prova1_040 -lm $PBS_O_WORKDIR/prova1.c
echo PBS: Compilazione completata.

echo 'PBS: Job in esecuzione su 8 cpu...'
echo '>>>'
for i in $(seq 10)
do
    /usr/lib64/openmpi/1.4-gcc/bin/mpirun -machinefile hostlist -np 8 $PBS_O_WORKDIR/prova1_040 3 100000 3 1
done
echo '>>>'
echo PBS: Job completato.
echo ---
```

Figura 3.5: Esempio di file *.pbs*

# Capitolo 4

## Implementazione dell'algoritmo

Questo capitolo è dedicato alla descrizione del funzionamento dell'algoritmo sviluppato per il calcolo della somma in ambiente di calcolo parallelo. Si esamineranno gli input passati al programma e gli output attesi al termine dell'esecuzione dello stesso. Il codice è disponibile nella sezione A.4 dell'appendice A a pagina 68.

## 4.1 Inizializzazione dell'ambiente di lavoro

Nella fase iniziale dell'esecuzione, si definiscono e inizializzano le variabili da utilizzare:

- `int strategia = NO_STRATEGY`
- `int q_num = 0`
- `int test = NO_TEST`
- `int time_calc = NO_TIME_CALC`

Si veda la descrizione nella sezione 3.1 a pagina 24.

---

- `int id_proc = 0`

È una variabile di tipo intero che memorizza il rank di un processore.

---

- `int n_proc = 0`

È una variabile di tipo intero che memorizza il numero di processori del contesto (si veda la descrizione di `MPI_Init` a pagina 13).

---

- `int rest = 0`

È una variabile di tipo intero che memorizza il resto della divisione tra la quantità di operandi `q_num` ed il numero dei processori `n_proc` per distribuire equamente gli operandi.

- `int q_loc = 0`

È una variabile di tipo intero che indica il numero di operandi assegnati ad un singolo processore.

---

- `int tag = 0`

È una variabile di tipo intero utilizzata genericamente per assegnare un identificativo unico alle comunicazioni che avvengono grazie alle funzioni della libreria *mpi.h*.

---

- `int tmp = 0`

È una variabile di tipo intero utilizzata genericamente nel codice del programma per memorizzare valori temporanei.

---

- `int i = 0`

- `int j = 0`

Sono variabili di tipo intero utilizzate genericamente nei cicli *for*.

- `int pow_proc = 0`

- `int pow_tmp = 0`

Sono variabili di tipo intero che memorizzano valori della potenza in base 2. Si utilizzano per il calcolo degli identificativi dei processori durante l'applicazione delle strategie 2 e 3.

---

- `int log_proc = 0`

È una variabile di tipo intero che memorizza il valore del logaritmo in base 2 del numero di processori. Si utilizza per determinare se la strategia 1 è applicabile o meno.

---

- `double *op_tmp`

È un vettore di elementi `double` che ha il compito di memorizzare gli operandi locali dei processori (solo il processore con identificativo 0 lo può allocare).

---

- `double *op_loc`

È un vettore di elementi `double`, allocato da tutti i processori, che ha il compito di memorizzare solo gli operandi della somma ad esso assegnati.

---

- `double sum = 0.0`

- `double sum_parz = 0.0`

Sono variabili di tipo `double` che memorizzano le somme parziali.

- `double t_start = 0.0`

- `double t_end = 0.0`

Sono variabili di tipo `double` che memorizzano gli istanti di tempo in cui inizia e finisce il calcolo della somma e l'applicazione della strategia scelta, rispettivamente

---

- `double t_loc = 0.0`
- `double t_tot = 0.0`

Sono variabili di tipo *double* che memorizzano il tempo, locale di ogni processore e totale di tutti i processori, impiegato dal programma per calcolare la somma, rispettivamente.

---

- `int int_rand = 0`
- `double double_rand = 0.0`

Sono una variabile di tipo intero e una variabile di tipo *double* utilizzate per memorizzare valori pseudo-casuali restituiti dalla funzione *rand()*.

---

- `int gauss_inf = 0`

È una variabile di tipo intero che memorizza il limite inferiore di un intervallo di numeri interi quando si esegue il test GAUSS\_TEST.

---

- `int gauss_tmp = 0`

È una variabile di tipo intero che memorizza il conteggio dei valori nell'intervallo  $[a, b]$  quando si esegue il test GAUSS\_TEST.

---

- `MPI_Status status`

Si utilizza per memorizzare le informazioni riguardo le comunicazioni che avvengono tramite le chiamate alle funzioni della libreria *mpi.h*.

## 4.2 Inizializzazione dell’ambiente MPI

Dopo l’inizializzazione delle variabili, si procede con la configurazione dell’ambiente di lavoro per un calcolatore basato sull’architettura MIMD a memoria distribuita utilizzando la libreria *MPI*. Al termine della configurazione, tutti i processori sono capaci di interagire e di eseguire operazioni di sincronizzazione fra di loro.

## 4.3 Lettura dei dati

A questo punto, si leggono e verificano i dati forniti in input al programma (*argv*).

1. `strategia = argToInt(argv[1]);`

È la strategia per l’esecuzione della somma scelta dall’utente o da un caso di test della suite. In particolare, tutti i valori che può assumere questo input sono specificati nella sezione 2.2 a pagina 8.

---

2. `q_num = argToInt(argv[2]);`

È la quantità di operandi da sommare.

---

3. `test = argToInt(argv[3]);`

Indica il test che sarà eseguito. In particolare, tutti i valori che può assumere questo input sono specificati nella sezione 2.2 a pagina 8.

---

4. `time_calc = argToInt(argv[4]);`

Permette di verificare se è richiesto il calcolo, la stampa ed il salvataggio del tempo di esecuzione impiegato. Gli unici due valori che può assumere questo input sono specificati nella sezione 2.2 a pagina 8.

Come richiesto dalle specifiche dell'algoritmo, si controlla se le strategie 2 e 3 siano applicabili, controllando il numero di processori come spiegato nella sezione 1.2 a pagina 3.

Per realizzare questo controllo, si calcola il  $\log_2(n\_proc)$  del numero di processori e, quindi, si utilizzano le funzioni *ceil()* e *floor()*. Queste, restituendo rispettivamente il tetto e la base, permettono di capire se il logaritmo calcolato non è un valore intero, e cioè se il numero di processori non è potenza di 2.

Inoltre, si controlla che il numero di processori sia maggiore di 1. In caso contrario, si esegue il calcolo della somma in sequenziale.

Verificata l'applicabilità della strategia passata in input, si utilizza la funzione *MPI\_Bcast()* per inviare gli argomenti letti dal processore 0 (root) a tutti i processori del contesto, incluso sé stesso.

## 4.4 Distribuzione degli operandi

Il prossimo passaggio è quello di distribuire equamente gli operandi tra tutti i processori del contesto da parte del processore 0: esso, infatti, è l'unico adibito alla lettura dei dati in input. Questa operazione è fondamentale per distribuire equamente il carico di lavoro tra tutti i processori.

### 4.4.1 Calcolo della quantità locale di operandi

La distribuzione dipende dal numero di processori utilizzati per l'esecuzione del programma. In particolare, a ogni processore è assegnato un numero di operandi minimo pari alla parte intera della divisione fra il totale degli operandi e il numero di processori.

$$q_{loc} = \frac{q_{num}}{n_{proc}} \quad (4.1)$$

Successivamente, si valuta il resto di questa divisione, assegnando un operando in più ai primi *rest* processori.

### 4.4.2 Allocazione del vettore degli operandi

Per iniziare la lettura degli operandi, utilizzando la funzione `void* calloc()`

- Si alloca in memoria lo spazio utile alla memorizzazione dei *q\_loc* operandi (per tutti i processori) con il puntatore *op\_loc*.
- Si alloca in memoria lo spazio utile alla memorizzazione di tutti i *q\_num* operandi (solo per il processore 0) con il puntatore *op\_tmp*.

### 4.4.3 Lettura e/o generazione del valore degli operandi

Una volta allocato lo spazio in memoria necessario, il processore 0 si occupa della lettura e/o generazione del valore degli operandi. Questa sezione tratta in dettaglio l'implementazione dello switch-case del codice A.17 a pagina 70.

- Il caso ***NO\_TEST***

In questo caso, l'utente ha scelto dal menù di creazione del *.pbs* di non applicare alcun caso di test, ma di eseguire il programma della somma liberamente. Quindi, se  $q_{num} \leq 20$ , allora gli operandi sono letti dagli argomenti *argv* in input al programma; invece, se  $q_{num} > 20$ , allora ad ogni operando si assegna un numero casuale reale compreso nell'intervallo  $[-100, 100] \in R$ .

---

- Il caso ***SUM\_ONE\_TEST***

Il primo caso di test della suite di testing verifica se l'algoritmo della somma calcola correttamente la somma di 1. Ad ogni operando del vettore è, quindi, assegnato il valore 1. L'output atteso è, ovviamente, proprio il numero di operandi *q\_num*.

---

- Il caso ***SUM\_SINGLE\_NUMBER\_TEST***

Questo secondo caso di test è molto simile al primo, ma concede all'utente la possibilità di scegliere l'unico valore da sommare, passato come argomento in input al programma. Ad ogni operando del vettore è, quindi, assegnato il valore scelto. L'output atteso è il risultato del prodotto tra il numero di operandi *q\_num* ed il valore scelto.

- Il caso ***SUM\_OPPOSITE\_NUMBER\_TEST***

In questo caso di test, si genera innanzitutto un numero casuale compreso nell'intervallo  $[0, 100] \in R$ . Successivamente:

- Alle posizioni nella prima metà del vettore degli operandi si assegna proprio il valore reale generato.
- Alle posizioni nella seconda metà del vettore degli operandi si assegna l'opposto del valore reale generato.
- Se il vettore ha dimensioni dispari, si lascia l'ultima posizione con valore nullo (per equilibrare le somme).

Così facendo, si può determinare il successo del test se la somma totale calcolata dai processori e' pari a 0 (cioè, gli operandi si annullano).

---

- Il caso ***GAUSS\_TEST***

L'ultimo caso di test è l'implementazione della generalizzazione della cosiddetta "somma di Gauss", che calcola la somma dei numeri naturali compresi nell'intervallo  $[a, b - 1] \in N$ . L'utente sceglie nel menù il valore di  $a$ , mentre il valore di  $b$  è dato dalla somma proprio di  $a$  e  $q\_num$ .

Ad ogni operando del vettore è assegnato, quindi, un valore compreso nell'intervallo  $[a, b - 1] \in N$ . L'output atteso, invece, è dato dalla seguente formula:

$$sum = \frac{a + b - 1}{2} \cdot (b - a) \quad (4.2)$$

#### 4.4.4 Assegnazione dei valori agli operandi

L'ultimo passaggio per completare la distribuzione degli operandi, è l'assegnazione vera e propria ai vettori locali. Si divide questa fase in due momenti:

- In primo luogo, si assegnano gli operandi locali del processore 0 tramite una semplice iterazione del suo vettore locale.
- Gli operandi rimanenti sono distribuiti dal processore 0 tramite l'utilizzo della funzione *MPI\_Send*, partendo dalla posizione *0* sul vettore temporaneo e la quantità di operandi locali *tmp* (in particolare, tale quantità deve essere ricalcolata dal processore 0 perché potrebbe non essere la stessa se il numero di processori non è divisibile per la quantità di operandi).

Al contrario, tutti gli altri processori (cioè con identificativi da *1* a *p*), attendono l'invio degli operandi locali da parte del processore 0 utilizzando la funzione *MPI\_Recv* (la correttezza della comunicazione è garantita dall'utilizzo del *tag* e del valore *DISTRIBUTION\_TAG*).

## 4.5 Applicazione della strategia

Avvenuta la distribuzione degli operandi, il calcolo della somma parziale si ottiene applicando lo stesso algoritmo (elencato nella sezione A.19 a pagina 75) su tutti i processori.

A questo punto, si applica la strategia scelta dall'utente secondo quanto riportato alla sezione 1.1 a pagina 2 per quanto riguarda la strategia 1, alla sezione 1.2 a pagina 3 per quanto riguarda la strategia 2, e alla sezione 1.3 a pagina 4 per quanto riguarda la strategia 3.

## 4.6 Calcolo dei tempi di esecuzione

Nel caso in cui si sia scelto di calcolare i tempi di esecuzione, si imposta una variabile  $t\_start$  come tempo di inizio del calcolo della somma per ogni processore.

Al termine del calcolo della somma, l'algoritmo imposta la variabile  $t\_end$  come tempo di fine, calcolandone la distanza per stabilire l'intervallo temporale di esecuzione, cioè:

$$t_{loc} = t_{start} - t_{end} \quad (4.3)$$

Si effettua, poi, tramite la funzione *MPI\_Reduce* un'operazione collettiva con lo scopo di calcolare il tempo massimo fra tutti i tempi  $t\_loc$  calcolati localmente.

Infine:

1. Si stampa il tempo impiegato dall'esecuzione utilizzando la notazione scientifica.
2. Si salvano all'interno di un file *.csv* tutte le informazioni riguardanti l'esecuzione, cioè: il test eseguito, la strategia scelta, il numero di processori, la quantità di operandi e il tempo impiegato (in formato decimale).

## 4.7 Stampa dell'output e terminazione

Al termine dell'esecuzione del programma, quindi, si stampano i risultati ottenuti, preceduti dal rank del processore se si è scelto di applicare la strategia 2 o 3.

Terminato il lavoro da parte di ogni processore, si libera lo spazio allocato in memoria per i vettori degli operandi *op* e *op\_loc*, e si termina l'esecuzione del programma con la funzione *MPI\_Finalize*.

In particolare, oltre alle stampe in console, l'esecuzione del comando *qsub* su tutti i file *.pbs* creati tramite l'apposito menù descritto nel capitolo 3 a pagina 23 produce 4 file:

1. Il programma vero e proprio, la cui esecuzione è stata descritta in questo capitolo.
2. Un file con estensione *.out*, contenente tutte le stampe ed i risultati ottenuti dall'esecuzione, compreso il valore della somma degli operandi coinvolti.
3. Un file con estensione *.err*, contenente il log d'errore, per il salvataggio delle informazioni di errore.
4. Un file con estensione *.csv*, contenente tutti i tempi di esecuzione e altri parametri già descritti nella sezione 4.6 a pagina 40.



# Capitolo 5

## Analisi dei tempi e delle prestazioni

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

- **Tempo medio impiegato**  $T_m(p)$

Per ogni esperimento sono state effettuate 10 esecuzioni ed è stata, successivamente, considerata la media aritmetica dei tempi  $T(p)$  di ciascuna prova.

$$T_m(p) = \frac{\sum_{n=0}^{10} T(p)}{10} \quad (5.1)$$

---

- **Speed Up**  $S(p)$

Misura la riduzione del tempo  $T(p)$  impiegato per l'esecuzione con  $p$  processori rispetto al tempo  $T(1)$  impiegato per l'esecuzione del programma su singolo processore.

$$S(p) = \frac{T(1)}{T(p)}, \text{ con } S(p)_{ideale} = p \quad (5.2)$$

- **Overhead**  $O_h$

Misura quanto lo speed up effettivo differisca da quello ideale.

$$O_h = p \cdot T(p) - T(1), \text{ con } O_h > 0 \quad (5.3)$$

---

- **Efficienza**  $E(p)$

Misura quanto l'esecuzione del programma sfrutta il parallelismo del calcolatore.

$$E(p) = S(p)/p, \text{ con } E(p) < E(p)_{ideale} = 1 \quad (5.4)$$

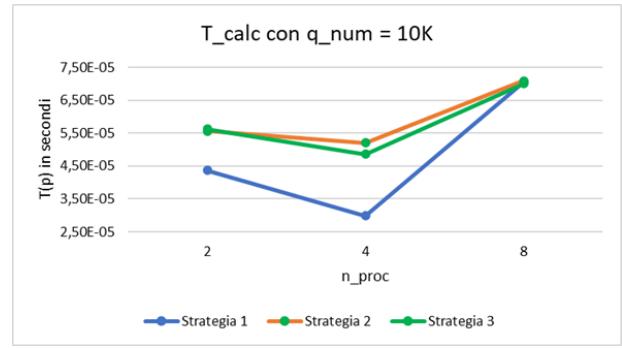
Nelle prossime sezioni, quindi, si mostrano i valori di tempo ottenuti ed i valori di speed up, overhead ed efficienza calcolati, sia in formato tabellare che tramite dei grafici (disegnati utilizzando MS Excel).

In particolare, per le seguenti analisi alle sezioni 5.1, 5.2 e 5.3 è stato eseguito il test della somma di numeri opposti (SUM\_OPPOSITE\_NUMBER\_TEST).

## 5.1 Analisi con $10^4$ operandi

		Tempo $T(p)$ in secondi		
		Strategia		
		1	2	3
n_proc	2	4,37E-05	5,56E-05	5,63E-05
	4	2,99E-05	5,21E-05	4,86E-05
	7	1,86E-02		
	8	7,06E-05	7,10E-05	7,01E-05

(a) Tabella



(b) Grafico

Figura 5.1: Tempo di esecuzione con  $10^4$  operandi.

Per il calcolo dell'Overhead, come anche dello Speed Up in figura 5.3, è stato eseguito l'algoritmo su un unico processore per ottenere il tempo di esecuzione  $T(1)$  del calcolo della somma in sequenziale, che è risultato essere il seguente:

$$T(1) = 4.85 \cdot 10^{-5} \text{ secondi.} \quad (5.5)$$

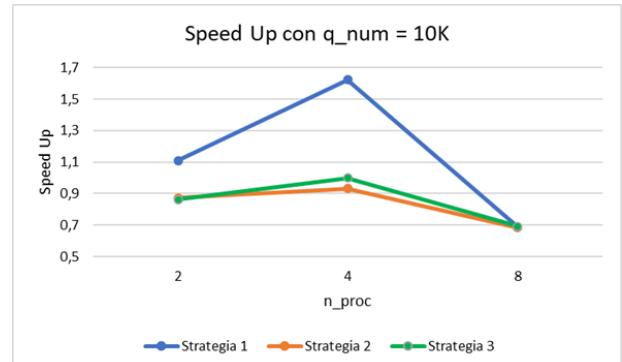
		Overhead $Oh(p)$ in secondi		
		Strategia		
		1	2	3
n_proc	2	3,89E-05	6,27E-05	6,41E-05
	4	7,10E-05	1,60E-04	1,46E-04
	7	1,30E-01		
	8	5,17E-04	5,19E-04	5,12E-04

Figura 5.2: Overhead con  $10^4$  operandi.

Infine, per il calcolo dello Speed Up, e quindi dell'Efficienza, non si è tenuto conto del caso in cui il numero di processori sia pari a 7, poichè non ne è possibile valutarne il comportamento nelle strategie 2 e 3.

		Speed Up S(p)		
		Strategia		
		1	2	3
n_proc	2	1,11	0,87	0,86
	4	1,62	0,93	1,00
	8	0,69	0,68	0,69

(a) Tabella

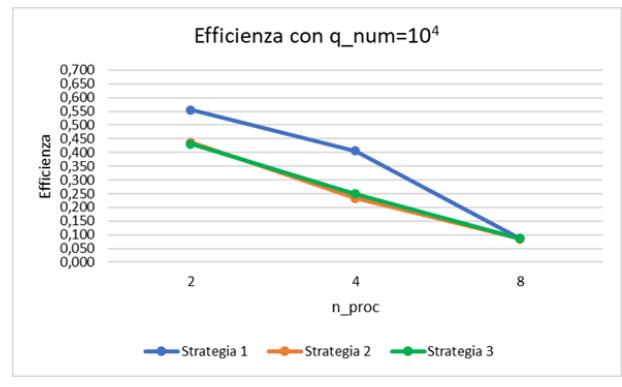


(b) Grafico

Figura 5.3: Speed Up con  $10^4$  operandi.

		Efficienza E(p)		
		Strategia		
		1	2	3
n_proc	2	0,56	0,44	0,43
	4	0,41	0,23	0,25
	8	0,09	0,09	0,09

(a) Tabella



(b) Grafico

Figura 5.4: Efficienza con  $10^4$  operandi.

I grafici mostrano chiaramente come la somma di  $10^4$  operandi con tutte le strategie impieghi un tempo di esecuzione  $T(4)$  leggermente migliore di  $T(2)$ , ma comunque troppo vicino al tempo di esecuzione  $T(1)$ .

Inoltre, si può anche notare che il tempo di esecuzione  $T(8)$  degrada in modo significativo rispetto a  $T(4)$ : il motivo è da associare all'elevato numero di comunicazioni che i processori devono eseguire per calcolare correttamente la somma.

In generale, lo speed up e l'efficienza sono di gran lunga distanti dai valori ideali.

## 5.2 Analisi con $10^5$ operandi

Il tempo di esecuzione  $T(1)$  del calcolo della somma in sequenziale è il seguente:

$$T(1) = 4.68 \cdot 10^{-4} \text{ secondi.} \quad (5.6)$$

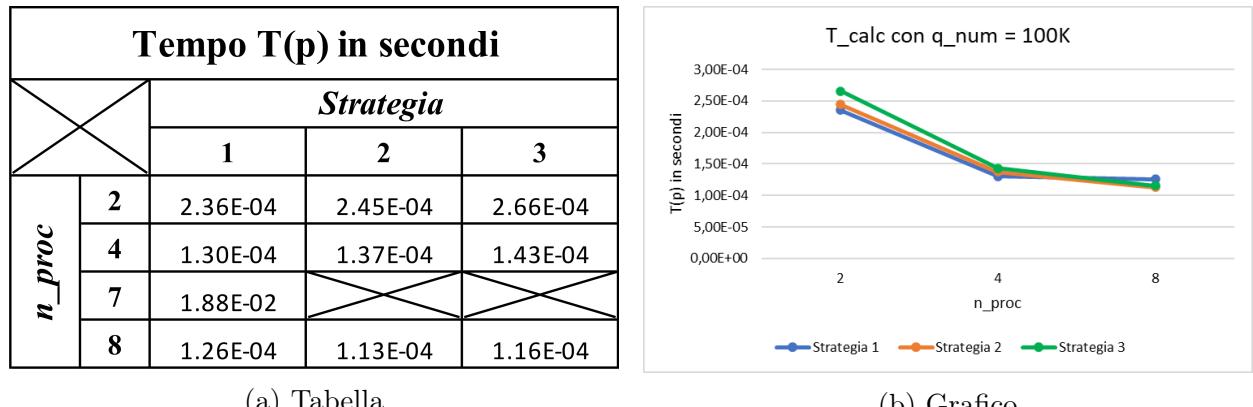


Figura 5.5: Tempo di esecuzione con  $10^5$  operandi.

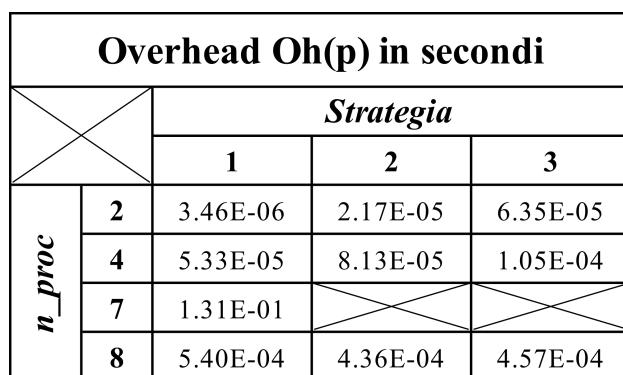
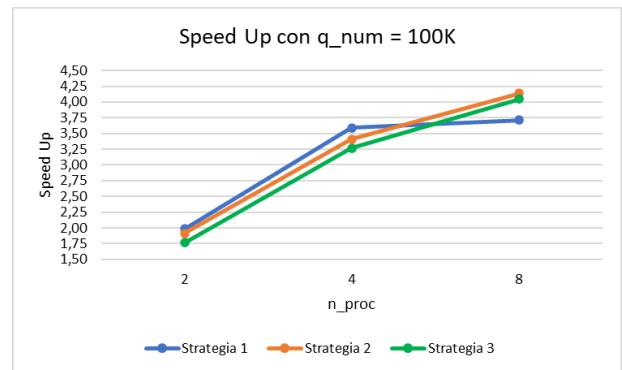


Figura 5.6: Overhead con  $10^5$  operandi.

Speed Up S(p)				
		Strategia		
		1	2	3
n_proc	2	1.99	1.91	1.76
	4	3.59	3.41	3.27
	8	3.71	4.14	4.05

(a) Tabella

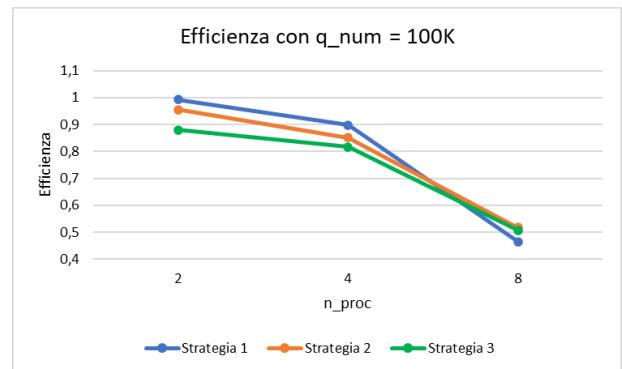


(b) Grafico

Figura 5.7: Speed up con  $10^5$  operandi.

Efficienza E(p)				
		Strategia		
		1	2	3
n_proc	2	0.993	0.956	0.880
	4	0.898	0.852	0.817
	8	0.464	0.517	0.506

(a) Tabella



(b) Grafico

Figura 5.8: Efficienza con  $10^5$  operandi.

L'esecuzione del programma con un numero di operandi pari a  $10^5$ , come si può notare nella tabella 5.8a e nel grafico 5.8b, sfrutta meglio il parallelismo del calcolatore rispetto all'esecuzione vista nella sezione 5.1 per 2 e 4 processori.

Tuttavia, per 8 processori si continua ad ottenere valori di speed up ed efficienza lontani da quelli ideali.

### 5.3 Analisi con $10^6$ operandi

Il tempo di esecuzione  $T(1)$  del calcolo della somma in sequenziale è il seguente:

$$T(1) = 4.25 \cdot 10^{-3} \text{ secondi.} \quad (5.7)$$

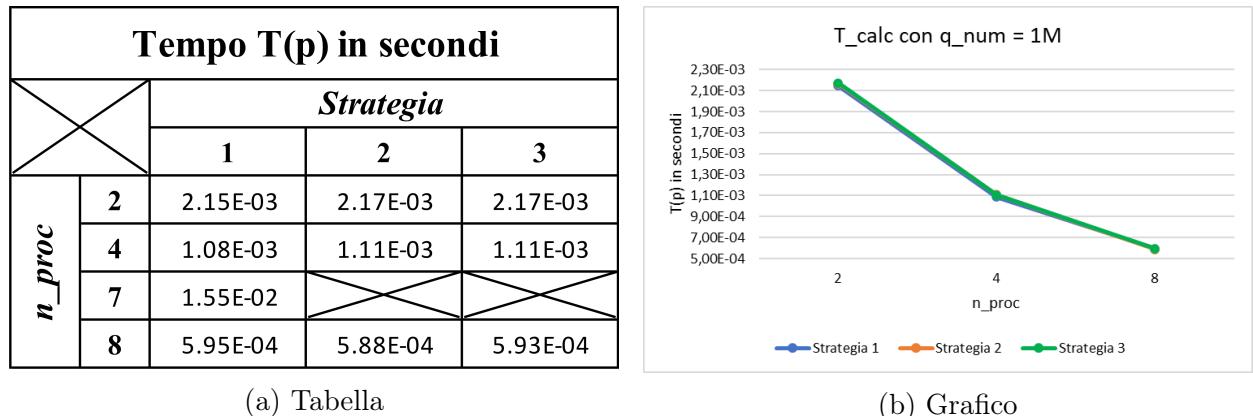


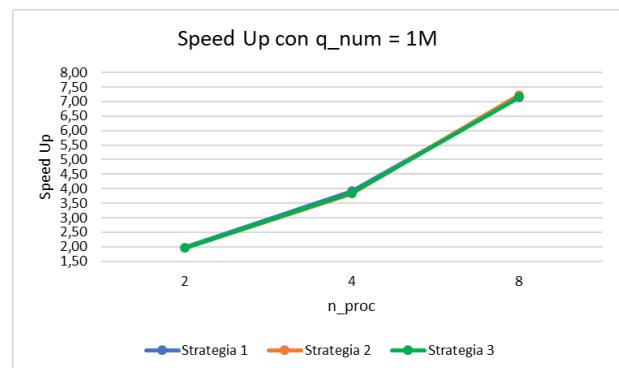
Figura 5.9: Tempo di esecuzione con  $10^6$  operandi.

Overhead $Oh(p)$ in secondi				
		Strategia		
		1	2	3
$n\_proc$	2	4.64E-05	8.44E-05	9.83E-05
	4	8.99E-05	1.84E-04	1.73E-04
	7	1.05E-01	X	X
	8	5.12E-04	4.55E-04	4.98E-04

Figura 5.10: Overhead con  $10^6$  operandi.

Speed Up S(p)				
		Strategia		
		1	2	3
n_proc	2	1.98	1.96	1.95
	4	3.92	3.83	3.84
	8	7.14	7.23	7.16

(a) Tabella

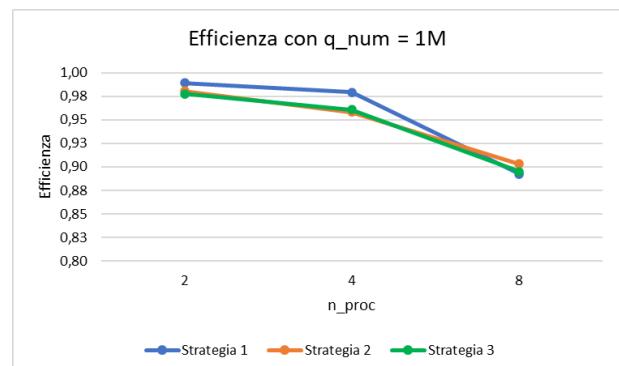


(b) Grafico

Figura 5.11: Speed up con  $10^6$  operandi.

Efficienza E(p)				
		Strategia		
		1	2	3
n_proc	2	0.989	0.981	0.977
	4	0.979	0.958	0.961
	8	0.892	0.903	0.895

(a) Tabella



(b) Grafico

Figura 5.12: Efficienza con  $10^6$  operandi.

L'esecuzione di quest'ultimo caso di test mostra, finalmente, dei valori ottimi rispetto alle sezioni 5.1 e 5.2 anche per l'esecuzione su 8 processori.

Quindi, a valle dell'analisi di tutti i tempi  $T(p)$ , tutti gli overhead  $O_h$ , tutti gli speed up  $S(p)$  e tutte le efficienze  $E(p)$ , si può concludere che l'algoritmo della somma è scalabile in quanto l'efficienza è (circa) costante al crescere del numero di processori  $n_{proc}$  e della dimensione del problema  $q_{num}$ , come mostrato nelle seguenti figure:

Efficienza E(p) per strategia 1				
		<i>q_num</i>		
		10 000	100 000	1 000 000
<i>n_proc</i>	2	0.555	0.993	0.989
	4	0.406	0.898	0.979
	8	0.086	0.464	0.892

(a) Strategia 1

Efficienza E(p) per strategia 2				
		<i>q_num</i>		
		10 000	100 000	1 000 000
<i>n_proc</i>	2	0.436	0.956	0.981
	4	0.233	0.852	0.958
	8	0.085	0.517	0.903

(b) Strategia 2

Efficienza E(p) per strategia 3				
		<i>q_num</i>		
		10 000	100 000	1 000 000
<i>n_proc</i>	2	0.431	0.880	0.977
	4	0.250	0.817	0.961
	8	0.087	0.506	0.895

(c) Strategia 3

Figura 5.13: Confronti dei valori dell'efficienza



# Capitolo 6

## Conclusioni

In conclusione, il presente progetto rappresenta un’importante esplorazione dell’efficacia dell’implementazione di un algoritmo per la somma di  $N$  numeri in un ambiente di calcolo parallelo con  $p$  processori su una architettura MIMD a memoria distribuita.

- Nel capitolo 1 è stato delineato chiaramente il problema affrontato, presentando le strategie di somma in contesti di parallelizzazione.
- Nel capitolo 2 è stata fornita una descrizione dei codici di errori utili a rappresentare le cause di terminazione del programma, delle costanti utili per la definizione di alcuni parametri nel codice e delle funzioni adoperate o sviluppate dal team di sviluppo e dalle librerie di linguaggio disponibili.
- Nel capitolo 3 è stato esaminato il processo di creazione del file `.pbs` necessario per l’esecuzione dell’algoritmo sul cluster, attraverso un apposito menù di creazione.
- Nel capitolo 4 è stata presentata l’implementazione dell’algoritmo vero e proprio, il cuore del nostro progetto, specificandone nel dettaglio le varie fasi dell’esecuzione.

- Nel capitolo 5, infine, è stata condotta un'analisi dettagliata delle prestazioni a seguito dell'esecuzione del programma sul cluster. I risultati ottenuti sono stati fondamentali per dimostrarne l'efficacia e la scalabilità in un ambiente di calcolo parallelo.

La documentazione termina con l'appendice A che mostra tutto il codice sorgente del progetto.

## 6.1 Futuri sviluppi

In particolare, si è pensato anche ai possibili miglioramenti che si possono implementare nel programma.

- Primo tra tutti, la capacità del programma di calcolare automaticamente la media dei tempi direttamente dai dati salvati nel file .csv in output utilizzando utility per l'elaborazione del testo come, ad esempio, *sed* e *awk*.
- Un altro possibile miglioramento, potrebbe essere quello di salvare / stampare i tempi di esecuzione solo nel caso in cui il test eseguito restituisce il risultato desiderato (al momento, è compito dell'utente e/o dello sviluppatore verificare la correttezza dei dati).
- Infine, si è pensato alla possibilità di aggiungere al programma la capacità di leggere in input file di testo contenenti i valori degli operandi, per ottenere una maggior versatilità.

# Appendice A

## Elenco dei listati

### A.1 auxfunc.c

```
1 int argToInt(char *arg) {
2
3     char *p;
4     long out_long = 0;
5     errno = 0;
6
7     if (strlen(arg) == 0) {
8         printf("Errore nella lettura degli argomenti di input!\n\n");
9         printf("Esecuzione terminata.\n");
10        MPI_Finalize();
11        exit(EMPTY_ARG_ERROR);
12    }
13
14    out_long = strtol(arg, &p, 10);
15
16    if (*p != '\0' || errno != 0) {
17        printf("Errore nella lettura degli argomenti di input!\n\n");
18        printf("Esecuzione terminata.\n");
19        MPI_Finalize();
20        exit(INPUT_ARG_ERROR);
```

```

21 }
22
23 if (out_long < INT_MIN || out_long > INT_MAX) {
24     printf("Errore nella lettura degli argomenti di input!\n\n"
25         );
26     printf("Esecuzione terminata.\n");
27     MPI_Finalize();
28     exit(NOT_INT_ARG_ERROR);
29 }
30
31 }  


```

Listing A.1: La funzione *int argToInt(char\*)*

```

1 double argToDouble(char *arg) {
2
3     char *p;
4     double out_double = 0.0;
5
6     errno = 0;
7
8     if (strlen(arg) == 0) {
9         printf("Errore nella lettura degli argomenti di input!\n\n"
10            );
11         printf("Esecuzione terminata.\n");
12         MPI_Finalize();
13         exit(EMPTY_ARG_ERROR);
14     }
15
16     out_double = strtod(arg, &p);
17
18     if (*p != '\0' || errno != 0) {
19         printf("Errore nella lettura degli argomenti di input!\n\n"
20            );
21         printf("Esecuzione terminata.\n");
22         MPI_Finalize();
23         exit(INPUT_ARG_ERROR);
24     }
25
26     return out_double;
27 }
28

```

25 }

Listing A.2: La funzione *double argToDouble(char\*)*

```
1 void writeTimeCSV(int test, int strategia, int n_proc, int
2   q_num, double t_tot) {
3
4   FILE *csv_file;
5
6   system(MKDIR_PATH" -p "CSV_TIME_PATH);
7
8   if ((csv_file = fopen(CSV_TIME_PATH"/"NOME_PROVA"_time.csv",
9     "a")) == NULL) {
10     printf("Errore durante l'esecuzione!\n");
11     printf("Applicazione terminata.\n");
12     MPI_Finalize();
13     exit(FILE_OPENING_ERROR);
14
15   fprintf(csv_file, "%d,%d,%d,%d,%1.9f\n",
16     test, strategia, n_proc, q_num, t_tot);
17
18   fclose(csv_file);
19
20   printf("%s aggiornato con successo!\n\n", CSV_TIME_PATH"/"
21   NOME_PROVA"_time.csv");
22 }
```

Listing A.3: La funzione *void writeTimeCSV(int, int, int, int, double)*

## A.2 menufunc.c

```
1 Si omette il codice di questa funzione in quanto contiene
2 solo le stampe del titolo dell'elaborato.
```

Listing A.4: La funzione *void printTitle()*

```
1 double getNumberFromInput() {
2     double out = 0;
3     char *buffer = NULL;
4     size_t bufsize = 0;
5     ssize_t chars_read;
6
7     // Inizializzazione del buffer con caratteri estratti dallo
8     // stream di input
9     chars_read = getline(&buffer, &bufsize, stdin);
10    printf("\n");
11
12    if (chars_read < 0) {
13
14        printf("Errore nella lettura dell'input!");
15        printf("Applicazione terminata.\n");
16        free(buffer);
17        exit(INPUT_LINE_ERROR);
18    } else {
19
20        double digit_val = 0.0;
21        int i = 0;
22        int exp_whole = 0;
23        int exp_fract = 0;
24        int pos_point = -1;
25
26        for (i = chars_read-2; i >= 0; i--) {
27            if (buffer[i] != '.' && buffer[i] != ',' ) {
28                exp_fract--;
29            } else {
30                pos_point = chars_read - (exp_fract * (-1)) - 2;
31                break;
32            }
33        }
34    }
35 }
```

```

34
35     i = chars_read - 2;
36     while(i >= 0) {
37         if (i > pos_point && pos_point != -1) {
38
39             if (isdigit(buffer[i])) {
40                 digit_val = pow(10, exp_fract) * (digit_val + (buffer
41                     [i] - '0')); // per ottenere il valore intero del
42                     carattere ASCII
43                 out = out + digit_val;
44             } else {
45                 out = 0.0;
46                 printf("Puoi inserire solo valori numerici reali!\n")
47                     ;
48                 printf("Applicazione terminata.\n");
49                 free(buffer);
50                 exit(NOT_REAL_NUMBER_ERROR);
51             }
52
53             exp_fract++;
54
55         } else if (i < pos_point || pos_point == -1) {
56
57             if (isdigit(buffer[i])) {
58                 digit_val = pow(10, exp_whole) * (digit_val + (buffer
59                     [i] - '0'));
60                 out = out + digit_val;
61             } else {
62                 if (i == 0 && buffer[0] == '-') {
63                     out = out * (-1);
64                 } else {
65                     out = 0.0;
66                     printf("Puoi inserire solo valori numerici reali!\n"
67                         );
68                     printf("Applicazione terminata.\n");
69                     free(buffer);
70                     exit(NOT_REAL_NUMBER_ERROR);
71                 }
72             }
73         }
74     }
75 }
```

```

68         exp_whole++;
69     }
70
71     digit_val = 0.0;
72     i--;
73 }
74 }
75 }
76
77 free(buffer);
78 return out;
79 }
```

Listing A.5: La funzione *double getNumberFromInput()*

```

1 int getIntegerFromInput() {
2
3     double out_double = 0.0;
4     int out_integer = 0;
5
6     out_double = getNumberFromInput();
7     out_integer = (int)out_double;
8
9     if (out_integer < 0) {
10        printf("Puoi inserire solo valori numerici naturali!\n");
11        printf("Applicazione terminata.\n");
12        exit(NOT_NATURAL_NUMBER_ERROR);
13    }
14
15    return out_integer;
16 }
```

Listing A.6: La funzione *int getIntegerFromInput()*

```

1 void checkScelta(int scelta, int lim_inf, int lim_sup) {
2     if(!(scelta >= lim_inf && scelta <= lim_sup)) {
3         printf("Puoi inserire solo un valore numerico intero
4             compreso tra %d e %d!\n", lim_inf, lim_sup);
5         printf("Applicazione terminata.\n");
6         exit(NOT_IN_RANGE_ERROR);
7     }
8 }
```

7 }

Listing A.7: La funzione *void checkScelta(int, int, int)*

```
1 void createPBS(int n_proc, int strategia, int q_num, int test,
2                 int time_calc, int pbs_count) {
3
4     char pbs_path[255] = {};
5     FILE *pbs_file;
6
7     int i = 0, int_op = 0;
8     double double_op = 0.0;
9
10    if (pbs_count <= 1) {
11        system(RM_PATH" -rf .../jobs");
12        system(MKDIR_PATH" -p .../jobs");
13    }
14
15    sprintf(pbs_path, " .../jobs/"NOME_PROVA"_%03d.pbs", pbs_count)
16    ;
17
18    if ((pbs_file = fopen(pbs_path, "a")) == NULL) {
19        printf("Errore durante l'esecuzione!\n");
20        printf("Applicazione terminata.\n");
21        exit(FILE_OPENING_ERROR);
22    }
23
24    fprintf(pbs_file,
25            "#!/bin/bash\n"
26            "\n"
27            "#PBS -q studenti\n"
28            "#PBS -l nodes="NODE_NUMBER";
29
30    if (time_calc == OK_TIME_CALC) {
31        fprintf(pbs_file, ":ppn="NODE_PROCESS);
32    }
33
34    fprintf(pbs_file,
35            "\n#PBS -N " NOME_PROVA "_%03d\n"
36            "#PBS -o .../output/" NOME_PROVA "_%03d.out\n"
37            "#PBS -e .../output/" NOME_PROVA "_%03d.err\n"
```

```

36      "\n",
37  pbs_count, pbs_count, pbs_count);
38
39  if (pbs_count <= 1) {
40      fprintf(pbs_file,
41          "rm -fr $PBS_O_HOME/prova1/output\n"
42          "mkdir -p $PBS_O_HOME/prova1/output\n\n"
43      );
44 }
45
46 fprintf(pbs_file, "echo --- \n");
47
48 if (time_calc == OK_TIME_CALC) {
49     fprintf(pbs_file, "sort -u $PBS_NODEFILE > hostlist\n");
50 }
51
52 fprintf(pbs_file,
53     "PBS_O_WORKDIR=$PBS_O_HOME/" NOME_PROVA "/codice\n"
54     "\n"
55     "echo PBS: la directory di lavoro e\' $PBS_O_WORKDIR\n"
56     "
57     "echo PBS: Compilazione in esecuzione...\n"
58     "/usr/lib64/openmpi/1.4-gcc/bin/mpicc "
59     "-o $PBS_O_WORKDIR/" NOME_PROVA "_%03d -lm
60         $PBS_O_WORKDIR/" NOME_PROVA ".c\n"
61     "echo PBS: Compilazione completata.\n"
62     "\n"
63     "echo 'PBS: Job in esecuzione su %d cpu...'\n"
64     "echo '>>>\n",
65 pbs_count, n_proc);
66
67 if (time_calc == OK_TIME_CALC) {
68     fprintf(pbs_file,
69         "for i in $(seq 10)\n"
70         "do\n"
71         "\t/usr/lib64/openmpi/1.4-gcc/bin/mpiexec "
72         "-machinefile hostlist -np %d ",
73     n_proc);
74 } else {

```

```

73     fprintf(pbs_file,
74         "/usr/lib64/openmpi/1.4-gcc/bin/mpexec "
75         "-machinefile $PBS_NODEFILE -n %d ",
76         n_proc);
77 }
78
79 fprintf(pbs_file, "$PBS_O_WORKDIR/" NOME_PROVA "_%03d %d %d %
80     d %d",
81     pbs_count, strategia, q_num, test, time_calc);
82
83 /* Come richiesto dalle specifiche dell'algoritmo, se la
84    quantita'
85    di operandi e' minore o uguale a 20, allora il valore di
86    ogni
87    singolo operando deve essere specificato dall'utente.
88 */
89
90 switch(test) {
91     case NO_TEST: {
92         if (q_num <= OP_MAX_QUANTITY) {
93             for (i=1; i <= q_num; i++) {
94                 printf("Inserisci il %do operando da sommare: \n", i)
95                 ;
96                 double_op = getNumberFromInput();
97                 fprintf(pbs_file, " %f", double_op);
98             }
99             break;
100         }
101         case SUM_SINGLE_NUMBER_TEST: {
102             printf("Inserisci il valore dell'operando da sommare:\n")
103             ;
104             int_op = getIntegerFromInput();
105             fprintf(pbs_file, " %d", int_op);
106             break;
107         }
108         case GAUSS_TEST: {
109             printf("Inserisci il limite inferiore dell'intervallo:\n"
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
279
280
281
282
283
284
285
286
287
288
289
289
290
291
292
293
294
295
296
297
298
299
299
300
301
302
303
304
305
306
307
308
309
309
310
311
312
313
314
315
316
317
318
319
319
320
321
322
323
324
325
326
327
328
329
329
330
331
332
333
334
335
336
337
338
339
339
340
341
342
343
344
345
346
347
348
349
349
350
351
352
353
354
355
356
357
358
359
359
360
361
362
363
364
365
366
367
368
369
369
370
371
372
373
374
375
376
377
378
379
379
380
381
382
383
384
385
386
387
388
389
389
390
391
392
393
394
395
396
397
398
399
399
400
401
402
403
404
405
406
407
408
409
409
410
411
412
413
414
415
416
417
418
419
419
420
421
422
423
424
425
426
427
428
429
429
430
431
432
433
434
435
436
437
438
439
439
440
441
442
443
444
445
446
447
448
449
449
450
451
452
453
454
455
456
457
458
459
459
460
461
462
463
464
465
466
467
468
469
469
470
471
472
473
474
475
476
477
478
479
479
480
481
482
483
484
485
486
487
488
489
489
490
491
492
493
494
495
496
497
498
499
499
500
501
502
503
504
505
506
507
508
509
509
510
511
512
513
514
515
516
517
518
519
519
520
521
522
523
524
525
526
527
528
529
529
530
531
532
533
534
535
536
537
538
539
539
540
541
542
543
544
545
546
547
548
549
549
550
551
552
553
554
555
556
557
558
559
559
560
561
562
563
564
565
566
567
568
569
569
570
571
572
573
574
575
576
577
578
579
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
598
599
599
600
601
602
603
604
605
606
607
608
609
609
610
611
612
613
614
615
616
617
618
619
619
620
621
622
623
624
625
626
627
628
629
629
630
631
632
633
634
635
636
637
637
638
639
639
640
641
642
643
644
645
646
647
648
649
649
650
651
652
653
654
655
656
657
658
659
659
660
661
662
663
664
665
666
667
668
669
669
670
671
672
673
674
675
676
677
678
679
679
680
681
682
683
684
685
686
687
688
689
689
690
691
692
693
694
695
696
697
697
698
699
699
700
701
702
703
704
705
706
707
708
709
709
710
711
712
713
714
715
716
717
718
719
719
720
721
722
723
724
725
726
727
728
729
729
730
731
732
733
734
735
736
737
737
738
739
739
740
741
742
743
744
745
746
747
748
748
749
750
751
752
753
754
755
756
757
758
759
759
760
761
762
763
764
765
766
767
768
769
769
770
771
772
773
774
775
776
777
778
778
779
779
780
781
782
783
784
785
786
787
787
788
789
789
790
791
792
793
794
795
796
797
797
798
799
799
800
801
802
803
804
805
806
807
808
809
809
810
811
812
813
814
815
816
817
817
818
819
819
820
821
822
823
824
825
826
827
827
828
829
829
830
831
832
833
834
835
836
837
837
838
839
839
840
841
842
843
844
845
846
847
847
848
849
849
850
851
852
853
854
855
856
857
858
858
859
860
861
862
863
864
865
866
866
867
868
868
869
869
870
871
872
873
874
875
875
876
877
877
878
878
879
879
880
881
882
883
884
885
885
886
887
887
888
889
889
890
891
892
893
893
894
895
895
896
896
897
897
898
898
899
899
900
901
901
902
902
903
903
904
904
905
905
906
906
907
907
908
908
909
909
910
910
911
911
912
912
913
913
914
914
915
915
916
916
917
917
918
918
919
919
920
920
921
921
922
922
923
923
924
924
925
925
926
926
927
927
928
928
929
929
930
930
931
931
932
932
933
933
934
934
935
935
936
936
937
937
938
938
939
939
940
940
941
941
942
942
943
943
944
944
945
945
946
946
947
947
948
948
949
949
950
950
951
951
952
952
953
953
954
954
955
955
956
956
957
957
958
958
959
959
960
960
961
961
962
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
970
971
971
972
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
980
981
981
982
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
990
991
991
992
992
993
993
994
994
995
995
996
996
997
997
998
998
999
999
1000
1000
1001
1001
1002
1002
1003
1003
1004
1004
1005
1005
1006
1006
1007
1007
1008
1008
1009
1009
1010
1010
1011
1011
1012
1012
1013
1013
1014
1014
1015
1015
1016
1016
1017
1017
1018
1018
1019
1019
1020
1020
1021
1021
1022
1022
1023
1023
1024
1024
1025
1025
1026
1026
1027
1027
1028
1028
1029
1029
1030
1030
1031
1031
1032
1032
1033
1033
1034
1034
1035
1035
1036
1036
1037
1037
1038
1038
1039
1039
1040
1040
1041
1041
1042
1042
1043
1043
1044
1044
1045
1045
1046
1046
1047
1047
1048
1048
1049
1049
1050
1050
1051
1051
1052
1052
1053
1053
1054
1054
1055
1055
1056
1056
1057
1057
1058
1058
1059
1059
1060
1060
1061
1061
1062
1062
1063
1063
1064
1064
1065
1065
1066
1066
1067
1067
1068
1068
1069
1069
1070
1070
1071
1071
1072
1072
1073
1073
1074
1074
1075
1075
1076
1076
1077
1077
1078
1078
1079
1079
1080
1080
1081
1081
1082
1082
1083
1083
1084
1084
1085
1085
1086
1086
1087
1087
1088
1088
1089
1089
1090
1090
1091
1091
1092
1092
1093
1093
1094
1094
1095
1095
1096
1096
1097
1097
1098
1098
1099
1099
1100
1100
1101
1101
1102
1102
1103
1103
1104
1104
1105
1105
1106
1106
1107
1107
1108
1108
1109
1109
1110
1110
1111
1111
1112
1112
1113
1113
1114
1114
1115
1115
1116
1116
1117
1117
1118
1118
1119
1119
1120
1120
1121
1121
1122
1122
1123
1123
1124
1124
1125
1125
1126
1126
1127
1127
1128
1128
1129
1129
1130
1130
1131
1131
1132
1132
1133
1133
1134
1134
1135
1135
1136
1136
1137
1137
1138
1138
1139
1139
1140
1140
1141
1141
1142
1142
1143
1143
1144
1144
1145
1145
1146
1146
1147
1147
1148
1148
1149
1149
1150
1150
1151
1151
1152
1152
1153
1153
1154
1154
1155
1155
1156
1156
1157
1157
1158
1158
1159
1159
1160
1160
1161
1161
1162
1162
1163
1163
1164
1164
1165
1165
1166
1166
1167
1167
1168
1168
1169
1169
1170
1170
1171
1171
1172
1172
1173
1173
1174
1174
1175
1175
1176
1176
1177
1177
1178
1178
1179
1179
1180
1180
1181
1181
1182
1182
1183
1183
1184
1184
1185
1185
1186
1186
1187
1187
1188
1188
1189
1189
1190
1190
1191
1191
1192
1192
1193
1193
1194
1194
1195
1195
1196
1196
1197
1197
1198
1198
1199
1199
1200
1200
1201
1201
1202
1202
1203
1203
1204
1204
1205
1205
1206
1206
1207
1207
1208
1208
1209
1209
1210
1210
1211
1211
1212
1212
1213
1213
1214
1214
1215
1215
1216
1216
1217
1217
1218
1218
1219
1219
1220
1220
1221
1221
1222
1222
1223
1223
1224
1224
1225
1225
1226
1226
1227
1227
1228
1228
1229
1229
1230
1230
1231
1231
1232
1232
1233
1233
1234
1234
1235
1235
1236
1236
1237
1237
1238
1238
1239
1239
1240
1240
1241
1241
1242
1242
1243
1243
1244
1244
1245
1245
1246
1246
1247
1247
1248
1248
1249
1249
1250
1250
1251
1251
1252
1252
1253
1253
1254
1254
1255
1255
1256
1256
1257
1257
1258
1258
1259
1259
1260
1260
1261
1261
1262
1262
1263
1263
1264
1264
1265
1265
1266
1266
1267
1267
1268
1268
1269
1269
1270
1270
1271
1271
1272
1272
1273
1273
1274
1274
1275
1275
1276
1276
1277
1277
1278
1278
1279
1279
1280
1280
1281
1281
1282
1282
1283
1283
1284
1284
1285
1285
1286
1286
1287
1287
1288
1288
1289
1289
1290
1290
1291
1291
1292
1292
1293
1293
1294
1294
1295
1295
1296
1296
1297
1297
1298
1298
1299
1299
1300
1300
1301
1301
1302
1302
1303
1303
1304
1304
1305
1305
1306
1306
1307
1307
1308
1308
1309
1309
1310
1310
1311
1311
1312
1312
1313
1313
1314
1314
1315
1315
1316
1316
1317
1317
1318
1318
1319
1319
1320
1320
1321
1321
1322
1322
1323
1323
1324
1324
1325
1325
1326
1326
1327
1327
1328
1328
1329
1329
1330
1330
1331
1331
1332
1332
1333
1333
1334
1334
1335
1335
1336
1336
1337
1337
1338
1338
1339
1339
1340
1340
1341
1341
1342
1342
1343
1343
1344
1344
1345
1345
1346
1346
1347
1347
1348
1348
1349
1349
1350
1350
1351
1351
1352
1352
1353
1353
1354
1354
1355
1355
1356
1356
1357
1357
1358
1358
1359
1359
1360
1360
1361
1361
1362
1362
1363
1363
1364
1364
1365
1365
1366
1366
1367
1367
1368
1368
1369
1369
1370
1370
1371
1371
1372
1372
1373
1373
1374
1374
1375
1375
1376
1376
1377
1377
1378
1378
1379
1379
1380
1380
1381
1381
1382
1382
1383
1383
1384
1384
1385
1385
1386
1386
1387
1387
1388
1388
1389
1389
1390
1390
1391
1391
1392
1392
1393
1393
1394
1394
1395
1395
1396
1396
1397
1397
1398
1398
1399
1399
1400
1400
1401
1401
1402
1402
1403
1403
1404
1404
1405
1405
1406
1406
1407
1407
1408
1408
1409
1409
1410
1410
1411
1411
1412
1412
1413
1413
1414
1414
1415
1415
1416
1416
1417
1417
1418
1418
1419
1419
1420
1420
1421
1421
1422
1422
1423
1423
1424
1424
1425
1425
1426
1426
1427
1427
1428
1428
1429
1429
1430
1430
1431
1431
1432
1432
1433
1433
1434
1434
1435
1435
1436
1436
1437
1437
1438
1438
1439
1439
1440
1440
1441
1441
1442
1442
1443
1443
1444
1444
1445
1445
1446
1446
1447
1447
1448
1448
1449
1449
1450
1450
1451
1451
1452
1452
1453
1453
1454
1454
1455
1455
1456
1456
1457
1457
1458
1458
1459
1459
1460
1460
1461
1461
1462
1462
1463
1463
1464
1464
1465
1465
1466
1466
1467
1467
1468
1468
1469
1469
1470
1470
1471
1471
1472
1472
1473
1473
1474
1474
1475
1475
1476
1476
1477
1477
1478
1478
1479
1479
1480
1480
1481
1481
1482
1482
1483
1483
1484
1484
1485
1485
1486
1486
1487
1487
1488
1488
1489
1489
1490
1490
1491
1491
1492
1492
1493
1493
1494
1494
1495
1495
1496
1496
1497
1497
1498
1498
1499
1499
1500
1500
1501
1501
1502
1502
1503
1503
1504
1504
1505
1505
1506
1506
1507
1507
1508
1508
1509
1509
1510
1510
1511
1511
1512
1512
1513
1513
1514
1514
1515
1515
1516
1516
1517
1517
1518
1518
1519
1519
1520
1520
1521
1521
1522
1522
1523
1523
1524
1524
1525
1525
1526
1526
1527
1527
1528
1528
1529
1529
1530
1530
1531
1531
1532
1532
1533
1533
1534
1534
1535
1535
1536
1536
1537
1537
1538
1538
1539
1539
1540
1540
1541
1541
1542
1542
1543
1543
1544
1544
1545
1545
1546
1546
1547
1547
1548
1548
1549
1549
1550
1550
1551
1551
1552
1552
1553
1553
1554
1554
1555
1555
1556
1556
1557
1557
1558
1558
1559
1559
1560
1560
1561
1561
1562
1562
1563
1563
1564
1564
1565
1565
1566
1566
1567
1567
1568
1568
1569
1569
1570
1570
1571
1571
1572
1572
1573
1573
1574
1574
1575
1575
1576
1576
1577
1577
1578
1578
1579
1579
1580
1580
1581
1581
1582
1582
1583
1583
1584
1584
1585
1585
1586
1586
1587
1587
1588
1588
1589
1589
1590
1590
1591
1591
1592
1592
1593
1593
1594
1594
1595
1595
1596
1596
1597
1597
1598
1598
1599
1599
1600
1600
1601
1601
1602
1602
1603
1603
1604
1604
1605
1605
1606
1606
1607
1607
1608
1608
1609
1609
1610
1610
1611
1611
1612
1612
1613
1613
1614
1614
1615
1615
1616
1616
1617
1617
1618
1618
1619
1619
1620
1620
1621
1621
1622
1622
1623
1623
1624
1624
1625
1625
1626
1626
1627
1627
1628
1628
1629
1629
1630
1630
1631
1631
1632
1632
1633
1633
1634
1634
1635
1635
1636
1636
1637
1637
1638
1638
1639
1639
1640
1640
1641
1641
1642
1642
1643
1643
1644

```

```

    );
107    int_op = getIntegerFromInput();
108    fprintf(pbs_file, " %d", int_op);
109    break;
110 }
111 default:
112     break;
113 }
114
115 if (time_calc == OK_TIME_CALC) {
116     fprintf(pbs_file, "\ndone\n");
117 } else {
118     fprintf(pbs_file, "\n");
119 }
120
121 fprintf(pbs_file,
122         "echo >>>`\n"
123         "echo PBS: Job completato.\n"
124         "echo ---\n"
125 );
126
127 fclose(pbs_file);
128 printf("%s creato con successo!%\n\n", pbs_path);
129 }

```

Listing A.8: La funzione *void createPBS(int, int, int, int, int, int)*

### A.3 menu.c

```
1 int strategia = NO_STRATEGY, q_num = 0, test = NO_TEST,
2     time_calc = NO_TIME_CALC;
3 int i = 0, pbs_count = 1;
4 FILE *pbs_file;
```

Listing A.9: Inizializzazione dell'ambiente di lavoro

```
1 printf("\n");
2 printf("Benvenuto nell'applicazione di testing per le
3     esercitazioni di\n");
4 printf("Parallel and Distributed Computing A.A. 2023-2024\n\n");
5 printTitle();
6 do {
7     printf("Scegli un'operazione da effettuare: \n");
8     printf("%d. \t Applicazione della strategia 1.\n",
9            FIRST_STRATEGY);
10    printf("%d. \t Applicazione della strategia 2.\n",
11           SECOND_STRATEGY);
12    printf("%d. \t Applicazione della strategia 3.\n",
13           THIRD_STRATEGY);
14    printf("%d. \t Esecuzione della suite di testing.\n",
15           TESTING_SUITE);
16    printf("%d. \t Chiudere l'applicazione.\n\n",
17           EXIT_APPLICATION);
18    strategia = getIntegerFromInput();
19    checkScelta(strategia, FIRST_STRATEGY, EXIT_APPLICATION);
20
21    if (strategia <= THIRD_STRATEGY) {
22        // SCELTA DEL NUMERO DI OPERANDI DA SOMMARE
23        // CREAZIONE DEL FILE DI ESECUZIONE .PBS
24    } else if (strategia == TESTING_SUITE) {
25        // APPLICAZIONE DELLA SUITE DI TESTING
26    }
27}
```

```
23 } while (strategia != EXIT_APPLICATION);
```

Listing A.10: Scelta della strategia da applicare

```
1 printf("Inserisci la quantita' di numeri da sommare: \n")
2 ;
3 q_num = getIntegerFromInput();
4
5 if (q_num <= 1) {
6     printf("Devi inserire almeno due operandi!\n");
7     printf("Applicazione terminata.\n");
8     exit(NOT_ENOUGH_OPERANDS);
}
```

Listing A.11: Scelta del numero di operando da sommare (caso NO\_TEST)

```
1 createPBS(8, strategia, q_num, NO_TEST, NO_TIME_CALC,
2           pbs_count++);
```

Listing A.12: Creazione del file di esecuzione *.pbs* (caso NO\_TEST)

```
1 /*
2     Nell'eseguire la suite di testing, il programma fornira
3     :
4     - il risultato della somma applicando la 1a strategia;
5     - il risultato della somma applicando la 2a strategia (
6         se possibile);
7     - il risultato della somma applicando la 3a strategia (
8         se possibile);
9     - i tempi di esecuzione relativi all'esecuzione delle 3
10    strategie.
11 */
12
13 printf("Scegli un test da eseguire: \n");
14 printf("%d. \t Somma di 1.\n", SUM_ONE_TEST);
15 printf("%d. \t Somma di un singolo numero.\n",
16        SUM_SINGLE_NUMBER_TEST);
17 printf("%d. \t Somma di numeri interi opposti.\n",
18        SUM_OPPOSITE_NUMBER_TEST);
19 printf("%d. \t Somma di 'N' numeri naturali di un
20        intervallo.\n", GAUSS_TEST);
21 printf("%d. \t Chiudere la suite di testing.\n\n",
22        EXIT_TEST);
```

```

15     test = getIntegerFromInput();
16     checkScelta(test, SUM_ONE_TEST, EXIT_TEST);
17
18     if (test != EXIT_TEST) {
19         for (strategia = FIRST_STRATEGY; strategia <=
20             THIRD_STRATEGY; strategia++) {
21             for (i = OP_MIN_EXP_TEST; i <= OP_MAX_EXP_TEST; i++)
22                 {
23
24                     q_num = pow(10, i);
25
26                     createPBS(1, strategia, q_num, test, OK_TIME_CALC,
27                               pbs_count++);
28                     createPBS(2, strategia, q_num, test, OK_TIME_CALC,
29                               pbs_count++);
30                     createPBS(4, strategia, q_num, test, OK_TIME_CALC,
31                               pbs_count++);
32                     createPBS(7, strategia, q_num, test, OK_TIME_CALC,
33                               pbs_count++);
34                     createPBS(8, strategia, q_num, test, OK_TIME_CALC,
35                               pbs_count++);
36                 }
37             }
38         }
39     }

```

Listing A.13: Applicazione della suite di testing

## A.4 proval.c

```
1 int strategia = NO_STRATEGY, q_num = 0;
2 int test = NO_TEST, time_calc = NO_TIME_CALC;
3
4 int id_proc = 0, n_proc = 0, rest = 0;
5 int q_loc = 0, tag = 0;
6 int tmp = 0, i = 0, j = 0;
7 int pow_proc = 0, pow_tmp = 0, log_proc = 0;
8
9 double *op_tmp, *op_loc;
10 double sum = 0.0, sum_parz = 0.0;
11
12 double t_start = 0.0, t_end = 0.0;
13 double t_loc = 0.0, t_tot = 0.0;
14
15 int int_rand = 0, gauss_inf = 0, gauss_tmp = 0;
16 double double_rand = 0.0;
17
18 MPI_Status status;
19
20 srand(time(NULL));
```

Listing A.14: Inizializzazione dell'ambiente di lavoro

```
1 MPI_Init(&argc, &argv);
2 MPI_Comm_rank(MPI_COMM_WORLD, &id_proc);
3 MPI_Comm_size(MPI_COMM_WORLD, &n_proc);
```

Listing A.15: Inizializzazione dell'ambiente MPI

```
1 if (id_proc == 0) {
2     printf("Inizio esecuzione.\n\n");
3
4     /*
5         Si affida al primo processore con id_proc == 0 il compito
6         di leggere gli argomenti in input 'argv[]'.
7     */
8
9     if (argc < 4) {
10         printf("Errore nella lettura degli argomenti di input!\n\n");
11     }
12 }
```

```

11     printf("Esecuzione terminata.\n");
12     MPI_Finalize();
13     exit(NOT_ENOUGH_ARGS_ERROR);
14 }
15
16 strategia = argToInt(argv[1]);
17 q_num = argToInt(argv[2]);
18 test = argToInt(argv[3]);
19 time_calc = argToInt(argv[4]);
20
21 if (n_proc > 1) {
22
23     if (!(ceil(log2(n_proc)) == floor(log2(n_proc)))) {
24         printf("Il numero di processori (%d) non e' potenza di
25               2.\n", n_proc);
26         strategia = FIRST_STRATEGY;
27     } else {
28         log_proc = log2(n_proc);
29     }
30
31     printf("Applicazione della strategia %d.\n", strategia);
32 } else {
33
34     printf("Il numero di processori (%d) non e' sufficiente
35           per applicare la strategia %d.\n", n_proc, strategia);
36     strategia = NO_STRATEGY;
37     printf("Calcolo della somma in sequenziale.\n");
38 }
39
40 }
41
42 MPI_Bcast(&strategia, 1, MPI_INT, 0, MPI_COMM_WORLD);
43 MPI_Bcast(&q_num, 1, MPI_INT, 0, MPI_COMM_WORLD);
44 MPI_Bcast(&test, 1, MPI_INT, 0, MPI_COMM_WORLD);
45 MPI_Bcast(&time_calc, 1, MPI_INT, 0, MPI_COMM_WORLD);
46 MPI_Bcast(&log_proc, 1, MPI_INT, 0, MPI_COMM_WORLD);

```

Listing A.16: Lettura dei dati

```

1 // Si distribuisce equamente il numero di operandi tra tutti
2 // i processori.
3 q_loc = q_num / n_proc;
4
5 // Gli operandi rimanenti sono distribuiti tra i primi 'rest'
6 // processori.
7 rest = q_num % n_proc;
8 if (id_proc < rest) {
9     q_loc++;
10 }
11
12 op_loc = (double *)calloc (q_loc, sizeof(double));
13
14 if (id_proc == 0) {
15
16     /*
17         Come prima, si affida al primo processore il compito di
18         leggere
19         gli operandi della somma (da argv[] se q_num <= 20)
20         oppure di
21         assegnarne un valore random.
22     */
23
24     for (i=0; i < n_proc; i++) {
25
26         /*
27             Il seguente controllo serve per aggiornare il
28             processore
29             con id_proc == 0 sulla quantita' locale di operandi di
30             tutti gli altri processori.
31         */
32
33         tmp = q_loc;
34         if (rest != 0 && i >= rest) {
35             tmp--;
36         }
37
38         op_tmp = (double *)calloc (tmp, sizeof(double));
39
40         if (tmp > 0) {
41             for (j=0; j < tmp; j++) {
42                 op_tmp[j] = 0;
43             }
44         }
45
46         if (id_proc == 0) {
47             for (j=0; j < tmp; j++) {
48                 op_tmp[j] = q_loc[j];
49             }
50         }
51
52         if (tmp > 0) {
53             for (j=0; j < tmp; j++) {
54                 q_loc[j] = 0;
55             }
56         }
57
58         if (id_proc == 0) {
59             for (j=0; j < tmp; j++) {
60                 q_loc[j] = op_tmp[j];
61             }
62         }
63
64         if (tmp > 0) {
65             for (j=0; j < tmp; j++) {
66                 op_tmp[j] = 0;
67             }
68         }
69
70         if (id_proc == 0) {
71             for (j=0; j < tmp; j++) {
72                 op_tmp[j] = q_loc[j];
73             }
74         }
75
76         if (tmp > 0) {
77             for (j=0; j < tmp; j++) {
78                 q_loc[j] = 0;
79             }
80         }
81
82         if (id_proc == 0) {
83             for (j=0; j < tmp; j++) {
84                 q_loc[j] = op_tmp[j];
85             }
86         }
87
88         if (tmp > 0) {
89             for (j=0; j < tmp; j++) {
90                 op_tmp[j] = 0;
91             }
92         }
93
94         if (id_proc == 0) {
95             for (j=0; j < tmp; j++) {
96                 q_loc[j] = op_tmp[j];
97             }
98         }
99
100        if (tmp > 0) {
101            for (j=0; j < tmp; j++) {
102                op_tmp[j] = 0;
103            }
104        }
105
106        if (id_proc == 0) {
107            for (j=0; j < tmp; j++) {
108                q_loc[j] = op_tmp[j];
109            }
110        }
111
112        if (tmp > 0) {
113            for (j=0; j < tmp; j++) {
114                op_tmp[j] = 0;
115            }
116        }
117
118        if (id_proc == 0) {
119            for (j=0; j < tmp; j++) {
120                q_loc[j] = op_tmp[j];
121            }
122        }
123
124        if (tmp > 0) {
125            for (j=0; j < tmp; j++) {
126                op_tmp[j] = 0;
127            }
128        }
129
130        if (id_proc == 0) {
131            for (j=0; j < tmp; j++) {
132                q_loc[j] = op_tmp[j];
133            }
134        }
135
136        if (tmp > 0) {
137            for (j=0; j < tmp; j++) {
138                op_tmp[j] = 0;
139            }
140        }
141
142        if (id_proc == 0) {
143            for (j=0; j < tmp; j++) {
144                q_loc[j] = op_tmp[j];
145            }
146        }
147
148        if (tmp > 0) {
149            for (j=0; j < tmp; j++) {
150                op_tmp[j] = 0;
151            }
152        }
153
154        if (id_proc == 0) {
155            for (j=0; j < tmp; j++) {
156                q_loc[j] = op_tmp[j];
157            }
158        }
159
160        if (tmp > 0) {
161            for (j=0; j < tmp; j++) {
162                op_tmp[j] = 0;
163            }
164        }
165
166        if (id_proc == 0) {
167            for (j=0; j < tmp; j++) {
168                q_loc[j] = op_tmp[j];
169            }
170        }
171
172        if (tmp > 0) {
173            for (j=0; j < tmp; j++) {
174                op_tmp[j] = 0;
175            }
176        }
177
178        if (id_proc == 0) {
179            for (j=0; j < tmp; j++) {
180                q_loc[j] = op_tmp[j];
181            }
182        }
183
184        if (tmp > 0) {
185            for (j=0; j < tmp; j++) {
186                op_tmp[j] = 0;
187            }
188        }
189
190        if (id_proc == 0) {
191            for (j=0; j < tmp; j++) {
192                q_loc[j] = op_tmp[j];
193            }
194        }
195
196        if (tmp > 0) {
197            for (j=0; j < tmp; j++) {
198                op_tmp[j] = 0;
199            }
200        }
201
202        if (id_proc == 0) {
203            for (j=0; j < tmp; j++) {
204                q_loc[j] = op_tmp[j];
205            }
206        }
207
208        if (tmp > 0) {
209            for (j=0; j < tmp; j++) {
210                op_tmp[j] = 0;
211            }
212        }
213
214        if (id_proc == 0) {
215            for (j=0; j < tmp; j++) {
216                q_loc[j] = op_tmp[j];
217            }
218        }
219
220        if (tmp > 0) {
221            for (j=0; j < tmp; j++) {
222                op_tmp[j] = 0;
223            }
224        }
225
226        if (id_proc == 0) {
227            for (j=0; j < tmp; j++) {
228                q_loc[j] = op_tmp[j];
229            }
230        }
231
232        if (tmp > 0) {
233            for (j=0; j < tmp; j++) {
234                op_tmp[j] = 0;
235            }
236        }
237
238        if (id_proc == 0) {
239            for (j=0; j < tmp; j++) {
240                q_loc[j] = op_tmp[j];
241            }
242        }
243
244        if (tmp > 0) {
245            for (j=0; j < tmp; j++) {
246                op_tmp[j] = 0;
247            }
248        }
249
250        if (id_proc == 0) {
251            for (j=0; j < tmp; j++) {
252                q_loc[j] = op_tmp[j];
253            }
254        }
255
256        if (tmp > 0) {
257            for (j=0; j < tmp; j++) {
258                op_tmp[j] = 0;
259            }
260        }
261
262        if (id_proc == 0) {
263            for (j=0; j < tmp; j++) {
264                q_loc[j] = op_tmp[j];
265            }
266        }
267
268        if (tmp > 0) {
269            for (j=0; j < tmp; j++) {
270                op_tmp[j] = 0;
271            }
272        }
273
274        if (id_proc == 0) {
275            for (j=0; j < tmp; j++) {
276                q_loc[j] = op_tmp[j];
277            }
278        }
279
280        if (tmp > 0) {
281            for (j=0; j < tmp; j++) {
282                op_tmp[j] = 0;
283            }
284        }
285
286        if (id_proc == 0) {
287            for (j=0; j < tmp; j++) {
288                q_loc[j] = op_tmp[j];
289            }
290        }
291
292        if (tmp > 0) {
293            for (j=0; j < tmp; j++) {
294                op_tmp[j] = 0;
295            }
296        }
297
298        if (id_proc == 0) {
299            for (j=0; j < tmp; j++) {
300                q_loc[j] = op_tmp[j];
301            }
302        }
303
304        if (tmp > 0) {
305            for (j=0; j < tmp; j++) {
306                op_tmp[j] = 0;
307            }
308        }
309
310        if (id_proc == 0) {
311            for (j=0; j < tmp; j++) {
312                q_loc[j] = op_tmp[j];
313            }
314        }
315
316        if (tmp > 0) {
317            for (j=0; j < tmp; j++) {
318                op_tmp[j] = 0;
319            }
320        }
321
322        if (id_proc == 0) {
323            for (j=0; j < tmp; j++) {
324                q_loc[j] = op_tmp[j];
325            }
326        }
327
328        if (tmp > 0) {
329            for (j=0; j < tmp; j++) {
330                op_tmp[j] = 0;
331            }
332        }
333
334        if (id_proc == 0) {
335            for (j=0; j < tmp; j++) {
336                q_loc[j] = op_tmp[j];
337            }
338        }
339
340        if (tmp > 0) {
341            for (j=0; j < tmp; j++) {
342                op_tmp[j] = 0;
343            }
344        }
345
346        if (id_proc == 0) {
347            for (j=0; j < tmp; j++) {
348                q_loc[j] = op_tmp[j];
349            }
350        }
351
352        if (tmp > 0) {
353            for (j=0; j < tmp; j++) {
354                op_tmp[j] = 0;
355            }
356        }
357
358        if (id_proc == 0) {
359            for (j=0; j < tmp; j++) {
360                q_loc[j] = op_tmp[j];
361            }
362        }
363
364        if (tmp > 0) {
365            for (j=0; j < tmp; j++) {
366                op_tmp[j] = 0;
367            }
368        }
369
370        if (id_proc == 0) {
371            for (j=0; j < tmp; j++) {
372                q_loc[j] = op_tmp[j];
373            }
374        }
375
376        if (tmp > 0) {
377            for (j=0; j < tmp; j++) {
378                op_tmp[j] = 0;
379            }
380        }
381
382        if (id_proc == 0) {
383            for (j=0; j < tmp; j++) {
384                q_loc[j] = op_tmp[j];
385            }
386        }
387
388        if (tmp > 0) {
389            for (j=0; j < tmp; j++) {
390                op_tmp[j] = 0;
391            }
392        }
393
394        if (id_proc == 0) {
395            for (j=0; j < tmp; j++) {
396                q_loc[j] = op_tmp[j];
397            }
398        }
399
400        if (tmp > 0) {
401            for (j=0; j < tmp; j++) {
402                op_tmp[j] = 0;
403            }
404        }
405
406        if (id_proc == 0) {
407            for (j=0; j < tmp; j++) {
408                q_loc[j] = op_tmp[j];
409            }
410        }
411
412        if (tmp > 0) {
413            for (j=0; j < tmp; j++) {
414                op_tmp[j] = 0;
415            }
416        }
417
418        if (id_proc == 0) {
419            for (j=0; j < tmp; j++) {
420                q_loc[j] = op_tmp[j];
421            }
422        }
423
424        if (tmp > 0) {
425            for (j=0; j < tmp; j++) {
426                op_tmp[j] = 0;
427            }
428        }
429
430        if (id_proc == 0) {
431            for (j=0; j < tmp; j++) {
432                q_loc[j] = op_tmp[j];
433            }
434        }
435
436        if (tmp > 0) {
437            for (j=0; j < tmp; j++) {
438                op_tmp[j] = 0;
439            }
440        }
441
442        if (id_proc == 0) {
443            for (j=0; j < tmp; j++) {
444                q_loc[j] = op_tmp[j];
445            }
446        }
447
448        if (tmp > 0) {
449            for (j=0; j < tmp; j++) {
450                op_tmp[j] = 0;
451            }
452        }
453
454        if (id_proc == 0) {
455            for (j=0; j < tmp; j++) {
456                q_loc[j] = op_tmp[j];
457            }
458        }
459
460        if (tmp > 0) {
461            for (j=0; j < tmp; j++) {
462                op_tmp[j] = 0;
463            }
464        }
465
466        if (id_proc == 0) {
467            for (j=0; j < tmp; j++) {
468                q_loc[j] = op_tmp[j];
469            }
470        }
471
472        if (tmp > 0) {
473            for (j=0; j < tmp; j++) {
474                op_tmp[j] = 0;
475            }
476        }
477
478        if (id_proc == 0) {
479            for (j=0; j < tmp; j++) {
480                q_loc[j] = op_tmp[j];
481            }
482        }
483
484        if (tmp > 0) {
485            for (j=0; j < tmp; j++) {
486                op_tmp[j] = 0;
487            }
488        }
489
490        if (id_proc == 0) {
491            for (j=0; j < tmp; j++) {
492                q_loc[j] = op_tmp[j];
493            }
494        }
495
496        if (tmp > 0) {
497            for (j=0; j < tmp; j++) {
498                op_tmp[j] = 0;
499            }
500        }
501
502        if (id_proc == 0) {
503            for (j=0; j < tmp; j++) {
504                q_loc[j] = op_tmp[j];
505            }
506        }
507
508        if (tmp > 0) {
509            for (j=0; j < tmp; j++) {
510                op_tmp[j] = 0;
511            }
512        }
513
514        if (id_proc == 0) {
515            for (j=0; j < tmp; j++) {
516                q_loc[j] = op_tmp[j];
517            }
518        }
519
520        if (tmp > 0) {
521            for (j=0; j < tmp; j++) {
522                op_tmp[j] = 0;
523            }
524        }
525
526        if (id_proc == 0) {
527            for (j=0; j < tmp; j++) {
528                q_loc[j] = op_tmp[j];
529            }
530        }
531
532        if (tmp > 0) {
533            for (j=0; j < tmp; j++) {
534                op_tmp[j] = 0;
535            }
536        }
537
538        if (id_proc == 0) {
539            for (j=0; j < tmp; j++) {
540                q_loc[j] = op_tmp[j];
541            }
542        }
543
544        if (tmp > 0) {
545            for (j=0; j < tmp; j++) {
546                op_tmp[j] = 0;
547            }
548        }
549
550        if (id_proc == 0) {
551            for (j=0; j < tmp; j++) {
552                q_loc[j] = op_tmp[j];
553            }
554        }
555
556        if (tmp > 0) {
557            for (j=0; j < tmp; j++) {
558                op_tmp[j] = 0;
559            }
560        }
561
562        if (id_proc == 0) {
563            for (j=0; j < tmp; j++) {
564                q_loc[j] = op_tmp[j];
565            }
566        }
567
568        if (tmp > 0) {
569            for (j=0; j < tmp; j++) {
570                op_tmp[j] = 0;
571            }
572        }
573
574        if (id_proc == 0) {
575            for (j=0; j < tmp; j++) {
576                q_loc[j] = op_tmp[j];
577            }
578        }
579
580        if (tmp > 0) {
581            for (j=0; j < tmp; j++) {
582                op_tmp[j] = 0;
583            }
584        }
585
586        if (id_proc == 0) {
587            for (j=0; j < tmp; j++) {
588                q_loc[j] = op_tmp[j];
589            }
590        }
591
592        if (tmp > 0) {
593            for (j=0; j < tmp; j++) {
594                op_tmp[j] = 0;
595            }
596        }
597
598        if (id_proc == 0) {
599            for (j=0; j < tmp; j++) {
600                q_loc[j] = op_tmp[j];
601            }
602        }
603
604        if (tmp > 0) {
605            for (j=0; j < tmp; j++) {
606                op_tmp[j] = 0;
607            }
608        }
609
610        if (id_proc == 0) {
611            for (j=0; j < tmp; j++) {
612                q_loc[j] = op_tmp[j];
613            }
614        }
615
616        if (tmp > 0) {
617            for (j=0; j < tmp; j++) {
618                op_tmp[j] = 0;
619            }
620        }
621
622        if (id_proc == 0) {
623            for (j=0; j < tmp; j++) {
624                q_loc[j] = op_tmp[j];
625            }
626        }
627
628        if (tmp > 0) {
629            for (j=0; j < tmp; j++) {
630                op_tmp[j] = 0;
631            }
632        }
633
634        if (id_proc == 0) {
635            for (j=0; j < tmp; j++) {
636                q_loc[j] = op_tmp[j];
637            }
638        }
639
640        if (tmp > 0) {
641            for (j=0; j < tmp; j++) {
642                op_tmp[j] = 0;
643            }
644        }
645
646        if (id_proc == 0) {
647            for (j=0; j < tmp; j++) {
648                q_loc[j] = op_tmp[j];
649            }
650        }
651
652        if (tmp > 0) {
653            for (j=0; j < tmp; j++) {
654                op_tmp[j] = 0;
655            }
656        }
657
658        if (id_proc == 0) {
659            for (j=0; j < tmp; j++) {
660                q_loc[j] = op_tmp[j];
661            }
662        }
663
664        if (tmp > 0) {
665            for (j=0; j < tmp; j++) {
666                op_tmp[j] = 0;
667            }
668        }
669
670        if (id_proc == 0) {
671            for (j=0; j < tmp; j++) {
672                q_loc[j] = op_tmp[j];
673            }
674        }
675
676        if (tmp > 0) {
677            for (j=0; j < tmp; j++) {
678                op_tmp[j] = 0;
679            }
680        }
681
682        if (id_proc == 0) {
683            for (j=0; j < tmp; j++) {
684                q_loc[j] = op_tmp[j];
685            }
686        }
687
688        if (tmp > 0) {
689            for (j=0; j < tmp; j++) {
690                op_tmp[j] = 0;
691            }
692        }
693
694        if (id_proc == 0) {
695            for (j=0; j < tmp; j++) {
696                q_loc[j] = op_tmp[j];
697            }
698        }
699
700        if (tmp > 0) {
701            for (j=0; j < tmp; j++) {
702                op_tmp[j] = 0;
703            }
704        }
705
706        if (id_proc == 0) {
707            for (j=0; j < tmp; j++) {
708                q_loc[j] = op_tmp[j];
709            }
710        }
711
712        if (tmp > 0) {
713            for (j=0; j < tmp; j++) {
714                op_tmp[j] = 0;
715            }
716        }
717
718        if (id_proc == 0) {
719            for (j=0; j < tmp; j++) {
720                q_loc[j] = op_tmp[j];
721            }
722        }
723
724        if (tmp > 0) {
725            for (j=0; j < tmp; j++) {
726                op_tmp[j] = 0;
727            }
728        }
729
730        if (id_proc == 0) {
731            for (j=0; j < tmp; j++) {
732                q_loc[j] = op_tmp[j];
733            }
734        }
735
736        if (tmp > 0) {
737            for (j=0; j < tmp; j++) {
738                op_tmp[j] = 0;
739            }
740        }
741
742        if (id_proc == 0) {
743            for (j=0; j < tmp; j++) {
744                q_loc[j] = op_tmp[j];
745            }
746        }
747
748        if (tmp > 0) {
749            for (j=0; j < tmp; j++) {
750                op_tmp[j] = 0;
751            }
752        }
753
754        if (id_proc == 0) {
755            for (j=0; j < tmp; j++) {
756                q_loc[j] = op_tmp[j];
757            }
758        }
759
760        if (tmp > 0) {
761            for (j=0; j < tmp; j++) {
762                op_tmp[j] = 0;
763            }
764        }
765
766        if (id_proc == 0) {
767            for (j=0; j < tmp; j++) {
768                q_loc[j] = op_tmp[j];
769            }
770        }
771
772        if (tmp > 0) {
773            for (j=0; j < tmp; j++) {
774                op_tmp[j] = 0;
775            }
776        }
777
778        if (id_proc == 0) {
779            for (j=0; j < tmp; j++) {
780                q_loc[j] = op_tmp[j];
781            }
782        }
783
784        if (tmp > 0) {
785            for (j=0; j < tmp; j++) {
786                op_tmp[j] = 0;
787            }
788        }
789
790        if (id_proc == 0) {
791            for (j=0; j < tmp; j++) {
792                q_loc[j] = op_tmp[j];
793            }
794        }
795
796        if (tmp > 0) {
797            for (j=0; j < tmp; j++) {
798                op_tmp[j] = 0;
799            }
800        }
801
802        if (id_proc == 0) {
803            for (j=0; j < tmp; j++) {
804                q_loc[j] = op_tmp[j];
805            }
806        }
807
808        if (tmp > 0) {
809            for (j=0; j < tmp; j++) {
810                op_tmp[j] = 0;
811            }
812        }
813
814        if (id_proc == 0) {
815            for (j=0; j < tmp; j++) {
816                q_loc[j] = op_tmp[j];
817            }
818        }
819
820        if (tmp > 0) {
821            for (j=0; j < tmp; j++) {
822                op_tmp[j] = 0;
823            }
824        }
825
826        if (id_proc == 0) {
827            for (j=0; j < tmp; j++) {
828                q_loc[j] = op_tmp[j];
829            }
830        }
831
832        if (tmp > 0) {
833            for (j=0; j < tmp; j++) {
834                op_tmp[j] = 0;
835            }
836        }
837
838        if (id_proc == 0) {
839            for (j=0; j < tmp; j++) {
840                q_loc[j] = op_tmp[j];
841            }
842        }
843
844        if (tmp > 0) {
845            for (j=0; j < tmp; j++) {
846                op_tmp[j] = 0;
847            }
848        }
849
850        if (id_proc == 0) {
851            for (j=0; j < tmp; j++) {
852                q_loc[j] = op_tmp[j];
853            }
854        }
855
856        if (tmp > 0) {
857            for (j=0; j < tmp; j++) {
858                op_tmp[j] = 0;
859            }
860        }
861
862        if (id_proc == 0) {
863            for (j=0; j < tmp; j++) {
864                q_loc[j] = op_tmp[j];
865            }
866        }
867
868        if (tmp > 0) {
869            for (j=0; j < tmp; j++) {
870                op_tmp[j] = 0;
871            }
872        }
873
874        if (id_proc == 0) {
875            for (j=0; j < tmp; j++) {
876                q_loc[j] = op_tmp[j];
877            }
878        }
879
880        if (tmp > 0) {
881            for (j=0; j < tmp; j++) {
882                op_tmp[j] = 0;
883            }
884        }
885
886        if (id_proc == 0) {
887            for (j=0; j < tmp; j++) {
888                q_loc[j] = op_tmp[j];
889            }
890        }
891
892        if (tmp > 0) {
893            for (j=0; j < tmp; j++) {
894                op_tmp[j] = 0;
895            }
896        }
897
898        if (id_proc == 0) {
899            for (j=0; j < tmp; j++) {
900                q_loc[j] = op_tmp[j];
901            }
902        }
903
904        if (tmp > 0) {
905            for (j=0; j < tmp; j++) {
906                op_tmp[j] = 0;
907            }
908        }
909
910        if (id_proc == 0) {
911            for (j=0; j < tmp; j++) {
912                q_loc[j] = op_tmp[j];
913            }
914        }
915
916        if (tmp > 0) {
917            for (j=0; j < tmp; j++) {
918                op_tmp[j] = 0;
919            }
920        }
921
922        if (id_proc == 0) {
923            for (j=0; j < tmp; j++) {
924                q_loc[j] = op_tmp[j];
925            }
926        }
927
928        if (tmp > 0) {
929            for (j=0; j < tmp; j++) {
930                op_tmp[j] = 0;
931            }
932        }
933
934        if (id_proc == 0) {
935            for (j=0; j < tmp; j++) {
936                q_loc[j] = op_tmp[j];
937            }
938        }
939
940        if (tmp > 0) {
941            for (j=0; j < tmp; j++) {
942                op_tmp[j] = 0;
943            }
944        }
945
946        if (id_proc == 0) {
947            for (j=0; j < tmp; j++) {
948                q_loc[j] = op_tmp[j];
949            }
950        }
951
952        if (tmp > 0) {
953            for (j=0; j < tmp; j++) {
954                op_tmp[j] = 0;
955            }
956        }
957
958        if (id_proc == 0) {
959            for (j=0; j < tmp; j++) {
960                q_loc[j] = op_tmp[j];
961            }
962        }
963
964        if (tmp > 0) {
965            for (j=0; j < tmp; j++) {
966                op_tmp[j] = 0;
967            }
968        }
969
970        if (id_proc == 0) {
971            for (j=0; j < tmp; j++) {
972                q_loc[j] = op_tmp[j];
973            }
974        }
975
976        if (tmp > 0) {
977            for (j=0; j < tmp; j++) {
978                op_tmp[j] = 0;
979            }
980        }
981
982        if (id_proc == 0) {
983            for (j=0; j < tmp; j++) {
984                q_loc[j] = op_tmp[j];
985            }
986        }
987
988        if (tmp > 0) {
989            for (j=0; j < tmp; j++) {
990                op_tmp[j] = 0;
991            }
992        }
993
994        if (id_proc == 0) {
995            for (j=0; j < tmp; j++) {
996                q_loc[j] = op_tmp[j];
997            }
998        }
999
1000        if (tmp > 0) {
1001            for (j=0; j < tmp; j++) {
1002                op_tmp[j] = 0;
1003            }
1004        }
1005
1006        if (id_proc == 0) {
1007            for (j=0; j < tmp; j++) {
1008                q_loc[j] = op_tmp[j];
1009            }
1010        }
1011
1012        if (tmp > 0) {
1013            for (j=0; j < tmp; j++) {
1014                op_tmp[j] = 0;
1015            }
1016        }
1017
1018        if (id_proc == 0) {
1019            for (j=0; j < tmp; j++) {
1020                q_loc[j] = op_tmp[j];
1021            }
1022        }
1023
1024        if (tmp > 0) {
1025            for (j=0; j < tmp; j++) {
1026                op_tmp[j] = 0;
1027            }
1028        }
1029
1030        if (id_proc == 0) {
1031            for (j=0; j < tmp; j++) {
1032                q_loc[j] = op_tmp[j];
1033            }
1034        }
1035
1036        if (tmp > 0) {
1037            for (j=0; j < tmp; j++) {
1038                op_tmp[j] = 0;
1039            }
1040        }
1041
1042        if (id_proc == 0) {
1043            for (j=0; j < tmp; j++) {
1044                q_loc[j] = op_tmp[j];
1045            }
1046        }
1047
1048        if (tmp > 0) {
1049            for (j=0; j < tmp; j++) {
1050                op_tmp[j] = 0;
1051            }
1052        }
1053
1054        if (id_proc == 0) {
1055            for (j=0; j < tmp; j++) {
1056                q_loc[j] = op_tmp[j];
1057            }
1058        }
1059
1060        if (tmp > 0) {
1061            for (j=0; j < tmp; j++) {
1062                op_tmp[j] = 0;
1063            }
1064        }
1065
1066        if (id_proc == 0) {
1067            for (j=0; j < tmp; j++) {
1068                q_loc[j] = op_tmp[j];
1069            }
1070        }
1071
1072        if (tmp > 0) {
1073            for (j=0; j < tmp; j++) {
1074                op_tmp[j] = 0;
1075            }
1076        }
1077
1078        if (id_proc == 0) {
1079            for (j=0; j < tmp; j++) {
1080                q_loc[j] = op_tmp[j];
1081            }
1082        }
1083
1084        if (tmp > 0) {
1085            for (j=0; j < tmp; j++) {
1086                op_tmp[j] = 0;
1087            }
1088        }
1089
1090        if (id_proc == 0) {
1091            for (j=0; j < tmp; j++) {
1092                q_loc[j] = op_tmp[j];
1093            }
1094        }
1095
1096        if (tmp > 0) {
1097            for (j=0; j < tmp; j++) {
1098                op_tmp[j] = 0;
1099            }
1100        }
1101
1102        if (id_proc == 0) {
1103            for (j=0; j < tmp; j++) {
1104                q_loc[j] = op_tmp[j];
1105            }
1106        }
1107
1108        if (tmp > 0) {
1109            for (j=0; j < tmp; j++) {
1110                op_tmp[j] = 0;
1111            }
1112        }
1113
1114        if (id_proc == 0) {
1115            for (j=0; j < tmp; j++) {
1116                q_loc[j] = op_tmp[j];
1117            }
1118        }
1119
1120        if (tmp > 0) {
1121            for (j=0; j < tmp; j++) {
1122                op_tmp[j] = 0;
1123            }
1124        }
1125
1126        if (id_proc == 0) {
1127            for (j=0; j < tmp; j++) {
1128                q_loc[j] = op_tmp[j];
1129            }
1130        }
1131
1132        if (tmp > 0) {
1133            for (j=0; j < tmp; j++) {
1134                op_tmp[j] = 0;
1135            }
1136        }
1137
1138        if (id_proc == 0) {
1139            for (j=0; j < tmp; j++) {
1140                q_loc[j] = op_tmp[j];
1141            }
1142        }
1143
1144        if (tmp > 0) {
1145            for (j=0; j < tmp; j++) {
1146                op_tmp[j] = 0;
1147            }
1148        }
1149
1150        if (id_proc == 0) {
1151            for (j=0; j < tmp; j++) {
1152                q_loc[j] = op_tmp[j];
1153            }
1154        }
1155
1156        if (tmp > 0) {
1157            for (j=0; j < tmp; j++) {
1158                op_tmp[j] = 0;
1159            }
1160        }
1161
1162        if (id_proc == 0) {
1163            for (j=0; j < tmp; j++) {
1164                q_loc[j] = op_tmp[j];
1165            }
1166        }
1167
1168        if (tmp > 0) {
1169            for (j=0; j < tmp; j++) {
1170                op_tmp[j] = 0;
1171            }
1172        }
1173
1174        if (id_proc == 0) {
1175            for (j=0; j < tmp; j++) {
1176                q_loc[j] = op_tmp[j];
1177            }
1178        }
1179
1180        if (tmp > 0) {
1181            for (j=0; j < tmp; j++) {
1182                op_tmp[j] = 0;
1183            }
1184        }
1185
1186        if (id_proc == 0) {
1187            for (j=0; j < tmp; j++) {
1188                q_loc[j] = op_tmp[j];
1189            }
1190        }
1191
1192        if (tmp > 0) {
1193            for (j=0; j < tmp; j++) {
1194                op_tmp[j] = 0;
1195            }
1196        }
1197
1198        if (id_proc == 0) {
1199            for (j=0; j < tmp; j++) {
1200                q_loc[j] = op_tmp[j];
1201            }
1202        }
1203
1204        if (tmp > 0) {
1205            for (j=0; j < tmp; j++) {
1206                op_tmp[j] = 0;
1207            }
1208        }
1209
1210        if (id_proc == 0) {
1211            for (j=0; j < tmp; j++) {
1212                q_loc[j] = op_tmp[j];
1213            }
1214        }
1215
1216        if (tmp > 0) {
1217            for (j=0; j < tmp; j++) {
1218                op_tmp[j] = 0;
1219            }
1220        }
1221
1222        if (id_proc == 0) {
1223            for (j=0; j < tmp; j++) {
1224                q_loc[j] = op_tmp[j];
1225            }
1226        }
1227
1228        if (tmp > 0) {
1229            for (j=0; j < tmp; j++) {
1230                op_tmp[j] = 0;
1231            }
1232        }
1233
1234        if (id_proc == 0) {
1235            for (j=0; j < tmp; j++) {
1236                q_loc[j] = op_tmp[j];
1237            }
1238        }
1239
1240        if (tmp > 0) {
1241            for (j=0; j < tmp;
```

```

35     switch(test) {
36         case NO_TEST: {
37             if (q_num <= 20) {
38                 for (j=0; j < tmp; j++) {
39                     op_tmp[j] = argToDouble(argv[j+5]);
40                 }
41             } else {
42
43                 for (j=0; j < tmp; j++) {
44                     double_rand = (double)rand();
45                     int_rand = (int)rand();
46
47                     // Si genera un numero casuale reale compreso tra
48                     // 0 e 100
49                     op_tmp[j] = (double_rand / RAND_MAX) *
50                     OP_MAX_VALUE;
51
52                     // Si ha il 33% di possibilita che op_tmp[j] < 0
53                     if (int_rand % 3 == 0) {
54                         op_tmp[j] = op_tmp[j] * (-1);
55                     }
56                 }
57             }
58             break;
59         }
60         case SUM_ONE_TEST: {
61             for (j=0; j < tmp; j++) {
62
63                 /*
64                  Il vettore degli operandi e' costituito di soli
65                  1.
66
67                  Il test termina con successo se la somma totale
68                  calcolata dai processori e' pari a 'q_num'.
69
70                 op_tmp[j] = 1;
71             }
72             break;

```

```

71 }
72 case SUM_SINGLE_NUMBER_TEST: {
73     for (j=0; j < tmp; j++) {
74
75     /*
76         Ad ogni posizione del vettore degli operandi e'
77         assegnato lo stesso valore reale, passato come
78         argomento 'argv[5]' al programma.
79
80         Il test termina con successo se la somma totale
81         calcolata dai processori e' pari a 'argv[5] *
82         q_num'.
83     */
84     op_tmp[j] = argToDouble(argv[5]);
85 }
86 break;
87 }
88 case SUM_OPPOSITE_NUMBER_TEST: {
89     for (j=0; j < tmp/2; j++) {
90
91     /*
92         Si genera un numero casuale reale compreso tra 0
93         e 100.
94
95         Alle posizioni nella prima meta' del vettore
96             degli
97             operandi si assegna il valore reale generato.
98
99         Alle posizioni nella seconda meta' del vettore
100             degli
101             operandi si assegna l'opposto del valore reale
102             generato.
103
104         Il test termina con successo se la somma totale
105             calcolata
106             dai processori e' pari a 0 (gli operandi si
107             annullano).
108     */
109 }
```

```

103
104     double_rand = (double)rand();
105     double_rand = (double_rand / RAND_MAX) *
106         OP_MAX_VALUE;
107
108     op_tmp[j] = double_rand;
109     op_tmp[j+tmp/2] = double_rand * (-1);
110
111     }
112 }
113 case GAUSS_TEST: {
114     gauss_inf = arg.ToDouble(argv[5]);
115     for (j=0; j < tmp; j++) {
116
117     /*
118         Il vettore degli operandi e' costituito da tutti
119         i
120         numeri interi racchiusi nell'intervallo
121         [gauss_inf, gauss_inf + q_num].
122
123         Il test termina con successo se la somma totale
124         calcolata dai processori si puo' ottenere con la
125         formula di Gauss, cioe' il prodotto tra media
126         aritmetica e numero di operandi da sommare.
127
128         */
129     op_tmp[j] = gauss_inf + gauss_tmp;
130     gauss_tmp++;
131     }
132     break;
133 }
134 default:
135     break;
136 }
137 if (i == 0) {
138
139     // Si assegnano gli operandi locali del processore con

```

```

140         id_proc == 0
141         for (j=0; j < tmp; j++) {
142             op_loc[j] = op_tmp[j];
143         }
144     } else {
145
146         /*
147             Si distribuiscono equamente gli operandi rimanenti a
148             tutti gli altri processori utilizzando la funzione MPI_Send
149             ().
150
151         tag = i + DISTRIBUTION_TAG;
152
153         MPI_Send(&op_tmp[0], tmp, MPI_DOUBLE, i, tag,
154             MPI_COMM_WORLD);
155
156         free(op_tmp);
157     }
158
159 } else {
160     tag = id_proc + DISTRIBUTION_TAG;
161
162     MPI_Recv(op_loc, q_loc, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD,
163             &status);
164 }
```

Listing A.17: Distribuzione degli operandi

```

1 if (time_calc == OK_TIME_CALC) {
2     MPI_Barrier(MPI_COMM_WORLD);
3     t_start = MPI_Wtime();
4 }
```

Listing A.18: Inizio del calcolo dei tempi di esecuzione

```

1  sum = 0.0;
2  for(i=0; i < q_loc; i++) {
3      sum = sum + op_loc[i];
4 }
```

Listing A.19: Calcolo della somma parziale

```

1  switch (strategia) {
2      case NO_STRATEGY: {
3          break;
4      }
5      case FIRST_STRATEGY: // Applicazione della strategia 1.
6      case SECOND_STRATEGY: // Applicazione della strategia 2.
7      case THIRD_STRATEGY: // Applicazione della strategia 3.
8      default:
9      {
10         printf("Comando non riconosciuto!\n");
11         break;
12     }
13 }
```

Listing A.20: Switch-case per le strategie da applicare

```

1  if (id_proc == 0) {
2      for(i=1; i < n_proc; i++) {
3          tag = i + FIRST_STRATEGY_TAG;
4          MPI_Recv(&sum_parz, 1, MPI_DOUBLE, i, tag,
5                  MPI_COMM_WORLD, &status);
6          sum = sum + sum_parz;
7      }
8  } else {
9      tag = id_proc + FIRST_STRATEGY_TAG;
10     MPI_Send(&sum, 1, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD);
11 }
```

Listing A.21: Applicazione della strategia 1

```

1  /*
2      Nella 2a strategia, il numero di passi di comunicazione
3      e' dato dal logaritmo in base 2 del numero di
4          processori,
5          gia' calcolato precedentemente.
6  */
```

```

6
7     for(i=0; i < log_proc; i++) {
8
9     /*
10      Si calcola l'identificativo di chi deve partecipare
11      alle comunicazioni all'i-esimo passo.
12    */
13
14     pow_proc = pow(2, i);
15
16     if ((id_proc % pow_proc) == 0) {
17
18     /*
19      Si calcola l'identificativo di chi deve ricevere
20      dalle comunicazioni dell'i-esimo passo.
21    */
22
23     pow_tmp = pow(2, i+1);
24
25     if ((id_proc % pow_tmp) == 0) {
26
27     /*
28      Questi processori hanno il compito di ricevere la
29      somma parziale da chi esegue la MPI_Send().
30      In questo caso, sia 'tag' che 'tmp' si
31      riferiscono
32      allo stesso processore (chi invia).
33    */
34
35     tmp = id_proc + pow(2, i);
36     tag = tmp + SECOND_STRATEGY_TAG;
37     MPI_Recv(&sum_parz, 1, MPI_DOUBLE, tmp, tag,
38              MPI_COMM_WORLD, &status);
39     sum = sum + sum_parz;
40
41   /*
42     I processori rimanenti avranno il solo compito di

```

```

43         inviare la propria somma parziale. In quest'altro
44         caso, 'tag' e 'tmp' hanno valori diversi:
45         - tag: si riferisce a se' stesso (chi invia);
46         - tmp: si riferisce al ricevente (chi riceve).
47     */
48
49     tmp = id_proc - pow(2, i);
50     tag = id_proc + SECOND_STRATEGY_TAG;
51     MPI_Send(&sum, 1, MPI_DOUBLE, tmp, tag,
52             MPI_COMM_WORLD);
53 }
54 }
55 }
```

Listing A.22: Applicazione della strategia 2

```

1 /*
2     Nella 3a strategia, il numero di passi di comunicazione
3     e' lo stesso della 2a strategia, ma al termine dell'
4     esecuzione
5     tutti i processori saranno a conoscenza della somma
6     totale.
7 */
8
9 for(i=0; i < log_proc; i++) {
10
11     /*
12         Rispetto alla 2a strategia, tutti i processori
13             partecipano
14             ad ogni passo di comunicazione. Operativamente, si
15             rimuove
16             il primo controllo '(id_proc % pow_tmp) == 0'.
17
18         Quindi, tutti i processori hanno il compito di
19             inviare le
20             proprie somme parziali e, al contempo, ricevere le
21             somme
22             parziali dagli altri processori in modo tale che
23             tutti
24             calcolino localmente la somma totale.
```

```

18
19     Operativamente , le operazioni di MPI_Send() e
20         MPI_Recv()
21     sono in comune a tutti i processori , cambia solo
22         l'identificativo di chi ha inviato / riceve.
23     */
24
25     pow_proc = pow(2 , i+1);
26     pow_tmp = pow(2 , i);
27
28     if ((id_proc % pow_proc) < pow_tmp) {
29         tmp = id_proc + pow_tmp;
30     } else {
31         tmp = id_proc - pow_tmp;
32     }
33
34     tag = id_proc + THIRD_STRATEGY_TAG;
35     MPI_Send(&sum , 1 , MPI_DOUBLE , tmp , tag , MPI_COMM_WORLD)
36         ;
37
38     tag = tmp + THIRD_STRATEGY_TAG;
39     MPI_Recv(&sum_parz , 1 , MPI_DOUBLE , tmp , tag ,
40             MPI_COMM_WORLD , &status);
41     sum = sum + sum_parz;
42 }

```

Listing A.23: Applicazione della strategia 3

```

1 if (time_calc == OK_TIME_CALC) {
2     t_end = MPI_Wtime();
3
4     // Si calcola la distanza di tempo tra l'istante iniziale e
5     // quello finale.
6     t_loc = t_end - t_start;
7
8     MPI_Reduce(&t_loc , &t_tot , 1 , MPI_DOUBLE , MPI_MAX , 0 ,
9                 MPI_COMM_WORLD);
10
11    if (id_proc == 0) {
12        printf("\nApplicazione della strategia %d terminata in %e
13              sec\n" , strategia , t_tot);

```

```

11     writeTimeCSV(test, strategia, n_proc, q_num, t_tot);
12 }
13 }
```

Listing A.24: Salvataggio del calcolo dei tempi di esecuzione

```

1  switch (strategia) {
2      case NO_STRATEGY:
3      case FIRST_STRATEGY:
4      {
5          if (id_proc == 0) {
6              printf("\nLa somma totale e' %f\n", sum);
7          }
8          break;
9      }
10     case SECOND_STRATEGY:
11     case THIRD_STRATEGY:
12     {
13         printf("\nProcesso n.%d\n", id_proc);
14         printf("La somma totale e' %f\n", sum);
15         break;
16     }
17     default:
18         printf("Errore nella stampa dell'output!\n");
19         break;
20 }
```

Listing A.25: Stampa dell'output

```

1  /*
2      Attendiamo che tutti i processori abbiano portato a termine
3      correttamente il loro carico di lavoro.
4  */
5
6  MPI_Barrier(MPI_COMM_WORLD);
7
8  // Al termine dell'esecuzione, si libera lo spazio allocato
9  // in memoria.
10 free(op_loc);
11
12 if (id_proc == 0) {
13     printf("\nEsecuzione terminata.\n");
```

```
13 }
14
15 MPI_Finalize();
16 return 0;
```

Listing A.26: Terminazione dell'esecuzione

# Sitografia

- [1] Open MPI, *La libreria mpi.h (v. 1.4.5)*, <https://www.open-mpi.org/doc/v1.4/>.
- [2] C++ Reference, *La libreria stdio.h*, <https://cplusplus.com/reference/cstdio>.
- [3] Carlos III University Of Madrid, *La funzione getline*, [https://www.it.uc3m.es/pbasanta/asng/course\\_notes/input\\_output\\_getline\\_en.html](https://www.it.uc3m.es/pbasanta/asng/course_notes/input_output_getline_en.html).
- [4] C++ Reference, *La libreria stdlib.h*, <https://cplusplus.com/reference/cstdlib>.
- [5] C++ Reference, *La libreria errno.h*, <https://cplusplus.com/reference/cerrno>.
- [6] C++ Reference, *La libreria limits.h*, <https://cplusplus.com/reference/climits>.
- [7] C++ Reference, *La libreria string.h*, <https://cplusplus.com/reference/cstring>.
- [8] C++ Reference, *La libreria tgmath.h*,  
<https://cplusplus.com/reference/ctgmath>.
- [9] C++ Reference, *La libreria time.h*, <https://cplusplus.com/reference/ctime>.
- [10] C++ Reference, *La libreria ctype.h*, <https://cplusplus.com/reference/cctype>.