

UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

PARALLEL AND DISTRIBUTED COMPUTING

PARALLEL SUM DOCUMENTATION

2022/2023

<i>Professori</i>	<i>Studente</i>
Giuliano Laccetti	Vincenzo Tramo
Anna Mele	N86002592

Indice

Definizione e Analisi del problema	3
1 Definizione e Analisi del problema	3
1.1 Che cosa fa il software	3
1.2 Codice sorgente	4
1.3 Come eseguire il software	4
1.4 La struttura del progetto	4
1.5 Il file <code>main.c</code>	4
1.5.1 Precondizioni del programma <code>main.c</code>	5
1.5.2 Estrarre i numeri dal file fornito in input al programma <code>main</code>	6
1.5.3 Generazione di numeri casuali	6
1.5.4 Codice programma <code>main.c</code>	6
Descrizione dell'algoritmo	9
2 Descrizione dell'algoritmo	9
2.1 Distribuzione dei numeri tra i processi	9
2.2 Calcolo della somma locale	10
2.3 Scelta della strategia da utilizzare	10
2.3.1 Strategia uno	11
2.3.2 Strategia due	12
2.3.3 Strategia tre	13
2.4 Stampa del risultato	14
2.5 I file <code>.pbs</code>	14
2.5.1 Il file <code>run.pbs</code>	15
2.5.2 Il file <code>performance_test.pbs</code>	16
Testing	17
3 Testing	17
3.1 Black Box Testing	17
3.2 Performance Testing	19
3.2.1 Grafici strategia uno	20
3.2.2 Grafici seconda strategia	22
3.2.3 Grafici terza strategia	23

Definizione e Analisi del problema

1 Definizione e Analisi del problema

Il seguente elaborato ha l'obiettivo di documentare l'implementazione dell'algoritmo per il calcolo della somma di N numeri reali in ambiente di calcolo parallelo su architettura MIMD (Multiple Instruction Multiple Data) a memoria distribuita utilizzando la libreria OpenMPI (<https://www.open-mpi.org/>). L'obiettivo del seguente documento è anche valutare le performance dell'algoritmo confrontando le varie strategie come spiegato in seguito.

1.1 Che cosa fa il software

Il software offre tre diverse implementazioni della somma parallela (denotate come strategia uno, strategia due e strategia tre) ognuna delle quali si differenzia dalle altre per come i processi calcolano e comunicano tra di loro le somme parziali. Essenzialmente il software che si vuole documentare partiziona un vettore di N numeri e ogni sottovettore della partizione viene assegnato ad un processo che ne calcola la somma parziale e, in base alla strategia utilizzata, invia la somma parziale ad altri processi e/o riceve la somma parziale calcolata da qualche processo. Alla fine le somme parziali vengono "unite" per ottenere la somma finale (anche questa fase dipende dalla strategia adottata).

L'utente intenzionato ad utilizzare il software può specificare come input:

- il numero di processi (minimo 1 massimo 8)
- il numero N di numeri da sommare
- i numeri da sommare (manualmente se $N \leq 20$ o random se $N > 20$)
- la strategia da utilizzare (strategia uno, due o tre)
- il pid del processo che deve stampare il risultato oppure il numero -1 per far stampare il risultato a tutti i processi

L'output del programma sarà:

- il risultato della somma totale o/e le somme parziali (in base all'input scelto)
- il tempo totale impiegato per risolvere il problema (cioè il tempo calcolato dall'ultimo processo terminato)

Un esempio di utilizzo è il seguente:

Codice 1: esempio d'utilizzo del software

PARALLEL SUM

Enter the number of processors (min 1 - max 8): 2

Enter the number N of numbers to add: 21

21 random numbers have just been generated

TOTAL CPU: 2

TOTAL NUMBERS: 21

TOTAL SUM: 25

Enter the strategy to use (1, 2 or 3): 1

Enter the pid that must print the result (0 to 1 or -1 if everyone has to print the result): -1

EXECUTING PARALLEL SUM

[PID 0] Result: 25

[TOTAL TIME] 0.000024 seconds.

[PID 1] Result: 963

1.2 Codice sorgente

Il software è scritto con il linguaggio C utilizzando la libreria Open MPI `mpi.h`. Il codice sorgente può essere trovato su GitHub (<https://github.com/vtramo/parallel-and-distributed-computing-experiments>). La repository su GitHub ha un branch `main` in cui il codice è scritto per essere eseguito in locale, e un branch `cluster` in cui il codice è scritto per essere eseguito sul cluster SCOPE. Tuttavia saranno presentati frammenti di codice in questo documento.

1.3 Come eseguire il software

L'esempio di utilizzo presentato nel frammento di Codice 1 è l'output di uno script bash che invoca il programma `main.c` in maniera opportuna dopo aver acquisito gli input necessari per la sua esecuzione. Per eseguire e provare il software basta recarsi nella cartella `parallel-sum/bin/run.sh` ed eseguire il comando:

```
1 ./run.sh
```

1.4 La struttura del progetto

La struttura del progetto è la seguente:

```
bin
  run.sh
build
  iterativesum
  main
  numbers
  random
include
  parallel_sum.h
  preconditions.h
src
  iterative_sum.c
  main.c
  parallel_sum.c
  preconditions.c
  random_integer_generator.c
tests
  performance_testing
    performance_test.sh
    performance.txt
  test_global.sh
  test_strategy_one.sh
  test_strategy_three.sh
  test_strategy_two.sh
```

- La cartella `bin` contiene i programmi principali per eseguire il software correttamente
- La cartella `build` contiene i programmi compilati
- La cartella `include` contiene le librerie create
- La cartella `src` contiene il codice sorgente
- La cartella `tests` contiene dei test per aumentare l'affidabilità del software e contiene dei test per misurare le performance delle varie strategie (incluso l'algoritmo sequenziale della somma)

1.5 Il file `main.c`

L'orchestratore del software è il file `main.c`. Per eseguire il software correttamente senza utilizzare lo script `run.sh` bisogna compilare il file `main.c` con il compilatore `mpicc` specificando il numero di processi ed eseguire l'artefatto così generato. Il programma `main` ha bisogno di tre parametri:

- **la strategia da utilizzare:** un numero compreso tra 1 e 3

- **il PID radice:** il PID del processo che deve stampare il risultato oppure -1 se tutti devono stampare il risultato
- **un file contenente i numeri da sommare:** prima di avviare il programma `main` bisogna costruire un file contenente una qualsiasi quantità di numeri separati da un qualsiasi carattere

Un corretta esecuzione del programma `main` (assumendo che il programma generato compilando il file `main.c` si chiami `main`) deve rispettare questo schema:

```
1 mpirun -np <n_cpu> main <strategy_id> <root_pid> <file_path_numbers>
```

1.5.1 Precondizioni del programma `main.c`

Prima di eseguire la somma parallela, il programma `main.c` controlla a runtime se le seguenti *precondizioni* siano rispettate:

- il numero di argomenti deve essere uguale 4
- il primo parametro `strategy_id` e il secondo parametro `root_pid` devono essere numeri
- il terzo parametro `file_path_numbers` deve essere un percorso di un file esistente e deve contenere dei numeri
- il primo parametro `strategy_id` deve essere un numero uguale a 1, 2 o 3
- se il primo parametro `strategy_id` è un numero uguale a 2 o 3, allora il numero totale di processi deve essere una potenza di 2
- il parametro `root_pid` deve essere un numero compreso tra 0 e il numero totale di processi - 1
- il numero totale di processi deve essere minore o uguale del numero totale di numeri da sommare

La violazione di una qualsiasi di queste precondizioni comporta l'interruzione del programma con un opportuno messaggio di errore che descrive il problema. Ad esempio, se eseguiamo uno dei test contenuti in `parallel-sum/tests/test_strategy_one.sh` che tenta di eseguire il programma `main` fornendo in input una strategia inesistente, il programma risponde correttamente con un messaggio di errore:

```
| TEST #3 |

Description: When passing an invalid id strategy should print
an error on stdout.

INPUT

- Number of processes: 4
- Strategy: -2
- Root PID: 1
- Numbers: 5 5 5 5

EXPECTED OUTPUT:

Correct usage: ../build/main <strategy_id> <root_pid> <file_path_numbers>
There is no strategy with id -2!

TEST OK!
```

Figura 1: Un test case per la strategia uno

Le precondizioni vengono controllate in un file separato per aumentare la leggibilità e la manutenibilità del codice. L'implementazione può essere trovata in `src/preconditions.c` mentre le definizioni in `include/preconditions.h`.

1.5.2 Estrarre i numeri dal file fornito in input al programma main

Per estrarre i numeri dal file fornito in input al programma `main` è stata realizzata una funzione utilitaria che ritorna un tipo di dato `Numbers*` il quale contiene un vettore di `int* numbers` e la sua dimensione `unsigned int size`:

```
typedef struct {
    unsigned int size;
    int* numbers;
} Numbers;

Numbers* read_numbers_from_file(char *file_name) {
    FILE *file = fopen(file_name, "r");
    int n = 0;
    int total_numbers = 0;
    int index = 0;
    int dim = 1;
    int *numbers = malloc(sizeof(int));
    while (fscanf(file, "%d", &n) > 0) {
        total_numbers++;
        numbers[index++] = n;
        numbers = realloc(numbers, sizeof(int) * ++dim);
    }
    fclose(file);
    Numbers *bean_numbers = malloc(sizeof(Numbers));
    bean_numbers->size = total_numbers;
    bean_numbers->numbers = numbers;
    return bean_numbers;
}
```

La funzione alloca inizialmente un vettore di `int*` di dimensione 1, apre e legge il file che contiene i numeri in sola lettura e per ogni numero n trovato, dopo essere stato inserito nel vettore, viene incrementata la dimensione del vettore di uno spazio pari a `sizeof(int)`. La funzione può essere trovata in `src/parallel_sum.c` mentre la sua definizione in `include/parallel_sum.h`.

1.5.3 Generazione di numeri casuali

Il software ha bisogno di generare numeri casuali non solo quando la dimensione N è maggiore di 20 ma anche per testare le performance su input di grandi dimensioni. Si è preferito realizzare un programma `random_integer_generator.c` per facilitare il suo riutilizzo in altre parti del software. Il programma può essere trovato nella cartella `src`.

1.5.4 Codice programma main.c

Di seguito viene mostrato il programma `main.c`. Nelle sezioni a seguire viene dettagliata ogni riga di codice laddove questa non risulti essere già chiara di per sé.

```
#include "../include/preconditions.h"
#include "../include/parallel_sum.h"

typedef struct {
    unsigned int start;
    unsigned int end;
} RangeExtremes;

RangeExtremes* computes_range_extremes(
    unsigned int total_number_of_processes,
    unsigned int total_numbers,
    unsigned int this_pid
);

int main(int argc, char **argv) {
    check_number_parameters(argc, argv);
```

```

check_parameters(argv);

int tmp_root_pid = atoi(argv[2]);
const int root_pid = (tmp_root_pid == -1 ? 0 : tmp_root_pid);
const bool everyone_must_print_the_result = (tmp_root_pid == -1);

MPI_Init(&argc, &argv);

int total_number_of_processes;
MPI_Comm_size(MPI_COMM_WORLD, &total_number_of_processes);
check_root_pid(root_pid, total_number_of_processes);

char *file_path_numbers = argv[3];
const Numbers *numbers_beau = read_numbers_from_file(file_path_numbers);
const int total_numbers = numbers_beau->size;
const int *numbers = numbers_beau->numbers;
check_total_number_of_processes_less_than_or_equal_to_the_total_numbers(
    total_number_of_processes,
    total_numbers
);

const int strategy_id = atoi(argv[1]);
check_strategy_id(strategy_id, total_number_of_processes, argv[0]);

int this_pid;
MPI_Comm_rank(MPI_COMM_WORLD, &this_pid);

RangeExtremes *range_extremes = computes_range_extremes(
    total_number_of_processes,
    total_numbers,
    this_pid
);
const int start = range_extremes->start;
const int end = range_extremes->end;
free(range_extremes);

const double t0 = MPI_Wtime();

int sum = computes_local_sum(numbers, start, end);

switch (strategy_id) {
    case STRATEGY_ONE:
        computes_strategy_one(&sum, this_pid, root_pid, total_number_of_processes);
        break;
    case STRATEGY_TWO:
        computes_strategy_two(&sum, this_pid, total_number_of_processes);
        break;
    case STRATEGY_THREE:
        computes_strategy_three(&sum, this_pid, total_number_of_processes);
        break;
    default:
        return EXIT_FAILURE;
}

const double t1 = MPI_Wtime();

if (everyone_must_print_the_result || this_pid == root_pid) {
    printf("[PID %d] Result: %d\n", this_pid, sum);
}

double total_time = t1 - t0;

```

```

    double max_total_time;

    MPI_Reduce(&total_time, &max_total_time, 1, MPI_DOUBLE, MPI_MAX, root_pid, MPI_COMM_WORLD);

    if (this_pid == root_pid) {
        printf("[TOTAL TIME] %f seconds.\n", max_total_time);
    }

    MPI_Finalize();

    return EXIT_SUCCESS;
}

RangeExtremes* computes_range_extremes(
    unsigned int total_number_of_processes,
    unsigned int total_numbers,
    unsigned int this_pid
) {
    RangeExtremes *range_extremes = malloc(sizeof(RangeExtremes));

    const int number_of_locations = total_numbers / total_number_of_processes;
    const int rest = total_numbers % total_number_of_processes;

    int start = this_pid * number_of_locations;
    int end = start + number_of_locations;

    int rest_offset_start = 0, rest_offset_end = 0;
    if (rest != 0) {
        if (this_pid == 0) {
            rest_offset_end = 1;
        } else if (this_pid <= rest) {
            rest_offset_start = this_pid;
            rest_offset_end = this_pid + ((this_pid < rest) ? 1 : 0);
        } else {
            rest_offset_start = rest_offset_end = rest;
        }
    }

    range_extremes->start = start + rest_offset_start;
    range_extremes->end = end + rest_offset_end;
    return range_extremes;
}

```

PARALLEL SUM

Enter the number of processors (min 1 - max 8): 2

Enter the number N of numbers to add: 21

21 random numbers have just been generated

TOTAL CPU: 2

TOTAL NUMBERS: 21

TOTAL SUM: 25

Enter the strategy to use (1, 2 or 3): 1

Enter the pid that must print the result (0 to 1 or -1 if everyone has to print the result): -1

EXECUTING PARALLEL SUM

[PID 0] Result: 25

[TOTAL TIME] 0.000024 seconds.

[PID 1] Result: 963

Descrizione dell'algoritmo

2 Descrizione dell'algoritmo

2.1 Distribuzione dei numeri tra i processi

Una volta controllate le precondizioni, aver inizializzato l'ambiente MPI con la subroutine `MPI_INIT`, aver ottenuto il pid del processo attuale con la subroutine `MPI_Comm_rank` e aver ottenuto il numero totale di processi con la subroutine `MPI_Comm_size` (andare nella Sezione ?? per maggior informazioni sulle subroutines di MPI utilizzate) bisogna partizionare i numeri ricevuti in input in un numero di blocchi distinti pari al numero totale di processi.

Siccome è stato scelto di dare i numeri come input al programma `main` piuttosto che, ad esempio, ottenere i numeri a runtime all'interno del programma, ogni processo ha l'intero vettore di numeri da sommare. Quello che bisogna fare è, dato il numero totale di numeri da sommare, il numero totale di processi coinvolti e l'identificativo del processo, capire l'indice di inizio e fine per quel particolare processo in maniera tale che ogni processo abbia il suo intervallo di indici tale che se intersecato con tutti gli altri intervalli degli altri processi, il risultato è l'intervallo vuoto. Per fare ciò è stata realizzata una funzione ad hoc che risolve tale problema:

```
typedef struct {
    unsigned int start;
    unsigned int end;
} RangeExtremes;

RangeExtremes* computes_range_extremes(
    unsigned int total_number_of_processes,
    unsigned int total_numbers,
    unsigned int this_pid
);
```

La funzione riceve in input `total_number_of_processes`, `total_numbers` e `this_pid` e restituisce il tipo di dato `RangeExtremes*` che contiene due interi senza segno `start` e `end`. Di seguito viene riportata l'implementazione di tale funzione:

```
RangeExtremes* computes_range_extremes(
    unsigned int total_number_of_processes,
    unsigned int total_numbers,
    unsigned int this_pid
) {
    RangeExtremes *range_extremes = malloc(sizeof(RangeExtremes));

    const int number_of_locations = total_numbers / total_number_of_processes;
    const int rest = total_numbers % total_number_of_processes;

    int start = this_pid * number_of_locations;
    int end = start + number_of_locations;

    int rest_offset_start = 0, rest_offset_end = 0;
    if (rest != 0) {
        if (this_pid == 0) {
            rest_offset_end = 1;
        } else if (this_pid <= rest) {
            rest_offset_start = this_pid;
            rest_offset_end = this_pid + ((this_pid < rest) ? 1 : 0);
        } else {
            rest_offset_start = rest_offset_end = rest;
        }
    }

    range_extremes->start = start + rest_offset_start;
    range_extremes->end = end + rest_offset_end;
    return range_extremes;
}
```

Il numero di locazioni per ogni processo è calcolato prendendo la parte intera del rapporto tra il numero totale di numeri da sommare e il numero totale di processi. Se il resto di questo rapporto non è 0, bisogna gestire vari casi. Supponendo che il resto sia 0, lo **start** e l'**end** di quel processo con identificativo **this_pid** è quello calcolato nella funzione. Se il resto non è 0 allora vuol dire che alcuni numeri sono avanzati e non sono stati assegnati a nessun processo. Esistono diverse soluzioni per risolvere questo problema, ad esempio queste due:

1. assegnare i numeri avanzati all'ultimo processo allargando l'indice **end** di quel processo di una quantità pari al numero di numeri avanzati
2. rendere gli intervalli di ogni processo più bilanciati possibili e quindi assegnare ad ogni intervallo uno dei numeri avanzati fino a quando non esauriscono

Sapendo che il resto massimo di una divisione a/b è $b-1$, la soluzione uno potrebbe essere adatta nel caso in cui il numero di processi sia relativamente basso. Ma quando il numero di processi scelto è alto, potrebbe esserci un forte **sbilanciamento** e questo potrebbe incidere sulle performance dell'algoritmo parallelo. La soluzione uno rende i primi $p-1$ intervalli ugualmente bilanciati (dove p è il numero di processi) mentre rende l'ultimo intervallo p sbilanciato (anche fortemente) dal resto degli intervalli.

Contrariamente, la soluzione due si pone l'obiettivo di bilanciare il più quanto possibile gli intervalli di ogni processo e lo fa assegnando ad ogni intervallo uno dei numeri avanzati fino a quando non esauriscono. Il numero massimo di numeri avanzati è rappresentato dalla quantità **rest** che non è altro il resto della divisione per ottenere il numero di locazioni. Il resto massimo è $p-1$ quindi questa soluzione permette di raggiungere la seguente proprietà (dove $|I_i|$ è la quantità di numeri che l'intervallo I_i contiene):

$$\text{per ogni } I_i, I_j \quad \text{abs}(|I_i| - |I_j|) \leq 1 \quad i, j \in \{0, \dots, p-1\}$$

Quindi questa soluzione garantisce un bilanciamento ottimale. L'implementazione utilizza la seconda soluzione.

L'implementazione utilizza la seconda soluzione che non fa altro che:

- aumentare di uno l'**end** del processo con pid 0
- ogni processo deve traslare il proprio **start** di tot posizioni in avanti in base alle traslazioni fatte dagli altri processi prima di lui (cioè tutti quei processi con pid minore) e traslare il proprio **end** di una posizione in avanti per fare spazio ad uno dei numeri avanzati. Ad esempio, se **this_pid** è 1, allora bisogna traslare lo **start** di questo processo di uno in avanti perché il processo con pid 0 ha traslato il proprio **end** di uno in avanti. Allo stesso modo, il processo con **this_pid** uguale a 2 deve traslare di due posizioni in avanti il proprio **start** perché il processo con pid 1 ha traslato il proprio **end** di due posizioni in avanti. Se il processo ha pid uguale a **rest** non deve traslare il proprio **end** di uno in avanti perché i numeri avanzati sono esauriti
- se il processo con pid i è maggiore del resto **rest**, deve solo traslare di **rest** posizioni in avanti il proprio **start** e il proprio **end** perché arrivati a questo punto i numeri che sono avanzati sono esauriti e correttamente assegnati ai processi prima di lui e le traslazioni in avanti fatte prima di lui sono pari a **rest** quindi deve semplicemente ricalibrare il proprio intervallo

2.2 Calcolo della somma locale

Una volta ottenuti i propri indici **start** e **end** è necessario che ogni processo calcoli la propria somma in locale. Viene utilizzata la seguente semplice funzione:

```
int computes_local_sum(const int *numbers, const int start, const int end) {
    int sum = 0;
    for (int i = start; i < end; i++) {
        sum += numbers[i];
    }
    return sum;
}
```

La funzione può essere trovata in `src/parallel_sum.c`.

2.3 Scelta della strategia da utilizzare

In base al valore di **strategy_id** viene avviata una strategia:

```

switch (strategy_id) {
    case STRATEGY_ONE:
        computes_strategy_one(&sum, this_pid, root_pid, total_number_of_processes);
        break;
    case STRATEGY_TWO:
        computes_strategy_two(&sum, this_pid, total_number_of_processes);
        break;
    case STRATEGY_THREE:
        computes_strategy_three(&sum, this_pid, total_number_of_processes);
        break;
    default:
        return EXIT_FAILURE;
}

```

Le tre strategie sono state suddivise in tre funzioni diverse che possono essere trovare in `src/parallel_sum.c`:

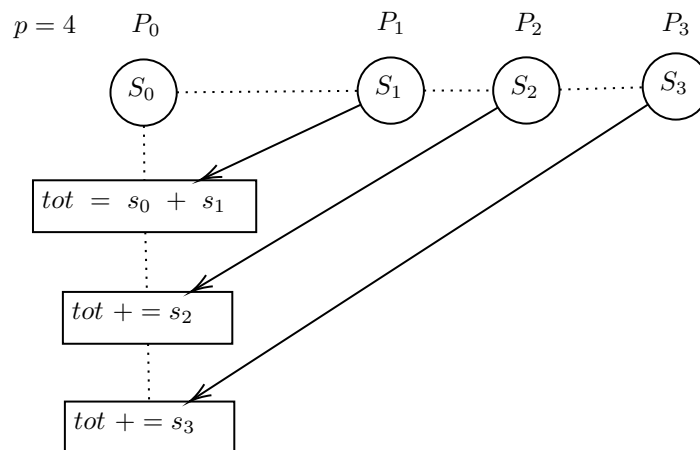
1.

```
void computes_strategy_one(
    int *sum,
    const int this_pid,
    const int root_pid,
    const int total_number_of_processes
);
```
2.

```
void computes_strategy_two(
    int *sum,
    const int this_pid,
    const int total_number_of_processes
);
```
3.

```
void computes_strategy_three(
    int *sum,
    const int this_pid,
    const int total_number_of_processes
);
```

2.3.1 Strategia uno



Ogni processore calcola la propria somma parziale (operazioni concorrenti). Ad ogni passo *ciascun processore* invia tale valore ad un unico processore prestabilito. Tale processore contiene la *somma totale*.

```

void computes_strategy_one(
    int *sum,
    const int this_pid,
    const int root_pid,
    const int total_number_of_processes
);

```

La funzione che utilizza la strategia uno ha bisogno come input:

- `this_pid`: l'identificativo del processo
- `root_pid`: l'identificativo del processo a cui inviare il risultato totale
- `total_number_of_processes`: il numero totale di processi

L'output è:

- `int *sum`: è la somma totale (ha senso solo per il pid con identificativo uguale a `root_pid`)

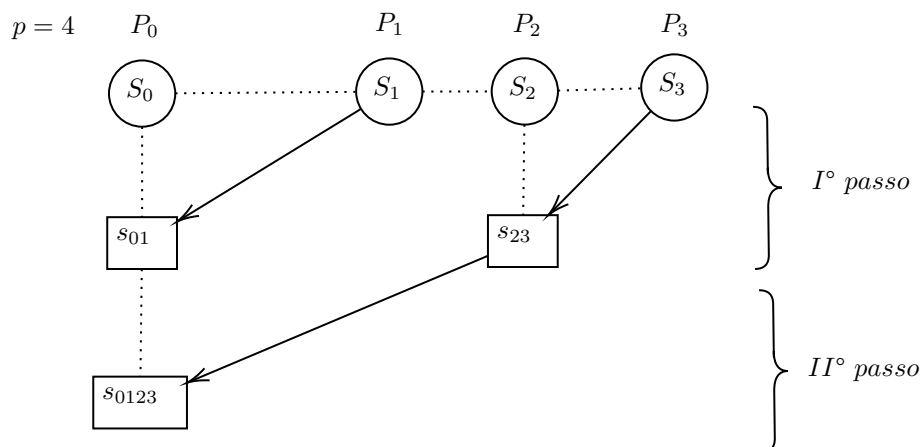
L'implementazione è la seguente:

```
void computes_strategy_one(  
    int *sum,  
    const int this_pid,  
    const int root_pid,  
    const int total_number_of_processes  
) {  
    if (this_pid == root_pid) {  
        for (int pid = 0; pid < total_number_of_processes; pid++) {  
            if (pid == this_pid) continue;  
            const int tag = MESSAGE_TAG + pid;  
            int partial_sum;  
            MPI_Recv(&partial_sum, 1, MPI_INT, pid, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);  
            *sum += partial_sum;  
        }  
    } else {  
        const int tag = MESSAGE_TAG + this_pid;  
        MPI_Send(sum, 1, MPI_INT, root_pid, tag, MPI_COMM_WORLD);  
    }  
}
```

Il processo con identificativo `root_pid` deve ricevere il risultato da tutti gli altri processi, quindi tramite `MPI_Recv` si mette in attesa di un messaggio con un TAG specifico per ogni processo che contiene un intero il quale rappresenta la somma parziale calcolata da uno dei processi. La somma parziale viene poi sommata alla somma totale.

Tutti gli altri processi inviano la propria somma parziale al processo con identificativo uguale a `root_pid`.

2.3.2 Strategia due



Questa strategia ha senso solo se il numero totale di processi è una potenza di 2 (ad esempio 2, 4 o 8). Ogni processore calcola la propria *somma parziale*. Ad ogni passo:

- coppie distinte di processori *comunicano contemporaneamente*
- in ogni coppia, un processore *invia* all'altro la propria *somma parziale che provvede l'aggiornamento della somma*

Il risultato è in un unico processore prestabilito, in questo caso il `root_pid` è sempre il processo con identificativo 0. L'implementazione è la seguente:

```
void computes_strategy_two(
    int *sum,
    const int this_pid,
    const int total_number_of_processes
) {
    for (int i = 0; i < (int)log2(total_number_of_processes); i++) {
        if (this_pid % (int)pow(2, i) == 0) {
            if (this_pid % (int)pow(2, i + 1) == 0) {
                const int pid_sender = this_pid + pow(2, i);
                const int tag = MESSAGE_TAG + pid_sender;
                int partial_sum;
                MPI_Recv(&partial_sum, 1, MPI_INT, pid_sender, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                *sum += partial_sum;
            } else {
                const int tag = MESSAGE_TAG + this_pid;
                const int pid_receiver = this_pid - pow(2, i);
                MPI_Send(sum, 1, MPI_INT, pid_receiver, tag, MPI_COMM_WORLD);
            }
        }
    }
}
```

La funzione che utilizza la strategia due ha bisogno come input:

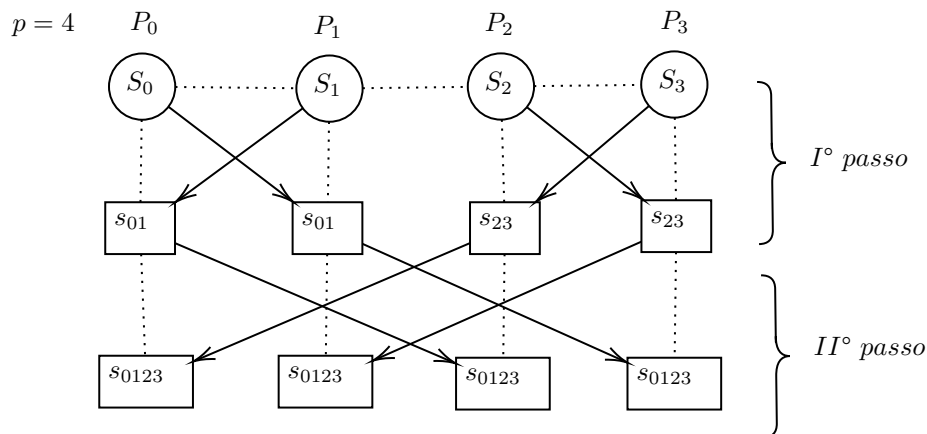
- `root_pid`: l'identificativo del processo a cui inviare il risultato totale
- `total_number_of_processes`: il numero totale di processi

L'output è:

- `int *sum`: è la somma totale (ha senso solo per il pid con identificativo uguale a 0)

In questa strategia si è scelto che il `root_pid` (il processo che riceve la somma totale) è sempre lo 0. Tale processo riceve solamente, non ha bisogno di inviare alcun risultato (come per la strategia uno il `root_pid` deve solo ricevere i risultati). In particolare deve ricevere la somma parziale calcolata da ogni coppia di processi. Ad esempio, se i processi ne sono 8, ci sono solo 4 coppie, escludendo la coppia tra il processore 0 e il processore 1, il processore con pid uguale a 0 deve ricevere le somme parziali dalla coppia 1, 2 e 3. Se il processo ha un identificativo dispari, deve inviare il risultato al processo immediatamente prima di lui (il predecessore). Se il processo è pari e diverso da 0 allora deve ricevere il risultato dal processo immediatamente dopo di lui e inviare il risultato al processo 0.

2.3.3 Strategia tre



Ogni processore calcola la propria *somma parziale*. Ad ogni passo:

- coppie distinte di processori *comunicano contemporaneamente*

- in ogni coppia i processori *si scambiano* le proprie *somme parziali*

Il risultato è in **tutti i processori**. L'implementazione è la seguente:

```
void computes_strategy_three(
    int *sum,
    const int this_pid,
    const int total_number_of_processes
) {
    for (int i = 0; i < (int)log2(total_number_of_processes); i++) {
        const int pid_receiver = (this_pid % (int)pow(2, i + 1) < (int)pow(2, i))
            ? this_pid + pow(2, i)
            : this_pid - pow(2, i);
        const int tag_receiver = MESSAGE_TAG + pid_receiver;
        const int tag_sender = MESSAGE_TAG + this_pid;
        int partial_sum;
        MPI_Request request;
        MPI_Isend(sum, 1, MPI_INT, pid_receiver, tag_receiver, MPI_COMM_WORLD, &request);
        MPI_Request_free(&request);
        MPI_Recv(&partial_sum, 1, MPI_INT, pid_receiver, tag_sender, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        *sum += partial_sum;
    }
}
```

La funzione che utilizza la strategia due ha bisogno come input:

- **root_pid**: l'identificativo del processo a cui inviare il risultato totale
- **total_number_of_processes**: il numero totale di processi

L'output è:

- **int *sum**: è la somma totale (in questo caso ha senso per tutti i processi)

A differenza delle altre strategie, questa strategia permette a tutti i processi di inviare e ricevere le somme parziali perché la finalità di tale strategia è quella di permettere a tutti i processi di ottenere la somma totale alla fine della computazione. In tutte le altre strategie abbiamo utilizzato una subroutine di MPI *bloccante* per inviare i messaggi (sarebbe a dire **MPI_Send**). In questa strategia è necessario utilizzare una subroutine **non bloccante** per inviare le somme parziali perché altrimenti il risultato sarebbe un deadlock. Il processo ha bisogno soltanto della somma parziale ricevuta dagli altri processi per continuare la propria computazione e non ha strettamente bisogno di aspettare che il messaggio venga ricevuto quando utilizza la subroutine per l'invio del messaggio. Per questo motivo utilizziamo una subroutine non bloccante per inviare il messaggio, sarebbe a dire **MPI_Isend**. Ad ogni passo un processo si mette in comunicazione con un altro processo e si scambiano reciprocamente le somme parziali.

2.4 Stampa del risultato

Alla fine viene stampato il risultato finale (un solo processo stampa oppure tutti i processi, in base ai parametri di input forniti) e il tempo massimo totale tra quelli calcolati da tutti i processi.

Ad esempio se avviamo il programma **main** eseguendo:

```
1 mpirun -np 4 main 2 -1 numbers.txt
```

Il risultato sarà:

```
[PID 0] Result: -925
[PID 1] Result: -1315
[PID 2] Result: 1436
[PID 3] Result: -288
[TOTAL TIME] 0.000074 seconds.
```

2.5 I file .pbs

Ci sono due file **.pbs**, uno per avviare il programma secondo una certa strategia per ottenere il risultato finale e il tempo totale (questo si trova nella cartella **bin**) e l'altro per misurare le performance del software in maniera automatica.

2.5.1 Il file run.pbs

Il seguente file pbs non è inteso per essere avviato direttamente con qsub su linea di comando ma **deve essere avviato utilizzando lo script run.sh**.

```
#!/bin/bash

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

sort -u $PBS_NODEFILE > hostlist
NCPU=$(wc -l hostlist)

echo -----
echo 'This job is allocated on '${NCPU}' cpu(s)'
echo 'Job is running on node(s):'
cat hostlist
PBS_O_WORKDIR=$PBS_O_HOME/parallel_sum_project/parallel_sum
echo -----
echo PBS: qsub is running on $PBS_O_HOST
echo PBS: originating queue is $PBS_O_QUEUE
echo PBS: executing queue is $PBS_QUEUE
echo PBS: working directory is $PBS_O_WORKDIR
echo PBS: execution mode is $PBS_ENVIRONMENT
echo PBS: job identifier is $PBS_JOBID
echo PBS: job name is $PBS_JOBNAME
echo PBS: node file is $PBS_NODEFILE
echo PBS: current home directory is $PBS_O_HOME
echo PBS: PATH = $PBS_O_PATH
echo -----

/usr/lib64/openmpi/1.4-gcc/bin/mpixexec -machinefile hostlist -n $N_CPU $PBS_O_WORKDIR/build/main "$STRATEGY"
rm "$PBS_O_WORKDIR/bin/$NUMBERS_FILE"
```

Un esempio di output utilizzando lo script run.sh:

PARALLEL SUM

```
Enter the number of processors (min 1 - max 8): 8
Enter the number N of numbers to add: 1
The total number of numbers to add cannot be less than the total number of processors!
Enter the number N of numbers to add: 2
The total number of numbers to add cannot be less than the total number of processors!
Enter the number N of numbers to add: 8
```

```
Number 1: 1
Number 2: 2
Number 3: 3
Number 4: 4
Number 5: 5
Number 6: 6
Number 7: 7
Number 8: 8
```

```
TOTAL CPU: 8
TOTAL NUMBERS: 8
TOTAL SUM: 1
```

```
Enter the strategy to use (1, 2 or 3): 1
```

Enter the pid that must print the result (0 to 7 or -1 if everyone has to print the result): 0

EXECUTING PARALLEL SUM
4005900.torque02.scope.unina.it

Il pbs run.pbs e' stato accodato. I risultati saranno stampati in run.out e run.err.

2.5.2 Il file performance_test.pbs

Il seguente pbs serve per testare le performance delle varie strategie. Esso avvia uno script performance_test.sh il quale misura le performance delle tre strategie e le performance dell'algoritmo iterativo dato come input il numero di CPU e il numero totale di numeri da sommare.

```
#!/bin/bash

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

sort -u $PBS_NODEFILE > hostlist
NCPU=$(wc -l hostlist)

echo -----
echo 'This job is allocated on '${NCPU}' cpu(s)'
echo 'Job is running on node(s):'
cat hostlist
PBS_O_WORKDIR=$PBS_O_HOME/parallel_sum_project/parallel_sum
echo -----
echo PBS: qsub is running on $PBS_O_HOST
echo PBS: originating queue is $PBS_O_QUEUE
echo PBS: executing queue is $PBS_QUEUE
echo PBS: working directory is $PBS_O_WORKDIR
echo PBS: execution mode is $PBS_ENVIRONMENT
echo PBS: job identifier is $PBS_JOBID
echo PBS: job name is $PBS_JOBNAME
echo PBS: node file is $PBS_NODEFILE
echo PBS: current home directory is $PBS_O_HOME
echo PBS: PATH = $PBS_O_PATH
echo -----

#echo "$N_CPU $TOT_RANDOM_NUMBERS $TOT_ITERATIONS"
$PBS_O_WORKDIR/tests/performance_testing/performance_test.sh "$N_CPU" "$TOT_RANDOM_NUMBERS" "$TOT_ITERATIONS"
```


Testing

3 Testing

Questa sezione è dedicata alle strategie di testing utilizzate per testare il software, verificare il suo corretto funzionamento, aumentare la sua affidabilità e valutare e confrontare le performance delle varie strategie. I test cases realizzati sono stati pensati per essere eseguiti in locale e non sul cluster SCOPE. Tutti i tests si trovano nella directory `tests` che si presenta come segue:

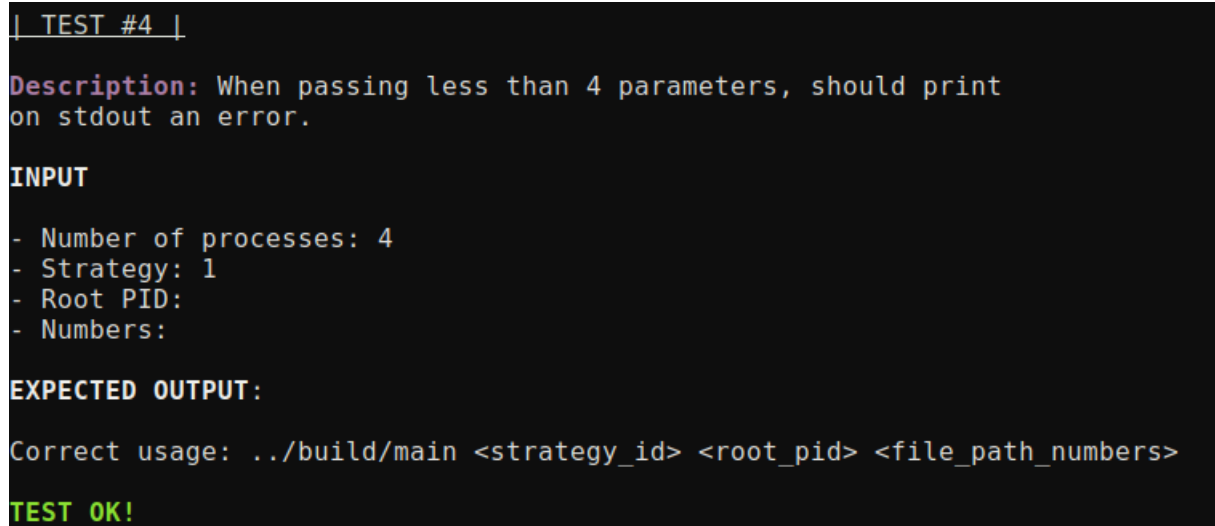
```
tests
performance_testing
  performance_suite_test.sh
  performance_test.sh
test_global.sh
test_strategy_one.sh
test_strategy_three.sh
test_strategy_two.sh
```

3.1 Black Box Testing

Sono stati realizzati una serie di test cases per verificare il corretto funzionamento del software, in particolare delle varie strategie implementate. Questa sezione è dedicata all'illustrazione dei tests presenti nella cartella `unit_testing`. La seguente directory contiene tre script:

- `test_global`: funzioni e variabili utilizzate dai diversi test suite scripts presenti nella cartella
- `test_strategy_one.sh`: test cases per testare la strategia uno
- `test_strategy_two.sh`: test cases per testare la strategia due
- `test_strategy_three.sh`: test cases per testare la strategia tre

Il file `test_strategy_one.sh` testa anche se il programma `main` risponde correttamente alle violazioni delle precondizioni illustrate nella sezione 1.5.1. Ad esempio il seguente test verifica se il programma stampa un messaggio di errore quando vengono passati meno di quattro parametri:



```
| TEST #4 |

Description: When passing less than 4 parameters, should print
on stdout an error.

INPUT

- Number of processes: 4
- Strategy: 1
- Root PID:
- Numbers:

EXPECTED OUTPUT:

Correct usage: ../build/main <strategy_id> <root_pid> <file_path_numbers>

TEST OK!
```

Figura 2: Test #4 - Strategy One

Verranno adesso elencati alcuni test cases per le strategie uno, due e tre.

```
| TEST #9 |

Description: When perform the sum with 4 processes with root PID
equals to -1 with 21 numbers, then the process with PID 0 should
print the correct total sum whereas all the others should
print correct partial sums.

INPUT

- Number of processes: 4
- Strategy: 1
- Root PID: -1
- Numbers: 319 417 491 387 215 -62 -310 -282 -472 -190 -388 -340 291 81 -354 -480 321 -440 -440 357 -316

EXPECTED OUTPUT:

[PID 0] Result: -1195
[PID 1] Result: -1642
[PID 2] Result: -802
[PID 3] Result: -518

TEST OK!
```

Figura 3: Test #9 - Strategy One

```
| TEST #6 |

Description: When perform the sum with 4 processes with root PID
equals to 2 with 35 numbers, then the process with PID 2 should
print the correct partial sum.

INPUT

- Number of processes: 4
- Strategy: 2
- Root PID: 2
- Numbers: 319 417 491 387 215 -62 -310 -282 -472 -190 -388 -340 291 81
-354 -480 321 -440 -440 357 -316 35 87 100 -100 800 -213 -873 -222 -66
6 -101 234 2222 41 35

EXPECTED OUTPUT:

[PID 2] Result: 980

TEST OK!
```

Figura 4: Test #6 - Strategy Two

```
| TEST #6 |

Description: When perform the sum with 4 processes with root PID
equals to 2 with 35 numbers, then the process with PID 2 should
print the correct total sum.

INPUT

- Number of processes: 4
- Strategy: 3
- Root PID: 2
- Numbers: 319 417 491 387 215 -62 -310 -282 -472 -190 -388 -340 291 81 -354 -480 321 -440 -440
357 -316 35 87 100 -100 800 -213 -873 -222 -666 -101 234 2222 41 35

EXPECTED OUTPUT:

[PID 2] Result: 184

TEST OK!
```

Figura 5: Test #6 - Strategy Three

3.2 Performance Testing

La cartella `performance_testing` contiene degli scripts per misurare le performance del software per ogni strategia. I risultati ottenuti verranno rappresentati come grafici. Fissiamo il numero di processi uguale a 8 (sfruttando tutto il parallelismo offerto dal cluster SCOPE) e vediamo come ogni strategia si comporta all'aumentare dell'input. Sull'asse delle ascisse c'è la dimensione dell'input mentre sull'asse delle ordinate lo **speed-up** oppure il tempo di esecuzione in secondi. Per misurare le performance dell'algoritmo con $p = 1$ è stato realizzato il miglior programma per la somma iterativa (si trova nella cartella `src` e si chiama `iterative_sum.c`):

```
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
#include <float.h>
#include <mpi.h>

typedef struct {
    unsigned int size;
    int* numbers;
} Numbers;

Numbers* read_numbers_from_file(char *file_name);

int main(int argc, char **argv) {
    if (argc != 2) {
        printf("Correct usage: %s <file_path_numbers>\n", argv[0]);
        return EXIT_FAILURE;
    }

    MPI_Init(&argc, &argv);

    char *file_name = argv[1];
    Numbers *bean_numbers = read_numbers_from_file(file_name);
    const int total_numbers = bean_numbers->size;
    const int *numbers = bean_numbers->numbers;

    const long double t0 = MPI_Wtime();

    int sum = 0;
    for (int i = 0; i < total_numbers; i++) {
        sum += numbers[i];
    }

    const long double t1 = MPI_Wtime();

    printf("[ITERATIVE SUM] Result: %d\n", sum);
    printf("[TOTAL TIME] %Le\n", t1 - t0);

    MPI_Finalize();

    return EXIT_SUCCESS;
}

Numbers* read_numbers_from_file(char *file_name) {
    FILE *file = fopen(file_name, "r");
    int n = 0;
    int total_numbers = 0;
    int index = 0;
    int dim = 1;
    int *numbers = malloc(sizeof(int));
    while (fscanf(file, "%d", &n) > 0) {
        total_numbers++;
        numbers[index++] = n;
    }
}
```

```

    numbers = realloc(numbers, sizeof(int) * ++dim);
}
fclose(file);
Numbers *bean_numbers = malloc(sizeof(numbers));
bean_numbers->size = total_numbers;
bean_numbers->numbers = numbers;
return bean_numbers;
}

```

Inoltre, vedremo come si comporta l'algoritmo della somma parallela con una determinata strategia all'aumentare del numero di processori (per la strategia uno, il range dei processori va da 2 a 8, per le altre due è possibile solo testare 2, 4 e 8).

La dimensione dell'input testata parte da 100.000 fino ad arrivare a 4.900.000 e vengono fatte 5 oppure 10 iterazioni per calcolarne la media e aumentare l'accuratezza del risultato ottenuto. Tutti i risultati sono in un file di testo `tests/performance_testing/performance_test_results.txt`. Tale file contiene una serie di records la cui semantica è:

1. Strategia adottata (1, 2, 3)
2. Numero CPU (1-8 per la strategia uno e 2, 4 e 8 per la strategia due e tre)
3. Dimensione input (range da 100.000 a 4.900.000)
4. Tempo di esecuzione medio dell'algoritmo parallelo (media di 5-10 iterazioni)
5. Tempo di esecuzione medio per l'algoritmo iterativo (media di 5-10 iterazioni)

3.2.1 Grafici strategia uno

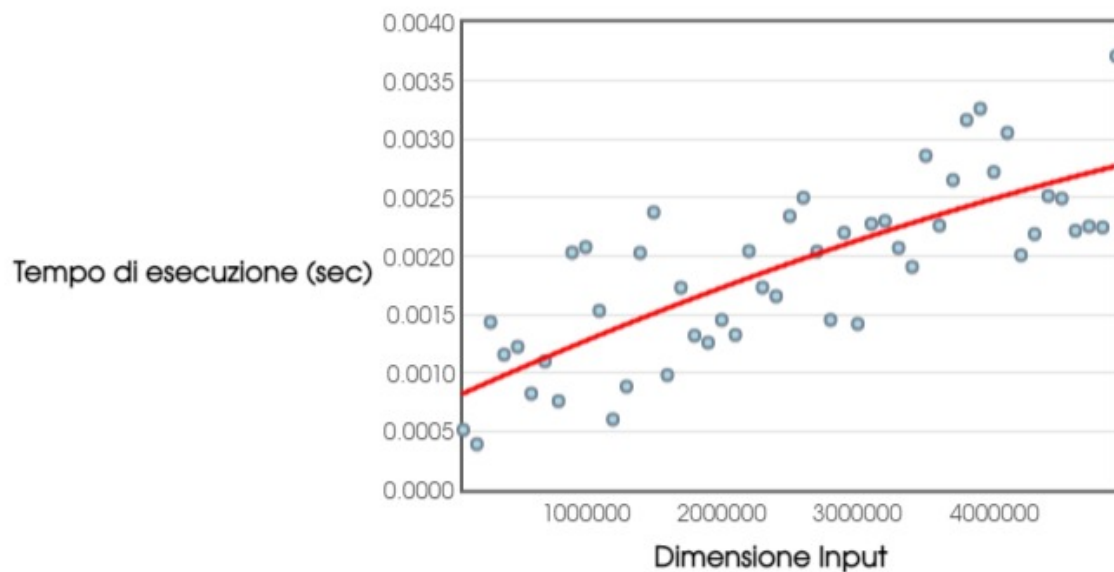


Figura 6: Tempo esecuzione prima strategia con 8 CPU all'aumentare dell'input

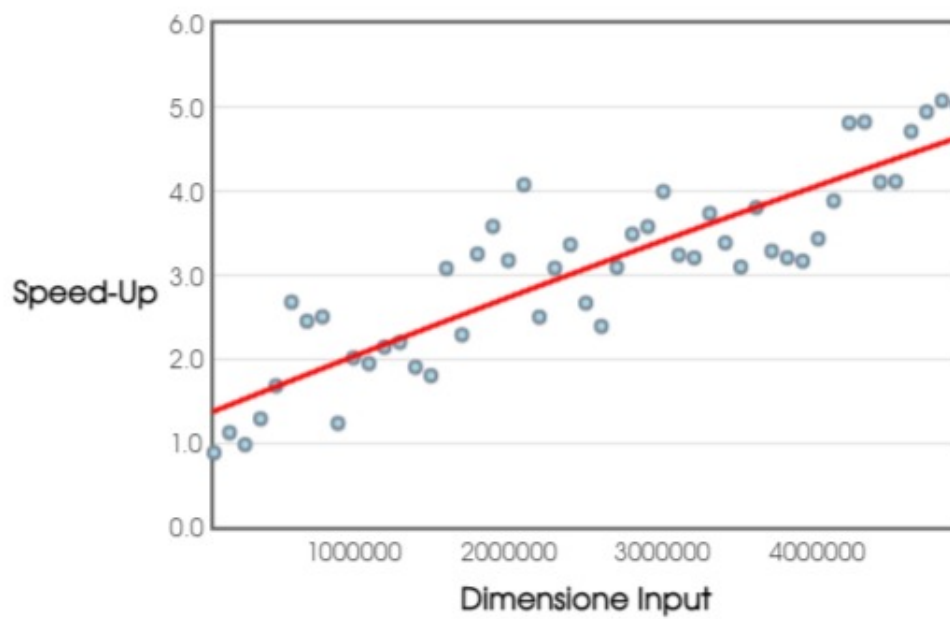


Figura 7: Speed-up prima strategia con 8 CPU all'aumentare dell'input

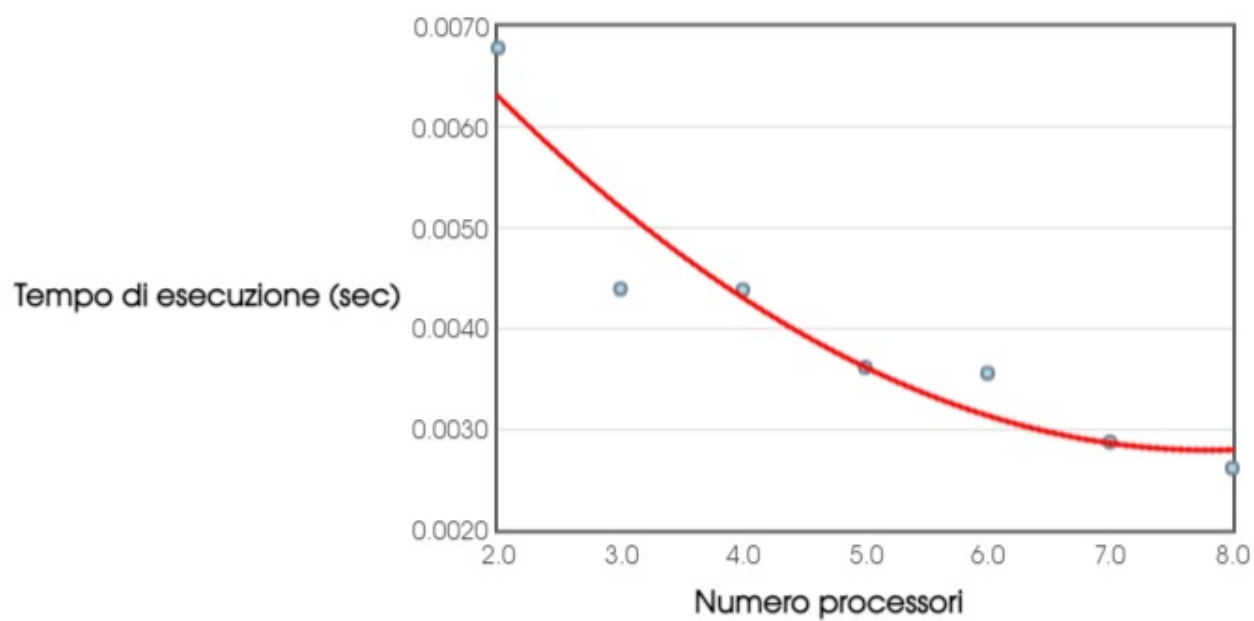


Figura 8: Tempo di esecuzione prima strategia all'aumentare dei processori

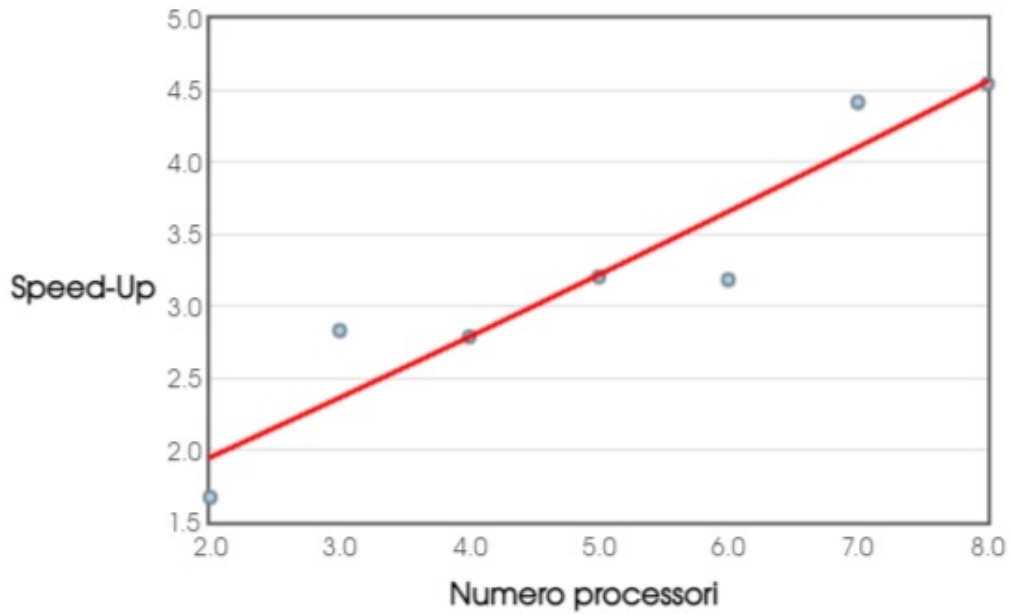


Figura 9: Speed-up prima strategia all'aumentare dei processori

3.2.2 Grafici seconda strategia

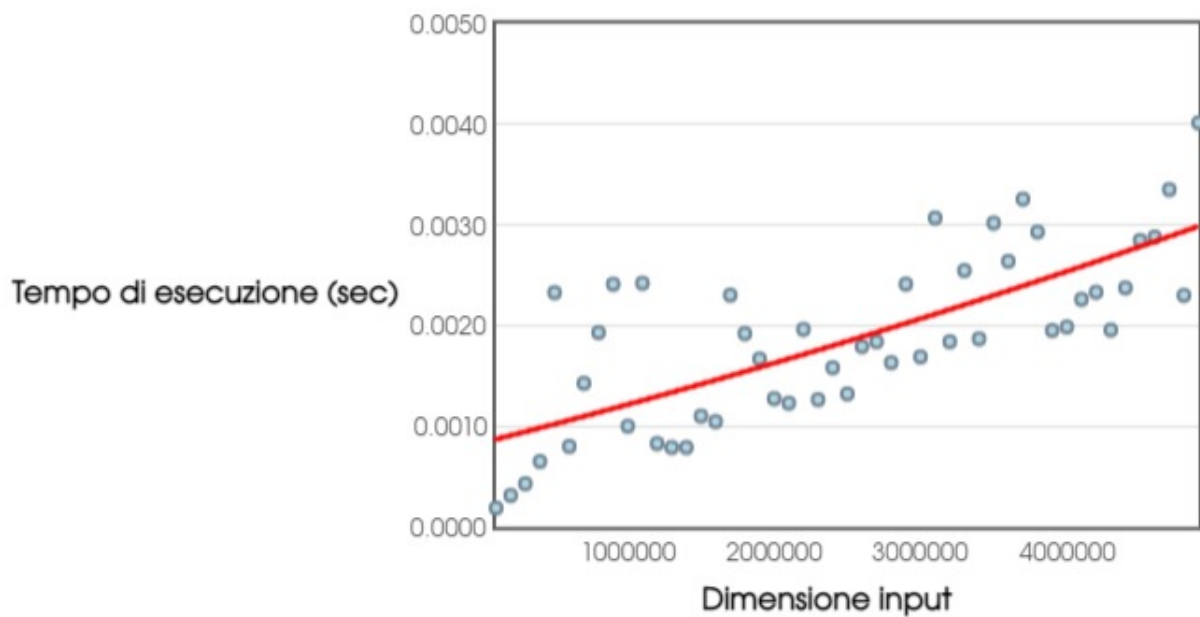


Figura 10: Tempo di esecuzione seconda strategia con 8 CPU all'aumentare dell'input

La seconda strategia può funzionare solo con un numero di processori pari a 2, 4 o 8. Pertanto il resto dei dati verranno riportati in forma tabellare.

Numero processori	Dimensione Input	Tempo medio esecuzione algoritmo parallelo	Tempo medio esecuzione algoritmo iterativo	Speed Up
2	4.900.000	0.00602856	0.011346	1.66155
4	4.900.000	0.00427246	0.011346	2.86273
8	4.900.000	0.0221652	0.011346	5.41795

Table 1: Dati seconda strategia

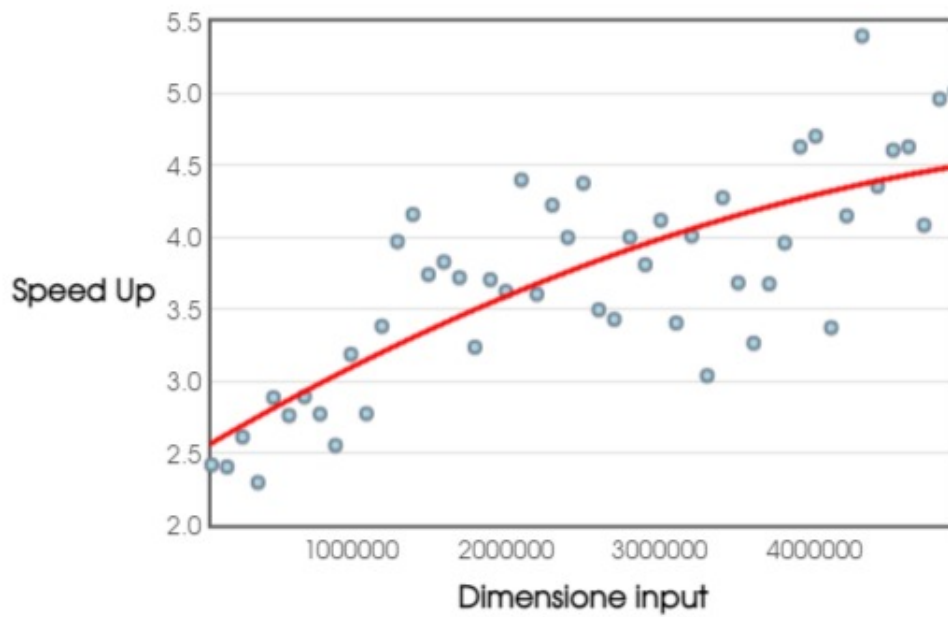


Figura 11: Speed Up seconda strategia con 8 CPU all'aumentare dell'input

3.2.3 Grafici terza strategia

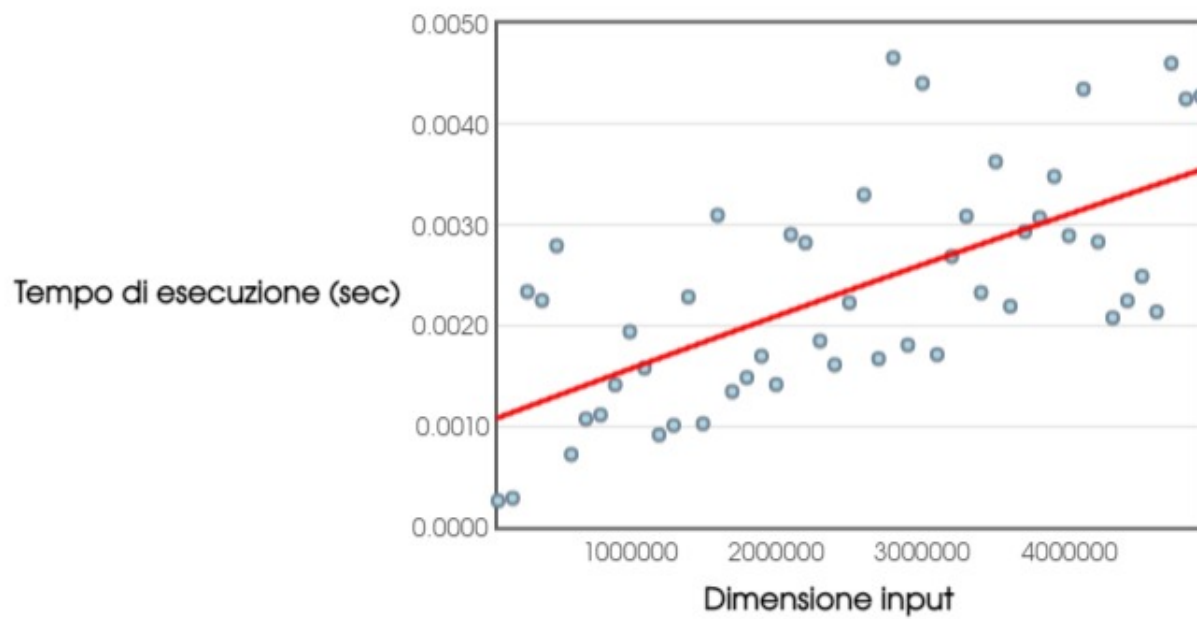


Figura 12: Tempo di esecuzione terza strategia con 8 CPU all'aumentare dell'input

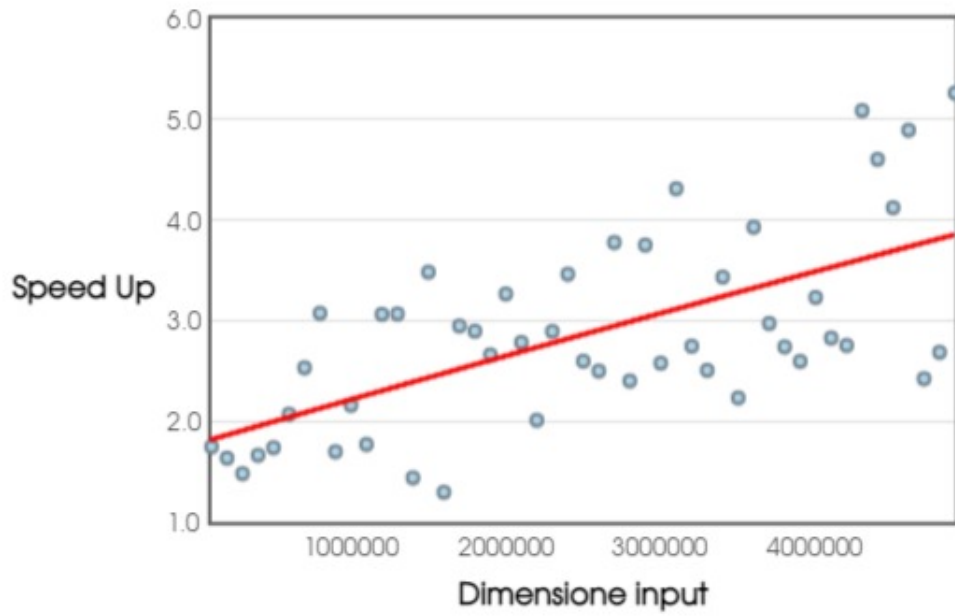


Figura 13: Speed Up terza strategia con 8 CPU all'aumentare dell'input

La terza strategia può funzionare solo con un numero di processori pari a 2, 4 o 8. Pertanto il resto dei dati verranno riportati in forma tabellare.

Numero processori	Dimensione Input	Tempo medio esecuzione algoritmo parallelo	Tempo medio esecuzione algoritmo iterativo	Speed Up
2	4.900.000	0.00610988	0.011346	1.85699
4	4.900.000	0.00404244	0.011346	3.02562
8	4.900.000	0.0028158	0.011346	5.26346

Table 2: Dati terza strategia