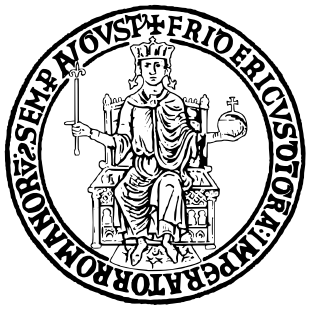
Università degli Studi di Napoli Federico II



Scuola Politecnica e delle Scienze di Base

Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione

Corso di Laurea Magistrale in Informatica

Corso di Parallel and Distributed Computing

Calcolo della somma di N numeri

Relatore
Prof. Giuliano Laccetti
Prof.ssa Valeria Mele

Candidato
Fabrizio Vitale N97/0449
Giovanni Falcone N97/1111
Luigi Mangiacapra N97/1945

Anno Accademico 2023-2024

Indice

Capitolo 1

Descrizione del problema

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

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

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

Il linguaggio scelto per lo sviluppo dell'algoritmo è il C.

Capitolo 2

Descrizione algoritmo

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

2.1 Struttura del programma

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

```
/
├── Main.c
├── Strategy.c
├── Strategy.h
├── Cmake file
└── ..
```

dove

- `Main.c` è file principale che contiene il main del programma il quale si occupa di verificare se i valori forniti in input sono validi, le inizializzazioni per utilizzare `MPI`, richiamare l'opportuna funzione dall'header file in base alla strategia scelta e, in fine, mostrare a schermo il risultato ottenuto dalla somma.
- `Strategy.c` è il file che contiene tutte le funzioni necessarie per la somma degli N valori in base alla strategia scelta.

- `Strategy.h` è il file che contiene i vari prototipi.
- ...

Capitolo 3

Descrizione routine

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

3.1 Funzioni *MPI* utilizzate

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

Descrizione: Inizializza l'ambiente di esecuzione MPI.

Parametri di input:

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

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

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

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

Parametri di input:

- `comm`: comunicatore

Parametri di output:

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

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

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

Descrizione: Restituisce la dimensione del gruppo associato al comunicatore.

Parametri di input:

- `comm`: comunicatore

Parametri di output:

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

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

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

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

Parametri di input:

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

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

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

Descrizione: Invia un messaggio in modalità standard e bloccante.

Parametri di input:

- `buf`: puntatore al buffer
- `count`: numero di elementi del buffer
- `datatype`: tipo di dato del buffer

- `dest`: identificativo del processo destinatario
- `tag`: identificativo del messaggio
- `comm`: comunicatore

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

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

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

Parametri di input:

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

Parametri di output:

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

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

```
int MPI_Finalize()
```

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

Errors: Restituisce solo `MPI_SUCCESS`

3.2 Funzioni *ausiliare* utilizzate

Capitolo 4

Testing

Capitolo 5

Analisi performance

Capitolo 6

Source code

```c
# include <stdio.h>
# include <stdlib.h>
# include <unistd.h>
# include <fcntl.h>
# include <sys/types.h>
# include <string.h>

# define BUFFER_SIZE 4096

static void err_sys(char *msg);

/**
* @brief Create a file object
*
* @param buffer
* @param file_name
*/
static void create_file(char *buffer, char *file_name);

int main(int argc, char **argv){
    char first_buffer[BUFFER_SIZE];
    char second_buffer[BUFFER_SIZE];
    int fd;
    ssize_t n_read;

    if(argc != 3){
        fprintf(stderr, "Usage %s: <file_name> <n>\n", argv[0]);
        return EXIT_FAILURE;
    }

    char *FILENAME = argv[1];
    int N = atoi(argv[2]);

    if((fd = open(FILENAME, O_RDONLY | O_RDONLY)) < 0)
        err_sys("open error");

    if((n_read = read(fd, first_buffer, N)) < 0)
        err_sys ("read error");
    first_buffer[n_read] = '\0';
```

```
40      printf("first buffer:\n%s\n\n", first_buffer);
41
42      // in questo modo evito di fare le altre chiamare ed esco ↩
        direttamente
43      if(N >= lseek(fd, 0, SEEK_END)){
44          printf("The file has been read in its entirety\n");
45          // creo il primo file
46          create_file(first_buffer, "part1");
47          return EXIT_SUCCESS;
48      }
49
50      // se non ho raggiunto la fine del file allora setto il cursore per↩
         leggere la seconda parte
51      if(lseek(fd, N, SEEK_SET) < 0)
52          err_sys("lseek error");
53
54      // leggo finche' e' possibile
55      while((n_read = read(fd, second_buffer, BUFFER_SIZE)) == 0);
56
57      if(n_read < 0)
58          err_sys ("read error");
59
60      printf("second buffer:\n%s\n\n", second_buffer);
61
62      close(fd);
63
64      // creo il primo file
65      create_file(first_buffer, "part1");
66      // creo il secondo file (nessun controllo perche' non sono uscito ↩
        dal programma nel primo if)
67      create_file(second_buffer, "part2");
68
69      return 0;
70  }
71
72
73  void create_file(char *buffer, char *file_name){
74      int fd;
75
76      if((fd = open(file_name, O_RDWR | O_CREAT | O_TRUNC, S_IRWXU)) < 0)
77          err_sys("open error");
78
79      if(write(fd, buffer, strlen(buffer)) != strlen(buffer))
80          err_sys("write error");
81
82      fprintf(stdout, "File %s created!\n", file_name);
83
84      close(fd);
85  }
86
87
88  void err_sys(char *error) {
89      perror(error);
90      exit(1);
```

```
91 }
92
```