UNIVERSITA' DEGLI STUDI DI NAPOLI "PARTHENOPE"

Dipartimento di Scienze e Tecnologie

Corso di Laurea Triennale in Informatica

Relazione di progetto per l'esame

"Calcolo Parallelo e Distribuito"

Buonomo Alessio 0124002066

Buonomo Alessio 0124002066

Definizione del problema

Si vuole realizzare l'implementazione di un algoritmo parallelo (np processori, in ambiente MPI-Docker) in cui ogni processore legga un differente vettore di dimensione M, effettui la somma dei suoi elementi nella memoria locale e concateni i risultati in un vettore contenuto nella memoria di ogni processore.

Descrizione approccio sequenziale

L'algoritmo in sequenziale, è risolvibile in maniera molto semplice. Vengono allocati e inizializzati con numeri randomici P array di dimensione M, ove P sta per il numero di processori e di conseguenza di array che vogliamo prendere in considerazione.

Viene dichiarato un array di dimensione P, che conterrà al suo interno, le somme ricavate mediante l'aggregazione degli elementi di ogni array utilizzando un ciclo for annidato.

Descrizione approccio parallelo

Per l'algoritmo in parallelo, la strategia è molto semplice, ogni processore dispone di uno degli P array in entrata, attraverso il quale effettuerà M-1 somme, per calcolare l'aggregazione degli elementi del suo array e conservarla in una variabile somma locale. Questa variabile, viene successivamente inviata da ogni processore a tutti gli altri (compreso chi invia) e inserita nell'array "somme_totali" contenuto in ogni processore, effettuando $(p-1)\cdot\tau com$.

La complessità computazionale con P = 1 dell'algoritmo in sequenziale risulta essere:

$T1 = P(M-1)\tau calc$

P = numero di array

M = dimensione degli array

Buonomo Alessio 0124002066

(M –1)τcalc = tempo per la somma locale

La complessità computazionale dell'algoritmo in parallelo risulta invece:

Tp=$(M-1)\tau$calc+$(p-1)\tau$com

$(M-1)\tau$calc = tempo per la somma locale

$(p-1)\tau$com = tempo per le communication

Ricordando che il tempo di spedizione è 2,3 volte più lento del tempo di un addizione, possiamo riscrivere il tutto come:
Tp=$(M-1)\tau$calc+c$(p-1)\tau$calc 2 <= c <= 3

Andiamo a definire i parametri di valutazione: Sp, Oh, Ep

Formula dello Speedup (Sp)

Lo speedup (Sp) è definito come il rapporto tra il tempo di esecuzione sequenziale ($T1$) e il tempo di esecuzione parallelo (Tp):

$Sp = Tp\,T1$

Utilizziamo le formule date per $T1$ e Tp:

$T1 = P(M-1)\tau$calc
$Tp = (M-1)\tau$calc $+ c(p-1)\tau calc$

Quindi, la formula per lo speedup (Sp) diventa:

$$Sp = \frac{P(M-1)\tau calc}{(M-1)\tau calc + c(p-1)\tau calc} < p$$

Formula dell'Overhead (Oh)

L'overhead (Oh) è definito come la differenza tra il tempo totale parallelo moltiplicato per il numero di processori e il tempo di esecuzione sequenziale:

$$Oh = pTp - T1$$

Utilizziamo le formule date per $T1$ e Tp:

$$Oh = p\big((M-1)\tau calc + c(p-1)\tau calc\big) - P(M-1)\tau calc$$

Semplifichiamo:

$$Oh = p(M-1)\tau calc + pc(p-1)\tau calc - P(M-1)\tau calc$$

Poiché P=p:

$$Oh = pc(p-1)\tau calc$$

Formula dell'Efficienza (Ep)

L'efficienza (Ep) è definita come il rapporto tra lo speedup (Sp) e il numero di processori (p):

$$EP = \frac{Sp}{p}$$

Usando la formula dello speedup:

$$Ep = \frac{(M-1)tcalc}{(M-1)tcalc + c(p-1)tcalc}$$

Descrizione codice parallelo

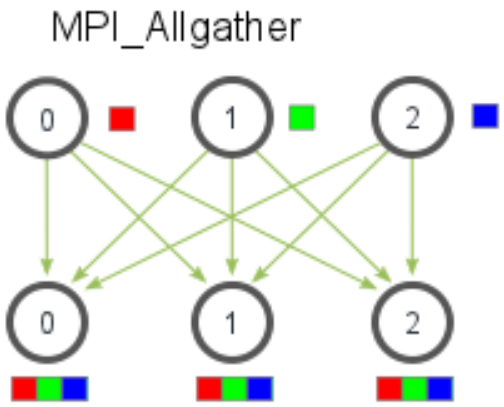
Per la parte di comunicazione, ovviamente essendo una comunicazione collettiva, ho subito adottato le operazioni collettive.

Inizialmente avevo ragionato su questo approccio, utilizzando per la comunicazione, la 3a strategia, utilizzando la funzione della libreria MPI: "**MPI_Allreduce()**", utilizzando la peculiarità dell'attributo **MPI_Op op**, che può essere posto a **MPI_OP_NULL**, per non eseguire un'operazione sul dato inviato.

Successivamente documentandomi ho trovato la funzione con firma "**int MPIAPI MPI_Allgather(_In_ void *sendbuf, _In_ int sendcount, _In_ MPI_Datatype sendtype, _Out_ void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm**

comm);)" che permette di inviare molti elementi a molti processi contemporaneamente, seguendo un modello di comunicazione molti a molti.

Con **MPI_Allgather**, un insieme di elementi distribuiti tra tutti i processi viene raccolto e redistribuito a tutti i processi. Fondamentalmente, **MPI_Allgather** può essere visto come una combinazione di **MPI_Gather** seguito da **MPI_Bcast**. L'illustrazione mostra come i dati sono distribuiti dopo una chiamata a **MPI_Allgather**.



Similmente a **MPI_Gather**, gli elementi da ciascun processo sono raccolti in ordine di rank. La differenza principale è che con **MPI_Allgather**, tutti i processi ricevono tutti gli elementi raccolti, non solo un processo radice. La dichiarazione della funzione per **MPI_Allgather** è quasi identica a quella di **MPI_Gather**, con l'unica differenza che non c'è un processo radice in **MPI_Allgather.**

Codice

Il seguente codice è visibile anche nella repo github

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>

// Funzione per eseguire l'algoritmo sequenziale
void sequential_algorithm(int P, int M, double *seq_time)
{
int **vectors = malloc(P * sizeof(int *)); // Allocazione memoria per i vettori
```

```
int *local_sums = malloc(P * sizeof(int)); // Allocazione memoria per le somme locali
clock_t start_time;
clock_t end_time;
double elapsed_time;

// Allocazione memoria per ogni vettore
for (int i = 0; i < P; i++)
{
vectors[i] = malloc(M * sizeof(int));
}

// Riempimento dei vettori con numeri casuali
for (int p = 0; p < P; p++)
{
srand(p); // Inizializzazione del generatore di numeri casuali con il seme p
for (int i = 0; i < M; i++)
{
vectors[p][i] = rand() % 10000; // Generazione di numeri casuali tra 0 e 9999
}
}
start_time = clock(); // Inizio del cronometro
// Calcolo della somma locale per ogni vettore
for (int p = 0; p < P; p++)
{
local_sums[p] = 0; // Inizializzazione della somma locale
for (int i = 0; i < M; i++)
{
local_sums[p] += vectors[p][i]; // Somma degli elementi del vettore
}
}
end_time = clock(); // Fine del cronometro
elapsed_time = (double)(end_time - start_time) / CLOCKS_PER_SEC; // Calcolo del tempo trascorso

// Stampa delle somme locali
for (int p = 0; p < P; p++)
{
printf("Numero del vettore %d: Somma Locale: %d\n", p, local_sums[p]);
```

```
}
// Stampa dell'array generale delle somme
printf("Array Generale delle somme: [");
for (int p = 0; p < P; p++)
{
printf(" %d ", local_sums[p]);
}
printf("]\n");

// Stampa del tempo di esecuzione sequenziale
printf("Tempo di esecuzione sequenziale: %f secondi\n\n", elapsed_time);
*seq_time = elapsed_time; // Salvataggio del tempo di esecuzione
// Deallocazione della memoria
for (int i = 0; i < P; i++)
{
free(vectors[i]);
}
free(vectors);
free(local_sums);
}
// Funzione per eseguire l'algoritmo parallelo
void parallel_algorithm(int M, double *par_time)
{
int *vector;
int *all_sums;
int local_sum = 0;
int size;
int rank;
double time_start;
double time_finish;
double elapsed_time;
double max_elapsed_time;

MPI_Comm_rank(MPI_COMM_WORLD, &rank); // Ottenere il rango del processo
MPI_Comm_size(MPI_COMM_WORLD, &size); // Ottenere il numero totale di processi
vector = malloc(M * sizeof(int)); // Allocazione memoria per il vettore locale
all_sums = malloc(size * sizeof(int)); // Allocazione memoria per le somme di tutti i processi
```

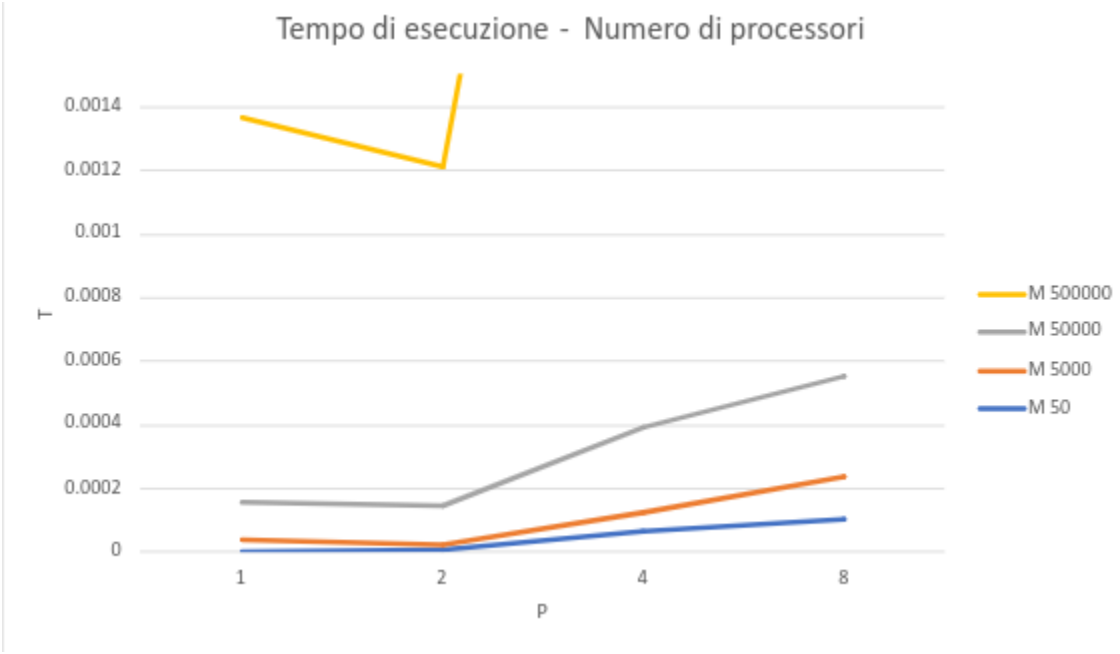
```
srand(rank); // Inizializzazione del generatore di numeri casuali con il rango del processo
for (int i = 0; i < M; i++)
{
vector[i] = rand() % M; // Riempimento del vettore con numeri casuali tra 0 e M-1
}
MPI_Barrier(MPI_COMM_WORLD); // Sincronizzazione dei processi
time_start = MPI_Wtime(); // Inizio del cronometro
// Calcolo della somma locale
for (int i = 0; i < M; i++)
{
local_sum += vector[i];
}
// Raccolta delle somme locali di tutti i processi
MPI_Allgather(&local_sum, 1, MPI_INT, all_sums, 1, MPI_INT, MPI_COMM_WORLD);
MPI_Barrier(MPI_COMM_WORLD); // Sincronizzazione dei processi
time_finish = MPI_Wtime(); // Fine del cronometro
elapsed_time = time_finish - time_start; // Calcolo del tempo trascorso
// Riduzione per ottenere il tempo massimo tra i processi
MPI_Reduce(&elapsed_time, &max_elapsed_time, 1, MPI_DOUBLE, MPI_MAX, 0,
MPI_COMM_WORLD);
// Stampa della somma locale per ogni processore
printf("Processore %d: Somma Locale: %d\n", rank, local_sum);
if (rank == 0)
{
// Stampa dell'array generale delle somme
printf("Array Generale delle somme: [");
for (int p = 0; p < size; p++)
{
printf(" %d ", all_sums[p]);
}
printf("]\n");
// Stampa del tempo di esecuzione parallela
printf("Tempo di esecuzione parallela (max): %f secondi\n\n", max_elapsed_time);
*par_time = max_elapsed_time; // Salvataggio del tempo di esecuzione
}
// Deallocazione della memoria
free(vector);
```

```
free(all_sums);
}
int main(int argc, char *argv[])
{
int rank, size, M;
double seq_time = 0.0;
double par_time = 0.0;

MPI_Init(&argc, &argv); // Inizializzazione di MPI
MPI_Comm_rank(MPI_COMM_WORLD, &rank); // Ottenere il rango del processo
MPI_Comm_size(MPI_COMM_WORLD, &size); // Ottenere il numero totale di processi
if (argc != 2)
{
if (rank == 0)
{
printf("Uso: %s <M>\n", argv[0]);
}
MPI_Finalize(); // Terminazione di MPI
return 1;
}
M = atoi(argv[1]); // Conversione dell'argomento in un intero
if (M <= 0)
{
if (rank == 0)
{
printf("M deve essere un intero positivo.\n");
}
MPI_Finalize(); // Terminazione di MPI
return 1;
}
if (rank == 0)
{
sequential_algorithm(size, M, &seq_time); // Esecuzione dell'algoritmo sequenziale
}
parallel_algorithm(M, &par_time); // Esecuzione dell'algoritmo parallelo
if (rank == 0)
{
```

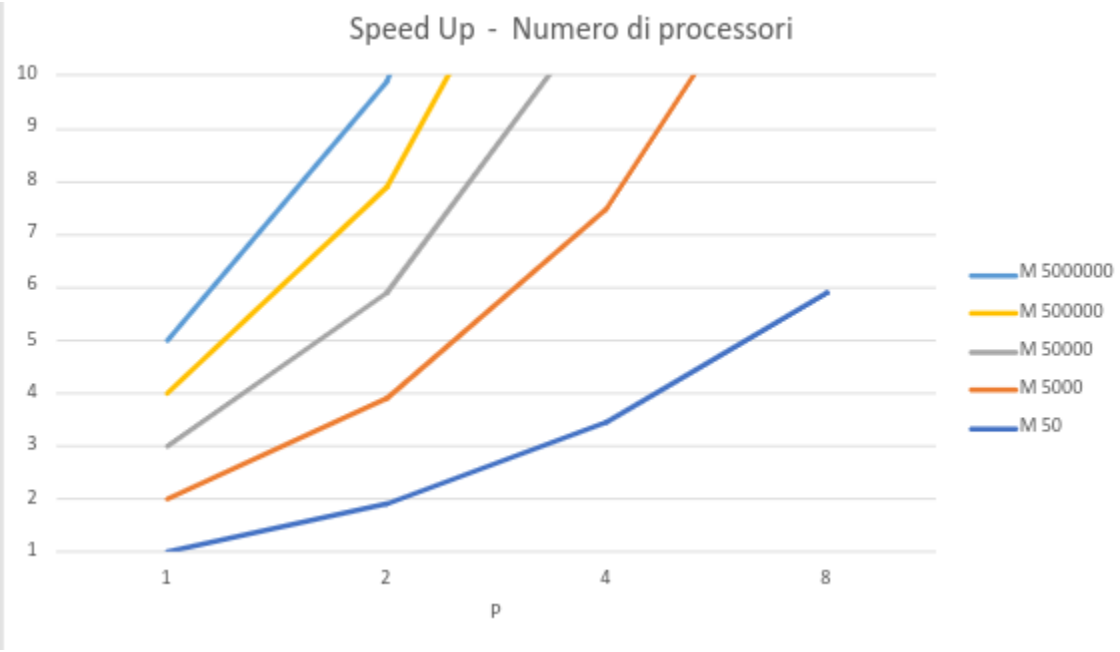
```
FILE *f = fopen("time.txt", "a"); // Apertura del file in modalità append
if (f == NULL)
{
f = fopen("time.txt", "w"); // Creazione del file se non esiste
if (f == NULL)
{
printf("Errore nell'apertura del file!\n");
exit(1);
}}
fprintf(f, "%d,%d,%f,%f\n", M, size, seq_time, par_time); // Scrittura dei tempi nel file
fclose(f); // Chiusura del file}
MPI_Finalize(); // Terminazione di MPI
return 0;
}
```

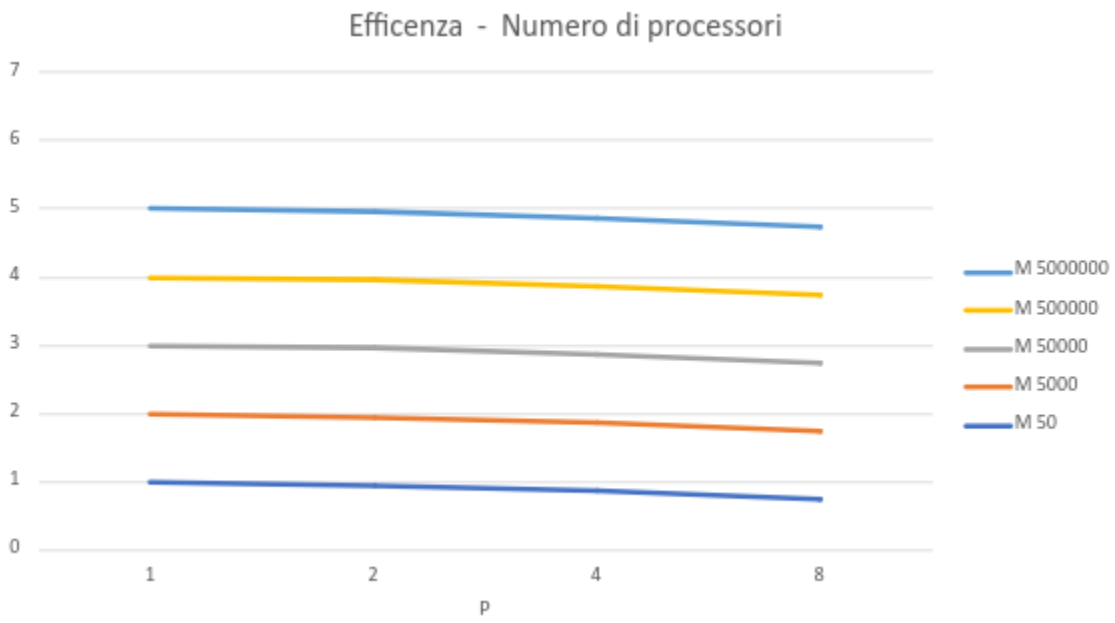
Grafici e dati raccolti



Osservando il grafico seguente, ove ho rimosso il valore M5000000, che aveva valori troppo distanti dagli altri, possiamo evidenziare un risultato anomalo l'aumentare dei tempi di esecuzione con l'aumentare del numero di processori. Di norma dovrebbe accadere che all'aumentare del numero delle unità processanti il tempo di esecuzione vada a calare. Ma essendo eseguiti i test, non su un cluster di processori, ma su un ambiente Docker, non è possibile avere un'idea appropriata dei tempi di esecuzione

effettivi dell'approccio parallelo. Di conseguenza anche i successivi grafici che sono fondati sui valori dei tempi di esecuzione ricavati, risultano poco accurati per descrivere un caso reale. La macchina sulla quale ho condotto i test, ricavati i tempi e i dati utili a formare i successivi grafici è un "Acer Aspire A515-51G" possiede un processore Intel® Core™ i7-8550U CPU @ 1.80GHz con 4 core, 8 threads per core, frequenza base a 2.4 Ghz, frequenza massima a 5.0 Ghz, con sistema operativo Ubuntu. Di seguito sono riportati i grafici che illustrano rispettivamente come variano lo speed-up e l'efficienza al variare del numero di processori utilizzati per dimensioni del problema differenti. Si precisa che nel grafico dello speed-up non si riporta lo speed-up ideale in quanto disterebbe troppo dai dati già presenti, non consentendo un'opportuna visualizzazione degli stessi. I dati sono presenti nel file time.xlsx in allegato.





Bibliografia e sitografia

- https://github.com/delucap/Docker_MPI , Pasquale De Luca
- https://stackoverflow.com/
- https://www.mpi-forum.org/
- https://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/ , Wes Kendall
- https://www.mpich.org/static/docs/latest/www3/MPI_Allgather.html
- https://mpitutorial.com/tutorials/
- https://www.youtube.com/watch?v=RoQJNx5npF4&ab_channel=SharcnetHPC

Buonomo Alessio 0124002066

Buonomo Alessio 0124002066