Relazione Analisi
Calcolo Parallelo

Prof.ssa Livia Marcellino

a.a. 2021/2022

Fabio Esposito: 0124002149
Emanuele Imparato: 0124002121
Mattia di Palma: 0124002448

Traccia progetto

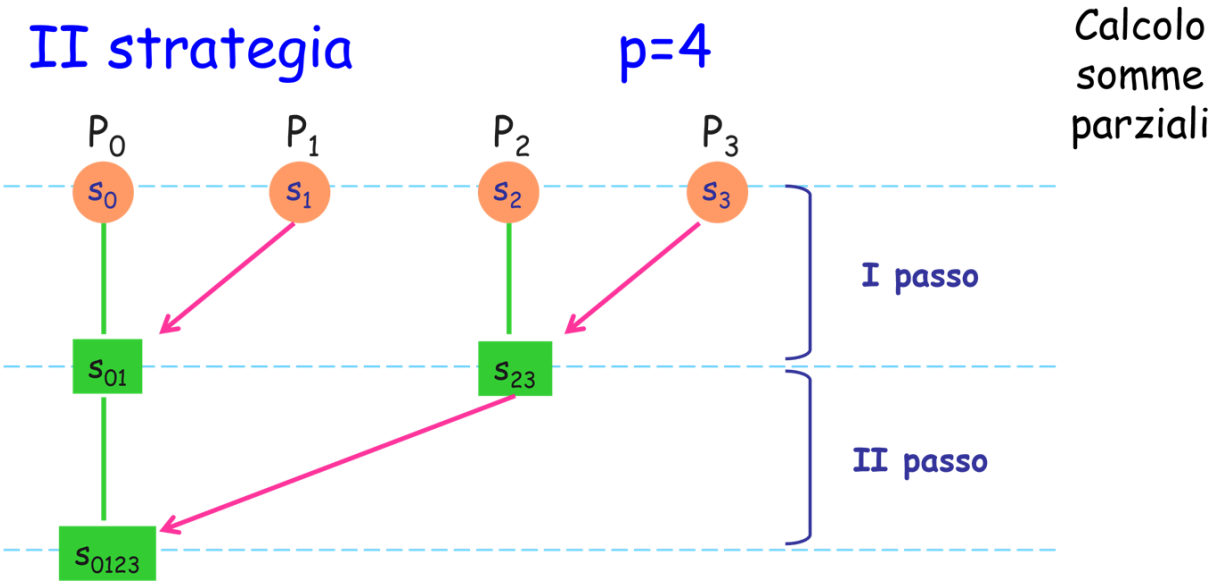
"Implementazione dell'algoritmo parallelo (np processori) della somma di N numeri (II strategia), in ambiente MPI-Docker e confronto con la routine MPI_reduce"

Definizione ed Analisi del problema

Lo scopo del software è il calcolo della somma di N numeri eseguito su architettura parallela MIMD-DM, inoltre valuteremo l'efficienza dell'algoritmo implementato e lo confronteremo con la routine MPI_reduce.
Come specificato dalla traccia l'algoritmo utilizza la seconda strategia, la quale prevede che, al termine della computazione, il risultato finale sia presente in un unico processore prestabilito, nel nostro caso sarà (P0).

Descrizione dell'approccio parallelo

1) La strategia al primo passo prevede la suddivisione e la distribuzione degli elementi da sommare ai processori
   La suddivisione cambia a seconda del numero di elementi inseriti:
   - Se il numero degli elementi da sommare è divisibile per il numero dei processori, tutti i processori riceveranno lo stesso numero di elementi da sommare.
   - Se il numero di elementi da sommare non è perfettamente divisibile per il numero dei processori, i processori con id strettamente minore del resto della divisione tra il numero degli elementi e il numero dei processori effettueranno delle somme aggiuntive.
2) La strategia al secondo passo prevede che i processori inviano o ricevono le somme parziali a seconda del loro id, ovvero:
   - i processori con id dispari inviano le somme parziali
   - i processori con id pari ricevono le somme parziali
3) La strategia al terzo passo prevede che il processore p0 con la somma totale degli addendi stampi a video il risultato.



Descrizione dell'algoritmo parallelo

E' importante notare che affinché questa strategia possa essere applicata sarà necessario disporre di un numero di processori pari ad una potenza del 2.
Come possiamo notare nel grafico soprastante le somme parziali vengono calcolate e accumulate nei processori coinvolti. I nodi disposti nei livelli sottostanti contengono quindi delle somme parziali di somme parziali ma del cui calcolo si farà carico uno solo dei due processori figli, ovviamente solo dopo aver ricevuto il risultato parziale dall'altro.
Questa strategia prevede che ad ogni passo l'algoritmo lavori in parallelo per coppie distinte di processori. Tuttavia il carico di gestione risulta parzialmente distribuito perché non tutti i processori proseguono le elaborazioni ad ogni passo temporale, di fatto alla fine dell'ultimo passo esclusivamente il nodo P0 conterrà la somma totale degli addendi a me che non venga utilizzata la funzione di broadcast.

Descrizione dell'algoritmo con relativo pseudocodice
Dopo aver definito le variabili, inizializziamo l'ambiente di esecuzione MPI con la funzione **MPI_Init**, dopodichè ad ogni processore verrà fornito un numero identificativo salvato in **menum** tramite la funzione ,**MPI_Comm_rank.**
La funzione **MPI_Comm_size** ci permette invece di salvare nella variabile **nproc** il numero di processori coinvolti nel calcolo.
La prima parte dell'algoritmo riguarda la lettura da tastiera del numero di elementi da sommare e gli elementi stessi salvandoli in un file.
Passiamo il numero degli elementi dal processore P0 ai restanti attraverso la funzione **MPI_Bcast.**
La seconda parte dell'algoritmo viene svolta in parallelo istanziando le seguenti variabili:

- **nloc** per salvare la divisione tra il numero dei processori ed n (size)
- **rest** per salvare il resto nel caso in cui il numero di addendi inseriti non sia perfettamente divisibile per il numero dei processori.

Inoltre istanziamo l'array per la suddivisione degli elementi ai processori interessati, distinguendo due casi *(di seguito pseudocodice)*.

```
if (menum < rest and rest>0) {
      nloc++;
      xloc = "istanziamo l'array di dimensione
nloc";
}
else{
      xloc = "istanziamo l'array di dimensione
nloc";
}
```

Se il numero di elementi non è divisibile per il numero dei processori, i suddetti avranno un array con un elemento in più se l'identificativo del processore è strettamente minore del resto della divisione (n/nproc).
La terza sezione di codice viene eseguita dal processore 0 dividendo in parti l'intero array inviandole agli altri processori attraverso due funzioni fondamentali ovvero **MPI_Send** e **MPI_Recv**. Dopodichè in parallelo i processori eseguono le prime somme parziali (S0,S1,S2,S3) salvandole nella variabile **sum** (di seguito pseudocodice):

```
if ( menum == 0 )
{
      for(i = 1 ; i < nproc ; i++)
                  processore che invia (MPI_SEND);
}
else
{
processore che riceve (MPI_RECV)
}

for (i = 0 ; i<nloc ;i++)
{
      sum+=xloc[i];

}
```

Nella quarta sezione di codice implementiamo la seconda strategia per il calcolo della somma di N numeri attraverso le due funzioni **SEND** e **RECV** che verranno descritte approfonditamente nella sezione delle routine implementate (di seguito pseudocodice).

```
Parziali = "allochiamo dinamicamente l'array di somme parziali".
for(i=0;i< logaritmobase2(nproc) ;i++){
 if ((menum%2^i)==0){
  if ((menum%2^i+1)==0)
  {
  "Riceve da menum+2^i e salva nell'array Parziali" (attraverso la recv)
  }
  else
  {
  "Spedisce a menum-2^i"  (attraverso la send)
  }
 }
}
```

L'ultima parte riguarda il calcolo dei tempi di esecuzione attraverso l'utilizzo della funzione **MPI_Wtime()** richiamata due volte.
La prima volta chiamata per il tempo iniziale e la seconda volta per il tempo finale. Sottraiamo infine i due tempi salvando il risultato in **t** e tramite l **MPI_Reduce** calcoliamo il massimo dei risultati e lo passiamo a p0

Input ed Output

Parametri di Input
I parametri che vengono richiesti interattivamente in ingresso sono due interi:

- Il primo parametro richiesto è il numero degli addendi da sommare, ovvero il (size). Quest'ultimo sarà il primo parametro salvato sul file .dat
- Il successivi parametri richiesti sono gli addendi che vogliamo sommare, questi ultimi sono contenuti nella variabile (numero), questa variabile verrà sovrascritta una volta che il numero appena inserito sarà salvato su un file con estensione .dat

Parametri di Output
I parametri di output visualizzati sono molteplici, ovvero:

- Distribuzione dei dati ai processori
- Invio delle somme parziali
- Stampa del risultato
- Tempo totale dell'algoritmo senza MPI_Reduce
- Tempo totale con MPI_Reduce

Al termine della relazione possono essere visionati screenshot di prova.

Routine implementate

Utilizzando l'ambiente **MIMD -DM** le routine principali sono basate sulla libreria **"MPI.h"** con l'appoggio della libreria **"Math.h** ". Le funzioni fondamentali utilizzate per implementare la seconda strategia sono **SEND, RECV e REDUCE.**
La funzione **SEND** ha sei parametri di input:
`MPI_Send(void* data,int count,MPI_Datatype datatype,int destination,int tag,MPI_Comm communicator)`

1. Il dato da inviare
2. Il count dei dati
3. Il Datatype
4. L'id del processore che riceve
5. Il tag del messaggio

6. **Il comunicator**

La funzione **RECV** ha sette parametri di input:
MPI_Recv(void* data,int count,MPI_Datatype datatype,int source,int tag,MPI_Comm communicator,MPI_Status* status)

1. **Il dato da ricevere**
2. **Il count dei dati**
3. **Il Datatype**
4. **L'id del processore che ha inviato**
5. **Il tag del messaggio**
6. **Il comunicator**
7. **Lo stato del massaggio**

L'implementazione della seconda strategia viene svolta come da traccia anche utilizzando la funzione **MPI_Reduce** per il calcolo delle somme totali.
La funzione **REDUCE** ha sette parametri di input:
MPI_Reduce(void* send_data,void* recv_data,int count,MPI_Datatype datatype,MPI_Op op,int root,MPI_Comm communicator)

1. **Il dato inviato**
2. **Il dato ricevuto**
3. **Il count del dato**
4. **Il Datatype**
5. **L'operazione da svolgere(nel nostro caso la MPI_Max per il calcolo del massimo dei tempi e la MPI_Sum per il salvataggio della somma finale)**
6. **L'id del processore master in cui viene salvato il dato**
7. **Il comunicator**

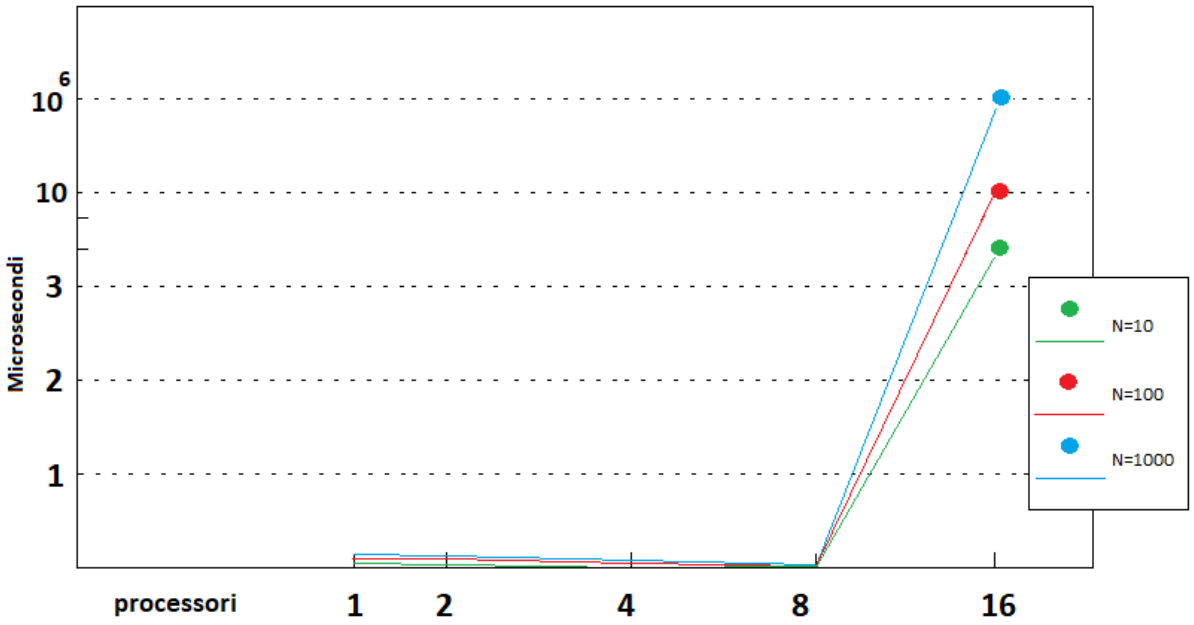
Utilizziamo inoltre l'appoggio della libreria **"Math.h"** per il calcolo dei processori che dovranno inviare le somme parziali e i processori che invece dovranno riceverle. Tutto questo attraverso l'uso dei logaritmi e delle potenze (pseudocodice pagina 5).

Analisi delle performance del software

Nella tabella che segue sono riportati i valori del tempo di esecuzione registrato al variare del numero di processori **p** e del numero di elementi da sommare **N**. Tutti i valori del **T(p)** sono espressi in microsecondi, in generale il **T(p)** viene rappresentato mediante la formula **$T_p(N)= (N/p -1 + Log_2 p)Tcalc$**
Analizzando tale formula, è importante notare che **N/p -1** indica le operazioni concorrenti,ovvero le operazioni locali -1 perchè sono le operazioni che verranno svolte effettivamente. Inoltre notiamo che **$Log_2 p$** rappresenta la profondità dell'albero e questa quantità fornisce l'esponente che deve avere la base 2 per riconoscere il numero di core.
Tcalc invece rappresenta il tempo di esecuzione di una singola operazione, questo tempo è influenzato dalla potenza di calcolo dell'architettura usata.
Ciascuno dei valori in tabella è stato ottenuto eseguendo prove sul codice tramite la generazione casuale del numero degli elementi e gli elementi stessi solamente per testare le tempistiche dell'algoritmo.
Di seguito è possibile visionare grafici e tabelle

p \ n	10	100	1000
1	1,143	2,035	8,467
2	2,243	10,345	18,675
4	3,410	5,213	25,345
8	9,240	24,532	42,320
16	38493,384	42345,219	49543,123

Tabella dei tempi di esecuzione T_p



Guardando la Tabella dei tempi di esecuzione possiamo notare che per N fissato non è sempre conveniente parallelizzare l'algoritmo. Ad esempio per N=1.000 i tempi di esecuzione crescono al crescere del numero dei processori, il caso più evidente mostrato è quando il numero dei processori è 16.

Nella seconda tabella che segue è riportato il valore dello **speed-up** ricordando che lo **speed-up** misura la riduzione del tempo di esecuzione rispetto all'algoritmo su un processore.

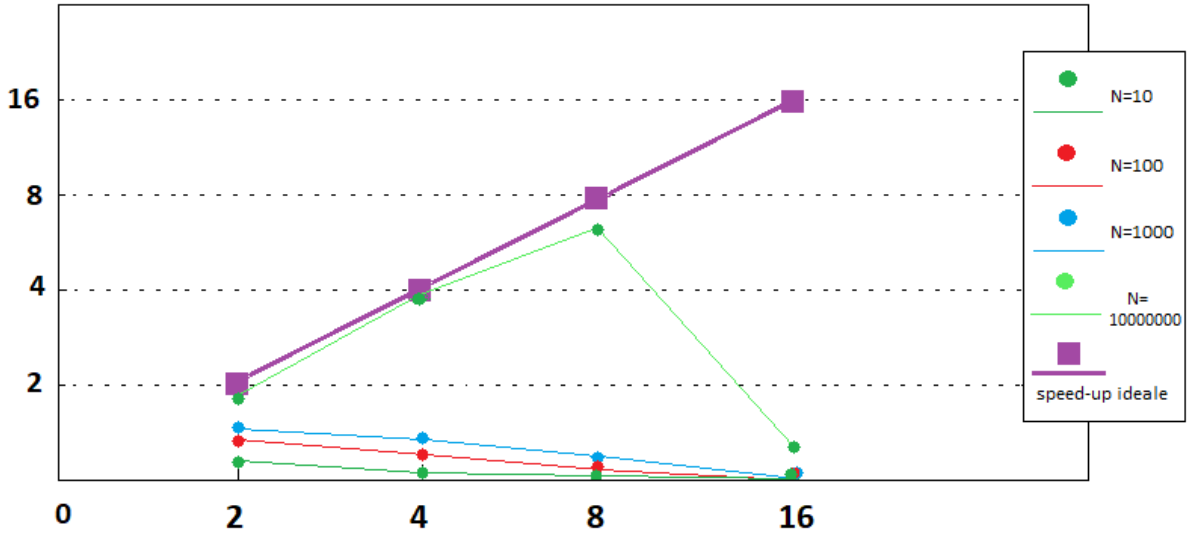
$S_p(n) = T_1(n)/T_p(n)$

p \ n	10	100	1000	10000000
2	0,509	0,196	0,453	1,943
4	0,331	0,390	0,334	3,894
8	0,123	0,082	0,200	6,794
16	0,00014	0,00094	0,0018	0,864

Valori dello Speed-up S_p al variare di N e di P

Nel grafico sottostante confrontiamo le varie esecuzioni con **lo speed-up ideale** il

cui il rapporto **$T_1(n)/T_p(n)$ è uguale a numero dei processori(P).**
Analizzando il grafico e la tabella osserviamo subito che i dati non si avvicinano mai allo speed up ideale per N = 10, N=100 ed N = 1000 mentre per N molto grandi come ad esempio N= 10000000 con P=2, P=4 e P=8 i valori registrati iniziano ad avvicinarsi molto allo speed-up ideale. Mentre per P > 8 il valore dello speed-up si allontana da quello ideale e questo ci fa capire che all'aumentare del numero dei processori non traiamo nessun beneficio dall'impiego dell'architettura parallela.



Nella tabella che segue è riportato il valore dell'**efficienza** ricordando che quantifica lo sfruttamento del parallelismo da parte dell'algoritmo.
$E_p(n) = S_P(n)/P$

p \ n	10	100	1000
2	0,254	0,098	0,226
4	0,082	0,095	0,083
8	0,015	0,010	0,025
16	0,00008	0,00005	0,00011

Valori efficienza E(p) al variare di N e p

Un altro concetto da analizzare riguardante l'efficienza è quello di **isoefficienza** ovvero la funziona che determina **il giusto rapporto N/P** (size e numero processori) tale che al crescere di entrambi **l'efficienza tenda ad una costante ($E_p(n) = k$)**
La formula per il calcolo dell'isoefficienza è basata sulla definizione dello **speed-up** sfruttando il concetto di **overhead da cui deriva $I = O_h(n1,P1)/O_h(n0,P0)$.**

Casi d'uso

In questa sezione mostreremo i casi d'uso riportando esempi di esecuzioni.
Di seguito gli esempi:

1. N = 10 , P = 4, elementi scelti in input {1,2,3,4,5,6,6,7,8,9}

```
cpd2021@1955a6e71152: /Docker_MPI/exercise_01
Inserisci il numero:5
Inserisci il numero:6
Inserisci il numero:6
Inserisci il numero:7
Inserisci il numero:8
Inserisci il numero:9

PARTE 1
DISTRIBUZIONE DEI DATI AI PROCESSORI

Sono 1 ->  4  5  6  Somma parziale = 15

Sono 0 ->  1  2  3  Somma parziale = 6

Sono 2 ->  6  7  Somma parziale = 13

Sono 3 ->  8  9  Somma parziale = 17

PARTE2
INVIO DELLE SOMME PARZIALI

Il processore 0 riceve dal processore 1 la somma parziale 15--> risultato 21

Il processore 2 riceve dal processore 3 la somma parziale 17--> risultato 30

Il processore 1 invia al processore 0 la somma parziale 15

Il processore 0 riceve dal processore 2 la somma parziale 30--> risultato 51

Il processore 3 invia al processore 2 la somma parziale 17

Il processore 2 invia al processore 0 la somma parziale 30


PARTE 3
STAMPA SOMMA CONTENUTA SOLO IN P0 SENZA BROADCAST

sono 0 e questa è la somma totale 51

sono 0 e questa è la somma totale utilizzando MPI_Reduce 51

Tempo algoritmo = 0.039833
Tempo algoritmo con MPI_reduce 0.002584
cpd2021@1955a6e71152:/Docker_MPI/exercise_01$
```

2. N = 16, P = 4, elementi scelti in input {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16}

```
cpd2021@1955a6e71152: /Docker_MPI/exercise_01
Inserisci il numero:11
Inserisci il numero:12
Inserisci il numero:13
Inserisci il numero:14
Inserisci il numero:15
Inserisci il numero:16

PARTE 1
DISTRIBUZIONE DEI DATI AI PROCESSORI

Sono 3 ->  13  14  15  16  Somma parziale = 58

Sono 1 ->  5  6  7  8  Somma parziale = 26

Sono 2 ->  9  10  11  12  Somma parziale = 42

Sono 0 ->  1  2  3  4  Somma parziale = 10

PARTE2
INVIO DELLE SOMME PARZIALI

Il processore 3 invia al processore 2 la somma parziale 58

Il processore 0 riceve dal processore 1 la somma parziale 26--> risultato 36

Il processore 0 riceve dal processore 2 la somma parziale 100--> risultato 136

Il processore 2 riceve dal processore 3 la somma parziale 58--> risultato 100

Il processore 2 invia al processore 0 la somma parziale 100

Il processore 1 invia al processore 0 la somma parziale 26


PARTE 3
STAMPA SOMMA CONTENUTA SOLO IN P0 SENZA BROADCAST

sono 0 e questa è la somma totale 136

sono 0 e questa è la somma totale utilizzando MPI_Reduce 136

Tempo algoritmo = 0.170750
Tempo algoritmo con MPI_reduce 0.000131
cpd2021@1955a6e71152:/Docker_MPI/exercise_01$
```

CODICE

```
#include <stdlib.h>
#include <errno.h>
#include <mpi.h>
#include <stdio.h>
#include <math.h>
int main (int argc, char* argv[])
{
FILE *filePointer;
int menum, nproc;
int n, nloc, tag, i, rest, tmp = 0, start = 0, sum = 0, sumtot = 0;
int * x, * xloc; // array per il salvataggio degli elementi
int size; //variabile size
int numero; //variabile numero
int z; //variabile contatore
int lg; // variabile per il calcolo del logaritmo in base due
int sumP=0; // variabile per somme parziali
double t1,t2,t,tempoparziale;
double tr1,tr2,tr;
double tempo_totale,tempo_totaleR;
MPI_Status status;
MPI_Init(NULL,NULL);
MPI_Comm_rank(MPI_COMM_WORLD, & menum);
MPI_Comm_size(MPI_COMM_WORLD, & nproc);

//PARTE 1 -> LETTURA DEL SIZE E DEGLI ELEMENTI DA TASTIERA
if (menum == 0) {

filePointer=fopen("/Docker_MPI/exercise_01/inputData.dat", "w");
if( filePointer==NULL ) {
        perror("Errore in apertura del file");
        exit(1);
}
printf("Inserisci il numero di elementi da sommare:");
fflush(stdout);
scanf("%d", &size); // legge il numero del size da tastiera
fprintf(filePointer, "%d\n", size);// scrive il numero del size da tastiera
for(z=0;z<size;z++){//ciclo da 0 al numero del size
    printf("Inserisci il numero:");
    fflush(stdout);
    scanf("%d", &numero);
    fprintf(filePointer, "%d\n", numero); //salvataggio nel file
}
fclose(filePointer);

    filePointer = fopen("/Docker_MPI/exercise_01/inputData.dat", "r");
    if (!filePointer) {
        printf("\nErrore durante l'apertura del file contenente i dati di input\nExit...\n");
        printf("Error %d \n", errno);
        return(-2);
    }
```

```
        fscanf(filePointer, "%d", &n);

        x = (int*)calloc(n,sizeof(*x));
        for( i = 0 ; i <n ; i++)
        {
        fscanf(filePointer,"%d",&x[i]);
        }
fclose(filePointer);
 }

// passaggio del size dal processore master ai vari nodi
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

MPI_Barrier(MPI_COMM_WORLD); // Blocco dei processori per l'inizio del calcolo dei tempi
t1 = MPI_Wtime(); //tempo iniziale

nloc = n/nproc; // Dimensione dei sottoarray distribuiti ai processori
rest = n%nproc; //Salvataggio resto della divisione
tag = 100;
lg = log(nproc)/log(2);

//Controllo sulla divisibilità del size inserito e il numero dei processori
if (menum < rest && rest>0) {
//allocazione dell'array con più elementi (caso di non divisibilità del size e del numero di
processori)
        nloc++;
        xloc = (int*)calloc(nloc,sizeof(*xloc));
else{
        xloc = (int*)calloc(nloc,sizeof(*xloc));
}

//PARTE DUE -> DISTRIBUZIONE DEI DATI (SUDDIVISIONE DEI SOTTOARRAY)
if ( menum == 0 )
{
        xloc = x ;
        tmp = nloc ;
        start = 0 ;
        for(i = 1 ; i < nproc ; i++)
        {
                start +=tmp;
            if (i==rest)
                tmp--;
                MPI_Send(&x[start],tmp,MPI_INT,i,tag,MPI_COMM_WORLD);
        }
}
else
{
MPI_Recv(xloc,nloc,MPI_INT,0,tag,MPI_COMM_WORLD,&status);
}

//Somme parziali (S0,S1,S2,S3)
```

```
for (i = 0 ; i<nloc ;i++)
{
	sum+=xloc[i];
}

if(menum == 0 )
{
printf("\nPARTE 1\nDISTRIBUZIONE DEI DATI AI PROCESSORI\n");
fflush(stdout);
}


MPI_Barrier(MPI_COMM_WORLD);


printf("\nSono %d -> ",menum);
fflush(stdout);
for (i = 0 ; i<nloc ;i++){
printf ( " %d ",xloc[i]);
fflush(stdout);
}

printf(" Somma parziale = %d\n",sum);
fflush(stdout);

MPI_Barrier(MPI_COMM_WORLD);

if(menum == 0){
	printf("\nPARTE2\nINVIO DELLE SOMME PARZIALI\n");
	fflush(stdout);
}
MPI_Barrier(MPI_COMM_WORLD);
t2 = MPI_Wtime();
t = t2-t1 ;
tempoparziale = t;
tr1 = MPI_Wtime(); //Tempo iniziale per la routine MPI_Reduce


MPI_Reduce(&sum,&sumtot,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);  //Calcolo somme totali con MPI_Reduce
tr2 = MPI_Wtime(); //Tempo finale per la routine MPI_Reduce

tr = tr2 -tr1 ; //tempo totale

MPI_Reduce(&tr, &tempo_totaleR, 1, MPI_DOUBLE, MPI_MAX,0,MPI_COMM_WORLD); //ricerca del massimo tempo totale

MPI_Barrier(MPI_COMM_WORLD);
//PARTE TRE -> CALCOLO SOMME PARZIALI UTILIZZANDO LE FUNZIONI SEND/RECV
t1 = MPI_Wtime();
int *parziali = (int*)calloc(lg,sizeof(*parziali));
int j = 0 ;
```

```
for (i = 0; i<lg ; i++){
	int p1 = pow(2,i);
	int p2 = pow(2,i+1);
	if ((menum%p1) == 0)
	{
		if ((menum%p2 ) == 0)
		{
			MPI_Recv(&sumP,1,MPI_INT,menum + p1,tag,MPI_COMM_WORLD,&status); //
funzione di ricezione del dato MPI_INT
				printf("\nIl processore %d riceve dal processore %d la somma parziale
%d",menum,menum+p1,sumP);
			fflush(stdout);
			sum += sumP; // Salvataggio delle varie somme compresa la finale
			parziali[j] = sum; //Salvataggio somme parziali
			j++;
			printf("--> risultato %d\n",sum);
			fflush(stdout);
		}
		else
		{
			MPI_Send(&sum,1,MPI_INT,menum - p1,tag,MPI_COMM_WORLD);//Funzione per
l'invio del dato MPI_INT
			printf("\nIl processore %d invia al processore %d la somma parziale
%d\n",menum,menum-p1,sum);
			fflush(stdout);
		}
	}
}

MPI_Barrier(MPI_COMM_WORLD);

//PARTE QUATTRO -> STAMPA A VIDEO DELLA SOMMA FINALE E DEI TEMPI CON LE DUE
ROUTINE (SEND/RECV, MPI_REDUCE)
if(menum == 0)
{
printf("\nPARTE 3\n");
	fflush(stdout);
	printf("STAMPA SOMMA CONTENUTA SOLO IN P0 SENZA BROADCAST");
}

MPI_Barrier(MPI_COMM_WORLD);
if(menum == 0 )
{
	printf("\nsono %d e questa è la somma totale %d\n ",menum,sum);
	printf("\nsono %d e questa è la somma totale utilizzando MPI_Reduce %d\n
",menum,sumtot);
	fflush(stdout);
}


MPI_Barrier(MPI_COMM_WORLD);
t2 = MPI_Wtime();
```

```
t = tempoparziale + (t2-t1);
MPI_Reduce(&t, &tempo_totale, 1, MPI_DOUBLE,      MPI_MAX,0,MPI_COMM_WORLD);
MPI_Barrier(MPI_COMM_WORLD);
if(menum == 0)
{
      printf("\nTempo algoritmo = %f\n",tempo_totale);
      printf("Tempo algoritmo con MPI_reduce %f\n",tempo_totaleR);
}

MPI_Finalize();
return 0 ;
}
```