UNIVERSITÀ DEGLI STUDI DI NAPOLI
FEDERICO II

PARALLEL AND DISTRIBUITED COMPUTING

Corso di Laurea in Informatica

Progetto 1

SOMMA DI N NUMERI IN AMBIENTE MIMD-DM

Studente:

Matteo Richard Gaudino

N86003226

ANNO ACCADEMICO 2020/2021

Indice

Capitolo 1

Presentazione e Analisi del Problema

Sommario

1.1 Il Problema

L'obiettivo è sviluppare un algoritmo per la somma di n numeri su un calcolatore MIMD a memoria distribuita utilizzando C e MPI pre lo sviluppo del programma.

1.2 Distribuzione dell'Input

Siano ρ il numero di processori e n la quantità di numeri da sommare. Ogni processore ha un numero identificativo $rank = 0...\rho - 1$. I numeri vanno distribuiti equamente quindi ogni processore riceverà

$$\left\lfloor \frac{n}{\rho} \right\rfloor$$

numeri da sommare. Se la divisione ha resto i restanti numeri verranno distribuiti ai primi $rest(\frac{n}{\rho})$ processi ($rest(\frac{n}{\rho}) < \rho$ per T. Divisione Euclida). In questa versione del programma l'input viene letto dal processo ρ_0 e distribuito agli altri.

```
MPI_Bcast(&nums_c, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&strategy, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&outputRank, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

```
int rest = nums_c % worldSize;
int numToSum = nums_c/worldSize + ((rank < rest)? 1: 0);

if(rank == 0){
    int tmp = nums_c/worldSize;
    int cursor = numToSum;

    for (int i = 1; i < worldSize; ++i){
        int inumToSum = tmp + ((i < rest)? 1: 0);
        MPI_Send(nums_v + cursor, inumToSum, MPI_INT, i, 0, MPI_COMM_WORLD);

        cursor += inumToSum;
    }
} else{
    nums_v = malloc(sizeof(int)*numToSum);
    MPI_Recv(nums_v, numToSum, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
}
```

1.3 Calcolo delle Somme Parziali

Ogni processo somma il vettore che ha ricevuto in input da ρ_0

```
int sum = 0;
for (int i = 0; i < numToSum; i++){
    sum += nums_v[i];
}
```

Capitolo 2

Comunicazione delle Somme Parziali

Sommario

2.1 Soluzione I

Ogni processo invia la propria somma a ρ_0 che si occupa di elaborare il risultato. Con questa soluzione solo ρ_0 conosce il risultato.

```
if (rank != 0){
    MPI_Send(&sum, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
} else{
    for (int i = 1; i < worldSize; i++){
        int sum2;
        MPI_Status info;
        MPI_Recv(&sum2, 1, MPI_INT, i, 0, MPI_COMM_WORLD, &info);

        sum += sum2;
    }
}
```

2.2 Soluzione II

I processi si dividono in coppie mittente-ricevente. Ad ogni iterazione il ricevente riceve la somma parziale del mittente e aggiorna la propia somma. Alla fine solo ρ_0 è a conoscenza del risultato.

```
for(int i = 0; i < commSteps; ++i){
    if((rank % pow2) == 0){
        if((rank % (pow2 << 1)) == 0){
            int sum2;
            MPI_Recv(&sum2, 1, MPI_INT, rank + pow2, 0, MPI_COMM_WORLD, &status);
            sum += sum2;
        } else{
            MPI_Send(&sum, 1, MPI_INT, rank - pow2, 0, MPI_COMM_WORLD);
        }
        pow2 <<= 1;
    }
}
```

- $pow2$ inizialmente settato a 1 serve a calcolare le potenze di 2
- $commSteps = log_2(\rho)$

2.3 Soluzione III

Come per la Soluzione II ad ogni iterazione i processi inviano e ricevono le somme parziali. La differenza è che alla fine **tutti** i processori sono a conoscenza del risultato

```
int sum2;
for(int i = 0; i < commSteps; ++i){
    if((rank % (pow2 << 1)) < pow2){
        MPI_Send(&sum, 1, MPI_INT, rank + pow2, 0, MPI_COMM_WORLD);
        MPI_Recv(&sum2, 1, MPI_INT, rank + pow2, 0, MPI_COMM_WORLD, &status);
    } else{
        MPI_Recv(&sum2, 1, MPI_INT, rank - pow2, 0, MPI_COMM_WORLD, &status);
        MPI_Send(&sum, 1, MPI_INT, rank - pow2, 0, MPI_COMM_WORLD);
    }
    sum += sum2;
    pow2 <<= 1;
}
```

Capitolo 3

Esecuzione e Testing

Sommario

3.1 Compilazione

Per compilare il programma eseguire il comando

mpicc -std=gnu99 sommaPar.c -o sommaPar -lm

Il comando -lm serve a linkare la libreria <math.h>. Per la compilazione del programma è necessario l'utilizzo dello standard gnu99 dato che vengono utilizzate macro definite in *unistd.h*.

3.2 Esecuzione

Il programma per essere eseguito correttamente su più macchine va lanciato con il comando mpiexec in questo modo

mpiexec -machinefile $< hostfile >$ **-np** $< \rho >$ **sommaPar** $< ... >$

La sintassi del programma sommaPar è invece strutturata in questo modo

sommaPar [-s strategy] [-o output_rank] -f filename

sommaPar [-s strategy] [-o output_rank] -r numToSum

sommaPar [-s strategy] [-o output_rank] -n numToSum num_1 ... num_n

Il falag **-s** stabilisce la strategia da utilizzare 1, 2 o 3. Nel caso in cui il numero di processori non sia una potenza di 2 le strategie 2 e 3 non sono applicabili e quindi verrà utilizzata la strategia 1. Il flag **-o** invece stabilisce quale processo stamperà l'output $0...\rho - 1$. Nel caso **-o** sia settato a -1 tutti i processi stamperanno il loro risultato con il proprio tempo di esecuzione locale. I flag **-s** e **-o** sono opzionali e i loro valori di default sono rispettivamente 1 e 0.

I flag **-f**, **-r**, **-n** stabiliscono da dove verrà prelevato l'input. Questi flag sono mutuamente esclusivi. Utilizzando il flag **-f** il programma preleverà l'input dal file chiamato $< filename >$ formattato in questo modo:

- La prima linea contiene n la quantità di numeri da sommare
- Le seguenti n linee contengono i numeri da sommare

Utilizzando il flag **-r** il programma genererà $< numToSum >$ numeri casuali interi unsigned di massimo 32bit e li utilizzerà per effettuare la somma. Utilizzando il flag **-n** il programma utilizzerà i numeri inseriti da riga di comando (num_1 ... num_n) per effettuare la somma. Per realizzare il parsing dell'input sono state utilizzate le funzioni presenti nel file header *unistd.h* ciò rende il programma compatibile solo con sistemi POSIX.

3.3 Testing

Il testing è stato effettuato sui calcolatori dell'infrastruttura SCoPE della Federico II. La lista dei componenti è reperibile dal sito ufficiale del datacenter. Il programma è stato testato eseguendo il seguente script scritto in bash. I file di output sono reperibili alla pagina GitHub del progetto.

```
echo "-------------- test 1 processor --------------"

head -1 hostlist > host1
for f in $(ls $PBS_O_WORKDIR/tests)
do
    echo Test: $f
    for i in {1..5}
    do
        res=$(/usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile host1 -np 1
        ↪  $PBS_O_WORKDIR/sommaPar -f $PBS_O_WORKDIR/tests/$f)
        echo $res >> $PBS_O_WORKDIR/results/p1$f
    done
done
```

```
for p in 2 4 8
do
   head -$p hostlist > host$p
    echo "-------------- test $p processor --------------"
    for strat in 1 2 3
    do
        for f in $(ls $PBS_O_WORKDIR/tests)
        do
            echo Strategy: $strat, test: $f
            for i in {1..5}
            do
                res=$(/usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile
                ↪  host$p -np $p $PBS_O_WORKDIR/sommaPar -s $strat -f
                ↪  $PBS_O_WORKDIR/tests/$f)
                echo $res >> $PBS_O_WORKDIR/results/p${p}s${strat}$f
            done
        done
    done
done
```

Lo script non fa altro che lanciare 5 volte il programma per ogni file di input presente nella cartella tests, per 1, 2, 4 e 8 processori. L'output è poi salvato nella cartella results in file con il seguente formato:

$$p < \rho > s < strategy >< filename >$$

Il comando $head - \$p\ hostlist > host\$p$ crea un file con le prime $< p >$ righe del file hostlist. Serve a far utilizzare solo i primi $< p >$ host.

Capitolo 4

Analisi delle prestazioni

Sommario

4.1 Premesse

Il programma è stato testato con set di numeri generati casualmente senza segno e di massimo 2^{32} bit. La dimensione dei file di test è minore o uguale a $100mb$. Ogni test è stato ripetuto 5 volte e i risultati riportati sotto sono la media dei tempi di queste 5 esecuzioni calcolate in millisecondi. I tempi per la lettura del file e la distribuzione dell'input sono stati esclusi dal conteggio finale. Per la strategia 3 è stata effettuata la media dei tempi di esecuzione di tutti i processori. Il codice sorgente completo dei test è reperibile sulla pagina GitHub del progetto a questo Link.

4.2 Algoritmo non parallelo

L'algoritmo non parallelo è stato testato settando -np e strategy a 1. La scelta della strategia è ininfluente dato che settando il numero di processori a 1 vengono evitate le sezioni di comunicazione. Non è stato necessario creare un algoritmo sequenziale ad hoc dato che le istruzioni

in più svolte dall'algoritmo parallelo sono così poche da non influire sul tempo di esecuzione totale

input	100	1000	10.000	100.000	1.000.000	10.000.000
millisecondi	0.0031	0,0075	0.0534	0.4388	4.4230	44.1731

4.3 2 Processori

input	100	1000	10.000	100.000	1.000.000	10.000.000
strategia 1	0.0160	0.0174	0.0468	0.2612	2.2342	22.2137
strategia 2	0.0159	0.0238	0.0481	0.2530	2.2944	22.2626
strategia 3	0.0182	0.0202	0.0445	0.2382	2.2422	22.2378

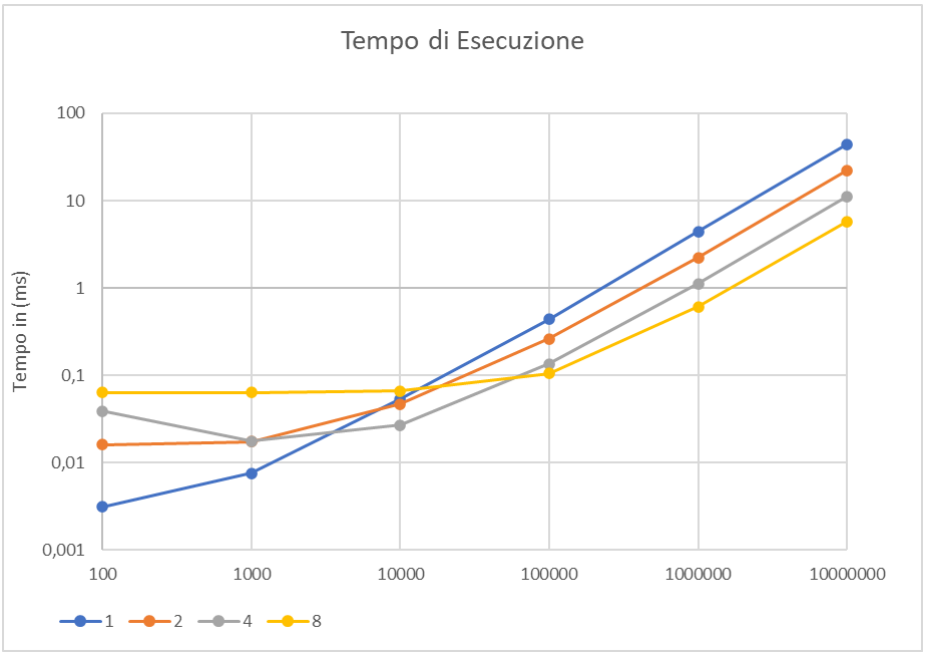
4.4 4 processori

input	100	1000	10.000	100.000	1.000.000	10.000.000
strategia 1	0.0388	0.0175	0.0268	0.1348	1.1251	11.1660
strategia 2	0.0288	0.0244	0.0421	0.1471	1.1304	11.5540
strategia 3	0.0303	0.0276	0.0347	0.1381	1.1759	11.1750

4.5 8 processori

input	100	1000	10.000	100.000	1.000.000	10.000.000
strategia 1	0.0637	0.0631	0.0659	0.1043	0.6131	5.7989
strategia 2	0.0413	0.0417	0.0427	0.0893	0.5860	5.6342
strategia 3	0.0345	0.0342	0.0385	0.0910	0.5902	5.6876

4.6 Considerazioni



Dal grafico si nota che l'algoritmo sequenziale è il più veloce per piccoli input seguito in ordine da quello a 2, 4 e 8 processori. Ciò perchè all'aumentare dei processori aumentano anche le istruzioni di comunicazioni che rallentano l'esecuzione. Al crescere dell'input e fissato il numero di processori il numero di istruzioni di cominicazione rimane costante. Ciò significa che per input grandi le istruzioni di comunicazione influiscono poco sul tempo di esecuzione totale. In questo caso dai 100.000 numeri in su l'utilizzo di più processori concede un guadagno enorme in termini di tempo.

La scelta della strategia è risultata ininfluente per quasi tutti i casi. Ciò perchè il numero di processori utilizzati è molto piccolo, probabilmente aumentando il numero di processori la strategia 2 potrebbe portare ad un reale guadagno rispetto alla strategia 1. In conclusione la strategia 1 è da preferire per un numero piccolo di processori data la semplicità di implementazione.

4.7 Speedup ed Efficienza

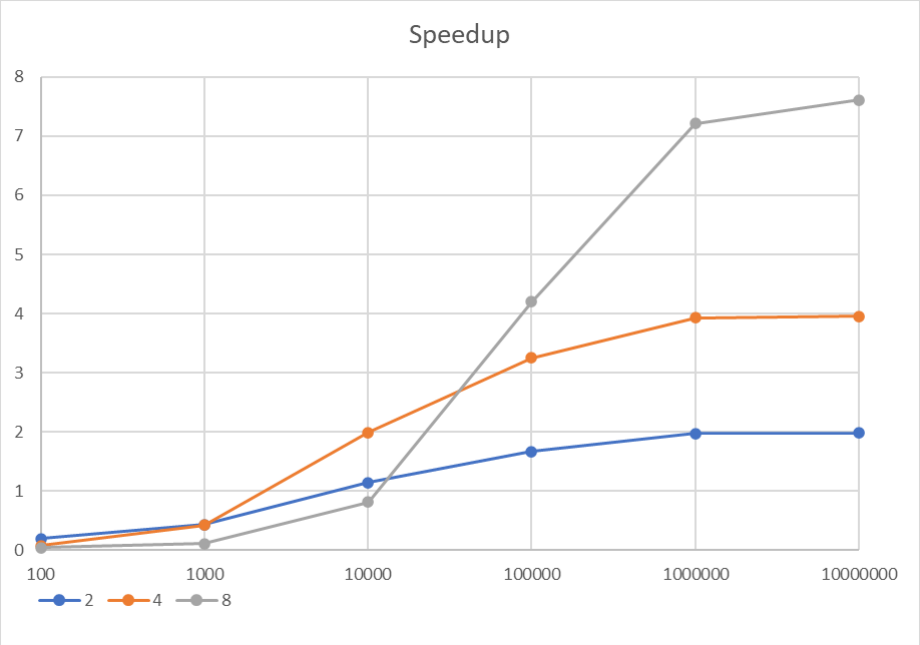
Siano Speedup ed Efficienza definiti come segue

$$S(p) = \frac{T(1)}{T(p)} \qquad E(p) = \frac{S(p)}{p}$$

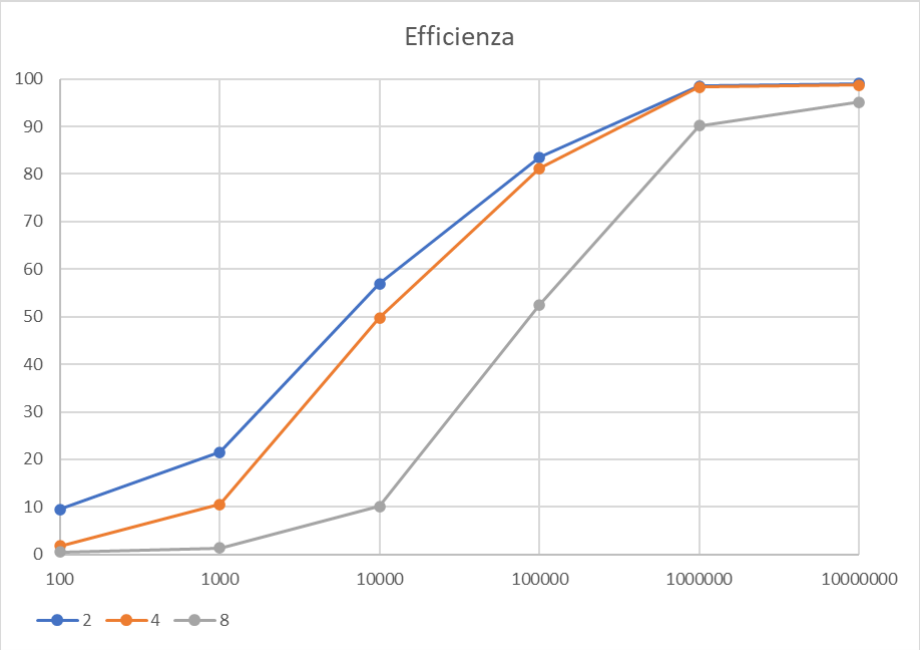
Applicando le equazioni ai dati riportati nelle tabelle si ottengono gli Speedup e le Efficienze delle varie esecuzioni. Per il calcolo sono stati utilizzati i tempi della strategia 1

input	100	1000	10.000	100.000	1.000.000	10.000.000
$S(2)$	0.19	0.43	1.14	1.67	1.97	1.98
$E(2)$	9.5%	21.5%	57%	83.5%	98.5%	99%
$S(4)$	0.07	0.42	1.99	3.25	3.93	3.95
$E(4)$	1.7%	10.5%	49.7%	81.2%	98.2%	98.7%
$S(8)$	0.04	0.11	0.81	4.20	7.21	7.61
$E(8)$	0.5%	1.3%	10.1%	52.5%	90.1%	95.1%

Di seguito i grafici con i dati delle tabelle:



Per input inferiori a 10.000 lo Speedup è inferiore a 1 per tutti i processori, ciò indica che le prestazioni sono inferiori all'algoritmo sequenziale. Per input dai 10.000 in su lo Speedup aumenta fino a rasentare lo Speedup ideale.



L'efficienza dell'algoritmo a 2 e 4 processori è molto simile, crescono entrambe fino a sfiorare il 100%. L'algoritmo con 8 processori anche se è il più veloce è molto meno efficiente rispetto agli altri per input medi, mentre per input grandi anch'esso sfiora il 100%.

4.8 Conclusione

Come si può verificare dai grafici l'utilizzo di un algoritmo parallelo per la risoluzione del problema ha realmente giovato ai tempi di esecuzione. Si è visto che all'aumentare dell'input Speedup ed Efficienza raggiungono asintoticamente i valori ideali. Ciò vuol dire che l'algoritmo ad 8 processori è circa 8 volte più veloce di quello ad 1 (per input adeguati), ossia il risultato che si spera di ottenere quando si progetta un algoritmo parallelo.

Appendice A

Codice completo con Documentazione interna

```c
/**
 *  Autore: Matteo Richard Gaudino
 *  Matricola: N86003226
 * **/

#include <mpi.h> // Recv, Send, ...
#include <stdio.h> // std in/out/err
#include <stdlib.h> // malloc, atoi
#include <time.h> // Per la generazione di numeri random
#include <math.h> // log2
#include <unistd.h> // getopt, optarg

/**
 * Utilizzo:
 *  mpiexec -machinefile <hostFile> -np <p> sommaPar <...>
 *
 * Sistassi
 *  sommaPar [-s strategy] [-o output_rank] -f filename
 *  sommaPar [-s strategy] [-o output_rank] -r numToSum
 *  sommaPar [-s strategy] [-o output_rank] -n numToSum num_1 ... num_n
 *
 * Semantica
 *
```

```
24   *  -s stabilisce la strategia da utilizzare 1, 2 o 3.
25   *     Nel caso il numero di processori non sia una potenza di 2 verrà
     ↪  utilizzata la strategia 1.
26   *     Il valore di default è 1. OPZIONALE
27   *
28   *  -o Stabilisce quale processo deve scrivere i risultati su stdout.
29   *     Se il valore è -1 tutti i processi scriveranno i propri risultati.
30   *     Il valore di default è 0. OPZIONALE
31   *
32   *  -f Il programma dovrà prendere i numeri da sommare in un file di input
33   *
34   *  -r Il programma dovrà generare <numToSum> numeri random e sommarli
35   *
36   *  -n Il programma prenderà in input <numToSum> numeri da sommare da riga di
     ↪  comando.
37   *     I umeri devono essere separati da uno spazio vuoto
38   *
39   *  Attenzione! -f -r -n sono mutuamente esclusivi. L'ordine dei flag non è
     ↪  rilevante
40   *
41   * **/
42
43  // Stabilisce che Strategia di input è stata scelta
44  enum INPUT_STRATEGY{
45      FILE_STRAT, // se usato -f
46      RANDOM_STRAT, // se usato -r
47      ARG_STRAT // se usato -n
48  };
49
50  int main(int argc, char** argv){
51
52      MPI_Init(&argc, &argv);
53
54      int worldSize, rank; // dimensione di MPI_COMM_WORLD e rank nel
        ↪  communicator
55      MPI_Status status; // Per MPI_Recv
56
```

```
57      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
58      MPI_Comm_size(MPI_COMM_WORLD, &worldSize);
59
60      int commSteps = log2(worldSize); // per strategie 2 e 3
61      unsigned long long pow2 = 1; // potenze di 2 per la strategia 2 e 3
62  // ---------------------- Lettura input ----------------------
63
64      int nums_c = 0; // dimensione del vettore
65      int* nums_v; // vettore da sommare
66
67      int strategy = 1; // strategia per la comunicazione
68      int outputRank = 0; // Processo che stampa l'output
69
70      if(rank == 0){
71          enum INPUT_STRATEGY in_strat; // strategia di input
72          FILE* testFile; // per -f
73          int opindex; // per -n
74
75          int op;
76          // Parsing dell'input
77          while ((op = getopt(argc, argv, "s:o:f:r:n:")) != -1){
78              switch (op){
79              case 's':
80                  strategy = atoi(optarg);
81              break;
82              case 'o':
83                  outputRank = atoi(optarg);
84              break;
85
86              case 'f':
87                  testFile = fopen(optarg, "r"); // Apre il file di input in
                    ↪  sola lettura
88                  in_strat = FILE_STRAT;
89              break;
90
91              case 'r':
92                  nums_c = atoi(optarg);
```

```
93                  in_strat = RANDOM_STRAT;
94              break;
95
96              case 'n':
97                  nums_c = atoi(optarg);
98                  in_strat = ARG_STRAT;
99                  opindex = optind;
100             break;
101             default:
102                 perror("Wrong input, read the Documentation.\n");
103                 MPI_Abort(MPI_COMM_WORLD, -1);
104                 break;
105             }
106         }
107
108         switch (in_strat){
109             case FILE_STRAT: // legge l'input da file
110                 fscanf(testFile, "%d" ,&nums_c);
111
112                 nums_v = malloc(sizeof(int)*nums_c);
113                 for (int i = 0; i < nums_c; i++){
114                     fscanf(testFile, "%d", &(nums_v[i]));
115                 }
116                 fclose(testFile);
117             break;
118             case RANDOM_STRAT:{
119                 srand(time(NULL)); // Genera numeri casuali
120                 unsigned long long max = 1UL << 32; // di massimo 32 bit
121
122                 nums_v = malloc(sizeof(int)*nums_c);
123
124                 for (int i = 0; i < nums_c; i++){
125                     nums_v[i] = rand()%max;
126                 }
127             }
128             break;
129             case ARG_STRAT:{
```

```
130                     nums_v = malloc(sizeof(int)*nums_c);
131
132                     int i = 0;
133                     while (opindex < argc && *argv[opindex] != '-'){ // Legge
                        ↪  dagli argomenti fichè non terminano
134                         nums_v[i++] = atoi(argv[opindex++]); // Un input
                            ↪  sbagliato porta a segmentation fault
135                     }
136                 }
137             break;
138         }
139
140         if((worldSize & (worldSize - 1)) != 0 && strategy != 1) { // Verifica
            ↪  se il numero dei processori è una potenza di 2
141             strategy = 1; // se non lo è si setta la strategia a 1
142             printf("Warning strategy is not a power of 2, strategy is set to
                ↪  1\n");
143         }
144     }
145
146 // ---------------------- Distribuzione input ----------------------
147
148     MPI_Bcast(&nums_c, 1, MPI_INT, 0, MPI_COMM_WORLD);
149     MPI_Bcast(&strategy, 1, MPI_INT, 0, MPI_COMM_WORLD);
150     MPI_Bcast(&outputRank, 1, MPI_INT, 0, MPI_COMM_WORLD);
151
152     int rest = nums_c % worldSize;
153     int numToSum = nums_c/worldSize + ((rank < rest)? 1: 0); // numeri da
        ↪  sommare per ogni processo
154
155     if(rank == 0){
156         int tmp = nums_c/worldSize; // Minimo numeri da sommare
157         int cursor = numToSum;
158
159         for (int i = 1; i < worldSize; ++i){
160             int inumToSum = tmp + ((i < rest)? 1: 0); // aggiunge 1 se i <
                ↪  rest
```

```
161             MPI_Send(nums_v + cursor, inumToSum, MPI_INT, i, 0,
                ↪  MPI_COMM_WORLD);
162
163             cursor += inumToSum;
164         }
165     } else{
166         nums_v = malloc(sizeof(int)*numToSum); // alloca spazio e riceve da p0
167         MPI_Recv(nums_v, numToSum, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
168     }
169
170
171     double t0, t1; // Per misurare i tempi
172     MPI_Barrier(MPI_COMM_WORLD);
173     t0 = MPI_Wtime(); // è partita la misurazione
174
175
176 // ---------------------- Calcolo ----------------------
177
178     long long sum = 0;
179     for (int i = 0; i < numToSum; i++){
180         sum += nums_v[i];
181     }
182
183 /* ---------------------- Soluzione 1 ---------------------- */
184     if(strategy == 1){
185         if (rank != 0){ // se non è p0 invia
186             MPI_Send(&sum, 1, MPI_LONG_LONG, 0, 0, MPI_COMM_WORLD);
187         } else{ // altrimenti ricevi e somma
188             for (int i = 1; i < worldSize; i++){
189                 long long sum2;
190                 MPI_Recv(&sum2, 1, MPI_LONG_LONG, i, 0, MPI_COMM_WORLD,
                    ↪  &status);
191
192                 sum += sum2;
193             }
194         }
195     }
```

```
196 /* ---------------------- Soluzione 2 ---------------------- */
197     else if(strategy == 2){
198         for(int i = 0; i < commSteps; ++i){ // ripeti log2(p) volte
199             if((rank % pow2) == 0){ // se partecipa alla comunicazione
200                 if((rank % (pow2 << 1)) == 0){ // se è un ricevente
201                     long long sum2;
202                     MPI_Recv(&sum2, 1, MPI_LONG_LONG, rank + pow2, 0,
                        ↪ MPI_COMM_WORLD, &status);
203                     sum += sum2;
204                 } else{ // altrimenti se è un mittente
205                     MPI_Send(&sum, 1, MPI_LONG_LONG, rank - pow2, 0,
                        ↪ MPI_COMM_WORLD);
206                 }
207                 pow2 <<= 1; // pow2 = pow2 * 2
208             }
209         }
210     }
211 /* ---------------------- Soluzione 3 ---------------------- */
212     else if(strategy == 3){
213         long long sum2;
214         for(int i = 0; i < commSteps; ++i){ // come la strategia II, ma
            ↪ partecipano tutti
215             if((rank % (pow2 << 1)) < pow2){
216                 MPI_Send(&sum, 1, MPI_LONG_LONG, rank + pow2, 0,
                    ↪ MPI_COMM_WORLD);
217                 MPI_Recv(&sum2, 1, MPI_LONG_LONG, rank + pow2, 0,
                    ↪ MPI_COMM_WORLD, &status);
218             } else{
219                 MPI_Recv(&sum2, 1, MPI_LONG_LONG, rank - pow2, 0,
                    ↪ MPI_COMM_WORLD, &status);
220                 MPI_Send(&sum, 1, MPI_LONG_LONG, rank - pow2, 0,
                    ↪ MPI_COMM_WORLD);
221
222             }
223             sum += sum2;
224             pow2 <<= 1;
225         }
```

```
226         }
227
228     /* ----------------------- Comunicazione dei risultati -----------------------
        ↪  */
229         t1 = MPI_Wtime(); // fine della misurazione
230         MPI_Barrier(MPI_COMM_WORLD);
231
232         double localTime = t1-t0;
233         double maxTime;
234
235         MPI_Reduce(&localTime, &maxTime, 1, MPI_DOUBLE, MPI_MAX, 0,
            ↪  MPI_COMM_WORLD); // calcola il tempo massimo impiegato
236         if(rank == 0) localTime = maxTime;
237
238         if (outputRank == -1){ // stampano tutti
239             printf("[%d] %lld %.10f\n", rank, sum, localTime);
240         } else if (rank == outputRank){ // stampa solo il processo selezionato
241             printf("[%d] %lld %.10f\n", rank, sum, localTime);
242         }
243
244         MPI_Finalize();
245         return 0;
246     }
```

Appendice B

File PBS con Test

```
#!/bin/bash
############################
#
#
# The PBS directives #
#
#
############################
#PBS -q studenti
#PBS -l nodes=8:ppn=8
#PBS -N progetto1
#PBS -o progetto1.out
#PBS -e progetto1.err
########################################
# -q coda su cui va eseguito il job #
# -l numero di nodi richiesti #
# -N nome job(stesso del file pbs) #
# -o, -e nome files contenente l'output #
########################################
#
#
# qualche informazione sul job #
#
#
########################################
echo -----------------------------------------------------
```

```
27 echo PBS: qsub is running on $PBS_O_HOST
28 echo PBS: originating queue is $PBS_O_QUEUE
29 echo PBS: executing queue is $PBS_QUEUE
30 echo PBS: working directory is $PBS_O_WORKDIR
31 echo PBS: execution mode is $PBS_ENVIRONMENT
32 echo PBS: job identifier is $PBS_JOBID
33 echo PBS: job name is $PBS_JOBNAME
34 echo PBS: node file is $PBS_NODEFILE
35 echo PBS: number of nodes is $NNODES
36 echo PBS: current home directory is $PBS_O_HOME
37 echo PBS: PATH = $PBS_O_PATH
38 echo ------------------------------------------------------
39 echo
40 echo Job reserved nodes:
41 cat $PBS_NODEFILE
42 echo
43 echo ------------------------------------------------------
44 echo
45 sort -u $PBS_NODEFILE > hostlist
46 head -1 hostlist > host1
47 NCPU=$(wc -l < hostlist)
48 cat hostlist
49 echo
50 echo ------------------------------------------------------
51 echo
52 PBS_O_WORKDIR=$PBS_O_HOME/CPD
53
54 /usr/lib64/openmpi/1.4-gcc/bin/mpicc -std=gnu99 $PBS_O_WORKDIR/sommaPar.c -o
   ↪ $PBS_O_WORKDIR/sommaPar -lm
55
56 echo compilation terminated
57 echo ------------------------------------------------------
58
59 echo Start testing
60 echo "-------------- test 1 processor --------------"
61
62 head -1 hostlist > host1
```

```
63 for f in $(ls $PBS_O_WORKDIR/tests)
64 do
65     echo Test: $f
66     for i in {1..5}
67     do
68         res=$(/usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile host1 -np 1
           ↪  $PBS_O_WORKDIR/sommaPar -f $PBS_O_WORKDIR/tests/$f)
69         echo $res >> $PBS_O_WORKDIR/results/p1$f
70     done
71 done
72
73
74 for p in 2 4 8
75 do
76    head -$p hostlist > host$p
77     echo "-------------- test $p processor --------------"
78     for strat in 1 2 3
79     do
80         for f in $(ls $PBS_O_WORKDIR/tests)
81         do
82             echo Strategy: $strat, test: $f
83             for i in {1..5}
84             do
85                 res=$(/usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile
                   ↪  host$p -np $p $PBS_O_WORKDIR/sommaPar -s $strat -f
                   ↪  $PBS_O_WORKDIR/tests/$f)
86                 echo $res >> $PBS_O_WORKDIR/results/p${p}s${strat}$f
87             done
88         done
89     done
90 done
```