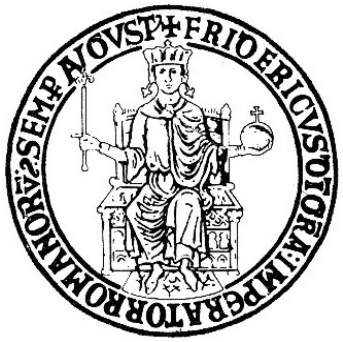
Università degli Studi di Napoli Federico II
Scuola Politecnica e delle Scienze di Base
Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione



Corso di Laurea Magistrale in Informatica
Insegnamento di Parallel and Distributed Computing
Anno accademico 2020/2021

Sviluppo di un algoritmo per la somma di N numeri reali in ambiente di calcolo parallelo su architettura MIMD a memoria distribuita

Autori:
Salvatore Bazzicalupo, N97000345
Annarita Della Rocca, N97000341

Docenti:
Prof. Giuliano Laccetti
Dott.sa Valeria Mele

Questa pagina è stata lasciata intenzionalmente bianca.

Indice

Capitolo 1

Definizione ed analisi del problema

Si vuole sviluppare un algoritmo per il calcolo della somma di N numeri reali, in ambiente di calcolo parallelo su architettura MIMD (Multiple Instruction stream Multiple Data) a memoria distribuita, utilizzando la libreria MPI.

L'algoritmo è composto da due parti principali: nella prima parte vi è la raccolta dei parametri in ingresso che sono il numero di valori da sommare, strategia da adottare per la somma, id del processore che dovrà stampare ed eventualmente i valori in ingresso da sommare. Nel dettaglio queste operazioni vengono svolte attraverso un opportuno script.

La seconda parte, performata mediante un algoritmo in C, prevede la raccolta e la distribuzione dei valori da sommare tra i vari processori. In questa porzione del programma vengono implementate le tre strategie menzionate in precedenza.

Un aspetto importante da tenere in considerazione è lo sviluppo in ambiente di calcolo parallelo che porta ad un approccio diverso non solo per la risoluzione, ma anche per le problematiche affrontate.

Di tali problematiche si rammenta della traslazione dell'albero attuata nel caso in cui venga fornito in ingresso un ID processor diverso da 0 nella seconda strategia. In tal caso i processori devono comunicare in ordine differente.

Le strategie verranno esplicitate nel dettaglio nel capitolo successivo.

Capitolo 2

Descrizione algoritmo

L'algoritmo riceve alcuni parametri in input:

- il numero N di valori da sommare
  - se $N \leq 20$ l'algoritmo leggerà i numeri dal file chiamato `input.txt`
  - se $N > 20$ verranno generati N numeri casuali compresi tra 0 e 1000
- la strategia da utilizzare, un numero compreso tra 1 e 3
- l'identificativo ID del processore che dovrà effettuare la somma e stampare il risultato; oppure -1 per poter permettere a tutti i processori di stampare.
  Nel caso delle strategie 1 e 2 sarà solo il processore 0 a stampare.

2.1 Job-Script PBS

L'algoritmo utilizza uno script PBS per ricevere i parametri in input specificati nella sezione precedente, nello script è presente la seguente porzione di codice:

```
1 ######################
2 ## CUSTOM VALUES ##
3 ######################
4 # Il numero di valori da sommare
5 NUMBERS_TO_SUM=10
6
7 # La strategia da applicare, valore compreso tra 1 e 3
8 STRATEGY=3
9
```

```
10 # Il processore che effettua i calcoli e stampa il risultato, -1 per
       far stampare a tutti nella strategia 3
11 PRINTER_ID=-1
12 ######################
```

Dove queste variabili hanno il seguente significato:

- `NUMBERS_TO_SUM` - quanti valori dovranno essere sommati. Come descritto nel capitolo 1: se maggiore di 20 saranno numeri generati in maniera casuale;
- `STRATEGY` - la strategia da utilizzare. Solo i valori 1, 2 o 3 sono consentiti;
- `PRINTER_ID` quale processore deve effettuare i calcoli e stampare il risultato;

Questo script si serve di un file input.txt che dovrà contenere i numeri reali da sommare separati da uno spazio. L'assenza di questo file può causare il lancio di un errore dallo script PBS.

Esempio di file `input.txt`:

`input.txt`

```
3.1415 1.0101 6.9999 42
```

2.2 Algoritmo in C

L'algoritmo prevede, oltre alle operazioni standard della libreria MPI, una fase di controlli di consistenza sui parametri passati in input.
Nel dettaglio viene controllato che:

- la strategia sia una delle tre possibili;
- il processore che dovrà fare i calcoli, chiamato `masterId`, sia un valore valido (compreso tra 0 ed il numero di processori scelto).
- Nel caso in cui il `masterId` sia -1, e cioè nel caso in cui è richiesta la stampa da parte di tutti i processori, ma la strategia adottata non sia la numero 3, tale valore verrà automaticamente mutato in 0 in quanto gli altri processori non contengono, né hanno modo di conoscere, la somma totale.

Dopo aver appurato che i valori in ingresso siano validi, viene richiamato il metodo `distributeNumbersAndGetPartialSum`.
Tale metodo si occupa di distribuire, ricevere e calcolare la somma del singolo processore.

Segue una porzione di codice contenente solo le parti ritenute salienti del metodo, quelle meno interessanti saranno riassunte tramite commenti.

```
1  // Numero dei valori da distribuire per ogni processore
2  int amountOfNumbers = inputSize / numberOfProcessors;
3
4  // Numeri non multipli del numero di processori da ripartire
5  int rest = inputSize % numberOfProcessors;
6
7  // Alcuni processori sommeranno più di un numero
8  if(processorId < rest)
9     amountOfNumbers++;
10
11 // Il processore 0 distribuisce i numeri
12 if(processorId == 0) {
13
14    if(inputSize <= 20) { /* Leggi da riga di comando i valori */ }
15    else { /* Genera valori casuali */ }
16
17    tmp = amountOfNumbers;
18    sentNumbers = 0;
19    /* Invia i valori a tutti i processori */
20    for (i = 1; i < numberOfProcessors; ++i) {
21        sentNumbers += tmp;
22        tag = TAG_START + i;
23
24        if (i == rest)
25           tmp--;
26
27        MPI_Send(&numbers[sentNumbers], tmp, MPI_FLOAT, i, tag,
      MPI_COMM_WORLD);
28    }
29 } else { // Riceve i valori da tutti i processori
30    tag = TAG_START + processorId;
```

```
31      MPI_Recv(numbers, amountOfNumbers, MPI_FLOAT, 0, tag,
        MPI_COMM_WORLD, &status);
32  }
33
34  // Effettua la propria somma parziale
35  for (i = 0; i < amountOfNumbers; ++i)
36      sum += numbers[i];
37
38  return sum;
```

Successivamente viene selezionata la singola strategia in base al valore in ingresso.

2.3 Strategie per la somma

L'algoritmo è suddiviso in ulteriori tre funzioni, una per ogni strategia che è possibile adottare.

La prima strategia si basa su un protocollo che prevede l'assunzione di un processore come "master", ovvero colui il quale si occupa di effettuare la somma totale dopo aver ricevuto le somme parziali dagli altri processori (slave).

Segue l'implementazione dei punti salienti con opportuni commenti che descrivono le scelte implementative effettuate:

```
1   float strategyOne(float sum, int processorId, int numberOfProcessors
        , int masterId) {
2
3       /* Se il processore è il masterId, procede a ricevere
4        * tutte le somme parziali */
5       if(processorId == masterId) {
6
7           /* Cicla i vari processori da cui ricevere i dati */
8           for (i = 0; i < numberOfProcessors; i++) {
9
10              /* Siccome non può ricevere da se stesso, salta l'
        iterazione */
11              if(masterId == i)
12                  continue;
```

```
13              /* Riceve la somma parziale e la somma a quella corrente
    */
14              tag = TAG_START + i;
15              MPI_Recv(&partialSum, 1, MPI_FLOAT, i, tag,
    MPI_COMM_WORLD, &status);
16              sum += partialSum;
17          }
18      } else {
19          /* Invia la propria somma al masterId */
20          tag = TAG_START + processorId;
21          MPI_Send(&sum, 1, MPI_FLOAT, masterId, tag, MPI_COMM_WORLD);
22      }
23
24      return sum;
25  }
```

Questa strategia non prevede alcun prerequisito, a differenza delle altre due.

La seconda strategia si basa sul concetto di alberi binari. Richiede quindi che il numero di processori sia una potenza di due; se tale condizione non è rispettata la strategia risulta non applicabile e di default verrà applicata la precedente.

Segue l'implementazione dei punti salienti con opportuni commenti che descrivono le scelte implementative effettuate:

```
1   float strategyTwo(float sum, int processorId, int numberOfProcessors
    , int masterId) {
2
3       /* L'id shiftato tenendo conto del valore di masterId */
4       int logicId = processorId + (numberOfProcessors - masterId);
5       logicId = logicId % numberOfProcessors;
6
7       /* Ciclo sull'altezza dell'albero */
8       for (i = 0; i < (int) log2(numberOfProcessors); ++i) {
9           tag = TAG_START + i;
10          if((logicId % (int) pow(2, i)) == 0) {
11              if((logicId % (int) pow(2, i + 1)) == 0) {
12                  /* Calcola l'id del vero processore a cui è
    necessario inviare i
13                   * la somma parziale */
```

```
14                int senderId = ((int) (processorId + pow(2, i)) %
    numberOfProcessors);
15                MPI_Recv(&partialSum, 1, MPI_FLOAT, senderId, tag,
    MPI_COMM_WORLD, &status);
16                sum += partialSum;
17            } else {
18                /* Ricevo il valore dal processore precedente, se < 0
     deve essere
19                 * reso positivo sommando numberOfProcessors */
20                int receiverId = processorId - (int) pow(2, i);
21                if(receiverId < 0) {
22                    receiverId += numberOfProcessors;
23                }
24                MPI_Send(&sum, 1, MPI_FLOAT, receiverId, tag,
    MPI_COMM_WORLD);
25            }
26        }
27    }
28    return sum;
29 }
```

Per questa strategia è stato utilizzato un protocollo che prevede la definizione di un id logico, quest'ultimo è l'id shiftato del valore selezionato come "Master processor", così facendo, tramite il modulo, il processo selezionato avrà id 0 mentre quelli antecedenti saranno shiftati a destra.

Per poter procedere all'applicazione di questa strategia dobbiamo costruire un albero, al fine di poterlo fare è necessario che il numero dei processori sia potenza di due. Dovendo fare tanti passi quanti sono i livelli dell'albero, ad ogni passo il numero dei processi che partecipano si dimezza. Per il primo passo tutti i processi riceveranno i dati. Dalla seconda iterazione in poi i processi verranno discriminati tra: quelli che ricevono, cioè quelli che parteciperanno all'iterazione successiva, e quelli che inviano, cioè quelli che non prenderanno parte all'iterazione successiva.

La terza strategia si basa anch'essa sul concetto di alberi binari con il medesimo vincolo della precedente; la differenza principale è che rispetto al far comunicare due processori alla volta, nella terza tutti i processori comunicano tra loro simultaneamente,

a fine esecuzione tutti contengono la somma totale.

Segue l'implementazione dei punti salienti con opportuni commenti che descrivono le scelte implementative effettuate:

```
1  float strategyThree(float sum, int processorId, int
       numberOfProcessors) {
2      /* Ciclo sull'altezza dell'albero */
3      for (i = 0; i < (int) log2(numberOfProcessors); ++i) {
4          tag = TAG_START + i;
5          if((processorId % (int) pow(2, i + 1)) < (int) pow(2, i)) {
6              int otherId = processorId + (int) pow(2, i);
7              MPI_Send(&sum, 1, MPI_FLOAT, otherId, tag, MPI_COMM_WORLD
       );
8              MPI_Recv(&partialSum, 1, MPI_FLOAT, otherId, tag,
       MPI_COMM_WORLD, &status);
9          } else {
10             int otherId = processorId - (int) pow(2, i);
11             MPI_Send(&sum, 1, MPI_FLOAT, otherId, tag, MPI_COMM_WORLD
       );
12             MPI_Recv(&partialSum, 1, MPI_FLOAT, otherId, tag,
       MPI_COMM_WORLD, &status);
13         }
14         sum += partialSum;
15     }
16
17     return sum;
18 }
```

Così come nella seconda strategia, anche nella terza è necessario che il numero di processi sia una potenza di due. Ogni processo riceve e spedisce sempre da/a processi diversi. È necessario quindi conoscere da chi ricevere e a chi spedire; ad ogni passo l'intervallo a cui invio e da cui ricevo va via via allargandosi.

2.4 Scelta della strategia

La scelta della strategia viene effettuata prendendo in input l'intero ad esse associate. Se il numero fornito in ingresso non è uno tra 1,2 e 3 esso risulta essere non valido.

In tal caso il programma provvederà a stampare un messaggio d'errore notificando l'utente del fatto che la strategia scelta risulti essere non valida.
Se la strategia scelta è una tra la seconda e la terza ma esse risultino essere non applicabili per via del vincolo sul numero di nodi (che devono essere potenza di due), allora l'utente verrà notificato dell'impossibilità di applicare tale strategia e di default verrà lanciata la numero uno.

Capitolo 3

Input ed Output

L'algoritmo opera su numeri reali; richiede i seguenti valori da riga di comando:

- Il numero N di valori da sommare;
- gli N numeri, se e solo se $N \leq 20$;
- la strategia da utilizzare, un numero compreso tra 1 e 3;
- l'identificativo ID del processore che dovrà effettuare la somma e stampare il risultato; oppure -1 per poter permettere a tutti i processori di stampare il risultato.
  Nel caso delle strategie 1 e 2 sarà solo il processore 0 a stampare.

L'output dell'algoritmo, è della seguente forma:

```
[Completed] Processor ID <id-selezionato>, total sum: <risultato>
Time elapsed: <ammontare-secondi> seconds
```

I valori tra parentesi angolari corrispondono a:

- `<id-selezionato>`: l'id del processore che si occupa di eseguire i calcoli e stampare il risultato ottenuto;
- `<risultato>`: il risultato della somma degli N valori;
- `<ammontare-secondi>`: il tempo totale impiegato dai processori per calcolare la somma richiesta.

3.1 Input/Output per valori definiti dall'utente

Supponendo di voler utilizzare la strategia 2, far stampare al processore 3 e sommare 10 numeri, si dovrà scrivere un input della seguente forma:

```
~/opt/usr/local/bin/mpiexec -np 4 ./main 10 2.5 2.5 2 1.5 1.5 10 10
   10 1 1 2 3
```

Si riceverà il seguente ouput:

```
[Completed] Processor ID 3, total sum: 42.000000
Time elapsed: 4.400000e-05 seconds
```

Si noti che il tempo impiegato per la somma può differire in base a vari fattori, tra cui il numero di processori, la potenza di calcolo dei processori, e così via.

3.2 Input/Output per valori casuali

Supponendo di voler utilizzare la strategia 1, far stampare al processore 2 e sommare 45 numeri, si dovrà scrivere un input della seguente forma:

```
~/opt/usr/local/bin/mpiexec -np 4 ./main 45 2 3
```

Si riceverà il seguente ouput:

```
[Completed] Processor ID 3, total sum: 250.767639
Time elapsed: 4.800000e-05 seconds
```

Si noti che il tempo impiegato per la somma può differire in base a vari fattori, tra cui il numero di processori, la potenza di calcolo dei processori, e così via. Inoltre il calcolo della somma potrà differire in base ai numeri casualmente generati.

3.3 Input/Output attraverso lo script PBS

È possibile personalizzare i valori in input attraverso lo script PBS.
Al suo interno sono contenute delle variabili che vanno modificate opportunamente, come descritto in precedenza. Si rimanda al capitolo 2.1 per ulteriori dettagli.

Supponendo di avere il seguente file `input.txt`

input.txt

```
3.1415 1.0101 6.9999 42
```

E la seguente configurazione:

```
1  ######################
2  ## CUSTOM VALUES ##
3  ######################
4  # Il numero di valori da sommare
5  NUMBERS_TO_SUM=4
6
7  # La strategia da applicare, valore compreso tra 1 e 3
8  STRATEGY=1
9
10 # Il processore che effettua i calcoli e stampa il risultato, -1 per
       far stampare a tutti nella strategia 3
11 PRINTER_ID=3
12 ######################
```

Avviando l'algoritmo utilizzando:

```
qsub sum-job-script.pbs
```

Si otterrà il seguente output:

```
[Completed] Processor ID 3, total sum: 53.151501
Time elapsed: 2.300000e-05 seconds
```

Anche in questo caso, come i precedenti, il tempo impiegato potrà variare.

Capitolo 4

Indicatori di errore

Tutti i messaggi di errore dell'applicativo seguono la seguente forma:
"ERROR - <descrizione-errore>: <parametri>"

Con i valori tra parentesi angolari:

- `<descrizione-errore>`: una sintetica descrizione dell'errore verificatosi;
- `<parametri>`: una lista della forma chiave:valore dei parametri che hanno causato l'errore.

In `main`:

```
1  /* Verifica che strategy e masterId siano interi */
2  if(!parseInt(argv[argc - 2], &strategy) || !parseInt(argv[argc - 1],
       &masterId)) {
3      fprintf(stderr, "ERROR - Cannot parse the masterId or strategy
       with values provided.\nstrategy:%s\nprinter:%s\n\n", argv[argc -
       2], argv[argc - 1]);
4      return 1;
5  }
6
7  /* Verifica che strategy sia compreso tra 1 e 3 */
8  if(strategy < 0 || strategy > 3) {
9      fprintf(stderr, "ERROR - Strategy number not allowed, must be a
       value between 1 and 3.\nstrategy:%d\n", strategy);
10     return 1;
11 }
```

In `distributeNumbersAndGetPartialSum`:

```c
/* Checks if number of values is correct */
if(!parseInt(argv[1], &inputSize) || inputSize < 0) {
    fprintf(stderr, "ERROR - Input size not allowed, must be a value between 0 and INT_MAX\ninputSize:%s\n", argv[1]);
    return 1;
}

/* Checks if the single value to sum is correct */
for (i = 0; i < inputSize; i++) {
    if(!parseFloat(argv[k++], &numbers[i])) {
        fprintf(stderr, "ERROR - Cannot parse a value to sum.\nvalue [%d]: %s\n", i, argv[k-1]);
        return 1;
    }
}
```

Il job-script può lanciare a sua volta il seguente errore se il file `input.txt` non esiste:

```
# Verifica l'esistenza del file
FILEPATH=$PBS_O_WORKDIR/input.txt
if [ -f $FILEPATH ]; then
   NUMBERS="$(cat $FILEPATH)"
else
   echo "[Job-Script] ERROR: File input.txt is missing"
   exit 1;
fi
```

Capitolo 5

Subroutine

Nell'algoritmo sono presenti varie subroutine utilizzate, queste sono documentate a livello interno del codice tramite commenti che hanno la seguente forma:

- breve descrizione del metodo;
- lista dei parametri con descrizione dettagliata del loro utilizzo;
- se presente un output, a seconda delle diverse condizioni possibili, viene descritta la sua forma.

Sono quindi riportate le firme dei metodi, la loro documentazione interna ed eventualmente una descrizione aggiuntiva.

5.1 Subroutines personalizzate

Di seguito, verranno descritte alcune subroutine personalizzate utilizzate nel progetto.

Le prime subroutine sono dei brevi metodi che permettono di evitare ripetizioni nel programma favorendo il riuso del codice e di fare error checking.

```
1 /**
2  * Converte una stringa in input in int
3  * @param str la stringa da convertire
4  * @param val dove viene salvato il risultato della conversione
5  * @return true se la conversione termina con successo, falso
     altrimenti
6  */
7 bool parseInt(char* str, int* val);
```

```
8
9  /**
10  * Converte una stringa in input in float
11  * @param str la stringa da convertire
12  * @param val dove viene salvato il risultato della conversione
13  * @return true se la conversione termina con successo, falso
      altrimenti
14  */
15 bool parseFloat(char* str, float* val);
16
17 /**
18  * Verifica se il numero fornito è una potenza di due
19  * @param x il valore da verificare
20  * @return true se è una potenza di 2, false altrimenti
21  */
22 bool isPowerOfTwo(unsigned long x);
23
24 /**
25  * Ritorna un numero casuale nel range definito
26  * @param min il numero minimo, incluso
27  * @param max il numero massimo, incluso
28  * @return float il numero casuale
29  */
30 float getRandomFloatNumberInRange(int min, int max);
```

Di seguito vediamo alcune subroutine che riguardano parti principali del programma.

La prima subroutine utilizzata riguarda la distribuzione dei numeri (dati in input o generati randomicamente) tra i vari processori:

```
1  /**
2   * Distribuisce o riceve i numeri tra i vari processori e ritorna la
      propria somma parziale
3   * @param argv gli argomenti del programma, da cui leggere il numero
      di input ed i valori
4   * @param startTime valore di ritorno che rappresenta l'inizio delle
      operazioni del processore
5   * @param processorId l'id del processore corrente, per discriminare
      se deve inviare o ricevere
6   * @param numberOfProcessors il numero totale di processori
```

```
 */
float distributeNumbersAndGetPartialSum(char** argv, double*
    startTime, int processorId, int numberOfProcessors);
```

La seguente routine opera come di seguito:

- per il processore con $id = 0$ legge i valori da argv e li invia a tutti i processori;
- per i processori con $id > 0$ riceve i valori;
- successivamente, in maniera indipendente dal proprio id, assegna il risultato di `WPI_Wtime()` alla variabile `startTime` ed infine effettua la propria somma parziale e la ritorna.

Infine sono presenti le 3 subroutine che rappresentano le singole strategie, il loro funzionamento è già stato esplicitato nel capitolo 2.3.

```
/**
 * Applica la strategia uno per la somma, il processore master
    riceve i valori dagli altri processori
 * @param sum la somma corrente del processore
 * @param processorId l'id del processore
 * @param numberOfProcessors il numero di processori, deve essere
    una potenza di due
 * @param masterId l'id del processo incaricato di fare i calcoli
 */
float strategyOne(float sum, int processorId, int numberOfProcessors
    , int masterId);

/**
 * Applica la strategia due per la somma, questa utilizza un albero
    per la risoluzione, quindi un pre-requisito
 * del metodo è che il numero di processori sia una potenza di due
 * @param sum la somma corrente del processore
 * @param processorId l'id del processore
 * @param numberOfProcessors il numero di processori, deve essere
    una potenza di due
 * @param masterId l'id del processo incaricato di fare i calcoli
 */
float strategyTwo(float sum, int processorId, int numberOfProcessors
    , int masterId);
```

```
19
20 /**
21  * Applica la strategia tre per la somma, questa utilizza un albero
      per la risoluzione, quindi un pre-requisito
22  * del metodo è che il numero di processori sia una potenza di due
23  * @param sum la somma corrente del processore
24  * @param processorId l'id del processore
25  * @param numberOfProcessors il numero di processori, deve essere
      una potenza di due
26  */
27 float strategyThree(float sum, int processorId, int
      numberOfProcessors);
```

Si noti che la strategia 3 non richiede il passaggio del `masterId` in quanto tutti i processori effettuano tutti i calcoli, di conseguenza non è richiesto al suo interno che venga applicata logica di differenziazione.

5.2 Subroutine MPI

Seguono le subroutines MPI utilizzate nel progetto.
Esse sono documentate differentemente da quelle personalizzate in quanto la documentazione ufficiale è disponibile su open-mpi.org [1], viene però fornita una breve descrizione.

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

Descrizione: inizializza l'ambiente di esecuzione MPI.
Parametri in input:

- **argc**: un puntatore al numero di argomenti del programma.
- **argv**: l'array degli argomenti del programma.

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

Descrizione: assegna un identificativo (chiamato rank) al processo appartenente ad un communicator.
Parametri in input:

- **comm**: il communicator da cui si vuole ottenere il rank.

Parametri in output:

- **rank (int)**: il rank del processo chiamante nel gruppo comm.

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

Descrizione: ritorna la dimensione del gruppo di processi appartenenti ad un communicator.
Parametri in input:

- **comm**: il communicator da cui si vuole ottenere la dimensione.

Parametri in output:

- **size (int)**: il numero di processori del gruppo comm.

```
1 double MPI_Wtime(void)
```

Descrizione: ritorna un valore temporale in secondi passato dall'avvio di un processo.
Output:

- **time (double)**: tempo trascorso dall'ultima volta in cui la funzione è stata chiamata.

```
1 int MPI_Reduce(const void *sendbuf, void *recvbuf, int count,
    MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm);
```

Descrizione: permette al processore root di ottenere il risultato dell'operazione op degli elementi memorizzati in *sendbuf. Tale risultato viene memorizzato in recvbuf.
Parametri in input:

- **sendbuf**: il valore su cui applicare l'operazione di reduce.
- **count**: il numero di valori.
- **datatype**: il tipo di dato del valore, tra MPI_INT, MPI_FLOAT, ...
- **op**: l'operazione da effettuare, tra MPI_SUM, MPI_PROD, ...
- **root**: l'identificativo del processo che riceverà il risultato.
- **comm**: il communicator

Parametri in output:

- **recvbuf**: il risultato dell'operazione.

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

Descrizione: effettua un invio di dati in modo sincrono (bloccante).
Parametri in input:

- **buf**: l'indirizzo del valore da inviare.
- **count**: il numero di valori.
- **datatype**: il tipo di dato del valore, tra MPI_INT, MPI_FLOAT, ...
- **dest**: l'identificativo del processo di destinazione.
- **tag**: un tag associato all'invio.
- **comm**: il communicator.

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

Descrizione: effettua una ricezione di dati in modo sincrono (bloccante).
Parametri in input:

- **count**: il numero di valori.
- **datatype**: il tipo di dato del valore, tra MPI_INT, MPI_FLOAT, ...

- **source**: l'identificativo del processo mittente.
- **tag**: un tag associato all'invio.
- **comm**: il communicator

Parametri di output:

- **buf**: l'indirizzo dove verrà memorizzato il valore ricevuto
- **status**: informazioni aggiuntive sulla ricezione

```
1 int MPI_Barrier(MPI_Comm comm)
```

Descrizione: fornisce un meccanismo sincronizzante per tutti i processori del communicator comm.
Parametri in input:

- **comm**: il communicator

```
1 int MPI_Finalize()
```

Descrizione: Termina l'ambiente di esecuzione MPI.

Capitolo 6

Analisi dei tempi

Sono state effettuate varie analisi dei tempi di esecuzione dell'algoritmo in base alla strategia, al numero di valori da sommare ed il numero di processori utilizzati.

I tempi che seguono sono una media di 20 esecuzioni per ogni strategia.

Sono state testate somme contenenti 100, 1.000, 10.000, 100.000, 1.000.000, 10.000.000 e 100.000.000 valori, di queste somme è stata calcolata la media per avere risultati più coerenti.

Sono stati raccolti tempi con 1, 4 e 8 processori; uno per poter calcolare lo speed up e l'efficienza mentre 4 ed 8, in quanto potenze di due, permettono di utilizzare le strategie 2 e 3.

6.1 Analisi sul numero di processori

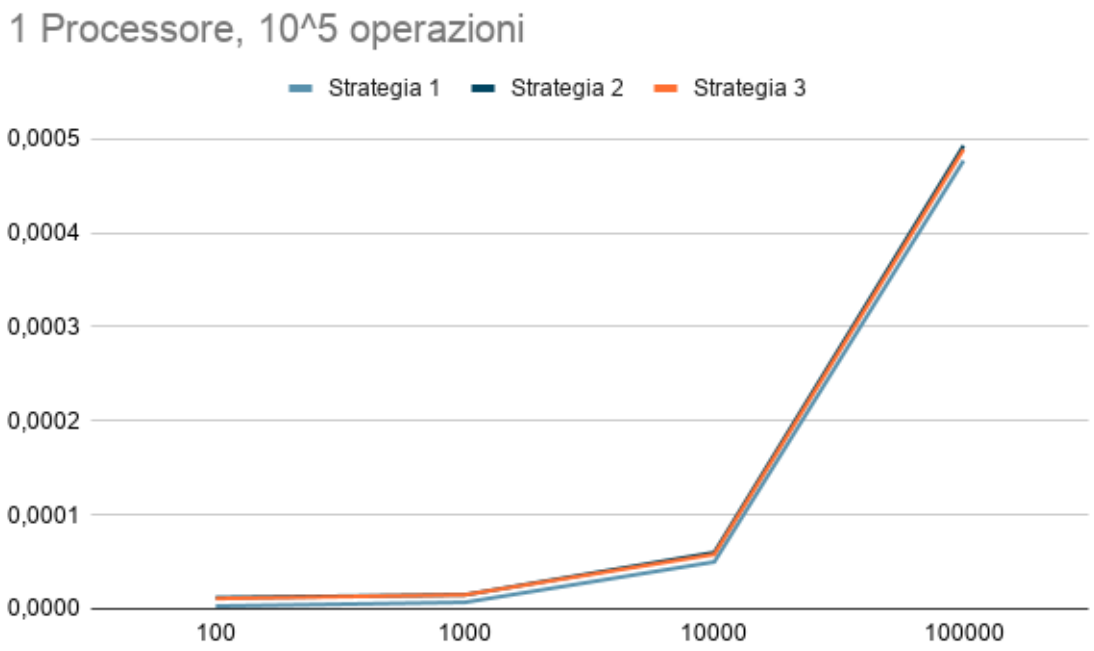
Seguono alcune analisi basate sul numero di processori indipendentemente dalla strategia adottata.

6.1.1 Esecuzione con 1 processore

I primi tempi sono stati raccolti utilizzando un solo processore, questo è essenziale anche per i calcoli successivi.

Come si può notare da questo grafico, indipendentemente dalla strategia, fino a 10^4 valori l'algoritmo è relativamente veloce, successivamente il tempo impiegato aumenta esponenzialmente.

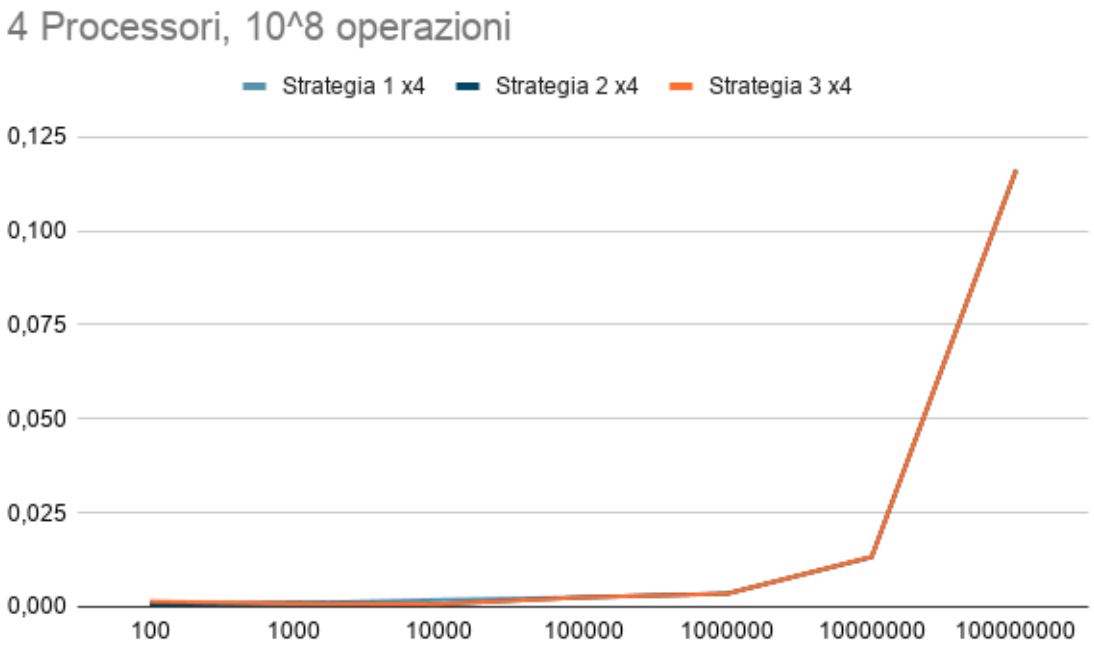
Con 10^8 valori la velocità dell'algoritmo impiega quasi mezzo secondo. Segue un altro grafico che copre i valori fino a 10^5 per mostrare i tempi in scala minore.



In questo grafico è possibile anche vedere come il primo algoritmo sia leggermente più performante rispetto agli altri, in quanto essi richiedono un setup più elaborato ed effettuano molte più operazioni di send e receive.

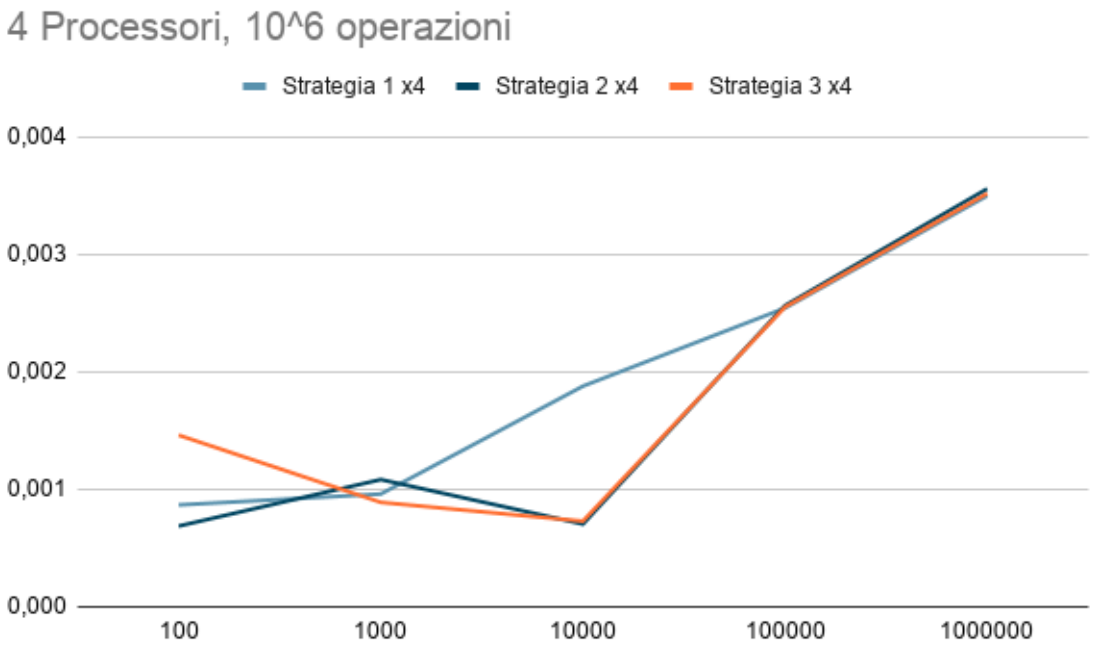
6.1.2 Esecuzione con 4 processori

Come per i 4 processori, viene mostrato il grafico fino a 10^8 operazioni.



Come è possibile notare, l'operazione più onerosa con un singolo processore si dimostra 3.7 volte più veloce, avendo una velocità media di 0,116036 contro 0,431933 del singolo processore.

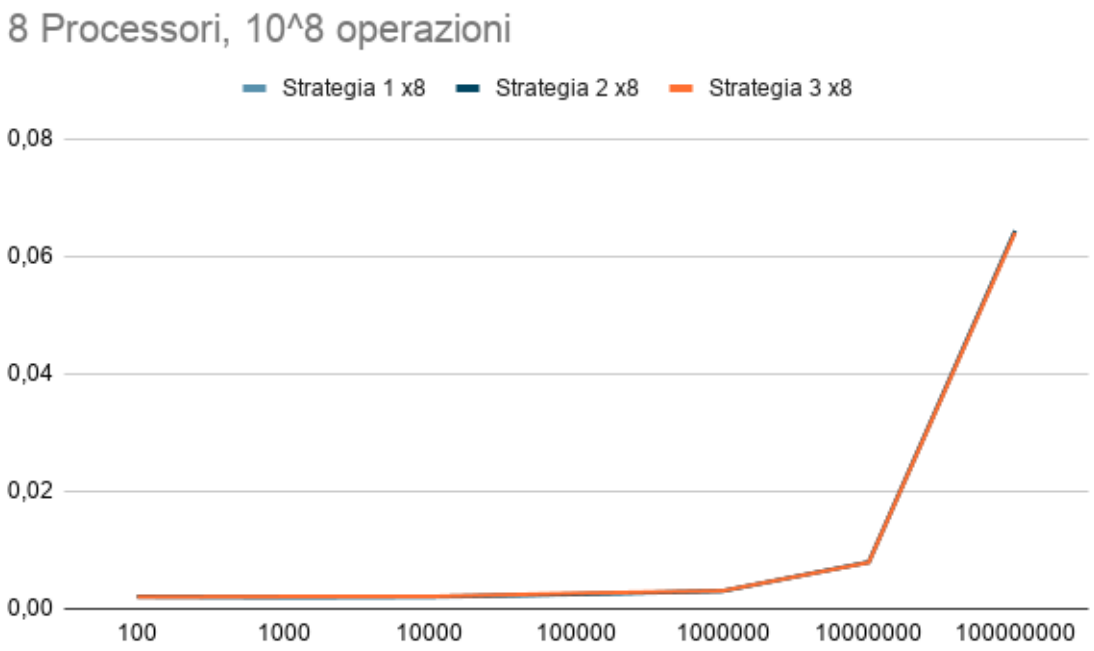
Segue il grafico fino a 10^6 operazioni.



Rispetto al singolo processore si presentano già delle differenze, si può notare come la prima strategia lentamente peggiora in performance, già con 10^4 ha una velocità minore rispetto alla 2 o 3.
Si può anche notare come la terza strategia con pochi valori ha le performance peggiori.

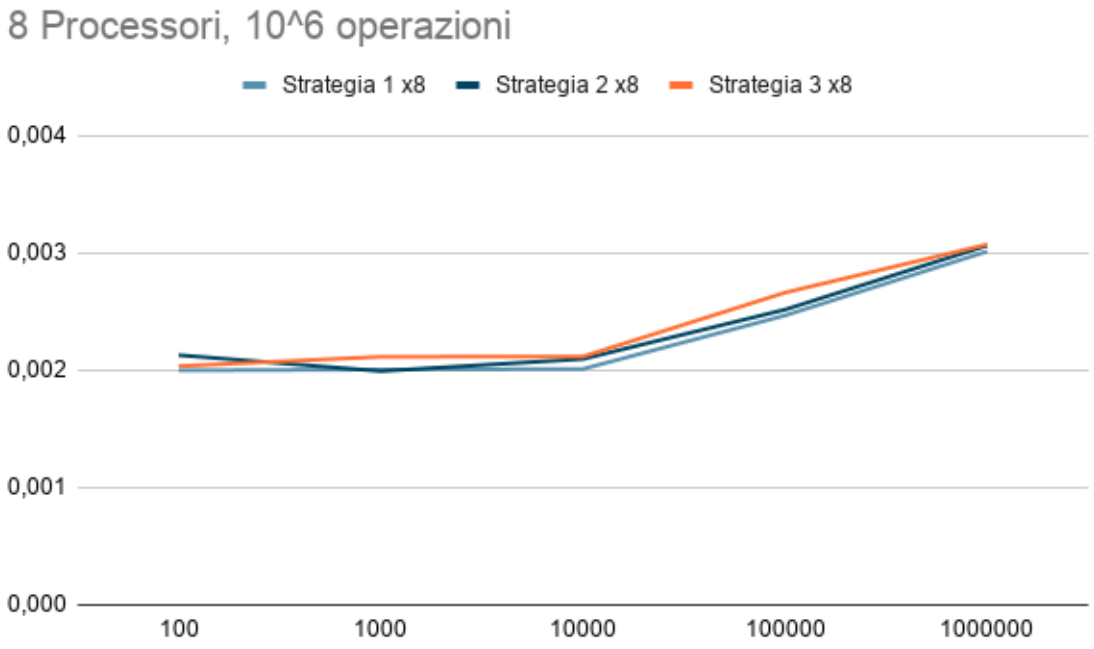
6.1.3 Esecuzione con 8 processori

Come precedentemente, viene mostrato il grafico fino a 10^8 operazioni.



Nuovamente, è possibile vedere come 8 processori impiegano la metà del tempo rispetto a 4 processori.

Segue il diagramma di 8 processori con 10^6 operazioni.



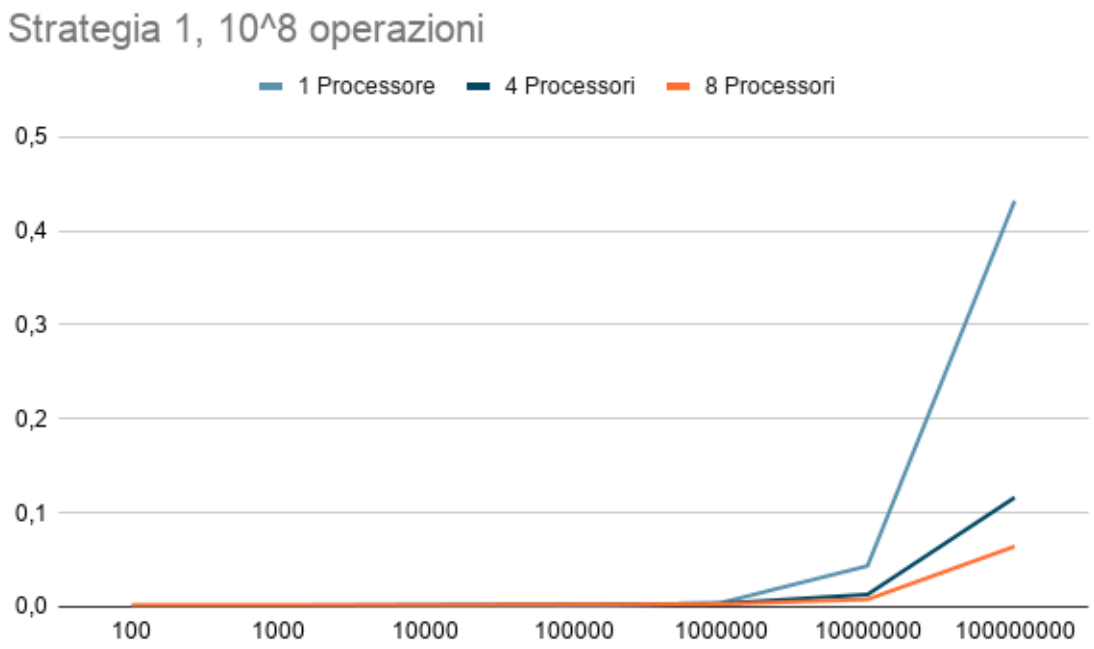
Si può notare come rispetto ai 4 processori si abbia un comportamento più uniforme. Le operazioni con meno valori però sono più lente rispetto a quelle con valori più elevati, questo vale anche per il calcolo a più processori rispetto quello con singolo processore.

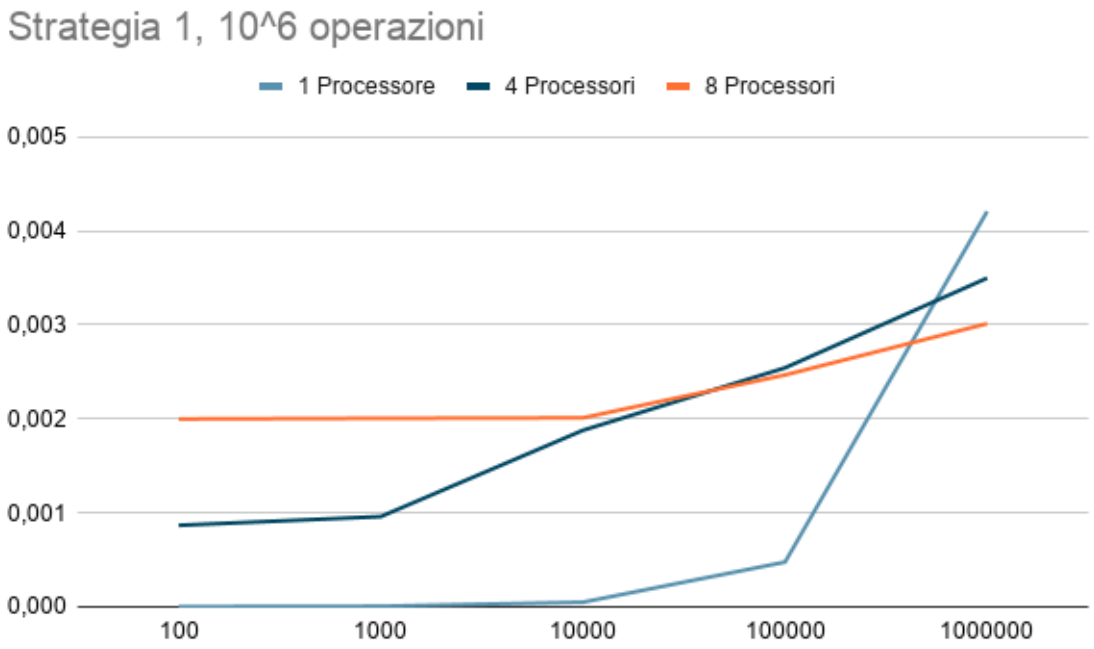
6.2 Analisi sulle strategie

Seguono le singole strategie differenziate dal numero di processori

6.2.1 Strategia uno

Seguono i diagrammi per 10^8 e 10^6 operazioni.

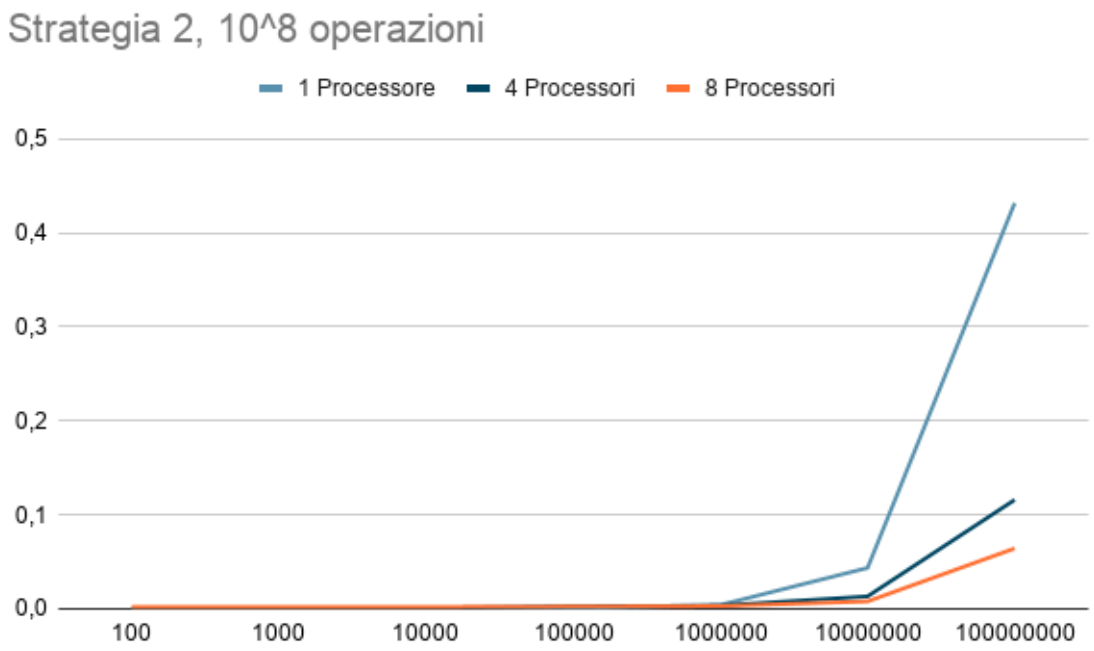


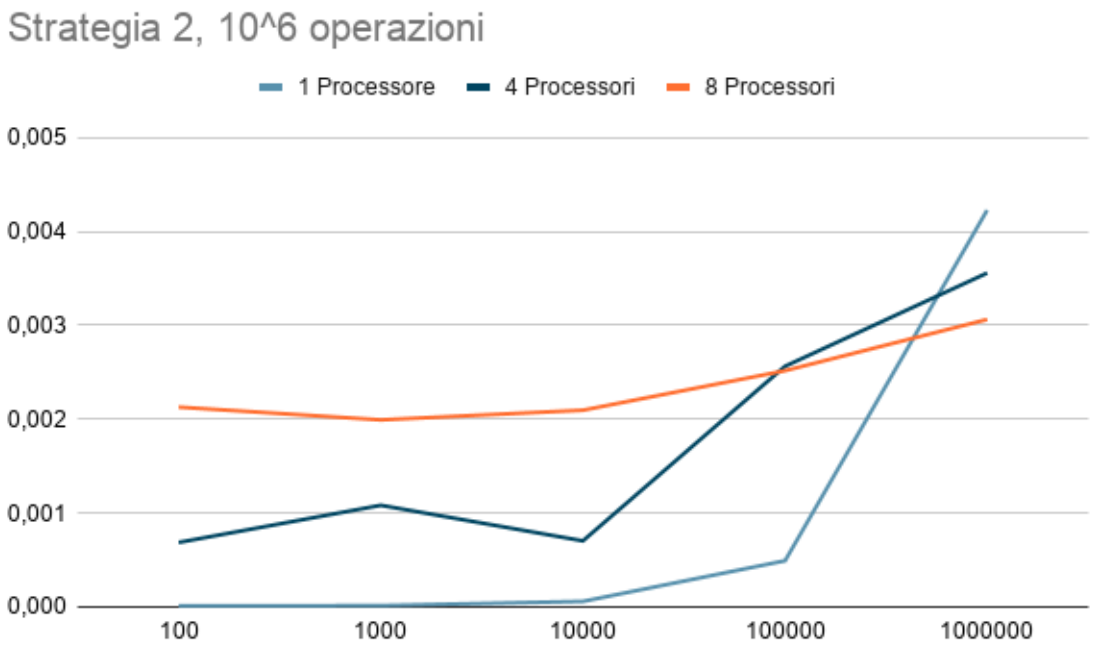


Si può facilmente notare che all'aumentare del numero di valori da sommare, la velocità di calcolo è fortemente influenzata dal numero di processori utilizzati. In particolare, per piccole operazioni un minor numero di processori offre risultati migliori, all'aumentare dei valori da sommare però le performance peggiorano notevolmente.

6.2.2 Strategia due

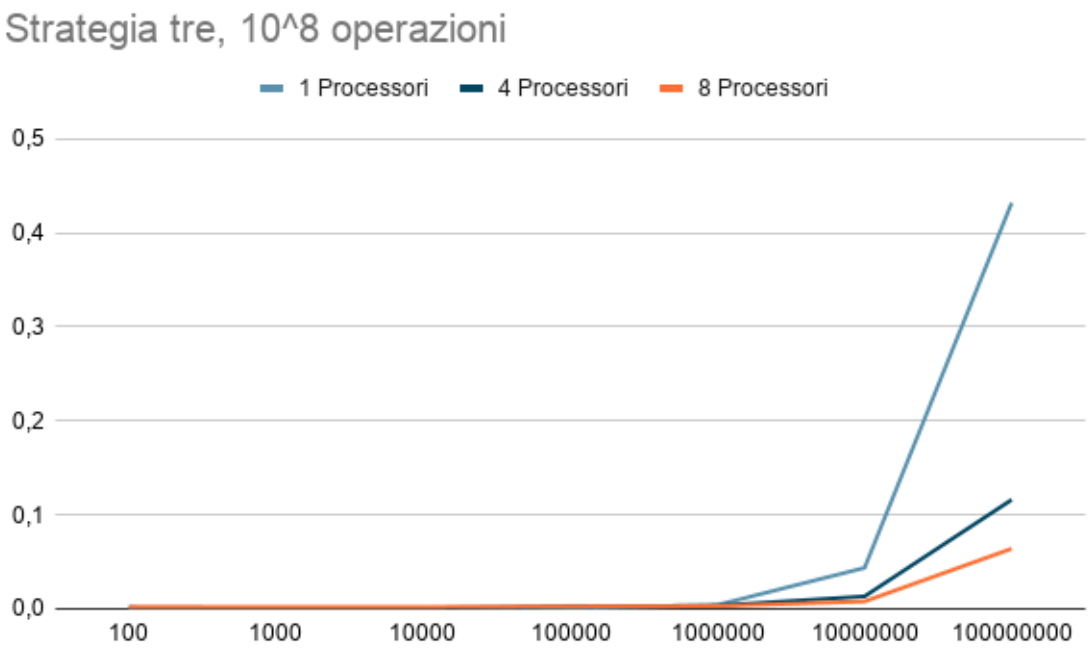
Seguono i diagrammi per 10^8 e 10^6 operazioni.

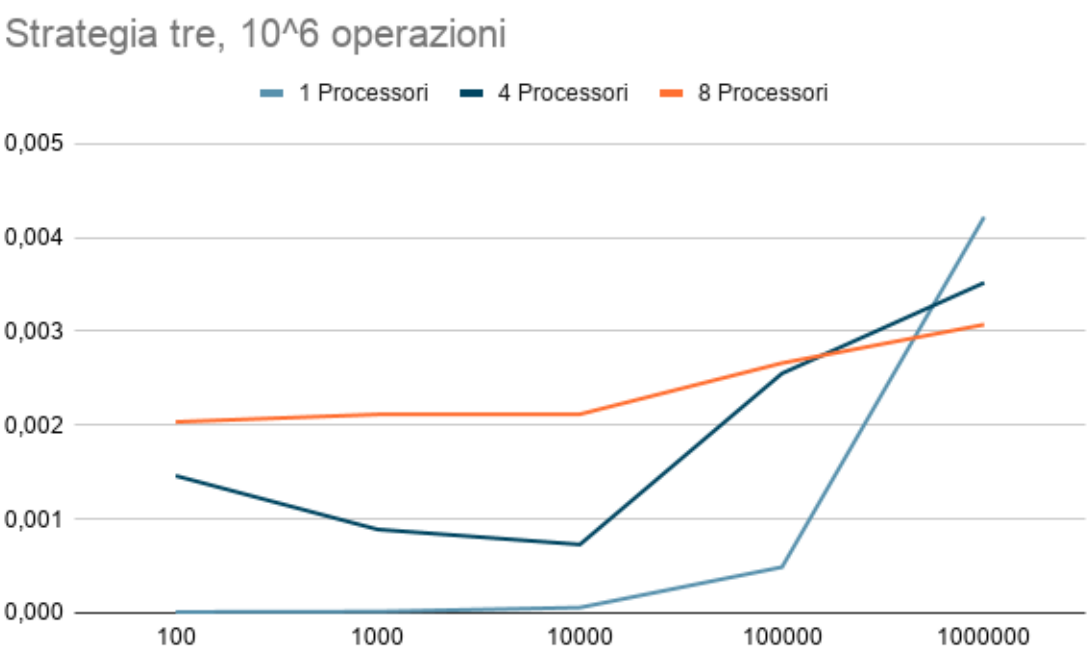




6.2.3 Strategia tre

Seguono i diagrammi per 10^8 e 10^6 operazioni.


Strategia tre, 10^8 operazioni
1 Processori
4 Processori
8 Processori
0,5
0,4
0,3
0,2
0,1
0,0
100
1000
10000
100000
1000000
10000000
100000000


Strategia tre, 10^6 operazioni
1 Processori
4 Processori
8 Processori
0,005
0,004
0,003
0,002
0,001
0,000
100
1000
10000
100000
1000000

6.3 Speed-up ed efficienza

Conseguentemente alle precedenti analisi è possibile calcolare lo speed-up e l'efficienza delle tre strategie in rapporto al numero di processori.

6.3.1 Calcolo dello speed-up

Lo speed-up misura la riduzione del tempo di esecuzione rispetto all'algoritmo su di un solo processore.

Definiamo lo speed-up come: $S(p) = \frac{T(1)}{T(p)}$, e cioè il rapporto tra il tempo impiegato dall'algoritmo per calcolare il risultato utilizzando un solo processore e il tempo impiegato per calcolare il risultato utilizzando p processori.

Seguono i valori di riferimento:

$T(1)$, tempi ottenuti con un solo processore:

	100	1,000	10,000	100,000	1,000,000	10,000,000	100,000,000
Strategia 1	0,000003	0,000007	0,000050	0,000477	0,004209	0,043610	0,431933
Strategia 2	0,000012	0,000015	0,000060	0,000493	0,004229	0,043652	0,431811
Strategia 3	0,000011	0,000015	0,000058	0,000489	0,004220	0,043697	0,432058

4 processori:

	100	1,000	10,000	100,000	1,000,000	10,000,000	100,000,000
Strategia 1	0,000868	0,000961	0,001879	0,002544	0,003499	0,013210	0,116359
Strategia 2	0,000688	0,001084	0,000705	0,002565	0,003559	0,013313	0,116036
Strategia 3	0,001462	0,000890	0,000730	0,002553	0,003519	0,013331	0,116140

8 processori:

	100	1,000	10,000	100,000	1,000,000	10,000,000	100,000,000
Strategia 1	0,002000	0,002007	0,002012	0,002468	0,003014	0,007951	0,064360
Strategia 2	0,002130	0,001994	0,002097	0,002518	0,003064	0,007966	0,064356
Strategia 3	0,002036	0,002116	0,002118	0,002663	0,003073	0,007912	0,064072

Seguono le tabelle contenenti i valori relativi al calcolo dello speed-up nelle tre strategie.

Strategia 1:

N° processori	100	1,000	10,000	100,000	1,000,000	10,000,000	100,000,000
4	0,00346	0,00728	0,02661	0,18750	1,20292	3,30129	3,71207
8	0,00150	0,00349	0,02485	0,19327	1,39648	5,48484	6,71120

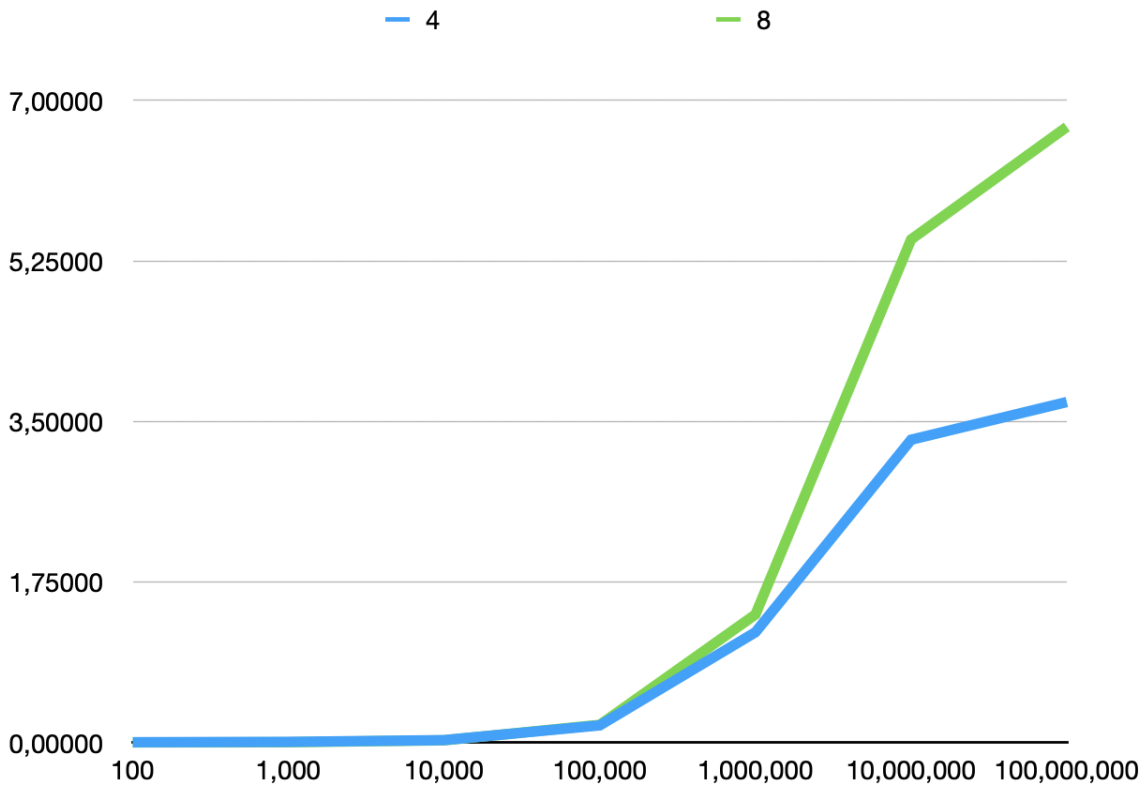
Strategia 2:

N° processori	100	1,000	10,000	100,000	1,000,000	10,000,000	100,000,000
4	0,01744	0,01384	0,08511	0,19220	1,18826	3,27890	3,72135
8	0,00563	0,00752	0,02861	0,19579	1,38022	5,47979	6,70972

Strategia 3:

N° processori	100	1,000	10,000	100,000	1,000,000	10,000,000	100,000,000
4	0,00752	0,01685	0,07945	0,19154	1,19920	3,27785	3,72015
8	0,00540	0,00709	0,02738	0,18363	1,37325	5,52288	6,74332

Tutte e 3 le strategie danno origine al seguente grafico:



6.3.2 Calcolo dell'efficienza

L'efficienza misura quanto l'algoritmo sfrutta il parallelismo del calcolatore.
Definiamo l'efficienza come: $E(p) = \frac{S(p)}{p}$, e cioè il rapporto tra lo speed-up su p processori e il numero di processori

Seguono le tabelle contenenti i valori relativi al calcolo dell'efficienza nelle tre strategie.

Strategia 1:

N° processori	100	1,000	10,000	100,000	1,000,000	10,000,000	100,000,000
4	4,00346	4,00728	4,02661	4,18750	5,20292	7,30129	7,71207
8	8,00150	8,00349	8,02485	8,19327	9,39648	13,48484	14,71120

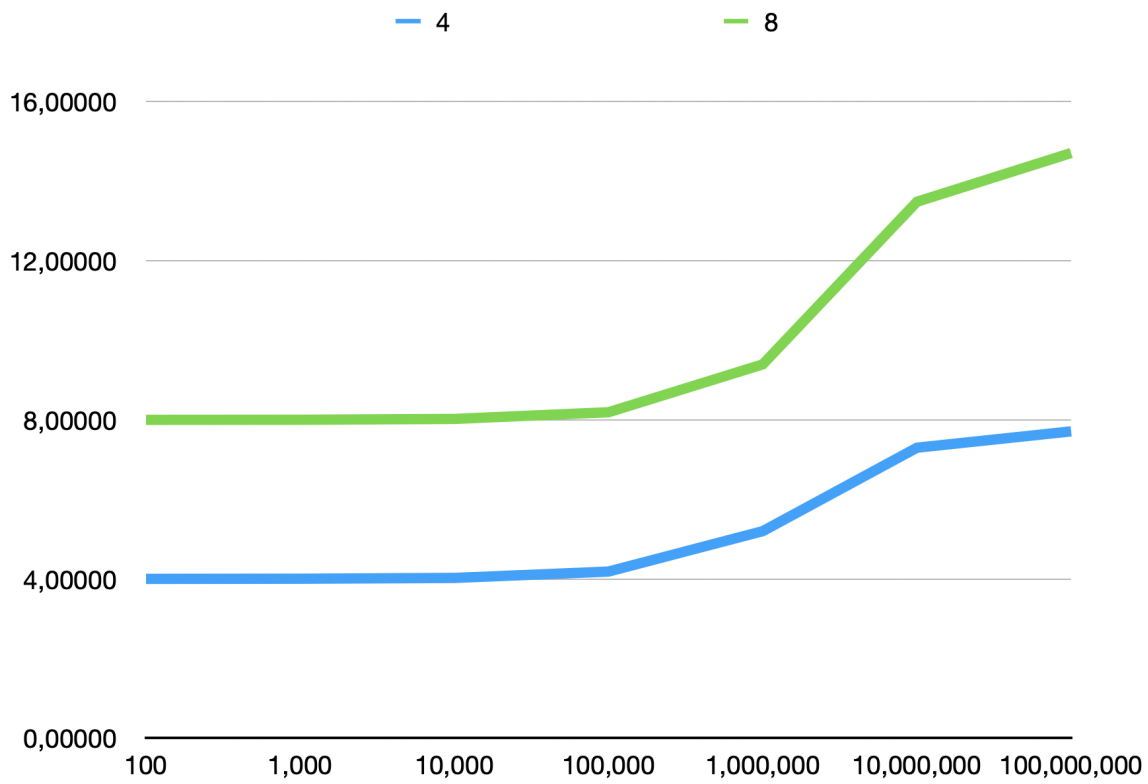
Strategia 2:

N° processori	100	1,000	10,000	100,000	1,000,000	10,000,000	100,000,000
4	0,00436	0,00346	0,02128	0,04805	0,29706	0,81973	0,93034
8	0,00070	0,00094	0,00358	0,02447	0,17253	0,68497	0,83872

Strategia 3:

N° processori	100	1,000	10,000	100,000	1,000,000	10,000,000	100,000,000
4	4,00752	4,01685	4,07945	4,19154	5,19920	7,27785	7,72015
8	8,00540	8,00709	8,02738	8,18363	9,37325	13,52288	14,74332

Tutte e 3 le strategie danno origine al seguente grafico:


4
8
16,00000
12,00000
8,00000
4,00000
0,00000
100
1,000
10,000
100,000
1,000,000
10,000,000
100,000,000

Capitolo 7

Esempi d'uso

L'utilizzo dell'algoritmo può avvenire tramite due modi, o utilizzandolo in maniera diretta, quindi invocando in locale le librerie MPI, oppure utilizzando uno script PBS. Seguono esempi d'uso per entrambi i casi.

7.1 Uso diretto

Ipotizzando di avere open-mpi installato nel path /opt/usr/local/bin/, ed avere nella stessa cartella il file main.c, contenente il codice sorgente, è necessario compilarlo eseguendo:

```
~/opt/usr/local/bin/mpicc -o main main.c
```

Dopo aver compilato il file, ipotizzando di voler sommare i seguenti 5 numeri:
3.1415 0.3456 3.1278 6.96969 4.20

Si ipotizza di voler utilizzare 4 processori, con strategia 1 e volere la stampa dal terzo processore.

Sarà sufficiente digitare:

```
~/opt/usr/local/bin/mpiexec -np 4 ./main 5 3.1415 0.3456 3.1278
   6.96969 4.20 1 3
```

Verrà stampato il seguente risultato:

```
[Completed] Processor ID 3, total sum: 17.784590
Time elapsed: 1.200000e-05 seconds
```

Si noti che il tempo impiegato di esecuzione può variare in base all'hardware del dispositivo.

7.2 Uso tramite script PBS

Utilizzando l'algoritmo su un cluster è necessario utilizzare uno script PBS, in appendice ne è fornito uno che verrà utilizzato in questo caso d'uso ed è chiamato sum-job-script.pbs.

Si ipotizzi di voler sommare gli stessi 5 numeri precedenti:
3.1415 0.3456 3.1278 6.96969 4.20

Si ipotizzi di voler utilizzare 8 processori, con strategia 2 e volere la stampa del processore numero 5.

Innanzitutto sarà necessario scrivere i numeri nel file chiamato input.txt, come segue:
3.1415 0.3456 3.1278 6.96969 4.20

Successivamente è necessario modificare la seguente parte dello script PBS:

```
...
#PBS -l nodes=8:ppn=8
#PBS -o sum.out
...
#######################
## CUSTOM VALUES ##
#######################
# How many numbers are going to be sum
NUMBERS_TO_SUM=5

# Strategy 1, 2 or 3
STRATEGY=2

# The processor that should print the result, -1 to let make all of
    them print
PRINTER_ID=5
#######################
```

Ed invocare lo script come segue:

```
qsub sum-job-script.pbs
```

Attendere il completamento e, supponendo che il risultato sia memorizzato nel file sum.out (come scritto nel file pbs) digitare:

```
cat sum.out
```

Verrà stampato il seguente risultato:

```
[Completed] Processor ID 3, total sum: 17.784590
Time elapsed: 1.200000e-05 seconds
```

Si noti che il tempo impiegato di esecuzione può variare in base all'hardware del dispositivo.

Bibliografia

[1] Open MPI Documentation,
`https://www.open-mpi.org/doc/`

Capitolo A

Codice

Segue il codice dell'algoritmo.

A.1 main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <errno.h>
#include <limits.h>
#include <stdbool.h>
#include <math.h>
#include <time.h>

#define MAX_RANDOM_NUMBER 1000
#define TAG_START 100

float distributeNumbersAndGetPartialSum(char** argv, double*
    startTime, int processorId, int numberOfProcessors);

// Strategie
float strategyOne(float sum, int processorId, int numberOfProcessors
    , int masterId);
float strategyTwo(float sum, int processorId, int numberOfProcessors
    , int masterId);
float strategyThree(float sum, int processorId, int
    numberOfProcessors);
```

```
19
20 // Funzioni accessorie
21 float getRandomFloatNumberInRange(int min, int max);
22 bool parseInt(char* str, int* val);
23 bool parseFloat(char* str, float* val);
24 bool isPowerOfTwo(unsigned long x);
25
26 /**
27  * Gli argomenti vengono passati nella forma: nInput valori[nInput]
      strategia masterId
28  * @param nInput numero di valori da sommare
29  * @param numbers[nInput] se nInput<=20 sono i numeri da sommare,
      altrimenti è ignorato
30  * @param strategy la strategia da utilizzare, valore compreso tra 1
       e 3
31  * @param masterId il processore che effettua i calcoli e stampa la
      somma, se strategy=3 e -1 stampano tutti
32  */
33 int main(int argc, char** argv) {
34     int processorId, numberOfProcessors;
35
36     MPI_Init(&argc, &argv);
37
38     MPI_Comm_rank(MPI_COMM_WORLD, &processorId);
39     MPI_Comm_size(MPI_COMM_WORLD, &numberOfProcessors);
40
41     /* I tempi dei singoli processori, ottenuti tramite differenza,
      mentre il tempo totale come il
42      * massimo valore tra i processori */
43     double startTime, endTime, processorTime, totalTime = 0.0;
44
45     /* Prova ad convertire da argv il valore di strategy e masterId
      */
46     int masterId, strategy;
47     if(!parseInt(argv[argc - 2], &strategy) || !parseInt(argv[argc -
       1], &masterId)) {
48         fprintf(stderr, "ERROR - Cannot parse the masterId or
      strategy with values provided.\n"
```

```
49                          "strategy:%s\nprinter:%s\n\n", argv[argc -
    2], argv[argc - 1]);
50          return 1;
51      }
52
53      if(strategy < 0 || strategy > 3) {
54          fprintf(stderr, "ERROR - Strategy number not allowed, must
    be a value between 1 and 3.\nstrategy:%d\n", strategy);
55          return 1;
56      }
57
58      if(masterId < -1 || masterId > numberOfProcessors) {
59          printf("WARNING - inserted masterId is not valid, the
    processor 0 will print instead.\n");
60          masterId = 0;
61      }
62      else if(masterId == -1 && (strategy == 1 || strategy == 2)) {
63          printf("WARNING - Print of all processor with strategy 1 or
    2 is not available, the processor 0 will print the result instead
    .\n");
64          masterId = 0;
65      }
66
67      /* Invoca la distribuzione dei numeri, assegna lo startTime ed
    ottiene la propria somma parziale */
68      float sum = distributeNumbersAndGetPartialSum(argv, &startTime,
    processorId, numberOfProcessors);
69
70      /* Per le strategie 2 e 3 è un requisito essenziale che il
    numero di processori sia una potenza di due */
71      if(strategy == 3 && isPowerOfTwo(numberOfProcessors)) {
72          sum = strategyThree(sum, processorId, numberOfProcessors);
73      } else if(strategy == 2 && isPowerOfTwo(numberOfProcessors)) {
74          sum = strategyTwo(sum, processorId, numberOfProcessors,
    masterId);
75      } else {
76          sum = strategyOne(sum, processorId, numberOfProcessors,
    masterId);
77      }
```

```
78
79      /* Calcola il tempo finale di esecuzione dalla somma parziale */
80      endTime = MPI_Wtime();
81      processorTime = endTime - startTime;
82
83      /* Passa al masterId il tempo maggiore impiegato */
84      int sendTo = masterId == -1 ? 0 : masterId;
85      MPI_Reduce(&processorTime, &totalTime, 1, MPI_DOUBLE, MPI_MAX,
        sendTo, MPI_COMM_WORLD);
86
87      if(masterId == processorId || masterId == -1) {
88          printf("[Completed] Processor ID %d, total sum: %f\n",
        processorId, sum);
89          if(totalTime > 0.0) {
90              printf("Time elapsed: %e seconds\n", totalTime);
91          }
92      }
93
94      MPI_Finalize();
95      return 0;
96  }
97
98  /**
99   * Converte una stringa in input in int
100  * @param str la stringa da convertire
101  * @param val dove viene salvato il risultato della conversione
102  * @return true se la conversione termina con successo, falso
       altrimenti
103  */
104 bool parseInt(char* str, int* val) {
105     char *temp;
106     bool result = true;
107     errno = 0;
108     long ret = strtol(str, &temp, 0);
109
110     if (temp == str || *temp != '\0' || ((ret == LONG_MIN || ret ==
        LONG_MAX) && errno == ERANGE))
111         result = false;
112
```

```
113     *val = (int) ret;
114     return result;
115 }
116 
117 /**
118  * Converte una stringa in input in float
119  * @param str la stringa da convertire
120  * @param val dove viene salvato il risultato della conversione
121  * @return true se la conversione termina con successo, falso
        altrimenti
122  */
123 bool parseFloat(char* str, float* val) {
124     *val = atof(str);
125     return true;
126 }
127 
128 /**
129  * Verifica se il numero fornito è una potenza di due
130  * @param x il valore da verificare
131  * @return true se è una potenza di 2, false altrimenti
132  */
133 bool isPowerOfTwo(unsigned long x) {
134     return (x & (x - 1)) == 0;
135 }
136 
137 /**
138  * Ritorna un numero casuale nel range definito
139  * @param min il numero minimo, incluso
140  * @param max il numero massimo, incluso
141  * @return float il numero casuale
142  */
143 float getRandomFloatNumberInRange(int min, int max) {
144     return (float) min + rand() / (float) RAND_MAX * max - min;
145 }
146 
147 /**
148  * Distribuisce o riceve i numeri tra i vari processori e ritorna la
        propria somma parziale
```

```
149  * @param argv gli argomenti del programma, da cui leggere il numero
         di input ed i valori
150  * @param startTime valore di ritorno che rappresenta l'inizio delle
         operazioni del processore
151  * @param processorId l'id del processore corrente, per discriminare
         se deve inviare o ricevere
152  * @param numberOfProcessors il numero totale di processori
153  */
154 float distributeNumbersAndGetPartialSum(char** argv, double*
        startTime, int processorId, int numberOfProcessors) {
155     int tmp, sentNumbers, tag, i;
156     float sum = 0;
157 
158     /* Prova a leggere da argv il numero di valori da sommare */
159     int inputSize;
160     if(!parseInt(argv[1], &inputSize) || inputSize < 0) {
161         fprintf(stderr, "ERROR - Input size not allowed, must be a
        value between 0 and INT_MAX\ninputSize:%s\n", argv[1]);
162         return 1;
163     }
164 
165     // Numero dei valori da distribuire per ogni processore
166     int amountOfNumbers = inputSize / numberOfProcessors;
167 
168     // Numeri non multipli del numero di processori da ripartire
169     int rest = inputSize % numberOfProcessors;
170     float* numbers = NULL;
171 
172     // Alcuni processori sommeranno più di un numero
173     if(processorId < rest) {
174         amountOfNumbers++;
175     }
176 
177     // Il processore 0 distribuisce i valori
178     if(processorId == 0) {
179         numbers = malloc(sizeof(float) * inputSize);
180 
181         if(inputSize <= 20) {
182             // Legge i valori da riga di comando
```

```
183             int k = 2;
184             for (i = 0; i < inputSize; i++) {
185                 if(!parseFloat(argv[k++], &numbers[i])) {
186                     fprintf(stderr, "ERROR - Cannot parse a value to
    sum.\nvalue[%d]: %s\n", i, argv[k-1]);
187                     return 1;
188                 }
189             }
190         } else {
191             // Genera valori casuali
192             srand(time(NULL));
193             for (i = 0; i <= inputSize; i++) {
194                 numbers[i] = getRandomFloatNumberInRange(0,
    MAX_RANDOM_NUMBER);
195             }
196         }
197 
198         tmp = amountOfNumbers;
199         sentNumbers = 0;
200 
201         // Invia i valori ai processori
202         for (i = 1; i < numberOfProcessors; ++i) {
203             sentNumbers += tmp;
204             tag = TAG_START + i;
205 
206             if (i == rest) {
207                 tmp--;
208             }
209 
210             MPI_Send(&numbers[sentNumbers], tmp, MPI_FLOAT, i, tag,
    MPI_COMM_WORLD);
211         }
212     } else { // Riceve i valori
213         numbers = malloc(sizeof(float) * amountOfNumbers);
214         MPI_Status status;
215 
216         tag = TAG_START + processorId;
217         MPI_Recv(numbers, amountOfNumbers, MPI_FLOAT, 0, tag,
    MPI_COMM_WORLD, &status);
```

```
218     }
219
220     // Si sincronizza con gli altri processori
221     MPI_Barrier(MPI_COMM_WORLD);
222     *startTime = MPI_Wtime();
223
224     // Effettua la propria somma parziale
225     for (i = 0; i < amountOfNumbers; ++i) {
226         sum += numbers[i];
227     }
228
229     free(numbers);
230     return sum;
231 }
232
233 /**
234  * Applica la strategia uno per la somma, il processore master
        riceve i valori dagli altri processori
235  * @param sum la somma corrente del processore
236  * @param processorId l'id del processore
237  * @param numberOfProcessors il numero di processori, deve essere
        una potenza di due
238  * @param masterId l'id del processo incaricato di fare i calcoli
239  */
240 float strategyOne(float sum, int processorId, int numberOfProcessors
    , int masterId) {
241     float partialSum;
242     int tag, i;
243     MPI_Status status;
244
245     if(processorId == masterId) {
246         for (i = 0; i < numberOfProcessors; i++) {
247             if(masterId == i) {
248                 continue;
249             }
250
251             tag = TAG_START + i;
252             MPI_Recv(&partialSum, 1, MPI_FLOAT, i, tag,
    MPI_COMM_WORLD, &status);
```

```
253             sum += partialSum;
254         }
255     } else {
256         tag = TAG_START + processorId;
257         MPI_Send(&sum, 1, MPI_FLOAT, masterId, tag, MPI_COMM_WORLD);
258     }
259 
260     return sum;
261 }
262 
263 /**
264  * Applica la strategia due per la somma, questa utilizza un albero
       per la risoluzione, quindi un pre-requisito
265  * del metodo è che il numero di processori sia una potenza di due
266  * @param sum la somma corrente del processore
267  * @param processorId l'id del processore
268  * @param numberOfProcessors il numero di processori, deve essere
       una potenza di due
269  * @param masterId l'id del processo incaricato di fare i calcoli
270  */
271 float strategyTwo(float sum, int processorId, int numberOfProcessors
       , int masterId) {
272     float partialSum = 0;
273     int tag, i;
274     MPI_Status status;
275 
276     int logicId = processorId + (numberOfProcessors - masterId);
277     logicId = logicId % numberOfProcessors;
278 
279     for (i = 0; i < (int) log2(numberOfProcessors); ++i) {
280         tag = TAG_START + i;
281         if((logicId % (int) pow(2, i)) == 0) {
282             if((logicId % (int) pow(2, i + 1)) == 0) {
283                 int senderId = ((int) (processorId + pow(2, i)) %
       numberOfProcessors);
284                 MPI_Recv(&partialSum, 1, MPI_FLOAT, senderId, tag,
       MPI_COMM_WORLD, &status);
285                 sum += partialSum;
286             } else {
```

```
287                 int receiverId = processorId - (int) pow(2, i);
288                 if(receiverId < 0) {
289                     receiverId += numberOfProcessors;
290                 }
291                 MPI_Send(&sum, 1, MPI_FLOAT, receiverId, tag,
    MPI_COMM_WORLD);
292             }
293         }
294     }
295     return sum;
296 }
297
298 /**
299  * Applica la strategia tre per la somma, questa utilizza un albero
    per la risoluzione, quindi un pre-requisito
300  * del metodo è che il numero di processori sia una potenza di due
301  * @param sum la somma corrente del processore
302  * @param processorId l'id del processore
303  * @param numberOfProcessors il numero di processori, deve essere
    una potenza di due
304  */
305 float strategyThree(float sum, int processorId, int
    numberOfProcessors) {
306     float partialSum = 0;
307     int tag, i;
308     MPI_Status status;
309
310     for (i = 0; i < (int) log2(numberOfProcessors); ++i) {
311         tag = TAG_START + i;
312         if((processorId % (int) pow(2, i + 1)) < (int) pow(2, i)) {
313             int otherId = processorId + (int) pow(2, i);
314             MPI_Send(&sum, 1, MPI_FLOAT, otherId, tag,
    MPI_COMM_WORLD);
315             MPI_Recv(&partialSum, 1, MPI_FLOAT, otherId, tag,
    MPI_COMM_WORLD, &status);
316         } else {
317             int otherId = processorId - (int) pow(2, i);
318             MPI_Send(&sum, 1, MPI_FLOAT, otherId, tag,
    MPI_COMM_WORLD);
```

```
319             MPI_Recv(&partialSum, 1, MPI_FLOAT, otherId, tag,
    MPI_COMM_WORLD, &status);
320         }
321         sum += partialSum;
322     }
323
324     return sum;
325 }
```

A.2 sum-job-script.pbs

```
#!/bin/bash

# Imposta le direttive per l'ambiente PBS
#PBS -q studenti
#PBS -l nodes=8:ppn=8
#PBS -N sum
#PBS -o sum.out
#PBS -e sum.err

sort -u $PBS_NODEFILE > hostlist

NCPU=$(wc -l < hostlist)
echo "[Job-Script] Starting with "$NCPU" CPUs..."

echo "[Job-Script] Compiling..."
PBS_O_WORKDIR=$PBS_O_HOME/8-november
/usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS_O_WORKDIR/sum
    $PBS_O_WORKDIR/sum.c

echo "[Job-Script] Checking input the values..."
########################
## CUSTOM VALUES ##
########################
# Il numero di valori da sommare
NUMBERS_TO_SUM=10

# La strategia da applicare, valore compreso tra 1 e 3
STRATEGY=2

# Il processore che effettua i calcoli e stampa il risultato, -1 per
     far stampare a tutti nella strategia 3
PRINTER_ID=5
########################

# Verifica se il file esiste per leggere i valori
FILEPATH=$PBS_O_WORKDIR/input.txt
```

```
35 if [ -f $FILEPATH ]; then
36     NUMBERS="$(cat $FILEPATH)"
37 else
38     echo "[Job-Script] ERROR: File input.txt is missing"
39     exit 1;
40 fi
41
42 echo "[Job-Script] Running the process..."
43 /usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile hostlist --np
    $NCPU $PBS_O_WORKDIR/sum $NUMBERS_TO_SUM $NUMBERS $STRATEGY
    $PRINTER_ID
```