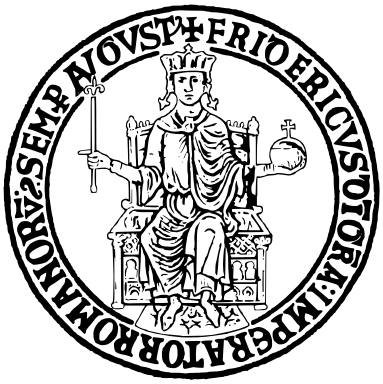


Università degli studi di Napoli Federico II

Scuola Politecnica e delle Scienze di base

Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione

Corso di laurea magistrale in Informatica

Parallel and Distributing Computing

Documentazione relativa al calcolo della somma di n numeri su architettura MIMD a memoria distribuita

Studente
Francesco Scarfato

Matricola
N86003769

2023-2024

Indice

Capitolo 1

Introduzione

Nel campo dell'informatica, viene definito *parallelismo* la decomposizioni di un problema in sotto problemi risolti contemporaneamente da più processori. L'introduzione del calcolo parallelo ha permesso di superare molti limiti tecnologici e di ottenere prestazioni di alto livello. L'architettura MIMD (Multiple Instruction Multiple Data) è un paradigma che consente di eseguire diverse operazioni su diversi processori simultaneamente. Nel caso di studio, ogni unità di elaborazione possiede una memoria indipendente, quindi la comunicazione tra processori è una parte fondamentale per la risoluzione di problemi.

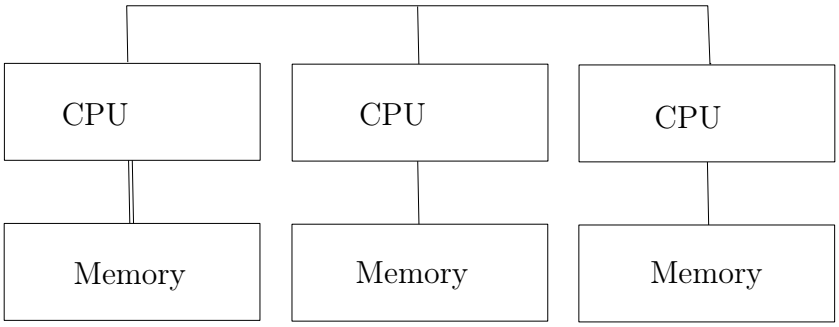


Figura 1.1: Architettura MIMD a memoria distribuita

Questo documento si propone di descrivere la problematica del calcolo della somma, descrivere in dettaglio le varie strategie con l'implementazione dell'algoritmo e analizzarne, tramite lo studio di dati, le performance.

Capitolo 2

Descrizione del problema

Il calcolo della somma di n numeri è un problema base fondamentale ed è presente in numerose applicazioni. Per sommare n numeri, sono necessarie $n-1$ operazioni di somma in quanto un'istruzione base può sommare solo due numeri alla volta. In un calcolatore sequenziale (mono processore), la complessità temporale per risolvere questo problema è $T(n-1)$ in quanto, a ogni passo temporale, viene eseguita una somma.

Siano $a_1, a_2, \ldots, a_n$ i valori di sommare e sia S la somma totale, allora un calcolatore sequenziale

$$a_{1,2} = a_1 + a_2 \qquad \text{(Passo 1)}$$
$$a_{1,2,3} = a_{1,2} + a_3 \qquad \text{(Passo 2)}$$
$$\ldots$$
$$S = a_n + a_{1,\ldots,n-1} \qquad \text{(Passo n-1)}$$

In un architettura parallela, è possibile risolvere questo problema in modo più efficiente[1] poiché vengono sempre eseguite $n-1$ operazioni ma alcune sono effettuate contemporaneamente. A causa di questa simultaneità, i passi temporali diminuiscono rispetto alla soluzione sequenziale. Sfruttare un ambiente parallelo, però, comporta la gestione di nuovi problemi:

- la distribuzione dei dati
- la comunicazione tra i processori

La distribuzione dei dati è il principale problema poiché la comunicazione e la conseguente sincronizzazione vengono gestiti tramite l'ambiente MPI con la sua libreria (capitolo 4).

Nota bene. Nel caso in cui il numero di elementi sia minore o uguale di 20, allora gli elementi da sommare sono dati in input, invece se sono maggiori di 20 allora vengono auto generati.

[1] L'efficienza in ambiente parallelo è un concetto misurabile tramite differenti parametri e il suo studio verrà affrontato nel capitolo 6

Capitolo 3

Descrizione dell'algoritmo

L'idea è dividere il problema in sotto problemi risolvibili nello stesso momento da diversi processori. La risoluzione del problema della somma può essere divisa in diversi step: dividere l'input, distribuirlo ai vari processori, sommare localmente (in ogni processore) i numeri ricevuti e comunicare la somma parziale agli altri processori.

3.1 Divisione e distribuzione dei dati

I problemi di divisione e distribuzione dei dati vengono risolti cercando di dividere omogeneamente gli n numeri tra i p processori, quindi ogni unità dovrebbe avere $\frac{n}{p}$ numeri da sommare. Siano $u_0, u_1, \ldots, u_p$ le unità di elaborazione (processori) con p numero di processori, $e_1, e_2, \ldots, e_n$ gli elementi da sommare con n numero di elementi, allora:

- nel caso in cui $\frac{n}{p}$ abbia come resto $r = 0$, allora ogni processore avrà in memoria esattamente $\frac{n}{p}$ elementi da sommare
- nel caso in cui $\frac{n}{p}$ abbia come resto $r = \alpha$ con $\alpha \neq 0$, allora ogni processore partendo da u_0 incrementerà di uno il suo buffer fino ad avere uno spazio per tutti gli elementi

Esempio. Sia $n = 20$ e $p = 4$, essendo $\frac{n}{p} = 5$ con resto 0, allora ogni processore avrà in memoria esattamente 5 elementi (come mostrato in figura 3.1).

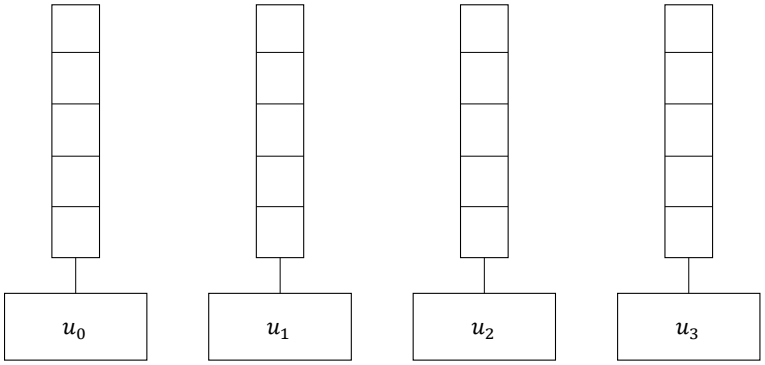


Figura 3.1: Divisione dati con $\frac{n}{p} = 5$ e resto uguale a 0

Esempio. Sia $n = 22$ e $p = 4$, essendo $\frac{n}{p} = 5$ con resto 2, allora u_0 e u_1 avranno in memoria 6 elementi mentre tutti gli altri ne avranno 5 (come mostrato in figura 3.2).

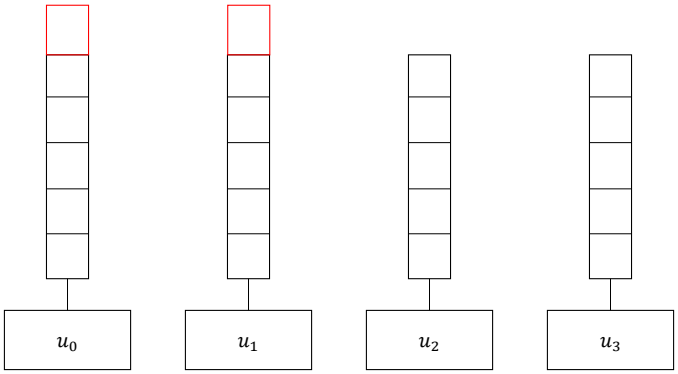


Figura 3.2: Divisione dati con $\frac{n}{p} = 5$ e resto uguale a 2

Terminata la divisione teorica degli input, inizia la distribuzione che, nel caso analizzato, è demandata al processore principale (con ID= 0) che comunica agli altri processori i numeri da sommare tramite funzioni di libreria MPI (capitolo 4).

3.2 Somma locale e come comunicarla

La somma locale, come detto, viene svolta da ogni processore ed è banale, in quanto è una somma sui numeri che il processore ha ricevuto in input.

Listing 3.1: Esempio di somma locale

```
1    for(i = 0; i < n; i++) {
2        sum = sum + elements[i];
3    }
```

Per comunicare la somma parziale, ci sono tre diverse strategie. Mentre la prima può essere sempre applicabile, le altre due dipendono dal numero di processori poiché sono attuabili solo se il numero di processori è una potenza di 2.

La prima strategia è molto semplice poiché, una volta calcolate le somme parziali, a uno a uno ogni processore manda il risultato della somma locale al processore principale che ne fa la somma. Con questa strategia, si risparmia tempo rispetto a un'architettura sequenziale, in quanto i processori effettuano contemporaneamente le somme parziali. Ma, nel momento della comunicazione, c'è un overhead molto consistente causato da: l'attesa di ogni processore per la comunicazione della somma e, una volta effettuata la comunicazione, l'attesa che l'intero algoritmo termini.

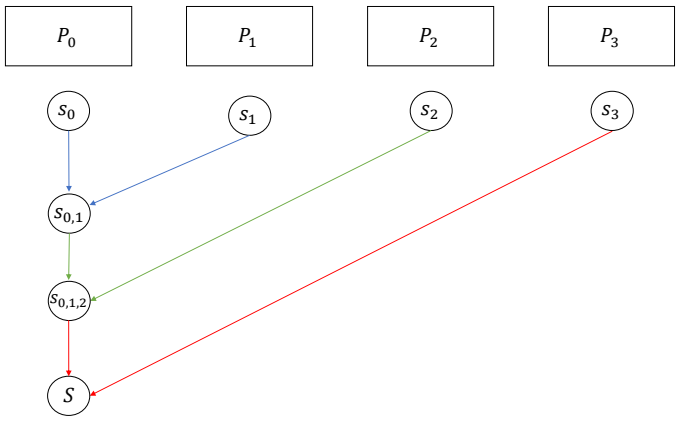


Figura 3.3: Prima strategia di comunicazione

Esempio. Sia $n = 8$, per calcolare la somma sono necessarie 7 operazioni. In un'architettura sequenziale, sono necessari 7 passi temporali (uno per ogni operazione). Invece, in un'architettura parallela con $p = 4$, allora ogni processore ha 2 numeri in memoria. Ogni processore effettua una operazione, quindi in tutto sono 4 operazioni, ma, essendo svolte contemporaneamente, è necessario un solo passo temporale. In seguito vengono effettuate 3 operazioni in 3 passi temporali dal processore principale per sommare le somme parziali.

La seconda strategia differisce dalla prima poiché a ogni passo vengono definite coppie di processori che devono comunicare. Nella coppia, uno manda la propria somma parziale e l'altro la riceve. Il guadagno è nella comunicazione parallela delle coppie, a differenza della prima in cui tutti aspettavano di comunicare al processore principale. Anche in questo approccio è presente un minimo overhead poiché il processore mittente nella coppia, una volta comunicata la somma parziale, deve attendere la fine dell'algoritmo.

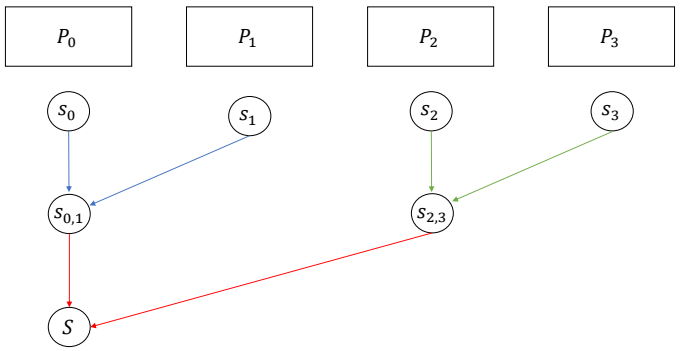


Figura 3.4: Seconda strategia di comunicazione

Esempio. Sia $n = 8$ e $p = 4$, in questa strategia, una volta calcolata la somma parziale (1 passo temporale per 4 operazioni), vengono calcolate $s_{0,1} = s_0 + s_1$ e $s_{2,3} = s_2 + s_3$ contemporaneamente. Sono effettuate 2 operazioni di somma al costo di un solo passo temporale.

La terza strategia è molto simile alla seconda, l'unica differenza è che le coppie si mandano reciprocamente la somma in modo che, alla fine, tutti i processori abbiano memorizzato il risultato della somma totale. In questo caso, le operazioni effettuate in tutto sono maggiori ma i passi temporali necessari sono uguali alla seconda strategia.

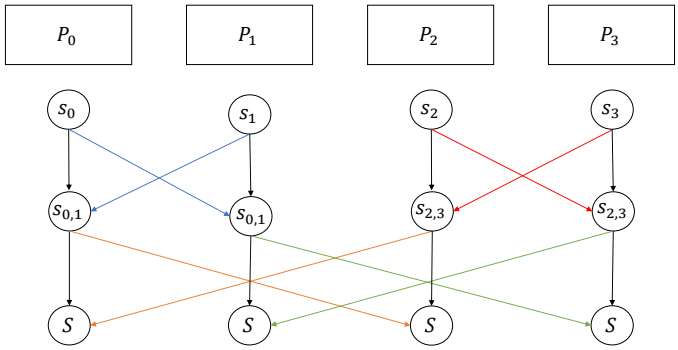


Figura 3.5: Terza strategia di comunicazione in cui a ogni coppia è associato un colore

Capitolo 4

Descrizione delle routine

In questo capitolo, segue l'analisi e la descrizione del codice sorgente con la spiegazione del workflow adottato per rendere il programma semplice, efficiente e user-friendly. È possibile individuare 4 file, ognuno con una funzionalità specifica (senza considerare il file eseguibile):

- *sum-numbers.c*, è il file che contiene il codice sorgente e implementa il `main`, le funzioni necessarie a risolvere il problema e alcune funzioni ausiliarie
- *sum-numbers.h*, è il file header che contiene i prototipi delle funzioni, la dichiarazione delle strutture dati personalizzate e le librerie usate
- *sum-numbers.sh*, è uno script bash di supporto che permette all'utente di passare i parametri in modo interattivo
- *sum-numbers.pbs*, è lo script che contiene le direttive PBS per configurare il job, seguite dalla compilazione del file C e dall'esecuzione dell'eseguibile generato

4.1 File header

Il file header *sum-numbers.h* contiene le librerie usate, due enumerazioni, una struct e i prototipi delle funzioni.

Le librerie usate sono:

- `mpi.h`, libreria standard per l'implementazione di software in ambienti paralleli che segue il modello *Message Passing*. Contiene tutte le funzioni, le variabili e le costanti per l'inizializzazione e la gestione della comunicazione tra processori
- `stdio.h`, libreria standard di C per la gestione dell'input/output
- `stdlib.h`, libreria standard di C che comprende alcune utility per allocare memoria (`malloc()`), per la conversione (`atoi()`) e per la generazione di numeri pseudocasuali

- `string.h`, libreria standard di C per la gestione e la manipolazione delle stringhe
- `math.h`, libreria standard che contiene utility per la gestione delle operazioni matematiche (`log2()`)
- `getopt.h`, libreria che contiene variabili e funzioni per gestire i parametri passati tramite linea di comando
- `ctype.h`, libreria che dichiara funzioni per la gestione dei singoli caratteri (usata per controllare se un carattere è effettivamente un numero con `isdigit()`)
- `time.h`, libreria standard di C per gestire tipi di dato temporali (usata per la generazione di numeri casuali dandogli come seed l'orario attuale con `time()`)

Listing 4.1: Librerie incluse nel codice

```
1  #include <mpi.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include <math.h>
6  #include <getopt.h>
7  #include <ctype.h>
8  #include <time.h>
```

Un `enumeration` è una coppia nome - valore. I valori che può assumere sono ristretti a un insieme d'interi costanti a cui viene associato un nome. Le `enumeration` definite sono: una per il tipo di strategia adottata e una che indica da dove recuperare l'input. Nell'`enumeration Strategy` ogni valore indica una delle strategie applicabili tra quelle esposte nel capitolo 3.

Listing 4.2: Enumeration Strategy

```
1     typedef enum{
2         FIRST_STRATEGY = 1,
3         SECOND_STRATEGY,
4         THIRD_STRATEGY
5     } Strategy;
```

Mentre nell'`enumeration Input` a ogni valore corrisponde un modo in cui l'algoritmo deve recuperare i valori da sommare. I valori possibili sono:

- `NONE`, di default, quando ancora non sono stati letti i parametri da riga di comando
- `FROM_FILE`, i valori da sommare devono essere recuperati da un file "input.txt"
- `RANDOM`, i valori vengono auto-generati randomicamente
- `DEBUG`, i valori da sommare hanno tutti valore 1 in modo da rendere il debugging efficiente

Listing 4.3: Enumeration Input

```
1  typedef enum{
2      NONE,
3      FROM_FILE,
4      RANDOM,
5      DEBUG
6  } Input;
```

Viene, inoltre, definita una struttura dati che memorizza tutte le informazioni e le opzioni necessarie a un singolo processore per risolvere il problema. La struttura dati è una `struct` ed è soprannominata `Utils`. La `struct` ha diversi attributi:

- `processorID`, è l'ID univoco del singolo processore che serve per l'identificazione e i messaggi di log
- `nProcessors`, è il numero di processori che stanno eseguendo il programma
- `strategyType`, è il tipo di strategia da adottare (come esposto precedentemente)
- `sum`, è la somma calcolata dal singolo processore
- `nElementsInOneprocessor`, è il numero di elementi memorizzato da un singolo processore
- `numbersInOneprocessor`, è il puntatore a un array che memorizza i numeri che un singolo processo deve sommare localmente
- `logNProcessors`, è il logaritmo del numero di processori che è necessario nel caso si adotti la seconda o la terza strategia
- `powersOf2`, è il puntatore che memorizza le potenze di 2 ed è necessario nel caso si adotti la seconda o la terza strategia

Nota bene. Il logaritmo e le potenze di 2 sarebbero potute essere calcolate *on-fly* quando vengono utilizzate ma potrebbe inficiare sullo studio e le analisi della strategia risolutiva

Listing 4.4: Struct Utils

```
1  //Utility per un processo
2  typedef struct{
3      int processorID;
4      int nProcessors;
5      Strategy strategyType;
6      int sum;
7      //Numero di elementi per ogni processore
8      int nElementsInOneprocessor;
9      //Array per memorizzare i numeri in ogni processo
10     int* numbersInOneprocessor;
11     //Memorizza il logaritmo in base 2 del numero di processori
12     int logNProcessors;
13     //Memorizza le potenze di 2
14     int* powersOf2;
15 } Utils;
```

A seguire (figura 4.1) vengono definiti i prototipi delle funzioni che sono implementate e usate nel file C.

```
41  /******** CORE* *******/
42  /*Funzione di gestione*/
43  int interpretCommandLine(int, char*[], Strategy*, Input*, int*);
44  void checkStrategy(Utils*);
45  void distributeData(Utils*, int, int[]);
46  void firstStrategy(Utils*);
47  void secondStrategy(Utils*);
48  void thirdStrategy(Utils*);
49
50  /******* INPUT ******/
51  /*Funzioni per generare/recuperare input*/
52  void generateRandomNumbers(int*, int);
53  void generate1(int*, int);
54  void retrieveInput(int*, int, char*);
55
56  /****** UTILS *****/
57  void printError(int, char*);
58  char* inputTypeToString(Input);
59  char* strategyToString(Strategy);
```

Figura 4.1: Prototipi delle funzioni

4.2 File sorgente (.c)

Il file sorgente è diviso in sezioni per rendere il codice più chiaro possibile.

1. **Inizializzazione dell'ambiente MPI** → tramite la funzione `MPI_Init()` viene inizializzato l'ambiente MPI. Successivamente `MPI_Comm_rank()` as-

segna un ID a ogni processo che viene salvato nell'attributo `processorID` della struttura `Utils` e, sempre nella stessa struttura, `MPI_Comm_size()` il numero di processori nell'attributo `nProcessors`. Infine la somma viene inizializzata a 0.

```
18  //Inizializza l'ambiente MPI
19  MPI_Init(&argc, &argv);
20  //Prende l'id dei processori e lo assegna
21  MPI_Comm_rank(MPI_COMM_WORLD, &utilsForProcessor.processorID);
22  //Prende il numero di processori
23  MPI_Comm_size(MPI_COMM_WORLD, &utilsForProcessor.nProcessors);
24
25  utilsForProcessor.sum = 0;
26
27  printf("[PID: %d/%d] Processor initialization\n", utilsForProcessor.processorID, utilsForProcessor.nProcessors);
```

Figura 4.2: Inizializzazione

2. **Configurazione del problema** → il problema viene configurato una volta letti i parametri passati da linea di comando. La configurazione viene gestita per larga parte dal processore principale (ID=0). Il programma, per essere chiamato, necessita di alcune opzioni. Per chiarezza, la sinossi del comando è

```
1       mpiexec -np <n-proc>
2           sum-numbers -s <1,2,3> -n <intero> [-d] [file.txt]
```

in cui `-s` è obbligatorio e indica il tipo di strategia da adottare, `-n` è obbligatorio e indica il numero di elementi da sommare, `-d` è opzionale e indica di avviare il programma in debug mode (i valori degli elementi da sommare sono tutti 1) e l'ultimo parametro è il nome del file da cui recuperare i valori nel caso in cui $n <= 20$. La configurazione inizia con il processore principale che chiama la funzione `interpretCommandLine()` che legge le opzioni elencate e le memorizza nella struttura dati `Utils utilsForOneProcessor` con i dovuti controlli. Segue una versione molto semplificata di questa funzione.[1]

```
1       int interpretCommandLine() {
2           while (opt = /*Lettura argomenti*/) {
3               switch (opt) {
4                   case 's': //Opzione per la strategia
5                       tmp = atoi(optarg);
6                       if (/*Controlli*/) {
7                           *strategyType = tmp;
8                       } else errore
9                       break;
10                  case 'n': //Opzione per il numero di elementi
11                      if (/*Controlli*/) {
12                          *nElements = tmp;
13                          if(*inputType == NONE) {
14                              if(tmp <= 20) {
15                                  *inputType = FROM_FILE;
16                              } else {
```

[1]Per la versione completa riga 112 nel file C

```
17                                      *inputType = RANDOM;
18                                  }
19                              }
20                          } else errore
21                          break;
22                      case 'd':
23                          *inputType = DEBUG;
24                          break;
25                  }
26              }
27              //Check se le opzioni obbligatorie sono state passate
28              return 0;
29          }
```

Dopo aver ottenuto le informazioni necessarie, il processore principale alloca la memoria per salvare i valori degli elementi da sommare e decide in che modo recuperare l'input tramite il controllo dell' attributo `strategyType` di `Utils`.

```
38      //Allocazione dell'array che conterrà i numeri
39      numbers =(int*) malloc(sizeof(int) * nElements);
40
41      //Controlla se l'input deve essere generato o recuperato
42      if(inputType == RANDOM) {
43          generateRandomNumbers(numbers, nElements);
44      } else if (inputType == FROM_FILE) {
45          //Nel caso dovesse essere recuperato da file allora l'ultimo argomento arg[optind] è il nome del file
46          retrieveInput(numbers, nElements, argv[optind]);
47      } else if (inputType == DEBUG) {
48          generate1(numbers, nElements);
49      }
50  }
```

Figura 4.3: Recupero degli elementi da sommare

- `generateRandomNumbers()` è la funzione che genera randomicamente i valori tramite la funzione `rand()` e li salva in `numbers`. Mentre con la funzione `srand(time(0))` si inizializza il seed per la generazione con l'orario attuale

```
324 //Genera numeri casuali e li memorizza in numbers
325 void generateRandomNumbers(int* numbers, int nElements) {
326     int i;
327
328     srand(time(0));
329     printf("[PID 0] Generated numbers:\n");
330     for (i = 0; i < nElements; i++) {
331         numbers[i] = rand()%100;
332         printf("\tElements %d value is %d\n", i, numbers[i]);
333     }
334 }
```

Figura 4.4: Generazione random degli elementi

- `retrieveInput()` recupera gli input da file tramite le funzioni di I/O di C. Il nome del file viene recuperato grazie a una variabile `optind` che permette di leggere gli ultimi parametri rimasti (nel nostro caso il nome del file)

```
336 //Recupera input da file
337 void retrieveInput(int* numbers, int nElements, char* file) {
338     int i;
339     FILE* myFile = fopen(file, "r");
340 
341     if (myFile == NULL) {
342         printError(0, "Can't read file. File name should be input.txt\n");
343         MPI_Abort(MPI_COMM_WORLD, -1);
344     }
345 
346     //Legge i numeri sul file e li memorizza in numbers
347     for (i = 0; i < nElements; i++) { fscanf(myFile, "%d", &numbers[i]); }
348     fclose(myFile);
349 }
```

Figura 4.5: Recupero degli elementi da file

- `generate1()` semplicemente popola l'array con tutti 1

```
351 //Memorizza tutti 1
352 void generate1(int* numbers, int nElements) {
353     int i;
354 
355     for (i = 0; i < nElements; i++) { numbers[i] = 1; }
356 }
```

Figura 4.6: Popolazione dell'array con valori uguali a 1

3. **Broadcast del tipo di strategia e del numero di elementi** → il processore principale manda agli altri processi il numero di elementi da sommare e il tipo di strategia da adottare tramite la funzione di broadcast di MPI. Si è deciso di mandare anche il tipo di strategia in modo che ogni processore si potesse calcolare parallelamente il logaritmo e le potenze di 2. L'alternativa sarebbe stata mandare successivamente anche il logaritmo e le potenze di 2 ma questo approccio è molto inefficiente in quanto le comunicazioni sono le operazioni che impiegano più tempo a essere completate

```
52     //Main processor manda il numero totale degli elementi per permettere a tutti
53     //i processori di calcolarsi il numero di elementi da memorizzare localmente
54     //Potremmo mandare la grandezza calcolata da P0 ma poi ci sarebbero molte comunicazioni
55     MPI_Bcast(&nElements, 1, MPI_INT, 0, MPI_COMM_WORLD);
56     //Main processor manda anche il tipo di strategia da adottare
57     MPI_Bcast(&utilsForProcessor.strategyType, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

Figura 4.7: Broadcast

4. **Controllo della strategia e inizializzazione opzionale di utility** → nel caso in cui la strategia selezionata sia la seconda o la terza, allora vengono inizializzate le variabili `logNProcessors` e l'array `powersOf2`. Tutto questo viene gestito dalla funzione `checkStrategy()` che ha come compito principale quello di verificare se la seconda/terza strategia è attuabile poiché la condizione necessaria è che il numero di processori sia potenza di 2. Quindi viene calcolato il logaritmo tramite la funzione `log2()` di

`math.h`, mentre le potenze di 2 vengono calcolate tramite left-shif su 1. Nel caso non fosse attuabile, viene configurata la prima strategia.

```
180     //Nel caso la strategia fosse la seconda o la terza il numero di processori deve essere potenza di 2
181     if(utilsForProcessor->strategyType == SECOND_STRATEGY || utilsForProcessor->strategyType == THIRD_STRATEGY) {
182         //Casting implicito poichè logNProcessores è intero mentre log2() restituisce double
183         utilsForProcessor->logNProcessors = log2(utilsForProcessor->nProcessors);
184         //Nel caso il numero di processori non sia potenza di 2 si applica la prima strategia
185         //Controllo se troncando il double rimane uguale (quindi potenza di 2) oppure no
186         if (log2(utilsForProcessor->nProcessors) != utilsForProcessor->logNProcessors) {
187             utilsForProcessor->strategyType = FIRST_STRATEGY;
188 
189             //Stampa unica (main processor) del tipo di input, numero di elementi e strategia
190             if (utilsForProcessor->processorID == 0) {
191                 printf("\n\n*** WARNING ***\nStrategy selected is incompatible with the processors number so the f
192             }
193         } else {
194             //Memorizza le potenze di 2 in un array in modo che non inficino nel calcolo del tempo dell'algoritmo
195             utilsForProcessor->powersOf2 = malloc(sizeof(int) * (utilsForProcessor->logNProcessors + 1));
196             for (i = 0; i <= utilsForProcessor->logNProcessors; i++) {
197                 //Le potenze di 2 vengono calcolate tramite lo shift a sinistra
198                 utilsForProcessor->powersOf2[i] = (1 << i);
199             }
200         }
201     }
```

Figura 4.8: Controllo della strategia e calcolo delle utility

5. **Distribuzione dei dati** → gli elementi da sommare vengono distribuiti seguendo l'approccio descritto nella sezione 3.1. Viene calcolato il numero di elementi per ogni processore tramite la divisione tra il numero di elementi totali e il numero di processori. Nel caso il resto sia diverso da 0, allora vengono aumentate le grandezze degli array dei primi processi in modo che contengano gli elementi in più.

   Nota bene. Il codice non viene postato poiché molto ampio ma può essere trovato nel file sorgente alla riga 205

6. **Somma locale**

7. **Applicazione della strategia** → viene applicata la strategia selezionata. Ogni strategia è implementata in una funzione

   - `firstStrategy()`
   - `secondStrategy()`
   - `thirdStrategy()`

Ci sono altre funzioni utili implementate come le `toString()` che trasformano il valore delle enumeration `Strategy` e `Input` in delle stringhe. Un ulteriore funzione implementata è `printError()` che permette di avere un messaggio standard di errore con un'eventuale soluzione al problema.

4.3 Script (.sh)

Una volta compilato il file sorgente, il programma è eseguibile da linea di comando come riportato qui.

```
1 mpiexec -np <n-proc>
2     sum-numbers -s <1,2,3> -n <intero> [-d] [file.txt]
```

Per semplificare l'uso del programma, è stato implementato uno script bash molto semplice che fornisce un'interfaccia all'utente. Le varie opzioni e i parametri saranno richiesti, interattivamente, in questa interfaccia. Lo script, una volta acquisiti i parametri, lancerà in esecuzione il programma. Nel caso in cui il programma dovesse essere seguito su cluster, allora lo script eseguirà a sua volta lo script PBS.

Capitolo 5

Descrizione dei test

I test sono stati effettuati sullo *SCoPE Datacenter*, infrastruttura con risorse di calcolo e storage accessibili mediante paradigmi di calcolo distribuito. Il datacenter ha messo a disposizione 8 nodi DELL blade M600, ognuno dotato di 2 processori Intel Xeon quadcore. Le varie esecuzioni sono state lanciate in batch tramite *Portable Batch System* (PBS). Tra i test rientrano anche gli esempi d'uso, nel quale è stato utilizzato un PBS più semplice e dei log più verbosi per rendere chiaro il più possibile il funzionamento del programma. Il PBS tipo utilizzato per gli esempi d'uso è questo:

Listing 5.1: Script PBS

```
1  #!/bin/bash
2
3  #PBS -q studenti
4  #PBS -l nodes=8:ppn=8
5  #PBS -o $PROJECT.out
6  #PBS -e $PROJECT.err
7
8  sort -u $PBS_NODEFILE > hostlist
9
10 cat $PBS_NODEFILE
11 PBS_O_WORKDIR=$PBS_O_HOME/$PROJECT
12
13 #Controlli per la configurazione
14
15 mpicc -o $PBS_O_WORKDIR/$PROJECT $PBS_O_WORKDIR/$PROJECT.c
16
17 mpiexec -machinefile hostlist -np ${NP} $PBS_O_WORKDIR/$PROJECT
```

e viene chiamato da questo comando

```
1  qsub -v PROJECT="<name>",NP=<processors>,S=<1,2,3>,
2         nELEM=<intero>,FILE="<input.txt>" script.pbs
```

in cui `PROJECT` è una variabile che prende il nome del progetto, `NP` è una variabile che memorizza il numero di processori da usare, `S` è la strategia da adottare, `nELEM` è il numero di elementi da sommare, `FILE` è il file opzionale da cui recuperare l'input.

5.1 Esempi di uso

Sia n numero di elementi e p numero di processori, di seguito sono forniti i casi d'uso per:

1. $n = 10$, $p = 8$ con l'applicazione della prima strategia
2. $n = 22$, $p = 4$ con l'applicazione della seconda strategia
3. $n = 1000000$, $p = 8$ con l'applicazione della terza strategia
4. $n = 22$, $p = 6$ provando ad applicare la terza strategia

Il primo esempio viene presentato per chiarire l'uso dell'interfaccia tramite lo script nella sezione 4.3. Viene lanciato lo script e, essendo $n \leqslant 20$, vengono chiesti gli elementi da sommare. Una volta inseriti gli elementi, lo script genera un file in cui saranno memorizzati. Infine lo script chiede gli ultimi parametri e lancia il PBS.

```
[SCRFNC00B@ui-studenti sum-numbers]$ ./sum-numbers.sh
**********************
*                    *
* SUM NUMBERS PROGRAM *
*                    *
**********************

Enter the number of elements you want to sum: 10
Insert the elements to sum
Insert the element 0: 1
Insert the element 1: 2
Insert the element 2: 3
Insert the element 3: 4
Insert the element 4: 5
Insert the element 5: 6
Insert the element 6: 7
Insert the element 7: 8
Insert the element 8: 9
Insert the element 9: 10
1) First strategy
2) Second strategy
3) Third strategy
4) Quit
Select your strategy (1, 2, 3): 1
First strategy selected.
Debug mode enabled (y/n)? n
---------------------------------------

4010815.torque02.scope.unina.it
```

(a) Lancio dello script .sh

```
[PID: 0/8] Processor initialization
[PID: 2/8] Processor initialization
[PID: 4/8] Processor initialization
[PID: 1/8] Processor initialization
[PID: 3/8] Processor initialization
[PID: 6/8] Processor initialization
[PID: 7/8] Processor initialization
[PID: 5/8] Processor initialization
---------- PROBLEM AND RESOLUTION CONFIGURATION ----------
-Input type: Read from command line
-Number of elements: 10
-Strategy type: First strategy


[PID 0] *** RESULT *** Total sum value is 55
```

(b) Risultato

Figura 5.1: Primo esempio d'uso

Il secondo esempio d'uso mostra il funzionamento del programma con numeri generati randomicamente (in quanto $n > 20$), cambiando il numero di processori e applicando la seconda strategia. È stato scelto un numero piccolo per verificare in modo più semplice la correttezza della somma.

Nota bene. Da questo esempio d'uso a seguire non verrà più usato lo script .sh ma sarà eseguito direttamente il programma passandogli i parametri in input

```
--------- OUTPUT ------------
[PID: 0/4] Processor initialization
[PID: 1/4] Processor initialization
[PID: 3/4] Processor initialization
[PID 0] Generated numbers:
        Elements 0 value is 71
[PID: 2/4] Processor initialization
        Elements 1 value is 64
        Elements 2 value is 50
        Elements 3 value is 83
        Elements 4 value is 56
        Elements 5 value is 38
        Elements 6 value is 34
        Elements 7 value is 91
        Elements 8 value is 44
        Elements 9 value is 19
        Elements 10 value is 58
        Elements 11 value is 85
        Elements 12 value is 97
        Elements 13 value is 87
        Elements 14 value is 34
        Elements 15 value is 75
        Elements 16 value is 0
        Elements 17 value is 78
        Elements 18 value is 30
        Elements 19 value is 17
        Elements 20 value is 30
        Elements 21 value is 64
---------- PROBLEM AND RESOLUTION CONFIGURATION ----------
-Input type: Randomically generated
-Number of elements: 22
-Strategy type: Second strategy


[PID 0] *** RESULT *** Total sum value is 1205
```

Figura 5.2: Secondo caso d'uso

Il terzo caso d'uso mostra il funzionamento della terza strategia con il numero massimo di processori e un input abbastanza grande ($n = 1000000$). In questo caso, verrà eseguito il programma in debug mode (valori tutti 1) per verificare la correttezza della somma.

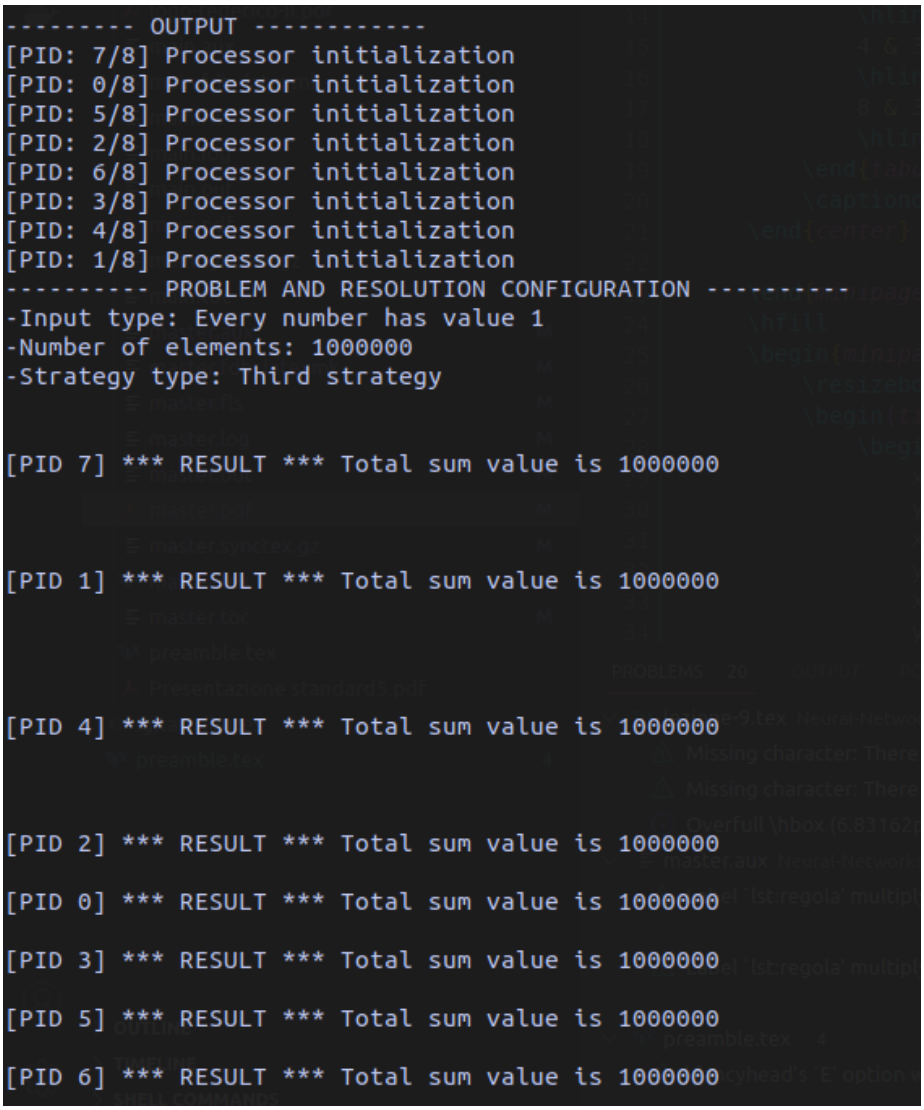


Figura 5.3: Terzo caso d'uso

Il quarto caso evidenzia che l'algoritmo usa la prima strategia nel caso in cui il numero di processori non sia una potenza di 2.

```
--------- OUTPUT ------------
[PID: 0/6] Processor initialization
[PID 0] Generated numbers:
        Elements 0 value is 74
        Elements 1 value is 34
        Elements 2 value is 72
        Elements 3 value is 59
        Elements 4 value is 88
        Elements 5 value is 17
        Elements 6 value is 1
        Elements 7 value is 35
        Elements 8 value is 96
        Elements 9 value is 18
        Elements 10 value is 56
        Elements 11 value is 57
        Elements 12 value is 44
        Elements 13 value is 21
        Elements 14 value is 84
        Elements 15 value is 56
        Elements 16 value is 38
        Elements 17 value is 63
        Elements 18 value is 13
        Elements 19 value is 14
        Elements 20 value is 73
        Elements 21 value is 62
[PID: 3/6] Processor initialization
[PID: 5/6] Processor initialization
[PID: 4/6] Processor initialization
[PID: 1/6] Processor initialization
[PID: 2/6] Processor initialization


*** WARNING ***
Strategy selected is incompatible with the processors number so the first strategy is set by default
---------- PROBLEM AND RESOLUTION CONFIGURATION ----------
-Input type: Randomically generated
-Number of elements: 22
-Strategy type: First strategy


[PID 0] *** RESULT *** Total sum value is 1075
```

Figura 5.4: Quarto caso d'uso

Capitolo 6

Analisi delle performance

Per eseguire l'analisi, sono stati presi dieci tempi d'esecuzione dell'algoritmo di risoluzione e ne è stata fatta la media, in modo da avere un risultato con un margine d'errore più basso. Questo non significa che i dati presi sono corretti al 100%, anzi, successivamente, sono state fatte alcune osservazioni su risultati inaspettati e su eventuali bug del sistema. I test sono stati eseguiti per studiare il tempo impiegato dall'algoritmo al variare dei processori con in aggiunta il calcolo e l'analisi dello speed-up e dell'efficienza. Per rendere più robusto lo studio, l'algoritmo è stato eseguito con diverse dimensioni: un milione di dati, 10 milioni di dati e 100 milioni di dati. Per rendere il tutto automatizzato, è stato implementato uno script PBS che, oltre a definire le direttive PBS come quello precedentemente presentato, calcola automaticamente tutti i dati necessari e li fornisce in output tramite una tabella.

```
#Direttive PBS e compilazione uguali

#Per gestire le float operation
export LC_NUMERIC=C

#Tempo impiegato dall'algoritmo con 1 processore
t1=0
efficiency=0

#Incrementa il numero processori
for (( p=1; p<=8; p++ )) do
    sum=0 #Somma di 10 tempi

    for (( i=0; i<10; i++ )) do
        #Esecuzione algoritmo e presa del tempo con awk
        tmp=$(mpiexec -machinefile hostlist -np $p $PROJECT
        -n $nELEM -s 1 | awk '{if (NR == 2) print $4}')

        #Somma dei 10 valori
        sum=$(echo "$sum $tmp" | awk '{printf("%.5f", $1+$2)}')
    done
    average=$(echo "$sum" | awk '{printf("%.6f", $1/10)}')

    if [ $p -eq 1 ]; then
        t1=$average #T(1) settato
        speedup=1
        efficiency=1
    else
        tp=$average
        #Calcolo speed-up T(1)/T(P)
        speedup=$(echo "$t1 $tp" | awk '{printf("%.6f", $1/$2)}')
        #Calcolo efficienza S(P)/P
        efficiency=$(echo "$speedup $p" | awk '{printf("%.6f", $1/$2)}')
    fi
```

6.1 Analisi su 1 milione di elementi

Queste sono le tabelle da cui sono stati generati i grafici per l'analisi su 1 milione di elementi al variare delle strategie e del numero di processori.

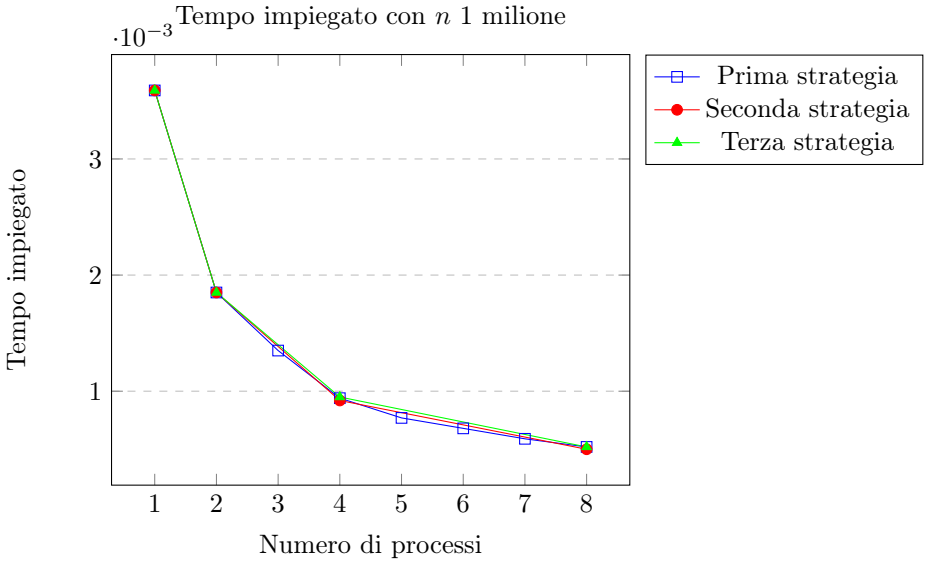
Tabella 6.1: 1^a strategia con 1 milione

p	$T(p)$	$S(p)$	$E(p)$
1	0.00359	1	1
2	0.00185	1.935	0.967
3	0.00135	2.863	0.954
4	0.00094	3.808	0.952
5	0.00077	4.648	0.929
6	0.00068	5.279	0.879
7	0.00059	6.084	0.869
8	0.00052	6.884	0.860

Tabella 6.2: 2^a e 3^a strategia con n 1 milione

p	$T(p)$	$S(p)$	$E(p)$
1	0.00358	1	1
2	0.00185	1.935	0.967
4	0.00092	3.902	0.975
8	0.00050	7.160	0.895
1	0.00359	1	1
2	0.00189	1.902	0.951
4	0.00095	3.784	0.946
8	0.00052	6.884	0.860

Il grafico sottostante conferma le aspettative, infatti all'aumentare del numero di processori il tempo cala. Da notare soprattutto il passaggio dall'algoritmo eseguito da un processore (quindi sequenziale) e lo stesso eseguito da due processori. La pendenza di questo passaggio evidenzia il netto miglioramento dell'algoritmo se eseguito in parallelo. In questo grafico, non vengono evidenziate le differenze tra le tre strategie, anche se nelle tabelle è possibile vedere un minimo miglioramento tra la prima strategia e le altre. Inoltre dal quarto processore fino ad arrivare all'ottavo, il tempo sembra diminuire in modo lineare.

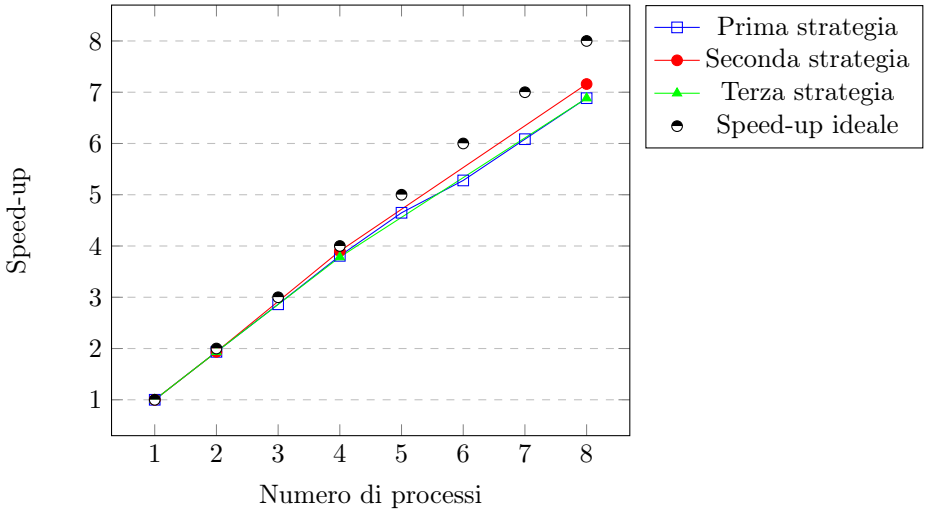


Lo speed-up misura la riduzione del tempo di esecuzione di un algoritmo parallelo rispetto all'esecuzione dello stesso algoritmo su un solo processore (sequenziale) ed è calcolabile come

$$\frac{T(1)}{T(p)}$$

In questo grafico sottostante è possibile osservare come l'aumento dello speed-up sembra essere lineare e si riescono a percepire le piccole differenze che ci sono tra le tre strategie. Infatti la seconda strategia rispetto alla prima, ha uno speed-up maggiore (visibile sul grafico) con 8 processori. Anche con 4 processori, è possibile cogliere, seppur minimamente, questa differenza Minima invece la differenza tra la prima e la seconda. L'aspettativa era che la terza strategia fosse allineata con la seconda, invece è molto simile alla prima. Questo comportamento, probabilmente, è causato dal maggiore tempo dedicato alle comunicazioni poichè (come visto nel capitolo 3) la terza strategia effettua più comunicazioni per rendere il risultato disponibile a tutti i processori. Inoltre all'aumentare dei processori, lo speed-up tende ad allontanarsi dallo speed-up ideale $S(p)^{id} = p$.

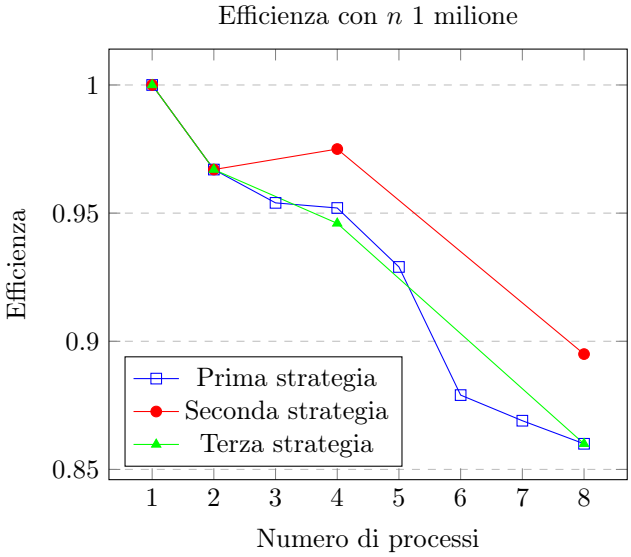
Speed-up con n 1 milione



L'efficienza misura quanto l'algoritmo sfrutta il parallelismo del calcolatore e si calcola con

$$\frac{S(p)}{p}$$

Come da aspettative, fissata la dimensione, all'aumentare dei processori l'efficienza diminuisce poichè la parte parallela di un algoritmo diminuisce. Quindi il grafico segue la legge di Amdahl. Inaspettato è il comportamento dell'algoritmo con 4 processori. Come detto all'inizio, i dati non sono corretti al 100% e, probabilmente, questa misurazione è stata falsata da un valore che ha invalidato poi il calcolo della media giusto. In generale è possibile notare che la seconda strategia è la più efficiente all'aumentare dei processori.



6.2 Analisi su 10 milioni di elementi

Queste sono le tabelle da cui sono stati generati i grafici per l'analisi su 10 milioni di elementi al variare delle strategie e del numero di processori.

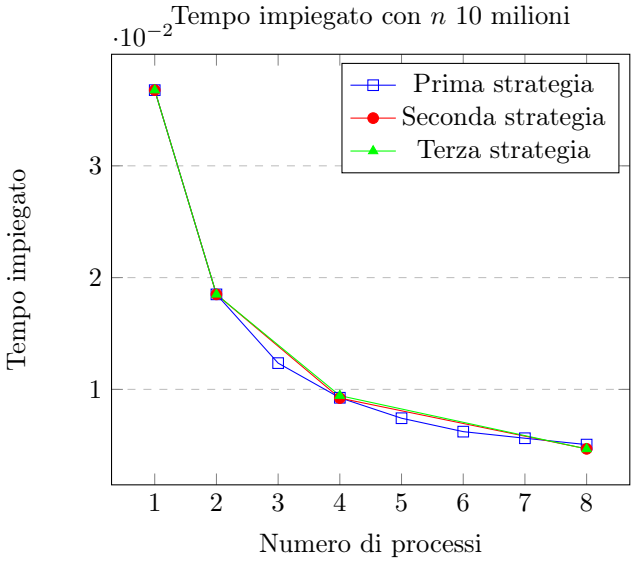
Tabella 6.3: 1^a strategia con 10 milioni

p	$T(p)$	$S(p)$	$E(p)$
1	0.03677	1	1
2	0.01850	1.987	0.9939
3	0.01235	2.976	0.9921
4	0.00925	3.972	0.9932
5	0.00742	4.954	0.9909
6	0.00622	5.915	0.9858
7	0.00563	6.531	0.9330
8	0.00506	7.265	0.9081

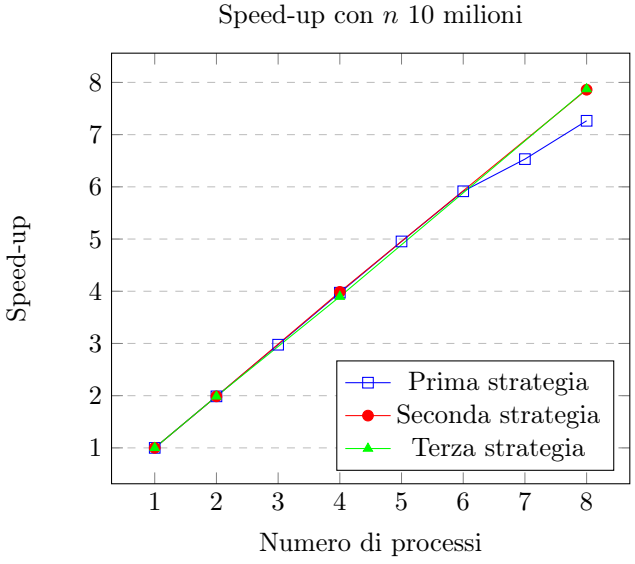
Tabella 6.4: 2^a e 3^a strategia con n 10 milioni

p	$T(p)$	$S(p)$	$E(p)$
1	0.03677	1	1
2	0.01850	1.987	0.9939
4	0.00922	3.988	0.9970
8	0.00468	7.860	0.9826
1	0.03677	1	1
2	0.01850	1.987	0.9939
4	0.00930	3.953	0.9844
8	0.00467	7.873	0.9842

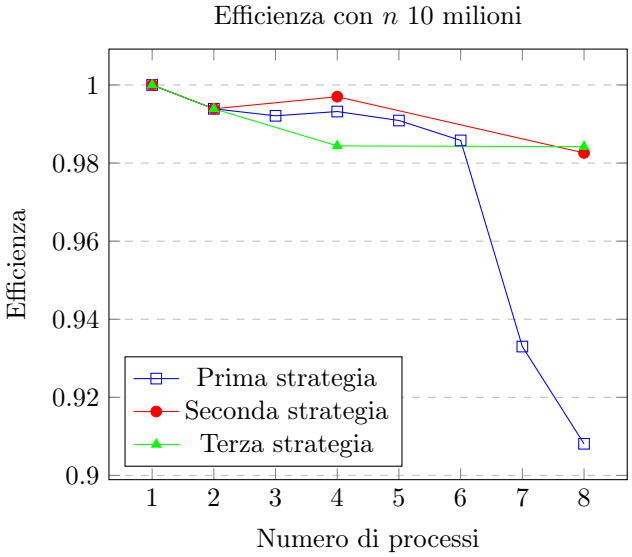
Il grafico conferma, anche all'aumentare della dimensione dell'input, l'osservazione vista nella sezione precedente ovvero che all'aumentare del numero di processori, il tempo impiegato scende inevitabilmente.



Lo stesso vale per il grafico dello speed-up. All'aumentare del numero di processori lo speed-up cresce cosi come la differenza per raggiungere lo speed-up ideale. Anche se, in questo caso, la seconda e la terza strategia sembrano non degradarsi. Essendo l'input più ampio, servono più processori per notare il degrado. Quindi, da questo grafico, è già possibile prevedere che la seconda e la terza strategia rimarranno molto efficienti.



Come volevasi dimostrare, l'efficienza della seconda e della terza strategia degrada di molto poco all'aumentare dei processori. In generale, come nella precedente sezione, l'efficienza si riduce. Questo degrado è minore rispetto a quello del grafico che misurava l'efficienza su 1 milione di dati.



6.3 Analisi su 100 milioni di elementi

Queste sono le tabelle da cui sono stati generati i grafici per l'analisi su 10 milioni di elementi al variare delle strategie e del numero di processori.

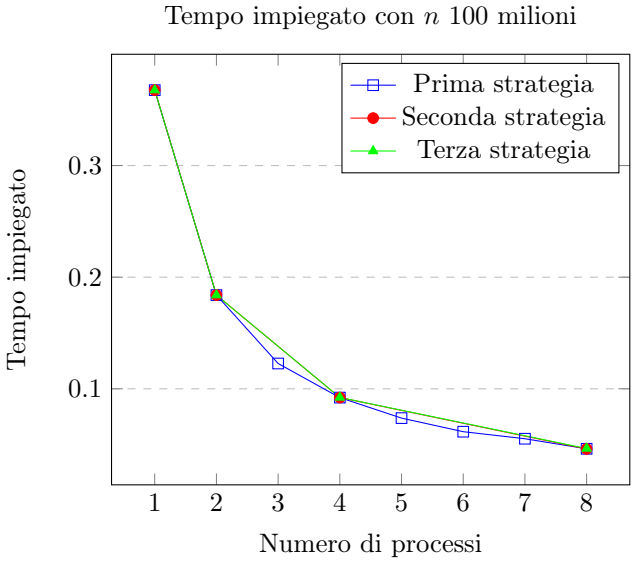
Tabella 6.5: 1^a strategia con 100 milioni

p	$T(p)$	$S(p)$	$E(p)$
1	0.36755	1	1
2	0.18395	1.998	0.9990
3	0.12268	2.996	0.9986
4	0.09214	3.989	0.9972
5	0.07376	4.982	0.9965
6	0.06159	5.967	0.9946
7	0.05300	6.934	0.9907
8	0.04628	7.941	0.9927

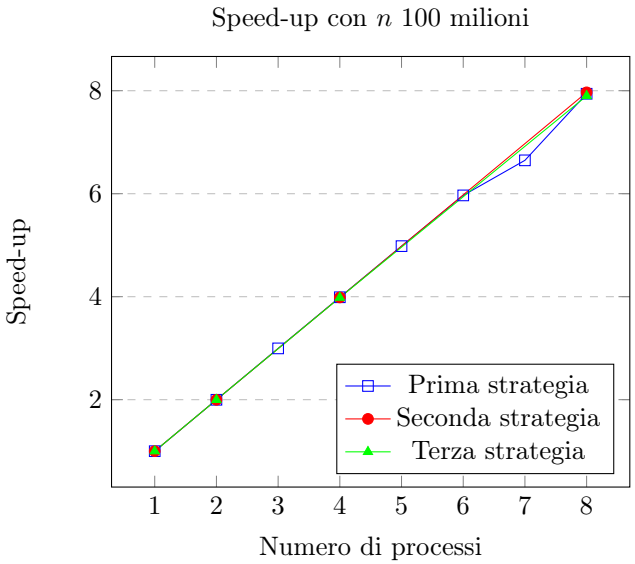
Tabella 6.6: 2^a e 3^a strategia con n 100 milioni

p	$T(p)$	$S(p)$	$E(p)$
1	0.36755	1	1
2	0.18395	1.998	0.9990
4	0.09218	3.986	0.9967
8	0.04612	7.968	0.9960
1	0.36755	1	1
2	0.18395	1.998	0.9990
4	0.09227	3.983	0.9958
8	0.04653	7.899	0.9874

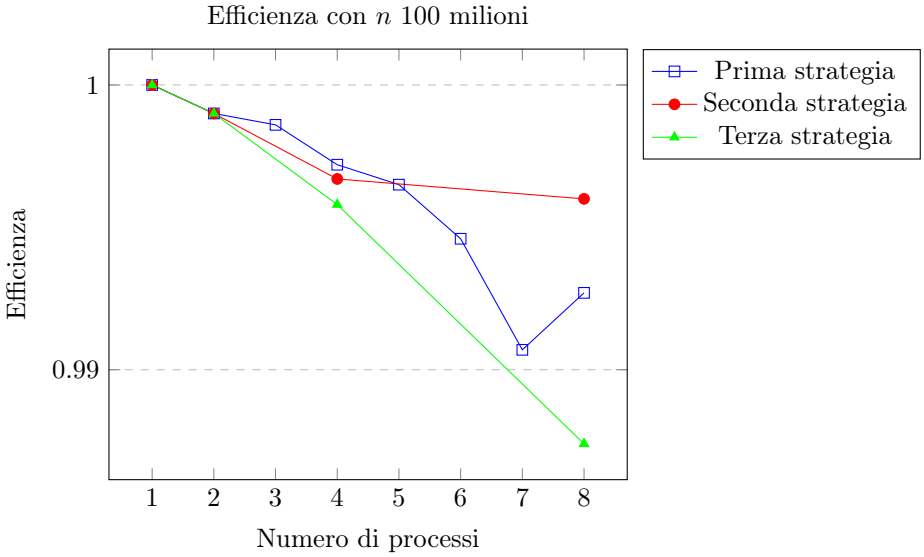
Anche qui, il tempo impiegato diminuisce all'aumentare dei processori.



La differenza tra le strategie si assottiglia ancora di più nel grafico dello speed-up con 100 milioni di dati, infatti è difficile notarla rispetto ai precedenti grafici dello speed-up con 1 e 10 milioni di numeri.



Tutte e 3 le strategie sono molto vicine a livello di efficienza. Dal grafico sottostante, si notano delle piccole differenze che però sono irrilevanti in quanto i punti cadono tutti tra 0.99 e 1 quindi in uno spazio molto ristretto.


Efficienza con n 100 milioni
Efficienza
1
0.99
1
2
3
4
5
6
7
8
Numero di processi
Prima strategia
Seconda strategia
Terza strategia