Università degli studi di Napoli Federico II

SCUOLA POLITECNICA E DELLE SCIENZE DI BASE

DIPARTIMENTO DI INGEGNERIA ELETTRICA E TECNOLOGIE DELLINFORMAZIONE

CORSO DI LAUREA MAGISTRALE IN INFORMATICA

Parallel and Distributed Computing
Elaborato 1

Professore:
Giuliano Laccetti

Matricola:
Alessandro Schiavo
N97000423

ANNO ACCADEMICO 2022 / 2023

Indice

1 Definizione ed analisi del problema **5**
1.0.1 Analisi ambiente 5
1.0.2 Caratteristiche del problema 6

2 Definizione dell'agoritmo **7**
2.0.1 Strategia I 8
2.0.2 Strategia II 8
2.0.3 Strategia III 8

3 Input e Output **9**

4 Indicatori di errore **11**

5 Subroutine **13**
5.0.1 Lettura degli input 13
5.0.2 Decodifica dell'input 13
5.0.3 Distribuzione dei dati 14
5.0.4 Lettura delle performance 14
5.0.5 Strategie 14

6 Analisi dei tempi **19**

7 Esempi d'uso **21**

8 Appendice **23**

Definizione ed analisi del problema

1.0.1 Analisi ambiente

Si intende definire un algoritmo in grado di utilizzare appieno i calcolatori MIMD a memoria distribuita per eseguire una somma di numeri reali. I calcolatori MIMD (Multiple Instruction Multiple Data) a memoria distribuita si basano su architetture a memoria condivisa virtuale dove le diverse unità di elaborazione eseguono istruzioni su dati diversi. Tali unità sono associate ad una memoria privata, mentre i dati da condividere sono trasmessi con messaggi sincroni e asincroni. L'architettura MIMD quindi presenta caratteristiche ottimali per un ambiente di calcolo parallelo dove :

- le memorie non condivise non presentano problemi di sincronizzazione
- solo i dati da condividere sono trasmessi
- facile modifica del numero di unità di elaborazione

Al contempo gli svantaggi sono multipli e derivano dal tipo di architettura, che demanda molteplici responsabilità allo sviluppatore, come: bilanciare il carico di lavoro tra i vari nodi (coppia memoria-unità) e distribuire i dati necessari per l'elaborazione. L'operazione deve quindi sfruttare l'ambiente di sviluppo individuando la soluzione migliore in base ai nodi presenti e agli input forniti.

In ambiente parallelo la complessità di tempo non è in grado di misurare l'efficienza degli algoritmi al variare del numero di processi, dato che non è proporzionale al numero di passi compiuti. Difatti ogni operazione è scomponibile in una componente sequenziale e una parallelizzabile.

$$T(n) = T_s + \frac{T_c}{p} + T_o(p) \text{ , con } T_o > 0 \text{ se } p > 1$$

Dove T_s risulta l'insieme delle operazioni esclusivamente sequenziali, T_c l'insieme delle istruzioni eventualmente simultanee sul numero di processi p e $T_o(p)$ il costo della comunicazione che è strettamente correlato al numero di processi.

Dall'analisi dei tempi si evidenzia che al crescere del numero di processi, a causa della presenza di $T_o(p)$, non necessariamente corrisponde una riduzione del tempo di esecuzione totale. Risulta quindi inevitabile analizzare l'efficienza con strumenti adeguati all'ambiente parallelo.

La funzione *speed-up* $S(p)$ indica la riduzione del tempo di esecuzione da sequenziale a parallelo dimostrando quanto l'implementazione si presti alla parallelismo; mentre la funzione di *efficienza* $E(p)$ indica la percentuale con la quale l'implementazione impiega le risorse a disposizione.

1.0.2 Caratteristiche del problema

L'implementazione deve leggere i seguenti input:

- N : numero di elementi da sommare :
  - se $N > 20$ l'algoritmo genera un insieme di numeri reali di cardinalità N oppure
  - se $N \leq 20$ l'algoritmo legge i numeri in input
- P strategia di comunicazione tra processi da applicare
- ID identificativo del processo che stampa il risultato :
  - se $ID = -1$ tutti i processi stampano il risultato
  - se $0 \leq ID \leq (P - 1)$ il processo indicato stampa il risultato

Definizione dell'agoritmo

L'operazione somma di numeri reali presenta una struttra del tipo : lettura degli addendi e un ciclo di somme parziali per ottenere il risultato finale. L'operazione assume dunque le caratteristiche di un albero binario, dove le foglie sono gli addendi e i nodi dello stesso livello sono somme parziali ricavate dal livello precedente; segue che processi diversi possono eseguire rami diversi dell'albero e scambiare le informazioni solo quando necessario.

La comunicazione tra processi è gestita con la libreria MPI ed è impiegata per inizializzare l'ambiente per i processi disponibili e lo scambio di messaggi secondo varie strategie di comunicazione.

Di seguito i macro passaggi che l'algoritmo compie.

```
somma() {
  exit_status = check_input(...);

  if (exit_status != 0) {
    exit(exit_status);
  } else if (num_processi == 1) {
    sequenziale(...);
  }
  numero_elementi_processo = calculate_elem_proc(...);
  parse_input(...);
  buffer = local_calculation(...);
  communication_strategy(...);
  print_result(...);
}
```

La funzione `communication_strategy()` definisce la strategia da utilizzare secondo l'input P e di conseguenza tutti i processi, dopo aver calcolato la propria somma parziale, eseguendo scambi di messaggi secondo tre tipi di implementazione.

2.0.1 Strategia I

La prima strategia demanda ad un singolo processo il compito di sommare le somme parziali dei vari processi. Ponendo come processo delle somme parziali $id = 0$, si ottiene che:

- i processi con $1 \leq id \leq (p-1)$ restano inutilizzati durante l'esecuzione dei livelli $l : 0 \leq l \leq (h-1)$
- a fine computazione solo il processo $id = 0$ contiene il risultato finale, aggiungendo un ulteriore scambio di messaggi in caso di lettura del risultato da altri processi
- il numero dei processi è proporzionale all'altezza dell'albero poichè un numero maggiore di processi implica ulteriori livelli con somme parziali da aggiungere al risultato finale

2.0.2 Strategia II

Per la seconda strategia i processi sono posti in gruppi di due e generalmente il primo riceve la somma parziale s_1 del secondo e comunica ad un altro gruppo di processi la somma $s_0 + s_1$. L'albero che si ottiene è un albero pieno con le foglie sullo stesso livello, quindi con altezza h minima ottenibile $O(log_n)$. Tale classe di alberi binari ha però una limitazione : il numero delle foglie è pari e quindi il numero di processi. Come per la prima strategia, il risultato è memorizzato solo in un processo.

2.0.3 Strategia III

La rappresentazione grafica della terza strategia è identica alla seconda, questo perchè le somme parziali sono calcolate secondo lo stesso principio, ma con l'aggiunta che sono eseguite ulteriori comunicazioni tra rami diversi dell'albero, in modo tale che ogni foglia (processo) possegga il risultato totale. Le limitazioni che valgono per la seconda sono applicate anche alla terza strategia.

Input e Output

Indicatori di errore

Sono presenti diversi indicatori di errore relativi al numero di input fornito, alla qualità dei valori e applicati eventuli controlli di relazione tra di essi.

Come indicato alla sezione1.0.2 gli input sono :

1. N : numero di elementi da sommare
2. p : strategia di comunicazione dei messaggi
3. id : identificativo del processo che stampa il risultato
4. $s_1, s_2, .., s_N$ valori d'input se $N < 21$

Errori numero elementi

Verifica	**Descrizione**
`N > 0`	Verifica numero di elementi maggiore di zero
Posto `M` il numero di elementi effettivamente passati in input: `N != M AND N < 21`	Se indicato $N < 21$ è compito dell'utente fornire in input lo stesso numero di numeri reali
Posto `R` il numero di processi : `(N/2) < R`	Devono esser presenti almeno due numeri reali per processo

Tabella 4.1: Indicatori di errore per il numero di elementi

Errori strategia

Verifica	Descrizione
`P < 1 OR P > 3`	Verifica che numero indicato corrisponda alla prima , seconda o terza strategia
Posto `P` il numero di processi e `IS_POW(P)` una funzione che determina le potenze di due: `S != S1 AND IS_POW(P)`	Se la strategia è la seconda o la terza, il numero di processi deve esser potenza di due

Tabella 4.2: Indicatori di errore per strategia

Errori identificativo

Verifica	Descrizione
`ID < -1 OR ID > (P-1)`	Posto `P` il numero di processi, determina che l'id indicato non sia al di fuori del range

Tabella 4.3: Indicatori di errore identificativo

Warnings

I *warnings* sono delle verifiche aggiuntive eseguite e un loro fallimento non implica la terminazione dell'algoritmo, ma eventualmente sono modificati degli input per proseguire.

Verifica	Descrizione
Posto `P` il numero di processi, `P = 1`	Avvisa a video che il calcolo sarà eseguito sequenzialmente
Posto `S3` come terza strategia: `S \neq S3 AND (ID != 0 OR ID = -1)`	Se selezionata la prima o seconda strategia solo il primo processo contiene il risultato da stampare, quindi se in input è specificato un processo diverso dal primo allora la verifica pone `ID = 1`

Tabella 4.4: Indicatori di errore identificativo

Subroutine

Le istruzioni che compongono l'operazione somma di numeri reali sono distribuite in micro operazioni (subroutine) che facilitano l'identificazione delle fasi di calcolo. Le principali subroutine sono elencate sotto forma di istruzioni e correlate da un breve riassunto delle principali caratteristiche ed eventuali implementazioni della libreria.

5.0.1 Lettura degli input

```
void check_input(int memum,
    int * exit_status,
    int argc,
    int * strategy,
    char ** argv,
    int * num_items_input,
    int * id);
```

La funzione demanda esclusivamente al primo processo la lettura degli input, eseguendo semplici controlli di qualità e quantità. I controlli determinano una serie di valori che sono condivisi ai processi dello stesso communicator, in particolare `exit_status` che individua casi di errori di terminazione del programma. Per la condivisione dei dati, sono eseguite in serie chiamate alla funzione della libreria `MPI_Bcast`.

5.0.2 Decodifica dell'input

```
void parse_input(char ** argv,
    int memum,
    int num_data_proc,
    int num_total_items,
    double ** recv_buffer);
```

La lista dei numeri reali inizializzabile in due modi : i numeri possono esser forniti dall'utente oppure esser generati casualmente.

Nel primo caso il processo con $id = 0$ legge un numero n di elementi , $2 \leq n \leq 20$, e li converte da stringhe di input a variabili di tipo `double`.

La cardinalità ristretta di n garantisce di non creare tempi di attesa della decodifica abbastanza lunghi da non sfruttare in maniera adeguata tutti i processi. Una volta decodificati gli input è chiamata la funzione `distribuite_data`

Nel secondo caso invece, ogni processo genera e memorizza autonomamente una quantità variabile di numeri casuali nella propria lista di riferimento. Il numero di elementi per processo è precedentemente determinato.

5.0.3 Distribuzione dei dati

```
void distribuite_data(int memum,
    double * send_buffer,
    int num_data_proc,
    double ** recv_buffer);
```

La subroutine distribuisce `send_buffer` ai processi del communicator utilizzando le funzioni della libreria `MPI_Gather` e `MPI_Scatterv`.

La prima raccoglie il numero di reali che ogni processo si spetta e memorizza le informazioni in una lista, dove la posizione i corrisponde al processo p_i.

La seconda funzione distribuisce la lista dei numeri reali impiegando due liste: una identifica il numero di reali che ogni processo si aspetta (determinato precedentemente) e la seconda indica la porzione della lista che ogni processo ottiene. Per la distribuzione è impiegata la variante di `MPI_Scatter` per distribuire ad ogni processo un numero diverso di elementi.

5.0.4 Lettura delle performance

```
void start_performance(double * start_time);
void read_performance(double start_time_proc, int memum);
```

Le funzioni determinano il tempo massimo impiegato per svolgere le operazioni di calcolo richieste.

Con `start_performance` i processi sono sincronizzati sulla stessa istruzione con la funzione della libreria `MPI_Barrier`, per far sì che tutti i processi del communicator siano pronti per eseguire le operazioni che seguono.

Con `read_performance` è impiegata la libreria con `MPI_Reduce` che identifica il massimo dei tempi ottenuti da ciascun processo.

5.0.5 Strategie

Strategia I

```
void first_strategy(int memum, double * local_sum) {
  if (memum == 0) {
    double sum_proc = 0;
    int memum_proc = 0;
    for (memum_proc = 1; memum_proc < num_procs; memum_proc++)
    {
      MPI_Recv( &sum_proc, 1, MPI_DOUBLE, memum_proc,
        TAG_STRATEGY(memum_proc), MPI_COMM_WORLD,
        MPI_STATUS_IGNORE);

      *local_sum += sum_proc;
    }
  } else {
    MPI_Send(local_sum, 1, MPI_DOUBLE, 0,
        TAG_STRATEGY(memum), MPI_COMM_WORLD);
  }
}
```

La prima strategia fa sì che ogni processo p_i, con $1 \leq id \leq (P-1)$ e con P numero di processi, invii la propria somma parziale al processo p_0.

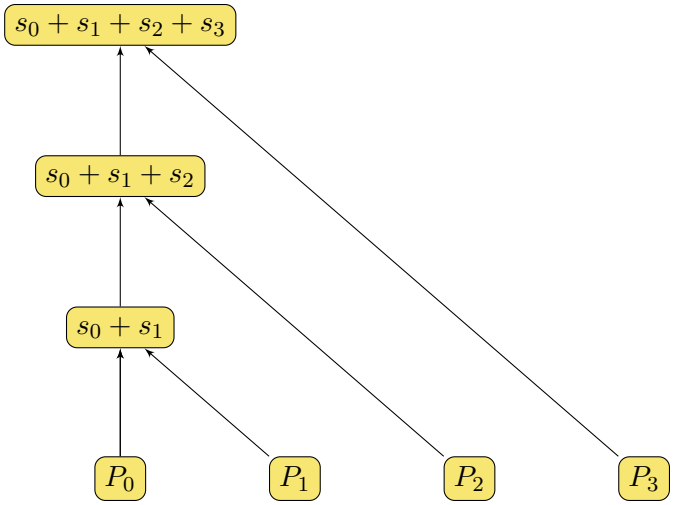

Figura 5.1: Esempio di strategia I con quattro processi

Strategia II

```
void second_strategy(int memum, double * partial_sum) {
  int * exp2;
  exponentials( & exp2);
  int steps = log2(num_procs);
  double tmp_buff;
  int step = 0;

  for (step = 0; step < steps; step++) {
    if ((memum % exp2[step]) == 0) {
```

```
10        if ((memum % exp2[step + 1]) == 0) {
11          MPI_Recv( & tmp_buff, 1, MPI_DOUBLE,
12              (memum + exp2[step]), TAG_STRATEGY(step),
13               MPI_COMM_WORLD, MPI_STATUS_IGNORE);
14          *partial_sum += tmp_buff;
15        } else {
16          MPI_Send(partial_sum, 1, MPI_DOUBLE,
17              (memum - exp2[step]), TAG_STRATEGY(step),
    MPI_COMM_WORLD);
18          }
19       }
20    }
21    if (exp2 != NULL) {
22      free(exp2);
23    }
24  }
```

Per la seconda strategia è generata una lista di valori che corrispondenti alle potenze di due e riusata da tutti i processi. Dopodichè sono calcolati di volta in volta i processi che ricevono e che inaviano la propria somma parziale. Il controllo `memum % exp2[step]) == 0` fa sì che ogni livello, il numero di processi coinvolti nella somma parziale siano dimezzati rispetto al livello precedente.

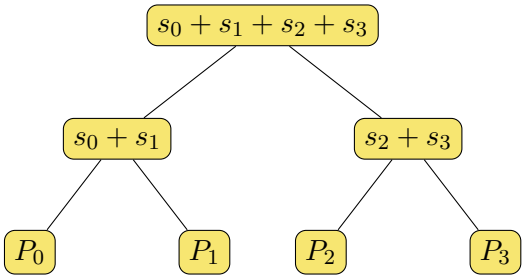

Figura 5.2: Esempio di strategia II con quattro processi

Strategia III

```
1  void third_strategy(int memum, double * partial_sum) {
2    int steps = 0;
3    int * exp2;
4    exponentials( & exp2);
5    steps = log2(num_procs);
6    int another_rank = 0;
7    int level_multiple = 0;
8    int level = 0;
9
10   for (level = 0; level < steps; level++) {
11     level_multiple = exp2[level];
12     if ((memum % (exp2[level + 1])) < level_multiple) {
13       another_rank = (memum + level_multiple);
```

```
14        MPI_Send(partial_sum, 1, MPI_DOUBLE, another_rank,
15          TAG_STRATEGY(level), MPI_COMM_WORLD);
16
17        double buff = 0;
18
19        MPI_Recv( & buff, 1, MPI_DOUBLE, another_rank
20          TAG_STRATEGY(level),MPI_COMM_WORLD
21                            MPI_STATUS_IGNORE);
22        *partial_sum = *partial_sum + buff;
23      } else {
24        another_rank = (memum - level_multiple);
25        MPI_Send(partial_sum, 1, MPI_DOUBLE, another_rank,
26          TAG_STRATEGY(level), MPI_COMM_WORLD);
27
28        double buff = 0;
29
30        MPI_Recv( & buff, 1, MPI_DOUBLE, another_rank,
31         TAG_STRATEGY(level), MPI_COMM_WORLD, MPI_STATUS_IGNORE);
32        *partial_sum = * partial_sum + buff;
33      }
34    }
35    if (exp2 != NULL) {
36      free(exp2);
37    }
38  }
```

La terza strategia si avvale anch'essa di una lista di potenze di due per determinare le comunicazioni da compiere. In tale strategia sono coinvolti tutti i processi in qualsiasi livello, garantendo comunque un numero di livelli minimo per l'esecuzione (vedi 2.0.3).

Ogni processo invia e riceve la somma parziale memorizzata e il processo con il quale comunicare è calcolato in base al proprio identificativo.

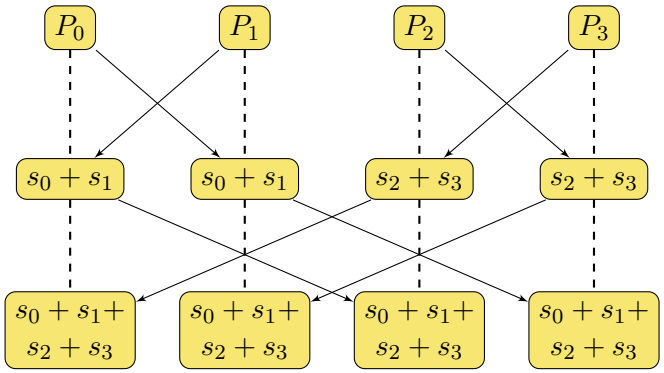

Figura 5.3: Esempio di strategia III con quattro processi

Analisi dei tempi

Esempi d'uso

I parametri forniti in input determinano il comportamento dell'algoritmo; modificando di conseguenza i tempi di esecuzione.

Ogni processo stampa il risultato

La variabile che indica il processo che stampa il risultato è *id* : se assume il valore di -1, tutti i processi stampano la propria somma parziale.

```
mpiexec -np 4 ./primo_elaborato 10000 3 -1

Somma totale : 506429.100210, da id = 0
Somma totale : 506429.100210, da id = 1
Somma totale : 506429.100210, da id = 2
Somma totale : 506429.100210, da id = 3
```

Al programma sono resi disponibili quattro processi con : mille numeri da generare casualmente, da ripartire nei quattro processi e indicando la terza strategia.

```
mpiexec -np 3 ./primo_elaborato 10000 3 -1

Strategia impostata sul valore di 1 poiché il numero
di processi non è potenza di 2...
Se è selezionata la prima o la seconda strategia allora,
solo il primo processo (id = 0) ha il risultato totale!
Per tale motivo id è posto a 0...

Somma totale : 512058.290683, da id = 0
```

Come specificato al capitolo 4, la terza e seconda strategia non sono applicabili su un numero di processi non potenza di due; inoltre per la prima strategia solo il primo processo può stampare il risultato.

Lettura numeri reali in input

```
mpiexec -np 2 ./primo_elaborato 10 3 1 1 2 3 4 5 6 7 8 9 10

Somma totale : 55.000000, da id = 1
```

La lista di numeri reali è letta in input solo se il numero indicato come primo parametro segue : $2 \leq N \leq 20$.

```
mpiexec -np 2 ./primo_elaborato 10 3 1 1 2 3 4 5 6 7 8 9

Il numero totale di elementi da sommare non corrisponde
agli elementi effettivamente passati in input
```

Se il numero degli elementi in lista non corrisponde al numero di elementi indicato come primo parametro, l'esecuzione termine con un errore.

Appendice