

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

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	17
7	Esempi d'uso	19
8	Appendice	21

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 non condivisa, 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

In particolare, 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 fino ad 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'agoritmo compie.

```
1 somma() {  
2     exit_status = check_input(...);  
3  
4     if (exit_status != 0) {  
5         exit(exit_status);  
6     } else if (num_processi == 1) {  
7         sequenziale(...);  
8     }  
9     numero_elementi_processo = calculate_elem_proc(...);  
10    parse_input(...);  
11    buffer = local_calculation(...);  
12    communication_strategy(...);  
13    print_result(...);  
14 }
```

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
- al variare del numero di processi con numero di elementi fisso, l'altezza dell'albero rimane costante poichè la componente T_c parallelizzabile racchiude solamente il livello $l = h$

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 parziale $s_0 + s_1$. L'albero che si ottiene è un albero bilanciato quindi con altezza h minima ottenibile (\log_n).

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.

Input e Output

Indicatori di errore

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

Come indicato alla sezione 1.0.2 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, \dots, 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: $N \neq M \text{ AND } N < 21$	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
$P < 1 \text{ OR } P > 3$	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: $S \neq S1 \text{ AND } IS_POW(P)$	Se la strategia è la seconda o la terza, il numero di processi deve essere potenza di due

Tabella 4.2: Indicatori di errore per strategia

Errori identificativo

Verifica	Descrizione
$ID < -1 \text{ OR } ID > (P-1)$	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: $S \neq S3 \text{ AND } (ID \neq 0 \text{ OR } ID = -1)$	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

```
1 void check_input(int memum,  
2     int * exit_status,  
3     int argc,  
4     int * strategy,  
5     char ** argv,  
6     int * num_items_input,  
7     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

```
1 void parse_input(char ** argv,  
2     int memum,  
3     int num_data_proc,  
4     int num_total_items,  
5     double ** recv_buffer);
```

La lista dei numeri reali è formarsi in due modi : i numeri possono esser forniti dall'utente oppure 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 piuttosto 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

```
1 void distribuite_data(int memum ,
2     double * send_buffer ,
3     int num_data_proc ,
4     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

```
1 void start_performance(double * start_time);
2 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

```

1 void first_strategy(int memum, double * local_sum) {
2     if (memum == 0) {
3         double sum_proc = 0;
4         int memum_proc = 0;
5         for (memum_proc = 1; memum_proc < num_procs; memum_proc++)
6         {
7             MPI_Recv( &sum_proc, 1, MPI_DOUBLE, memum_proc,
8                 TAG_STRATEGY(memum_proc), MPI_COMM_WORLD,
9                 MPI_STATUS_IGNORE);
10
11             *local_sum += sum_proc;
12         }
13     } else {
14         MPI_Send(local_sum, 1, MPI_DOUBLE, 0,
15             TAG_STRATEGY(memum), MPI_COMM_WORLD);
16     }
17 }

```

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

Strategia II

```

1 void second_strategy(int memum, double * partial_sum) {
2     int * exp2;
3     exponentials( & exp2);
4     int steps = log2(num_procs);
5     double tmp_buff;
6     int step = 0;
7
8     for (step = 0; step < steps; step++) {
9         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),
18                     MPI_COMM_WORLD);
19             }
20         }
21     }
22     if (exp2 != NULL) {
23         free(exp2);
24     }
25 }

```

Per la seconda strategia è prima generate una lista di valori che corrisponde alle potenze di due, riusata da tutti i processi. Dopodichè sono calcolati di volta in volta i processi che ricevono e che mandano 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.

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 basa anch'essa su una lista di potenze di due impiegata dai processi 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 manda e riceve la somma parziale memorizzata a discriminando, in base al proprio *id*, il processo con il quale comunicare.

Analisi dei tempi

Esempi d'uso

Appendice