UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

Immagine che contiene nero, oscurità

Descrizione generata automaticamente

SCUOLA POLITECNICA E DELLE SCIENZE DI BASE

DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELLE

TECNOLOGIE DELL’INFORMAZIONE

CORSO DI LAUREA IN INFORMATICA

*CALCOLO DELLA SOMMA DI N NUMERI REALI, IN AMBIENTE DI CALCOLO PARALLELO SU*

*ARCHITETTURA MIMD A MEMORIA DISTRIBUITA, UTILIZZANDO LA LIBRERIA MPI*

**Professori Studenti**

Prof. ssa Valeria Mele Francesco Junior Iaccarino – Matrc.

Prof. Giuliano Laccetti Fabiola Salomone – N86002870.

Anno Accademico - 2023-2024

INDICE

1. Introduzione al problema 5

2. Descrizione dell’algoritmo 6

2.1 Descrizione dell’algoritmo 6

2.2 Strategie e confronti 7

3 Input, Output ed errori 8

3.1 Input e restrizioni 8

3.2 Restrizioni 9

3.3 Output 9

4 Descrizione delle routine 11

4.1 Routine della libreria MPI 11

4.1.2 Routine – MPI\_Send 11

4.1.3 Routine – MPI\_Recv 11

4.1.4 Routine – MPI\_Comm\_rank 12

4.1.5 Routine – MPI\_Comm\_size 12

4.1.6 Routine – MPI\_Reduce 13

4.1.7 Routine – MPI\_Init 13

4.1.8 Routine – MPI\_Finalize 13

4.1.9 Routine – MPI\_Wtime 14

4.1.10 Routine – MPI\_Abort 14

4.1.11 Routine – fmod 14

4.1.12 Routine – pow 14

4.1.13 Routine - malloc 15

4.1.14 Routine – atof 15

4.1.15 Routine – srand 15

4.1.16 Routine – rand 16

4.1.17 Routine – atoi 16

4.2 Subroutine personali 17

4.2.2 Subroutine – check\_pointer 17

5 Descrizione dei test 18

5.1 Scirpt PBS utilizzato per l’esecuzione dell’algoritmo 18

5.2 Esempi d’uso 19

5.2.2 Esempi 1 19

5.2.3 Esempi 2 19

5.2.4 Esempi 3 20

5.2.5 Esempi 4 21

5.2.6 Esempi 5 21

6 Analisi delle performance 23

6.1 Tempo di esecuzione utilizzando un numero p>1 processori 23

6.2 Speed – Up 24

6.3 Efficienza 24

6.4 Analisi preliminare dei tempi 25

6.4.1 Analisi preliminare strategia 1 25

6.4.2 Analisi preliminare strategia 2 e 3 25

6.4 Analisi tempo reale algoritmo 26

6.4.1 Strategia 1 26

6.4.2 Strategia 2 28

6.4.3 Strategia 3 28

7 Codice Sorgente 29

# Introduzione al problema

Si vuole realizzare un algoritmo per il calcolo della somma di N numeri reali, in ambiente di calcolo parallelo su architettura MIMD a memoria distribuita, utilizzando la libreria MPI.

L’algoritmo deve implementare le tre strategie di comunicazione relative alla somma e prendere in input le seguenti informazioni :

1. **Numero di valori da sommare (N):** Questo valore rappresenta quanti numeri reali devono essere sommati. L'utente deve fornire questo numero come input;
2. **Valori da sommare:** Nel caso in cui il numero di valori da sommare (N) sia minore o uguale a 20, l'utente deve fornire direttamente i valori (numeri reali) da sommare come parte dell'input. In caso contrario, se N è maggiore di 20, l'algoritmo deve generare casualmente i valori reali da sommare;
3. **Numero di strategia da utilizzare (I, II, III):** L'utente deve specificare quale delle tre strategie di comunicazione (I, II o III) desidera utilizzare per calcolare la somma dei valori;

Inoltre, l'algoritmo deve gestire i casi in cui l'utente scelga le strategie II o III ma non siano applicabili (ad esempio, a causa del numero di processori disponibili). In questi casi, l'algoritmo dovrebbe automaticamente applicare la strategia I.

# Descrizione dell’algoritmo

In questo capitolo, viene descritto l’algoritmo suddividendolo in quattro fasi e dettagliata ognuna di essa e descritta la terza strategia mettendola a confronto con le altre due .

## 2.1 Descrizione dell’algoritmo

L'algoritmo è strutturato in quattro fasi distinte:

1. **Fase 1**: Distribuzione dell'input tra i vari processi;
2. **Fase 2**: Calcolo delle somme parziali;
3. **Fase 3**: Strategie di comunicazione;
4. **Fase 4**: Stampa del risultato.

Nella prima fase il processo con rank 0 prende in input il valore N, la strategia

da utilizzare, l’id del processo che dovrà stampare il risultato e gli eventuali

valori da sommare e gli invii agli altri. Gli altri processi riceveranno solo una

parte divisa equamente di valori da sommare (N fratto numero di processi

ciascun processo, con al massimo una differenza di 1 elemento tra i var

processi).

La seconda fase sussiste in una normale somma locale di valori, eseguita da ciascun processo.

Nella terza fase, in base alla strategia di comunicazione e l’id del processo che dovrà stampare poi il risultato, verrà completata la somma.

Nella quarta fase verrà stampato il risultato così come richiesto dall’input. Nel caso in cui debbano stampare tutti i processi, per convenzione il processo

ricevente nella strategia 1 e 2 sarà rank 0, per poi inviare in broadcast, prima

della stampa, la somma a tutti gli altri processi.

## 2.2 Strategie e confronti

In questa sezione, analizzeremo in dettaglio la terza strategia di comunicazione utilizzata nel programma. Questa stategia mira a garantire che, al termine della computazione tutti i processi abbiano accesso alla somma totale dei valori passati in input. Tuttavia ciò comporta un aumento significcativo del numero di comunicazioni tra i processi.

In questa strategia, i valori in input vengono suddivisi equamente tra i vari processi, ciascuno dei quali calcola la sua somma parziale. Dopo aver completato le sonne parziali, i processi iniziano a comunicare tra loro in coppie distinte.

La modalità usata, per effettuare questa comunicazione, è la seguente : entrambi i processi comunicano le rispettive somme parziali e aggiungono il valore ricevuto alla propria somma.

Durante ciascuna interazione di comunicazione, ogni processo con un determinato rank comunicherà con un altro processo che ha un rank di i+(2^passo) oppure i-(2^passo), a seconda del risultato della divisione tra il suo rank e 2^(i+1).

È importante notare che questa strategua comporta un certo numero di comunicazioni ridondanti, ma questa comunicazioni avvengono in parallelo durante ogni interazione. Questa strategia è preferibile quando è essenzile che tutti i processi abbiano accesso alla somma totale dei valori

A differenza della strategia 2, che coinvolge comunicazioni a coppie di processi senza comunicazione bidirezionali, la stategia 3 coinvolge comunicazioni bidirezionali tra i processi. Al fine della stategia2, solo uno dei processi avrà la somma totale, mentre gli altri avranno somma parziali. È possibile selezionare quale processo otterrà il risultato finale ruotando l’albero delle comunicazioni dei processi,

La strategia 1, d’altra parte, prevede che ciascun processo calcoli la sua somma parziale e la comunichi al processo P0, che successivamente somma tutte le somme parziali ricevute. Anche in questo caso è possibile selezionare il processo ricevente. Tuttavia questa strtegia comporta un aumento esponensiale del numero di comunicazioni sequenziali tra i processi e il processo ricevente all’aumentare del numero dei processi.

Ad esempio, con 16 processi, il processo ricevente dovrà ricevere 15 somme parziali in modo sequenziale, mentre con 128 processi dovrà gestire 127 somme parziali e cosi via.

# Input, Output ed errori

## 3.1 Input e restrizioni

Come descritto nel capitolo relativo all’introduzione, i parametri in input da passare

sono i seguenti:

* **Numero di valori da sommare (N):** Questo valore rappresenta quanti numeri reali devono essere sommati. L'utente deve fornire questo numero come input;
* **Valori da sommare:** Nel caso in cui il numero di valori da sommare (N) sia minore o uguale a 20, l'utente deve fornire direttamente i valori (numeri reali) da sommare come parte dell'input. In caso contrario, se N è maggiore di 20, l'algoritmo deve generare casualmente i valori reali da sommare;
* **Numero di strategia da utilizzare (I, II, III):** L'utente deve specificare quale delle tre strategie di comunicazione (I, II o III) desidera utilizzare per calcolare la somma dei valori;

Inoltre, abbiamo scelto di includere l’opzione di specificare l’ID del processo

responsabile della stampa del risultato. Tenendo conto che l’utente possa desiderare

gli stampare il risultato di tutti i processi e in tal caso è sufficiente impostare l’ID su

-1.

I parametri devono essere passati tutti come argomento all’interno del file PBS, alla

riga corrispondente l’esecuzione del programma, nel seguente ordine:

1. ID processo stampa;
2. Strategia da utilizzare;
3. Numero di valori da sommare N;
4. Eventuali valori da sommare se N<=20;

Per comprendere meglio dove inserire tali parametri, la riga del file PBS è la

seguente:

*/usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile $PBS\_NODEFILE -n*

*$NCPU $PBS\_O\_WORKDIR/out <ID> <Strategia> <N> <Val1>…<Valn>*

## 3.2 Restrizioni

I parametri devono rispettare determinate restizioni/costrizioni:

* L’ID del processo deve rientrare nell’intervallo compreso tra -1 e il rank del processo più alto. Ad esempio, se ci sono 8 processi, l’ID deve essere compreso tra -1 e 7.
* La strategia deve essere compresa fra 1 e 3. Come descritto in precedenza, nel caso in cui il l’utente indica la 2 o la 3 stategia ma esse non sono applicabili il sistema applica autonomamente la 1 stategia.
* Il numero di valori da sommare passati deve combaciare con il numero N indicato. Nel caso N fosse > 20 e vengono indicati valori da sommare il programma andrà in errore. Nel caso in cui N<=20 e il numero di valori da sommare è diverso da N il programma andrà in errore.

Vengono inoltre eseguiti controlli su tutti i valori passati come parametri, e in caso di qualsiasi violazione di queste restrizioni, il programma genererà un messaggio a video indicando l’errore corrispondente.

## 3.3 Output

Vediamo ora alcuni esempi di output che possiamo aspettarci in caso di determinati input di esempio. Esempi mostrati con due processori. Ordine input come indicato in precedenza: <ID>, <Strategia>, <N>, <Val1>,…,<Valn>

* INPUT: 0 1 6 2 3 2 3 3 4
* OUTPUT: “Somma calcolata da 0 è: 17.000000”
* INPUT: -1 1 6 8 3 3 5 5 4
* OUTPUT: “Somma calcolata da 1 è: 28.000000” “Somma calcolata da 0 è: 28. 000000”
* INPUT: -1 1 6 8 3 2 7 7
* OUTPUT: “ERRORE. VALORE N INDICATO NON CORRISPONDENTE AI PARAMETRI PASSATI IN INPUT”
* INPUT: -1 3 21
* OUTPUT: “Somma calcolata da 0 è: 98.400000” “Somma calcolata da 1 è: 98.400000” (L’output potrebbe variare in quanto con 21 numeri i valori da sommare sono generati casualmente)
* INPUT: 1 1 6 1 2 3 4 5 6
* OUTPUT: “Somma calcolata da 1 è: 21.000000”

Tutti gli output verranno stampati all’interno del file “*output.out*” mentre eventuali errori relativi a compilazione o esecuzione verranno stampati nel file “*error.err*” (non sono incusi i messaggi di violazione delle restrizioni degli input che sono invece presenti nel file “*output.out*”).

# Descrizione delle routine

## Routine della libreria MPI

Il software descritto, si avvale dell’ausilio di alcune routine della libreria MPI per il calcolo parallelo.

Di seguito, per ognuna di esse si fornisce il prototipo utilizzato nel programma

e la descrizione dei relativi parametri di input e di output.

### Routine – MPI\_Send

La funzione MPI\_send viene utilizzata per inviare dati da un processo a un altro processo all’interno dell’ambiente MPI.

Si occupa di bloccare il mittente fino a quando il messaggio è stato accettato dal processo destinatatio. Il processo destinatario può ricevere il messaggio utilizzando la funzione MPI\_Recv. La sua firma è :

**int MPI\_Send(const void \*buf, int count, MPI\_Datatype datatype, int dest, int tag, MPI\_Comm comm)**

In cui :

* buf : indirizzo iniziale del buffer di invio;
* count : numero di elementi da inviare dal buffer;
* datatype : tipo di ogni dato del buffer da inviare ;
* dest : tag del messaggio;
* comm : Communicator che specifica il gruppo di processi coinvolti nella comunicazione.

### Routine – MPI\_Recv

La funzione MPI\_Recv viene utilizzata per ricevere dati da un processo mittente a all’interno dell’ambiente MPI.

Si occupa il processo ricevente fino a quando il messaggio è stato ricevuto dal processo mittente. La sua firma è :

**int MPI\_Recv(void \*buf, int count, MPI\_Datatype datatype, int source, int tag, MPI\_Comm comm, MPI\_Status \*status)**

In cui :

* count : massimo numero di elementi da ricevere nel buffer;
* datatype : tipo di ogni dato da ricevere nel buffer
* source : rank della sorgente
* tag : target del messaggio
* comm : communicator
* buf : indirizzo iniziale del buffer di ricezione (parametro di output)
* status : oggetto status (parametro di output)

### Routine – MPI\_Comm\_rank

La funzione MPI\_Comm\_rank viene utilizzata per ottenere il proprio rank o indentificatore all’interno di un comunicatore specifico. Il “rank” è un numero univoco assegnato a ciascun processo all’interno del comunicatore, che serve a identificare univocamente ciascun processo all’interno del gruppo. La firma è :

**int MPI\_Comm\_rank(MPI\_Comm comm, int \*rank)**

In cui :

* comm : Communicator
* rank : Rank del processo chimante nel gruppo comm (parametro di output)

### Routine – MPI\_Comm\_size

La funzione MPI\_Comm\_size viene utilizzata per ottenere il numero totale di processi all’interno di un comunicatore specifico. Questo comunicatore rappresenta un gruppo di processi che possono comunicare tra loro. La firma è:

**int MPI\_Comm\_size(MPI\_Comm comm, int \*size)**

In cui:

* comm : Communicator
* size : numero di processi nel gruppo comm (parametri di output)

### Routine – MPI\_Reduce

La funzione MPI\_Reduce si occupa di ridurre i dati distribuiti tra processi, tale funzione nello specifico aggrega i dati da tutti i processi coinvolti in una singola operazione e restituisce un risultato al processo specificato come destinazione della riduzione. La firma è:

**int MPI\_Reduce(const void \*sendbuf, void \*recvbuf, int count, MPI\_Datatype datatype, MPI\_Op op, int root, MPI\_Comm comm)**

In cui :

* sendbuf: Indirizzo del buffer di invio
* count: Numero di elementi del buffer da inviare
* datatype: Tipo di dato degli elementi nel buffer
* op: Operazione di riduzione
* root: Rank del processo che deve ricevere il dato
* comm: Communicator
* recvbuf: Indirizzo del buffer di ricezione (parametro di output)

### Routine – MPI\_Init

La funzione MPI\_Init consente a processi separati di comunicare tra loro in un ambiente di calcolo distribuito o parallelo. In particolare tale funzione è utilizzata per inizializzare l’ambiente MPI prima di iniziare a utilizzare altre funzioni MPI. La firma è:

**int MPI\_Init(int \*argc, char \*\*\*argv)**

In cui:

* argc : puntatore al numero di argomenti
* argv : puntatore al vettore degli argomenti

### Routine – MPI\_Finalize

La funzione MPI\_Finalize consente a processi separati di comunicare tra loro in un ambiente di calcolo distribuito o parallelo. È importante chiamare la funzione alla fine di ogni programma MPI per garantire che l’ambiente MPI sia pulito prima dell’uscita. La firma è :

**int MPI\_Finalize( void )**

### Routine – MPI\_Wtime

La funzione MPI\_Wtime è utilizzata per calcolare il tempo trascorso tra due èunti del codice e valutare le prestazioni di un sistema parallelo. In particolare questa funzione restituisce il tempo trascorso in secondi dalla “mezzanotte” fino al momento della chiamata. La firma è:

**double MPI\_Wtime(void)**

### Routine – MPI\_Abort

La funzione MPI\_Abort viene utilizzata per interrompere l’esecuzione del programma stesso. Essa è utile in situazioni in cui si verifica un errore grave o una condizione eccezionale che richede l’interruzione immediata del sistema parallelo. La firma è:

**int MPI\_Abort(MPI\_Comm comm, int errorcode)**

In cui:

* Comm : Communicator del task da abortire
* Errorcode : codice di errore

### Routine – fmod

La funzione MPI\_fmod viene utilizzata per calcolare il resto della divisione tra due numeri in virgola mobile (“double”). La firma è:

**double fmod(double x, double y)**

In cui :

* x : Valore floating point al numeratore della divisione
* y : Valore floating point al denominatore della divisione

### Routine – pow

Questa funione restituisce la base elevata a potenza dell'esponente. La firma è

**double pow (double base, double exponent)**

In cui:

* Base: valore della base
* Exponent: valore dell’esponente

### Routine - malloc

La funzione alloca un blocco di memoria, restituendo un puntatore all'inizio del blocco. Il contenuto del blocco di memoria appena allocato non viene inizializzato e rimane con valori indeterminati. In caso di successo, viene restituito un puntatore al blocco di memoria allocato dalla funzione.

Il tipo di questo puntatore è sempre void\*, che attraverso il casting può diventare del tipo desiderato per essere dereferenziabile.

Se la funzione non è riuscita ad allocare il blocco di memoria richiesto, viene restituito un puntatore nullo. La firma è:

**void\* malloc (size\_t size)**

In cui:

* size: Dimensione del blocco di memoria, in byte.

### Routine – atof

Analizza la stringa C str, interpretando il suo contenuto come un numero in virgola mobile e restituendo il suo valore come un doppio.

La funzione scarta innanzitutto tutti i caratteri di spazio bianco (come in isspace) necessari fino a trovare il primo carattere non di spazio bianco. Poi, a partire da questo carattere, prende il maggior numero possibile di caratteri validi, seguendo una sintassi simile a quella dei letterali in virgola, e li interpreta come un valore numerico. Il resto della stringa dopo l'ultimo carattere valido viene ignorato e non ha alcun effetto sul comportamento di questa funzione. La firma è:

**double atof (const char\* str)**

In cui:

* str: stringa C che inizia con la rappresentazione di un numero in virgola mobile.

### Routine – srand

Il generatore di numeri pseudocasuali viene inizializzato utilizzando l'argomento passato come seed.

Per ogni diverso valore di seme utilizzato in una chiamata a srand, si può prevedere che il generatore di numeri pseudocasuali generi una diversa successione di risultati nelle successive chiamate a rand.

Due inizializzazioni diverse con lo stesso seed genereranno la stessa successione di risultati nelle chiamate successive a rand.

Se il seed è impostato a 1, il generatore viene reinizializzato al suo valore iniziale e produce gli stessi valori di prima di qualsiasi chiamata a rand o srand.

Per generare numeri simili a quelli casuali, srand viene solitamente inizializzato a un valore caratteristico del tempo di esecuzione, come il valore restituito dalla funzione time (dichiarato nell'intestazione <ctime>). Questo valore è abbastanza distintivo per la maggior parte delle esigenze di randomizzazione banali. La firma è:

**void srand (unsigned int seed)**

In cui:

* seed: un valore intero da utilizzare come seme per l'algoritmo del generatore di numeri pseudocasuali

### Routine – rand

Restituisce un numero intero pseudocasuale nell'intervallo compreso tra 0 RAND\_MAX (RAND\_MAX è una costante definita in <cstdlib>.).

Questo numero è generato da un algoritmo che restituisce una sequenza di numeri apparentemente non correlati ogni volta che viene chiamato. Questo algoritmo utilizza un seed per generare la serie, che deve essere inizializzato a un valore distintivo utilizzando la funzione srand. La firma è:

**int rand (void)**

### Routine – atoi

Analizza la stringa C interpretando il suo contenuto come un numero intero, che viene restituito come valore di tipo int.

La funzione scarta innanzitutto tutti i caratteri di spazio bianco (come in isspace) necessari fino a trovare il primo carattere non di spazio bianco. Quindi, a partire da questo carattere, prende un segno iniziale opzionale più o meno seguito dal maggior numero possibile di cifre in base 10 e le interpreta come un valore numerico. La firma è:

**int atoi (const char \* str)**

In cui:

* str: stringa C che inizia con la rappresentazione di un numero integrale.

## Subroutine personali

### Subroutine – check\_pointer

La routine verifica che gli argomenti passati nel vettore “*argv*” rispettino tutte le restizioni imposte nel capitolo 2. Nel caso in cui venga individuata una violazione di una qualsiasi delle restrizioni, la procedura stamperà un messaggio appropriato su standard output e chiamera la funzione “MPI\_Abort(comm, err)” interrompendo l’esecuzione del programma. La firma è :

**void check\_input(int argc, char\* argv[], int nproc)**

In cui :

* argc : per il numero di argomenti
* argv : vettore degli argomenti
* nproc : numero di processi totale

# Descrizione dei test

## Scirpt PBS utilizzato per l’esecuzione dell’algoritmo

1. #!/bin/bash

2.

3. #PBS -q studenti

4. #PBS -l nodes=8

5. #PBS -N elaborato1

6. #PBS -o output.out

7. #PBS -e error.err

8.

9. NCPU=`wc -l < $PBS\_NODEFILE`

10. echo ------------------------------------------------------

11. echo ' This job is allocated on '${NCPU}' cpu(s)'

12. echo 'Job is running on node(s): '

13. cat $PBS\_NODEFILE

14. PBS\_O\_WORKDIR=$PBS\_O\_HOME/Somma

15. echo ------------------------------------------------------

16. echo PBS: qsub is running on $PBS\_O\_HOST

17. echo PBS: originating queue is $PBS\_O\_QUEUE

18. echo PBS: executing queue is $PBS\_QUEUE

19. echo PBS: working directory is $PBS\_O\_WORKDIR

20. echo PBS: execution mode is $PBS\_ENVIRONMENT

21. echo PBS: job identifier is $PBS\_JOBID

22. echo PBS: job name is $PBS\_JOBNAME

23. echo PBS: node file is $PBS\_NODEFILE

24. echo PBS: current home directory is $PBS\_O\_HOME

25. echo PBS: PATH = $PBS\_O\_PATH

26. echo ------------------------------------------------------

27. echo "Eseguo/usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS\_O\_WORKDIR/out $PBS\_O\_WORKDIR/elaborato1.c"

28. /usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS\_O\_WORKDIR/out $PBS\_O\_WORKDIR/elaborato1.c

29. echo "Eseguo:/usr/lib64/openmpi/1.4-gcc/bin/mpiexec -np 8 -machinefile $PBS\_NODEFILE $PBS\_O\_WORKDIR/out"

30. /usr/lib64/openmpi/1.4-gcc/bin/mpiexec -np 8 -machinefile $PBS\_NODEFILE $PBS\_O\_WORKDIR/out 0 1 1000

Questo file ci permette di compilare ed eseguire il programma C denominato ”elaborato1.c” che si occupa di calcolare la sonna di n numeri (reali) facendo uso di 1°, 2° e 3° strategia su un cluster di 8 nodi con un totale di 4 processi. Le uscite dell’esecuzione vengono reindirizzate ai file “output.out” ed “error.out”.

Descriviamo brevemente i vari parametri presenti nel file pbs :

1. Definizione delle direttive PBS :

* ‘ #PBS -q studenti ‘ : specifica la coda su cui verrà eseguito il job
* ‘ #PBS -l nodes=8 ‘ : richiede l’allocazione di 8 nodi
* ‘ #PBS -N elaborato1 ‘ : assegna un nome al job, in questo caso “elaborato1”.
* ‘ #PBS -o output.out ‘ e ‘ #PBS -e error.err ‘ : specifica i file in cui verranno indirizzati i messaggi di output e gli errori

1. Calcolo del numero di CPU

* ‘ NCPU = ‘wc -l < $PBS\_NODEFILE’’: utilizza il file ‘$PBS\_NODEFILE’ per determinare il nomero di CPU assegnate al job

1. Visualizzazione delle infomazioni del job :

* Il blocco seguente stampa varie informazioni sul job, tra cui il numero di CPU, i nodi assegnati e altre variabili di ambiente rilevanti

1. Impostazione della directory di lavoro

* ‘PBS\_O\_WORKDIR=$PBS\_O\_HOME/Somma’ : imposta la directory di lavoro del job come ‘=$PBS\_O\_HOME/Somma’

1. Compilazione del codice C

* ‘echo "Eseguo /usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS\_O\_WORKDIR/out$PBS\_O\_WORKDIR/elaborato1.c" : stampa un messaggio relativo alla compilazione del file C ‘elaborato1.c’ utilizzando il compilatore ‘mpicc’, e il risultato verrà salvato in ‘$PBS\_O\_WORKDIR/out’

1. Esecuzione del programma

* ‘usr/lib64/openmpi/1.4-gcc/bin/mpiexec -np 8 -machinefile $PBS\_NODEFILE $PBS\_O\_WORKDIR/out 0 1 1000’ : stampa un messaggio relativo all’esecuzione del programma compilato con ‘mpiexece’. Il programma accetta alcuni argomenti come ‘ 0 1 1000’

Il primo passo per eseguire e compilare il programma “elaborato1.c” è eseguire il file pbs, ciò viene fatto accedendo al cluster, posizionandoci nella cartella contenente tale file e eseguendo sul terminale il comando “*qsub elaborato1.pbs*”. Alla sua esecuzione si generano diversi file, in particolare:

1. L’secutore relativo al programma scritto in linguaggio C;
2. Un file “output.out” contenente l’output del programma. Questo output è generato dai valori dati in input nell’ultima riga del file “Elaborato1.pbs” e include informaizoni come il numero di processori paralleli utilizzati, il file eseguibile, l’ID del processo (dove -1 richiama tutti i processi), la strategia utilizzata e in fine il numero di valori da sommare;
3. Un file “error.err” contenente eventuali errori del programma. In caso di esecuzione senza errore, questo file rimarrà vuoto, come in questo caso.

## Esempi d’uso

Vengono riportati alcuni esempi d’uso, partendo dall’esecuzione del file PBS fino a mostrare l’output del programma.

### Esempi 1

Tenendo conto del file “Elaborato1.pbs” , definito nel paragrago 5.1, abbaimo un primo esempio d’output del programma, dove diamo in input :

1. Id : 0
2. Strategia : 1
3. N (numeri da sommare) : 100

L’output del programma è il seguente :

Immagine che contiene testo, Carattere, schermata

Descrizione generata automaticamente

Figura 1 - Output esempio 1

Dove viene visualizzato anche il tempo calcolato in quanto è stato usato l’algoritmo che preleva anche i tempi di esecuzione.

### Esempi 2

Altro esempi, in cui usiamo :

1. Processori : 4
2. Id : -1
3. Strategia : 3
4. N (numeri da sommare) : 3
5. Valori (da sommare) : 2 1 5

Eseguiamo lo stesso file, con *qsub,*  con estensione pbs solo modifichiamo l’ultima riga del codice con:

1. /usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile $PBS\_NODEFILE -n 4 $PBS\_O\_WORKDIR/out -1 3 2 1 5

L’output prodotto in tal caso sarà:

Immagine che contiene testo, Carattere, schermata

Descrizione generata automaticamente

Figura 2 - Output esempio 2

### Esempi 3

Altro esempi, in cui usiamo :

1. Processori : 8
2. Id : -1
3. Strategia : 2
4. N (numeri da sommare) : 1000

Eseguiamo lo stesso file, con *qsub,*  con estensione pbs solo modifichiamo l’ultima riga del codice con:

1. /usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile $PBS\_NODEFILE -n 8 $PBS\_O\_WORKDIR/out -1 2 1000

L’output prodotto in tal caso sarà:

Immagine che contiene testo, Carattere, schermata

Descrizione generata automaticamente

Figura 3 - Output esempio 3

### Esempi 4

Altro esempi, in cui usiamo :

1. Processori : 5
2. Id : -1
3. Strategia : 2
4. N (numeri da sommare) : 1000

Eseguiamo lo stesso file, con *qsub,*  con estensione pbs solo modifichiamo l’ultima riga del codice con:

1. /usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile $PBS\_NODEFILE -n 5 $PBS\_O\_WORKDIR/out -1 2 1000

L’output prodotto in tal caso sarà:

Immagine che contiene testo, Carattere, schermata

Descrizione generata automaticamente

Figura 4 - Output esempio 4

Questo è un caso in cui viene in input indicata la seconda strategia dall’utente ma evidentemente tutti gli i processori sono già occupati e il sistema applica automaticamente la prima stategia per garantire che il programma possa essere eseguito senza interruzioni dovute alla mancanza di risorse.

### Esempi 5

Altro esempi, in cui usiamo :

1. Processori : 5
2. Id : -1
3. Strategia : 3
4. N (numeri da sommare) : 3000

Eseguiamo lo stesso file, con *qsub,*  con estensione pbs solo modifichiamo l’ultima riga del codice con:

1. /usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile $PBS\_NODEFILE -n 5 $PBS\_O\_WORKDIR/out -1 3 3000

L’output prodotto in tal caso sarà:

Immagine che contiene testo, Carattere, schermata

Descrizione generata automaticamente

Figura 5 - Output esempio 5

Questo è un caso in cui viene in input indicata la terza strategia dall’utente ma evidentemente tutti gli i processori sono già occupati e il sistema applica automaticamente la prima stategia per garantire che il programma possa essere eseguito senza interruzioni dovute alla mancanza di risorse.

# Analisi delle performance

In questo capitolo andremo a a valutare le prestazioni del nostro tramite i seguenti parametri : tempo d’esecuzione (T(p)), speed-up (E(p)) ed efficienza (E(p)).

È importante sottolineare che questi sono parametri utilizzati per valutare un software parallelo.

## Tempo di esecuzione utilizzando un numero p>1 processori

Sia :

una funzione che, dato in ingresso la dimensione dell’input n ∈ N, fornisca il tempo di esecuzione (passi temporali) di un algoritmo in ambiente parallelo dotato di p processori.

Dunque, si ha che:

* T(1): tempo di esecuzione utilizzando un solo processore (equivalente

al tempo di esecuzione dell’algoritmo sequenziale);

* T(2): tempo di esecuzione utilizzando 2 processori;
* T(p): tempo di esecuzione utilizzando p processori.

Generalmente, notiamo che, indicheremo tale parametro con il simbolo **Tp**.

Inoltre la formula relativa a tal parametro varia se si considera la prima, la seconda o la terza strategia, infatti :

**Strategia 1**:

* T(p) = (n/p + p – 2) Tcalc

**Strategia 2**:

* T(p) = (n/p – 1 + log2(p)) Tcalc

**Strategia 3**:

* T(p) = (n/p – 1 + log2(p)) Tcalc

## Speed – Up

Lo Speed-UP indica la riduzione del tempo di esecuzione rispetto all’utilizzo di un solo processore, utilizzando invece ***p*** processori.

In simboli:

con

Il valore dello speed-up ideale dovrebbe essere pari al numero p dei processori,

perciò l’algoritmo parallelo risulta migliore quanto più ***Sp*** è prossimo a ***p***.

Ciò vale per la prima, seconda e terza strategia.

## Efficienza

Calcolare solo lo speed-up spesso non basta per effettuare una valutazione

corretta, poiché occorre “rapportare lo speed-up al numero di processori”, e

questo può essere effettuato valutando l’efficienza.

Siano dunque ***p*** il numero di processori ed ***Sp*** lo speed-up ad esso relativi.

Si definisce efficienza il parametro:

Essa fornisce un’indicazione di quanto sia stato usato il parallelismo nel calcolatore.

Idealmente, dovremmo avere che:

e quindi l’algoritmo parallelo risulta migliore quanto più Ep è vicina ad 1. Ciò vale per la prima, seconda e terza strategia.

## Analisi preliminare dei tempi

### Analisi preliminare strategia 1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **p** | **n** | **T(p)** | **S(p)** | **E(p)** |
| 1 | 30 | 29 Tcalc | 1,0 | 1,0 |
| 2 | 30 | 15 Tcalc | 1,94 | 0,97 |
| 4 | 30 | 9,5 Tcalc | 3,05 | 0,76 |
| 8 | 30 | 9,75 Tcalc | 2,97 | 0,371 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **p** | **n** | **T(p)** | **S(p)** | **E(p)** |
| 1 | 800.000 | 799999 Tcalc | 1 | 1 |
| 2 | 800.000 | 400000 Tcalc | 1,9999975 | 0,99999875 |
| 4 | 800.000 | 200002 Tcalc | 3,999955 | 0,99998875 |
| 8 | 800.000 | 100006 Tcalc | 7,99951003 | 0,99993875 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **p** | **n** | **T(p)** | **S(p)** | **E(p)** |
| 1 | 80.000.000 | 79999999 Tcalc | 1 | 1 |
| 2 | 80.000.000 | 40000000 Tcalc | 1,99999998 | 0,99999999 |
| 4 | 80.000.000 | 20000002 Tcalc | 3,99999955 | 0,99999989 |
| 8 | 80.000.000 | 10000006 Tcalc | 7,9999951 | 0,99999939 |

### Analisi preliminare strategia 2 e 3

Vengono considerate entrambe le strategie in quanto le formule sono le stesse.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **p** | **n** | **T(p)** | **S(p)** | **E(p)** |
| 1 | 30 | 29 Tcalc | 1,0 | 1,0 |
| 2 | 30 | 14,3 Tcalc | 2,03 | 1,015 |
| 4 | 30 | 7,10 Tcalc | 4,08 | 1,02 |
| 8 | 30 | 3,65 Tcalc | 7,94 | 0,9925 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **p** | **n** | **T(p)** | **S(p)** | **E(p)** |
| 1 | 800.000 | 799999 Tcalc | 1 | 1 |
| 2 | 800.000 | 400000 Tcalc | 1,9999975 | 0,99999875 |
| 4 | 800.000 | 200001 Tcalc | 3,999975 | 0,99999375 |
| 8 | 800.000 | 100002 Tcalc | 7,99983 | 0,99997875 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **p** | **n** | **T(p)** | **S(p)** | **E(p)** |
| 1 | 80.000.000 | 79999999 Tcalc | 1 | 1 |
| 2 | 80.000.000 | 40000000 Tcalc | 1,99999998 | 0,99999999 |
| 4 | 80.000.000 | 20000001 Tcalc | 3,99999975 | 0,99999994 |
| 8 | 80.000.000 | 10000002 Tcalc | 7,9999983 | 0,99999979 |

## Analisi tempo reale algoritmo

Di seguito, mostreremo i gragici dei tempi di esecuzione media al variare del numero di processi (p), considerando solo input di dimensioni significativi (non input piccoli come 100 in quanto in quel caso il tempo delle comunicazioni è molto maggiore rispetto ai tempi delle somme, di conseguenza il tempo calcolato andrebbe a dipendere di molto dal tempo delle comunicazioni. Di conseguenza otterremo che all’aumentare del numero p il tempo aumenterebbe).

I gradici sono basati su dieci misurazioni di tempo e rappresentano la media di tali tempi per ciascun valore di p, dove p può essere 1, 2, 4, o 8.

L’obiettivo è visualizzare come i tempi di esecuzione cambiano al variare del numero di processi utilizzati.

### Strategia 1

Viene analizzato il tempo di esecuzione dell’algoritmo utilizzando la strategia 1 :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **p** | **n** | **T(p)** | **S(p)** | **E(p)** |
| 1 | 800.000 | 0,001722 s | 1 | 1 |
| 2 | 800.000 | 0,000873 s | 1,97250859 | 0,9862543 |
| 4 | 800.000 | 0,000493 s | 3,49290061 | 0,87322515 |
| 8 | 800.000 | 0,000028 s | 61,5 | 7,6875 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **p** | **n** | **T(p)** | **S(p)** | **E(p)** |
| 1 | 80.000.000 | 0,340482 s | 1 | 1 |
| 2 | 80.000.000 | 0,172482 s | 1,97401468 | 0,98700734 |
| 4 | 80.000.000 | 0,107882 s | 3,1560594 | 0,78901485 |
| 8 | 80.000.000 | 0,105219 s | 3,23593648 | 0,40449206 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **p** | **n** | **T(p)** | **S(p)** | **E(p)** |
| 1 | 200.000.000 | 0,853965 s | 1 | 1 |
| 2 | 200.000.000 | 0,445223 s | 1,91806129 | 0,95903064 |
| 4 | 200.000.000 | 0,268358 s | 3,18218574 | 0,79554643 |
| 8 | 200.000.000 | 0,262142 s | 3,2576428 | 0,40720535 |

In tutte e tre le configurazione, l’uso di più processi porta a notevoli miglioramenti delle prestazioni, ma l’efficienza può diminuire quandi si usano un numero molto elevato di processori.

Rappresentiamo i seguenti dati su un piano cartesiano in cui sulll’asse orizzontale rappresentiamo il numero dei processori (1, 2, 4, 8) mentre sull’asse verticale il tempo d’esecuzione dell’algoritmo T(p) :

Dal grafico si evince che con 1 processore, all’aumentare di N otteniamo un numero significativo dei tempi. Man mano che ci avviciniamo a 8 processori, il tempo tende a convergere verso un valore basso. Questo suggerisce che l’uso di più processori può accelerare notevolmente l’esecuzione del programma, specialmente per problemi di dimensione più grandi.

Tuttavia, è importante notare che per input di dimensioni molto piccole (come N=100), i tempi di comunicazione tra i processi possono avere un impatto significativo sui tempi totale, portando a risultati non lineari. Pertanto, per tali input, l’aumento del numero di processori potrebbe non comportare un miglioramento significativo delle prestazioni.

### Strategia 2

Viene analizzato il tempo di esecuzione dell’algoritmo utilizzando la strategia 2 :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **p** | **n** | **T(p)** | **S(p)** | **E(p)** |
| 1 | 800.000 | 0,000022 s | 1 | 1 |
| 2 | 800.000 | 0,000046 s | 1,97250859 | 0,9862543 |
| 4 | 800.000 | 0,000043 s | 3,49290061 | 0,87322515 |
| 8 | 800.000 | 0.000074 s | 61,5 | 7,6875 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **p** | **n** | **T(p)** | **S(p)** | **E(p)** |
| 1 | 80.000.000 | 0,341071 s | 1 | 1 |
| 2 | 80.000.000 | 0,172471 s | 1,97401468 | 0,98700734 |
| 4 | 80.000.000 | 0,108817 s | 3,1560594 | 0,78901485 |
| 8 | 80.000.000 | 0,105319 s | 3,23593648 | 0,40449206 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **p** | **n** | **T(p)** | **S(p)** | **E(p)** |
| 1 | 200.000.000 | 0,851029 s | 1 | 1 |
| 2 | 200.000.000 | 0,445358 s | 1,91806129 | 0,95903064 |
| 4 | 200.000.000 | 0,268073 s | 3,18218574 | 0,79554643 |
| 8 | 200.000.000 | 0,261382 s | 3,2576428 | 0,40720535 |

Rappresentiamo i seguenti dati su un piano cartesiano in cui sulll’asse orizzontale rappresentiamo il numero dei processori (1, 2, 4, 8) mentre sull’asse verticale il tempo d’esecuzione dell’algoritmo T(p). Dove mettiamo a confronto T(p) al cambiare dell’input N.

Si evince che l’aumento del numero di processori porta a un miglioramento significativo delle prestazioni in tutte e tre le configurazioni, con una diminuzione del tempo di esecuzione e un aumento di S(P). Tuttavia, è importante notare che l’efficienza E(P) può variare e può essere influenzata da fattori come la comunicaizone tra i processi.

### Strategia 3

Viene analizzato il tempo di esecuzione dell’algoritmo utilizzando la strategia 3 :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **p** | **n** | **T(p)** | **S(p)** | **E(p)** |
| 1 | 800.000 | 0,003367 | 1 | 1 |
| 2 | 800.000 | 0,001727 | 1,949623625 | 0,974811812 |
| 4 | 800.000 | 0,00898 | 0,374944321 | 0,09373608 |
| 8 | 800.000 | 0,00493 | 0,68296146 | 0,085370183 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **p** | **n** | **T(p)** | **S(p)** | **E(p)** |
| 1 | 80.000.000 | 0,341128 | 1 | 1 |
| 2 | 80.000.000 | 0,177774 | 1,918885776 | 0,959442888 |
| 4 | 80.000.000 | 0,10794 | 3,160348342 | 0,790087085 |
| 8 | 80.000.000 | 0,105088 | 3,24611754 | 0,405764692 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **p** | **n** | **T(p)** | **S(p)** | **E(p)** |
| 1 | 200.000.000 | 0,853328 | 1 | 1 |
| 2 | 200.000.000 | 0,431338 | 1,978327901 | 0,98916395 |
| 4 | 200.000.000 | 0,268282 | 3,180712832 | 0,795178208 |
| 8 | 200.000.000 | 0,261452 | 3,26380368 | 0,40797546 |

Rappresentiamo i seguenti dati su un piano cartesiano in cui sulll’asse orizzontale rappresentiamo il numero dei processori (1, 2, 4, 8) mentre sull’asse verticale il tempo d’esecuzione dell’algoritmo T(p). Dove mettiamo a confronto T(p) al cambiare dell’input N.

Possiamo notare che in tutte e tre le configurazioni, l’aumento del numero di processori da 1 a 2 porta a un notevole miglioramente delle prestazioni. Tuttavia, l’aumento ulteriore a 4 o 8 processori, può portare a un aumento dei costi dovuti all’overhead di gestione dei processi, che può superare i benefici dell’elaborazione parallela.

# Codice Sorgente

1. #include <stdio.h>

2. #include <stdlib.h>

3. #include <math.h>

4. #include "mpi.h"

5. #include <time.h>

6.

7. /\*Algoritmo per il calcolo della somma di N numeri reali,

8. in ambiente di calcolo parallelo su architettura MIMD a memoria distribuita\*/

9.

10. /\*----------------------------------------------------------

11. I parametri in input vanno inseriti in questo ordine:

12.   1. Numero strategia da utilizzare

13.   2. Numero di valori totali in input N

14.   3. Se N<=20 inserire i valori da sommare, altrimenti inserire solo i 3 parametri precedenti

15. -----------------------------------------------------------\*/

16.

17.

18. void check\_input(int argc, char \*argv[], int nproc);

19.

20. int main(int argc, char \*argv[]) {

21. MPI\_Init(&argc, &argv);

22.

23. int menum, nproc, n, nloc, tag, i, rest, strategia, id, start, tmp, id\_tot\_sum;

24. double \*x;

25. double \*xloc;

26. double t0, t1, timep, timetot, sum\_parz, sum = 0.0;

27. MPI\_Status status;

28.

29. MPI\_Comm\_rank(MPI\_COMM\_WORLD, &menum); // Prendo il rank

30. MPI\_Comm\_size(MPI\_COMM\_WORLD, &nproc); // Prendo il numero di processi

31.

32. // --- INIZIO DISTRIBUZIONE INPUT

33. if (menum == 0) {

34.

35. check\_input(argc, argv, nproc); // Verifico che i parametri rispettino le restrizioni indicate. In caso contrario MPI\_Abort().

36.

37. n = atoi(argv[3]); // Assegno il numero di valori alla variabile n

38. x = (double \*)malloc(n \* sizeof(double)); // Alloco l'array x

39.

40. // --- ASSEGNO VALORI IN ARRAY x A SECONDA DEL NUMERO DI DATI CONSIDERATI

41. if (n > 20) {

42. srand(time(NULL));

43. for (i = 0; i < n; i++)

44. x[i] = ((double)rand() \* 10.0) / (double)RAND\_MAX; // Genero un double random compreso tra 0 e 10

45. } else {

46. for (i = 0; i < n; i++)

47. x[i] = atof(argv[i + 4]); // Assegno i valori passati come argomento ad x

48. }

49.

50. id = atoi(argv[1]); // Assegno rank processo che dovrà stampare il risultato

51. strategia = atoi(argv[2]); // Assegno la strategia da utilizzare

52.

53. if (strategia != 1 && (nproc & (nproc - 1)) != 0) // SE LA STRATEGIA E' DIVERSA DA 1 E "nproc" NON E' UNA POTENZA DI 2 ALLORA FORZO STRATEGIA A 1

54. strategia = 1;

55. printf("STRATEGIA UTILIZZATA PER LA SOMMA: %d\n", strategia); // STAMPO STRATEGIA UTILIZZATA (STAMPA SOLO PROCESSO RANK 0)

56.

57. // --- FINE ASSEGNAZIONE

58.

59. for (i = 1; i < nproc; i++) {

60. tag = 10 + i;

61. MPI\_Send(&n, 1, MPI\_INT, i, tag, MPI\_COMM\_WORLD); // INVIO VALORE n AI PROCESSI DIVERSI DA 0

62. tag = 11 + i;

63. MPI\_Send(&strategia, 1, MPI\_INT, i, tag, MPI\_COMM\_WORLD); // INVIO STRATEGIA AI PROCESSI DIVERSI DA 0

64. tag = 12 + i;

65. MPI\_Send(&id, 1, MPI\_INT, i, tag, MPI\_COMM\_WORLD); // INVIO ID PROCESSO CHE DEVE STAMPARE AI PROCESSI DIVERSI DA 0

66. }

67. } else {

68. tag = 10 + menum;

69. MPI\_Recv(&n, 1, MPI\_INT, 0, tag, MPI\_COMM\_WORLD, &status); // RICEZIONE VALORE n DA PROCESSO 0

70. tag = 11 + menum;

71. MPI\_Recv(&strategia, 1, MPI\_INT, 0, tag, MPI\_COMM\_WORLD, &status); // RICEZIONE STRATEGIA DA PROCESSO 0

72. tag = 12 + menum;

73. MPI\_Recv(&id, 1, MPI\_INT, 0, tag, MPI\_COMM\_WORLD, &status); // RICEZIONE ID PROCESSO STAMPA DA PROCESSO 0

74. }

75. // --- INIZIO DISTRIBUZIONE INPUT

76.

77. nloc = n / nproc; // Calcolo n locale

78. rest = n % nproc; // calcolo eventuale resto divisione

79. if (menum < rest) nloc = nloc + 1; // fix n locale

80. xloc = (double \*)malloc(nloc \* sizeof(double)); // Alloco array xloc per quanti valori il processo i-esimo ne ha bisogno (nloc)

81.

82. // --- INIZIO DISTRIBUZIONE VALORI DA SOMMARE AI VARI PROCESSI

83. if (menum == 0) {

84. xloc = x;

85. tmp = nloc;

86. start = 0;

87. for (i = 1; i < nproc; i++) {

88. start = start + tmp;

89. tag = 22 + i;

90. if (i == rest) tmp = tmp - 1;

91. MPI\_Send(&x[start], tmp, MPI\_DOUBLE, i, tag, MPI\_COMM\_WORLD);

92. }

93. } else {

94. tag = 22 + menum;

95. MPI\_Recv(xloc, nloc, MPI\_DOUBLE, 0, tag, MPI\_COMM\_WORLD, &status);

96. }

97. // --- FINE DISTRIBUZIONE VALORI DA SOMMARE

98.

99. MPI\_Barrier(MPI\_COMM\_WORLD); // BARRIERA PER CALCOLARE POI I TEMPI

100. t0 = MPI\_Wtime(); // Prendo il primo tempo

101.

102. // --- INIZIO CALCOLO SOMMA LOCALE

103. for (i = 0; i < nloc; i++) {

104. sum = sum + xloc[i];

105. }

106. // --- FINE CALCOLO SOMMA LOCALE

107.

108. // --- INIZIO PRIMA STRATEGIA

109. if (strategia == 1) {

110. id\_tot\_sum = id;

111. if (id\_tot\_sum == -1) id\_tot\_sum = 0; // Se devono stampare tutti, rank 0 avrà la somma totale

112. if (menum == id\_tot\_sum) {

113. for (i = 0; i < nproc; i++) {

114. if (menum != i) { // ricevo solo dai processi diversi da me stesso

115. tag = 80 + i;

116. MPI\_Recv(&sum\_parz, 1, MPI\_DOUBLE, i, tag, MPI\_COMM\_WORLD, &status); // Ricevo dal processo i

117. sum = sum + sum\_parz; // Aggiungo la somma parziale inviatami alla mia somma parziale

118. }

119. }

120. } else {

121. tag = 80 + menum;

122. MPI\_Send(&sum, 1, MPI\_DOUBLE, id\_tot\_sum, tag, MPI\_COMM\_WORLD); // Invio sum al processo che dovrà stampare alla fine (se -1 invio a 0)

123. }

124. }

125. // --- FINE PRIMA STRATEGIA

126. // --- INIZIO SECONDA STRATEGIA (STRATEGIA COSTRUITA IN MANIERA TALE CHE L'ALBERO RUOTI NEL CASO IN CUI DEBBANO STAMPARE

127. // PROCESSI DIVERSI DA 0)

128. else if (strategia == 2) {

129. int recv, send;

130. id\_tot\_sum = id;

131. if (id\_tot\_sum == -1) id\_tot\_sum = 0; // Se devono stampare tutti, rank 0 avrà la somma totale

132. for (i = 0; i < (log10(nproc) / log10(2)); i++) {

133. if (fmod((menum - id\_tot\_sum + nproc) % nproc, (pow(2, i))) == 0) { // chi partecipa (rotazione data dal primo membro di fmod)

134. if (fmod((menum - id\_tot\_sum + nproc) % nproc, (pow(2, i + 1))) == 0) { // decido chi deve ricevere e chi inviare

135. tag = menum;

136. recv = (int)(menum + pow(2, i)) % nproc; // da chi ricevo la somma parziale

137. MPI\_Recv(&sum\_parz, 1, MPI\_DOUBLE, recv, tag, MPI\_COMM\_WORLD, &status);

138. sum = sum + sum\_parz;

139. } else {

140. send = (((int)(menum - pow(2, i)) % nproc) + nproc) % nproc; // a chi invio la somma parziale (formula modulo non negativo)

141. tag = send;

142. MPI\_Send(&sum, 1, MPI\_DOUBLE, send, tag, MPI\_COMM\_WORLD);

143. }

144. }

145. }

146. }

147. // --- FINE SECONDA STRATEGIA

148. // --- INIZIO TERZA STRATEGIA. Send e Recv sono invertite negli if per evitare Dead Lock.

149. // I processi nell'else non fanno nessuna somma ma aggiornano solo il proprio "sum" col valore prelevato dalla Recv

150. else if (strategia == 3) {

151. for (i = 0; i < (log10(nproc) / log10(2)); i++) {

152. if (fmod(menum, (pow(2, i + 1))) < pow(2, i)) {

153. tag = menum;

154. MPI\_Recv(&sum\_parz, 1, MPI\_DOUBLE, menum + pow(2, i), tag, MPI\_COMM\_WORLD, &status);

155. tag = menum + pow(2, i);

156. MPI\_Send(&sum, 1, MPI\_DOUBLE, menum + pow(2, i), tag, MPI\_COMM\_WORLD);

157. } else {

158. tag = menum - pow(2, i);

159. MPI\_Send(&sum, 1, MPI\_DOUBLE, menum - pow(2, i), tag, MPI\_COMM\_WORLD);

160. tag = menum;

161. MPI\_Recv(&sum\_parz, 1, MPI\_DOUBLE, menum - pow(2, i), tag, MPI\_COMM\_WORLD, &status);

162. }

163. sum = sum + sum\_parz; // Ogni processo fa la somma col valore sum\_parz che ha ricevuto dall'apposito processo

164. }

165. }

166. // --- FINE TERZA STRATEGIA

167.

168. t1 = MPI\_Wtime(); // Prendo il secondo tempo

169. timep = t1 - t0; // Calcolo la differenza per capire il tempo impiegato dal singolo processo

170. MPI\_Reduce(&timep, &timetot, 1, MPI\_DOUBLE, MPI\_MAX, 0, MPI\_COMM\_WORLD); // Calcolo il tempo massimo e lo pongo in "timetot" al processo 0

171.

172. // --- INIZIO STAMPA RISULTATI

173. if (id == -1) {

174. if (strategia != 3)

175. MPI\_Bcast(&sum, 1, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD); // Invio in broadcast da 0 la somma a tutti (quando id = -1 la somma totale la ha solo rank 0 per convenzione)

176.

177. printf("Somma calcolata da %d è %lf\n", menum, sum); // stampano tutti i processi la somma totale

178. } else if (menum == id) {

179. printf("Somma calcolata da %d è %lf\n", menum, sum); // stampa solo il processo "id" che avrà la somma totale

180. }

181. // --- FINE STAMPA RISULTATI

182.

183. if (menum == 0) {

184. printf("Tempo massimo %lf\n", timetot); // Stampo tempo massimo di somma + strategia adottata

185. }

186.

187. MPI\_Finalize();

188. return 0;

189. }

190.

191. void check\_input(int argc, char \*argv[], int nproc) {

192. if (argc < 4) {

193. printf("INSERIRE ALMENO 3 ARGOMENTI:\n1. ID processo stampa\n2. Strategia da utilizzare\n3. Numero di dati in input\n");

194. MPI\_Abort(MPI\_COMM\_WORLD, 1);

195. }

196. if (atoi(argv[1]) >= nproc) {

197. printf("NUMERO PROCESSO NON VALIDO!!! INSERIRE UN VALORE COMPRESO TRA -1 e %d\n", nproc - 1);

198. MPI\_Abort(MPI\_COMM\_WORLD, 2);

199. }

200. if (atoi(argv[2]) < 1 || atoi(argv[2]) > 3) {

201. printf("NUMERO STRATEGIA NON VALIDO!!! INSERIRE VALORE COMPRESO TRA 1 E 3\n");

202. MPI\_Abort(MPI\_COMM\_WORLD, 3);

203. }

204. if (atoi(argv[3]) <= 0) {

205. printf("NUMERO DATI INPUT NON VALIDO!!! INSERIRE UN VALORE NON NEGATIVO\n");

206. MPI\_Abort(MPI\_COMM\_WORLD, 4);

207. }

208. if (atoi(argv[3]) > 20 && argc > 4) {

209. printf("INSERITI DATI IN INPUT CON VALORE N MAGGIORE DI 20. NON INSERIRE NESSUN DATO IN INPUT\n");

210. MPI\_Abort(MPI\_COMM\_WORLD, 5);

211. }

212. if ((atoi(argv[3]) > 0 && atoi(argv[3]) <= 20) && argc - 4 != atoi(argv[3])) {

213. printf("NUMERO DI DATI IN INPUT NON CORRETTO CON IL VALORE INDICATO N\n");

214. MPI\_Abort(MPI\_COMM\_WORLD, 6);

215. }

216. }

217.

218.