Università Degli Studi di Napoli

Parthenope

Documentazione esterna di un Software Parallelo

per la relazione relativa al progetto di
Calcolo Parallelo e Distribuito

Ilardo Gianluca – 0124/2368
Zibaldo Francesco – 0124/2176

Prof. Livia Marcellino

14/06/2022

Sommario

Traccia

Implementazione di un algoritmo parallelo (np processori) per il calcolo dell'operazione c=a+b con a,b vettori di dimensione N, in ambiente MPI-Docker

Definizione ed Analisi del problema

Questo software effettua la somma puntuale tra due vettori di dimensione N. Prevede pertanto in input due vettori, a e b, di dimensione N e restituisce in output un vettore c, di dimensione N. L'elemento i-simo del vettore c sarà dunque la somma tra l'elemento i-simo del vettore a e l'elemento i-simo del vettore b.

Per comprendere meglio il problema, lo osserviamo partendo dall'analisi dell'algoritmo sequenziale.
L'algoritmo deve effettuare la somma puntuale tra due vettori di dimensione N, come spiegato nella definizione ed analisi del problema. Inizialmente, nell'algoritmo sequenziale, vengono dichiarati due vettori a e b e un intero n che rappresenterà la loro dimensione. Successivamente, attraverso delle opportune scanf, l'utente inserisce la dimensione n degli array e, uno ad uno, i rispettivi elementi di a e b. L'algoritmo poi, attraverso un ciclo for che scorre gli elementi dei vettori, calcola la somma puntuale e la inserisce nel vettore c. Infine, stampa a video il risultato, ovvero gli n elementi dell'array c.
È qui riportato in pseudocodice il blocco di codice principale dell'algoritmo, ovvero il ciclo for:

```
 for( i = 0 to n-1)
{
    [i] = a[i] + b[i];
    print(c[i]);
}
```

A partire da ciò, possiamo ricavare la complessità computazionale dell'algoritmo sequenziale, ovvero $T_1(N)$.

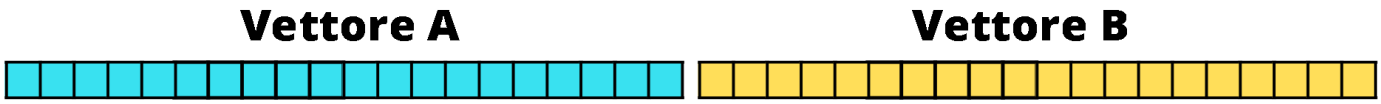
Il ciclo for esegue una somma ad ogni iterazione e le iterazioni sono N (da 0 a N-1), quindi le somme sono N.

Pertanto, $T_1(N) = N$

Descrizione dell'approccio parallelo

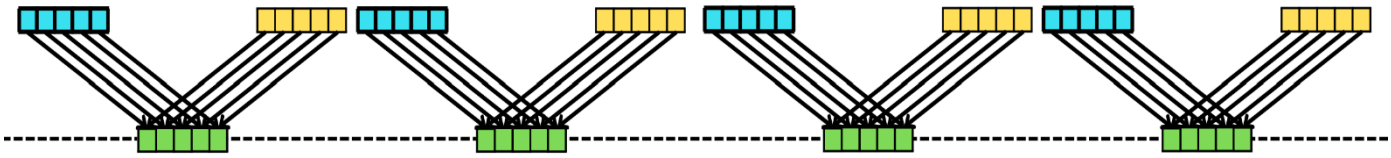
Il calcolo di ogni elemento del vettore c può essere fatto indipendentemente dal calcolo di ogni altro suo elemento. c_i sarà completamente diverso ed indipendente da c_{i+1}. Ad esempio, nel calcolo di c_0 saranno coinvolti solamente a_0 e b_0. Dunque, si tratta di un problema definibile elementwise, per questa ragione completamente parallelizzabile. La strategia di parallelizzazione prevede la suddivisione del vettore a e del vettore b in sottovettori (rispettivamente a_loc_i e b_loc_i) da assegnare a ciascun processore/unità processante. Questi ultimi effettueranno le somme locali separatamente, senza la necessità di collezionare nulla. Da ciò possiamo intuire che non ci sarà nessuna frazione sequenziale, ed inoltre possiamo prevedere, almeno dal punto di vista teorico, delle ottime aspettative circa l'efficienza, magari prossima all'efficienza ideale.

Consideriamo un esempio grafico per entrare meglio nella logica dell'algoritmo.
In questo esempio consideriamo una dimensione computazionale N pari a 20, ovvero i due vettori A e B sono formati da 20 elementi e di conseguenza anche il vettore C.



Per semplicità consideriamo che il numero di processori p sia 4. Il numero di elementi N è esattamente divisibile per il numero p di processori. Da ciò ricaviamo che ad ogni processore saranno affidati 5 elementi del vettore A e 5 elementi del vettore B.
Nel caso in cui il numero di elementi N non sia esattamente divisibile per il numero di processori, si procederà ridistribuendo gli elementi rimanenti, assegnando un elemento in più ai

processori il cui identificativo risulta strettamente minore del resto.
I processori, dopo aver ricevuto i loro cinque elementi di ciascun vettore, procedono nell'effettuare le somme puntuali dei vettori, ovvero nel calcolo degli elementi del vettore C.
Il processore P_0 calcolerà i primi 5 elementi del vettore C (da c_0 a c_4); il processore P_1 definirà i successivi 5 elementi del vettore C (da c_5 a c_9); il processore P_2 provvederà al calcolo dei seguenti 5 elementi del vettore C (da c_{10} a c_{14}); infine il processore P_3 effettuerà cinque somme da assegnare agli ultimi 5 elementi del vettore C (da c_{15} a c_{19}).



Bisognerà organizzare opportunamente le stampe, per ottenere la stampa di tutti gli elementi del vettore C.

Attraverso le metriche di fattibilità di un algoritmo parallelo, valutiamo la bontà dell'algoritmo.

Premessa: le spedizioni relative alla distribuzione dei dati iniziali non sono contemplate nei conti di speed-up, overhead ed efficienza.

Speed-up

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

$$Sp = \frac{T_1}{T_p}$$

poiché:

$T_1 = N$ e $T_p = \frac{N}{nproc}$

$$Sp = \frac{N}{\frac{N}{nproc}} \rightarrow Sp = nproc \rightarrow Sp = p = \mathbf{S_p^{ideale}}$$

Lo speed-up pertanto coincide con lo speed-up ideale, ovvero abbiamo raggiunto, almeno dal punto di vista teorico, la massima aspirazione possibile.

Overhead Totale

L’overhead misura quanto lo speed-up differisce da quello ideale. Possiamo prevedere che, avendo uno speed-up pari allo speed-up ideale, questa differenza sarà nulla.

$$O_h = (pT_p - T_1)\, t_{calc}$$

poiché:

$$T_1 = N \qquad \text{e} \qquad T_p = \frac{N}{p}$$

$$O_h = (p\ \frac{N}{p} - N)\, t_{calc} \quad \text{->} \quad O_h = 0$$

Poiché in altri termini l’overhead rappresenta lo spreco, ovvero quanto non sono riuscito a parallelizzare, tale risultato ci indirizza verso la consapevolezza di aver scelto una buona strategia di parallelizzazione.

Efficienza

L’efficienza è definita come il rapporto tra lo Speed-up ed il numero di processori.

$$E_p = \frac{S_p}{p}$$

poiché:

$$Sp = p = S_p^{ideale}$$

$$E_p = \frac{S_P^{ideale}}{p} = 1 = E_p^{ideale}$$

L’efficienza misura quanto l’algoritmo sfrutta il parallelismo del calcolatore e, come avevamo osservato già precedentemente, non vi è alcuna frazione sequenziale, bensì il problema è completamente parallelizzabile.

Speed-up con la caratterizzazione di Ware-Amdahl

Con la legge di Ware-Amdahl viene caratterizzato lo speed-up, in modo da rivedere questa relazione.

$$S_p = \frac{1}{\alpha + \frac{1-\alpha}{p}}$$

dove:

- α indica il tempo per eseguire la parte sequenziale
- $\frac{1-\alpha}{p}$ indica il tempo per eseguire la parte parallela

Osservazione: In questo caso abbiamo utilizzato la formulazione base della legge di Ware-Amdahl, in quanto nel nostro algoritmo non c'è parallelismo medio, ovvero c'è una netta distinzione tra la frazione sequenziale e la frazione parallela.

Sia **p=4, N=20, a_loc=5, b_loc=5**

Nel nostro caso abbiamo unicamente una fase di Calcolo delle Somme, senza necessità di una fase di Collezione dei risultati. La fase di Calcolo delle somme è puramente parallela.

Poiché la complessità computazionale dell'algoritmo sequenziale è N, il numero totale delle somme da eseguire sarà 20.

$$T_1 = N = 20$$

Nella fase di calcolo delle somme, che risulta totalmente parallela, il numero di somme che si effettuano è pari al numero di elementi da sommare, ovvero 5. Infatti, ogni processore dovrà eseguire la somma puntuale tra gli elementi del vettore a e gli elementi del vettore b che gli sono stati assegnati.

Queste 5 somme saranno eseguite contemporaneamente dai 4 processori.
Verranno per tanto eseguite 5*4 = 20 somme

pertanto:

$$1 - \alpha = \frac{20}{20} \qquad \text{e} \qquad \frac{1-\alpha}{p} = \frac{20}{20} * \frac{1}{4} = \frac{1}{4}$$

Come osservato precedentemente, non va considerata nessuna fase sequenziale.

Quindi:

$$\alpha = 0$$

Ora abbiamo tutti gli elementi da sostituire per calcolare lo Speed-up:

$$S_p = \frac{1}{\alpha + \frac{1-\alpha}{p}} \rightarrow \qquad S_p = \frac{1}{0 + \frac{\frac{20}{20}}{4}} = \frac{1}{\frac{1}{4}} = 4$$

Come si era già precedentemente stimato, lo speed-up è pari al numero di processori (in questo caso 4), dunque è pari allo Speed-up ideale.

Isoefficienza

L'isoefficienza è una funzione di tre variabili p_0, p_1, n_0 e definisce la costante che lega la nuova dimensione del problema n_1 da scegliere per valutare la scalabilità di un algoritmo.
Con scalabilità ci si riferisce infatti alla capacità di un algoritmo di mantenere costante l'efficienza al crescere del numero dei processori e della dimensione del problema.

$$\mathbf{I\ (p_0,\ p_1,\ n_0) = \frac{O_h(p_1,n_1)}{O_h(p_0,n_0)}}$$

Ma $\mathbf{O_h(p_1, n_1) = 0}$ **e** $\mathbf{O_h(p_0, n_0) = 0}$

$$\mathbf{I\ (p_0,\ p_1,\ n_0) = \frac{0}{0}} \Rightarrow \textbf{Forma Indeterminata}$$

Per convenzione l'isoefficienza viene posta uguale ad infinito, ovvero posso usare qualunque costante moltiplicativa per calcolare n1 e quindi controllare la scalabilità dell'algoritmo.

Descrizione dell'algoritmo parallelo

Il nostro algoritmo parallelo inizia con la dichiarazione delle varabili principali che useremo:
-gli array a e b;
-la loro dimensione n;
-la variabile di iterazione i;
-la variabile rest e la variabile indice indice_displs (inizialmente settata a 0) che ci serviranno per gestire la non perfetta divisibilità degli elementi;
-le variabili menum e nproc che rappresenteranno rispettivamente l'identificativo dei processori e il numero totale dei processori, e che vengono utilizzate dalle routine MPI_Comm_size ed MPI_Comm_rank;
-gli array count_send e displs, che conterranno rispettivamente il numero di elementi da assegnare ad ogni processore con la scatterv e la posizione da cui iniziare a prendere questi ultimi.

Seguirà poi, grazie al costrutto condizionale if(menum == 0), un blocco di istruzioni eseguito solo dal processore master (con identificativo 0), all'interno del quale l'utente può scegliere la dimensione degli array ed i loro elementi attraverso una scanf e degli opportuni cicli for.
Successivamente al blocco di istruzioni eseguito dal processore master, la routine MPI_Bcast condivide la dimensione n con gli altri processori, ed ha inizio un blocco di istruzioni che definisce gli array count_send e displs, gestendo anche il caso della non perfetta divisibilità tra il numero di elementi dei vettori ed il numero di processori.
Per illustrare al meglio questo blocco di istruzioni, ne abbiamo selezionato le parti salienti e lo abbiamo reso in pseudocodice:

```
rest = n%nproc;
for (i = 0 to nproc)
{
    count_send[i] = n/nproc;
    if (rest > 0)
    {
        count_send[i] = count_send[i] + 1;
        rest = rest - 1;
    }
displs[i] = indice_displs;
indice_displs = indice_displs+count_send[i]
}
```

Come possiamo osservare, viene calcolato il resto della divisione tra il numero di elementi e il numero di processori. Successivamente, entrati nel ciclo for che va da 0 al numero di processori, si scorre l'array count_send[], assegnando ad ogni suo elemento il valore della divisione tra il numero di elementi e il numero di processori. Per gestire la non perfetta divisibilità però, se il resto calcolato precedentemente è maggiore di 0, allora l'elemento i-esimo di count_send sarà incrementato di uno, e il resto decrementato di uno. In questo modo ai primi "rest-processori", sarà assegnato un elemento in più. Infine, viene riempito l'array displs: ogni elemento viene reso uguale alla variabile indice_displs, che inizialmente è uguale a 0 (quindi displ[0] sarà 0), ma ad ogni iterazione del ciclo for viene sommato l'elemento i-esimo di count_send. Questi calcoli faranno in modo che ad ogni processore sarà inviato il giusto numero di elementi,

presenti all'interno dell'array count_send (per intenderci, al processore con identificativo 0 saranno inviati count_send[0] elementi, al processore con identificativo 1 count_send[1] elementi e così via).
Fatto questo calcolo, verranno allocati i sottoarray ap, bp e cp, di dimensione count_send[menum]: nei primi due verranno distribuiti gli elementi di a e b attraverso la function scatterv (spiegata all'interno del paragrafo delle routine), nel terzo ci sarà la somma puntuale degli elementi di ab e bp.
Infine, nel codice è presente un semplice ciclo for che somma gli elementi di ap e bp all'interno dell'array cp, e stampa la somma di ogni processore a video. Mostriamo attraverso lo pseudocodice il ciclo for:

```
for(i=0 to cont_send[menum]
    cp[i] = ap[i] + bp[i];
print(ap[i] + bp[i] = cp[i])
```

Il codice si chiude con la routine MPI_Finalize(), che segna la fine della fase parallela dell'algoritmo, e il return 0.

Input e Output

Il nostro software prende inizialmente in input un intero che indica la dimensione N; esso definisce il numero di elementi che costituiranno i vettori A, B, C.

Una volta definito il numero di elementi, il software prende in input N interi che rappresentano gli elementi del vettore A ed N interi che rappresentano gli elementi del vettore B. Terminato l'inserimento, gli elementi vengono divisi tra i processori, viene effettuato il calcolo e viene stampato a video il risultato della somma puntuale tra i vettori. L'output ottenuto a video va interpretato, dunque, come il risultato della somma elemento per elemento tra i vettori. Vengono quindi restituiti degli interi e in aggiunta viene indicata la provenienza del risultato, ovvero il processore che ha calcolato la specifica somma.

Consideriamo, ad esempio, come dimensione N il valore 8.

Procediamo definendo il vettore A con i seguenti dati:

2, 5, 6, 3, 7, 9, 0, 4

Inizializziamo il vettore B con i seguenti valori:

6, 5, 8, 3, 1, 2, 7, 9

Possiamo aspettarci che vengano stampati a video i seguenti output:

"risultato = 8 from 0"

"risultato = 10 from 0"

"risultato = 14 from 1"

"risultato = 6 from 1"

"risultato = 8 from 2"

"risultato = 11 from 2"

"risultato = 7 from 3"

"risultato = 13 from 3"

Routine implementate

- **MPI_Init (int *argc, char ***argv);**

La routine in questione appartiene alle funzioni per definire l'ambiente.

Questa routine inizializza l'ambiente di esecuzione MPI. Inizializza il communicator MPI_COMM_WORLD. Ha due dati di input che sono gli argomenti del main.

- **MPI_Comm_size (MPI_Comm comm, int *size);**

Anche questa routine appartiene alle funzioni per definire l'ambiente.

Questa routine permette al processore chiamante di memorizzare nella variabile size il numero totale di processori concorrenti appartenenti al communicator MPI_COMM_WORLD.

- **MPI_Comm_rank (MPI_Comm comm, int *rank);**

Questa routine è una delle funzioni necessarie per definire l'ambiente.

Permette al processore chiamante, appartenente al communicatore MPI_COMM_WORLD, di memorizzare il proprio identificativo nella variabile rank.

- **MPI_Bcast(void *buffer, int count,MPI_Datatype datatype,int root, MPI_Comm comm);**

Questa routine appartiene alle funzioni per le comunicazioni collettive di un messaggio.

Il processore con identificativo root spedisce a tutti i processori del comunicator comm lo stesso dato memorizzato in *buffer.

Nel nostro software questa routine è stata utilizzata per distribuire il size n degli array a e b e il size n_el_proc dei sottoarray ap, bp, e cp, dal processore master agli altri processori.

- buffer -> indirizzo del dato da spedire
- count -> numero dei dati da spedire
- datatype -> tipo dei dati da spedire
- root -> identificativo del provessore che spedisce a tutti
- comm -> identificativo del communicator

- **MPI_Scatterv(void *send_buff, int *send_counts, int *displs, MPI_Datatype send_type, void *recv_buff, int recv_count, MPI_Datatype recv_type, int root, MPI_Comm comm);**

Questa routine fa parte delle funzioni relative alle comunicazioni collettive di un messaggio,

Il processore con identificativo root suddivide i dati contenuti in send_buff, inviando segmenti di lunghezza variabile, indicata nel vettore send_counts. Attraverso il vettore displs ci si sposta sul vettore send_buff. Il primo segmento viene affidato al processore con identificativo 0, il secondo al processore con identificativo 1 e così via.

Nel nostro software questa routine è stata utilizzata per distribuire i sottoarray degli array a e b, ap e bp, tra i vari processori. Il root in questo caso è il nostro processore master, con identificativo 0.

- *send_buff -> indirizzo del dato da spedire
- send_counts -> vettore che specifica il numero di elementi da inviare a ciascun processore
- displs -> vettore che specifica lo spostamento su send_buff
- send_type -> tipo dei dati da spedire
- *recv_buff -> indirizzo del dato in cui ricevere
- recv_count ->numero dei dati da ricevere
- recv_type -> tipo dei dati da ricevere
- root -> identificativo del processore che spedisce a tutti
- comm -> identificativo del communicator

- MPI_finalize();

Questa routine fa parte delle funzioni per definire l'ambiente.

Determina la fine del programma MPI. Dopo questa routine non è possibile richiamare nessun'altra routine MPI.

Analisi delle performance del software

Abbiamo preso i tempi d'esecuzione e li abbiamo riportati in tabelle. I tempi sono stati presi variando la dimensione dell'input e il numero di processori impiegati. Inoltre, abbiamo inserito dei grafici per lo speed-up e per l'efficienza. Le esecuzioni sono state effettuate da una macchina con le seguenti caratteristiche:

- **Processore intel core 7t 9th gen**
- **Ram 16 gb**
- **Scheda video Nvidia Geforce Gtx 1660 ti**
- **Memoria 1 terabyte di SSD**
- **12 core**

Tempi di esecuzione

Numero di processori	N = 10	N = 100	N = 1000	N = 10000	N = 100000	N = 500000
1	0,00001	0,00006	0,00074	0,01346	0,01499	6,20640
2	0,00005	0,00005	0,00045	0,0057	0,06423	3,53541
4	0,00076	0,00072	0,00088	0,004	0,04746	2,07015
8	0,00468	0,00469	0,00477	0,01631	0,06362	2.05179
16	0,14889	0,30998	0,30025	0,31172	0,43987	2.24021

Speed-up

Numero di processori	N = 10	N = 100	N = 1000	N = 10000	N = 100000	N = 500000
2	0,2	1,2	1,6444444	2,361403509	0,23338004	1,755496534
4	0,0131579	0,0833333	0,8409091	3,365	0,315844922	2,99804362
8	0,0021368	0,0127932	0,1551363	0,825260576	0,23561773	3,024870966
16	6,716E-05	0,0001936	0,0024646	0,043179777	0,03407825	2,770454556

Efficienza

Numero di processori	N = 10	N = 100	N = 1000	N = 10000	N = 100000	N = 500000
2	0,1	0,6	0,8222222	1,180701754	0,11669002	0,877748267
4	0,0032895	0,0208333	0,2102273	0,84125	0,078961231	0,749510905
8	0,0002671	0,0015991	0,019392	0,103157572	0,029452216	0,378108871
16	4,198E-06	1,21E-05	0,000154	0,002698736	0,002129891	0,17315341

Grafico dei tempi

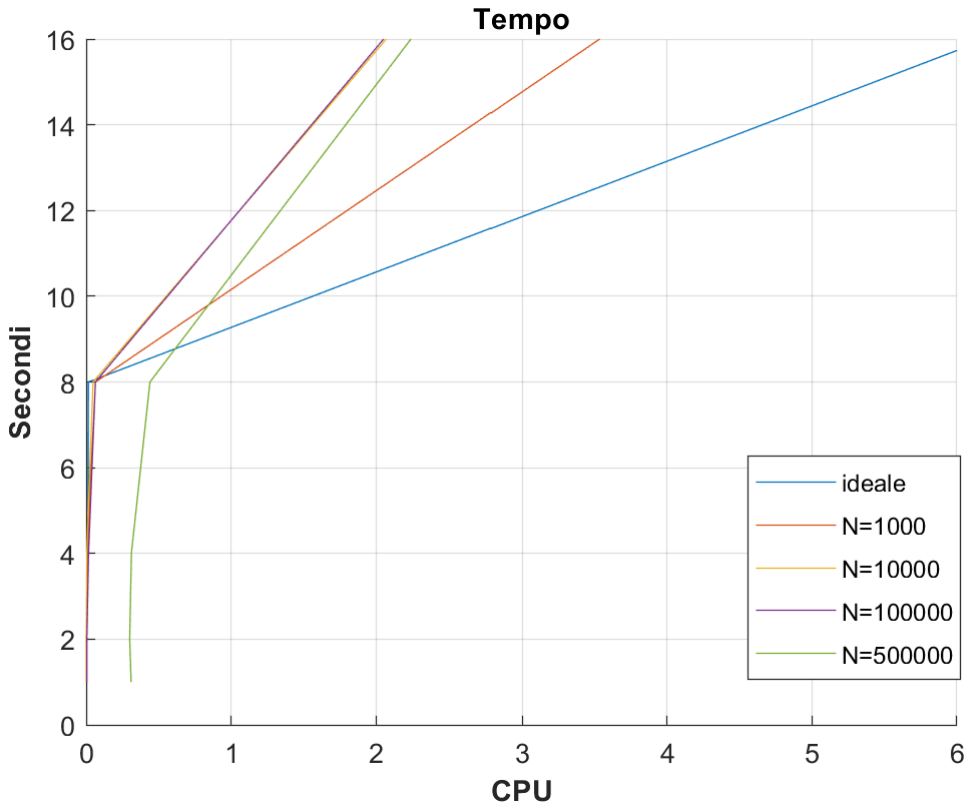


Grafico dello Speed-up

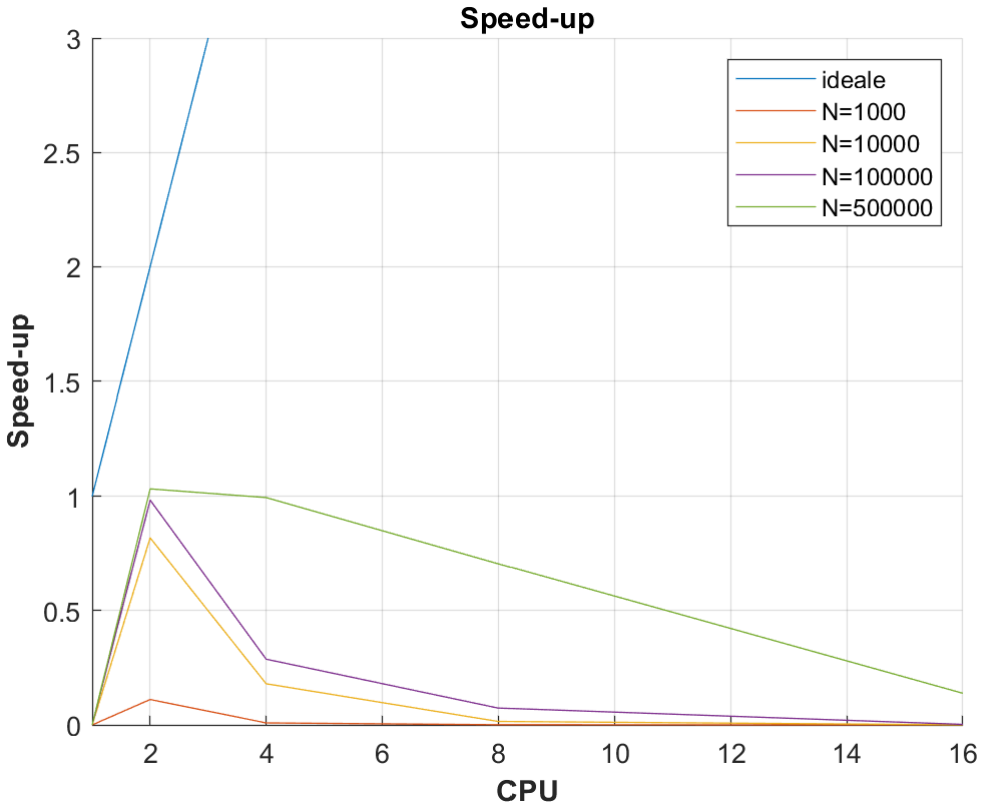
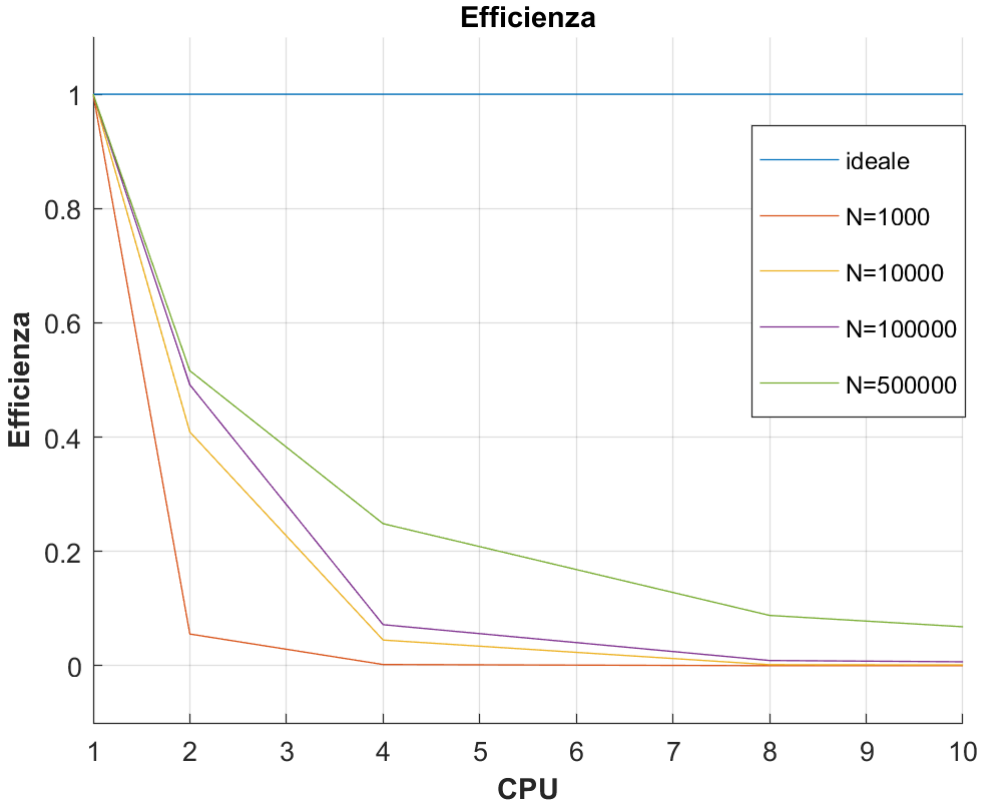


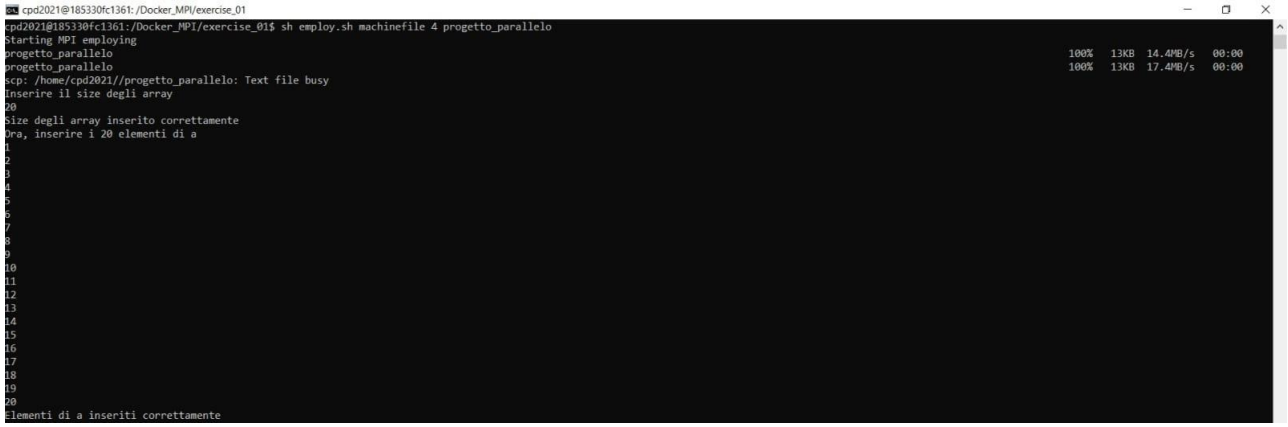
Grafico dell'Efficienza



Esempi d'uso

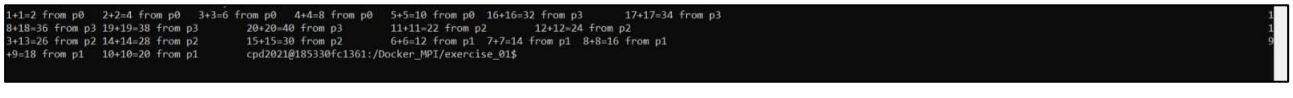
In questa sezione sono rappresentati due esempi di esecuzione del nostro algoritmo.
Nel primo esempio la dimensione N è pari a 20 elementi (perfettamente divisibile tra i quattro processori utilizzati).
Nel secondo esempio N è pari a 9 elementi (non perfettamente divisibile tra quattro processori utilizzati).
Iniziamo con l'osservare il primo:

Inserimento da parte dell'utente del size degli array e degli n elementi di a:



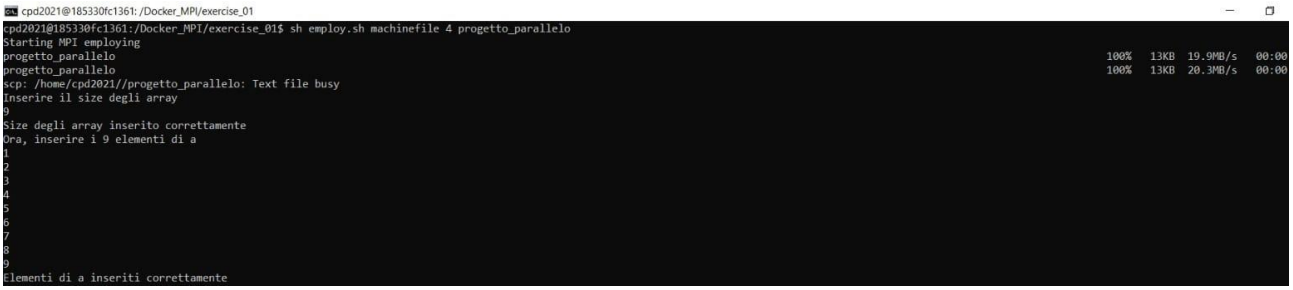
Inserimento degli n elementi di b:

Visualizzazione della somma puntuale degli elementi di a e b, effettuata dai vari processori:



Adesso, osserviamo il secondo esempio:

Inserimento da parte dell'utente del size degli array e degli n elementi di a:



Inserimento degli n elementi di b:

Visualizzazione della somma puntuale degli elementi di a e b, effettuata dai vari processori:

Osservazione: gli elementi inseriti negli esempi sono i numeri da 1 a n, ma solo per una maggiore comodità nell'inserimento e nella visualizzazione degli esempi; può essere infatti inserito qualsiasi intero e la somma verrà comunque effettuata correttamente. Inoltre, gli elementi di c non sono stampati in ordine: questo perché i processori effettuano le operazioni in parallelo e non in ordine di identificativo. Avremmo potuto riassemblare l'array c, all'interno del processore master, con la routine Gatherv() e far stampare in ordine gli elementi del vettore c, ma abbiamo scartato questa opzione in quanto l'operazione non era esplicitamente richiesta dalla traccia. Inoltre, riassemblare il vettore c avrebbe inficiato le metriche di fattibilità dell'algoritmo, poiché avrebbe comportato una maggiore complessità di tempo.

Riferimenti bibliografici

Come riferimenti bibliografici, per la stesura del codice e della relazione, sono stati utilizzate alcune slide delle lezioni del corso di *"Calcolo parallelo e distribuito"* della docente Livia Marcellino, presenti sul portale E-Learning dell'università Parthenope, e il manuale *"MPI MPICH Model MPI Implementation Referen e Manual"*, presente sul medesimo portale, di cui riportiamo gli autori:

William Gropp;

Ewing Lusk Mathematics and Computer Science Division;

Argonne National Laboratory;

Nathan Doss;

Anthony Skjellum;

Department of Computer Science;

Mississippi State University.

Appendice

Riportiamo il codice scritto, compresa la documentazione interna: abbiamo commentato opportunamente il codice affinché sia di facile lettura e comprensione per chi lo analizza.

Algoritmo Parallelo

```
1. #include "mpi.h"
2. #include <stdio.h>
3. #include <stdlib.h>
4.
5.
6.
7. int main (int argc, char *argv[])
8. {
9.     //dichiarazione degli array da sommare
10.       int * a;
11.       int * b;
12.
13.       int nproc;
14.       int menum;
15.
16.       int i, n;
17.       int rest;
18.       int indice_displs = 0;
19.             //inizializzazione dell'ambiente parallelo
20.   MPI_Init (&argc, &argv);
21.
22.   //numero di processori inserito nella variabile nproc
23.   MPI_Comm_size (MPI_COMM_WORLD, &nproc);
24.
25.   //identificativo dei processori inserito nella variabile
    menum
26.   MPI_Comm_rank (MPI_COMM_WORLD,&menum);
27.
28.   //array che indica quanti elementi assegnare ad ogni pro
   cessore
29.   int * count_send = (int *) malloc(sizeof(int)*nproc);
30.
31.   //array che indica da quale posizione degli array assegn
   are gli elementi ad ogni processore
32.   int * displs = (int *) malloc(sizeof(int)*nproc);
33.
```

```
34.   //dichiarazione dei sottoarray
35.   int * ap;
36.   int * bp;
37.   int * cp;
38.   /*operazioni effettuate dal processore master:
39.         l'utente sceglie le dimensioni e gli elementi dell
   'array*/
40.       if (menum == 0)
41.       {
42.                 printf("Inserire il size degli array\n");
43.                 scanf ("%d", &n);
44.                 a = (int *) malloc(sizeof(int)*n);
45.                 b = (int *) malloc(sizeof(int)*n);
46.
47.                 printf("Size degli array inserito correttame
   nte\n");
48.
49.                 printf("Ora, inserire i %d elementi di a\n",
    n);
50.                 for(i = 0; i < n; i++)
51.                     scanf ("%d", &a[i]);
52.   printf("Elementi di a inseriti correttamente\n");
53.
54.                 printf("Ora, inserire i %d elementi di b\n",
    n);
55.                 for(i = 0; i < n; i++)
56.                         scanf ("%d", &b[i]);
57.
58.       printf("Elementi di b inseriti correttamente, adesso
    verrà visualizzata la somma\n");
59.
60.       }
61.                 //distribuzione della dimensione degli array
    agli altri processori
62.                 MPI_Bcast (&n, 1, MPI_INT, 0, MPI_COMM_WORLD
   );
63.   rest = n%nproc;
64.
65.                 //gestione della distribuzione degli element
   i e della non perfetta divisibilità
66.   /*ad ogni processore vengono assegnati n/nproc elementi,
    ma se questa divisone da resto maggiore di 0, allora i pri
   mi "rest" processori avranno un elemento in più*/
67.                 for (i = 0; i < nproc; i++)
68.                 {
69.                     count_send[i] = n/nproc;
70.                     if (rest > 0)
71.                     {
72.                         count_send[i]++;
```

```
                        rest--;
                    }

                displs[i] = indice_displs;
                indice_displs += count_send[i];
    }

                /*allocazione dei sottoarray:
                  ognuno è di dimensione pari al numero di e
    lementi che gli verrà assegnato*/
                ap = (int *) malloc(sizeof(int)*count_send[m
    enum]);
                bp = (int *) malloc(sizeof(int)*count_send[m
    enum]);
                cp = (int *) malloc(sizeof(int)*count_send[m
    enum]);

                //distribuzione degli array a e b con le rou
    tine MPI_Scatterv
                MPI_Scatterv(a, count_send, displs, MPI_INT,
     ap, count_send[menum], MPI_INT, 0, MPI_COMM_WORLD);

                MPI_Scatterv(b, count_send, displs, MPI_INT,
     bp, count_send[menum], MPI_INT, 0, MPI_COMM_WORLD);

                //somma degli elementi dei sottarray e stamp
    a dei risultati
    for(i = 0; i < count_send[menum]; i++)
                {
                    cp[i] = ap[i] + bp[i];
                    printf("%d+%d=%d from p%d\t", ap[i], bp[
    i], cp[i], menum);

                }

        //fine dell'ambiente parallelo
        MPI_Finalize();


        return 0;
    }
```

Algoritmo Sequenziale

```c
#include <stdio.h>
#include <stdlib.h>
int main ()
{
    //dichiarazione degli array
    int * a;
    int * b;
    int * c;
    int i, n;

      //all'utente viene fatto scegliere il size e gli elementi degli array
      printf("inserire il size degli array\n");
      scanf ("%d", &n);
      a = (int *) malloc(sizeof(int)*n);
      b = (int *) malloc(sizeof(int)*n);
      c = (int *) malloc(sizeof(int)*n);
       printf("inserire i %d elementi di a\n", n);
      for(i=0; i<n; i++)
          scanf ("%d", &a[i]);

      printf("inserire i %d elementi di b\n", n);
      for(i=0; i<n; i++)
          scanf ("%d", &b[i]);

      /*ciclo for che scorre tutti gli elementi dei due array
      e li somma nell'array c, poi stampa gli elementi di c*/
      for(i = 0; i < n; i++)
      {
          c[i] = a[i] + b[i];
          printf("%d\t", c[i]);
      }

      return 0;

  }
```