Università degli Studi di Napoli
"Parthenope"

Corso di Calcolo Parallelo e Distribuito

Progetto Somma di N numeri

II Strategia

Virginia Bellino
Matr. 108/1570

Indice

INDIVIDUAZIONE ED ANALISI DEL PROBLEMA

DEFINIZIONE DEL PROBLEMA

Sviluppare un algoritmo per il calcolo della somma di N numeri in ambiente di calcolo parallelo su *architettura MIMD a memoria distribuita*, che utilizzi la libreria MPI.

Per la comunicazione dei risultati finali, tale algoritmo utilizzerà la 2° strategia, la quale prevede che, al termine della computazione, il risultato finale sia presente in un unico processore prestabilito.

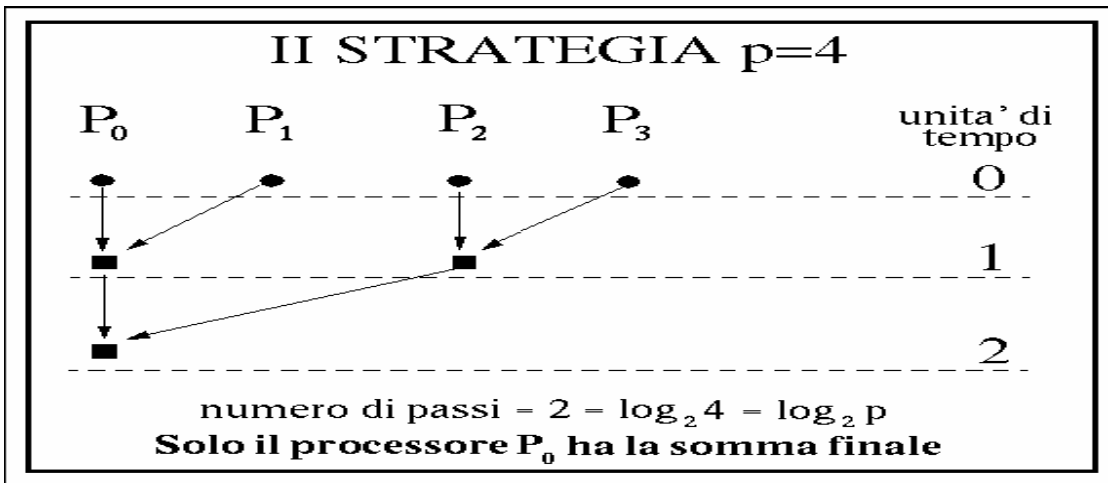
DESCRIZIONE DELL'ALGORITMO

- **Caratteristiche generali**

L'algoritmo implementa la seconda strategia di calcolo per una somma parallela.

Tale strategia prevede che ciascun processore calcoli inizialmente la sua somma parziale, poi, coppie distinte di processori comunicano tra loro le somme calcolate. In ciascuna coppia, un processore invia all'altro la sua somma parziale. Alla fine il risultato si troverà in un unico processore predefinito.

Ad esempio, supponendo di avere a disposizione p = 4 processori, l'applicazione della seconda strategia può essere schematizzata nel modo seguente:



Come si può notare, nel primo passo i processori P1 e P3 inviano le loro somme parziali rispettivamente a P0e a P2. Nel secondo passo invece, sarà soltanto il processore P0 che avrà il risultato finale.

Rispetto alla prima strategia, questo algoritmo ha sicuramente il vantaggio di *dimezzare i passi temporali per giungere al risultato* e di fare in modo che, in ciascun passo, *coppie distinte di processori operino concorrentemente*.
Tuttavia, l'indubbio svantaggio è quello di avere uno *sbilanciamento nel carico di lavoro dei processori*, poiché risulta che ad ogni passo parte dei processori rimane inattiva, e alla fine è attivo soltanto il processore che contiene il risultato.

Rispetto alla terza strategia invece, le differenze sono praticamente nulle, tranne per il fatto che, alla fine, nella terza strategia il risultato si trova in tutti i processori.

- **Codice**

L'algoritmo inizializza l'ambiente MPI usando le apposite subroutine.

Vi è poi la fase di lettura e distribuzione dei dati in cui, se ci si trova nel processore P0, prima viene richiesto di inserire da tastiera il numero di addendi che si desidera sommare, poi il processore P0 crea un vettore vuoto di grandezza pari al numero di elementi da sommare.

Dopo tali operazioni, il processore P0 manda un messaggio a tutti i processori del comunicatore per informarli su quanti numeri saranno sommati, usando la funzione MPI_Broadcast:

```
/*invio del valore di n a tutti i processori appartenenti a MPI_COMM_WORLD*/
MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
```

Successivamente, viene calcolato il numero di addendi da assegnare a ciascun processore e viene allocato il vettore che sarà utilizzato per le somme locali sui processori.

La presenza di un ciclo **if** ci pone poi di fronte a quanto segue:

1. se l'identificativo di processore corrisponde a quello del processore **P0,** allora viene inizializzata la generazione random degli addendi utilizzando l'ora attuale del sistema, e poi viene creato in modalità random il vettore di N addendi da sommare nel seguente modo:

```
/*Inizializza la generazione random degli addendi utilizzando.. */
 /*..l'ora attuale del sistema */
    srand((unsigned int) time(0));

    for(i=0; i<n; i++)
    {
            /*creazione del vettore contenenti addendi generati a random*/
            *(vett+i)=( int )rand( )%50+1;
    }
```

L'utilizzo della generazione random di addendi rende più agevole testare l'algoritmo con quantità elevate di numeri, che risulterebbe oltremodo faticoso inserire manualmente.

NOTA: a titolo puramente di controllo è stato inserito un ciclo for per la stampa degli elementi da sommare subito dopo la generazione random dei medesimi, che consente però la visualizzazione solo se non si superano i 100 valori.

2. Successivamente alla generazione random degli elementi vi è un ciclo **for** con cui P0 invia i dati parziali prestabiliti agli altri processori per il calcolo usando la funzione **SEND** di MPI;

3. se l'identificativo di processore non corrisponde a P0, allora occorre ricevere i dati spediti dal processore P0, e quindi….

```
/*SE non siamo il processore P0 riceviamo i dati trasmessi dal processore P0*/
    else

    {
    /* tag è uguale numero di processore */
    tag = menum;

    /* fase di ricezione */
    MPI_Recv(vett_loc,nloc,MPI_INT,0,tag,MPI_COMM_WORLD,&info);

    }/* end else */
```

La fase di lettura e distribuzione termina dunque con l'invio dei dati per effettuare le somme parziali a ciascun processore, e la ricezione dei medesimi, tramite l'utilizzo delle funzioni MPI di **send** e **recv.**

Nella **fase di calcolo**, dopo aver sincronizzato tutti i processori del comunicatore con una chiamata a **MPI_Barrier,** e aver calcolato il tempo di inizio con la

chiamata a **MPI_Wtime**, vengono calcolate con un ciclo **for** le somme parziali su ciascun processore.

Viene poi stabilito quanti passi bisognerà eseguire per giungere al risultato, memorizzando tali passi (che sono potenze di 2) nel vettore *potenze*.

Successivamente (fase di comunicazione tra i processori), viene calcolato e memorizzato nella variabile **sendTo** l'identificativo del processore a cui spedire la propria somma parziale.

Se l'identificativo corrisponde invece a quello del processore P0 (processore in cui alla fine avremo il risultato finale), allora vengono ricevuti i dati dal processore il cui id è memorizzato nella variabile **recvBy** e poi viene calcolato il risultato finale nel seguente modo:

```
else if (r = = 0) // se sono il processore P0
{
        recvBy = menum + potenze [i];
        tag = menum;
        /*ricezione del risultato della somma parziale di recvBy*/
        MPI_Recv(&tmp,1,MPI_INT,recvBy,tag,MPI_COMM_WORLD,&info);
        /*calcolo della somma parziale al passo i */
        sommaloc = sommaloc + tmp;
} //end else
```

Una successiva sincronizzazione dei processori con MPI_Barrier precede il calcolo del tempo finale.

```
MPI_Barrier (MPI_COMM_WORLD);
T_fine=MPI_Wtime()-T_inizio; // calcolo del tempo di fine
```

L'operazione di riduzione implementata con la chiamata a

```
MPI_Reduce(&T_fine,&T_max,1,MPI_DOUBLE,MPI_MAX,0,MPI_COMM_WORLD);
```

consente di ottenere il tempo di esecuzione per il calcolo della somma.

Chiude l'algoritmo la stampa del risultato finale da parte del processore P0 che, si ribadisce, è l'unico a detenere il medesimo al termine della computazione.

ANALISI DEL SOFTWARE

INDICAZIONI DI UTILIZZO PER L'UTENTE

- ***Compilazione ed esecuzione***

Per compilare il programma sorgente occorre digitare sul prompt dei comandi la seguente istruzione:

```
$ mpicc progetto_somma_vb.c
```

Per eseguire il programma occorre digitare sul prompt dei comandi la seguente istruzione:

```
$ mpirun –np x a.out
```

Note:

mpicc è l'istruzione che consente di compilare il programma sorgente del tipo **nomesorgente.c**

mpirun -np è l'istruzione che consente di eseguire il programma

a.out indica generalmente il nome dell'eseguibile

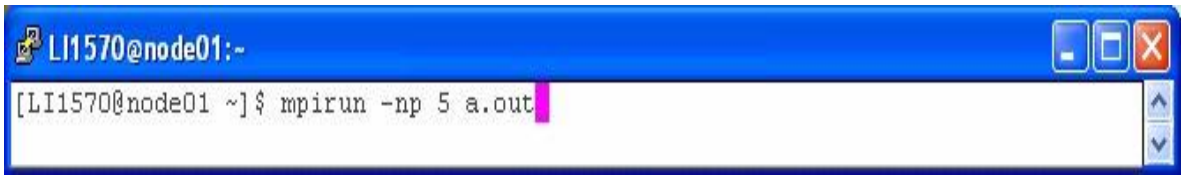
ATTENZIONE:

per applicare la strategia prevista dall'algoritmo occorre fornire in input al programma un numero di processori pari ad una potenza di 2, ovvero con *x* uguale a 1,2,4,8.

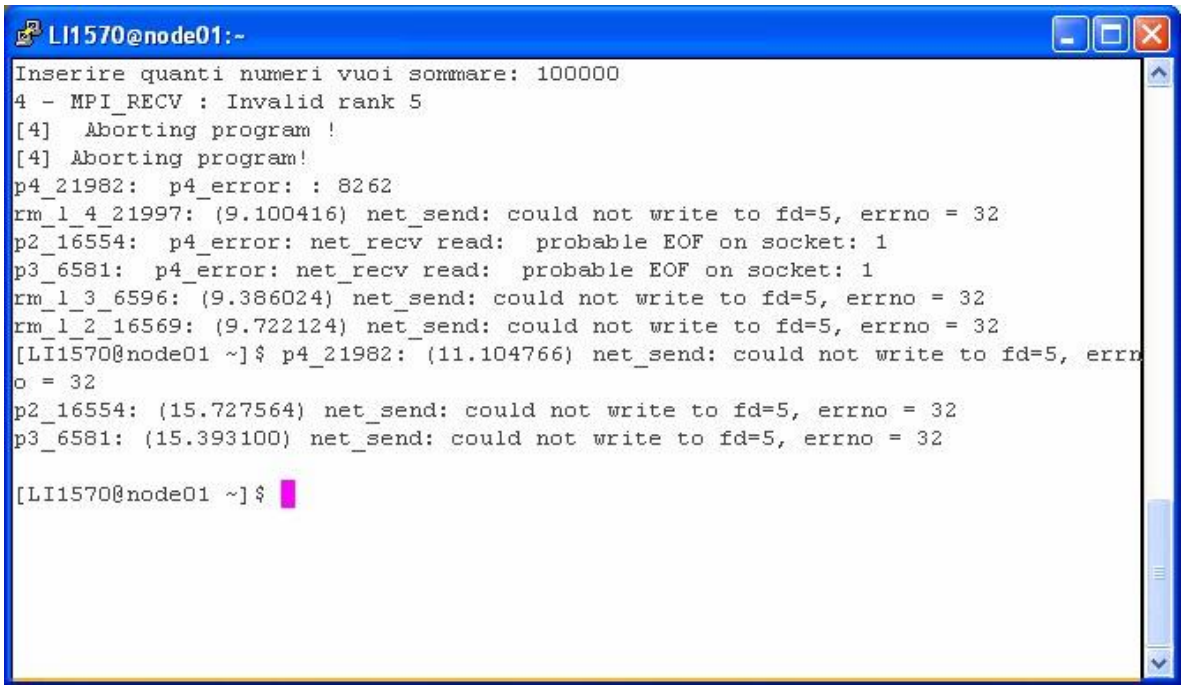
Valori alternativi danno luogo a messaggi di errore.

Esempio:

inserendo un numero di processori che non è una potenza di 2, ad esempio 5….



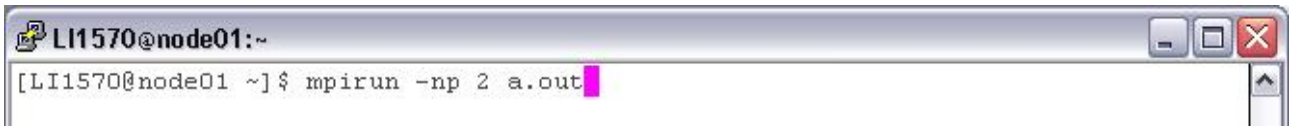
…..si ottiene il seguente messaggio di errore :



- ***Esempi d'uso***

In fase di esecuzione, occorre fornire in input al programma il numero di processori che si desidera utilizzare.

Esempio:

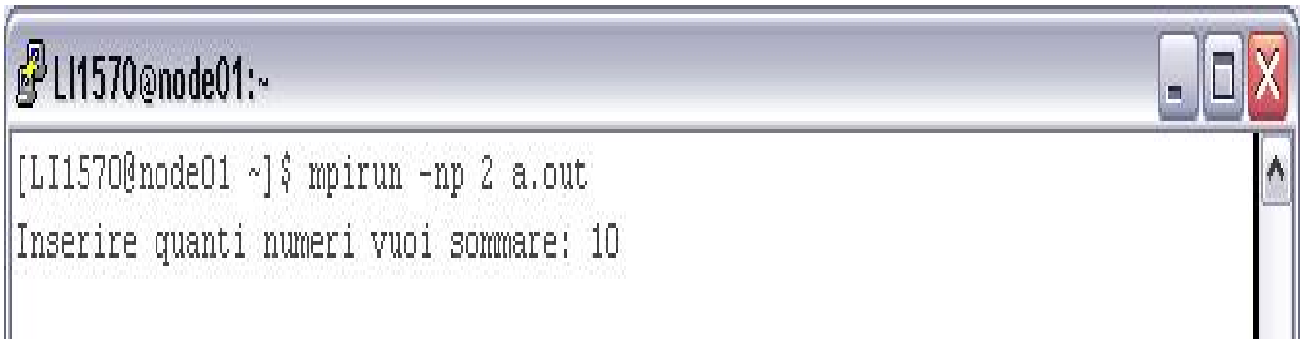


2 è, nell'esempio, il numero di processori da utilizzare.
a.out è invece il nome dell'eseguibile.

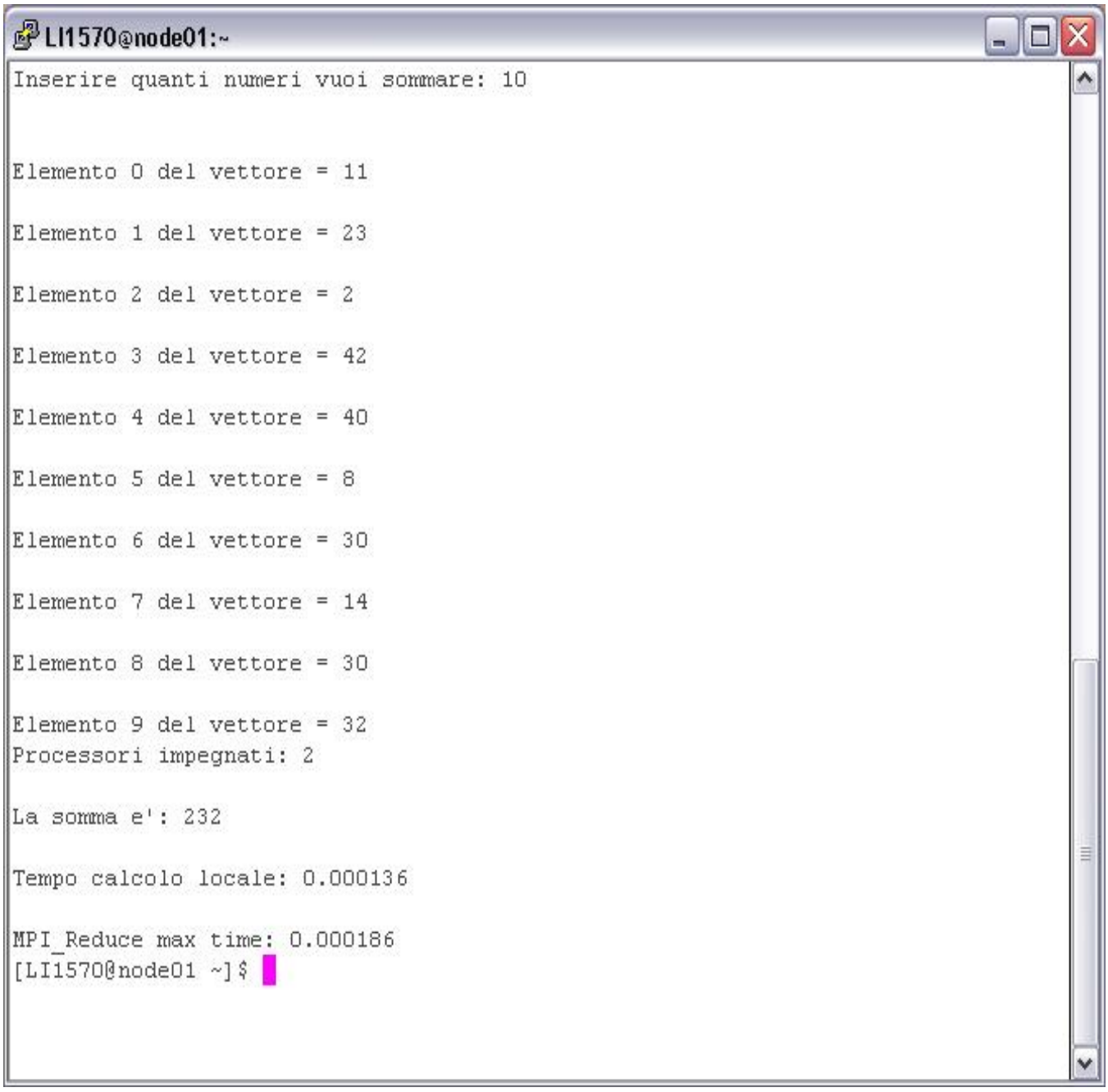
Successivamente, viene richiesto all'utente di inserire il numero di elementi che si desidera sommare.

Esempio 1:

in questo esempio, il numero di elementi da sommare è inferiore a 100, e dunque l'utente riceve in output anche la stampa dei medesimi

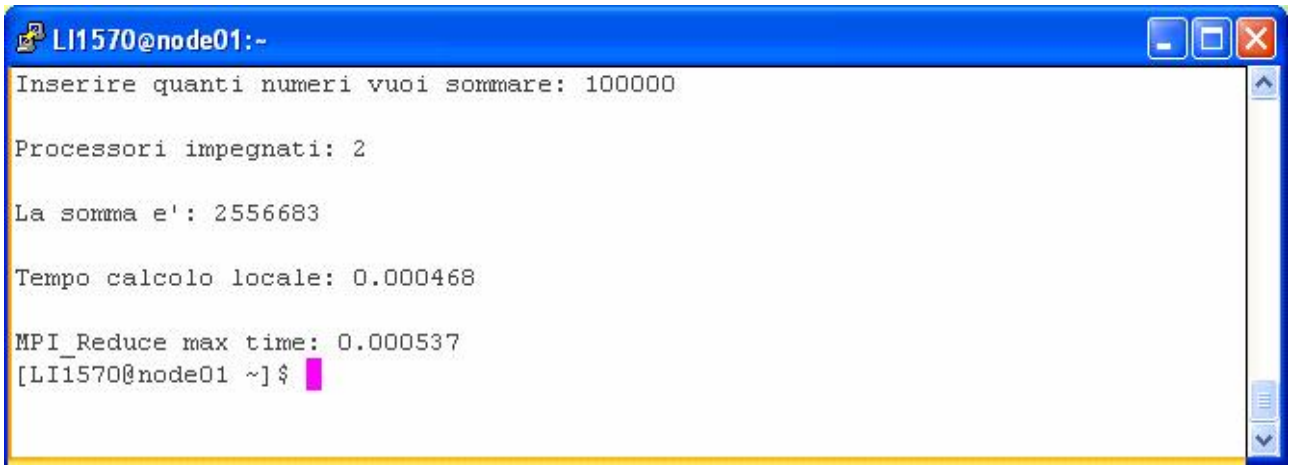


L'utente riceverà in output quanto segue:



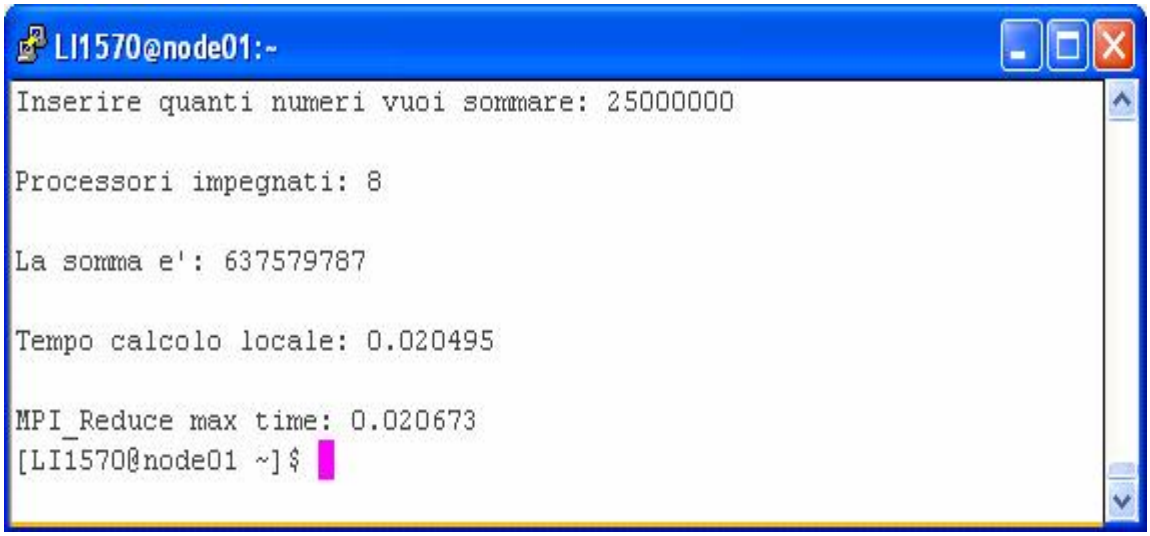
Esempio 2:

il numero di processori resta invariato (p = 2) ma aumenta la dimensione dell'input. Scompare per questo motivo dall'output la stampa a video del vettore di elementi da sommare



Esempio 3:

questa volta cambiamo sia la dimensione dell'input che il numero di processori impiegati. L'output ottenuto risulta identico, nella forma, al precedente esempio



FUNZIONI AUSILIARIE UTILIZZATE

Il software descritto, pur componendosi di un'unica routine che effettua le operazioni, 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.

Funzioni per inizializzare l'ambiente MPI

- **MPI_Init(&argc,&argv);**
  argc e *argv* sono gli argomenti del main

- **MPI_Comm_rank(MPI_COMM_WORLD, &menum);**
  MPI_COMM_WORLD (input): identificativo del comunicatore entro cui avvengono le comunicazioni

  menum (output): identificativo di processore nel gruppo del comunicatore specificato

- **MPI_Comm_size(MPI_COMM_WORLD, &nproc);**
  MPI_COMM_WORLD (input): nome del comunicatore entro cui avvengono le comunicazioni

  nproc (output): numero di processori nel gruppo del comunicatore specificato

Funzioni di comunicazione collettiva in ambiente MPI

- **MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);**
  comunicazione di un messaggio a tutti i processori appartenenti al comunicatore specificato.

  I parametri sono:

  n: indirizzo dei dati da spedire
  1: numero dei dati da spedire
  MPI_INT: tipo dei dati da spedire
  0 : identificativo del processore che spedisce a tutti

MPI_COMM_WORLD : identificativo del comunicatore entro cui avvengono le comunicazioni

Funzioni di comunicazione bloccante in ambiente MPI

- **MPI_Send(vett+ind,nloc,MPI_INT,i,tag,MPI_COMM_WORLD);**
  spedizione di dati

  I parametri sono:
  vett+ind (input): indirizzo del dato da spedire
  nloc (input): numero dei dati da spedire
  MPI_INT (input): tipo del dato inviato
  i (input): identificativo del processore destinatario
  tag (input): identificativo del messaggio inviato
  MPI_COMM_WORLD (input): comunicatore usato per l'invio del messaggio

- **MPI_Recv(vett_loc,nloc,MPI_INT,0,tag,MPI_COMM_WORLD,&info);**
  ricezione di dati

  I parametri sono:
  vett_loc: indirizzo del dato su cui ricevere
  nloc : numero dei dati da ricevere
  MPI_INT : tipo dei dati da ricevere
  0: identificativo del processore da cui ricevere
  tag (input): identificativo del messaggio
  MPI_COMM_WORLD (input): comunicatore usato per la ricezione del messaggio
  info: vettore che contiene informazioni sulla ricezione del messaggio

Funzione di sincronizzazione MPI

- **MPI_Barrier(MPI_COMM_WORLD);**
  La funzione fornisce un meccanismo sincronizzante per tutti i processori del comunicatore MPI_COMM_WORLD

- **MPI_Wtime( )**
  Tale funzione restituisce un tempo in secondi

Funzioni per operazioni collettive in ambiente MPI

- **MPI_Reduce(&T_fine,&T_max,1,MPI_DOUBLE,MPI_MAX,0,MPI_COMM_WORLD);**

  T_fine : indirizzo dei dati su cui effettuare l'operazione
  T_max : indirizzo del dato contenente il risultato
  1 : numero dei dati su cui effettuare l'operazione
  MPI_DOUBLE : tipo degli elementi da spedire
  MPI_MAX : operazione effettuata
  0: identificativo del processore che conterrà il risultato
  MPI_COMM_WORLD: identificativo del comunicatore

Funzione di chiusura ambiente MPI

- **MPI_Finalize( );**

La funzione determina la fine di un programma MPI. Dopo di essa non si può più chiamare nessuna altra routine MPI.

ANALISI DEI TEMPI

Introduzione

Osserviamo ora il nostro algoritmo analizzando in dettaglio alcune caratteristiche atte a valutare le prestazioni di un software parallelo.

Esse consentiranno all'utente di capire in quale situazione è più opportuno utilizzare l'algoritmo e quando invece il suo utilizzo non reca alcun palese vantaggio.

Tali caratteristiche sono:

- *Tempo di esecuzione utilizzando un numero p>1 di processori.*
  Generalmente indicheremo tale parametro con il simbolo **T(p)**

- ***Speed – up***
  riduzione del tempo di esecuzione rispetto all'utilizzo di un solo processore, utilizzando invece p processori.
  In simboli ……

  $$S(p) = T(1) / T(p)$$

  Il valore dello speed – up ideale dovrebbe essere pari al numero p dei processori, perciò l'algoritmo parallelo risulta migliore quanto più S(p) è prossimo a p.

- **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 S(p) lo speed - up ad esso relativi.
  Si definisce efficienza il parametro……

  $$E(p) = S(p) / p$$

  Essa fornisce *un'indicazione di quanto sia stato usato il parallelismo nel calcolatore.*
  Idealmente, dovremmo avere che:
  $$E(p) = 1$$
  e quindi l'algoritmo parallelo risulta migliore quanto più E(p) è vicina ad 1.

Valutazione dei tempi, dello speed-up e dell'efficienza

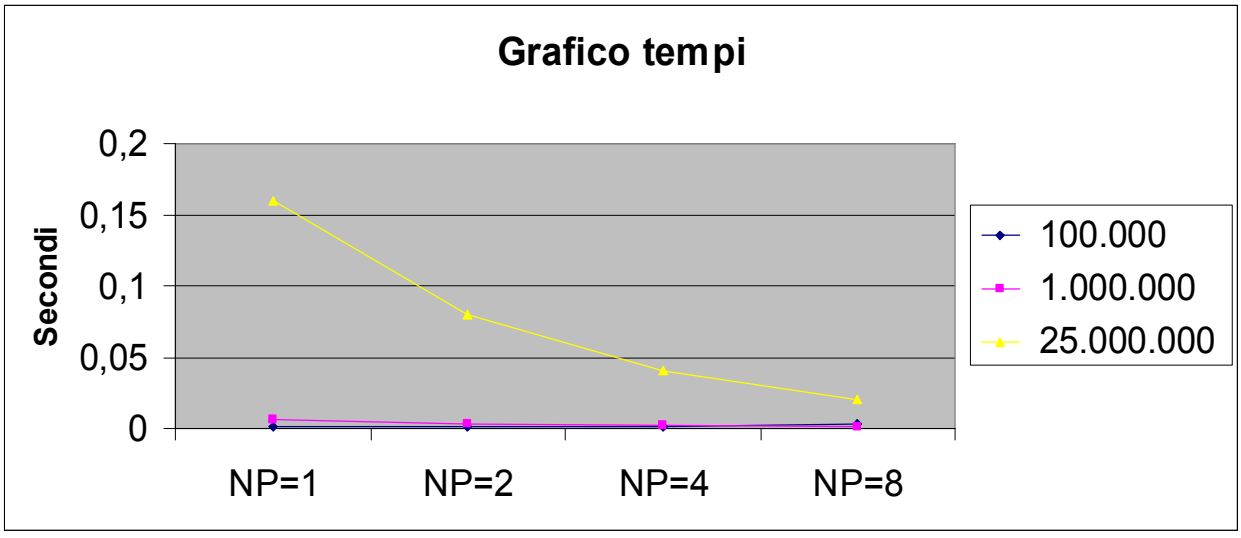
Le tabelle che seguono mostrano i dati raccolti analizzando ciascuna delle caratteristiche sopraelencate.

La prima riga di ogni tabella contiene il *valore dei dati forniti in input*, mentre la prima colonna elenca il *numero di processori impiegati nella computazione*.

Accanto a ciascuna tabella viene mostrato il relativo grafico che evidenzia l'andamento dei valori esaminati.

Tabella dei tempi

	100.000	1.000.000	25.000.000
NP=1	0,00068	0,006385	0,159291
NP=2	0,000537	0,003501	0,079856
NP=4	0,000605	0,002091	0,04041
NP=8	0,002801	0,001408	0,020673



Come si può notare osservando il grafico, per un esiguo numero di valori da sommare, aumentando il numero di processori utilizzati nel calcolo, si registra addirittura un lieve *peggioramento dei tempi di elaborazione*.

Per registrare un sensibile miglioramento dei tempi occorre aumentare il numero dei processori utilizzati e la quantità dei valori forniti in input.

Tabella dello speed - up

	100.000	1.000.000	25.000.000
Sp 1	1	1	1
Sp 2	1,26629423	1,823764639	1,99472801
Sp 4	1,12396694	3,053562889	3,941870824
Sp 8	0,24277044	4,534801136	7,705267741

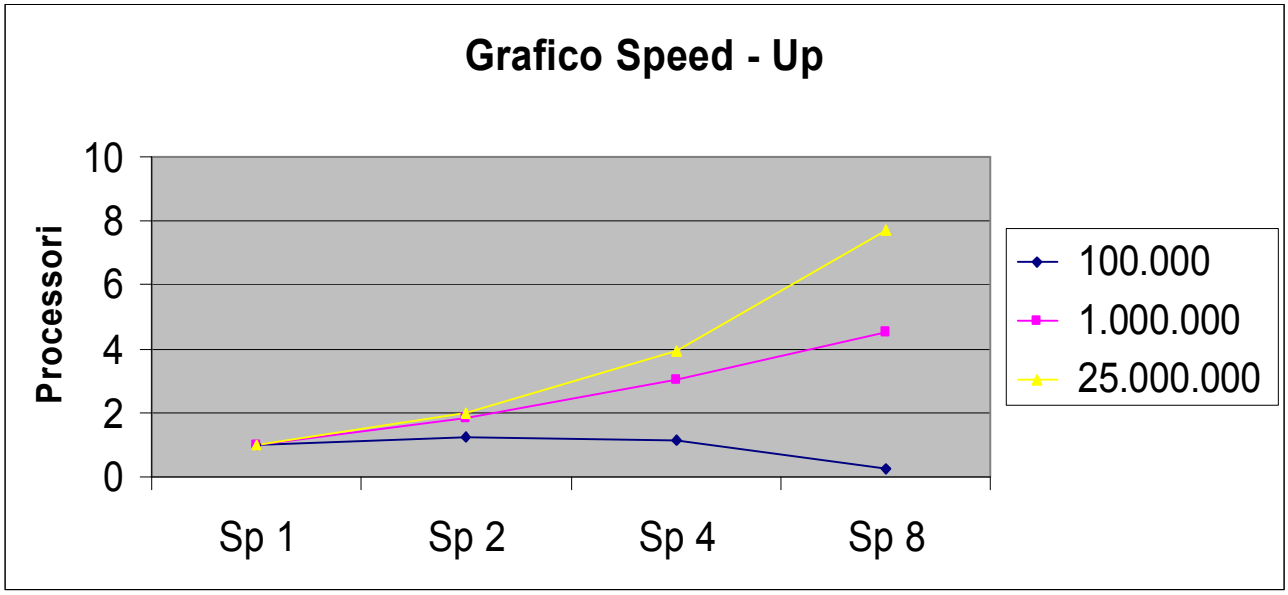
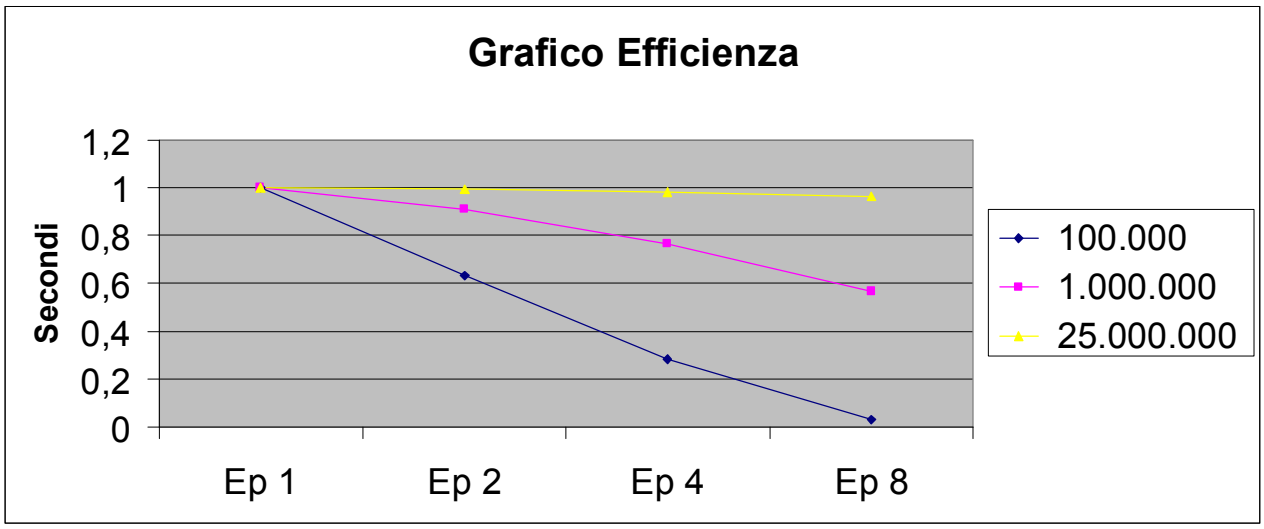


Tabella dell'efficienza

	100.000	1.000.000	25.000.000
Ep 1	1	1	1
Ep 2	0,63314711	0,911882319	0,997364005
Ep 4	0,28099174	0,763390722	0,985467706
Ep 8	0,0303463	0,566850142	0,963158468



Un'attenta analisi dei risultati ottenuti mostra che i migliori vantaggi dall'uso del software si ottengono quando le dimensioni dell'input sono molto elevate.

Con enormi quantità di dati infatti, i valori dello speed – up e dell'efficienza tendono ad avvicinarsi sempre più ai rispettivi valori ideali (si consulti al riguardo l'introduzione della presente sezione), qualunque sia il numero di processori impiegati nella computazione.

Ad esempio, con input pari a 25 milioni di dati, i valori dello speed up sono molto prossimi al valore ideale, specialmente per p = 2 processori, e la situazione migliora se l'utente aumenta la quantità dei valori in input.
La curva dell'efficienza invece, si assesta in prossimità del valore ideale qualunque sia il numero dei processori utilizzati.

Complessivamente, si può dunque ribadire un giudizio positivo sull'algoritmo, specialmente con elevate dimensioni di dati.

BIBLIOGRAFIA DI RIFERIMENTO

1. A.Murli – Lezioni di Calcolo Parallelo – Ed. Liguori

2. www.mat.uniroma1.it/centro-**calcolo**/HPC/materiale-corso/sliMPI.pdf
   (consultato per interessanti approfondimenti su MPI)

3. www.orebla.it/module.php?n=**c**_num_**casuali**
   (consultato per approfondimenti sulla funzione rand per generare numeri casuali)

4. Bellini – Guidi - LINGUAGGIO C/GUIDA ALLA PROGRAMMAZIONE – McGRAW HILL
   (per un utile ripasso del linguaggio C)

CODICE SORGENTE DEL PROGETTO

```
/*
 ***********************************************
 *    ALGORITMO PER IL CALCOLO PARALLELO       *
 *         DELLA SOMMA DI N NUMERI             *
 *              II STRATEGIA                   *
 *                                             *
 *                                             *
 *           VIRGINIA     BELLINO              *
 *              MATR. 108/1570                 *
 *                                             *
 ***********************************************
*/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <mpi.h>

int main (int argc, char **argv)
{
     /*dichiarazioni variabili*/
     int menum,nproc,tag;
     int n,nloc,i,somma,resto,nlocgen;
     int ind,p,r,sendTo,recvBy,tmp;
     int *potenze,*vett,*vett_loc,passi=0;
     int sommaloc=0;

     /*variabili per verificare il livello delle prestazioni*/
     double T_inizio,T_fine,T_max;

     MPI_Status info;

     /*Inizializzazione dell'ambiente di calcolo MPI*/
     MPI_Init(&argc,&argv);
     /*assegnazione IdProcessore a menum*/
     MPI_Comm_rank(MPI_COMM_WORLD, &menum);
     /*assegna numero processori a nproc*/
     MPI_Comm_size(MPI_COMM_WORLD, &nproc);

     /*seleziona il processore con ID=0*/
     if (menum==0)
     {
          system("cls"); // pulisci schermo
          printf("Inserire quanti numeri vuoi sommare: ");
          scanf("%d",&n);

          /*allocazione dinamica di un vettore di n elementi interi nel processore P0*/
          vett=(int*)calloc(n,sizeof(int));
     }

     /*invio del valore di n a tutti i processori appartenenti a MPI_COMM_WORLD*/
     MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
```

```
    /*calcolo del numero di addendi da assegnare a ciascun processore*/
    nlocgen=n/nproc;

    /*Calcolo del resto.Stabiliamo se ci sono altri addendi da ripartire*/
    /*tra i processori*/
    resto=n%nproc;

    /* Se ci sono addendi in piu, il processore di identificativo menum */
    /* incrementa il numero di addendi ricevuti .... */
    if(menum<resto)
    {
        nloc=nlocgen+1;
    }

    /*...altrimenti gli addendi restano quelli assegnati in precedenza */
    else
    {
        nloc=nlocgen;
    }


    /*allocazione dinamica del vettore per le somme parziali */

    vett_loc=(int*)calloc(nloc, sizeof(int));




/*@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@*/


    if (menum==0)
    {
        /*Inizializza la generazione random degli addendi utilizzando.. */
        /*..l'ora attuale del sistema*/
        srand((unsigned int) time(0));



        for(i=0; i<n; i++)
        {
            /*creazione del vettore contenenti addendi generati a random*/
            *(vett+i)=(int)rand()%50+1;
        }

        // Stampa del vettore che contiene i dati da sommare (flag di controllo)
        // Se sono superiori a 100, però, salta la visualizzazione
        if (n<100)
        {
            for (i=0; i<n; i++)
            {
                printf("\n\nElemento %d del vettore %d =",i,*(vett+i));
            }
        }
```

```
        for(i=0;i<nloc;i++)
        {
            /*creazione del vettore locale per il processore P0*/
            *(vett_loc+i)=*(vett+i);
        }



        ind=nloc;


        /*ciclo for con cui P0 invia i dati parziali prestabiliti .. */
        /*...agli altri processori per il calcolo*/
        for(i=1; i<nproc; i++)
        {
            tag=i;  /* assegnazione id del messaggio*/
            /*SE ci sono addendi in sovrannumero da ripartire tra i processori ...*/
            if (i<resto)
            {
                /*il processore P0 gli invia il corrispondete vettore locale considerando un addendo in piu'*/
                MPI_Send(vett+ind,nloc,MPI_INT,i,tag,MPI_COMM_WORLD);
                ind=ind+nloc;
            }  // END THEN

            else
            {
                /*il processore P0 gli invia il corrispondete vettore locale*/
                MPI_Send(vett+ind,nlocgen,MPI_INT,i,tag,MPI_COMM_WORLD);
                ind=ind+nlocgen;
            }// end else
        }//end for
    }// end THEN


    /*SE non siamo il processore P0 riceviamo i dati trasmessi dal processore P0*/
    else

    {
        // tag è uguale numero di processore
        tag=menum;

        /*fase di ricezione*/
        MPI_Recv(vett_loc,nloc,MPI_INT,0,tag,MPI_COMM_WORLD,&info);
    }// end else

    /*@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@*/
    /* @@@@@@   FINE FASE DI LETTURA E DISTRIBUZIONE DEI DATI   @@@@@@*/
    /*@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@*/



    /*@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@*/
    /* @@@@@@   INIZIO FASE DI CALCOLO   @@@@@@*/
    /*@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@*/
```

```
/*fornisce un meccanismo sincronizzante per tutti i processori del MPI_COMM_WORLD*/
/*ogni processore si ferma aspettando che l'istruzione venga eseguita dal resto dei processori*/
MPI_Barrier(MPI_COMM_WORLD);

T_inizio=MPI_Wtime(); //inizio del cronometro per il calcolo del tempo di inizio

for(i=0;i<nloc;i++)
{
     /*ogni processore effettua la somma parziale*/
     sommaloc=sommaloc+*(vett_loc+i);
}

// p è il numero di processori
p=nproc;

while(p!=1)
{
     /*shifta di un bit a destra (calcolo del logaritmo in base 2)*/
     p=p>>1;
     /*determina il numero dei passi per sapere quante spedizioni di somme parziali bisogna fare*/
     passi++;
}

/*allocazione dinamica del vettore potenze*/
potenze=(int*)calloc(passi+1,sizeof(int));

for(i=0;i<=passi;i++)
{
     /*creazione del vettore potenze di elementi passi+1, contenente le potenze di 2*/
     potenze[i]=p<<i;
}

/* Fase di comunicazione tra processori*/
/* viene mandata la somma parziale con i dati da sommare */
// finché ci sono ancora dei passi da eseguire ...
for(i=0;i<passi;i++)
{
     // ... calcolo identificativo del processore
     r=menum%potenze[i+1];

     // Se l'identificativo non corrisponde a quello del processore P0...
     if(r==potenze[i])
     {
          // calcolo dell'identificativo del processore a cui spedire la somma locale
          sendTo=menum-potenze[i];
          tag=sendTo;
          /*invio del risultato della propria somma parziale a sendTo*/
          MPI_Send(&sommaloc,1,MPI_INT,sendTo,tag,MPI_COMM_WORLD);
     }//end then
     else if(r==0)  // se sono il processore P0
     {
          recvBy=menum+potenze[i];
          tag=menum;
          /*ricezione del risultato della somma parziale di recvBy*/
```

```
            MPI_Recv(&tmp,1,MPI_INT,recvBy,tag,MPI_COMM_WORLD,&info);
            /*calcolo della somma parziale al passo i*/
            sommaloc=sommaloc+tmp;
        }//end else
    }// end for

    MPI_Barrier(MPI_COMM_WORLD);
    T_fine=MPI_Wtime()-T_inizio;  // calcolo del tempo di fine

    /*determinazione del tempo di esecuzione per il calcolo della somma*/
    MPI_Reduce(&T_fine,&T_max,1,MPI_DOUBLE,MPI_MAX,0,MPI_COMM_WORLD);

    if(menum==0)
    {
        /*stampa a video dei risultati finali*/
        printf("\nProcessori impegnati: %d\n", nproc);
        printf("\nLa somma e': %d\n", sommaloc);
        printf("\nTempo calcolo locale: %lf\n", T_fine);
        printf("\nMPI_Reduce max time: %f\n",T_max);
    }// end if

    /*routine chiusura ambiente MPI*/
    MPI_Finalize();
}// fine programma
```