Università degli Studi di Napoli
"Parthenope"

Corso di Calcolo Parallelo e Distribuito

Progetto Somma di N numeri

II Strategia

Giovanni Di Cecca
Matr. 108/1569

Indice

Definizione ed analisi del problema

Il software che si analizzerà di seguito ha lo scopo di sommare un certo numero di valori (generati casualmente e non superiore a 100.000.000 di valori) distribuendo il carico non più su un unico processore, bensì usando un numero di calcolatori pari o superiori a due (architetture di tipo MIMD).

L'infrastruttura usata per distribuire il carico di lavoro è quella del Message Passing Interface (MPI).

Descrizione dell'Algoritmo

Introduzione

La strategia usata per risolvere il problema della ***somma di n numeri*** è quella dell'albero binario: un numero di processori pari a 2^n che calcolano le somme parziali, fino a "consegnare" i dati ad un unico processore che contiene la somma finale.

La differenza sostanziale tra la strategia scelta e le altre, è quella che ha numero di scambi di messaggi pari a $log_2 n$.

Questo metodo è sicuramente più vantaggioso rispetto alla I Strategia dove lavora principalmente un solo processore che somma le somme parziali ricevute dalle altre macchine della rete, ma a differenza della III Strategia, nella quale il risultato finale è contenuto in tutti computer della rete che esegue il calcolo, nella II il risultato finale è contenuto in un solo processore: il master.

Come espresso precedentemente, il sistema di calcolo usato è quello del **Message Passinge Interface**, mediante l'uso del middleware dal centro di calcolo parallelo e supercomputer degli **Argonne National Laboratories**.

Descrizione dell'Algoritmo

Caratteristiche generali

L'algoritmo può essere suddiviso in quattro parti principali:

- Inizializzazione dell'ambiente di calcolo
- Inserimento dei dati
- Distribuzione dei dati
- Calcolo delle somme parziali e totale

Per poter meglio analizzare le performances dell'algoritmo al suo interno è stato inserito il sistema di controllo del tempo.

Analizzeremo nel dettaglio le parti indicate

- **Inizializzazione dell'ambiente di calcolo**

```
// Variabili per verificare il livello delle prestazioni
double T_inizio,T_fine,T_max;

// Contiene le informazioni sulla ricezione del messaggio
MPI_Status info;

// Inizializzazione dell'ambiente di calcolo MPI
MPI_Init(&argc,&argv);

// Identificativo del processore (contenuto in menum)
MPI_Comm_rank(MPI_COMM_WORLD, &menum);

// Verifica del numero di processori della rete di calcolo
MPI_Comm_size(MPI_COMM_WORLD, &nproc);
```

Alla fine di questa fase la rete sa che deve aspettarsi dei dati da calcolare

- **Inserimento dei dati**

Il processore P0, dopo aver inizializzato il vettore dinamicamente ed effettuato il broadcast alla rete viene effettuato l'inserimento dei dati con il metodo di generazione casuale Random, mediante la seguente funzione:

```
// Inizializza la generazione random degli addendi
// utilizzando... l'ora attuale del sistema
srand((unsigned int) time(0));
// Inserisi i dati meterialmente nel vettore
for(i=0; i<n; i++)
{
     //creazione del vettore contenenti
     //addendi generati a random
     *(vett+i)=(int)rand()%50+1;
}
```

Per controllare i dati immessi ho inserito una flag che legge i dati inseriti nel vettore

```
// Stampa del vettore che contiene i dati da sommare
// 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));
    }
}
```

- **Distribuzione dei dati**

Il processore P0 invia i dati ai vari processori della rete:

```
// 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 più
    MPI_Send(vett+ind,nloc,MPI_INT,i,tag,MPI_COMM_WORLD);
    ind=ind+nloc;
    }
    else
    {
        // il processore P0 gli invia il corrispondete
        // vettore locale
    MPI_Send(vett+ind,nlocgen,MPI_INT,i,tag,MPI_COMM_WORL
D);
        ind=ind+nlocgen;
    }
```

```
}//end for
// 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
```

- **Calcolo delle somme parziali e totali**

Dopo il primo passaggio, Il sistema calcola le somme parziali usando $\log_2 n$ processori.

Per fare ciò usiamo un sistema di sincronizzazione definito da MPI e poi calcoliamo il $\log_2 n$ dei processori

Per sincronizzare le operazioni usiamo la funzione `MPI_Barrier` che consente di far partire tutte le somme in modo contemporaneo.

Analogamente iniziamo anche a contare il tempo di esecuzione del programma.

Il codice è il seguente

```
    // Uso di MPI_Barrier per la fase di sincronizzazione
    // 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);
```

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

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

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

    // Fase di calcolo del logaritmo in base 2 di nproc
    // per selezionare
    // i processori che effettueranno le somme parziali
    while(p!=1)
    {
        // shifta di un bit a destra
        p=p>>1;

        // determina il numero dei passi per sapere quante
        // spedizioni di somme parziali bisogna fare
        passi++;
    }

    // allocazione dinamica del vettore pot che
    // calcola la potenza di 2^n
    pot=(int*)calloc(passi+1,sizeof(int));

    for(i=0;i<=passi;i++)
    {
```

```
                // creazione del vettore pot di elementi
                // passi+1, contenente le potenze di 2
                pot[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%pot[i+1];

                // Se l'identificativo non corrisponde a
                // quello del processore P0...
                if(r==pot[i])
                {
                        // calcolo dell'identificativo del
                        // processore a cui spedire la
                        // somma locale
                        inviaA=menum-pot[i];
                        tag=inviaA;
                        // invio del risultato della propria
                        // somma parziale a inviaA

    MPI_Send(&sumloc,1,MPI_INT,inviaA,tag,MPI_COMM_WORLD)
;
                }//end then
                else if(r==0) // se sono il processore P0
                {
                        ricevida=menum+pot[i];
```

```
                    tag=menum;
                    // ricezione del risultato della
                    // somma parziale di ricevida

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

            MPI_Barrier(MPI_COMM_WORLD);

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

            // Calcolo del tempo di esecuzione
            MPI_Reduce(&T_fine,&T_max,1,MPI_DOUBLE,MPI_MAX,0
,MPI_COMM_WORLD);
```

Compilazione ed esecuzione

Il programma è studiato per essere il più possibile indipendente dalla piattaforma usata (Linux o Windows).

- Compilazione sotto Linux

Se il programma viene compilato in un sistema Linux, la sintassi è la seguente

```
$ mpicc sommagdc.c <invio>
```

Successivamente sarà generato un file **a.out** eseguibile

- Esecuzione del programma sotto Linux

Per eseguire il programma si deve seguire la seguente sintassi:

```
$ mpirun –np x a.out <invio>
```

con *x* il numero di processori che si vuole usare.

A questo punto il programma in esecuzione chiede quanti dati devono essere calcolati.

- Compilazione sotto Microsoft Windows XP

Se il programma viene compilato in Windows la sintassi è la seguente

Aprire il **Prompt dei comandi**[1]:

```
C:\>gcc sommagdc.c -lmpich
```

Successivamente sarà generato un file **a.exe**

- Esecuzione del programma sotto Microsoft Windows XP[2]

Per eseguire il programma si deve seguire la seguente sintassi:

```
C:\> mpirun –np x a <invio>
```

con *x* il numero di processori che si vuole usare.

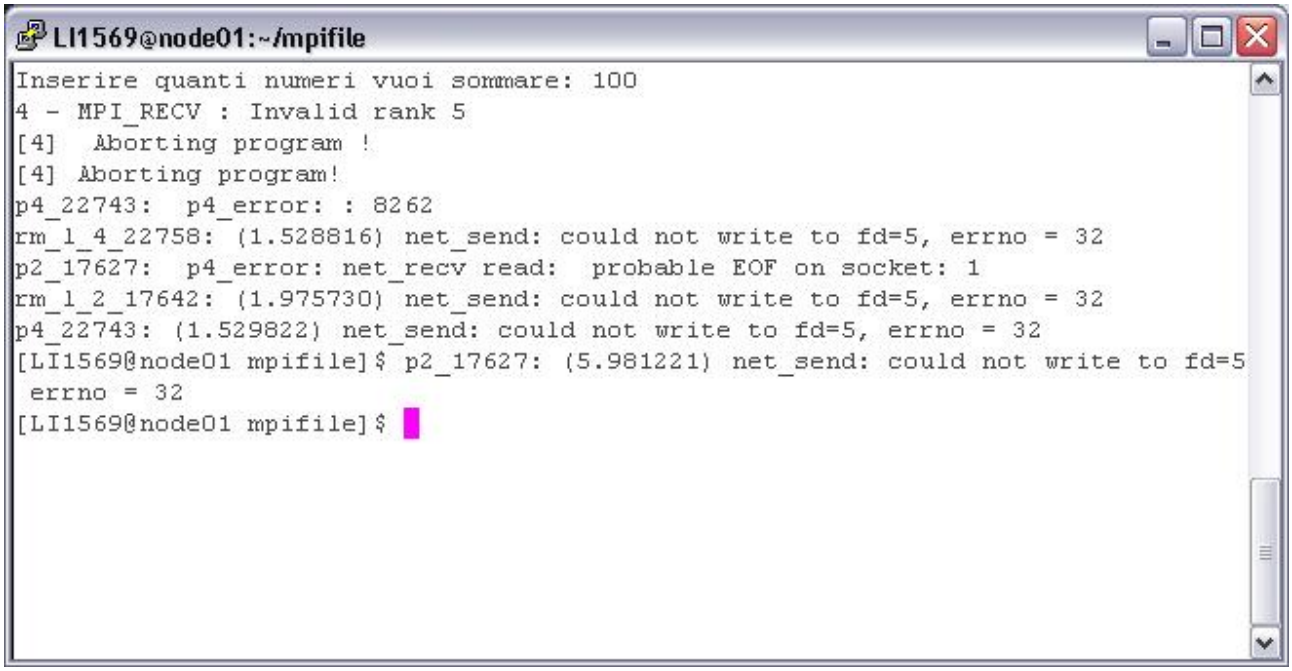
A questo punto il programma in esecuzione chiede quanti dati devono essere calcolati.

[1] Si presuppone che sia stato installato il compilatore MingW, le librerie MPICH NT e le **Variabili d'ambiente di Windows** contengano il path del compialtore e delle librerie

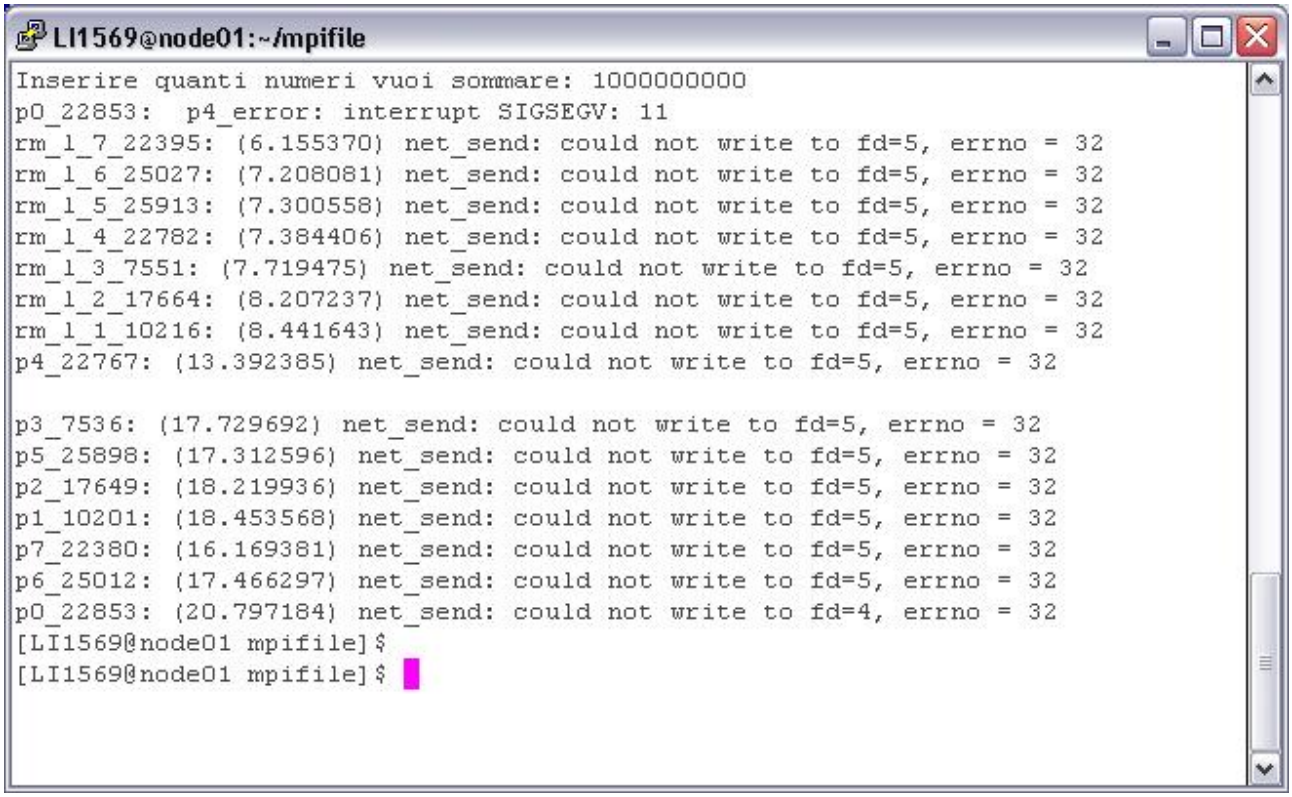
[2] Si presuppone che il sistema abbia identificato la presesenza di 1 o più computer e che sia stato configurato il programma **MPICH Configuration tool**.

Indicatori di errore

Il programma può usare un numero 2^n di processori (ad esempio 1,2,4,8), in caso contrario termina visualizzando gli errori che ha riscontrato



Se si supera i 10^8 valori (cioè considerando 1 miliardo di valori da sommare) il programma va in errore



Funzioni 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

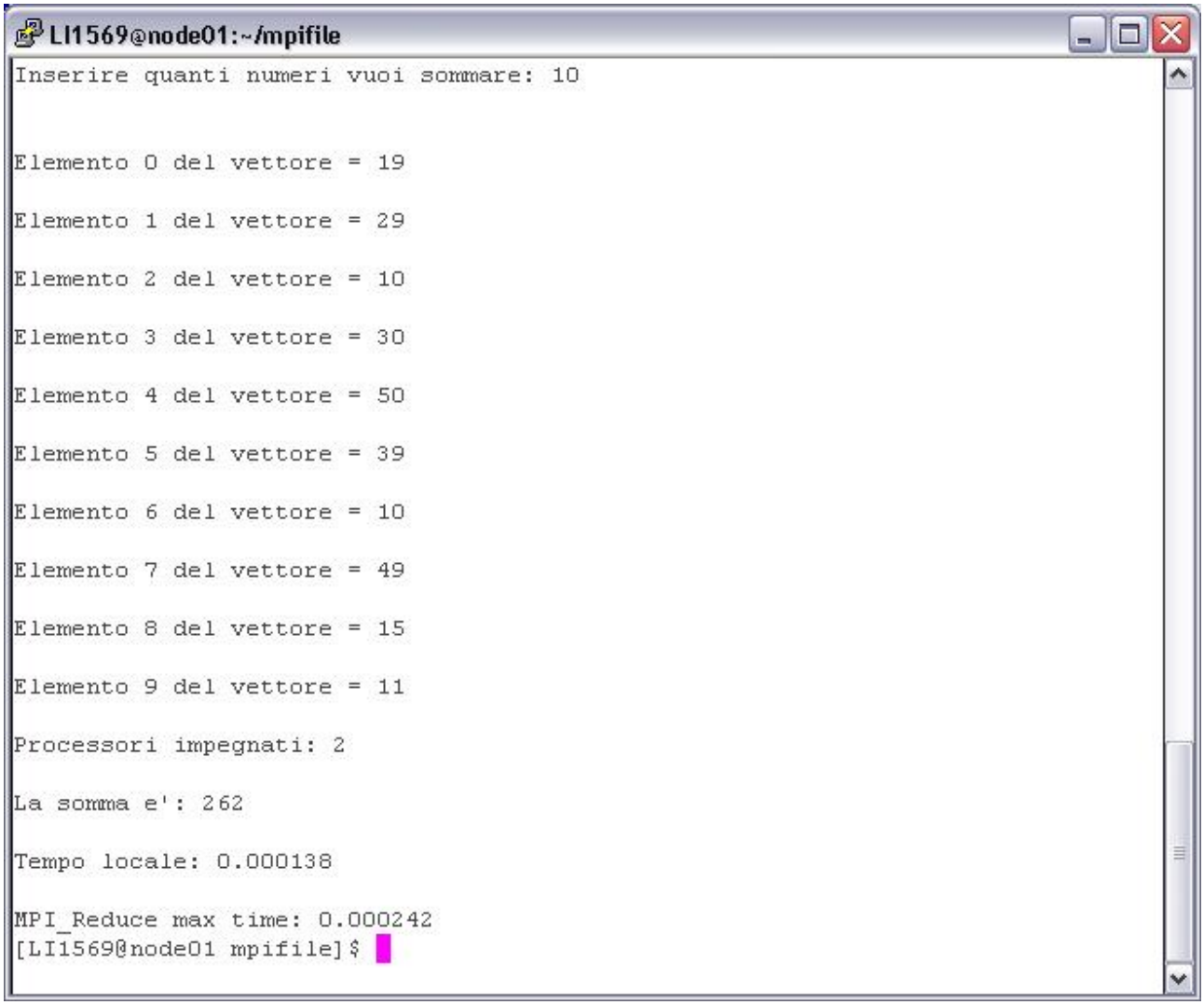
Funzione di chiusura ambiente MPI

- **MPI_Finalize();**

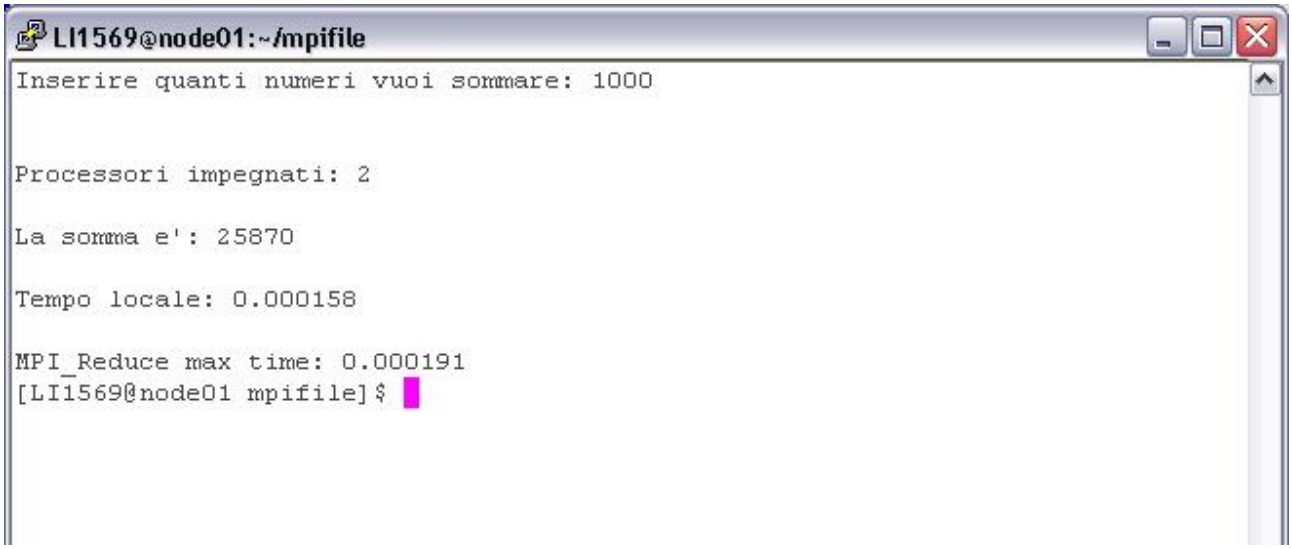
La funzione determina la fine di un programma MPI. Dopo di essa non si può più chiamare nessuna altra routine MPI.

Esempi d'uso

Lancio del programma su due processori (ovvero due macchine della rete)



Esempio d'uso con più di 100 valori



Analisi del Tempo

Introduzione

Ora osserveremo le prestazioni che ha il nostro algoritmo quando viene usato. Per fare ciò analizzeremo le seguenti caratteristiche:

- *Tempo di esecuzione utilizzando un numero p>1 processori*.
  Generalmente indicheremo tale parametro con il simbolo $\boldsymbol{T_p}$

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

  In simboli ……

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

  Il valore dello speedup 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……

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

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

Idealmente, dovremmo avere che:

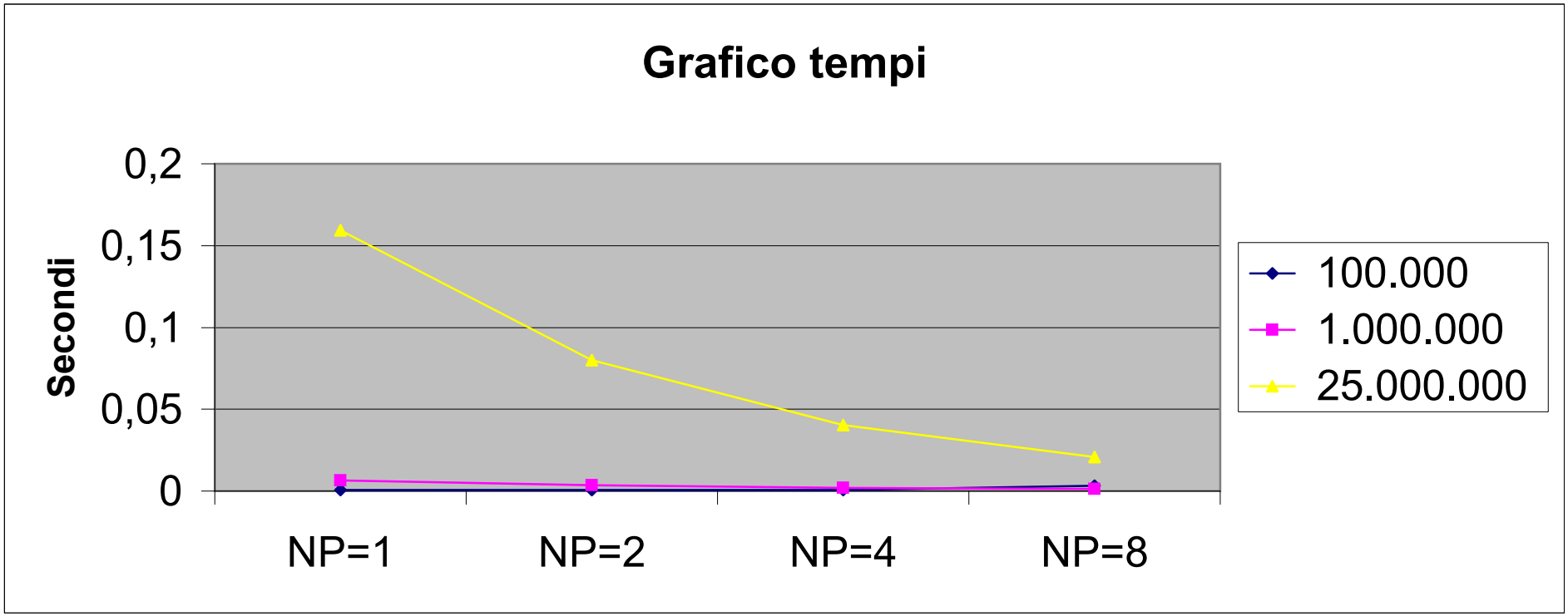
$$E_p = 1$$

e quindi l'algoritmo parallelo risulta migliore quanto più E_p è vicina ad 1.

Nelle prossime pagine analizzeremo i tempi delle elaborazioni, presi direttamente dal calcolo, ed in seguito analizzeremo lo speed – up e l'efficienza.

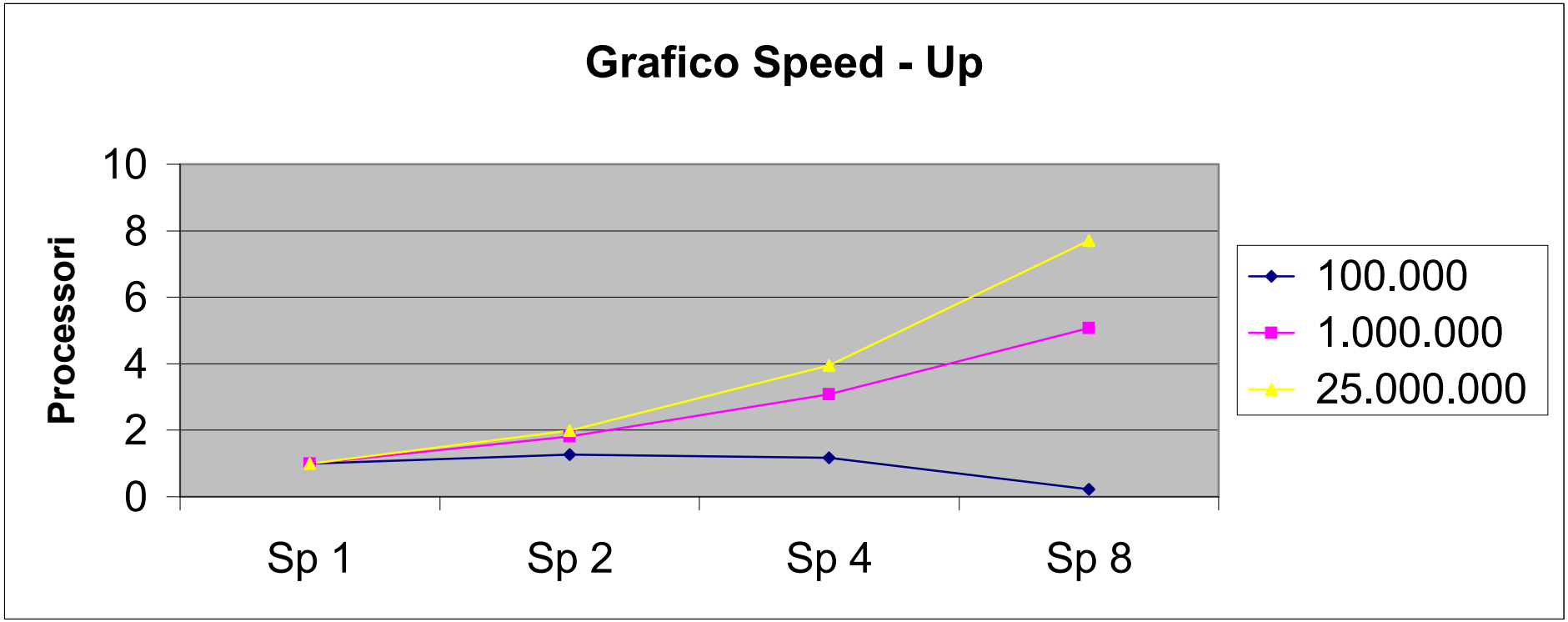
Tempi delle elaborazioni

	100.000	1.000.000	25.000.000
NP=1	0,00068	0,006385	0,159291
NP=2	0,000537	0,003517	0,079856
NP=4	0,000583	0,002073	0,04041
NP=8	0,003105	0,001259	0,020673



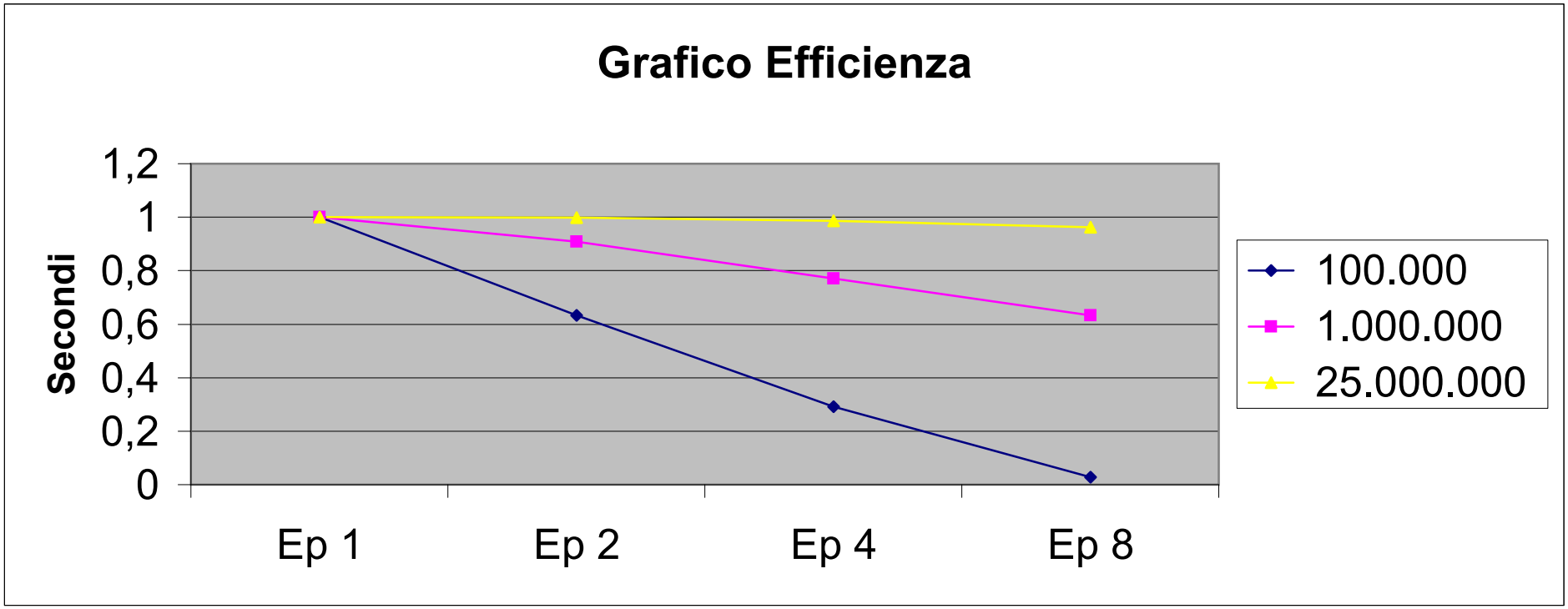
Speed - Up

	100.000	1.000.000	25.000.000
Sp 1	1	1	1
Sp 2	1,266294227	1,815467728	1,99472801
Sp 4	1,166380789	3,080077183	3,941870824
Sp 8	0,21900161	5,071485306	7,705267741



Efficienza

	100.000	1.000.000	25.000.000
Ep 1	1	1	1
Ep 2	0,633147114	0,907733864	0,997364005
Ep 4	0,291595197	0,770019296	0,985467706
Ep 8	0,027375201	0,633935663	0,963158468



Come possiamo osservare dai dati nella prima tabella, per piccole quantità di dati (sotto pari a 100.000 valori) più paralleliziamo i dati, più abbiamo un "peggioramento" delle prestazioni.

Quoquomodo, abbiamo che all'aumentare del numero di processori, per più grandi quantità di dati, il parallelismo gioca un ruolo fondamentale nel velocizzare il calcolo.

Per meglio evidenziare tale problema ho evidenziato in verde i tempi migliori ed in rosso quelli peggiori.

Nella seconda tabella, quello dello Speed – Up (il rapporto dell'esecuzione tempo del calcolo su un processore e del tempo del calcolo su più processori parallelo) conferma i dati dei tempi di esecuzione, cioè nel caso di 25.000.000 di valori sommati, il valore che esce fuori dal rapporto è quasi uguale al numero di processori impiegati (7,705 è prossimo a 8)

Nella terza tabella dell'efficienza, per questo tipo di algoritmo, notiamo come la tendenza al valore ideale 1 aumenta col numero di procssori.

Possiamo notare una piccola differenza nel caso di 25.000.000 che risulta essere "un po' meno efficente" per questo volume di dati, all'aumentare il numero dei processori.

Bibliografia

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

2. Slide del corso di Calcolo Parallelo e distribuito
   http://www.dma.unina.it/~murli/didattica/mat_didattico_nav_cp0708.html

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

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

5. Aitken Peter G., Jones Bradley J. - Programmare in C – Apogeo
   (consultato per ripasso di alcune funzioni)

Codice Sorgente

Di seguito viene proposto il codice sorgente del programma sin qui descritto, nella visualizzazione dell'editor di testo Notepad++ Portable per Windows

```
/*
 ************************************************
 Somma di n numeri calcolati su macchine tipo MIMD

 Progetto Realizzato da

 Giovanni Di Cecca
 Matr. 108 / 1569

 E-Mail: giovanni.dicecca@gmail.com

 Web Site: http://www.dicecca. net

 ************************************************
*/

// Parte preprocessore
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "mpi.h"

// Inizio del programma
int main (int argc, char **argv)
{
     // Dichiarazioni variabili
     int menum,nproc,tag;  // Variabili per MPI
     int n,nloc,i,sum,resto,nlocgen;
     int ind,p,r,inviaA,ricevida,tmp;
     int *pot,*vett,*vett_loc,passi=0;
     int sumloc=0;

     /*
     *****************************************
     ** Fase di inizializzazione dell'ambiente di calcolo
     *****************************************
     */

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

     // Contiene le informazioni sulla ricezione del messaggio
     MPI_Status info;

     // Inizializzazione dell'ambiente di calcolo MPI
     MPI_Init(&argc,&argv);

     // Identificativo del processore (contenuto in menum)
     MPI_Comm_rank(MPI_COMM_WORLD, &menum);

     // Verifica del numero di processori della rete di calcolo
     MPI_Comm_size(MPI_COMM_WORLD, &nproc);


     /*
```

```
*******************************************
** Fase di inserimento dei dati
*******************************************
*/

// Seleziona il processore P0
if (menum==0)
{
     system("clear"); // 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;
}

// Crea dinamicamente dei vettori locali nei processori per le somme parziali
vett_loc=(int*)calloc(nloc, sizeof(int));

// Il processore P0  inserisce i dati nel vettore
if (menum==0)
{
     // Inizializza la generazione random degli addendi utilizzando... l'ora attuale del sistema
     srand((unsigned int) time(0));

     // Inserisi i dati meterialmente nel vettore
     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 più
                MPI_Send(vett+ind,nloc,MPI_INT,i,tag,MPI_COMM_WORLD);
                ind=ind+nloc;
            } // END THEN

            else
            {
                // Il processore P0 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


    /*
    ******************************************
    ** Calcolo delle somma parziali e totali
    ******************************************
    */

    // Uso di MPI_Barrier per la fase di sincronizzazione
    // 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
     sumloc=sumloc+*(vett_loc+i);
}

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

// Fase di calcolo del logaritmo in base 2 di nproc per selezionare
// i processori che effettuerann le somme parziali
while(p!=1)
{
     // shifta di un bit a destra
     p=p>>1;

     // determina il numero dei passi per sapere quante spedizioni di somme parziali bisogna fare
     passi++;
}

// allocazione dinamica del vettore pot che calcola la potenza di 2^n
pot=(int*)calloc(passi+1,sizeof(int));

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

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

```
            tag=menum;
            // ricezione del risultato della somma parziale di ricevida
            MPI_Recv(&tmp,1,MPI_INT,ricevida,tag,MPI_COMM_WORLD,&info);
            // calcolo della somma parziale al passo i
            sumloc=sumloc+tmp;
        }//end else
    }// end for

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

    // Calcolo del tempo di esecuzione
    MPI_Reduce(&T_fine,&T_max,1,MPI_DOUBLE,MPI_MAX,0,MPI_COMM_WORLD);

    if(menum==0)
    {

        // Visualizza i dati
        printf("\n\nProcessori impegnati: %d\n", nproc);
        printf("\nLa somma e': %d\n", sumloc);
        printf("\nTempo locale: %lf\n", T_fine);
        printf("\nMPI_Reduce max time: %f\n",T_max);
    }// end if

    // routine chiusura MPI
    MPI_Finalize();
}// fine prg
```