

Marco Casu

♯ Programmazione di Sistemi Multicore ♯



SAPIENZA
UNIVERSITÀ DI ROMA

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica
Dipartimento di Informatica

Questo documento è distribuito sotto la licenza [GNU](#), è un resoconto degli appunti (eventualmente integrati con libri di testo) tratti dalle lezioni del corso di Programmazione di Sistemi Multicore per la laurea triennale in Informatica. Se dovessi notare errori, ti prego di segnalarmeli.



INDICE

1	Parallelismo : Motivazioni	3
1.1	Introduzione	3
1.2	Modelli di Parallelismo	4
2	Memoria Distribuita : MPI	7
2.1	La libreria OpenMpi	7
2.2	Rank e Comunicazione	8
2.3	Design di Programmi Paralleli	10
2.3.1	Pattern di Design Parallelo	11
2.4	Comunicazione non Bloccante e Comunicazione Collettiva	12
2.4.1	Send e Recv Immediate	13
2.4.2	Esempi di Applicazione	13
2.4.3	Operazioni Collettive	15
2.5	Valutazione del Tempo	19

CAPITOLO

1

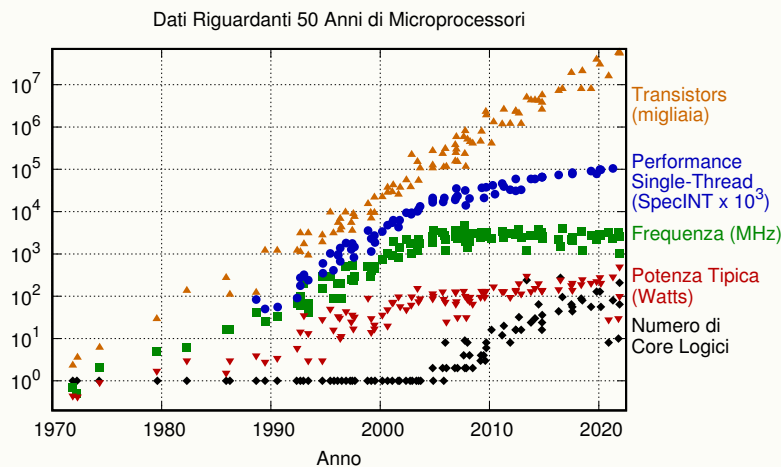
PARALLELISMO : MOTIVAZIONI

1.1 Introduzione

In una *GPU* (Graphics Processing Unit), nota anche come scheda video, ci sono circa 80 miliardi di transistor, e vengono utilizzate per allenare i grossi modelli di intelligenza artificiale, i quali necessitano migliaia di GPU, non è un caso se *Nvidia* ad oggi, con il boom dell'IA, è una delle aziende più quotate al mondo. Le GPU, e la loro programmazione, sono uno fra i principali argomenti di questo corso.

L'evoluzione dell'hardware, ha portato i grossi sistemi di computazione, ad essere formati da svariate unità di calcolo piuttosto che una singola unità molto potente, i processori stessi di uso comune, ad oggi sono composti da più *core*.

La legge di Moore riguarda una stima empirica che mette in correlazione lo scorrere del tempo con l'aumentare della potenza di calcolo dei processori, se inizialmente, a partire dagli anni 70, tale potenza raddoppiava ogni due anni, ad oggi tale andamento è andato rallentando, raggiungendo un incremento 1.5 in 10 anni.



L'obiettivo di costruire calcolatori sempre più potenti è dipeso dalla necessità dell'Uomo di risolvere problemi sempre più complessi, come ad esempio, la risoluzione del genoma umano.



Il motivo per il quale non è possibile costruire processori monolitici sempre più potenti, risiede in un *limite fisico* riguardante la densità massima possibile dei transistor in un chip.

1. transistor più piccoli \longrightarrow processori più veloci
2. processori più veloci \longrightarrow aumento del consumo energetico
3. aumento del consumo energetico \longrightarrow aumento del calore
4. aumento del calore \longrightarrow problemi di inaffidabilità dei transistor



1.2 Modelli di Parallelismo

L'informatico che intende scrivere del codice per un sistema multicore, deve esplicitamente sfruttare i diversi core, limitandosi a scrivere un codice sequenziale, non starebbe sfruttando a pieno l'hardware a disposizione, rendendo il processo meno efficiente di quanto potrebbe essere.

La maggior parte delle volte, un algoritmo sequenziale, non può essere direttamente tradotto in un algoritmo parallelo, per questo bisogna scrivere il codice facendo riferimento all'hardware di destinazione. Si consideri adesso il seguente codice sequenziale, che ha lo scopo di sommare n numeri dati in input.

```

1  sum = 0;
2  for (i=0; i<n; i++){
3      x = compute_next_value (...);
4      sum += x;
5  }
```

Si vuole rendere tale algoritmo parallelo, sapendo di essere a disposizione di p core.

```

1  local_sum = 0;
2  first_index = ...;
3  last_index = ...;
4  for (local_i=first_index; first_index<last_index; local_i++){
5      local_x = compute_next_value (...);
6      local_sum += local_x;
7  }
```

In tale esempio, ogni core possiede le sue variabili private non condivise con gli altri core, ed esegue indipendentemente il blocco di codice. Ogni core conterrà la somma parziale di n/p valori.

Esempio (24 numeri, 8 core) :

valori : 1, 4, 3, 9, 2, 8, 5, 1, 1, 6, 2, 7, 2, 5, 0, 4, 1, 8, 6, 5, 1, 2, 3, 9

core	0	1	2	3	4	5	6	7
local_sum	8	19	7	15	7	13	12	14

A questo punto, per ottenere la somma totale, vi sarà un core *master* che riceverà le somme parziali da tutti gli altri core, per poi eseguire la somma finale.

```

1  if (master){
2      sum = local_sum;
3      for c : core{
4          if (c!=self){
5              sum += c.local_sum;
6          }
7      }
8  } else{
9      send local_sum to master;
10 }
```

Dividere i dati per poi far eseguire la stessa computazione ai diversi nodi è la forma più semplice di parallelismo. La soluzione adottata non è ideale, in quanto, in seguito al calcolo delle somme parziali, tutti i core escluso il master non staranno eseguendo calcoli. Una possibile idea alternativa è di far sì che a coppie i nodi si condividano le somme parziali per poi calcolarne una somma comune, sviluppando uno scambio di dati ad albero, come mostrato in figura 1.1.

Possiamo identificare due tipi di parallelismo :

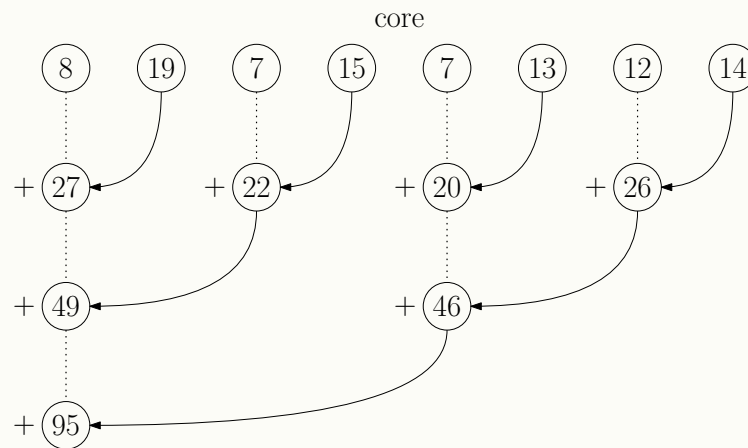


Figura 1.1: calcolo somme a coppie

- **parallelismo dei task** : fra i core vengono divise diverse attività che vengono svolte autonomamente.
- **parallelismo dei dati** : i dati da elaborare vengono divisi, ogni core eseguirà la stessa computazione ma su una porzione diversa dei dati.

Quando si scrive un programma parallelo bisogna prestare attenzione alla *sincronizzazione* dei processi, in quanto potrebbero dover accedere ad una stessa area di memoria. Risulta cruciale saper mettere in *comunicazione* i vari core, e suddividere equamente il *carico di lavoro* fra di essi. Verranno considerate 4 diverse tecnologie per la programmazione multicore :

- *MPI* (Message Passing Interface) [libreria]
- *Posix* Threads [libreria]
- *OpenMP* [libreria e compilatore]
- *CUDA* [libreria e compilatore]

La programmazione delle GPU richiederà un diverso compilatore, e non il solito `gcc`, in quanto l'architettura della scheda video differisce da quella del processore, e con essa le istruzioni.

I sistemi paralleli possono essere categorizzati sotto vari aspetti.

- **shared memory** : Tutti i core accedono ad un'area di memoria comune. L'accesso e la sincronizzazione vanno gestiti con cautela.
- **distributed memory** : Ogni core ha un area di memoria privata, e la comunicazione avviene attraverso un apposito canale per lo scambio dei messaggi.

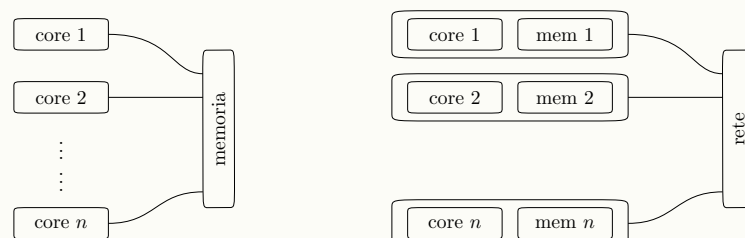


Figura 1.2: modelli di parallelismo

Vi è un'altra suddivisione nei sistemi paralleli :

- **MIMD** : Ogni core ha una control unit indipendente, diversi core possono eseguire diverse istruzioni nello stesso momento.



- **SIMD** : Vi è un singolo program counter per tutti i core, che eseguono in maniera parallela le stesse istruzioni. Due core non possono eseguire operazioni diverse nello stesso momento.

Le GPU hanno una struttura *SIMD*.

	shared memory	distributed memory
SIMD	CUDA	
MIMD	Pthreads/OpenMP/CUDA	MPI

Fin'ora sono stati utilizzati 3 termini chiave riguardante i tipi di programmazione, sebbene non vi sia una definizione comunemente accettata, la seguente verrà adottata in tale contesto :

- *concorrente* : più processi sono attivi in uno stesso momento
- *parallela* : diverse entità cooperative che operano in maniera ravvicinata per un obiettivo comune.
- *distribuita* : diverse entità cooperative.

La programmazione parallela o distribuita implica che sia anche concorrente, non è vero il contrario.

CAPITOLO

2

MEMORIA DISTRIBUITA : MPI

MPI è una libreria standard (avente varie implementazioni) necessaria allo sviluppo di codice multiprocesso a memoria distribuita. Precisamente, ogni core ha una memoria privata inaccessibile dall'esterno, e la comunicazione avviene attraverso una rete di interconnessione, (ad esempio, un bus), tale modello è detto **message passing**.

2.1 La libreria OpenMpi

Alla compilazione ed avvio di un programma che sfrutta MPI, ogni core eseguirà il programma, sarà la logica di esso a suddividere il carico di lavoro, tramite i costrutti decisionali. Verrà utilizzata un'implementazione nota come *openMpi*, è possibile installare la libreria su sistemi operativi linux tramite il comando

```
sudo apt-get install libopenmpi-dev
```

Il seguente esempio, mostra un programma che scrive sulla console una stringa, e tramite MPI, tale processo è avviato su ogni core.

```
1  #include <stdio.h>
2  #include <mpi.h>
3  //voglio lanciare il programma su piu unita di calcolo
4  int main(int argc, char **argv){
5      int p = MPI_Init(NULL,NULL);
6      //Il parametro in output di MPI_Init e' uno status sull'errore
7      if(p == MPI_SUCCESS){
8
9      } else {
10         printf("qualcosa e' andato storto");
11         MPI_Abort(MPI_COMM_WORLD,p);
12         //Con MPI_Abort tutti i processi su tutti i core avviati verranno terminati
13     }
14     printf("hello world");
15     MPI_Finalize(); //Serve per terminare la libreria
16     return 0;
17 }
```

I programmi MPI non vengono compilati con **gcc**, ma con **mpicc**

```
mpicc hello_world.c -o hello_world.out
```

Una volta ottenuto l'eseguibile, è possibile lanciare il programma con **mpirun** specificando il numero di core sulla quale verrà eseguito il programma, tale numero, se non specificato con apposite flag, deve



essere minore o uguale al numero di core fisici presenti sulla macchina.

```
mpirun -n 4 hello_world.out
```

Ogni funzione della libreria ha una dicitura che inizia con `"MPI_"`. Ogni funzione di libreria deve essere chiamata fra

- `MPI_Init` - configurazione ed avviamento della libreria
- `MPI_Finalize` - chiusura e deallocazione della memoria

Tali righe stabiliscono il blocco di codice in cui verranno eseguite funzioni MPI.



2.2 Rank e Comunicazione

Ogni processo MPI è univocamente identificato da un numero intero detto *rank*, se p processi sono attivi, avranno gli identificatori $1, 2, \dots, p - 1$.

Un **comunicatore** è un insieme di processi, i quali hanno la possibilità di scambiarsi messaggi, si può pensare ad un comunicatore come un'etichetta, e processi con la stessa etichetta possono comunicare fra loro. È identificabile nel codice tramite la struttura dati `MPI_Comm`, e all'avvio di MPI, viene sempre definito un comunicatore di default `MPI_COMM_WORLD` che contiene tutti i processi.

L'identificatore di ogni processo è in realtà relativo ad ogni comunicatore, due processi diversi possono condividere il rank se relativo a comunicatori diversi. Ci sono due funzioni importanti che riguardano questi ultimi

- `int MPI_Comm_rank(MPI_Comm comm, int *rank)` : Prende in input un comunicatore ed un numero intero, e salva dentro tale numero il rank del processo chiamante relativo al comunicatore dato.
- `int MPI_Comm_size(MPI_Comm comm, int *size)` : Prende in input un comunicatore ed un numero intero, e salva dentro tale intero il numero di processi all'interno del comunicatore.

```

1  #include <stdio.h>
2  #include <mpi.h>
3  //voglio lanciare il programma su piu unita di calcolo
4  int main(int argc, char **argv){
5      int p = MPI_Init(NULL,NULL);
6      //Il parametro in output di MPI_Init e' uno status sull'errore
7      if(p == MPI_SUCCESS){
8
9      } else{
10         printf("qualcosa e' andato storto");
11         MPI_Abort(MPI_COMM_WORLD,p);
12         //Con MPI_Abort tutti i processi su tutti i core avviati verranno terminati
13     }
14     int size;
15     MPI_Comm_size(MPI_COMM_WORLD, &size);
16     int rank;
17     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
18     printf("hello world, im the process %d/%d",rank,size);
19     MPI_Finalize(); //Serve per terminare la libreria
20     return 0;
21 }

```

La comunicazione avviene tramite due funzioni, il cui comportamento è simile alla comunicazione tramite `pipe`.

L'inizio dei messaggi avviene tramite `int MPI_Send`, i cui parametri sono

- `void* msg_buf_p` l'area di memoria da trasferire al processo destinatario



- `int msg_size` il numero di elementi (non l'occupazione in byte) del messaggio da trasferire
- `MPI_Datatype msg_type` il tipo di elemento da trasferire. Sono definiti dei tipi standard che incorporano tutti i tipi più comuni del *C*
- `int dest` il rank del processo destinatario
- `int tag` un tag da dare al messaggio per identificarlo
- `MPI_Comm communicator` il comunicatore su cui avviene la comunicazione

Può dipendere dall'implementazione, ma solitamente quando un processo fa una `MPI_Send`, si arresta finché il messaggio inviato non viene ricevuto dal destinatario, allo stesso modo, un destinatario che si appresta a ricevere un messaggio viene arrestato fino al ricevimento. Le chiamate di comunicazione MPI sono quindi bloccanti.

Per ricevere dati, viene utilizzata la chiamata `MPI_Recv` i cui parametri sono

- `void* msg_buf_p` l'area di memoria su cui verrà salvato il messaggio
- `int buf_size` il numero di elementi (non l'occupazione in byte) del messaggio da ricevere
- `MPI_Datatype buf_type` il tipo di elemento da ricevere
- `int source` il rank del processo mittente
- `int tag` il tag del messaggio da ricevere
- `MPI_Comm communicator` il comunicatore su cui avviene la comunicazione
- `MPI_Status* status` lo status riguardante l'esito della comunicazione

OpenMpi definisce la seguente lista di tipi `MPI_Datatype` :

<code>MPI_CHAR</code>	carattere
<code>MPI_INT</code>	intero
<code>MPI_FLOAT</code>	float a singola precisione
<code>MPI_DOUBLE</code>	float a doppia precisione
<code>MPI_LONG</code>	intero long
<code>MPI_SHORT</code>	intero short
<code>MPI_UNSIGNED_CHAR</code>	carattere senza segno
<code>MPI_UNSIGNED_INT</code>	intero senza segno
<code>MPI_UNSIGNED_LONG</code>	intero long senza segno
<code>MPI_UNSIGNED_SHORT</code>	intero short senza segno

Il seguente programma fa sì che ogni processo invii un messaggio al processo di rank 0, e quest'ultimo lo stampi a schermo.

```

1 #include <stdio.h>
2 #include <mpi.h>
3
4 int main(int argc, char **argv)
5 {
6     int p = MPI_Init(NULL, NULL);
7     // Il parametro in output di MPI_Init e' uno status sull'errore
8     if (p != MPI_SUCCESS)
9     {
10         printf("qualcosa e' andato storto");
11         MPI_Abort(MPI_COMM_WORLD, p);
12         // Con MPI_Abort tutti i processi su tutti i core avviati verranno terminati
13     }
14     int size;
15     MPI_Comm_size(MPI_COMM_WORLD, &size);
16     int str_size = 256;
17     int rank;

```



```

18 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
19 if (rank == 0)
20 {
21     printf("hello world, i am process 0. I will recive and print.\n", rank, size);
22     char str[str_size];
23     for (int i = 1; i < size; i++)
24     {
25         MPI_Recv(str, str_size, MPI_CHAR, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
26         printf("(STRING RECIVED) : %s", str);
27     }
28 }
29 else
30 {
31     char str[str_size];
32     sprintf(str, "hello world, i am process %d of %d\n", rank, size);
33     // Si invia al processo 0
34     MPI_Send(str, str_size, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
35 }
36
37 MPI_Finalize(); // Serve per terminare la libreria
38 return 0;
39 }
40

```

Quando un processo esegue una `MPI_Recv`, fra i vari messaggi, viene cercato quello di cui matchano il tag, il comunicatore, ed il mittente, lo scopo del `tag` è quello di essere un ulteriore separatore logico per la comunicazione. Anche i tipi dei messaggi devono combaciare, inoltre il numero di byte da ricevere deve essere maggiore o uguale al numero di byte inviati

$$ByteRecv \geq ByteSent$$

Nella chiamata `MPI_Recv`, i campi `source` e `tag` possono essere riempiti con, rispettivamente, `MPI_ANY_SOURCE` e `MPI_ANY_TAG` per non eseguire il controllo su mittente e tag nel ricevimento. È comunque possibile sapere qual'è il mittente, dato che tale informazione è salvata nel campo `MPI_Status`.



2.3 Design di Programmi Paralleli

Data la specifica di un programma, quali sono le regole da seguire per partizionare il carico di lavoro fra i vari processi? Non esistono delle regole adatte ad ogni evenienza, ma è stata definita una metodologia largamente generica, la **Foster's methodology**.

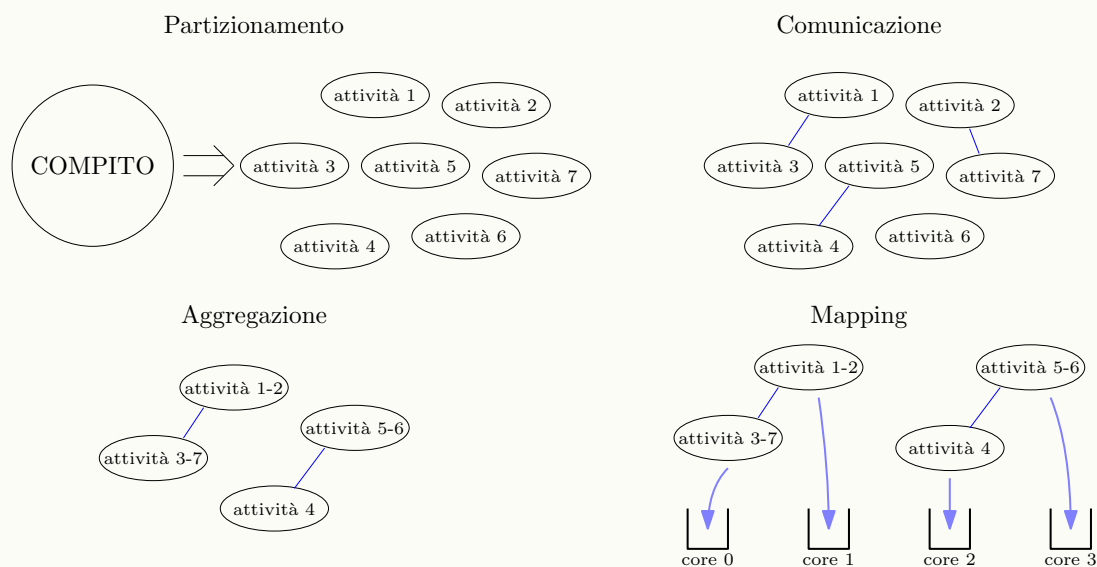


Figura 2.1: Foster's methodology

1. *Partizionamento* : si identificano delle attività di base indipendenti fra loro che possono essere eseguite in parallelo.
2. *Comunicazione* : determinare quali sono le attività stabilite nel punto precedente che per essere eseguite necessitano di uno scambio di messaggi.
3. *Aggregazione* : identificare le attività precedentemente stabilite che devono necessariamente essere eseguite in sequenza, ed aggregarle in un'unica attività.
4. *Mapping* : assegnare ai vari processi le attività definite in precedenza in modo che il carico di lavoro sia uniformemente distribuito. Idealmente la comunicazione deve essere ridotta al minimo.

2.3.1 Pattern di Design Parallelo

La struttura di un programma parallelo può essere definita secondo due pattern, si può dire che esistono due modi di *parallelizzare* un programma

- **GPLS (Globally Parallel, Locally Sequential)** : L'applicazione vede diversi task sequenziali venire eseguiti in parallelo.
- **GSLP (Globally Sequential, Locally Parallel)** : L'applicazione segue uno specifico "flusso" di esecuzione sequenziale, di cui alcune parti vengono eseguite in parallelo.

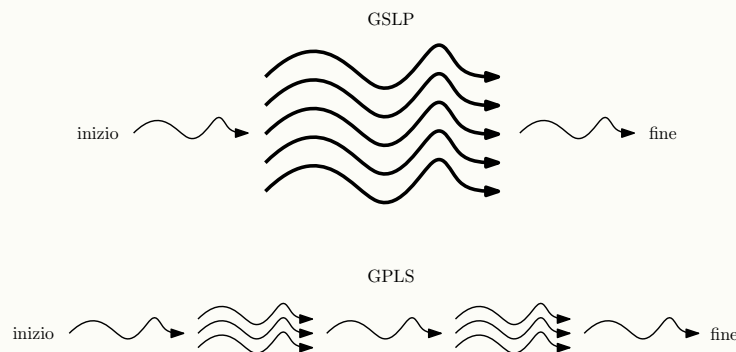


Figura 2.2: GPLS e GSLP

Esempi di GPLS

- **Single Program Multiple Data** : La logica dell'applicazione viene mantenuta in un unico eseguibile, tipicamente il programma segue la seguente struttura
 1. Inizializzazione del programma
 2. Ottenimento degli identificatori
 3. Esecuzione del programma in diverse ramificazioni in base ai core coinvolti
 4. Terminazione del programma
- **Multiple Program Multiple Data** : Quando la memoria da utilizzare è elevata è necessario suddividere il carico su più programmi, che spesso vengono eseguiti su differenti piattaforme.
- **Master-Worker** : Ogni processo può essere
 - Worker - Esegue la computazione
 - Master - Gestisce il carico di lavoro e lo assegna ai processi worker, colleziona i risultati ottenuti da questi ultimi e si occupa spesso delle operazioni di I/O o interazione con l'utente.
- **Map-Reduce** : Una versione modificata del paradigma Master-Worker, in cui i nodi worker eseguono due tipi di operazioni
 - Map : Esegue la computazione su un insieme di dati che risulta in un insieme di risultati parziali (ad esempio, esegue la somma su ogni elemento di un vettore)
 - Reduce : Collezione i risultati parziali e ne deriva un risultato finale (ad esempio, somma tutti gli elementi di un vettore ottenendo un unico scalare)



Esempi di GSLP

- **Fork-Join** : C'è un unico "padre" in cui avviene l'esecuzione, quando necessario, tale padre potrebbe eseguire una `fork` generando dei nodi figli, che eseguono la computazione per poi terminare, facendo sì che il padre continui.
- **Loop-Parallelism** : Risulta estremamente semplice da utilizzare e viene spesso applicata quando un programma sequenziale deve essere adattato al multiprocesso. Consiste nel parallelizzare ogni esecuzione di un ciclo `for`, è necessario che le iterazioni però siano indipendenti fra loro.

```

1 //Esempio di Fork-Join
2 mergesort(A, lo , hi){
3     if lo < hi{
4         mid = lo + (hi-lo) / 2
5         fork mergesort(A, lo , mid)
6         mergesort(A, mid , hi)
7
8         join
9         merge(A, lo , mid , hi)
10    }
11 }
```



2.4 Comunicazione non Bloccante e Comunicazione Collettiva

Il contesto canonico di utilizzo di MPI è su un'insieme di server connessi fra loro (memoria privata), quando un processo esegue una `MPI_Send`, il buffer in cui è contenuto il messaggio viene copiato e salvato dalla memoria principale alla memoria dell'interfaccia di rete (NIC Memory), per poi venire trasferito attraverso la rete verso la memoria NIC del destinatario, da lì, verrà poi trasferita nella memoria principale di quest'ultimo.

L'utilizzo di una `MPI_Send` è quindi dispendioso dal punto di vista computazionale, in quanto sono coinvolte molteplici operazioni di scrittura e chiamate di sistema, è quindi buona regola, eseguire il minor numero di `MPI_Send` possibile

Ad esempio, è più conveniente eseguire una sola chiamata in cui si trasferiscono 200 byte piuttosto che due chiamate in cui si trasferiscono 100 byte ciascuna.

Si è detto in precedenza che `MPI_Send` è bloccante, in realtà, MPI utilizza, se non specificato diversamente, una metodologia di comunicazione standard, se il messaggio da trasferire è piccolo, è probabile che venga immediatamente trasferito venendo salvato su un buffer del destinatario. Diversamente, nel caso di un messaggio grande, la chiamata sarà bloccante in quanto MPI deve assicurarsi che il destinatario abbia allocato la memoria sufficiente per riceverlo.

In entrambi i casi, MPI si assicura che il messaggio da inviare non vada perso, il programma riottiene il controllo solo quando il buffer utilizzato per contenere il messaggio è di nuovo disponibile, si dice che la `MPI_Send` è *locally blocking*. Oltre la comunicazione standard, vi sono altri modi di inviare messaggi

- **Buffered** : Tramite la chiamata `MPI_Bsend`, l'operazione è sempre *locally blocking*, ma l'utente deve fornire manualmente un buffer in cui salvare il messaggio da inviare.
- **Sincrona** : Tramite la chiamata `MPI_Ssend`, l'operazione è globalmente bloccante, il controllo viene restituito esclusivamente quando il destinatario ha ricevuto il messaggio chiamando `MPI_Recv`. Risulta utile per far sì che un processo attenda che un altro arrivi ad un certo punto della computazione.
- **Ready** : Tramite la chiamata `MPI_Rsend`, se il destinatario non ha già effettuato una `MPI_Recv`, tale chiamata fallisce, è quindi necessario che esso sia già in attesa di ricevere.

2.4.1 Send e Recv Immediate

Le chiamate `MPI_Recv` e `MPI_Send` sono considerate poco performanti in quanto il processo chiamante potrebbe bloccare la sua esecuzione, in alcuni casi può essere utile una chiamata non bloccante per la trasmissione dei dati, soprattutto quando il mancato ricevimento di essi non causa errori nell'esecuzione del programma. Le funzioni non bloccanti messe a disposizione da MPI sono dette **funzioni immediate**, e permettono l'overlap fra computazione e comunicazione. Se al momento di una chiamata di ricevimento non ci sono dati da leggere, il programmatore dovrà gestire esplicitamente la situazione.

La chiamata `MPI_Isend` ha gli stessi parametri della funzione non immediata, eccetto un parametro aggiuntivo, `MPI_Request *req`, necessario per avere informazioni sullo status della chiamata.

La chiamata `MPI_Irecv` ha gli stessi parametri della funzione non immediata, eccetto per l'assenza del parametro sullo status originario, e l'aggiunta del parametro `MPI_Request *req`, necessario per avere informazioni sullo status della chiamata.

La funzione `int MPI_Wait(MPI_Request *request, MPI_Status *status)` fa sì che il processo si blocchi finché un invio o una ricezione non è andato a buon termine. È una chiamata bloccante.

La funzione `int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)` controlla se una chiamata di invio o ricezione è andata o no a buon fine, salvando l'esito del risultato nel campo `flag`.

Esistono altre varianti di `Wait` e `Test`

- `Waitall`
- `Waitany`
- `Testany`
- etc...

2.4.2 Esempi di Applicazione

Il seguente esempio mostra un programma in cui n processi (in questo caso 4) si scambiano informazioni in una configurazione "ad anello", in cui ognuno invia e riceve a/da i suoi vicini, l'utilizzo di chiamate non bloccanti è utile per evitare situazioni di deadlock.

```

1 #include "mpi.h"
2 #include <stdio.h>
3 int main(void) {
4     int numtasks, rank, next, prev, buf[2];
5     MPI_Request reqs[4]; // variabili necessarie per le chiamate Irecv e Isend
6     MPI_Status stats[4]; // variabili necessarie per Waitall
7     MPI_Init(NULL, NULL);
8     MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
9     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
10    // Determina vicino a sinistra e a destra
11    prev = (rank-1) % numtasks;
12    next = (rank+1) % numtasks;
13    // Operazioni di comunicazione
14    MPI_Irecv(&buf[0], 1, MPI_INT, prev, 0, MPI_COMM_WORLD, &reqs[0]);
15    MPI_Irecv(&buf[1], 1, MPI_INT, next, 0, MPI_COMM_WORLD, &reqs[1]);
16    MPI_Isend(&rank, 1, MPI_INT, prev, 0, MPI_COMM_WORLD, &reqs[2]);
17    MPI_Isend(&rank, 1, MPI_INT, next, 0, MPI_COMM_WORLD, &reqs[3]);
18    // Qui puo' essere eseguita computazione nel mentre che gli altri processi comunicano
19    // Attende la fine delle operazioni non bloccanti
20    MPI_Waitall(4, reqs, stats);
21    MPI_Finalize();
22 }
```

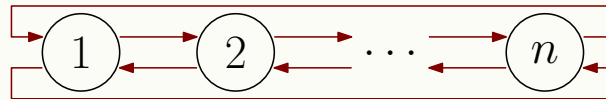


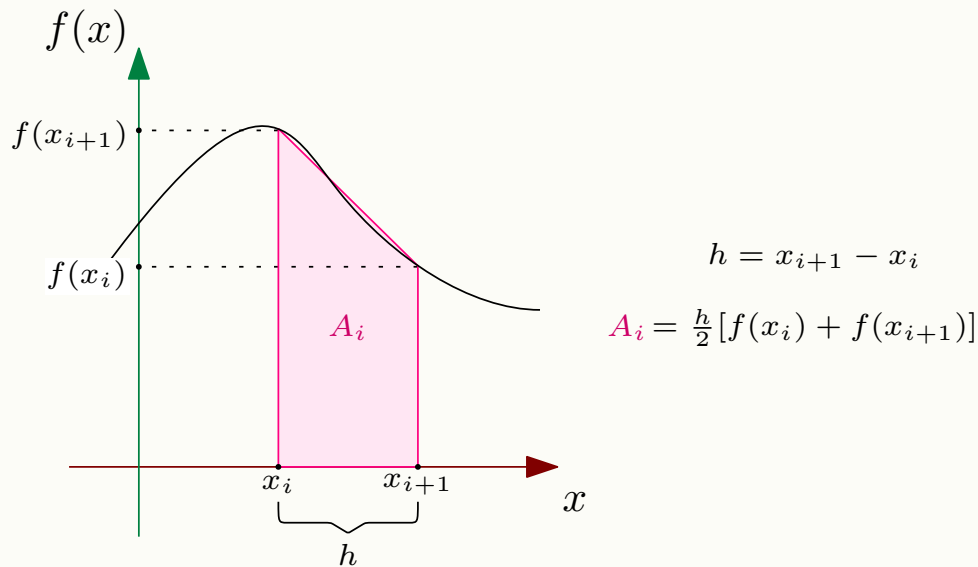
Figura 2.3: configurazione ad anello

Integrazione numerica

Si consideri adesso il seguente esempio, si vuole scrivere un programma che esegua l'integrazione numerica di una generica funzione $f(x)$ tramite la regola del trapezoido. Tale metodo consiste nel dividere l'intervallo di integrazione in n intervalli

$$\{(x_0, x_1), (x_1, x_2), (x_2, x_3) \dots (x_{n-1}, x_n)\}$$

lunghe h , di cui verrà calcolata l'area approssimandola ad un trapezio.



L'integrale approssimato sarà la somma totale di tutti i trapezoidi

$$\frac{h}{2} [f(x_1) + f(x_2)] + [f(x_2) + f(x_3)] + \dots + [f(x_{n-1}) + f(x_n)]$$

```

1 /* Input : a, b, n */
2 h = (b-a)/n;
3 approx = (f(a)+f(b))/2;
4 for (i=1; i<=n-1; i++){
5     x_i = a+i*h;
6     approx += f(x_i);
7 }
8 approx=h*approx;

```

Se ne vuole dare un'implementazione parallela in cui i vari processi eseguiranno il calcolo di un trapezio, le somme parziali verranno inviate al processo di rank 0 che si occuperà di calcolare la somma totale (paradigma MAP REDUCE).

```

1 double Trap(double left, double right, int count, double base_len){
2
3     double estimate, x;
4     // f e' la funzione integranda
5     estimate = (f(left)+f(right))/2.0;
6     for(int i = 1; i<=count-1; i++){
7         x=left+i*base_len;
8         estimate+=f(x);
9     }
10    return estimate*base_len;
11 }

```



```

1 int main(void){
2
3     int my_rank;
4     int comm_size;
5     int n = 1024; //numero di intervalli, piu' e' grande, piu' la stima sara' precisa
6     double a = 0.0; //estremo sinistro di integrazione
7     double b = 3.0; //estremo destro di integrazione
8     double h; //lunghezza intervalli;
9     double local_a, local_b;
10    double local_sum;
11    double total_sum;
12    int source;
13
14    MPI_Init(NULL, NULL);
15    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
16    MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
17
18    h = (b-a)/n;
19    local_n = n/comm_size; //numero di trapezoidi per ogni processo
20
21    local_a = a+my_rank*local_n*h;
22    local_b = local_a+local_n*h;
23    local_sum = Trap(local_a, local_b, local_n, h); //calcolo somma parziale
24
25    if(my_rank!=0){
26        //Invio la somma parziale al processo con rank 0
27        MPI_Send(&local_sum, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
28    }
29    else{
30        total_sum = local_sum;
31        for(source = 1; source < comm_sz; source++){
32            MPI_Recv(&local_sum, 1, MPI_DOUBLE, source, 0,
33                    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
34            total_sum += local_sum;
35        }
36
37        printf("con n = %d trapezoidi, la somma approssimata della funzione\n
38              da %f a %f e' %.15e .\n", n, a, b, total_sum);
39    }
40
41    MPI_Finalize();
42    return 0;
43 }

```

2.4.3 Operazioni Collettive

Qual'è il problema con l'implementazione del trapezoide appena mostrata? Il processo di rank zero ha un carico di lavoro superiore rispetto ogni altro processo, infatti, quest'ultimo oltre la somma dei suoi trapezi locali, deve calcolare la somma totale, inoltre deve occuparsi di ricevere i dati da tutti gli altri processi.

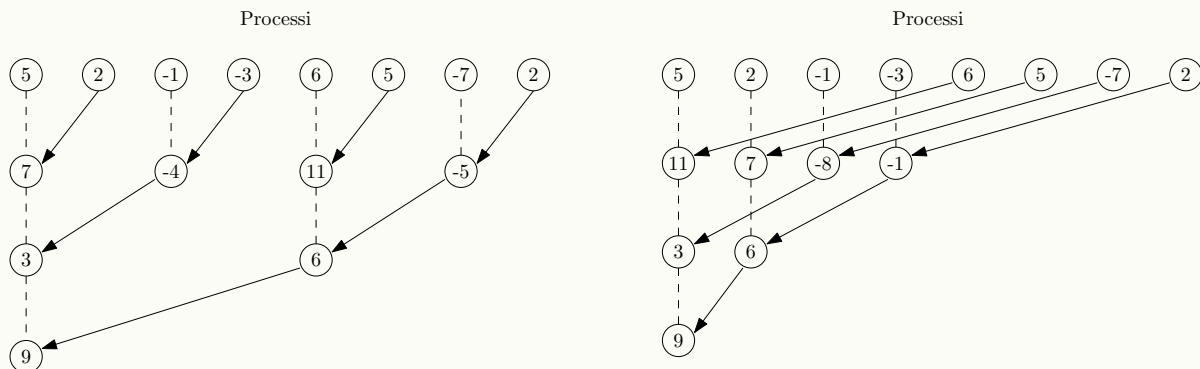


Figura 2.4: alberi differenti (entrambi validi)



Si può pensare di suddividere il carico di lavoro ad albero, come già visto in figura 1.1, facendo sì che il suo carico di lavoro sia logaritmico in funzione del numero dei rank. Nell'esempio visto, ogni processo condivide i suoi dati parziali con quello adiacente (da un punto di vista di numero di identificazione), ma nulla vieta agli ultimi processi di condividere i dati con i primi, come in figura 2.4. L'ottimalità di una soluzione piuttosto che di un'altra può dipendere da diversi fattori non sempre analizzabili, come la topologia fisica della rete attraverso cui sono collegate le macchine che eseguono i processi.

A tal proposito, MPI fornisce una funzionalità che permette di eseguire operazioni di aggregazione di risultati senza preoccuparci della logica di comunicazione per il trasferimento dei dati parziali. La funzione in questione è `int MPI_Reduce`, con i seguenti parametri

- `void* input_data_o` è il puntatore alla variabile in ingresso (somma parziale)
- `void* output_data_o` è il puntatore al valore che sarà riempito con il valore totale aggregato
- `int count` è il numero di elementi da aggregare
- `MPI_Datatype datatype` è il tipo dei valori in questione
- `MPI_Op operator` è l'operazione di aggregazione (somma, moltiplicazione, XOR, etc...)
- `int dest_process` è il rank del processo che riceverà il risultato
- `MPI_Comm comm` il comunicatore in questione

Le operazioni di aggregazione supportate da MPI sono le seguenti

Operazione	Significato
<code>MPI_MAX</code>	Massimo
<code>MPI_MIN</code>	Minimo
<code>MPI_SUM</code>	Somma
<code>MPI_PROD</code>	Prodotto
<code>MPI_LAND</code>	AND logico
<code>MPI_BAND</code>	AND bit a bit
<code>MPI_LOR</code>	OR logico
<code>MPI BOR</code>	OR bit a bit
<code>MPI_LXOR</code>	XOR logico
<code>MPI_BXOR</code>	XOR bit a bit
<code>MPI_MAXLOC</code>	Massimo insieme al suo indice
<code>MPI_MINLOC</code>	Minimo insieme al suo indice

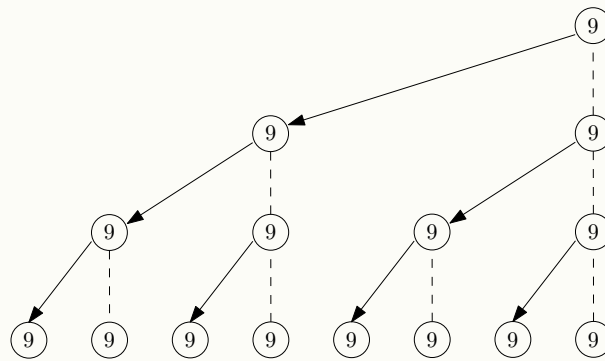
È possibile anche definire delle operazioni personalizzate tramite la chiamata `MPI_Op_create`.

Quando viene chiamata una funzione collettiva, è importante che ogni processo del comunicatore la chiami, altrimenti l'esecuzione rimane bloccata in uno stato di attesa, dato che ogni processo attende che tutti gli altri siano arrivati a tale operazione. Ovviamente, tutti i processi che eseguono un'operazione di questo tipo devono definire lo stesso processo che riceverà l'output. Tutti i processi escluso quello di destinazione, nel campo `void* output_data_o` possono specificare qualsiasi valore.

Non essendo presente alcun tag, nella comunicazione collettiva, le operazioni verranno matchate in base all'ordine di esecuzione. Nell'esempio del trapezoide 2.4.2 vi è un problema, nel caso si volesse decidere arbitrariamente l'intervallo di integrazione, o il numero di trapezi, il processo di rank 0 dovrà occuparsi di leggere i dati da stdin, per poi condividerli ad ogni altro processo.

Si ricordi che in MPI, esclusivamente il processo di rank 0 può interagire con lo stdin.

È chiaro che su di esso sia riportato un carico di lavoro maggiore, è quindi possibile suddividere il carico facendo sì che il processo 0 condivida i dati con due altri processi, e questi due li dividano a loro volta con altri due processi ciascuno, creando un albero di condivisione.



Anche in questo caso, la logica con la quale scambiarsi le informazioni può variare, e quella ottimale può dipendere dalle condizioni della rete ed altri fattori difficilmente analizzabili, per questo MPI fornisce una funzione, `MPI_Bcast` che si occupa di eseguire il broadcast da un processo verso tutti gli altri. I parametri sono i seguenti :

- `void* data_p` è il puntatore alla variabile da condividere
- `int count` è il numero di elementi da condividere
- `MPI_Datatype datatype` è il tipo dei valori in questione
- `int source_process` è il rank del processo che condivide il valore
- `MPI_Comm comm` il comunicatore in questione

Il parametro `void* data_p` fungerà sia da input, che da output, nel caso il processo chiamante sia colui che condivide il valore, in `data_p` sarà presente il valore condiviso, altrimenti, in `data_p` sarà presente il valore ricevuto.

Esempio di funzione per leggere input da tastiera

```

1 void Get_input(int my_rank, a, int b, int n){
2     if(my_rank==0){
3         printf("enter input:\n");
4         scanf("%d %d %d", a, b, n);
5     }
6     MPI_Bcast(a, 1, MPI_INT, 0, MPI_COMM_WORLD);
7     MPI_Bcast(b, 1, MPI_INT, 0, MPI_COMM_WORLD);
8     MPI_Bcast(n, 1, MPI_INT, 0, MPI_COMM_WORLD);
9 }

```

A questo punto, si supponga di voler fare un'operazione di aggregato, per poi avere il risultato condiviso fra tutti i processi, concettualmente, ciò equivale ad eseguire una `MPI_Reduce` seguita da una `MPI_Bcast`. MPI fornisce una funzione a tal proposito, ottimizzata a dovere, ossia `MPI_Allreduce`, con i seguenti parametri

- `void* input_data_o` è il puntatore alla variabile in ingresso (somma parziale)
- `void* output_data_o` è il puntatore al valore che sarà riempito con il valore totale aggregato
- `int count` è il numero di elementi da aggregare
- `MPI_Datatype datatype` è il tipo dei valori in questione
- `MPI_Op operator` è l'operazione di aggregazione (somma, moltiplicazione, XOR, etc...)
- `MPI_Comm comm` il comunicatore in questione

I parametri sono identici alla `MPI_Reduce`, eccetto per l'assenza del processo di destinazione, dato che in questo caso, ogni processo avrà il risultato.

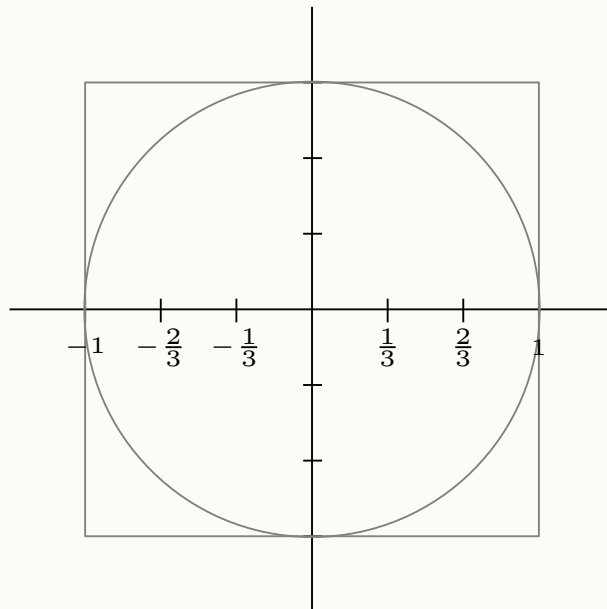
Le operazioni collettive sono diventate particolarmente importanti nell'ultimo periodo in quanto sono utilizzate nella stragrande maggioranza dei programmi paralleli che vengono eseguiti per il training delle reti neurali odierne. Le grosse aziende di informatica, hanno iniziato a produrre delle proprie librerie proprietarie

- NCCL (*Nvidia*)
- RCCL (*AMD*)
- OneCCL (*Intel*)
- MSCCL (*Microsoft*)

MPI durante le operazioni aggregate utilizza delle euristiche per stimare quale sia il miglior modo di condividere i dati fra i nodi, è possibile forzare tale decisione attraverso delle opportune variabili d'ambiente. Il punto è che MPI non è consapevole dell'hardware sul quale i processi sono eseguiti, per questo le aziende hanno iniziato a produrre librerie proprietarie, appositamente ottimizzate per girare sulle piattaforme dedicate.

Stima del π

Si vuole scrivere un programma che tramite il metodo di Montecarlo calcoli il valore stimato di π distribuendo il lavoro su più processi tramite MPI. L'algoritmo utilizzato per il calcolo è semplice, si consideri un cerchio di raggio unitario, inscritto in un quadrato 2×2 .



L'area del cerchio, è uguale a $\pi r^2 \Rightarrow \pi$, l'area del quadrato è 4. Sia A_c l'insieme di tutti i punti compresi nel cerchio, e sia A_q l'insieme di tutti i punti compresi nel quadrato. Risulta che il numero di punti nell'area del cerchio stanno all'area π , come il numero di punti che stanno nell'area del quadrato stanno a 4.

$$\frac{|A_c|}{\pi} = \frac{|A_q|}{4}$$

In realtà, non ha senso considerare la cardinalità di $|A_c|$ o di $|A_q|$, in quanto sono insiemi infiniti, supponiamo allora che tali insiemi siano finiti e di cardinalità n , si denotano A_c^n e A_q^n , chiaramente $A_c^n \subseteq A_q^n$. Si ha che

$$\lim_{n \rightarrow \infty} 4 \cdot \frac{|A_c^n|}{|A_q^n|} = \pi$$

L'algoritmo consiste nel calcolare un numero n di punti casuali, sia c il numero di punti interni al cerchio, ossia i punti (x, y) tali da rispettare $x^2 + y^2 \leq 1$. Numericamente, $4 \frac{n}{c}$ approssimerà π , con una precisione sempre maggiore all'aumentare di n .

L'algoritmo si renderà parallelo, distribuendo equamente il numero di punti da calcolare a tutti i processi presenti.



```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <mpi.h>
4  #include <time.h>
5
6  int main(int argc, char **argv)
7  {
8
9      int precision = 1000; // Numero di punti generati casualmente
10
11     if (argc > 1)
12     {
13         precision = atoi(argv[1]);
14     }
15
16     MPI_Init(NULL, NULL);
17     srand(time(NULL));
18
19     int my_rank;
20     int my_size;
21     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
22     MPI_Comm_size(MPI_COMM_WORLD, &my_size);
23
24     int local_precision = precision / my_size; /* Numero di punto da generare per
25                                                ogni processo */
26     int local_circle_point = 0;
27
28     for (int i = 0; i <= local_precision; i++)
29     {
30         double x = (double)rand() / RAND_MAX * 2.0 - 1.0; // Generazione punto
31         double y = (double)rand() / RAND_MAX * 2.0 - 1.0;
32         if (x * x + y * y < 1) // Controllo se il punto e' nel cerchio
33             local_circle_point++;
34     }
35
36     int total_circle_point = 0;
37     MPI_Reduce(&local_circle_point, &total_circle_point, 1, MPI_INT, MPI_SUM,
38               0, MPI_COMM_WORLD);
39
40     if (my_rank == 0)
41     {
42         double esteem = ((double)total_circle_point / precision * 4);
43         printf("Su %d precision, la stima del pi greco e' : %lf\n",
44               precision, esteem);
45     }
46
47     MPI_Finalize();
48     return 0;
49 }

```

Risultati della Computazione

Numero punti generati	Valore π stimato
100	4.08
1000	3.408
10000	3.1031
100000	3.13416
1000000	3.143736
100000000	3.141725



2.5 Valutazione del Tempo

Valutare il tempo di esecuzione di un programma multicore non è banale. MPI fornisce una funzione `double MPI_Wtime`, ritorna un valore che rappresenta il tempo passato da un certo riferimento fisso. Basta valutare questo tempo in due punti diversi del codice e farne la differenza.



```
1 double start, finish;  
2 start=MPI_Wtime();  
3 /*codice*/  
4 finish=MPI_Wtime();  
5 printf( "%d", finish-start );
```

Ogni processo, seguirà un'evoluzione dello stesso codice differente, e non contemporanea fra gli altri. Se verrà calcolato il tempo trascorso per l'esecuzione di una sezione di codice, ogni processo resituirà un tempo diverso. Il tempo totale del programma, sarà dato dal massimo dei tempi forniti da ogni processo. Si può usare l'operazione collettiva

```
1 double local_start, local_finish, local_elapsed, elapsed;  
2 local_start=MPI_Wtime();  
3 /*codice*/  
4 local_finish=MPI_Wtime();  
5 local_elapsed=local_finish-local_start;  
6 MPI_Reduce(&local_elapsed, &elapsed, 1, MPI_DOUBLE, MPI_MAX, 0, comm);  
7 printf( "%d", elapsed );
```

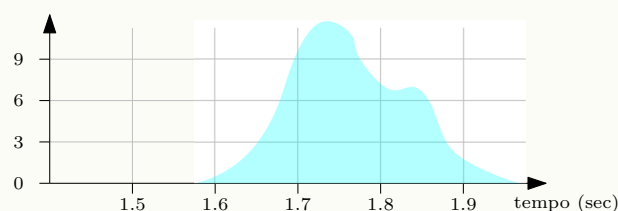
Ovviamente, tale metodo è valido con l'assunzione che tutti i processi inizino nello stesso momento (in particolare, la valutazione del tempo di inizio), un'esecuzione reale però è sfasata, e ciò porterebbe ad un tempo di esecuzione che non corrisponde a quello effettivo.

Quando si esegue un'operazione di reduce, tutti i processi devono arrivare ad uno punto comune nel codice, "sincronizzandosi", esiste un'operazione, `MPI_Barrier`, il cui unico scopo è attendere che tutti i processi la eseguano prima di continuare nell'esecuzione. Tale operazione comunque, non garantisce che i processi si sincronizzino una volta eseguita, quindi approssima il comportamento sincronizzato. Nella pratica, è molto complesso garantire tale proprietà.

La misurazione del tempo impiegato da un processo è non deterministica, in quanto quest'ultimo è soggetto alle interruzioni del sistema operativo ed ai cambi di contesto, che possono variare in maniera apparentemente aleatoria. Nel caso in cui il programma venga eseguito in rete, tale *rumore* (il tempo casuale aggiunto all'esecuzione normale di un processo) è ancora maggiore.

Generalmente, è corretto eseguire più volte un processo misurando i tempi ad ogni esecuzione, per avere una misura statistica. Il minimo corrisponderà al caso ideale, il massimo al caso peggiore, la mediana al caso più frequente, è corretto fornire i dati di ogni esecuzione per avere un quadro chiaro dei tempi effettivi. In breve per valutare il tempo

1. Si mette una funzione barriera all'inizio dell'esecuzione
2. Si trova il massimo dei tempi di ogni processo
3. Si provano diverse esecuzioni ottenendo una distribuzione



Le interferenze ed i rumori sono stati grande oggetto di studio nella valutazione dei tempi, in particolare, si è osservato un rapporto di proporzionalità diretta fra il rumore ed il numero di processi impiegati in un calcolo, esiste quindi un limite, in cui l'aumento dei nodi volti ad un calcolo non garantisce un miglioramento dei tempi di esecuzione, bensì il contrario.