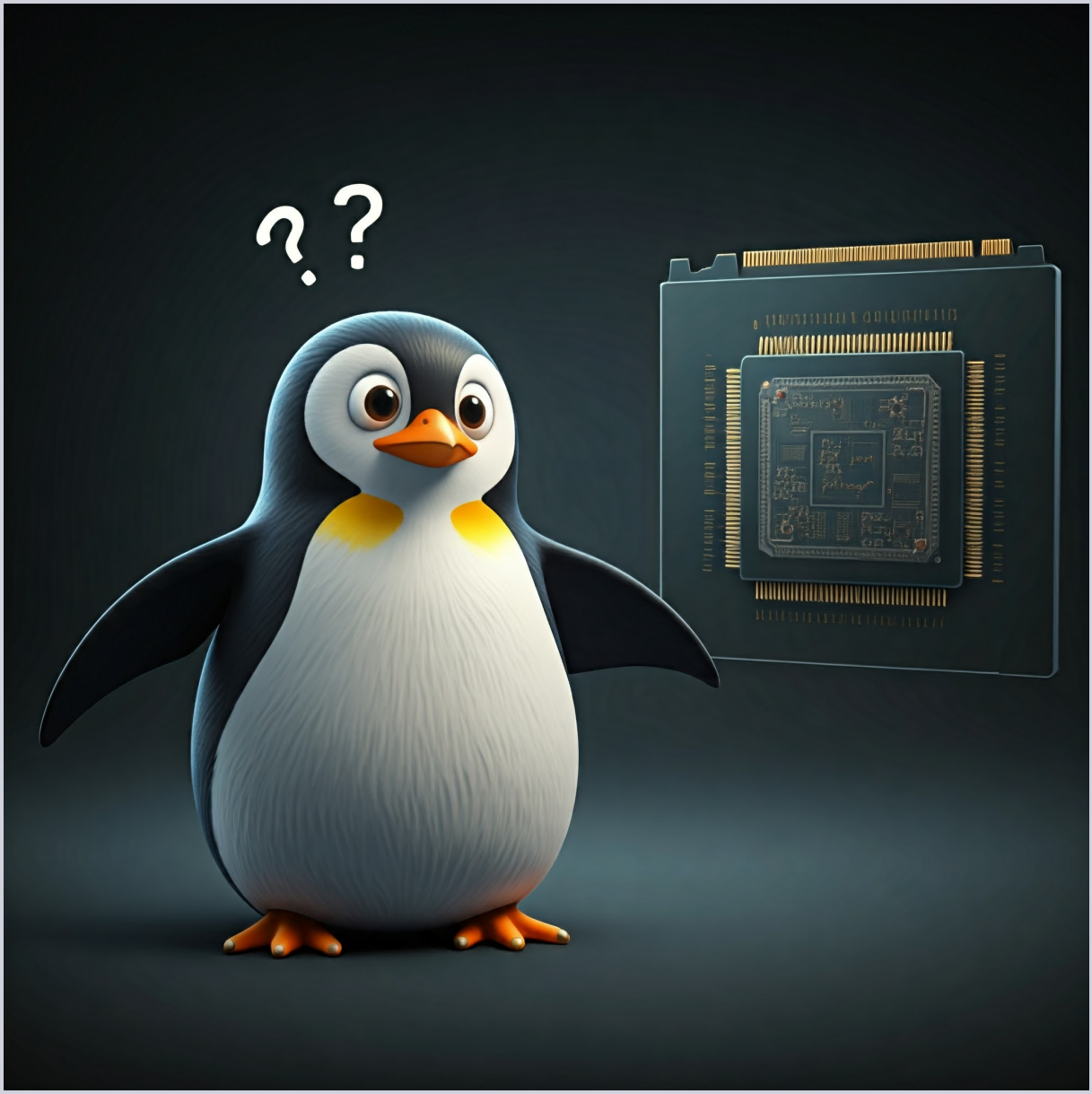
Marco Casu

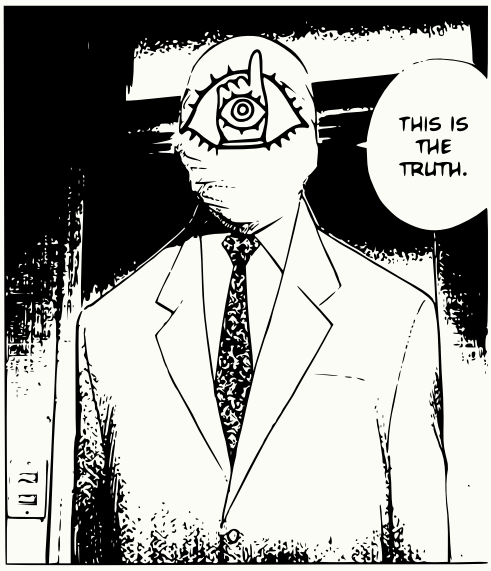
Programmazione di Sistemi Multicore



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
2.5.1 Scalabilità Forte e Scalabilità Debole . . . 22
2.6 Operazioni su Vettori e Matrici . . . 23
2.6.1 Scatter e Gather . . . 23
2.6.2 Ultime Collettive di tipo "All" . . . 27
2.7 Tipi di Dato Custom . . . 27

3 Memoria Condivisa : Posix Threads **29**
3.1 Introduzione ai Thread . . . 29
3.1.1 Prodotto Matrice-Vettore con Pthread . . . 30
3.2 Sezioni Critiche . . . 31
3.2.1 Busy Waiting e Mutex . . . 32
3.2.2 Semafori, Barriere e Variabili di Condizione . . . 33
3.2.3 Stima di π con Pthread . . . 36

CAPITOLO

1

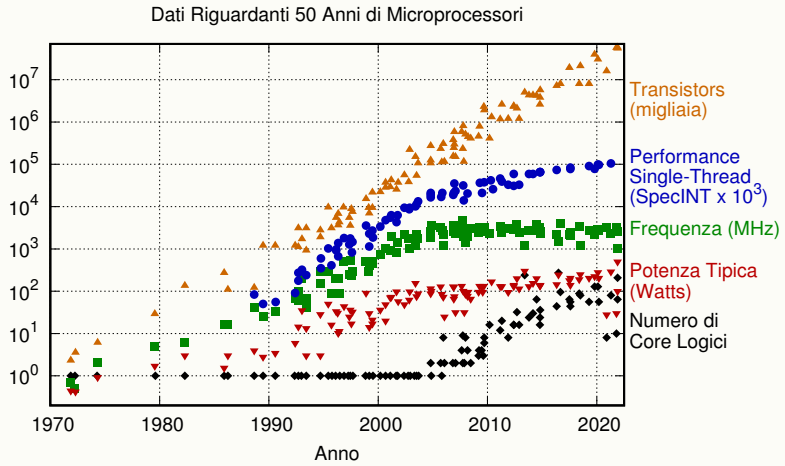
PARALLELISMO : MOTIVAZIONI

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'obbiettivo 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.

```
sum = 0;
for(i=0; i<n; i++){
    x = compute_next_value(...);
    sum += x;
}
```

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

```
local_sum = 0;
first_index = ...;
last_index = ...;
for(local_i=first_index; first_index<last_index; local_i++){
    local_x = compute_next_value(...);
    local_sum += local_x;
}
```

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 : $1,4,3, \quad 9,2,8, \quad 5,1,1, \quad 6,2,7, \quad 2,5,0, \quad 4,1,8, \quad 6,5,1, \quad 2,3,9$

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.

```
if(master){
    sum = local_sum;
    for c : core{
        if(c!=self){
            sum += c.local_sum;
        }
    }
}else{
    send local_sum to master;
}
```

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 si 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 :

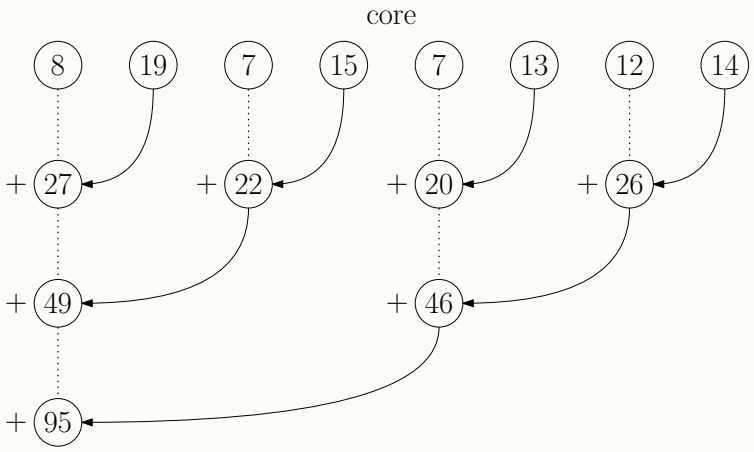


Figura 1.1: calcolo somme a coppie

- **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.

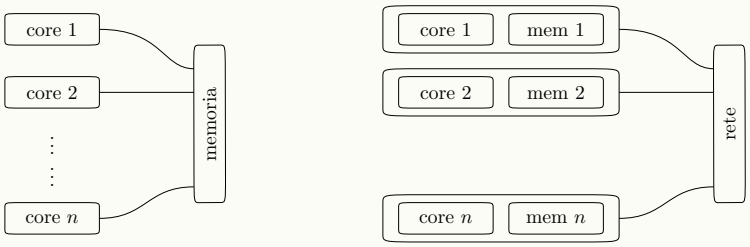


Figura 1.2: modelli di parallelismo

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 obbiettivo 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 interconnesione, (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 $1, 2 \ldots, p-1$.

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.

```c
#include <stdio.h>
#include <mpi.h>
//voglio lanciare il programma su piu unita di calcolo
int main(int argc, char **argv){
    int p = MPI_Init(NULL,NULL);
    //Il parametro in output di MPI_Init e' uno status sull'errore
    if(p == MPI_SUCCESS){

    }else{
        printf("qualcosa e' andato storto");
        MPI_Abort(MPI_COMM_WORLD,p);
        //Con MPI_Abort tutti i processi su tutti i core avviati verranno terminati
    }
    int size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    printf("hello world, im the process %d/%d",rank,size);
    MPI_Finalize(); //Serve per terminare la libreria
    return 0;
}
```

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

L'inzio dei messagi 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` :

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

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

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char **argv)
{
    int p = MPI_Init(NULL, NULL);
    // Il parametro in output di MPI_Init e' uno status sull'errore
    if (p != MPI_SUCCESS)
    {
        printf("qualcosa e' andato storto");
        MPI_Abort(MPI_COMM_WORLD, p);
        // Con MPI_Abort tutti i processi su tutti i core avviati verranno terminati
    }
    int size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    int str_size = 256;
    int rank;
```

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

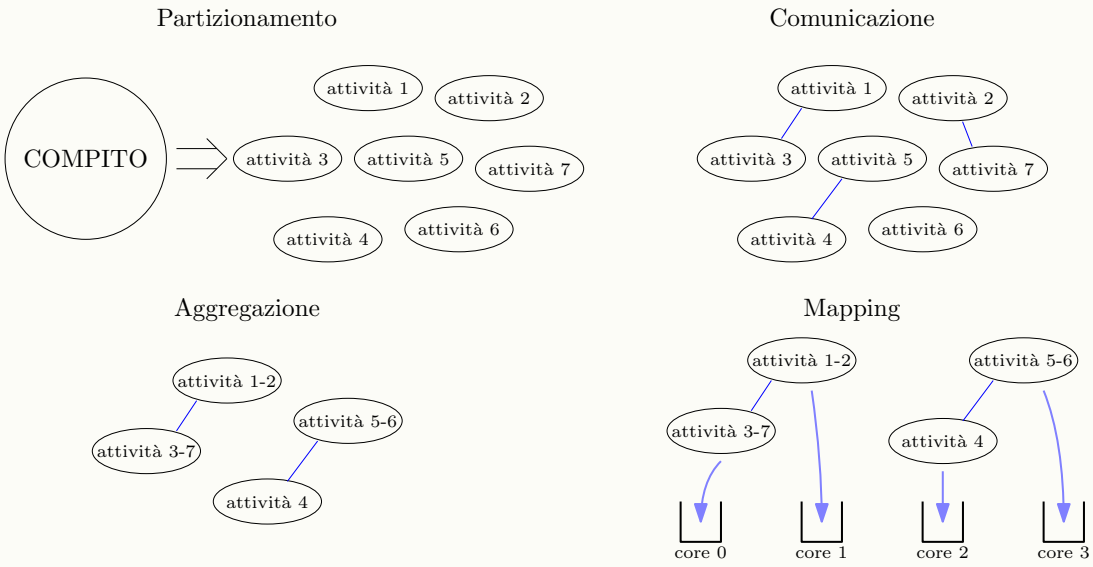


Figura 2.1: Foster's methodology

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.

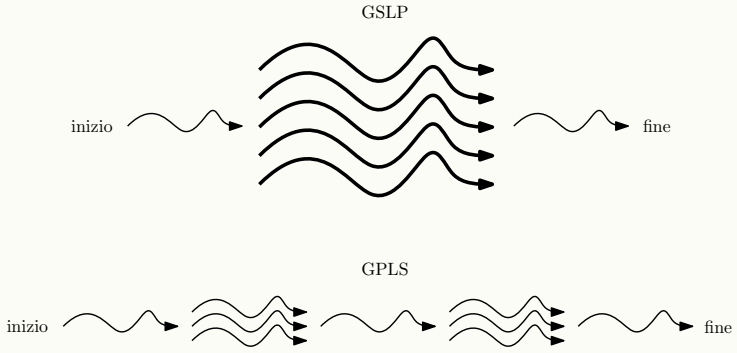


Figura 2.2: GPLS e GSLP

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 : Colleziona 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 si 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 iterazione 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 li, 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 sufficente per riceverlo.

In entrambi i casi, MPI si assicura che il messaggio da inviare non vada perso, il programmma 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 si 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 si 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.

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

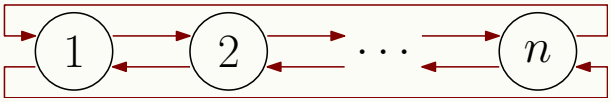


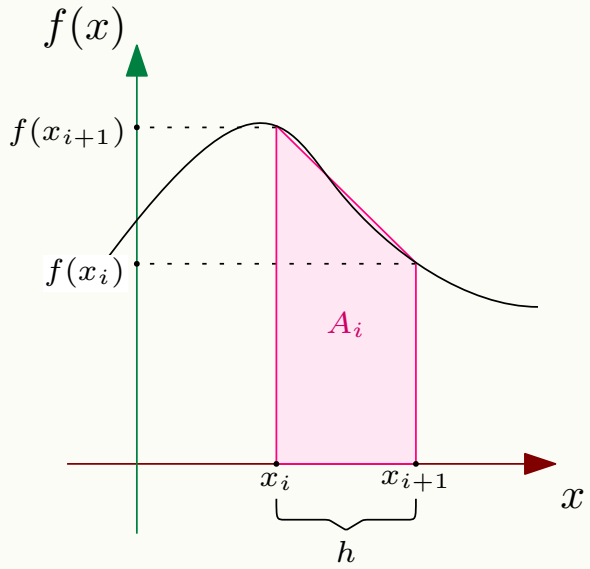
Figura 2.3: configurazione ad anello

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 trapezoide. Tale metodo consiste nel dividere l'intervallo di integrazione in n intervalli

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

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



$$h = x_{i+1} - x_i$$

$$A_i = \frac{h}{2}[f(x_i) + f(x_{i+1})]$$

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

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

```
/* Input : a, b, n */
h = (b-a)/n;
approx = (f(a)+f(b))/2;
for(i=1;i<=n-1;i++){
    x_i = a+i*h;
    approx += f(x_i);
}
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).

```
double Trap(double left, double right, int count, double base_len){

    double esitmate, x;
    // f e' la funzione integranda
    esitmate = (f(left)+f(right))/2.0;
    for(int i = 1; i<=count-1; i++){
        x=left+i*base_len;
        estimate+=f(x);
    }
    return estimate*base_len;
}
```

```c
int main(void){

    int my_rank;
    int comm_size;
    int n = 1024; //numero di intervalli, piu' e' grande, piu' la stima sara' precisa
    double a = 0.0; //estremo sinistro di integrazione
    double b = 3.0; //estremo destro di integrazione
    double h; //lunghezza intervalli;
    double local_a, local b;
    double local_sum;
    double total_sum;
    int source;

    MPI_Init(NULL,NULL);
    MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
    MPI_Comm_size(MPI_COMM_WORLD,&comm_size);

    h=(b-a)/n;
    local_n = n/comm_size; //numero di trapezoidi per ogni processo

    local_a = a+my_rank*local_n*h;
    local_b=local_a+local_n*h;
    local_sum = Trap(local_a, local_b, local_n, h); //calcolo somma parziale

    if(my_rank!=0){
        //Invio la somma parziale al processo con rank 0
        MPI_Send(&local_sum, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
    }
    else{
        total_sum = local_sum;
        for(source = 1;source<comm_sz; source++){
            MPI_Recv(&local_sum, 1, MPI_DOUBLE, source, 0,
                        MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            total_sum+=local_sum
        }

        printf("con n = %d trapezoidi, la somma approssimata della funzione\n
               da %f a %f e' %.15e .\n", n, a, b, total_sum);
    }

    MPI_Finalize();
    return 0;
}
```

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.

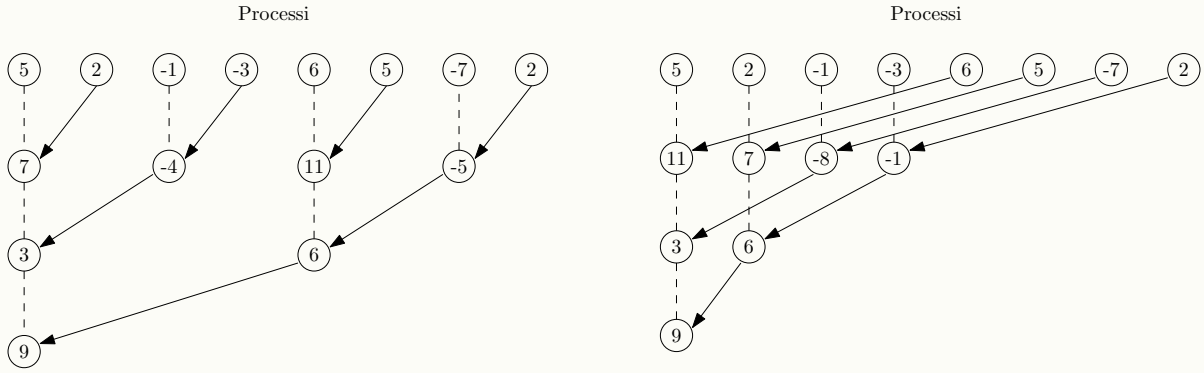


Figura 2.4: alberi differenti (entrambi validi)

Si può pensare di suddividere il carico di lavoro ad albero, come già visto in figura 1.1, facendo si 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 reta 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
`MPI_MAX`	Massimo
`MPI_MIN`	Minimo
`MPI_SUM`	Somma
`MPI_PROD`	Prodotto
`MPI_LAND`	AND logico
`MPI_BAND`	AND bit a bit
`MPI_LOR`	OR logico
`MPI_BOR`	OR bit a bit
`MPI_LXOR`	XOR logico
`MPI_BXOR`	XOR bit a bit
`MPI_MAXLOC`	Massimo insieme al suo indice
`MPI_MINLOC`	Minimo insieme al suo indice

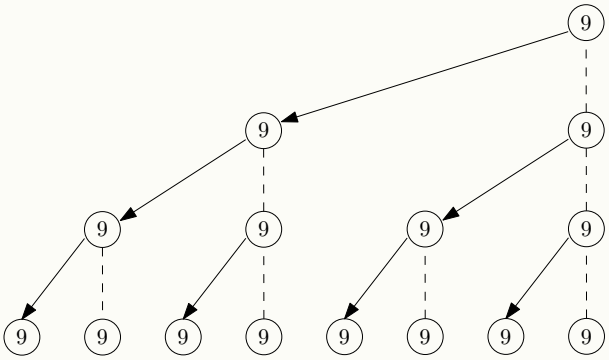
È possibile anche definire delle operazioni personalizzate tramite la chiamata `MPI_Op_create`.

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 si che il processo 0 condivida i dati con due altri processi, e questi due li condividano 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.

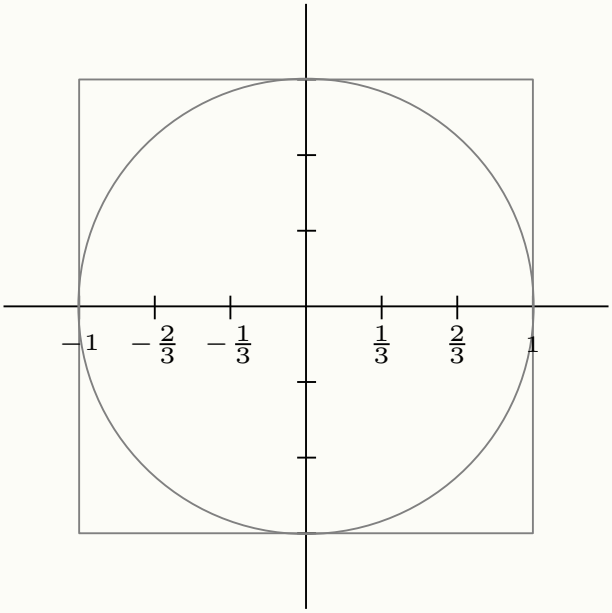
Le operazioni collettive sono diventate particolarmente importanti nell'ultimo periodo in quanto sono utilizzate nella stragrande maggioranza dei programmi paralleli che vengono eseguiti per il training delle reti neurali odierne. Le grosse aziende di informatica, hanno iniziato a produrre delle proprie 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 $\pi r^2 \implies \pi$, 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

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

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, $4\dfrac{n}{c}$ 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.

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <time.h>

int main(int argc, char **argv)
{

    int precision = 1000; // Numero di punti generati casualmente

    if (argc > 1)
    {
        precision = atoi(argv[1]);
    }

    MPI_Init(NULL, NULL);
    srand(time(NULL));

    int my_rank;
    int my_size;
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &my_size);

    int local_precision = precision / my_size; /* Numero di punto da generare per
                                                  ogni processo */
    int local_circle_point = 0;

    for (int i = 0; i <= local_precision; i++)
    {
        double x = (double)rand() / RAND_MAX * 2.0 - 1.0; // Generazione punto
        double y = (double)rand() / RAND_MAX * 2.0 - 1.0;
        if (x * x + y * y < 1) // Controllo se il punto e' nel cerchio
            local_circle_point++;
    }

    int total_circle_point = 0;
    MPI_Reduce(&local_circle_point, &total_circle_point, 1, MPI_INT, MPI_SUM,
                                                    0, MPI_COMM_WORLD);

    if (my_rank == 0)
    {
        double esteem = ((double)total_circle_point / precision * 4);
        printf("Su %d precision, la stima del pi greco e' : %lf\n",
                                                precision, esteem);
    }

    MPI_Finalize();
    return 0;
}
```

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.

```
double start,finish;
start=MPI_Wtime();
/*codice*/
finish=MPI_Wtime();
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

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

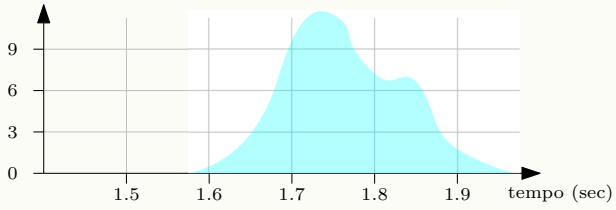
Ovviamente, tale metodo è valido con l'assunzione che tutti i processi inizino nello stesso momento (in particolare, la valutazione del tempo di inizio), un'esecuzione reale però è sfasata, e ciò porterebbe ad un tempo di esecuzione che non corrisponde a quello effettivo.

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.

La seguente tabella, raccoglie i tempi di esecuzione in secondi di un algoritmo che esegue il prodotto fra matrici quadrate.

tempo (sec)	ordine della matrice				
numero processi	1024	2048	4096	8192	16384
1	4.1	16	64	270	1100
2	2.3	8.5	33	140	560
4	2	5.1	18	70	280
8	1.7	3.3	9.8	36	140
16	1.7	2.6	5.9	19	71

Ovviamente, con l'aumento delle dimensioni dell'input, aumenta anche il tempo di esecuzione, e quest'ultimo decresce con l'aumentare del numero di processi. Questo fattore di proporzionalità inversa non è però valido per un arbitrario numero di processo, de facto, ci sarà un limite per cui vale, ad un certo punto, l'aumentare dei processi non migliorerà il tempo di esecuzione. Per la stima dei tempi si definiscono le seguenti variabili

- $T_s(n)$ è il tempo di esecuzione di un programma se eseguito in maniera sequenziale, con un input di dimensione n.
- $T_p(n,p)$ è il tempo di esecuzione di un programma se eseguito in maniera parallela con p processi, e con un input di dimensione n.
- $S(n,p) = \dfrac{T_s(n)}{T_p(n,p)}$ è detto **speed up** dell'applicazione e misura il margine di miglioramento di un programma quando si esegue in parallelo piuttosto che in sequenziale.

Ovviamente nel rapporto dello speed up i tempi sequenziali e paralleli vanno misurati sullo stesso hardware. L'ideale sarebbe uno speed up lineare, nell'effettivo, l'aumentare dei processi fa diminuire lo speed up, invece l'aumentare dell'input lo fa aumentare. La seguente tabella riporta i valori dello speed up del programma che esegue il prodotto fra matrici.

speed up	ordine della matrice				
numero processi	1024	2048	4096	8192	16384
1	1	1	1	1	1
2	1.8	1.9	1.9	1.9	2
4	2.1	3.1	3.6	3.9	3.9
8	2.4	4.8	6.5	7.5	7.9
16	2.4	6.2	10.8	14.2	15.5

Nota bene : $T_p(n,1) \neq T_s(n)$, generalmente il tempo di esecuzione parallela con 1 processo è maggiore del tempo di esecuzione sequenziale (dato che il setup dell'ambiente per il calcolo parallelo ha un costo).

La **scalabilità** di un processo è definita come segue

$$Sc(n,p) = \frac{T_p(n,1)}{T_p(n,p)}$$

L'**efficienza** invece è un valore compreso fra zero ed 1 definito come segue

$$E(n,p) = \frac{S(n,p)}{p} \in (0,1]$$

efficienza	ordine della matrice				
numero processi	1024	2048	4096	8192	16384
1	1	1	1	1	1
2	0.89	0.94	0.97	0.96	0.98
4	0.51	0.78	0.89	0.96	0.98
8	0.3	0.61	0.82	0.94	0.98
16	0.15	0.39	0.68	0.89	0.97

Chiaramente, se le dimensioni dell'input sono modeste, l'utilizzo di tanti processi risulta inutile, l'efficienza è quindi bassa. L'utilizzo del multi processo ha senso quando l'input è di grande dimensione, e si presta ad essere diviso e computato da nodi paralleli.

2.5.1 Scalabilità Forte e Scalabilità Debole

Un programma si dice **strongly scalable** (fortemente scalabile) se, data una dimensione fissa dell'input n, lo speed up ha una buona crescita all'aumentare dei processi. È invece **weakly scalable** (debolmente scalabile) se, l'aumentare dell'input, e l'aumentare del numero dei processi, il tempo di esecuzione varia di poco.

Rendere un applicazione strongly scalable è estremamente difficile, per questo nell'effettivo viene considerato sempre il weak scaling quando si vuole parallelizzare un compito. È possibile avere una stima approssimativa di quanto un'applicazione possa scalare? Dato un programma, si definisce **frazione seriale** la porzione di esso che è impossibile da parallelizzare, e viene espressa come un valore fra 0 ed 1. Tale parte non potrà essere influenzata dalla parallelizzazione, la **legge di Amdahl** stabilisce che lo scaling di un'applicazione è limitato dalla frazione seriale. Per un n dimensione in input fissata, si ha

$$T_p(p) = (1-\alpha)T_s + \alpha\frac{T_s}{p}$$

Dove $\alpha \in [0,1]$ rappresenta la frazione parallelizzabile, e $1-\alpha$ la frazione seriale. Lo speed up sarà quindi

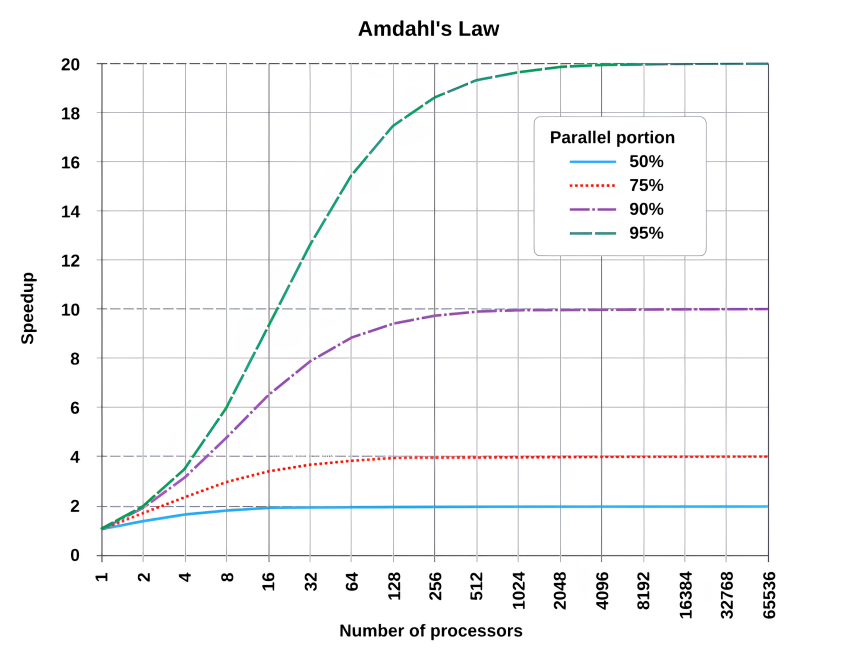
$$S(p) = T_s\frac{1}{(1-\alpha)T_s + \alpha\frac{T_s}{p}}$$

Il valore dello speed up è limitato superiormente, infatti all'aumentare dei processi :

$$\lim_{p\to\infty} S(p) = \tag{2.1}$$

$$\lim_{p\to\infty} T_s\frac{1}{(1-\alpha)T_s + \alpha\frac{T_s}{p}} = \tag{2.2}$$

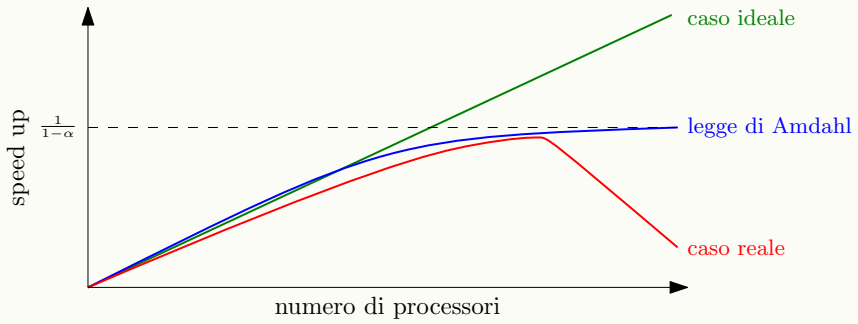
$$\frac{T_s}{(1-\alpha)T_s} = \frac{1}{1-\alpha} \tag{2.3}$$



Questa legge non tiene conto del week scaling e decreta un comportamento ideale/approssimato. La **legge di Gustafson** tiene conto del week scaling

$$S(n,p) = (1-\alpha) + \alpha p$$

Nel caso reale, l'aumentare del numero di processi potrebbe far incrementare la frazione seriale.



2.6 Operazioni su Vettori e Matrici

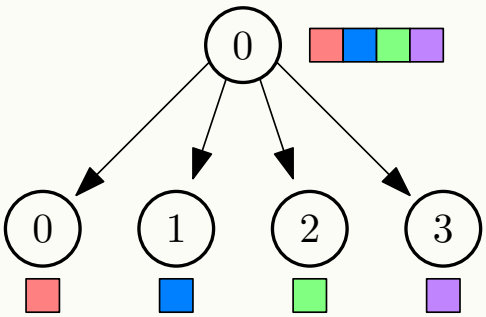
2.6.1 Scatter e Gather

Consideriamo un algoritmo sequenziale che calcoli la somma di due vettori dati in input

```
void vector_sum(double x[], double y[], double z[], int n){
    for(int i=0; i<n; i++){
        z[i]=x[i]+y[i];
    }
}
```

Come si può distribuire tale compito su diversi processi? Si da l'assunzione che i vettori da sommare siano stati letti da standard input, quindi sono presenti nella memoria del processo con rank 0. Si vuole dividere il vettore equamente fra i diversi processi.

A tal proposito, esiste la funzione collettiva `int MPI_Scatter`, che si occupa di prendere un vettore e di dividerlo equamente fra tutti i processi di un dato comunicatore.

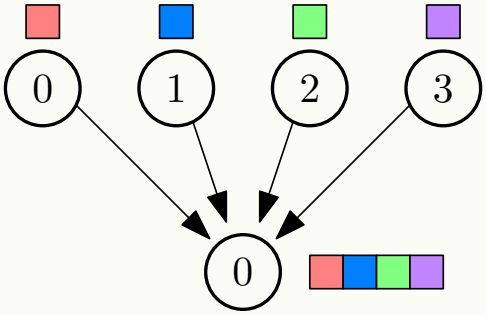


I parametri della funzione sono i seguenti

- `void *send_buf_p` il buffer contenente il vettore da dividere
- `int send_count` il numero di elementi da inviare ad ogni processo, non il numero di elementi totali del vettore
- `MPI_Datatype send_type` il tipo degli elementi del vettore
- `void *recv_buf_p` il buffer che conterrà la frazione di vettore ricevuta, per il mittente, può coincidere con il buffer di invio
- `int recv_count` analogo a `send_count`
- `MPI_Datatype recv_type` analogo a codeesend_type
- `int src_proc` il rank del processo che condividerà il vettore
- `MPI_Comm comm` il comunicatore in questione

La funzione assume che il numero di elementi da condividere sia divisibile per il numero di processi, l'ordine di condivisione avviene in base al rank. Il nodo che condivide il vettore può specificare la macro `MPI_IN_PLACE` come buffer di destinazione, e manterrà i valori nel vettore originale.

La funzione collettiva `int MPI_Gather` è l'inversa della `scatter`, ogni nodo definisce un vettore, e ne verrà eseguita la concatenazione, per poi essere inviata ad un processo.



L'ordine di concatenazione è sempre dato dal rank. I parametri sono pressocché identici alla `scatter`

- `void *send_buf_p` il buffer contenente il vettore parziale da inviare
- `int send_count` il numero di elementi contenuti nel vettore parziale
- `MPI_Datatype send_type` il tipo degli elementi del vettore
- `void *recv_buf_p` il buffer che conterrà il vettore finale composta dalla concatenazione dei vettori parziali
- `int recv_count` analogo a `send_count`
- `MPI_Datatype recv_type` analogo a codeesend_type
- `int dest_proc` il rank del processo che riceverà il vettore
- `MPI_Comm comm` il comunicatore in questione

Esempio di codice : Il seguente programma, fa si che il processo di rank 0 generi due vettori con valori casuali, che verranno divisi fra i vari processi che eseguiranno delle somme parziali, trovando sezioni parziali del vettore somma, che verrà poi ri-unito con l'operazione collettiva.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <mpi.h>

void print_vector(int n, int *v, char name)
{
    printf("%c = [ ", name);
    for (int i = 0; i < n - 1; i++)
    {
        printf("%d, ", v[i]);
    }
    printf("%d ]\n", v[n - 1]);
}

int main(int argc, char **argv)
{
    MPI_Init(NULL, NULL);

    int my_rank;
    int size;
    int local_size = 3;
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int local_vector[size];
    int local_vector2[size];
```

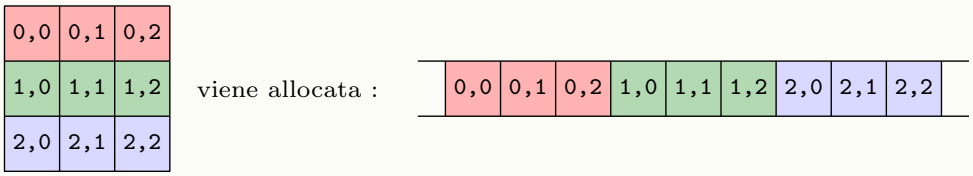
```
27      MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
28
29      if (my_rank == 0)
30      {
31
32          int inputVector[local_size * size];
33          int inputVector2[local_size * size];
34          srand(time(NULL));
35          for (int i = 0; i < local_size * size; i++)
36          {
37              inputVector[i] = rand() % 100;
38              inputVector2[i] = rand() % 100;
39          }
40          print_vector(local_size * size, inputVector, 'A');
41          print_vector(local_size * size, inputVector2, 'B');
42          printf("\n");
43          MPI_Scatter(inputVector, local_size, MPI_INT, local_vector, local_size,
44                      MPI_INT, 0, MPI_COMM_WORLD);
45          MPI_Scatter(inputVector2, local_size, MPI_INT, local_vector2, local_size,
46                      MPI_INT, 0, MPI_COMM_WORLD);
47      }
48      else
49      {
50          MPI_Scatter(NULL, local_size, MPI_INT, local_vector, local_size,
51                      MPI_INT, 0, MPI_COMM_WORLD);
52          MPI_Scatter(NULL, local_size, MPI_INT, local_vector2, local_size,
53                      MPI_INT, 0, MPI_COMM_WORLD);
54      }
55      printf("process %d :\n     A_%d = [ ", my_rank, my_rank);
56      for (int i = 0; i < local_size - 1; i++)
57      {
58          printf("%d, ", local_vector[i]);
59      }
60      printf("%d ]\n     B_%d = [", local_vector[local_size - 1], my_rank);
61      for (int i = 0; i < local_size - 1; i++)
62      {
63          printf("%d, ", local_vector2[i]);
64      }
65      printf("%d ]\n A_%d+B_%d = [", local_vector2[local_size - 1], my_rank, my_rank);
66      for (int i = 0; i < local_size - 1; i++)
67      {
68          printf("%d, ", local_vector2[i] + local_vector[i]);
69      }
70      printf("%d ]\n\n", local_vector2[local_size - 1] + local_vector[local_size - 1]);
71
72      int vec_sum[local_size];
73      for (int i = 0; i < local_size; i++)
74      {
75          vec_sum[i] = local_vector2[i] + local_vector[i];
76      }
77
78      if (my_rank == 0)
79      {
80          int outVector[local_size * size];
81          MPI_Gather(vec_sum, local_size, MPI_INT, outVector, local_size,
82                     MPI_INT, 0, MPI_COMM_WORLD);
83          print_vector(local_size * size, outVector, 'O');
84      }
85      else
86      {
87          MPI_Gather(vec_sum, local_size, MPI_INT, NULL, local_size,
88                     MPI_INT, 0, MPI_COMM_WORLD);
89      }
90
91      MPI_Finalize();
92
93      exit(0);
94  }
```

Le operazioni collettive sui vettori possono essere eseguite anche sulle matrici, anche se ciò dipende dall'allocazione di esse. Quando si dichiara una matrice di dimensioni statiche nel seguente modo

```
int matrix[n][n];
```

essa viene allocata in maniera contigua in memoria.

`int matrix[3][3];`



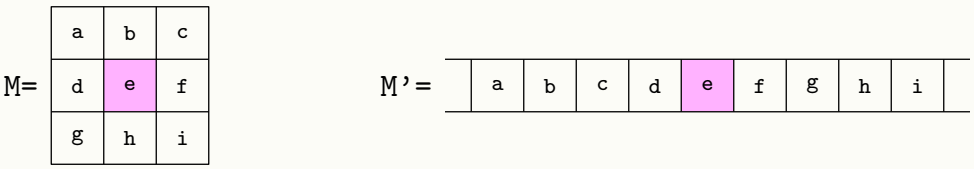
In tal caso, le operazioni collettive di *reduce* funzioneranno senza problemi dato che agiscono su blocchi di memoria contigui. Il punto è che una dichiarazione di questo tipo ha senso esclusivamente se è nota a priori la dimensione della matrice, nel caso più generale, una matrice si dichiara dinamicamente come un puntatore di puntatori di interi.

```
int** a;
a = (int**) malloc(sizeof(int*)*num_rows);
for(int i = 0; i < num_rows; i++){
    a[i] = (int*) malloc(sizeof(int)*num_cols);
}
```

Le righe della matrice in questo caso *non* sono contigue in memoria, non è quindi possibile utilizzare operazioni di *reduce*, per far ciò, bisogna collassare la matrice in un array bidimensionale regolando l'accesso degli indici. Su una matrice statica, l'accesso avviene per mezzo di due indici `matrix[i][j]`, se la matrice è collassata in un array, un solo indice deve codificare entrambi gli indici della matrice, nella pratica, si fa così

```
matrix[i*num_col+j]
```

Una matrice allocata dinamicamente come un unico array può essere soggetta alle operazioni di *reduce*.



`M[1][1]=e=M[1*3+1]=M[4]`

Prodotto fra Vettori e Matrici

Come visto nel corso di Algebra, il prodotto scalare fra due vettori è definito come segue

$$\begin{bmatrix} a_1 & a_2 & \dots & a_n \end{bmatrix} \cdot \begin{bmatrix} b_1 \\ b_2 \\ \dots \\ b_n \end{bmatrix} = a_1 \cdot b_1 + a_2 \cdot b_2 + \dots + a_n \cdot b_n$$

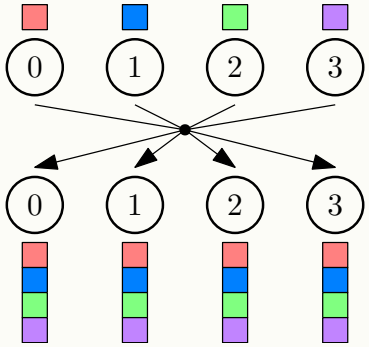
Il prodotto fra una matrice $n \times n$ ed un vettore $\bar{b}$ di n componenti, è un vettore sempre di n componenti, di cui ogni i-esimo componente è il prodotto scalare fra la i-esima riga della matrice ed il vettore $\bar{b}$.

$$\begin{bmatrix} a_{1_1} & a_{1_2} & \dots & a_{1_n} \\ a_{2_1} & a_{2_2} & \dots & a_{2_n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n_1} & a_{n_2} & \dots & a_{n_n} \end{bmatrix} \cdot \begin{bmatrix} b_1 \\ b_2 \\ \dots \\ b_n \end{bmatrix} = \begin{bmatrix} a_{1_1}b_1 + a_{1_2}b_2 + \dots + a_{1_n}b_n \\ a_{2_1}b_1 + a_{2_2}b_2 + \dots + a_{2_n}b_n \\ \vdots \\ a_{n_1}b_1 + a_{n_2}b_2 + \dots + a_{n_n}b_n \end{bmatrix}$$

```
int A[n][n]; //matrice
int b[n]; //vettore input
int y[n]; //vettore risultato
for(i=0;i<n;i++){
    for(j=0;j<n;j++){
        y[i]=A[i][j]*b[j] //prodotto scalare implicitamente definito
    }
}
```

2.6.2 Ultime Collettive di tipo "All"

La collettiva `int MPI_Allgather` si occupa di unire i dati di un vettore come la normale funzione di *gather*, per poi condividere i dati con tutti i processi di un comunicatore, logicamente, equivale ad una gather seguita da una broadcast, ma è implementata in maniera più efficiente.



- `void *send_buf_p`
- `int send_count`
- `MPI_Datatype send_type`
- `int recv_count`
- `MPI_Datatype recv_type`
- `MPI_Comm comm`

La collettiva `MPI_Allscatter` equivale ad una reduce seguita da una scatter.

$$\begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} = \text{Allscatter} \Rightarrow \begin{bmatrix} a_1+b_1+c_1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ a_2+b_2+c_2 \\ \vdots \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ \vdots \\ a_n+b_n+c_n \end{bmatrix}$$

2.7 Tipi di Dato Custom

Abbiamo visto come MPI implementi il tipo `MPI_Datatype` per riferirsi ai tipi di dato presenti nel C. In C, è però possibile creare dei tipi di dato aggiuntivi (struct).

```c
struct t{
    float a,
    float b,
    int n
};
```

Cosa va definito nel capo `data_type` quando si intente inviare tramite una funzione MPI una struttura? MPI permette di creare dei tipi custom, è possibili definire **tipi derivati**, che equivalgano (in quanto disposizione in memoria) alla struct in questione. Un tipo derivato non è altro che una sequenza di datatype di base alla quale è assengata una certa disposizione in memoria.

`(MPI_DOUBLE,0), (MPI_DOUBLE,16),(MPI_INT,24)`

Un tipo derivato va definito tramite la funzione `int MPI_Type_create_struct` :

- `int count` il numero di elementi distinti nel tipo derivato

- `int array_of_blocklengths[]` ogni elemento potrebbe essere un vettore, va quindi specificata la lunghezza di ogni elemento della struct.
- `MPI_Aint array_of_displacements` la disposizione degli elementi della struct in memoria
- `MPI_Datatype *new_type_p` il tipo degli elementi della nuova struttura

Per ottenere la disposizione è possibile usare il puntatore del valore, è pero auspicabile utilizzare la funzione di liberia `MPI_Get_address` , in quanto in alcuni sistemi (seppur pochi) i puntatori potrebbero non rappresentare un indirizzo di memoria.

```
MPI\_Aint a_addr, b_addr, n_addr;
MPI_Get_address(&a, &a_addr);
array_of_displacements[0]=0;
MPI_Get_address(&b, &b_addr);
array_of_displacements[1]=b_addr-a_addr;
MPI_Get_address(&n, &n_addr);
array_of_displacements[2]=n_addr-a_addr;
```

Una volta creata tale struttura, bisogna eseguire la funzione `MPI_Type_commit` che, eseguendo opportune ottimizzazioni, decide il metodo di salvataggio della struttura in memoria. Quando non è più necessario quel datatype, si può chiamare la funzione `MPI_Type_free` .

CAPITOLO

3

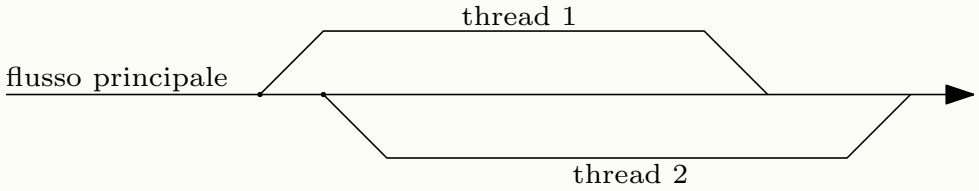
MEMORIA CONDIVISA : POSIX THREADS

3.1 Introduzione ai Thread

In MPI, un insieme di nodi comunica tramite lo scambio di messaggi su un apposito canale, all'interno di ogni nodo, sono presenti delle unità di calcolo con più cores, ossia più sotto-unità che comunicano ed operano contemporaneamente su una memoria condivisa.

Se MPI si occupa della comunicazione fra più nodi, è necessario un paradigma volto alla comunicazione dei differenti core di un'unità di calcolo che possono accedere ad una memoria comune.

Un processo è un istanza di un programma (in esecuzione) che viene caricata sulla CPU, possiamo identificare i *singoli flussi indipendenti di computazione* di un processo, denominati **thread**, analoghi ai processi ma più "leggeri" nella loro creazione e gestione, due thread appartenenti ad uno stesso processo hanno differenti program counter e stack ma condividono lo stesso spazio di indirizzamento dinamic (heap), la comunicazione è semplice, in quanto basta scrivere sulla stessa area di memoria.



Lo standard **Posix Threads (Pthreads)** è implementato con una libreria in C e specifica l'API per l'interazione fra i thread, è necessario specificare nel codice la direttiva di inclusione

```
# include <pthread.h>
```

La libreria fornisce il tipo di dato `pthread_t`, non è importante la sua struttura interna, basti sapere che tale tipo identifica univocamente un thread all'interno di un processo, a volte viene denotato *thread handler*. La creazione di un thread avviene tramite la chiamata di libreria `int pthread_create`, i cui parametri sono

- `pthread_t* thread_p` verrà assegnato l'identificatore del thread creato.
- `const pthread_attr_t* attr_p` attributi in input che per adesso verranno ignorati.

- `void* (*start_routine ) ( void* )` la funzione che verrà eseguita dal thread creato.
- `void* arg_p` i parametri della funzione sopracitata.

All'avvio di un thread è quindi necessario specificarne la funzione da eseguire, una funzione può essere passata in input in quanto rappresenta un puntatore alla zona di memoria in cui è contenuta la procedura da eseguire.

```
void func(int a){
    /*codice*/
}

void(*func_ptr)(int)=func; /*puntatore a funzione in cui se ne specifica anche il tipo
                           [int] del parametro*/
```

Convenzionalmente, la funzione del thread è di tipo puntatore a void ed ha come parametro di ingresso un puntatore a void (caso più generale possibile) che verrà eventualmente castato.

```
void *thread_function(void *args_p);
```

Per attendere la terminazione di un thread da parte del processo principale è necessario utilizzare la funzione `pthread_join`, valida per un singolo thread, i parametri sono

- `pthread_t thread` l'identificatore del thread di cui attendere la terminazione
- `void **value_ptr` il valore di ritorno del thread

La funzione `pthread_self` ritorna l'identificatore del thread chiamante. Il seguente esempio mostra la creazione di diversi thread che stampano a schermo una stringa.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define THREAD_COUNT 4

void *Hello(void *rank)
{
    int my_rank = rank;
    printf("hello from thread %ld\n", my_rank);
    return NULL;
}

int main()
{
    pthread_t thread_handles[THREAD_COUNT];
    for (int i = 0; i < THREAD_COUNT; i++) /*creazione dei thread*/
        pthread_create(&thread_handles[i], NULL, Hello, (void *)i);
    printf("hello from main thread\n");
    for (int i = 0; i < THREAD_COUNT; i++) /*terminazione dei thread*/
        pthread_join(thread_handles[i], NULL);
    return 0;
}
```

Riguardo il numero dei thread, sarebbe corretto considerare tanti thread quanti sono i core fisici sulla macchina che esegue il programma. Se si vogliono passare *molteplici* parametri ad un thread è necessario definire un'apposita struct che li comprenda.

3.1.1 Prodotto Matrice-Vettore con Pthread

Si vuole eseguire il calssico algoritmo di moltiplicazione di una matrice per un vettore, la cui computazione seriale si ricorda essere

```
/* m := righe di A*/
/* n := elementi di di x*/
/* y := vettore output*/
for(int i = 0;i<m;i++){
    y[i]=0.0;
    for(int j=0;j<n;j++){
        y[i]+=A[i][j]*x[j];
    }
}
```

Si vuole parallelizzare l'esecuzione fra più thread, come prima cosa, si supponga di avere tanti thread quante sono le righe della matrice in input (quindi, gli elementi del vettore in output), e di far calcolare ad ogni thread i il valore dell'i-esimo elemento del vettore in output.

$$\begin{vmatrix} a_{1_1} & \dots & \dots & a_{1_n} \\ \vdots & & \ddots & \vdots \\ a_{m_1} & \dots & \dots & a_{m_n} \end{vmatrix} \times \begin{vmatrix} x_1 \\ \vdots \\ x_n \end{vmatrix} = \begin{vmatrix} y_1 & \text{thread 1} \\ \vdots & \vdots \\ y_m & \text{thread } m \end{vmatrix}$$

In generale, se il numero di thread è $t < m$, ogni q-esimo thread computerà $\frac{m}{t}$ righe, ossia dalle

- $q \cdot {}^{m}/_{t}$-esima riga
- alla $(q+1){}^{m}/_{t} - 1$-esima riga

Si ha il seguente codice

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define MAT_ORDER 6
#define THREAD_COUNT 6

/*Siano A la matrice, x il vettore in input ed y il vettore in output*/

void *mat_vec_product(void *rank)
{
    int my_rank = rank;
    int local_m = MAT_ORDER / THREAD_COUNT;
    int first_row = my_rank * local_m;
    int last_row = ((my_rank + 1) * local_m) - 1;

    for (int i = first_row; i <= last_row; i++)
    {
        y[i] = 0;

        for (int j = 0; j < MAT_ORDER; j++)
        {
            y[i] += A[i][j] * x[j];
        }
    }
}

int main()
{
    pthread_t handler[THREAD_COUNT];
    for (int i = 0; i < THREAD_COUNT; i++)
        pthread_create(&handler[i], NULL, mat_vec_product, (void *)i);
    for (int i = 0; i < THREAD_COUNT; i++)
        pthread_join(handler[i], NULL);
    return 0;
}
```

3.2 Sezioni Critiche

Si consideri il seguente scenario : Vi è una variabile (intera) `x` inizializzata a 0 condivisa da due thread, entrambi si adoperano per incrementare la variabile di 1, ci si aspetta, che dopo l'esecuzione la variabile `x` sia uguale a 2. Il punto è che l'incremento di una variabile (somma) in C, equivale ad una sequenza di istruzioni assembly che possono essere intergogliate dai due thread.

istanti di tempo	thread 0	thread 1
1	inizalizzato dal main thread	
2	legge la variabile `x=0` e la carica nel registro r1	inizalizzato dal main thread
3	incrementa il registro r1=0 di 1, avendo r1=1	legge la variabile `x=0` e la carica nel registro r2
4	salva il valore del registro in `x`, si ha `x=1`	incrementa il registro r2=0 di 1, avendo r2=1
5	termina	salva il valore del registro in `x`, si ha `x=1`
6		termina

Alla fine dell'esecuzione la variabile `x` sarà uguale ad 1 perché i due thread non si sono sincronizzati correttamente. Tale problema è noto come **race condition** ed ha comportato la nascita di una vasta teoria riguardante la sincronizzazione dei processi paralleli che accedono a risorse condivise, in questa sezione verranno presentate delle possibili soluzioni.

3.2.1 Busy Waiting e Mutex

Il tema centrale è la mutua esclusività delle variabili, l'accesso ad esse non può venire contemporaneamente da parte di più thread, con **busy waiting**, si denota una metodologia che consiste nel far attendere un processo che tenta di accedere ad una variabile già in uso, lasciandolo bloccato in un ciclo in cui ad ogni iterazione controlla la disponibilità della risorsa.

Nel seguente esempio, ogni thread viene bloccato in un ciclo finché la variabile `flag` non diventa uguale al loro rank. A seguito dell'utilizzo dovranno incrementare la variabile per permettere al prossimo thread di accedere.

```
1    while(flag!=my_rank);
2    x+=1;
3    flag++;
```

Il busy waiting, seppur semplice nella sua implementazione, comporta svantaggi difficilmente trascurabili

- un processo in attesa occuperà inutilmente la CPU comportando uno spreco di computazione
- Il compilatore, se nota un indipendenza apparente fra due istruzioni, potrebbe invertirne l'ordine, ad esempio, trasformando il codice come segue

```
1    x+=1;
2    while(flag!=my_rank); /* sequenza di istruzioni errata */
3    flag++;
```

Tale riarrangiamento è parte di una serie di ottimizzazioni che fa il compilatore. Disattivare tali ottimizzazioni comporterebbe un uso meno efficiente dei registri per tutte le altre istruzioni.

Le **Mutex** permettono una gestione più sofisticata degli accessi alle risorse mutualmente esclusive, una Mutex è una variabile utilizzata per restringere l'accesso ad una risorsa, per poi rilasciarla una volta utilizzata.

Una mutex viene gestita dal tipo di dato `pthread_mutex_t`, una volta definita, va inizializzata tramite la funzione `int pthread_mutex_init`, assegnandole la variabile il cui accesso deve essere regolamentato. La funzione ha i seguenti parametri

- `pthread_mutex_t *mutex_p` la mutex a cui si riferisce
- `const pthread_mutexattr_t *attr_p` degli attributi che per adesso saranno ignorati.

Quando si vuole accedere ad una variabile protetta da una mutex bisogna controllarne lo stato, che può essere *bloccato* o *libero*. Si richiede l'accesso alla variabile tramite la chiamata

```
int pthread_mutex_lock(pthread_mutex_t *mutex_p)
```

Se l'accesso è libero, il thread continuerà la sua esecuzione, altrimenti verrà *sospeso* (senza occupare tempo sulla CPU), e potrà continuare la sua esecuzione solo quando la lock sarà rilasciata.

Una volta che un thread ha terminato l'accesso alla variabile, deve rilasciare la lock con la chiamata

```
int pthread_mutex_unlock(pthread_mutex_t *mutex_p)
```

Una volta terminata l'utilità di una mutex, va eliminata con la chiamata

```
int pthread_mutex_destroy(pthread_mutex_t *mutex_p)
```

Il fenomeno della **Starvation** si verifica quando l'esecuzione di un thread o di un processo viene sospesa o impedita per un tempo indefinito, anche se è in grado di continuare l'esecuzione. È tipicamente associata all'imposizione di priorità o alla mancanza di equità nella pianificazione o nell'accesso alle risorse. Se una mutex è bloccata, il thread viene bloccato e inserito in una coda di thread in attesa. Se la coda è FIFO, non si verificherà starvation.

Un **Deadlock** invece identifica una situazione di attesa circolare in cui ogni elemento di un insieme di thread è sia in attesa del rilascio di una lock, sia in possesso di una lock necessaria agli altri thread.

```
1 pthread_mutex_lock(&a);
2 pthread_mutex_unlock(&b);
3 /*in attesa di a per
4 rilasciare b*/
5
```

```
1 pthread_mutex_lock(&b);
2 pthread_mutex_unlock(&a);
3 /*in attesa di b per
4 rilasciare a*/
5
```

In termini di prestazioni il Mutex mostra performance più elevate rispetto il busy waiting quando il numero di thread è superiore al numero di core fisici, si osservi la seguente tabella che riporta i tempi impiegati (in secondi) per stimare il valore di π su una macchiana con 8 core.

num. thread	busy waiting	mutex
1	2.9	2.9
2	1.45	1.45
4	0.73	0.73
8	0.38	0.38
16	0.5	0.38
32	0.8	0.4
64	3.56	0.38

3.2.2 Semafori, Barriere e Variabili di Condizione

Il busy waiting permette l'accesso esclusivo ai thread imponendo anche un certo ordine, ma presenta degli svantaggi considerevoli, che però non presenta l'utilizzo delle mutex, seppur quest'ultimo lascia l'ordine di accesso indeterminatoe deciso dal sistema operativo, ci sono situazioni in cui si vuole un accesso ordinato senza dover subire i problemi del busy waiting.

Lo standard Posix definisce un costrutto simile al mutex ma più elaborato : i **Semafori**, che rappresentano delle mutex *non binarie*. Per utilizzarli, si include la direttiva

```
# include <semaphore.h>
```

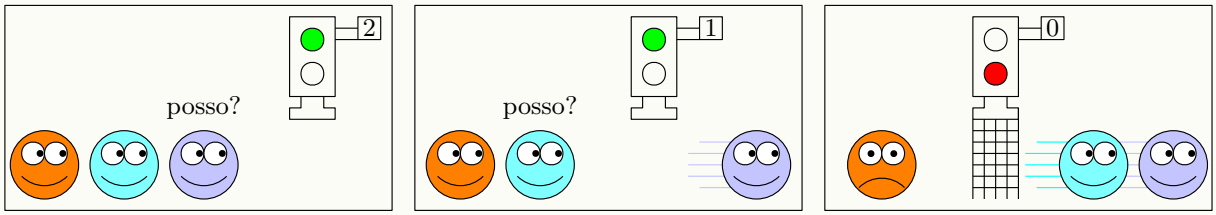
L'handler dei semafori è il tipo `sem_t` , un semaforo viene inizializzato con `int sem_init` i cui parametri sono

- `sem_t* semaphore_p`
- `int shared` se uguale ad 1, sarà condiviso anche fra diversi processi
- `unsigned initial_val` il numero intero con il quale è inizializzato un semaforo

Il funzionamento è il seguente, quando un thread vuole accedere ad una sezione, chiama la funzione `sem_wait(sem_t *sem)` , ad ogni semaforo è associato un numero intero, se tale numero è maggiore di zero al momento della chiamata, il processo potrà procedere, e decrementerà di 1 il valore del semaforo.

Se il valore è uguale a zero, il processo si arresterà entrando in una coda di attesa, attendendo il via libera per ripartire.

Quando un thread termina con l'utilizzo del semaforo, può chiamare la funzione `sem_post(sem_t * sem)` che farà procedere un thread in attesa del semaforo nell'esecuzione. Se la coda è vuota, il valore del semaforo verrà incrementato di 1.



Con **barriera** si intende un costrutto che viene inserito in un certo punto del codice, in cui ogni thread che lo esegue, per poter continuare l'esecuzione oltre tale punto deve attendere che ogni altro thread vi sia arrivato, serve a risincronizzare l'esecuzione dei thread facendoli ripartire da un punto comune. Un tipico esempio può essere quello di voler far noto in fase di debug che tutti thread hanno raggiunto un certo punto del codice.

```
1    /*punto comune del programma*/
2    barriera;
3    if(my_rank==0){
4        printf("tutti i thread hanno raggiunto questo punto \n");
5        fflush(stdout);
6    }
```

L'implementazione di una barriera tramite l'uso del busy waiting e delle mutex è intiuitiva, si utilizza un contatore condiviso inizializzato a zero (il cui accesso è protetto da mutex) e si incrementa di 1 per ogni thread che arriva a tal punto. Ogni thread sarà in busy waiting finché il contatore non sarà uguale al numero dei thread.

```
1    int counter = 0;
2    int thread_number;
3    pthread_mutex barrier_mutex;
4
5    void *thread_function(void *arg_p){
6        ...
7        /*implementazione della barriera*/
8        pthread_mutex_lock(&barrier_mutex);
9        counter++;
10       pthread_mutex_unlock(&barrier_mutex);
11       while(counter<thread_number);
12       /*fine della barriera*/
13       ...
14   }
```

Una barriera può anche essere implementata con un semaforo. L'implementazione mostrata presenta un problema, se fosse necessario riutilizzare tale barriera, bisognerebbe resettare la variabile `counter` a zero. Tale reset però deve avvenire solo *in seguito* al passaggio di tutti i thread della barriera, potrebbe capitare che un thread non si sia reso conto che la variabile counter avesse raggiunto il valore sufficiente per proseguire, rimanendo bloccato nella barriera. Questo problema, nonostante possa sembrare banale, è in realtà un intralcio considerevole e non ha soluzioni immediate.

Può essere di aiuto l'uso di una **variabile di condizione**, queste ultime sono un *oggetto* associato ad una mutex, che permettono ad un thread di sospendere la sua esecuzione fino all'accadere di un determinato *evento*, si definiscono con il tipo `pthread_cond_t` . A seguito di tale evento, il thread può essere svegliato con un apposito segnale. Una variabile di condizione descrive un *evento*.

```
1    lock mutex;
2    if CONDIZIONE SI AVVERA : /* avvisa i thread che la condizione si e' avverata*/
```

```
3           signal thread(s);
4       else : /*se non e' verificata, si libera la lock e si attende la condizione*/
5           unlock mutex
6           SOSPENSIONE ESECUZIONE
7       unlock mutex;
8
```

In seguito, l'implementazione di una barriera con delle variabili di condizione.

```
1   int counter = 0; /*variabile condivisa*/
2   pthread_mutex_t mutex;
3   pthread_cond_t cond_var;
4   ...
5
6   void *thread\_function(void *args_p){
7       ...
8       /*barriera*/
9       ptrhead_mutex_lock(&mutex);
10      counter++;
11      if(counter==thread_count){
12          counter=0;
13          pthread_cond_broadcast(&cond_var); /*avvisa tutti i thread che
14                                               la condizione si e' avverata*/
15      }else{
16          while(pthread_cond_wait(&cond_var,&mutex)!=0);
17      }
18      ptrhead_mutex_unlock(&mutex);
19  }
```

Si osservi la riga di codice 16

```
while(pthread_cond_wait(&cond_var,&mutex)!=0);
```

La chiamata `pthread_cond_wait` sospende il processo, perché allora è necessario inserirla all'interno di un while come se andasse verificato periodicamente? È buona norma dato che il sistema operativo potrebbe inaspettatamente svegliare un processo dallo stato di attesa prima che la condizione si avveri, queste *sveglie spurie* vanno quindi gestite.

- `pthread_cond_signal(pthread_cond_t* cond_var_p);` sveglia un thread in attesa
- `pthread_cond_broadcast(pthread_cond_t* cond_var_p);` sveglia tutti i thread in attesa
- `pthread_cond_init(pthread_cond_t* cond_p,pthread_condattr_t* cond_attr_p)` inizializza una variabile di condizione
- `pthread_cond_destroy(pthread_cond_t* cond_p)` termina una variabile di condizione
- La funzione

  `pthread_cond_wait(pthread_cont_t* cond_var_p, pthread_mutex_t mutex_p)`

  si occupa di

  1. fare l'unlock della mutex
  2. sospendere il thread finché non arriva un segnale di risveglio
  3. fare la lock della mutex una volta svegliato il thread

3.2.3 Stima di π con Pthread

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>

unsigned precision = 100000; /*punti generati da ogni thread*/
const unsigned thread_number = 6;

unsigned total_tosses;
unsigned point_in_center = 0;

/*gestione accesso a variabili condivise*/
pthread_mutex_t mutex;

void *thread_function(void *arg_p)
{
    int local_circle_point = 0;

    for (int i = 0; i <= precision; i++)
    {
        double x = (double)rand() / RAND_MAX * 2.0 - 1.0; /*Generazione punto casuale*/
        double y = (double)rand() / RAND_MAX * 2.0 - 1.0;
        local_circle_point += (x * x + y * y < 1); /*Controllo se e' nel cerchio*/
    }

    /*L'accesso alla variabile condivisa deve essere mutualmente esclusivo*/
    pthread_mutex_lock(&mutex);
    point_in_center += local_circle_point;
    pthread_mutex_unlock(&mutex);
}

int main(int argc, char **argv)
{
    srand(time(NULL));

    if (argc > 1)
    {
        precision = atoi(argv[1]);
    }

    total_tosses = precision * thread_number;
    pthread_mutex_init(&mutex, NULL);
    pthread_t tids[thread_number];

    for (int i = 0; i < thread_number; i++)
    {
        pthread_create(&tids[i], NULL, thread_function, NULL);
    }

    for (int i = 0; i < thread_number; i++)
    {
        pthread_join(tids[i], NULL);
    }

    double esteem = ((double)point_in_center / (double)(total_tosses)) * 4;
    printf("valore di pi greco stimato : %lf\n", esteem);

    return 0;
}
```