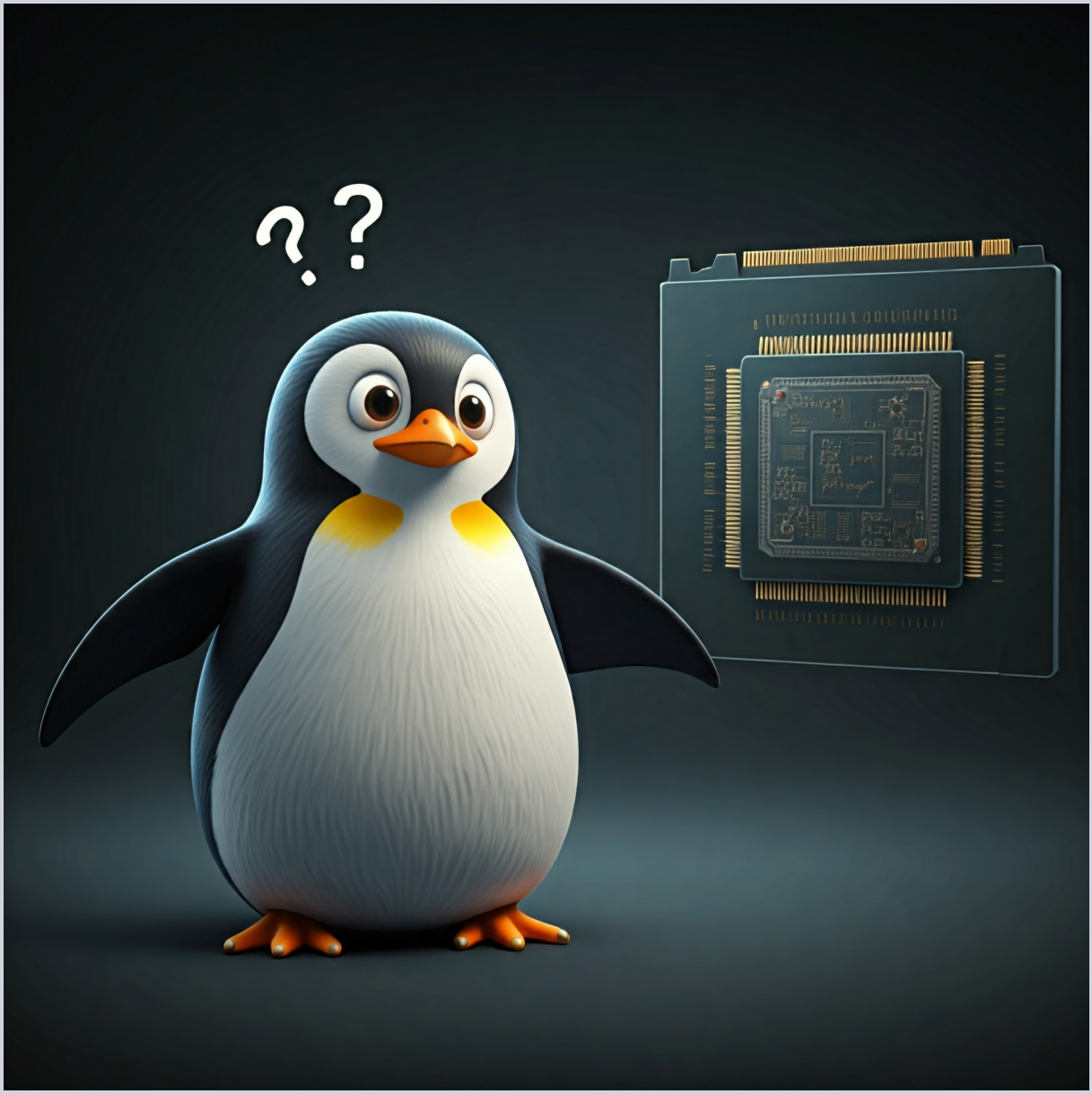
Marco Casu

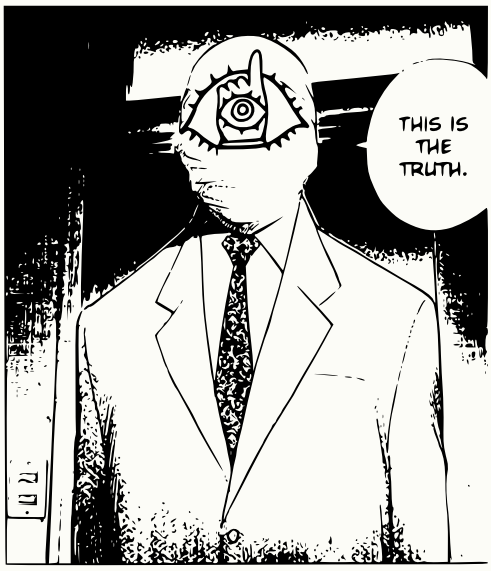
Programmazione di Sistemi Multicore



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

CAPITOLO

1

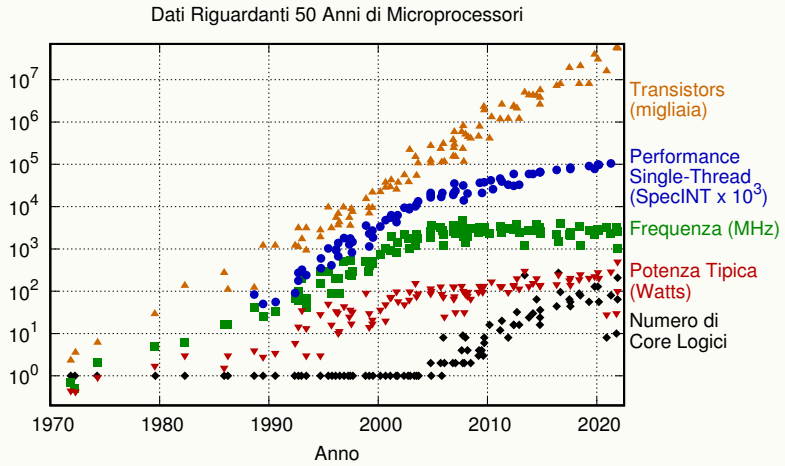
PARALLELISMO : MOTIVAZIONI

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 :

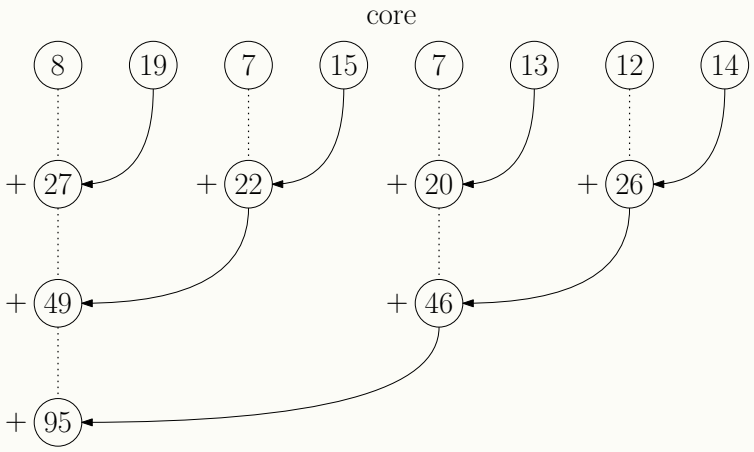


Figura 1.1: calcolo somme a coppie

- **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.

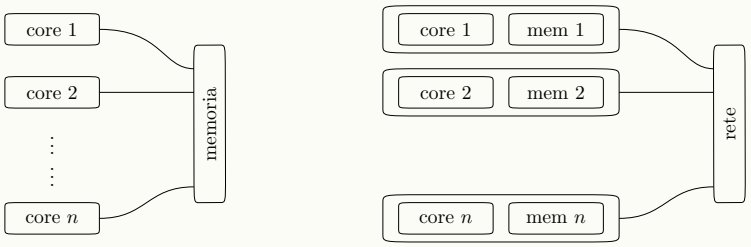


Figura 1.2: modelli di parallelismo

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**.

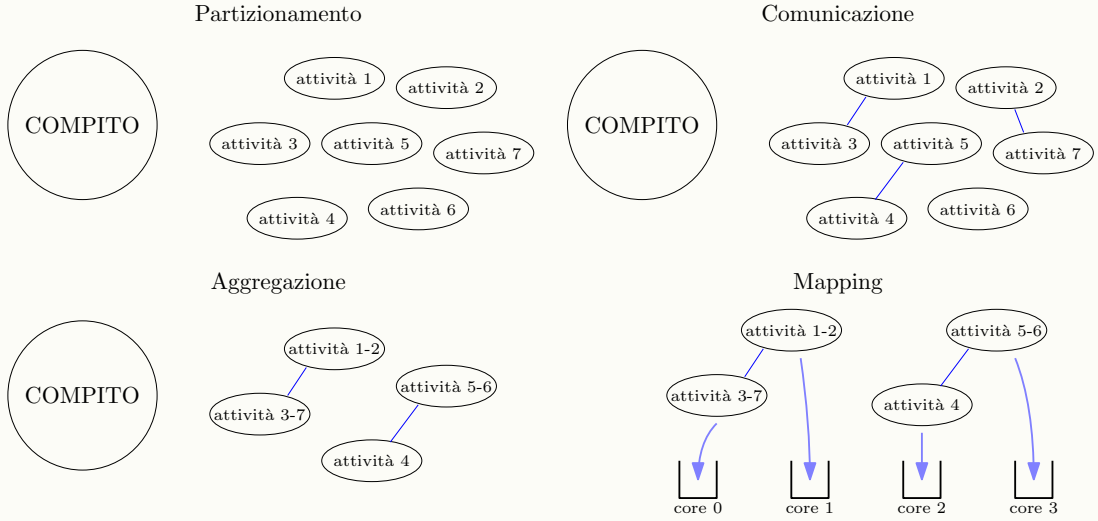


Figura 2.1: Foster's methodology

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.

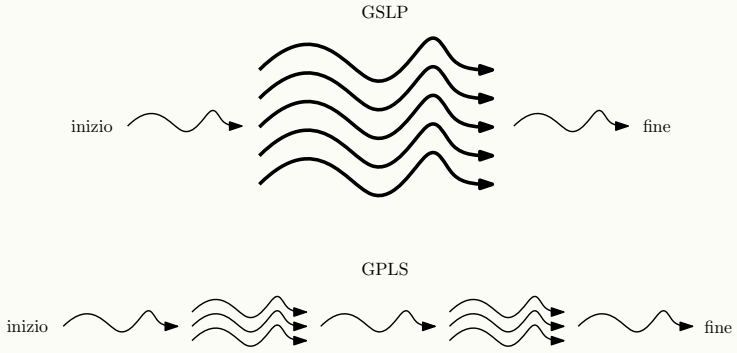


Figura 2.2: GPLS e GSLP

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 }
```