# Progettazione di Sistemi Embedded e Multicore

Leonardo Biason

October 14, 2024

# Chapter 1

# Programmazione Parallela

Ad oggi, molte tasks e molti programmi necessitano di grandi capacità di calcolo, soprattutto nel campo delle AI, e costruire centri di elaborazione sempre più grandi può non sempre costituire una soluzione. Certo, negli ultimi anni abbiamo assistito a una grande evoluzione dei microprocessori e delle loro potenze, ma non è abbastanza avere hardware sempre più potente, serve saperlo impiegare bene.

La società Nvidia, produttrice di **GPU**s (Graphical Processing Units) è riuscita, in questi anni, a produrre schede grafiche contenenti sempre più milioni di transistors, rendendo una GPU un oggetto estremamente complicato.

Come detto in precedenza, avere oggetti così potenti non è abbastanza, serve saper usare questi ultimi nel modo migliore possibile, ed è proprio questo l'obiettivo di questo corso.

# Chapter 2

# Message Passing Interface (MPI)

**MPI** (acronimo di **M**essage **P**assing **I**nterface) è una libreria usata per **programmare sistemi a memoria distribuita**, e delle varie librerie menzionate nell'introduzione è l'unica pensata per sistemi a memoria distribuita. Questo vuol dire che la memoria e il core usato per ogni thread o processo sono **unici**. Tale core e memoria possono essere collegati attraverso vari metodi: un bus, la rete, etc...

MPI fa uso del paradigma **Single Program Multiple Data** (**SPMD**), quindi ci sarà un **unico programma** che verrà compilato e poi eseguito da vari processi o threads. Per determinare cosa ogni processo o thread deve fare, si usa semplicemente un'istruzione di **branching**, come l'if-else o lo switch.

Siccome la memoria non è condivisa tra i vari processi, l'unico modo per passarsi dei dati è attraverso l'invio di **messaggi** (da qui il nome della libreria). Per esempio, abbiamo visto come fare un semplice "Hello world" in C in modo sequenziale, ma possiamo anche renderlo parallelo tramite MPI. Ad esempio:

Code

```c
#include <stdio.h>

int main() {
   printf("Hello world");
   return 0;
}
```

Per rendere questo Hello World un programma parallelo tramite MPI, serve includere la libreria `mpi.h` e usare alcune funzioni della libreria. Vediamo intanto come potremmo scrivere il programma:

Code

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

int main(void) {
   // Per usare MPI, serve usare una funzione chiamata MPI_INIT;
   int r = MPI_Init(NULL, NULL);

   if(r != MPI_SUCCESS) {
      printf("C'è stato un errore con il programma");
      MPI_Abort(MPI_COMM_WORLD, r);
   }

   printf("Hello world");

   // Per terminare l'esecuzione di tutti i threads si usa MPI_Finalize
   MPI_Finalize();
   return 0;
}
```

Nel precedente codice sono state usate alcune funzioni e alcuni valori di MPI, che possiamo notare grazie al

prefix "MPI_", **comune a tutte le definizioni**, siano esse di funzioni, variabili o costanti, **della libreria**:

- `MPI_Init()`: **inizializza** un programma su più processi o threads, e restituisce come output un int, che identifica se è stato possibile inizializzare con successo la libreria di MPI o meno (ovverosia restituisce 0 se la libreria è stata inizializzata con successo, un altro numero altrimenti);
- `MPI_SUCCESS`: è il segnale con cui è possibile comparare l'output di `MPI_Init` per controllare se MPI è stato inizializzato correttamente o meno;
- `MPI_Abort(MPI_COMM_WORLD, <mpi_boot_result>)`: **abortisce** l'esecuzione di MPI, ad esempio nel caso in cui l'inizializzazione non sia stata eseguita con successo;
- `MPI_Finalize()`: **interrompe** l'esecuzione di MPI a fine programma.

Per compilare ed eseguire un programma con MPI si usa `mpicc`, che è un wrapper del compilatore `gcc` di C. Un comando che viene usato per compilare un programma che usa MPI può essere il seguente:

```
$ mpicc <file>.c -o <output>
$ mpicc -g -Wall <file>.c -o <output>    # Fa stampare i warning in console
```

Il compilatore ha molte flags che possono essere usate, così da personalizzare il processo di compilazione. Nel secondo comando si può notare l'uso di due flags che possono risultare comode in fase di debug:

- `-Wall`: fa stampare in console **tutti i warnings** del compilatore;
- `-g`: fa stampare in console varie **informazioni di debug**.

Questo è per quanto riguarda la compilazione, ma per eseguire il programma invece? Dovremo usare `mpirun`, attraverso il seguente comando:

```
$ mpirun -n <numero_core_fisici> <programma>
$ mpirun --oversubscribe -n <numero_core> <programma>
      # Permette di usare più core di quelli fisici
```

Normalmente MPI esegue il codice solo sui core fisici di una CPU, tuttavia è possibile far sì che questa limitazione non venga considerata. La flag `--oversubscribe` permette di lanciare il programma su $n$ processi, dove $n \geq$ numero di core fisici.

In un programma complesso, è spesso utile sapere quale core esegue quale parte di programma, magari anche per assegnare dei compiti diversi ad ogni core. In questi casi, possiamo differenziare i processi in base al loro **rank**.

DEFINITION

**Rank**

Il **rank** di un processo appartenente a un programma di MPI è un **indice incrementale**, nell'intervallo $[0, 1, 2, \ldots, p)$, che viene assegnato ad ogni processo.