Progettazione di Sistemi Embedded e Multicore

Leonardo Biason

October 16, 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:

```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:

```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.

2.1 Misurazione delle Prestazioni

Una volta che sviluppiamo un programma con MPI, è importante misurarne le prestazioni. Esistono vari modi per controllare l'efficienza di un programma, e qui ne vedremo alcuni.

Un modo molto semplice per capire l'efficienza di un programma consiste nel misurare il tempo di esecuzione, e possiamo farlo tramite la funzione MPI_Wtime(), la quale ritorna un timestamp che misura il tempo dal momento in cui il programma viene eseguito. Per poter ottenere il tempo di esecuzione, bisognerà sottrarre dal tempo a fine esecuzione il tempo di quanto la funzione viene chiamata.

In un programma parallelo, ogni processo calcola il suo tempo, ma può essere che alcuni processi impieghino meno tempo, soprattutto se ogni processo esegue operazioni diverse. Per questo motivo, l'efficienza si calcola considerando il tempo maggiore. Per ovviare a questo problema, tutti i processi possono mandare a un solo processo i loro tempi, e tramite una chiamata di MPI_Reduce() si può ritornare solo il tempo maggiore.

```c
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,
    MPI_COMM_WORLD);

if (rank == 0) {
   printf("Tempo totale di esecuzione: %e\n", elapsed);
}
```

Tuttavia, un altro problema si presenta: **non tutti i processi possono iniziare allo stesso tempo**. E questo chiaramente è un problema, soprattutto se il processo che inizia in ritardo inizia parecchio in ritardo. Un modo per far sì che tutti i processi inizino assieme è tramite l'uso di una funzione chiamata `MPI_Barrier()`. Tale funzione è una **funzione collettiva**, che si propaga a tutti i processi, e che li blocca fino al momento in cui tutti non raggiungono tale barriera. Possiamo considerare questa funzione come una sorta di checkpoint, che **posta all'inizio del programma** fa sì che tutti i processi inizino ad eseguire il loro codice "assieme".

Tuttavia, se anche questa collettiva generasse ritardi venendo propagata, non avremmo più un inizio collettivo. Riuscire a sincronizzare tutti i processi assieme è complicato, e richiederebbe l'uso di clock interni per ogni core. Tuttavia, in casi in cui non sia fondamentale avere grande precisione, `MPI_Barrier()` **garantisce** una **buona approssimazione**.

Ma è abbastanza eseguire l'applicazione solo una volta per avere una buona misurazione? Generalmente no, e ci sono vari motivi: sappiamo che il sistema operativo, di tanto in tanto, può effettuare dei cambi di contesto, per cui uno o più core vengono usati per eseguire delle operazioni dell'OS. Questo chiaramente può aumentare i tempi per alcune esecuzioni. In altri casi invece potrebbe essere che lo scheduler della memoria liberi la cache prima del necessario, etc...

Tali motivi di ritardo vengono chiamati **rumore** (o **noise**). In genere, per misurare l'efficienza di un programma, **conviene sempre basarsi su tutte le esecuzioni di un programma**, considerando media, mediana, il tipo di distribuzione, eventuali intervalli di confidenza, etc... Il tempo di esecuzione minimo, da solo, non indica molto, poiché per vari motivi di interferenza può non sempre (e in realtà non accade quasi mai) indicare la vera efficienza del programma.

2.1.1 Speed-Up ed Efficienza

Generalmente, potremmo aspettarci che se un processo viene eseguito da quanti più core possibili, allora potremmo ottenere tempi sempre più bassi. Benché in molti casi questo sia vero, lo speed-up dell'esecuzione parallela non è sempre esistente aumentando il numero di processi e di core. Infatti, consideriamo il seguente esempio:
Se notiamo la prima colonna, il tempo di esecuzione con 8 e 16 core è identico. Questo perché, oltre a un certo punto, anche dividendo le operazioni su più core, raggiungeremmo un tempo minimo di inizializzazione che non è ottimizzabile. Dunque, lo speed-up terminerebbe. Ma cosa intendiamo effettivamente con speed-up?

DEFINITION

Speed-up

Definiamo le seguenti variabili:

- $T_s(n)$: il **tempo di esecuzione** di un problema di dimensione n in **seriale**;
- $T_p(n, p)$: il **tempo di esecuzione** di un problema di dimensione n eseguito in **parallelo** su p core;

Definition

Lo **speed-up** di un'applicazione in parallelo su p core viene dunque definito come il rapporto tra il tempo di esecuzione in seriale e il tempo di esecuzione in parallelo con p core:

$$S(n, p) = \frac{T_s(n)}{T_p(n, p)}$$

Se $S(n, p) = p$, allora lo speed-up viene definito come **speed-up lineare**.

C'è un dettaglio importante di questa precedente definizione: il tempo di esecuzione in sequenziale **non è uguale** al tempo di esecuzione in parallelo con $p = 1$, anzi, tendenzialmente $t_p(n, 1) \geq t_s(n)$. A causa di questo, possiamo definire due implementazioni diverse della precedente formula:

$$S(n, p) = \frac{T_s(n)}{T_p(n, p)} \qquad S(n, p) = \frac{T_p(n, 1)}{T_p(n, p)}$$

Speed-up **Scalabilità**

Grazie alla definizione di speed-up, possiamo definire anche il concetto di **efficienza**:

Definition

Efficienza

L'**efficienza** di un programma viene calcolata come il rapporto tra lo speed-up di un programma e il numero di core su cui viene eseguito:

$$E(n, p) = \frac{S(n, p)}{p} = \frac{T_s(n)}{p \cdot T_p(n, p)}$$

2.1.2 Scalabilità Forte e Scalabilità Debole

Esistono due tipi di scalabilità, in base ai risultati che si ottengono dalle misurazioni dell'efficienza di un programma: **scalabilità forte** e **scalabilità debole**.

- Per **scalabilità forte** si intende quando, dato un problema di grandezza n, se si incrementa il numero di processi p, allora l'efficienza rimane alta;
- Per **scalabilità debole** si intende quando, dato un problema di grandezza n e un numero di processi p, se incrementando ugualmente n e p, allora l'efficienza rimane alta.

comm_size	Ordine della matrice				
	1024	2048	4096	8192	16384

2.1.3 Leggi di Amdhal e Gustafson

Quando dobbiamo rendere un programma parallelo, sappiamo che possiamo parallelizzare solo alcune operazioni: ad esempio, leggere dal disco dei dati, richiedere dei dati in input all'utente o mandare dati attraverso un comunicatore di MPI. Avremo dunque una parte **sempre sequenziale** e una parte **parallelizzabile**. La frazione del programma sempre sequenziale viene indicata con α. C'è una legge, chiamata **legge di Amdhal**, che definisce lo speed-up possibile di un'applicazione.

Definition

Legge di Amdhal

Per la **legge di Amdhal**, lo speed-up di un'applicazione, se resa parallela, è limitato dalla **frazione di codice sequenziale** α:

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

Definition

Lo speed-up calcolabile sarà dunque uguale a

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

Generalmente, se portassimo il numero di core p all'infinito, raggiungeremmo il seguente valore:

$$\lim_{x \to +\infty} S(n,\, p) = \frac{1}{1-\alpha}$$

La legge di Amdhal tuttavia ha dei problemi: intanto non tiene conto della scalabilità debole (poiché la legge di Amdhal considera un n costante)

Definition

Legge di Gustafson

La legge di Gustafson definisce ciò che si chiama **speed-up scalabile**, e che prende in assunzione che, se si aumenta il numero di processi p per una costante α, allora anche la dimensione del problema n aumenta di α.

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