Programmazione di Sistemi Multicore

Marco Panunzio & Davide Pucci

2016-2017

Indice

Capitolo 1

Capitolo 1

1.1 Panoramica

1965:
"La complessità di un microcircuito, misurata ad esempio tramite il numero di transistori per chip, raddoppia ogni 18 mesi. (Gordon More) "

La prima legge di Moore è tratta da un'osservazione empirica di Gordon Moore, cofondatore di Intel con Robert Noyce: nel 1965, Gordon Moore, che all'epoca era a capo del settore R&D della Fairchild Semiconductor e tre anni dopo fondò la Intel, scrisse infatti un articolo su una rivista specializzata nel quale illustrava come nel periodo 1959-1965 il numero di componenti elettronici (ad esempio i transistor) che formano un chip fosse raddoppiato ogni anno.
Nel 1975 questa previsione si rivelò corretta e prima della fine del decennio i tempi si allungarono a due anni, periodo che rimarrà valido per tutti gli anni ottanta. La legge, che verrà estesa per tutti gli anni novanta e resterà valida fino ai nostri giorni, viene riformulata alla fine degli anni ottanta ed elaborata nella sua forma definitiva, ovvero che il numero di transistori nei processori raddoppia ogni 18 mesi. Questa legge è diventata il metro e l'obiettivo di tutte le aziende che operano nel settore, come Intel e AMD.

Nel 2001, ci si accorse che l'aumento della potenza dei processori portava ad un drastico aumento del consumo energetico e della dissipazione del calore. Per questa ragione, dal 2005, si iniziò ad aumentare il numero di *core* del processore, invece del *clock rate*.
Da questo punto nasce la definizione di *programmazione sequenziale* e di *programmazione parallela*, che influiscono in diversi ambiti:

- *Programmazione*: dividere il lavoro in diversi *thread* di esecuzione, in sincronizzazione l'uno con l'altro;

- *Algoritmi*: come può l'attività parallela fornire *speedup* al lavoro offerto dall'algoritmo (aumento del *throughput*, *lavoro/unità di tempo*);
- *Strutture dati*: nasce la necessità di fornire accesso concorrente alle strutture dati.

1.1.1 Concorrenza

La nascita della parallelizzazione in programmazione ha portato alla comparsa di un problema preponderante: la gestione della concorrenza.
Questa impone che venga gestito in maniera adeguata l'accesso alle risorse condivise dai vari thread, dove le operazioni devono essere scandite in ordine corretto in maniera tale da fornire risultati corretti e dove si necessita di sincronizzazione attraverso funzioni primitive (come i *semafori*).
In sintesi la differenza tra *parallelismo* e *concorrenza* è che la prima offre l'uso di diverse risorse extra per risolvere un problema più rapidamente, la seconda offre la corretta gestione degli accessi alle risorse condivise.

1.2 Background teorico

Lo *speedup* su n processi è definito come:

$$S_n = \frac{t_s}{t_n}$$

Dove:

- t_s è il tempo di esecuzione su un solo processo;
- n è il numero di processori;
- t_n è il tempo di esecuzione su n processori.

L'efficienza, invece, è definita come:

$$E_n = \frac{S_n}{n} = \frac{t_s}{n \times t_n}$$

1.2.1 Legge di Amdahl

Nel 1967, *Amdahl* definisce la seguente legge: *dato un'applicazione parallela, dove la frazione f è inerentemente sequenziale, il migliore speedup possibile su n processori è definito dalla seguente formula*

$$S(n) \leq \frac{1}{f + \frac{1-f}{n}} \Rightarrow S(\infty) = \lim_{n \to \infty} S(n) = \frac{1}{f}$$

Implicazioni Dunque lo *speedup* di un programma che utilizza più processori è limitato dal tempo necessario dalla frazione sequenziale dello stesso.
Ad esempio: un programma necessita di 20h su un singolo core, e una particolare porzione del programma che richiede 1h per l'esecuzione (quini il 5%) non può essere parallelizzata, mentre le rimanenti 19h (ovvero il 95%) può esserlo. Indifferentemente da quanti processori sono destinati alla parallelizzazione, il *wallclock time* (*tempo di esecuzione*) non può essere minore di 1h. Quindi, teoricamente lo *speedup* è limitato al massimo fino a 20x.
Inoltre, praticamente può andare anche peggio, perché non abbiamo tenuto conto del costo del *load balancing*, della *comunicazione*, della *coordinazione* tra threads, che porta ad un *overhead addizionale*.

1.2.2 Legge di Gustafson

1988:

$$S(n) \leq \alpha + n(a - \alpha) = n - \alpha(n - 1)$$

Dove:

- n è il numero di cores;
- a è la parte sequenziale;
- b è la parte parallela (eseguita da un core degli n);
- $a + b$ è il tempo di esecuzione parallela su n cores (senza *overhead*);
- $a + n \times b$ è il tempo totale di esecuzione serializzata;
- $\alpha = \frac{a}{a+b}$ è la frazione sequenziale del tempo di esecuzione parallelo.

Sostanzialmente, *Gustafson* si accorse che aumentando il numero n di processori, con lo stesso ammontare di tempo, si potevano risolvere problemi di grandezza maggiore e che la grandezza del problema cresce monotonamente con n, così che la frazione sequenziale del *workload* non domina.
Quindi, aumentando il *workload* insieme con il numero di processori, si ottiene più *speedup*.

Implicazioni $S(n) \leq n - \alpha(n - 1)$, e quindi se la frazione sequenziale α è piccola, si può arrivare ad ottenere uno *speedup* vicino ad n.
Inoltre, nasce la definizione di *forte* e *debole scalabilità*:

- *scalabilità forte* (*strong scaling*): vogliamo sapere quanto rapidamente possiamo completare l'analisi di un particolare insieme di dati incrementando il numero di processori;
- *scalabilità debole* (*weak scaling*): vogliamo sapere se possiamo analizzare più dati approssimativamente nello stesso tempo, incrementando il numero di processori.

1.3 Sistemi di Calcolo parallelo

Vecchia classificazione hardware:

	Singola istruzione	**Istruzione multipla**
Single data	SISD	MISD
Multiple data	SIMD	MIMD

Dove:

- **SISD**: nessuna parallelizzazione, né nei *mainframes* (*data stream*), né nelle istruzioni;
- **SIMD**: parallelizzazione dei dati (*stream processors*, *GPUs*). Una singola *control unit* spedisce la stessa istruzione a più processori che lavorano su dati diversi;
- **MISD**: istruzioni multiple operanti sugli stessi *data stream* (inusuale);
- **MIMD**: istruzioni multiple operanti indipendentemente su *data stream* multipli (computer più recenti). Ogni processore ha la sua *control unit* e può eseguire dierse istruzioni su dati differenti.

Esistono due forme primarie di scambio dati tra task paralleli:

1. accedendo a uno spazio dati condiviso: *memoria condivisa* (*multi-processori*). Piattaforme:
   (a) **NUMA** (macchina con *non-uniform memory access*): un processore può accedere alla sua memoria locale più rapidamente rispetto alla memoria non locale;
   (b) **UMA** (macchina con *uniform memory access*): tutti gli accessi hanno pari velocità.
2. scambiando messaggi in una rete: *memoria distribuita* (*multi-computers*);
3. utilizzando sistemi ibridi (*hybrid systems*), sia con memoria condivisa che distribuita: ogni nodo ha una sua memoria ed n processori. Quando necessita di più di n processori, utilizza lo scambio di messaggi in rete;
4. utilizzando sistemi multicore (*multicore systems*), estendendo i sistemi ibridi e prevendendo l'utilizzo di *memoria cache*, *accesso alla memoria principale* e di una connessione *node-to-node* attraverso la rete;
5. utilizzando sistemi accellerati (*accelerated systems*), dove le computazioni vengono eseguite sia dalla CPU che dagli *acceleratori*. Esistono diversi tipi di acceleratori:
   - **GPGPU** (*general purpose graphical processing unit*), fornisce grafica via hardware, utilizzando modelli, librerie e compilatori indipendenti come *CUDA* e *OpenCL*;
   - **MIC** (*many integrated core*), una tradizionale CPU hardware.

1.4 Modelli di Programmazione parallela

Nella prima parte del corso si utilizzerà *memoria condivisa con thread espliciti*. Precedentemente, un programma sequenziale aveva:

1. un *program counter*;
2. una *call stack*;
3. degli oggetti nell'*heap*;
4. dei campi statici.

Ora, in programmi a memoria condivisa con thread espliciti:

1. un set di *thread*, ognuno con il suo *program counter* e la sua *call stack*;
2. i thread possono implicitamente condividere campi statici e oggetti;
3. *scheduler* di thread.

1.4.1 Parallelizzazione in Java

Java fornisce un framework builtin basico per la gestione dei thread, attraverso gli oggetti *java.lang.Thread* e l'*overriding* del metodo *run()*, per definire le attività che il singolo thread eseguirà. Per lanciare il thread, basterà chiamare il metodo *start()*.

```
class ExampleThread extends java.lang.Thread {

        int i;

        ExampleThread(int i) {
                this.i = i;
        }

        public void run() {
                System.out.println("Thread " + i + " says hi");
                System.out.println("Thread " + i + " says bye");
        }
}
class M {
        public static void main(String[] args) {
                for (int i=1, i <= 20; i++) {
                        ExampleThread t = new ExampleThread(i);
                        t.start();
                }
        }
}
```