# 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 $\alpha$ 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.

# Capitolo 2

# Capitolo 2

## 2.1 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();
                }
        }
}
```

### 2.1.1 Fork/Join

La tecnica di programmazione parallela *fork/join* nasce dalla modalità di gestire la parallelizzazione attraverso l'utilizzo dei metodi *fork()* (non propriamente un metodo - java -, ma la funzione primitiva con cui vengono generati tutti i

processi, così come i processi figli e i thread) e *join()* (utilizzato dal chiamante al thread figlio per rimanervi agganciato fino alla terminazione dello stesso).

## 2.2 Quanti thread?

La gestione del numero dei thread partoriti da un programma gioca un ruolo fondamentale nell'ottimizzazione del programma stesso. E' importantissimo sapere gestire il problema del numero e il tipo di lavoro eseguito dai thread adeguatamente, altrimenti l'utilizzo stesso di questi ultimi rischierebbe di rendere l'esecuzione dell'applicativo più lenta di quella in modalità serializzata.
L'idea più comune è quella di utilizzare un numero di thread pari al numero di core del processore in utilizzo, ma non si tratta di una soluzione accurata. D'altra parte, è molto più utile avere un numero di thread largamente più alto rispetto a quello dei core:

- *forward-portable*: codice indipendente dal numero di processori;
- viene gestita una notevole quantità di lavoro in maniera distribuita attraverso grandi pile di thread, alternati così l'uno all'altro;
- viene garantito il *load balancing*, perché gestiti piccole porzioni di lavoro alla volta.

Il problema fondamentale è che se necessitiamo di acquisire risultati su diversi thread, è pesante doversi scorrere l'intera lista per ricercare informazioni e prelevarle dagli stessi o da alcuni di essi.
Altro problema molto pratico: la classe builtin Java *java.lang.Thread* non è adatta alla gestione di piccoli tasks; così, un numero elevato di thread, porta ad un grande *overhead*.

#### Divide et Impera

Per risolvere questo tipo di problema (nella gestione e ricerca degli innumerevoli thread in coda, per l'acquisizione di informazioni relative alla loro esecuzione per esempio), si ricorre spesso a logiche algoritmiche diverse, come nel caso di *divide et impera*.
L'idea è quella di generare ricorsivamente, attraverso ciascun thread, nuovi thread. Così che ognuno di essi ne abbia in gestione altri, ma riducendo la mole di ciascuno e il costo di ricerca delle informazioni.

## 2.3 Libreria ForkJoin

Dato il pesante costo dei thread builtin Java, che portano un pesantissimo overhead (senza considerare la comune necessità di inizializzare migliaia di thread in contemporanea), il corso suggerisce di migrare su un framework alternativo: *ForkJoin*.

Scritto secondo la tecnica presentata come *Fork/Join*, appunto, e seguendo l'idea algoritmica del *divide-et-impera*, fa uso delle seguenti classi:

- *ForkJoinPool*: contenitore che esegue tutti i task *fork-join* (per un'implementazione corretta, instanziarne sempre uno solo);
- *RecursiveTask¡V¿*: utilizzato come thread da eseguire - attraverso la sottoclasse - e fa in modo di ritornare un risultato;
- *RecursiveAction*: come un *RecursiveTask*, ma non ritorna alcun risultato;
- *ForkJoinTask¡V¿*: è la superclasse del *RecursiveTask¡V¿* e della *RecursiveAction*, implementa i metodi *fork()* e *join()*. Non verrà mai utilzzato direttamente, ma contiene *Javadoc* molto utili e completi.

Per utilizzare il framework, occorre innanzitutto creare un *ForkJoinPool*, utilizzato dall'intero programma (ha quindi senso inserirlo in un campo statico); una volta fatto, si lancia l'*invoke()* del pool sulla *RecursiveAction* (si possono comunque utilizzare i metodi *fork()*, *compute()*, *join()*, sull'istanza della action). Per esempio:

```
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveAction;

class ExampleThread {

        static final ForkJoinPool fjPool = new ForkJoinPool();

        static int sum(int[] array) {
                SumArray t = new SumArray(array, 0, array.length);
                fjPool.invoke(t);
                return t.ans;
        }
}
```

Qualora invece fosse necessario che il thread ritornasse un risultato, occorre utilizzare il *RecursiveTask*, che presenta alcune differenze nella gestione rispetto al vecchio *Thread* builtin:

- Non estende **Thread**, ma **RecursiveTask¡T¿**;
- Non fa override di **run()**, ma di **compute()**;
- Non usa un campo **ans**, ma ritorna un'istanza di **V** dal **compute()**;
- Non chiama un metodo **start()**, ma **fork()**; Si chiama il metodo **join()** per ottenere il risultato; Non si chiama il metodo **run()**, ma si chiama un **pool** che lo esegua; Non si chiama, in alternativa, il metodo **run()**, ma **compute()**.

Quindi, l'implementazione della classe *SumArray* sarà la seguente:

```
class SumArray extends RecursiveTask<Integer> {

        int lo; int hi; int[] arr;

        SumArray(int[] arr, int lo, int hi) { ... }

        protected Integer compute() {
                if (hi - lo < SEQUENTIAL_CUTOFF) {
                        int ans = 0;
                        for (int i=lo; i<hi; i++)
                                ans += arr[i];
                        return ans;
                } else {
                        SumArray left = new SumArray(arr, lo, (hi+
                            lo)/2);
                        SumArray right = new SumArray(arr, (hi+lo)
                            /2, hi);
                        left.fork();
                        int rightAns = right.compute();
                        int leftAns = left.join();
                        return leftAns + rightAns;
                }
        }
}
```

**Note importanti**

1. **Sequential threshold**: la libreria suggerisce di fare approssimatamente tra le 100 e le 5000 operazioni di base in ogni sezione del vostro algoritmo;
2. La libreria necessita di tempo per *scaldarsi*, facendo utilizzo di particolari *cache* che portano ad un miglioramento nella performance dei task lunghi.

### 2.3.1 Riduzioni

Possiamo notare che la somma degli elementi di un array, attraverso l'idea del *divide-et-impera*, ha portato ad un miglioramento, passando da un costo di $O(n)$ sequenziale a quello di $O(\log n)$ parallelizzato.
Esistono però innumerevoli problemi come questo:

- Massimo o minimo elemento;
- Esiste un elemento che soddisfa un proprietà;
- L'elemento più a sinistra che soddisfi una proprieta;
- ...

Le computazioni di questo tipo prendono il nome di *Riduzioni* (o *reduces*), ovvero, che producono una risposta singola da un una collezione, tramite un operatore associativo (esempio: massimo, conteggio, più a destra, più a sinistra, somma, prodotto, ...; non esempio: mediana, sottrazione, ...). In ogni caso, il

risultato non deve necessariamente essere un numero, ma anche un *array*; d'altra parte, alcune informazioni sono inerentemente sequenziali, come per esempio l'elaborazione dell'elemento $arr[i]$, che dipende interamente dall'elaborazione precedente $arr[i-1]$.
L'utilizzo delle *Map* cerca di risolvere questo problema, perchè opera su una collezione indipendentemente, per creare una collezione della stessa dimensione.
Problema *vector addition* con *ForkJoin*:

```
class VecAdd extends RecursiveAction {

        int lo; int hi; int[] res; int[] arr1; int[] arr2;

        VecAdd(int lo, int hi, int[] res, int[] arr1, int[] arr2) {
             ... }

        protected void compute() {
                if (hi - lo < SEQUENTIAL_CUTOFF) {
                        for (int i=lo; i<hi; i++)
                                res[i]  = arr1[i] + arr2[i];
                } else {
                        int mid = (hi+lo)/2;
                        VecAdd left = new VecAdd(lo, mid, res, arr1
                            , arr2);
                        VecAdd right = new VecAdd(mid, hi, res,
                            arr1, arr2);
                        left.fork();
                        right.compute();
                        left.join();
                }
        }
}
class Main {

        static final ForkJoinPool fjPool = new ForkJoinPool();

        int[] add(int[] arr1, int[] arr2) {
                assert (arr1.length == arr2.length);

                int[] ans = new int[arr.length];
                fjPool.invoke(new VecAdd(0, arr.length, ans, arr1,
                    arr2));
                return ans;
        }
}
```

Le mappe e le riduzioni sono i cavalli di battaglia della programmazione parallelizzata, e rappresentano i più importanti pattern, per questo è necessario capire quando un algoritmo può essere scritto in termini di mappe e riduzioni.

**Google e MapReduce** *Google* stessa ha lavorato pesantemente su questo ordine di problemi, realizzando una classe *MapReduce* (o la sua versione open-source, *Hadoop*). L'idea è quella di realizzare mappe/riduzioni su grandi set di dati utilizzando alcune macchine ausiliarie, separando il lavoro ricorsivo di *divide-et-impera* dalla computazione effettiva.

**Alberi bilanciati**

Le mappe e le riduzioni lavorano molto bene su alberi bilanciati, infatti, per esempio, l'elemento minimo in un albero binario *non ordinato*, ma *bilanciato*, rimane $O(\log n)$, dati sufficienti processori.
Come calcolare il *sequential cutoff*? Basta salvarsi il numero di discendenti per ciascun nodo, facilmente manutenibile, oppure approssimarlo attraverso, per esempio, l'alteza dell'albero *AVL*.
Inoltre, il tipo di struttura dati influisce pesantemente sulla computazione parallelizzata: per esempio, per il parallelismo, gli alberi bilanciati generalmente lavorano meglio delle liste, riuscendo ad ottenere risultati con tempistiche esponenzialmente più brevi ($O(\log n$ contro $O(n)$).

### 2.3.2 Work e Span

L'obiettivo principale con il framework *ForkJoin* è quello di ottenere una performance a runtime asintoticamente ottimale, rispetto al numero di processori disponibile.
Per fare questa analisi, occore definire il *work* e lo *span*. Dato $T_P$ il tempo di esecuzione con $P$ processori disponibili:

- *work*: tempo di esecuzione con $T_1$;
- *span*: tempo di esecuzione con $T_\infty$.

Tendenzialmente, l'esecuzione di un programma utilizzando al strategia *fork/join* può essere vista come un *DAG*, dove i nodi sono le porzioni di codice da eseguire e gli archi, invece, le dipendenze di precedenza tra un nodo e l'altro. Quindi, un *fork() chiude* un nodo e crea due archi uscenti (quindi genera un thread e continua il thread corrente); un *join() chiude* un nodo e crea un nodo con due archi entranti (il nodo appena *chiuso* e l'ultimo nodo del thread *joinato*).
Ancora, se $T_P$ è stato definito come il tempo di esecuzione con $P$ processori disponibili, il *work* è la somma del tempo di esecuzione di tutti i nodi del *DAG*, mentre lo *span* è la somma del tempo di esecuzione di tutti i nodi del percorso più lungo all'interno del *DAG*.
Lo *speedup* su $P$ processori è quindi definito come $\frac{T_1}{T_P}$, dove lo speedup $P$ con $P$ processori è lo speedup lineare perfetto. Il *parallelismo* è il massimo speedup possibile $\frac{T_1}{T_\infty}$.
Quindi, conoscendo $T_1$ e $T_\infty$ e volendo conoscere $T_P$ per un dato $P$ (ad esempio, $P = 4$), ignorando il sistema di caching di *ForkJoin*, $T_P = \Omega(\frac{T_1}{P})$ e $T_P = \Omega(T_\infty)$.
Infatti, l'esecuzione asintoticamente ottimale sarebbe: $T_P = \Theta(\frac{T_1}{P+T_\infty})$.
In sintesi, il framework *ForkJoin* è capace di dare una garanzia sul tempo di esecuzione previsto asintoticamente ottimale.

**Raccomandazioni**

1. tutti i thread che vengono creati devono fare orientativamente la stessa quantità di lavoro;
2. tutti i thread devono fare un piccolo task.

# Capitolo 3

# Capitolo 4

## 3.1 Pattern paralleli più complessi

Il problema della somma dei prefissi sembra un problema senza soluzione parallelizzabile.
Partiamo dal problema: dato un input $int[]$, bisogna produrre un output $int[]$, dove:

$$output[i] = input[0] + input[1] + \ldots + input[i]$$

Un ipotetico codice sequenziale potrebbe dettare come segue:

```
int[] prefix_sum(int[] input) {
        int[] output = new int[input.length];
        output[0] = input[0];
        for (int i=1; i<input.length; i++)
                output[i] = output[i-1] + input[i];
        return output;
}
```

Non sembra essere un problema parallelizzabile perché:

- il $DAG$ è una catena, a causa delle dipendenze tra nodi;
- ha un *work* pari a $O(n)$ e uno *span* $O(n)$.

In ogni caso, un algoritmo differente può raggiungere *span* $O(\log n)$).

### 3.1.1 Hillis & Steele (1986)

L'idea è che all'inizio di ciascuna iterazione $i$, per $i \geq 0$; ogni elemento $A[x]$ dell'array contiene la somma di al massimo $2^i$ elementi precedenti, incluso $input[x]$).

**Caso base** Sicuramente nel primo caso, per $i = 0$, è verificato, in quanto vi sono elementi singoli.

**Passo induttivo** Se la proprietà è vera per l'iterazione $i$, vuol dire che è vera anche per l'iterazione $i+1$:

- durante l'iterazione $i+1$, settiamo $A[x] = A[x] + A[x-2^i]$;
- per ipotesi induttiva: $A[x] = input[x]+input[x-1]+\ldots+input[x-(2^i-1)]$;
- per ipotesi induttiva: $A[x-2^i] = input[(x-2^i)] + input[(x-2^i)-1] + \ldots + input[(x-2^i)-(2^i-1)]$;
- quindi, dopo l'iterazione $i+1$, $A[x]$ conterrà la somma (al massimo) dei suoi $2^i + 2^i = 2^{i+1}$ elementi immediatamente precedenti.

### Analisi dell'algoritmo

Avremo $\log n$ iterazioni, quindi $span = O(\log n)$; inoltre, ogni iterazione avrà $work = O(n)$, per un totale di $\Theta(n \times \log n)$.
Quindi il *work* è più alto rispetto alla soluzione sequenziale, ma si riesce a parallelizzare. La soluzione *parallel-prefix* ha un parallelismo di $\frac{n}{\log n}$, quindi, speedup esponenziale.
Questo tipo di soluzione, compie due passi:

1. costruisce un albero in un movimento *bottom-up*, il passo *up*: crea un albero binario, dove la radice ha somma nell'intervallo $[0, n)$ e dove se un nodo ha somma nell'intervallo $[lo, hi)$ e $hi > lo$, allora:
   - il figlio sinistro ha somma in $[lo, middle)$;
   - il figlio destro ha somma in $[middle, hi)$;
   - una foglia ha somma in $[i, i+1)$, per esempio, $input[i]$.

   Questa è una facile computazione *fork-join*: combina risultati dalla costruzione di un albero binario con tutte le somme degli intervalli (con *work* $O(n)$ e *span* $O(\log n)$).
2. attraversa l'albero in un movimento *top-down*, il passo *down*, operando su un valore *fromLeft*:
   - la radice da un valore *fromLeft* di 0;
   - il nodo prende questo valore *fromLeft* e:
     - passa lo stesso al figlio sinistro;
     - passa $fromLeft$ sommato alla somma del figlio sinistro al figlio destro.
   - alla foglia in posizione $i$: $output[i] = fromLeft + input[i]$.

   Ancora, sembra una facile computazione *fork-join*: attraversare l'albero costruito nel primo passo e non produrre nessun risultato (con *work* $O(n)$ e *span* $O(\log n)$).

Ancora, aggiungere un *sequential cutoff* è molto semplice: per la fase *up*, si tratta di una semplice somma, per cui il nodo foglia mantiene la somma di un intervallo; per la fase *down*:

```
output[lo] = fromLeft + input[lo];
for (i=lo+1; i<hi; i++)
        output[i] = output[i-1] + input[i];
```

### 3.1.2 Problemi con condizione di verifica

Dato un array *input*, produrre un array *output* contenente solo gli elementi per cui $f(element)$ è *true*.
Esempio:

| | |
|---|---|
| **input** | $[17, 4, 6, 8, 11, 5, 13, 19, 0, 24]$ |
| **f** | $element > 10$ |
| **output** | $[17, 11, 13, 19, 24]$ |

E' parallelizzabile? Rintracciare gli elementi è molto semplice (attraverso una mappa), ma metterli al giusto posto sembra complesso:

1. Mappa parallela per calcolare un bit-vector per gli elementi per cui $element > 10 = true$;

| | |
|---|---|
| **input** | $[17, 4, 6, 8, 11, 5, 13, 19, 0, 24]$ |
| **bits** | $[1, 0, 0, 0, 1, 0, 1, 1, 0, 1]$ |

2. prefix-sum parallelo del bit vector:

| | |
|---|---|
| **bitsum** | $[1, 1, 1, 1, 2, 2, 3, 4, 4, 5]$ |

3. mappa parallela per produrre l'output:

| | |
|---|---|
| **output** | $[17, 11, 13, 19, 24]$ |

```
output = new array of size bitsum[n-1]
FORALL (i=0; i<input.length; i++) {
        if (bits[i] == 1)
                output[bitsum[i]-1] = input[i];
}
```

**Analisi** L'algoritmo ha un *work* $O(n)$ e *span* $O(\log n)$.

### 3.1.3 QuickSort parallelo

| # | **passo** | **costo** |
|---|---|---|
| 1 | Prendere un elemento *pivot* | $O(1)$ |
| 2 | Partizionare i dati in $A = elem < pivot$, $B = pivot$, $C = elem > pivot$ | $O(n)$ |
| 3 | Ordinare ricorsivamente $A$ e $C$ | $2T(\frac{n}{2})$ |

L'approccio più semplice è eseguire il terzo passo in parallelo, con i seguenti risultati: *work* $\Theta(n \times \log n)$ inalterato, *span* $T(n) = \Theta(n) + 1T(\frac{n}{2}) = \Theta(n)$ alterato e quindi *parallelismo* $O(\log n)$.
Avere uno speedup di $O(\log n)$ con un numero infinito di processori non è un granché, si può fare molto meglio se parallelizzato anche la fase di partizionamento:

- si prenda come *pivot* il mediano dell'albero;
- inserire gli elementi minori del *pivot* in un array ausiliario *aux*, poi il *pivot* e poi i maggiori;
- lanciati i sort paralleli ricorsivamente sui due subarray.

Per ogni *pack*, abbiamo un *work* di $O(n)$ e *span* $O(\log n)$. Con $O(\log n)$ di *span* per il partizionamento, il miglior caso in assoluto e l'atteso per il *quicksort* è:

$$T(n) = O(\log n) + 1T(\tfrac{n}{2}) = O(\log^2 n)$$

### 3.1.4 MergeSort parallelo

| # | **passo** | ***work* nel *worst-case*** |
|---|---|---|
| 1 | Ordinare la metà di sinistra e la metà di destra | $2T(\frac{n}{2})$ |
| 2 | Unire i risultati | $\Theta(n)$ |

Esattamente come nel *quicksort*, eseguire i due ordinamenti in parallelo cambia lo *span* in $T(n) = O(n) + 1T(\frac{n}{2}) = O(n)$. Ancora, il *parallelismo* è di $O(\log n)$. Perciò, per migliorare, occorre parallelizzare il *merge* dei due risultati (non utilizzando il *parallel-prefix, questa volta*).
Dovendo unire i due subarray, consideriamo che quello di grandezza maggiore dei due abbia $m$ elementi. Quindi, in parallelo:

- unire i primi $m/2$ elementi della metà più grande con gli elementi appropriati della metà minore;
- unire i secondi $m/2$ elementi della metà più grande con il resto della metà minore.

Questo approccio porta a un *work* di:

$$W(n) = W(\tfrac{3n}{4}) + W(\tfrac{n}{4}) + O(\log n) = O(n)$$

Per lo *span* invece:

$$S(n) = S(\tfrac{3n}{4}) + O(\log n) = O(\log^2 n)$$

Nel caso sequenziale: $T(n) = 2T(\frac{n}{2}) + \Theta(n) = O(n \times \log n)$.
Quindi, rispetto al sequenziale, il parallelo mantiene lo stesso *work*, ma lo *span* è pari a: $T(n) = 1T(\frac{n}{2}) + \Theta(n) = \Theta(n)$.
Utilizzando il parallel merge:

- *work*: $T(n) = 2T(\frac{n}{2}) + \Theta(n) = \Theta(n \times \log n)$;
- *span*: $T(n) = 1T(\frac{n}{2}) + \Theta(\log^2 n) = \Theta(\log^3 n)$;
- *parallelismo*: $O(\frac{n}{\log^2 n})$.

Non è ottimale come il *quicksort*, ma garantisce quantomeno un *worst-case*.

# Capitolo 4

# Capitolo 5

## 4.1 Concorrenza

Gli algoritmi hanno e devono avere sempre una struttura molto semplice per evitare che si verifichino *race conditions*, gestendo nella maniera migliore possibile gli accessi (anche simultanei) alle risorse condivise tra diversi clienti.
Questo richiede coordinazione e sincronizzazione per evitare che accessi simultanei, facendo in modo tale che qualcuno si *blocchi*, fino a che il thread che ha acceduto non ha finito di usare la risorsa.
In questi casi, i thread non sono utili ai fini del parallelismo, ma per avere maggiore responsività nella struttura del codice (come la risposta a eventi *GUI*), maggior sfruttamento della *CPU* e isolamento degli errori.

### 4.1.1 Condivisione

E' molto comune nei programmi concorrenti che diversi thread possano accedere alle stesse risorse in memoria, ma questo può causare sovrapposizioni nell'accesso e modifica delle stesse.
Si introduce quindi la *mutua esclusione* nella *sezione critica* (ovvero dove si accede/modifica quella risorsa di memoria), utilizzabile attraverso il linguaggio in uso. Una soluzione di base sono i *lock*, che seguono tre fasi fondamentali:

- *new*: crea un nuovo *lock*, con stato *not held*;
- *acquire*: acquisice un *lock*, portandolo allo stato *held*;
- *release*: porta il *lock* di nuovo a *not held*.

Quindi l'istanza *lock* sarà posta ai margini della *sezione critica* nel codice, prima chiamando l'*acquire()* e dopo il *release()*.
In ogni caso, l'uso dei *lock* non è granché comodo, perché prevede che si tenga sempre conto dello stato del *lock*, casa scarsa performance e come lavora sui

*setters* e *getters*?
Una parziale soluzione potrebbe essere usare i *re-entrant lock* (o *recursive lock*), che detiene sia il thread che lo sta bloccando e un conteggio. L'*acquire()* ha successo se e solo se il thread che lo vuole bloccare è lo stesso che lo sta richiedendo, in questo caso viene incrementato il conteggio. Nel *release()*, se il conteggio è $> 0$, il conteggio viene decrementato, e una volta a 0, lo stato del *lock* viene posto a *not held*.

### 4.1.2 Java

*Java* supporta i re-entrant locks in maniera primitiva, tramite lo statement builtin *synchronized*, che valuta l'espressione in input (perché ogni oggetto in *Java*, ad eccezione dei primitivi, è un *lock*), acquisisce il *lock*, entrando nella parentesi graffa "{", bloccandolo se necessario, e lo rilascia uscendo dalla parentesi graffa "}".

```
class BankAccount {

        private int balance = 0;
        private Object lk = new Object();

        int getBalance() {
                synchronized (lk) { return balance; }
        }

        void setBalance(int x){
                synchronized (lk) { balance = x; }
        }

        void withdraw(int amount) {
                synchronized (lk) {
                        int b = getBalance();
                        if (amount > b) {
                                setBalance(b - amount);
                        }
                }
        }
}
```

Altro metodo di utilizzare il *synchronized* è di porlo come scopo sul metodo:

```
class BankAccount {

        private int balance = 0;

        synchronized int getBalance() { return balance; }
        synchronized void setBalance(int x) { balance = x; }

        synchronized void withdraw(int amount) {
                int b = getBalance();
                if(amount > b) {
                        setBalance(b - amount);
                }
        }
}
```

I *lock* sono privati, e in questo modo prevengono il fatto che il codice in altre classi possa eseguire operazioni sensibili.

Una *race condition* si verifica quando il risultato computazionale dipende dallo scheduling, e quindi due thread si sovrappongono, e rappresenta un bug esistente solo a causa della concorrenza. Due tipi di *race condition* diversi sono i *data races* e i *bad interleavings*: la differenza sostanziale sta nel fatto che un *data race* si verifica quando si presentano due accessi *read/write* o *write/write* alla stessa locazione di memoria simultaneamente. Dl'altra parte, una *bad interleaving*, invece, è causata da l'esposizione in sezioni critiche di dati in uno stato intermedio inconsistente, nonostanza di *data race*.
E' quindi responsabilità del programmatore evitare che si verifichino *data races*.

#### Variabili volatili

Oltre alle soluzioni già proposte, esiste poi un altro metodo, quello di utilizzare le variabili *volatili*: questo sistema forza l'applicazione a non fare *caching* delle variabili inizializzate come volatili, così da necessitare un accesso diretto alla memoria per ogni interazione con le suddette. E' un sistema intelligente e più efficiente rispetto alla *synchronized*, ma da evitare in caso di lunghe sezioni critiche.

### 4.1.3 Utilizzare correttamente i locks

Per ogni locazione di memoria, occorre osservare almeno una delle seguenti regole:

1. *thread-local*: non utilizzare la variabile in più di un thread: dove possibile, sempre meglio evitare di condividere risorse, ma questo è possibile solo se il thread non deve comunicare attraverso di esse;

2. *immutabile*: non scrivere nella locazione di memoria: se il thread utilizza una locazione di variabili in lettura per tutta la sua esecuzione, allora è

inutile complicarsi la vita, non c'è bisogno di sincronizzazione; in tal caso è utile passare nuovi oggetti ai thread;

3. *synchronized*: utilizzare la sincronizzazione per controllare gli accessi.

Una volta appurato che rispettiamo adeguatamente le prime due regole, occorre sapere come mantenere i dati consistenti, evitando *data races*:

1. non permettere mai che due thread accedano simultaneamente in *read/write* o *write/write* alla stessa loaczione;
2. per ciascuna locazione di memoria che necessita di sincronizzazione, controllare che sia sempre gestita nelle fasi di *read* o *write*, utilizzando *lock guards*;
3. cominciare sempre con una politica di *coarse-grained* e muoversi verso la *fine-grained* soltanto se specificatamente richiesto e/o necessario. Esistono infatti due politiche di *locking*:
   - *coarse-grained*: utilizzando meno *locks*, ciascuno per più oggetti:
     - è più semplice da implementare;
     - più semplice gestire le operazioni che accedono alle locazioni di memoria.
   - *fine-grained*: utilizzando più *locks*, ciascuno per meno oggetti:
     - più accessi contemporanei (più performante rispetto al *coarse-grained*, che invece bloccherebbe anche quando non necessario).
4. evitare di fare grandi computazioni o attività I/O nelle sezioni critiche, ma fare in ogni caso attenzione a dare - in questo modo - accessi alternati in lettura scrittura di tipo *A-B-A*;
5. pensare sempre in termini di quali operazioni necessitano di essere atomiche, e formare le sezioni critiche limitatamente a quelle operazioni;
6. utilizzare librerie builtin ogni volta che vengono in tuo aiuto.

## 4.2 Deadlocks

Una *deadlock* si presenta quando ci sono $T_1, \ldots, T_n$ thread tali che:

- per ciascuna $i = 1, \ldots, n-1$, $T_i$ è in attesa di una risorsa bloccata da $T_{i+1}$;
- $T_n$ è in attesa di una risorsa bloccata da $T_1$.

In altre parole, si tratta di un ciclo di attese; pertanto, evitare in programmazione di andarci incontro, significa evitare che si presenti mai un possibile ciclo. Quindi, ancora, occorre cercare di rendere le sezioni critiche più piccole possibile, utilizzare più possibile una granularità *coarsen* dei locks e fare in modo tale che ogni lock venga acquisito sempre nello stesso ordine.

## 4.3 Lock lettore/scrittore

In questo caso il *lock* può essere di tre tipi:

1. non acquisito;
2. acquisito per scrittura da un thread;
3. acquisito per lettura da uno o più thread.

Quindi, avrà i seguenti metodi:

- *new()*: inizializzerà un nuovo *lock* con stato *non acquisito*;
- *acquire_write()*: attende se lo stato corrente del *lock* è *acquisito per lettura* o *acquisito per scrittura*, altrimenti lo rende *acquisito per scrittura*;
- *release_write()*: rende lo stato del *lock non acquisito*;
- *acquire_read()*: attende se lo stato corrente del *lock* è *acquisito per scrittura*, altrimenti lo rende/mantiene *acquisito per lettura* e incrementa il numero di lettori;
- *release_read()*: decrementa il numero di lettori e, se questo è 0, rende lo stato del *lock non acquisito*.

Quindi, nel problema *lettore/scrittore*, tendenzialmente viene data la priorità agli scrittori. Il *synchronized* di *Java* non supporta il lock *lettore/scrittore*, ma esiste la libreria *java.util.concurrent.locks.ReentrantReadWriteLock*:

- non hanno stessa interfaccia: per questo i metodi *readLock()* e *writeLock()* ritornano oggetti che hanno a loro volta i metodi *lock()* e *unlock()*;
- non da priorità allo scrittore.