Programmazione di Sistemi Multicore

Davide Pucci

2016-2017

Indice

Capitolo 1

Background Teorico

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 processori è definito come:

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

Dove:

- t_s è il tempo di esecuzione su un solo processore;
- 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 (derivata dalla legge di **Amdahl**);
- *scalabilità debole* (*weak scaling*): vogliamo sapere se possiamo analizzare più dati approssimativamente nello stesso tempo, incrementando il numero di processori (derivata dalla legge di *Gustafson*).

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 diverse istruzioni su dati differenti.

Esistono due forme primarie di scambio dati tra task paralleli, seguite da altre ibride più evolute:

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 utilizzando *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

Fork/Join

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 quella del *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 (associate ad un'esecuzione sequenziale);

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 altri 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 indipendentemente su una collezione, 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 opensource, *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_∞.

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_∞ 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

Pattern

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)

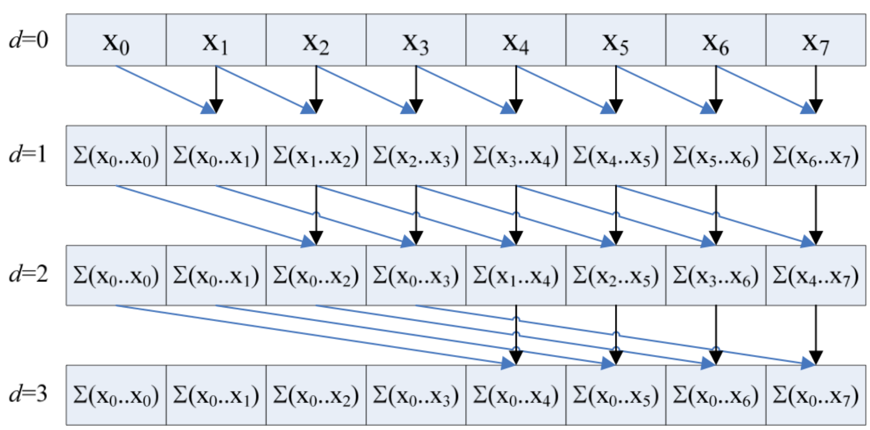
L'idea è che all'inizio di ciascuna iterazione i, per $i \geq 0$; ogni elemento $A[x]$ dell'array contiene la somma di al massimo 2^i elementi precedenti, incluso $input[x]$).

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 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.



Prefix sum (parallel-prefix)

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 Packs: 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	$2T(\frac{n}{2})$

ha *equazione di ricorrenza* pari a: $R(n) = O(n) + 2 \times R(\frac{n}{2})$
Per il *Teorema master*: $O(n \times \log n)$
Il *quicksort* segue tre fasi: scelta dell'elemento *pivot* (che assumeremo sia sempre l'elemento mediano), partizionamento e ordinamento.
Per applicare una parallelizzazione seguiremo due soluzioni:

1. rendere le chiamate sui due array ricorsive, ottenendo un miglioramento tale: *work* inalterato, *span* pari a $O(n) + 2 \times R(\frac{n}{2}) \Rightarrow O(n) + R(\frac{n}{2})$

2. ottimizzazione dell'operazione di partizionamento, utilizzando il *packing*. Viene lanciato un primo *packing* la cui condizione è che gli elementi siano $< pivot$, e poi un secondo per cui la condizione sia che gli elementi siano $> pivot$.
   Questo ha *work* $O(n)$ (perchè il *work* nei *packing* è tale) e *span* $O(n) + R(\frac{n}{2}) \Rightarrow O(\log n) + R(\frac{n}{2})$

Con queste modifiche arriviamo a:

- *work*: $O(n)$
- *span*: $O(\log^2 n)$

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	$\Theta(n)$

Il *MergeSort* generico ha *equazione di ricorrenza*: $R(n) = O(n) + 2 \times R(\frac{n}{2})$, che per il *Teorema master* è pari a $O(n \times \log n)$. Le due soluzioni per applicare la parallelizzazione sono le seguenti:

1. Utilizzare la parallelizzazione nelle chiamate ricorsive, portando ad un miglioramento tale: $R(n) = O(n) + 2 \times R(\frac{n}{2}) \Rightarrow O(n) + R(\frac{n}{2})$

2. Ottimizzare l'operazione di *merge*: quello che accade è che l'array viene splittato in due sottoarray minori o di pari dimensioni o di una differenza di dimensione di 1. In quest'ultimo caso, si sceglie un indice i dall'array più grande perchè referenzi l'elemento di mezzo. Nell'array minore, viene scelto un indice j tale che, applicata la *ricerca binaria* (di costo $O(\log n)$), alla sua sinistra abbia tutti elementi minori di quello a cui punta l'indice i nell'array maggiore. A questo punto vengono lanciate due chiamate ricorsive - parallelamente -, prendendo, nel primo caso, gli array di sinistra, nell'array minore e maggiore, e nel secondo caso quelli di destra.
   Poichè l'array maggiore sarà diviso sempre in subarray di dimensione $n/4$ e poichè, nel caso limite, nell'array maggiore, ci troveremo in una situazione tale che il subarray di sinistra avrà dimensione 0 e quello di destra $3/4 \times n$, allora: $R(n) = O(\log n) + R(\frac{n}{4}) + R(\frac{3 \times n}{4}) = O(n)$.
   Applicando poi la parallelizzazione anche su queste chiamate ricorsive: $R(n) = O(\log n) + R(\frac{3 \times n}{4}) = O(\log^2 n)$
   Quindi:

   - *work*: $O(n) + 2 \times R(\frac{n}{2}) = n \times \log n$
   - *span*: $O(\log^2 n) + R(\frac{n}{2}) = O(\log^3 n)$
   - *parallelismo*: $\frac{n}{\log^2 n}$

Questo approccio porta a un *work* di:

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

Per lo *span* invece:

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

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

Concorrenza e Deadlocks

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*, causa scarsa performance e occorre sapre come

gestire coerentemente i *setters* e *getters*.
Una parziale soluzione potrebbe essere usare il *re-entrant lock* (o *recursive lock*), che detiene sia il thread che lo sta bloccando e un contatore. 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 contatore. Nel *release()*, se il conteggio è > 0, il contatore 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. D'altra parte, una *bad interleaving*, invece, è causata da l'esposizione in sezioni critiche di dati in uno stato intermedio inconsistente, nonostante la presenza 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 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 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 dà priorità allo scrittore.

4.4 Variabili condizione

Si parte da un esempio canonico: la necessità di dover condividere lavoro tra thread, attraverso un coda di dimensione fissa, in cui i (thread) produttore accodano il risultato del loro lavoro e i (thread) consumatori lo prelevano per elaborarlo.

Per fare in modo che questa logica funzioni, occorre *sincronizzazione* nell'accesso alla coda:

```
class Buffer<E> {

        E[] array = (E[]) new Object[SIZE];

        synchronized void enqueue(E elt) {
                if (isFull()) {
                        ...
                } else {
                        /* add to array and adjust back */
                }
        }
        synchronized E dequeue(E elt) {
                if (isEmpty()) {
                        ...
                } else {
                        /* take from array and adjust front */
                }
        }

}
```

Ma nei casi di *isFull()* e *isEmpty()*? La cosa migliore sarebbe lasciare in attesa il richiedente e notificarlo quando la risorsa richiesta è disponibile, senza lanciare eccezioni ma soprattutto, senza lasciare che la risorsa rimanga bloccata nel frattempo.
Per questa ragione vengono utilizzate le *variabili condizione*, per notificare thread in attesa quando la condizione che causa la loro attesa è variata. Quindi:

```
synchronized void enqueue(E elt) {
        if (isFull()) {
                this.wait(); // release lock and wait
        }
        /* add to array and adjust back */
        if (!isFull()) {
                this.notify(); // wake somebody up
        }
}
synchronized E dequeue(E elt) {
        if (isEmpty()) {
                this.wait(); // release lock and wait
        }
        /* take from array and adjust front */
        if (!isEmpty()) {
                this.notify(); // wake somebody up
        }
}
```

Java Per *Java* ogni oggetto è una *variabile condizione* (e un *lock*). Altri linguaggi spesso usano diversificare i due concetti.

Nell'ultimo esempio, quindi, vengono introdotti *wait()* e *notify()*:

- *wait()*: registra il thread richiedente come interessato e poi *atomicamente* rilascia il *lock* sulla risorsa e blocca il thread; quando l'esecuzione riprenderà, la prima operazione sarà quella di riacquisire il *lock*;
- *notify()*: prende uno dei thread in attesa e lo sveglia, se ce ne sono.

Problema #1 Nel tempo che intercorre tra la riattivazione di un thread e la sua riacquisizione del *lock*, potrebbe essere che la *variabile condizione* diventi falsa ancora. Per ovviare a questo problema, occorre ricontrollare sempre la condizione dopo aver ottenuto il *lock*:

```
synchronized void enqueue(E elt) {
        while (isFull()) {
                this.wait(); // release lock and wait
        }
        ...
}
synchronized E dequeue(E elt) {
        while (isEmpty()) {
                this.wait(); // release lock and wait
        }
        ...
}
```

Problema #2 Anche se ci sono più threads in attesa, ne viene svegliato sempre solo uno. La soluzione al problema è molto semplice: avvertire sempre tutti i thread, attraverso *notifyAll()*:

```
synchronized void enqueue(E elt) {
        ...
        if (!isFull()) {
                this.notifyAll(); // wake everybody up
        }
}
synchronized E dequeue(E elt) {
        ...
        if (!isEmpty()) {
                this.notifyAll(); // wake everybody up
        }
}
```

Altro approccio potrebbe essere chiamare, senza condizione *if*, sempre il *notifyAll()* (o *notify()*), ma questo sarebbe utile soltanto se l'*enqueuer* e il *dequeuer* aspettassero su *variabili condizione* diverse.
In ogni caso, *Java* non sembra supportarlo, ma la classe *ReentrantLock* di *java.util.concurrent.locks* contiene un metodo *newCondition()* che ritorna un nuovo oggetto *Condition* associato al *lock*.

4.5 Definizioni

Alcuni definizioni utili:

- *Race condition*: fenomeno in cui il risultato finale dell'esecuzione di diversi thread dipende dalla temporizzazione e/o dalla sequenza con cui i thread vengono eseguiti.
  - *Data race*: più thread cercano di modificare una risorsa condivisa simultaneamente. A causa delle operazioni di scheduling che potrebbero mettere in *swap* i thread in qualsiasi momento, non è possibile prevedere in maniera accurata in quale ordine effettivo i thread cercheranno di accedere alla risorsa.
  - *Bad interleaving*: stato inconsistente derivato dall'esposizione di uno stadio intermedio in un'operazione all'interno di una sezione critica. Esempio: operazione di *add* ad una *stack*, che prima aggiunge un elemento all'array di *backing* interno e poi incrementa il campo *size*.
- *Mutual exclusion*: la mutua esclusione è un meccanismo che permette l'accesso ad una risorsa da al più un thread per volta.
- *Deadlock*: situazione in cui due (o più) thread che condividono la stessa risorsa non permettono l'un l'altro di accedere alla stessa, creando una catena e portando ad un blocco definitivo dell'esecuzione di entrambi.

Capitolo 5

OpenCL

5.1 Memory

In *OpenCL*, esistono due tipi di *Memory object* (per accessi ad una regione della memoria globale):

1. oggetto *Buffer*: un semplice array, completamente esposto tra *kernel* ed accessibile attraverso puntatori;

2. oggetto *Image*: definisce una regione della memoria bi(o tri)-dimensionale, con strutture dati accessibili solo con funzioni *read()* e *write()*, molto utili per interfacciarsi con API grafiche come *OpenGL*.

5.1.1 Buffer

I *buffer* si dichiarano nell'host con tipo *cl_mem*.

Esempio Per creare un *buffer* *d_X*, copiare i *byte* da *h_X* a *d_X* e copiarlo in memoria:

```
cl_mem d_X = clCreateBuffer(context, CL_MEM_READ_ONLY |
    CL_MEM_COPY_HOST_PTR, sizeof(float)*count, h_X, NULL);
```

Per evitare confusione nel capire se una variabile *host* è un semplice array *C* o un *buffer* *OpenCL* si usa la convenzione del prefisso ***h_*** per indicare array *C* regolari dell'**h**ost e di quello ***d_*** per indicare buffer del **d**evice.
Esistono altre *flag* per il tipo di memoria, come *CL_MEM_WRITE_ONLY* e *CL_MEM_READ_WRITE*.
Per copiare il buffer sulla memoria host in *h_Y* (*CL_TRUE* è bloccante, *CL_FALSE* non è bloccante):

```
clEnqueueReadBuffer(queue, d_Y, CL_TRUE, 0, sizeof(float)*count,
    h_Y, 0, NULL, NULL);
```

5.2 Program

Gli oggetti *program* contengono un *context*, i sorgenti (o il binario) del *kernel* e la lista dei *target devices* e le opzioni di build. Le API *C* creano un oggetto program con: *clCreateProgramWithSource()* e/o *clCreateProgramWithBinary()*.

OpenCL usa la compilazione a runtime, perché in generale non può sapere i dettagli del *target device* quando il programma viene compilato.

5.2.1 Creazione e build del programma

Il codice sorgente del *kernel* può essere definito sia come una stringa sia come da leggere da file. Quindi il *cl_program* incapsula codice sorgente e la sua ultima build riuscita. Per compilare il programma:

```
cl_program program = clCreateProgramWithSource(context, 1, (const
    char**) &KernelSource, NULL, &err);
```

La compilazione del programma crea una libreria dinamica da cui possono essere prelevati *kernel* specifici:

```
err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
// check errors
if (err != CL_SUCCESS) {
        size_t len;
        char buffer[2048];
        clGetProgramBuildInfo(program, device_id,
            CL_PROGRAM_BUILD_LOG, sizeof(buffer), buffer, &len);
        printf("%s\n", buffer);
}
```

5.3 Kernel

Per creare un oggetto *kernel*:

```
kernel = clCreateKernel(program, "square", &err);
// set kernel parameters
err  = clSetKernelArg(kernel, 0, sizeof(cl_mem), &d_X);
err |= clSetKernelArg(kernel, 1, sizeof(unsigned int), &count);
```

Dopodiché, per accodare comandi, occorre scrivere dei *buffer* dall'host alla memoria globale:

```
err = clEnqueueWriteBuffer(commands, d_X, CL_FALSE, 0, sizeof(float
    )*count, h_X, 0, NULL, NULL);
```

Poi, per accodare il *kernel* per l'esecuzione:

```
err = clEnqueueTask(commands, kernel, 0, NULL, NULL);
// or
err = clEnqueueNDRangeKernel(commands, kernel, 1, NULL, &global, &
    local, 0, NULL, NULL);
```

Per leggere il risultato dei comandi:

```
err = clEnqueueReadBuffer(commands, d_X, CL_TRUE, sizeof(float)*
    count, h_X, 0, NULL, NULL);
```

5.4 Panoramica Host

In generale il codice *host* si suddivide delle seguenti fasi:

1. definizione di piattaforma e code;
2. definizione di oggetti in memoria;
3. creazione *programma*;
4. compilazione *program*;
5. creazione e configurazione *kernel*;
6. esecuzione *kernel*;
7. lettura dei risultati.

5.5 Parallelismo in OpenCL

L'idea di fondo è l'esecuzione della funzione *__kernel* per ogni punto del dominio del problema, invece di utilizzare i *loop*. Quindi, il seguente codice con *loop* tradizionale:

```
void mul(const int n, const float *a, const float *b, float *c) {
        int i;
        for (i = 0; i < n; i++) {
                c[i] = a[i] * b[i];
        }
}
```

si parallelizza come segue:

```
// tutte le iterazioni del loop vengono chiamate simultaneamente ed
     eseguite in parallelo
__kernel void mul(__global const float *a, __global const float *b,
     __global float *c) {
        int id = get_global_id(0);
        c[id] = a[id] * b[id];
}
```

5.5.1 Dimensionalità del problema

Occorre definire la dimensionalità del problema in relazione al priblema stesso. Quando viene eseguito il *kernel* vengono specificate al massimo 3 dimensioni, per ciascuna delle quali viene specificata a sua volta la grandezza totale del problema, chiamata la *global size*.
Ogni singola istanza del *kernel* nell'*NDRange* è un *work-item* e ognuno di essi esegue lo stesso *kernel* su dati diversi. Ogni *work-item* hanno *global ID* unico.
Ogni *work-item* fa parte di un *work-group*, con cui condivide la memoria locale e con cui può sincronizzarsi. Si può specificare il numero di *work-item* in un *work-group*, e questo numero prende il nome di *local size* (locale al *work-group*) (in alternativa, lo sceglie la *runtime OpenCL*, non in maniera ottimale).
Ogni *work-group* ha un *work-group ID* univoco e i suoi *work-items* hanno anche loro un *ID* univoco locale al *work-group*.

5.5.2 OpenCL C

L'*OpenCL C* è derivato dall'*ISO C99*, e presenta alcune restrizioni: non ha ricorsioni, né puntatori a funzioni, ...
I tipi builtin sono:

- tipi *scalari*: *char*, *uchar*, *short*, *ushort*, *int*, *uint*, *long*, *ulong*, *float*, *bool*, *intptr_t*, *ptrdiff_t*, *size_t*, *uintptr_t*, *void*, ...;
- tipi *immagine*: *image2d_t*, *image3d_t* e *sampler_t*;
- tipi *vettore* e *puntatori*;
- funzioni di conversione *data-type*.

Esistono poi diversi qualificatori:

- qualificatori di funzione: *__kernel*, che dichiara una funzione come un *kernel*;
- qualificatori di indirizzi: *__global*, *__local*, *__constant*, *__private*, tutti necessari se argomenti di una funzione *__kernel*.

Ancora, alcune funzioni per *work-items*: *get_work_dim()*, *get_global_id()*, *get_local_id()*, *get_group_id()*.
Per la sincronizzazione vengono invece utilizzate le *barriere* (tutti i *work-items* in un *work-group* devono eseguire la funzione *barrier()* prima che ogni *work-item* possa proseguire) e le *memory fences* (che garantiscono ordinamento nelle operazioni in memoria).
Infine, non sono permessi *puntatori a funzione*, i *puntatori a puntatori* sono permessi all'interno di un *kernel*, ma non come argomento allo stesso, i campi *bit* non sono supportati, così come gli array e le strutture a lunghezza variabile, così come la riscorsione, e i tipi *double* sono opzionali in *OpenCL 1.1*, ma in ogni caso è una *key word* riservata.

Capitolo 6

Esercizi

6.1 Legge di Amdahl e Gustafson

6.1.1 Domanda 1

Sono forniti dua algoritmi per risolvere un problema su input di dimensione n: A_1 esegue n^3 operazioni, con il 40% di tempo di esecuzione seriale; A_2 esegue n^4 operazioni, con lo 0.1% di tempo di esecuzione seriale.

1. Se A_1 viene eseguito su una piattaforma con 2 processori e $n = 10^2$, di quanti processori P abbiamo bisogno perchè l'esecuzione di A_2 su n risulti più veloce usando P processori?

   Soluzione $T_P = T(f) + T(s) = f + (1-f)/n$. Quindi, con $n = 10^2$:
   $T_{P_1} = \frac{40}{10^2} \times n^3 + \frac{\frac{60}{10^2} \times n^3}{2} = \frac{40}{10^2} \times 10^6 + \frac{\frac{60}{10^2} \times 10^6}{2} = 40 \times 10^4 + 30 \times 10^4 = 70 \times 10^4 = 7 \times 10^5$.
   $T_{P_2} = \frac{0{,}1}{10^2} \times n^4 + \frac{\frac{99{,}9}{10^2} \times n^4}{P} = \frac{1}{10^3} \times 10^8 + \frac{\frac{999}{10^3} \times 10^8}{P} = 10^5 + \frac{999}{P} \times 10^5 = 10^5 \times (1 + \frac{999}{P})$.
   $T_{P_2} \leq T_{P_1} \Rightarrow 10^5 \times (1 + \frac{999}{P}) \leq 7 \times 10^5 \Rightarrow 1 + \frac{999}{P} \leq 7 \Rightarrow \frac{999}{P} \leq 6 \Rightarrow \frac{1}{P} \leq \frac{6}{999} \Rightarrow P \geq \frac{333}{2}$.

2. Qual è lo speedup di A_2 su 100 processori in base alla legge di *Amdahl*?

   Soluzione $S(n) \leq \frac{1}{f + \frac{1-f}{n}}$. Quindi:
   $S(100) = \frac{1}{\frac{1}{1000} + \frac{1 - \frac{1}{1000}}{100}} = \frac{1}{\frac{1}{1000} + \frac{\frac{999}{1000}}{100}} = \frac{1}{\frac{1+99900}{1000}} = \frac{1000}{99901} \simeq \frac{1}{10}$.

3. Per lo speedup, è importante che il tempo di esecuzione di A_2 sia n^4?

   Soluzione No, perché nella derivazione della legge di *Amdahl* il tempo si semplifica: $S(n) = \frac{t_s}{t_n} \leq \frac{t_s}{f \times t_s + \frac{(1-f) \times t_s}{n}} = \frac{1}{f + \frac{1-f}{n}}$

6.1.2 Domanda 2

Rispondete alle seguenti domande, motivando le risposte:

1. Un programma ha efficienza 50% usando 16 processori. Qual è la percentuale di codice seriale del programma in base alla legge di *Amdahl*?

   Soluzione $E(n) = \frac{S(n)}{n} = 50\%$
   $\frac{1}{n} \times \frac{1}{f + \frac{1-f}{n}} = \frac{5}{10} \Rightarrow n \times (f + \frac{1-f}{n}) = \frac{10}{5} \Rightarrow n \times f + 1 - f = 2 \Rightarrow 16f + 1 - f = 2 \Rightarrow 15f = 1 \Rightarrow f = \frac{1}{15}$

2. Se un programma P_1 ha uno speedup maggiore di un programma P_2 e i due programmi eseguono lo stesso work, è vero che P_1 è sempre veloce almeno quanto P_2? Se l'affermazione è vera dimostratela, altrimenti fornite un controesempio.

   Soluzione La consegna è ambigua:

   - Se per *work* si intende lo speedup $S(1)$ (ovvero, in contrapposizione allo *span*), allora sì, è vero.
   - Se per *work* si intende il *workload*, allora dipende: dipende dal numero di processori e, in un caso di parità in sede di esecuzione degli algoritmi, dalla frazione sequenziale.

3. Illustrate la legge di *Gustafson*, dimostrandola e spiegandone le implicazioni.

   Soluzione La legge di *Gustafson* (1988) detta: $S(n) \leq n - \alpha \times (n - 1)$
   Dove:

   n numero di cores;
   a parte sequenziale;
   b parte parallela, eseguita su un solo core;
   $a + b$ tempo di esecuzione su n core;
   $a + n \times b$ tempo di esecuzione totale serializzato;
   $\alpha = \frac{a}{a+b}$ frazione sequenziale del tempo di esecuzione parallelo.

   Sostanzialmente, *Gustafson* si accorse che incrementando il numero di processori si potessero risolvere problemi di grandezza maggiore, nello stesso arco di tempo e che, quindi, aumentando il *workload* con il numero di cores si potesse ottenere maggiore speedup. Nasce la definizione di *scaling*:

   - *strong scaling*: quanto varia il tempo nell'analisi di un campione di dati, all'aumentare del numero dei cores;
   - *weak scaling*: se varia il tempo nell'analisi di un campione di dati, all'aumentare del numero di cores.

6.2 Fork/Join

6.2.1 Domanda 1

Il seguente codice Java calcola lo speculare di un array:

```
static int[] reverse(int[] array) {
        int[] answer = new int[array.length];
        fjPool.invoke(new Reverse(answer, array, 0, array.length));
        rerutn answer;
}

class Reverse extends RecursiveAction {
        int[] in, out;
        int lo, hi;

        Reverse(int[] o, int[] i, int l, int h) {
                out = o; in = i; lo = l; hi = h;
        }

        public void compute() {
                if (hi == lo+1) {
                        out[in.length - lo - 1] = in[lo];
                } else {
                        Reverse left  =
                                new Reverse(out,in,lo,(hi+lo)/2);
                        Reverse right =
                                new Reverse(out,in,(hi+lo)/2,hi);
                        left.fork();
                        right.fork();
                        right.join();
                }
        }
}
```

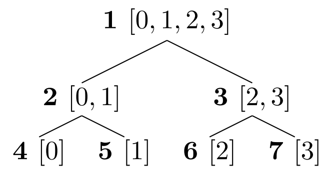
- Il codice contiene un errore: quale?

  Soluzione Per un migliore utilizzo dei thread, occorre lanciare in *fork()* il primo, ottenere il risultato con *compute()* dal secondo e, infine, ottenere il risultato del primo con un *join()*:

```
public void compute() {
        if (hi == lo+1) {
                out[in.length - lo - 1] = in[lo];
        } else {
                Reverse left  =
                        new Reverse(out,in,lo,(hi+lo)/2);
                Reverse right =
                        new Reverse(out,in,(hi+lo)/2,hi);
                left.fork();
                right.compute();
                left.join();
        }
}
```

- Disegnate il DAG di esecuzione per $n = 4$, etichettando i nodi del DAG in modo da individuare il thread associato a ciascun nodo e la porzione di array su cui il thread opera.

Soluzione



- Discutete *work* e *span* dell'algoritmo in funzione del numero n di elementi dell'array.

Soluzione L'algoritmo ha *work* pari a $O(n)$ (numero nodi albero) e *span* pari a $O(2 \times \log n) = O(\log n)$ (altezza albero).

- Spiegate in modo sintetico come ottimizzare il codice.

Soluzione Come spiegato nel primo punto, è poco ottimizzato lanciare come da consegna i due thread ricorsivi: è suggerito invece lanciare in background il primo dei due thread e nel frattempo attendere (dato che tra l'altro i due thread hanno orientativamente lo stesso tempo di esecuzione) attivamente con il *compute()* la conclusione del secondo, per poi riagganciarsi al primo con un *join()* per poterne prelevare il risultato.

- Se $n = 2^{30}$, quanti thread fork-join sono creati dal codice (opportunamento corretto come descritto al punto *a*)? Applicando un cut-off sequenziale di 1024, quanti thread fork-join sarebbero creati?

Soluzione 2^{31}
Nel caso di cut-off sequenziale di 1024: $2^{31} - 1024 = 2^{31} - 2^{10} = 2^{21}$

- Supponete ora di voler memorizzare l'array speculare *in loco*, ovvero di non avere un array di output ma di voler modificare direttamente l'array di input. Come modifichereste il codice in tale scenario?

Soluzione In questo scenario itererei parallelamente sulla prima metà degli elementi dell'array, sostituendoli quindi con gli speculari, nella seconda metà non presa in considerazione nell'iterazione.

6.2.2 Domanda 2

Descrivete l'algoritmo parallelo per la fusione di due array ordinati (*merge*). Illustrate sinteticamente un esempio di esecuzione e analizzate *work* e *span*, assumendo che i due array in input contengano n elementi ciascuno (è sufficiente spiegare come impostare le equazioni di ricorrenza, senza svolgerle esplicitamente).

Soluzione Il *MergeSort* generico ha *equazione di ricorrenza*: $R(n) = O(n) + 2 \times R(\frac{n}{2})$, che per il *Teorema master* è pari a $O(n \times \log n)$. Le due soluzioni per applicare la parallelizzazione sono le seguenti:

1. Utilizzare la parallelizzazione nelle chiamate ricorsive, portando ad un miglioramento tale: $R(n) = O(n) + 2 \times R(\frac{n}{2}) \Rightarrow O(n) + R(\frac{n}{2})$

2. Ottimizzare l'operazione di *merge*: quello che accade è che l'array viene splittato in due sottoarray minori o di pari dimensioni o di una differenza di dimensione di 1. In quest'ultimo caso, si sceglie un indice i dall'array più grande perchè referenzi l'elemento di mezzo. Nell'array minore, viene scelto un indice j tale che, applicata la *ricerca binaria* (di costo $O(\log n)$), alla sua sinistra abbia tutti elementi minori di quello a cui punta l'indice i nell'array maggiore. A questo punto vengono lanciate due chiamate ricorsive - parallelamente -, prendendo, nel primo caso, gli array di sinistra, nell'array minore e maggiore, e nel secondo caso quelli di destra.
   Poichè l'array maggiore sarà diviso sempre in subarray di dimensione $n/4$ e poichè, nel caso limite, nell'array maggiore, ci troveremo in una situazione tale che il subarray di sinistra avrà dimensione 0 e quello di destra $3/4 \times n$, allora: $R(n) = O(\log n) + R(\frac{n}{4}) + R(\frac{3 \times n}{4}) = O(n)$.
   Applicando poi la parallelizzazione anche su queste chiamate ricorsive: $R(n) = O(\log n) + R(\frac{3 \times n}{4}) = O(\log^2 n)$
   Quindi:

   - *work*: $O(n) + 2 \times R(\frac{n}{2}) = n \times \log n$
   - *span*: $O(\log^2 n) + R(\frac{n}{2}) = O(\log^3 n)$
   - *parallelismo*: $\frac{n}{\log^2 n}$

6.2.3 Domanda 3

Immaginate di avere una classe *Worker1* contenente un metodo pubblico *doTask1()*, una classe *Worker2* contenente un metodo pubblico *doTask2()*, ... Supponete inoltre che la vostra applicazione crei quattro thread $T_1, \ldots, T_4$ i cui metodi *run()* contengono il seguente codice:

T_1 *worker1.doTask1()*; *worker2.doTask2()*;

T_2 *worker5.doTask5()*;

T_3 *worker3.doTask3()*;

T_4 *worker4.doTask4()*;

dove le esecuzioni di *doTask1()*, *doTask3()* e *doTask4()* richiedono 20 unità di tempo ciascuna, *doTask2()* ne richiede 30 e *doTask5()* 40. Infine, assumente che il thread *main* esegua il seguente frammento di codice:

1. $T_1.start()$
2. $T_2.start()$
3. $T_2.join()$
4. $T_3.start()$
5. $T_1.join()$
6. $T_3.join()$
7. $T_4.start()$

Rispondete alle seguenti domande, motivando le risposte:

1. Qual è il minimo tempo per l'esecuzione di questo programma? Mostrate un *interleaving* delle esecuzioni dei metodi *doTask1()*, ..., *doTask5()* che permetta di ottenere il minimo tempo di esecuzione.

   Soluzione 80 unità di tempo.

2. Il minimo tempo per l'esecuzione sarebbe diverso se le righe 5 e 6 nel *main* fossero invertite? Perchè?

   Soluzione No, non cambia, perchè è in ogni caso compensato dall'attesa più lunga del *task3*, a seguito del quale il *task1* avrebbe già ultimato la sua esecuzione.

3. Avendo un numero sufficiente di processori, potrebbe accadere che *doTask4()* sia eseguito in parallelo a *doTask1()*? Perchè?

Soluzione No, perchè l'esecuzione del *task4* è condizionata dal precedente *join()* sul *task1*.

4. Avendo un numero sufficiente di processori, potrebbe accadere che *doTask3()* sia eseguito in parallelo a *doTask1()*? Perchè?

   Soluzione Sì, per la ragione inversa della risposta precedente, ma solamente se il *task1* venga eseguito dopo il *task2*, nel *thread1*.

6.2.4 Domanda 4

Descrivete il problema del *packing*, presentate un algoritmo di *packing* parallelo e analizzatene le prestazioni. Discutete inoltre l'applicazione di tale algoritmo al *quicksort*, illustrando l'implementazione di *partition* e analizzando *work*, *span* e *parallelismo* ottenuto nel caso migliore.

Soluzione

- **packing**: pattern per cui venga applicata una condizione di verifica su ciascun elemento di un array in input, restituendo in output un array con i soli elementi per cui tale condizione è verificata.
  Segue tre passaggi: il primo è la generazione di un array bit-vector che inserisca 1 se la condizione è verificata, 0 altrimenti. Il secondo è la generazione di un array prefix-sum del bit-vector. L'ultimo è la generazione di una mappa, a partire dall'array di input, applicando il filtraggio dell'array generato in seconda fase.
  Ha *work* $O(n)$ e *span* $O(\log n)$.

- **QuickSort**: ha *equazione di ricorrenza* pari a: $R(n) = O(n) + 2 \times R(\frac{n}{2})$
  Per il *Teorema master*: $O(n \times \log n)$
  Il *quicksort* segue tre fasi: scelta dell'elemento *pivot* (che assumeremo sia sempre l'elemento mediano), partizionamento e ordinamento.
  Per applicare una parallelizzazione seguiremo due soluzioni:

  1. rendere le chiamate sui due array ricorsive, ottenendo un miglioramento tale: *work* inalterato, *span* pari a $O(n) + 2 \times R(\frac{n}{2}) \Rightarrow O(n) + R(\frac{n}{2})$
  2. ottimizzazione dell'operazione di partizionamento, utilizzando il *packing*. Viene lanciato un primo *packing* la cui condizione è che gli elementi siano $< pivot$, e poi un secondo per cui la condizione sia che gli elementi siano $> pivot$.
     Questo ha *work* $O(n)$ (perchè il *work* nei *packing* è tale) e *span* $O(n) + R(\frac{n}{2}) \Rightarrow O(\log n) + R(\frac{n}{2})$

  Con queste modifiche arriviamo a:

  – *work*: $O(n)$
  – *span*: $O(\log^2 n)$

6.3 Concorrenza

6.3.1 Domanda 1

Considerate il seguente programma Java, il cui scopo è di mantenere il numero di posti liberi in un garage, regolando le entrate di nuove auto:

```
class ParkingGarage {
        private int places;

        public ParkingGarage(int places) {
                if (places < 0)
                        places = 0;
                this.places = places;
        }

        // enter parking garage
        public synchronized void enter() {
                while (places == 0);
                places--;
        }

        // leave parking garage
        public synchronized void leave() {
                places++;
        }
}
```

Durante l'esecuzione il numero *places* di posti liberi deve essere sempre ≥ 0. Rispondete alle seguenti domande, motivando le risposte:

1. Il programma soffre di *race condition*? Se sì, di che tipo?

   Soluzione Sì, soffre di *deadlock*.

2. Mostrate un *interleaving* delle istruzioni che porta il programma a non terminare la sua esecuzione.

   Soluzione Se un cliente richiede il garace con *places* settato a 0, allora necessariamente rimarrà bloccato sul ciclo *while* del metodo *enter()* senza essere in grado di uscirne.

3. Definireste la non-terminazione descritta al punto 2 come un *deadlock*? Perchè?

   Soluzione Perchè rimane in attesa di una risorsa, attesa per la quale nè il richiedente - il cliente - nè l'offerente - il garage - è in grado di terminare.

4. Per ovviare al problema descritto al punto 2, un programmatore modifica il codice come segue (il costrutture e il metodo *leave()* sono inalterati):

```
private synchronized boolean isFull() { return places == 0; }
private synchronized void reducePlaces() { places--; }

public void enter() {
        while (isFull());
        reducePlaces();
}
```

Il nuovo programma soffre di *race condition*? Il nuovo programma termina sempre correttamente la sua esecuzione?

Soluzione Si può presentare un *data race*, perchè nel momento in cui un client riesce ad ottenere il lock *isFull()* e ottiene l'ok, potrebbe venir bloccato dallo scheduler, e lo stesso potrebbe sbloccare all'esecuzione un altro thread che ha fa riesce correttamente a fare il *reducePlace()*. Il primo thread si ritroverà in uno stato incosistente.

5. Date un'implementazione corretta della classe **ParkingGarage**, preferibilmente senza attesa passiva (*spin waiting*).

```
class ParkingGarage {
        private int places;

        public ParkingGarage(int places) {
                if (places < 0)
                        places = 0;
                this.places = places;
        }

        // enter parking garage
        public synchronized void enter() {
                while (places == 0) {
                        this.wait();
                }
                places--;
        }

        // leave parking garage
        public synchronized void leave() {
                places++;
                this.notify();
        }
}
```

6.3.2 Domanda 2

Dite se le seguenti affermazioni sono vere o false, motivando le risposte.

1. La keyword *synchronized* garantisce sia la mutua esclusione degli accessi ad un oggetto, sia l'atomicità della sequenza di istruzioni di accesso.

   Soluzione No, garantisce solo la prima.

   Soluzione Falso, non si eviterebbero *data race* (un thread richiede in *get* una risorsa, mentre questa sta venendo modificata tramite *set*: non avendo il *get* alcun lock, otterrà una risposta incosistente).

2. Per avere un *deadlock* un programma deve accedere in mutua esclusione ad almeno due oggetti diversi.

   Soluzione Falso, anche ad uno solo (vedi esercizio precedente).

3. Un programma che usa un meccanismo di locking consistente può avere *data race*.

   Soluzione Falso.

4. Un programma che usa un meccanismo di locking consistente può avere *bad interleaving*.

   Soluzione Vero.

6.3.3 Domanda 3

Considerate il seguente programma Java:

```
public class Application extends Thread {
        public static X x;
        public static Y y;
        public static Z z;
        public void run() {
                z = new Z(); x = new X(z); y = new Y(z);
                System.out.print("C");
                execute1();
                z.h();
                execute2();
        }
        public void execute1() {
                System.out.print("A");
                x.start();
        }
        public void execute2() {
                y.start();
                System.out.print("L");
        }
        public static void main(String[] args) {
                new Application().start();
        }
}
class X extends Thread {
        public Z z;
        public X(Z zz) { z = zz; z.n = 0; }
        public void run() {
                synchronized(z) {
                        z.n = 1;
                        z.notify():
                        System.out.print("K");
                }
        }
}
class Y extends Thread {
        public Z z;
        public Y(Z zz) { z = zz; }
        public void run() {
                System.out.print("J");
                synchronized(z) {
                        while(z.n == 0) {
                                try {
                                        z.wait();
                                } catch (InterruptedException e) {}
                        }
                        System.out.print("Q");
                }
        }
}
class Z {
        public int n;
        public void h() { System.out.print("P"); }
}
```

Rispondete alle seguenti domande, motivando le risposte:

1. Illustrate almeno tre possibili output per il programma e i relativi *interleaving* delle istruzioni *Java* che li producono.

2. Dite se le seguenti affermazioni sono vere o false:

   (a) *"A"* è sempre stampata per seconda.

   (b) *"K"* è sempre stampata dopo *"P"*.

   (c) *"J"* è sempre stampata dopo *"P"*.

   (d) *"Q"* può essere stampata prima di *"K"*.

3. Supponete che il metodo *h()* nella classe *Z* sia modificato come segue:

```
public void h() {
        n = 100;
        System.out.print("P");
}
```

   Il programma così modificato soffre di *race condition*? Il programma termina sempre la sua esecuzione producendo una stampa?

4. Supponete che il metodo *h()* nella classe *Z* sia modificato come segue:

```
synchronized void h() {
        n = 0;
        System.out.print("P");
}
```

   Il programma così modificato soffre di *race condition*? Il programma termina sempre la sua esecuzione producendo una stampa?

6.3.4 Domanda 4

Spiegate la differenza tra strategie di locking *coarse-grained* e *fine-grained* usando come esempio l'implementazione di liste concorrenti.

Soluzione

- *coarse-grained*: utilizzo di meno *lock* per più oggetti;
- *fine-grained*: utilizzo di più *lock* per più oggetti.

L'implementazione nel caso delle liste concorrenti è dettata da alcune regole strutturali di questo tipo di liste:

- avendo una lista linkata, ogni nodo referenzia il successivo;
- per ogni operazione, occorre iterare sull'intera lista sequenzialmente.

Quindi, le implementazioni:

1. *coarse-grained*: per ogni operazione - sia di *read* che *write* - bisogna acquisire il *lock* sull'intera lista. Questa implementazione porta ad un *bottleneck* eccessivo.

2. *fine-grained*: non è possibile inserire un lock per nodo, proprio per una caratteristica di struttura della lista, perchè si avrebbe incosistenza causata dal fatto che ogni nodo referenzia anche il successivo. Perciò è utile, per qualsiasi modifica, acquisire il lock sul nodo e sul suo predecessore.
   Anche questa soluzione potrebbe rallentare inutilmente perchè, dovendosi scorrere ogni volta l'intera lista, per ogni lock incontrato, bisognerebbe attendere di avere la possibilità di acquisirlo, anche se non direttamente necessario alla modifica desiderata.

3. *reader/writer*: come nel caso del *fine-grained*, ma si ha un sistema di priorità. Il client che itera per una modifica, lo fa da reader, quando arriva al nodo interessato (col suo predecessore), fa un *upgrade* e acquisisce i lock da writer. In ogni caso, si ripresenta il problema che i writer sono bloccanti in ogni caso.

6.4 OpenCL

6.4.1 Domanda 1

Scrivete un kernel *OpenCL* che riceve un array A di interi di dimensione n e calcola un array B di *float* di dimensione $\lfloor n/2 \rfloor$. B è tale che, per ogni posizione dispari p in A, $B[p/2]$ contiene la media degli alementi $A[p-1]$, $A[p]$ e $A[p+1]$, se esistono. Discutete qual è lo spazio di indicizzazione e il numero di *work-item* eseguiti.

Soluzione codice:

```
__kernel void half(__global const int* A,
                   __global float* B,
                   __global const int len) {

        const int pos = get_local_id(0);

        if (pos % 2 == 1) {
                int sum = 0;
                int cnt = 1;
                sum += A[pos];

                if (pos > 0) {
                        sum += A[pos-1];
                        cnt += 1;
                }

                if (pos+1 <= len) {
                        sum += A[pos+1];
                        cnt += 1;
                }

                B[pos/2] = sum/cnt;
        }
}
```

Lo spazio di indicizzazione è mono-dimensionale e il numero di *work-item* eseguiti è pari alla lunghezza *len* dell'array A.
Occorre passare come *global_size* la lunghezza dell'array A.

6.4.2 Domanda 2

Si vogliono trovare tutte le occorrenze di una stringa patterna all'interno di una stringa di ricerca. Si scriva il codice di un kernel *OpenCL* per risolvere tale problema, assumendo che il kernel abbia la seguente intestazione:

```
void __kernel PatternMatcher(__global char* pattern_string,
                               const unsigned long pattern_length,
                             __global char* buffer_string,
                               const unsigned long buffer_length,
                             __global unsigned int* results)
```

L'array *results* calcolato in output ha lunghezza *buffer_length* ed è tale che $results[i] = 1$ se le posizioni da i a $i + pattern\_length - 1$ dell'array *buffer_string* contengono i caratteri della stringa *patter_string*, 0 altrimenti.

Soluzione codice:

```
__kernel void PatternMatcher(__global char* pattern_string,
                               const unsigned long pattern_length,
                             __global char* buffer_string,
                               const unsigned long buffer_length,
                             __global unsigned int* results) {

        int i = get_global_id(0);

        if (buffer_string[i] == pattern_string[0]) {
                for (int j = 1; j < pattern_length; j++) {
                        if (i+j > buffer_length ||
                        buffer_string[i+j] != pattern_string[j]) {
                                buffer_string[i] = 0;
                                return;
                        }
                }
                buffer_string[i] = 1;
        }
}
```