UNIVERSITÀ DEGLI STUDI DI MILANO
FACOLTÀ DI SCIENZE E TECNOLOGIE

CORSO DI LAUREA TRIENNALE IN INFORMATICA

PARALLELISMO E HASKELL

Relatore: Prof. Carlo Mereghetti
Correlatore: Prof. Beatrice Palano
Correlatore: Prof. Alberto Momigliano

Tesi di Laurea di:
Jacopo Francesco Zemella
Matr. Nr. 813518

ANNO ACCADEMICO 2015-2016

Prefazione

La prima domanda da porsi per comprendere al meglio le ragioni dietro il lavoro in oggetto è la seguente: come è possibile ottimizzare al meglio le risorse della macchina? La seconda è conseguentemente: esclusa la multiprogrammazione, limitata all'esecuzione concorrente dei processi, che alternative ho a disposizione, se il problema riguarda un programma solo?
Come da titolo una possibile alternativa è rappresentata dal parallelismo, una vera e propria "parallelizzazione" del lavoro mediante l'ausilio di "lavoratori" (i processori) che eseguono ciascuno una parte del lavoro (i threads) concorrentemente.
Naturalmente il passaggio dal sequenziale (ovvero programmazione priva di parallelismo) al parallelo non è decisamente indolore. Oltre a una pianificazione decisamente più complessa del programma da eseguire, subentrano numerose problematiche legate alla coordinazione dei processori o alla dimensione del problema.
Va dunque detto questa tesi non ha come finalità il rendere obsoleto il concetto di programmazione sequenziale: al contrario verranno mostrati casi in cui la parallelizzazione non solo è sconveniente, ma addirittura meno performante della sua controparte.
Lo scopo ultimo del lavoro è piuttosto mostrare in primo luogo come molti problemi, spesso anche relativamente semplici nella loro comprensione e implementazione sequenziali, risultino lenti e poco efficienti (soprattuto dal punto di vista di tempo di esecuzione) al crescere della loro dimensione. In questi casi il parallelismo trova un suo perché, arrivando a migliorare l'esecuzione ottimizzando le risorse della macchina.
Dimostrato questo concetto, subentra il problema di come implementare queste caratteristiche in un programma. Per questa seconda parte ci serviremo di un linguaggio di programmazione di comodo utilizzo, Haskell, un linguaggio funzionale avente a disposizione astrazioni e librerie che consentono di parallelizzare problemi generici già a livello di codice, a vantaggio del programmatore.

Indice

Parte I

Calcolo Sequenziale

Capitolo 1

Algoritmi sequenziali

Prima di avviarci nella teoria del calcolo parallelo è doverosa un'introduzione al calcolo sequenziale e, nello specifico, al concetto di algoritmo.
La definizione informale di un algoritmo è un concetto generalmente noto: un algoritmo è una sequenza finita di operazioni elementari (univoche e non ambigue) che, date in esecuzione a un agente, manipola diversi valori in input per per ottenerne degli altri in output. In altre parole un algoritmo definisce implicitamente una funzione da un dominio di definizione (input) a un codominio specifico (output) e tale che per ogni input appartenente al dominio esista sempre un output corrispondente. Detto ciò, se definiamo un algoritmo A, chiameremo $\mathrm{f}_A(x)$ la funzione che associa ad ogni x del dominio la corrispondente uscita $\mathrm{f}_A(x)$. In questa definizione è racchiuso implicitamente il problema risolto dall'algoritmo.
Formalmente, dato un problema $\mathrm{f} : I \to S$, in cui I rappresenta l'insieme delle varie istanze e S l'insieme delle soluzioni, possiamo affermare che un algoritmo A risolve tale problema se $\mathrm{f}_A(x) = \mathrm{f}(x)$ per ogni istanza x.
È bene precisare che per ogni problema f possono esistere numerosi algoritmi che lo risolvono in maniera differente, e poiché l'utilizzo di un algoritmo comporta sempre l'utilizzo di un certo quantitativo di risorse (tempo di esecuzione, memoria, ecc.), il saper scegliere o costruire un algoritmo che le sappia gestire in maniera adeguata non è un particolare di poco conto. Un cattivo utilizzo delle risorse, nel peggiore dei casi, può rappresentare un vero e proprio ostacolo a livello di esecuzione, e per questo motivo è importante trovare un modo per valutare la bontà di un algoritmo. Un metodo largamente utilizzato per ottenere questa valutazione è quello di adottare un modello di calcolo preciso e di tradurre l'algoritmo in modo da poter essere interpretato dallo stesso.
Poiché la nostra analisi è orientata principalmente allo studio della complessità sequenziale, ci concentreremo su un comodo modello di calcolo,

specializzato in questo campo: la macchina di Turing.

1.1 Macchina di Turing Deterministica

La macchina di Turing (MdT) è un modello ideale di calcolatore dalle meccaniche intuitive e semplici, solitamente usato più per l'analisi computazionale che per l'implementazione di calcolatori reali. Vista la sua semplicità, rappresenta uno dei modelli più utilizzati per identificare e studiare il calcolo sequenziale.
Informalmente una MdT è composta da un insieme di stati interni, in cui si può trovare la macchina (si definisce anche uno "stato iniziale", in cui si trova all'inizio dell'esecuzione, e un insieme di stati finali), un nastro di lunghezza infinita suddiviso in celle e una testina posizionata su una di esse, capace di leggere, scrivere o cancellare caratteri sul nastro. La macchina analizza il nastro una cella alla volta e in base al carattere letto e allo stato interno corrente esegue una "mossa", così composta:

- la macchina cambia il proprio stato interno;
- la macchina esegue un'operazione di scrittura o muove la testina di una cella a destra o sinistra.

Se la sequenza di mosse eseguite è finita diciamo che la macchina si arresta sull'input considerato e diciamo che tale input è accettato se lo stato raggiunto nell'ultima configurazione è finale.
Formalmente, invece, possiamo vedere la MdT come una sestupla:

$$\mathrm{M} = < \mathrm{Q}, \sum, \mathrm{q}_0, \mathrm{B}, \delta, \mathrm{F} >$$

In particolare:

- Q è l'insieme degli stati interni;
- $\sum$ è l'alfabeto con cui vengono espressi i dati sul nastro;
- $\mathrm{q}_0 \in Q$ è lo stato iniziale;
- $B \in \sum$ è un simbolo particolare, detto simbolo vuoto o blank;
- δ è la funzione di transizione definita come δ: $Q \times \sum \rightarrow Q \times \sum \times$ {-1,1};
- $F \subseteq Q$ è l'insieme di stati finali.

Per ogni $q \in Q$ e ogni $a \in \sum$, la funzione $\delta(q, a)$ definisce una tripla (p, b, l), dove p rappresenta il nuovo stato, b il carattere scritto nella cella corrente e l il movimento che esegue la testina, rispettivamente a destra se l = -1, a sinistra se l = +1.
Una configurazione particolare della macchina M è composta dallo stato della macchina, dal contenuto del nastro e dalla posizione della testina. Il tutto è esprimibile come una stringa: αqβ, con $\alpha \in \sum^*$, $q \in Q$, $\beta \in \sum^+$. In questo caso α rappresenta la stringa a sinistra della testina, q lo stato corrente di M, mentre β una stringa collocata a destra della testina, seguita da infiniti caratteri blank. Notiamo che allo stato iniziale la configurazione diventa la stringa $q_0\beta$. In questo contesto definiamo un'altra operazione binaria sull'insieme delle configurazioni C, l'operazione $\vdash_M$, tale che per ogni $C1, C2 \in C$, vale che $C1 \vdash_M C2$ se e solo se C1 raggiunge C2 in una mossa. Più precisamente, data la configurazione αq$\beta \in C$ allo scatto di una transizione $\delta(q, b) = (p, b, l)$ (supponiamo che $\beta = b\beta', \beta' \in \sum^*, b \in \sum$) distinguiamo due casi:

- se l = +1 allora:

$$\alpha qb\beta \vdash_M \begin{cases} \alpha cp\beta' & : \beta' \neq \epsilon \\ \alpha cpB & : \beta' = \epsilon \end{cases}$$

- se l = -1 e $\alpha \neq \epsilon$ allora, posto $\alpha = \alpha' a$, $\alpha' \in \sum^*, a \in \sum$:

$$\alpha qb\beta \vdash_M \begin{cases} \alpha' pa & : c = B, \beta' = \epsilon \\ \alpha pac\beta' & : altrimenti \end{cases}$$

Osserviamo che se $\delta(q, b)$ non è definito , oppure $l = -1$ e $\alpha = \epsilon$, allora non esiste una configurazione $C2 \in C$ tale che $\alpha q\beta \vdash_M C$. In questo caso diciamo che q e una configurazione di arresto per M. Senza perdita di generalità possiamo supporre che ogni configurazione accettante sia una configurazione di arresto.
Un insieme finito C_i con *i = 1,..,m*, di configurazioni di M è una computazione di M su input $w \in \sum$, se $C_0 = C_0(w)$, $C_{i-1} \vdash_M C_i \forall i = 1, 2, .., m$ e C_m è di arresto. Se C_m è anche accettante, diciamo che M accetta l'input w. Viceversa, se M non termina, possiamo dire che l'input w genera una computazione definita da infinite configurazioni.
Inoltre, se M si arresta su ogni input $x \in \sum^*$, diciamo che M risolve il problema di decisione $< \sum^*, q >$ dove, $\forall x \in \sum^*$:

$$q(x) = \begin{cases} 1 & se\ M\ accetta\ x \\ 0 & altrimenti \end{cases}$$

La MdT ha avuto un ruolo fondamentale nell'informatica teorica: poiché è matematicamente dimostrato che per qualsiasi modello di calcolo ragionevole esiste una macchina di Turing associata (tesi di Church-Turing), questo ha reso possibile definire uno standard nell'analisi computazionale degli algoritmi e una loro classificazione dal punto di vista risolutivo.

1.2 Complessità sequenziale

Quando analizziamo un algoritmo dobbiamo tenere in considerazione due caratteristiche: il fatto che risolva correttamente il problema che gli viene chiesto di risolvere (correttezza) e quante risorse impiega per essere eseguito (complessità). Le risorse utilizzate da un algoritmo sono essenzialmente il tempo di calcolo e la memoria utilizzata, che possiamo esprimere come funzioni a valori interi positivi. Nello specifico un algoritmo A su input x viene rappresentata da una funzione $T_A(x)$ per il tempo e da $M_A(x)$ per lo spazio. Cercare di dare una definizione formale di queste misure può risultare problematico, soprattutto perché la variabile x può assumere tutti i valori in input. Per questo motivo si cerca di raggruppare le varie istanze del problema a seconda della loro dimensione, definendo una funzione che associa a ogni ingresso un numero naturale che rappresenta la quantità di informazione contenuta. Per fare un esempio, la dimensione di un numero naturale n è $\lfloor \log(n) \rfloor + 1$, ovvero la sua lunghezza in codice binario.
Tuttavia classificare le varie istanze mediante la loro dimensione non è sufficiente: è possibile che a input diversi $x \neq y$, ma aventi la stessa dimensione ($|x| = |y|$), si possa ottenere $T_A(x) \neq T_A(x')$. Questo fatto è ancora più evidente nel caso pratico: l'esecuzione di un programma su un calcolatore dipende sempre da numerosi fattori come la macchina usata, il linguaggio, il compilatore, ecc.
Per ovviare a questo problema si definiscono le funzioni in modo da ottenere una stima **assoluta** di quella che è la complessità di un algoritmo. Ad esempio tra le varie stime ottenibili per il tempo $T_A(x)$ vengono solitamente considerati i seguenti casi:

- caso peggiore: $T_A^w : \mathbb{N} \to \mathbb{N}$, $T_A^w(n) = \max(T_A(x)\ t.c.\ |x| = n)$;
- caso migliore: $T_A^b : \mathbb{N} \to \mathbb{N}$, $T_A^b(n) = \min(T_A(x)\ t.c.\ |x| = n)$;
- caso medio: $T_A^a : \mathbb{N} \to \mathbb{R}$ con I_n = numero di istanze $x \in I$ di dimensione n, $T_A^a(n) = \frac{\sum_{|x|=n} T_A(x)}{I_n}$;

Prediligere una stima piuttosto che un'altra è una decisione puramente legata all'utilizzo dell'algoritmo. Dal punto di vista analitico, però, si tende a prediligere l'andamento nel caso peggiore, ovvero nel caso in cui l'input richiede il maggior numero di iterazioni per terminare, o nel caso medio, che analizza la computazione da un punto di vista statistico.
Presentiamo un esempio classico per capire come ottenere il tempo di calcolo: l'ordinamento di un array. Vogliamo ordinare un vettore $[x_1, .., x_n]$ di n numeri in ordine crescente nel seguente modo:

Per i = 1,..,n esegui iterativamente le seguenti operazioni
- seleziona il minimo tra i valori $[x_i, .., x_n]$
- scambialo con x_i

In questo caso l'algoritmo esegue $n + (n-1) + .. + 1 = \frac{n(n-1)}{2}$ passi per ottenere un vettore ordinato.
Nel modello della MdT, data una macchina M = <Q,$\sum$,q_0,B,δ,F> e un input definito da una stringa $w \in \sum^*$, definiamo con $T_M(w)$ come il massimo numero di mosse da effettuare per terminare l'esecuzione.
Ovviamente $T_M(w) = +\infty \leftrightarrow$ M non termina su w.
Quindi possiamo definire una funzione $T_M(n)$ di stima del tempo di calcolo nel seguente modo:

$$T_M(n) = \max(T_M(w) \ t.c. \ w \in \sum^*, |w| = n)$$

Inoltre, data f una funzione a valori reali positivi, diciamo che M esegue una computazione in tempo $f(n)$ se $T_M(n) \leq f(n) \ \forall n \in \mathbb{N}$.

1.2.1 Analisi Asintotica della Complessità

Abbiamo illustrato come avviene lo studio dei vari casi di computazione di un algoritmo. Tuttavia, per confrontare tra di loro algoritmi che risolvono lo stesso problema, così da scegliere il migliore disponibile, si ricorre a criteri di valutazione più specifici. Quello più utilizzato è il calcolo computazionale asintotico (solitamente riferito al caso medio o peggiore di un algoritmo).
Sostanzialmente si tratta di valutare la complessità di un algoritmo su input avente un entrata di dimensioni molto grandi, ovvero ponendo $n \to +\infty$.
Ovviamente questo meccanismo vacilla se utilizziamo l'algoritmo per problemi relativamente piccoli, ma rilevare anche una piccola differenza nell'ordine di grandezza della complessità di due procedure può determinare enormi differenze in termini di prestazioni.
Proviamo, per esempio, a confrontare diversi algoritmi che risolvono lo stesso problema aventi complessità asintoticamente pari a: $n, \ n\log(n), \ n^2, \ 2^n$.

Supponendo che ogni operazione venga eseguita in $1\mu s$ l'esecuzione richiederà un tempo di esecuzione pari a[1]

	$n = 10$	$n = 100$	$n = 1000$	$n = 10000$	$n = 100000$
n	$10\mu s$	$0.1ms$	$1ms$	$10s$	$0.1s$
$n\log(n)$	$23\mu s$	$460.5\mu s$	$6.9ms$	$92.1ms$	$1.15s$
n^2	$0.1ms$	$10ms$	$1s$	$100s$	$2.7h$
2^n	$1ms$	$10^{14}c$	∞	∞	∞

Notiamo che per problemi avente complessità lineare (n *o* $n\log(n)$) la computazione avvenga in tempo relativamente buono per problemi anche di grandi dimensioni. Al contrario algoritmi aventi complessità n^k *con* $k \geq 2$ risultano convenienti solo per problemi contenuti, mentre andando su complessità esponenziali il problema esplode anche sulle piccole dimensioni.

1.2.2 Efficienza degli Algoritmi

Mediante valutazioni precedenti si possono classificare la gran parte degli algoritmi realizzabili, distinguendoli mediante la loro complessità asintotica. Introduciamo così il concetto di **efficienza**: un algoritmo si dice efficiente quando la sua complessità è di **ordine polinomiale**, ovvero che vale $\mathcal{O}(n^k)$ con $k \geq 1$; un algoritmo si dice invece inefficiente quando la sua complessità è di **ordine superpolinomiale** (per esempio, $\mathcal{O}(n!)$ o $\mathcal{O}(k^n)$ con $k \geq 2$).
Va puntualizzato che non sempre è possibile creare un algoritmo efficiente (soprattutto nella MdT che abbiamo descritto). L'insieme dei problemi che è possibile risolvere a complessità polinomiale è detta classe P, nelle MdT deterministiche, e NP, nelle MdT non deterministiche. Non tratteremo nel dettaglio questi concetti, soprattutto perché sono ancora oggetto di dibattito[2], ma è bene specificare che esistono strumenti per stabilire se un problema è risolvibile o no in modo efficiente.
Un ultimo concetto da tener presente nella complessità algoritmi è il principio di **ottimalità**: dato un problema risolto con complessità peri a $f(n)$, diciamo che l'algoritmo che lo risolve è **ottimo** se qualunque altro algoritmo che lo risolve ha complessità $\mathcal{O}(f(n))$.

[1] Nella tabella s= secondi, h = ore, c = secoli.

[2] Il problema P = NP? legato ai problemi NP-completi è uno dei problemi irrisolti più famosi della matematica odierna.

Capitolo 2

Esempi

Di seguito verranno presentati alcuni esempi di algoritmi sequenziali, molti dei quali verranno implementati a livello di codice alla fine dell'elaborato.

2.1 Ordinamento

Quello dell'ordinamento di un insieme di oggetti confrontabili è uno degli problemi tipici del calcolo algoritmico. Ne esistono di diversi tipi, il che lo rende un esempio perfetto per confrontare le varie alternative.
Prendiamo ad esempio due algoritmi piuttosto comuni in questo campo: quicksort[1] e mergesort[2]. In questo caso, ad esempio i vari casi di esecuzione cambiano:

- caso migliore: $\mathcal{O}(n \log n)$ per quicksort e $\mathcal{O}(n \log n)$ per mergesort;
- caso peggiore: $\mathcal{O}(n^2)$ per quicksort e $\mathcal{O}(n \log n)$ per mergesort;
- caso medio: $\mathcal{O}(n \log n)$ per quicksort e $\mathcal{O}(n \log n)$ per mergesort;

Pseudocodice

```
Quicksort(L = [a1,...,an])
    if(n <= 1) then
```

[1] In linguaggio naturale: preso un elemento x in una lista, se ne creano due: quella degli elementi minori di x e quella degli elementi maggiori. Si richiama poi l'algoritmo sulle nuove liste, dopodiché si uniscono i risultati aggiungendo x in mezzo agli stessi.

[2] In linguaggio naturale: è il tipico algoritmo divide-et-impera; una lista avente uno o nessun elemento è ordinata, altrimenti si divide la lista in input in due e si esegue mergesort su entrambe; alla fine si estrae dalle due liste (ordinate) il minore tra i valori in testa, fino ad esaurire tutti gli elementi presenti.

```
        return L
    else
        scegli un elemento "p" in L
    calcola la lista "left" di elementi minori di p
    calcola la lista "right" di elementi maggiori di p
    left = Quicksort(left)
    right = Quicksort(right)
    return left : [p] : right


Mergesort(L = [a1,..,an])
    if(n <= 1) then
        return L
    else
        dividi la lista L a metà, creando le liste "left" e "right"
        left = Mergesort(left)
        right = Mergesort(right)
        return merge(left, right)

merge(left, right)
    if(left = []) return right
    else if(right = []) return left
    else
        h1 = head(left)
        h2 = head(right)
        if (h1 < h2) then
            t = tail(left)
            return [h1] : merge(t, right)
        else
            t = tail(right)
            return [h2] : merge(left, t)
```

2.2 Algebra Lineare

Facciamo degli esempi riguardanti le operazioni tra matrici:

- prodotto tra matrici
- calcolo del determinante

Nel primo caso il prodotto di matrici è un operazione direttamente proporzionale alle dimensioni degli operandi. In caso di prodotto di una matrice

MxN con una MxP sarà necessario un algoritmo di complessità asintotica $\mathcal{O}(N * M * P)$.

Pseudocodice

```
Prodotto(A = matrice MxN, B = matrice NxP)
C = matrice MxP di zeri
    for (i from 1 to M)
        for (j from 1 to P)
            for (k from 1 to N)
                c_ij = c_ij + (a_ik * b_kj)
            end for
        end for
    end for
    return C
```

2.3 Grafi

Parte II

Calcolo Parallelo

Capitolo 3

Gli Algoritmi Paralleli

3.1 Modelli Paralleli

Spesso uno dei problemi principali inerenti al calcolo sequenziale è il tempo di esecuzione: si pensi al miglior algoritmo possibile su un problema semplice come l'ordinamento di un set di oggetti e si immagini di aumentare il numero di elementi in maniera spropositata.
Inoltre, in casi di sistemi in tempo reale (real-time), il tempo di calcolo diventa essenziale, al punto che una mancata esecuzione in tempi utili può portare a un deterioramento del sistema stesso.
Una soluzione a questo problema è rappresentata dalla suddivisione del lavoro in parti da eseguire contemporaneamente da più processori, nel tentativo di ottenere un'ulteriore ottimizzazione dei tempi di calcolo.
I sistemi di questo tipo vengono detti *paralleli*, per distinguerli dalla loro controparte monoprocessore, chiamati *sequenziali*. Analogamente un algoritmo che può essere eseguito su un sistema multiprocessore è detto *algoritmo parallelo*.

3.1.1 La macchina PRAM

Analogamente alla RAM per il calcolo sequenziale, sottoprodotto della macchina di Turing, il modello di calcolo più semplice per rappresentare sistemi multiprocessore è la macchina PRAM.
Si tratta di un modello avente le seguenti caratteristiche:

- un insieme di p processori uguali;
- una memoria globale condivisa tra tutti i processori;

- un'unità di controllo che consente ai processori di accedere alla memoria globale (chiamata MAU, Memory Access Unit).

L'esecuzione dei processori avviene in maniera indipendente: l'elaborazione di un processore è scorrelata dall'attività dei suoi compagni. La comunicazione e lo scambio di dati, invece, avviene mediante la memoria globale, a cui i processori accedono in tempo $\mathcal{O}(1)$ (Random Access).
L'attività di un singolo processore continua a procedere in modo sequenziale: a ogni ciclo di clock un processore può decidere se effettuare operazioni sui dati che possiede o se effettuare operazioni di lettura e scrittura sulla memoria condivisa. Ad un singolo ciclo di clock corrispondo molteplici istruzioni eseguite in contemporanea e in questo modo si ottiene parallelismo. Le macchine che si reggono su questo principio sono chiamate anche SIMD (Single Instruction Multiple Data).
Si può inoltre distinguere i tipi di macchine parallele in base all'accesso dei processori a memoria condivisa:

- EREW (Exclusive Read Exclusive Write): l'accesso contemporaneo a memoria condivisa non è consentito;
- CREW (Concurrent Read Exclusive Write): l'accesso contemporaneo a memoria condivisa è consentito solo in lettura;
- CRCW (Concurrent Read Concurrent Write):: l'accesso contemporaneo a memoria condivisa è consentito in lettura e in scrittura.

Il nostro scopo è confrontare l'esecuzione parallela con quella sequenziale, pertanto verranno considerati solo problemi relativi alle macchine EREW o CREW.

Parallelizzare un problema

Ogni problema parallelo nasce come algoritmo sequenziale: trasformare un algoritmo iterativo in un algoritmo parallelo significa scegliere l'insieme di operazioni da eseguire parallelamente sui vari processori. Non vale il viceversa: la sola presenza di operazioni atomiche vale come condizione sufficiente per asserire che non tutti i problemi sequenziali possano essere parallelizzati.

3.2 Esecuzione negli Algoritmi Paralleli

3.2.1 La legge di Amdahl

L'aumento delle prestazioni di un sistema manipolando le risorse a disposizione è un concetto più vecchio della parallelizzazione. Dato che un algoritmo può essere diviso in componenti di esecuzione è ragionevole pensare che riducendo i tempi di calcolo di di una di quelle sezioni migliorerà in maniera sostanziale anche l'efficienza dell'algoritmo stesso. Tuttavia ci sono delle limitazioni.
Si tratta di un concetto è riassumibile nella **legge di Amdahl**:

"Il miglioramento che si può ottenere su una certa parte del sistema è limitato dalla frazione di tempo in cui tale attività ha luogo"

In parole povere, migliorate le prestazioni di una parte di un algoritmo, esiste un limite di ottimizzazione, legato principalmente al livello di migliorie apportate ed a quanto tempo quella sezione va effettivamente in esecuzione. La formula che descrive questo fenomeno è la seguente:

$$\frac{1}{(1-P)+\frac{P}{S}} \tag{3.1}$$

in cui P è la percentuale di tempo in cui la sezione migliorata va in esecuzione e S è il livello di miglioramento apportato (ad esempio raddoppiando la velocità di esecuzione S equivale a 2).
Notiamo subito che se aumentiamo la velocità all'infinito, ovvero se $S \to \infty$, il miglioramento delle prestazioni è limitato da P, il che ci riporta all'enunciato precedente.
Detto ciò, anche l'aggiunta di processori in parallelo è legato a questa legge, traducendosi in:

$$\frac{1}{(1-P)+\frac{P}{N}} \tag{3.2}$$

in cui N è il numero di processori a disposizione.
Il programma parallelo ideale quindi, non è quello che viene eseguito su molti processori, quanto più quello che riduce il valore *(1 - P)* al minimo possibile.

3.2.2 Tempo di calcolo

Un algoritmo parallelo può essere generalmente definito in questo modo:

```
for (i from 0 to NumProcessori) do
```

```
            operazioneParallela
end
```

in cui `NumProcessori` rappresenta il numero di processori che vogliamo sfruttare, mentre `operazioneParallela` rappresenta una sequenza di istruzioni da eseguire esclusivamente su un singolo processore.
Indichiamo ora il tempo di calcolo legato ad un algoritmo A con $\mathrm{T}_A(n,p)$, valore dipendente cioè sia dalla dimensione dell'input che dal numero di processori. Notiamo che il caso base, in cui $p = 1$ coincide con il valore relativo al caso dell'esecuzione sequenziale, trattandosi di esecuzione monoprocessore. Questo elemento servirà per stimare la bontà dell'esecuzione parallela.
Introduciamo a tal proposito due nuove misurazioni: lo *Speedup* e l'*Efficienza*, che indicheremo rispettivamente con $\mathrm{S}_A(p)$ e $\mathrm{E}_A(n,p)$.
Nel dettaglio:

$$\mathrm{S}_A(p) = \frac{\mathrm{T}_A(n,1)}{\mathrm{T}_A(n,p)} \tag{3.3}$$

$$\mathrm{E}_A(n,p) = \frac{\mathrm{S}_A(p)}{p} = \frac{\mathrm{T}_A(n,1)}{p\mathrm{T}_A(n,p)} \tag{3.4}$$

Lo Speedup in questo caso rappresenta il guadagno in termini di tempo di esecuzione relativa all'utilizzo di p processori. Si tratta comunque di una misura avente un limite matematico: come dimostrato dalla legge di Amdahl l'accelerazione dell'esecuzione è superiormente limitata dalla percentuale di codice eseguito in parallelo, dunque è impossibile ottenere Speedup lineare e direttamente proporzionale a p.
L'efficienza, invece, indica quanto l'algoritmo sfrutta effettivamente il parallelismo. Nel caso ideale un algoritmo parallelo ha efficienza 1, ovvero quando lo speedup ha andamento lineare ($S(p) = p$), nel caso pratico è una funzione monotona decrescente, essendo inversamente proporzionale a p. Ciò significa che aumentando i processori diminuisce l'efficienza dell'algoritmo parallelo, anche in casi in cui lo Speedup continui ad aumentare.
Per questo motivo possiamo definire il seguente enunciato:

$$\forall k \geq 1 \Rightarrow 1 \geq \mathrm{E}_A(n,p) \geq \mathrm{E}_A(n,p/k) \tag{3.5}$$

Ciò che limita lo speedup, e di conseguenza l'efficienza, degli algoritmi paralleli viene solitamente chiamato *Overhead* di esecuzione, ovvero calcoli non necessari alla sola risoluzione del problema. Osserviamo da cosa si compone:

$$\mathrm{O}_A(p) = p\mathrm{T}_A(n,p) - \mathrm{T}_A(n,1) \tag{3.6}$$

Parte III

Programmazione Parallela

Capitolo 4

Il Parallelismo oggi

Cominciamo a discutere della parallelizzazione in senso più concreto partendo dalla **prima legge di Moore**:

"La complessità di un microcircuito, misurata ad esempio tramite il numero di transistori per chip, raddoppia ogni 18 mesi"

Dal punto di vista industriale la creazione di processori più potenti e aventi una frequenza di clock superiore comincia a perdere significato, come dimostrato dal fatto che la potenza dei singoli core abbiano avuto un improvviso blocco verso gli inizi degli anni 2000. Questo fenomeno ha reso evidenti i limiti della legge di Moore sopracitata: di per sè i processori non sono migliorati, eppure la potenza di calcolo delle architetture non ha accennato a rallentare. Come ci spieghiamo tutto ciò?
Di certo uno sviluppo tecnologico c'è stato, si è solo spostato verso un altro campo: i multiprocessori. Raggiunti i limiti fisici legati alla capacità di elaborazione dei core, la strategia migliore (e per certi versi anche la sola disponibile) per aumentare la potenza di calcolo è stata la concentrazione di molteplici processori installati sulla stessa macchina che lavorano in parallelo.
A causa di ciò, il problema della parallelizzazione è concetto sempre più presente nel panorama informatico moderno, che può aprire svariate opportunità di ricerca anche dal punto di vista dei linguaggi di programmazione. L'idea di concedere ad un programmatore gli strumenti giusti per ottimizzare le risorse della macchina già nella compilazione del codice da eseguire è al tempo stesso affascinante e allettante. Non è quindi un caso che numerosi linguaggi si siano attrezzati per fornire strutture adatte a questo scopo.

In generale, quando si tratta di paradigmi di programmazione, distinguiamo due diverse tecniche di programmazione parallela:

- implicita: il sistema riconosce autonomamente i meccanismi da adoperare per dividere il problema in modo da poterne eseguire le parti parallelamente; di fatto il programmatore non effettua nessuna precisazione sulla natura dell'esecuzione e scrive il codice come se fosse un programma sequenziale;

- esplicita: il ruolo del programmatore è quello di partizionare il problema nel modo da lui ritenuto migliore; la stesura del codice è cruciale.

La parallelizzazione esplicita è quella che ci interessa maggiormente: vogliamo trovare un modo efficace per parallelizzare un'esecuzione in maniera diretta, definendo le parti da suddividere all'interno del problema già a livello di codice.

Capitolo 5

Haskell

Tra tutti i linguaggi che consentono una gestione delle risorse interne delle architetture multi processore, Haskell è un ottimo candidato per essere il più fruibile tra tutti quelli disponibili.
Oltre a fornire un'interessante serie di caratteristiche formali, quali la programmazione funzionale, la lazy evaluation, le funzioni higher-order e altre che vedremo in dettaglio, possiede già un ricco quantitativo di librerie e strutture che consentono la creazione di programmi paralleli già nel codice

5.1 Introduzione al linguaggio Haskell

Haskell è un linguaggio di funzionale puro caratterizzato da una tipizzazione forte e statica, creato alla fine degli anni '80, disponibile e integrabile su tutte le principali piattaforme software odierne. Possiede le caratteristiche più note dei linguaggi di programmazione (funzionali e non), tra cui la lazy evaluation, il polimorfismo e le funzioni di ordine superiore, in virtù del lambda calcolo, su cui si regge dal punto di vista matematico. Oltre a questo supporta i tipi di dati più comuni, come gli interi e i valori booleani, e permette di crearne di nuovi sotto forma di strutture dati.
Particolare di non poco conto per il nostro interesse, è già pienamente avviato nel mondo dell'elaborazione parallela e dispone di numerose librerie in open source che supportano programmazione parallela e concorrente.
Descrivere nel dettaglio un intero linguaggio di programmazione rischia di risultare troppo complicato, pertanto ci limiteremo a descriverne i punti salienti per concentrarci sul lato della programmazione parallela; un tutorial su come muovere i primi passi in Haskell è disponibile sul sito `https://www.schoolofhaskell.com/`.

5.1.1 Installazione

Per poter eseguire correttamente Haskell è necessario installare il compilatore associato, GHC, the Glasgow Haskell Compiler, disponibile online su `https://www.haskell.org/ghc/` per tutte piattaforme Linux, Windows e Mac OS.
Oltre a questo è disponibile anche il tool Cabal per l'installazione di package aggiuntivi, scaricabile dal sito `https://www.haskell.org/cabal/`.
Il package che utilizzemo per questo lavoro saranno Parallel (`https://hackage.haskell.org/package/parallel`) per l'esecuzione parallela.
Inoltre è consigliabile un editor di programmazione adatto allo scopo. Un semplice editor di testo è sufficiente, ma è consigliabile utilizzare IDE più specifici, come Leksah o Eclipse. I file di haskell hanno tutti formato *.hs* .

5.2 Perchè Haskell?

Ci sono svariati motivi legati all'utilizzo di Haskell per programmare in parallelo:

- il paradigma sfruttato da Haskell facilita la creazione di codice pulito e conciso;
- possiede una semantica relativamente semplice da acquisire;
- il codice conciso permette di evitare errori di programmazione o di rilevarne più facilmente;
- programmi molto più facili da testare;
- consente un livello di astrazione molto alto;
- le librerie disponibili consentono già una programmazione parallela implicita ad alto livello.

Va fatto notare che la maggior parte delle caratteristiche che sono state enunciate sono rappresentate dal semplice fatto che Haskell è un linguaggio funzionale. In generale i linguaggi funzionali hanno sempre dimostrato un ottimo utilizzo nella progettazione del software, poiché, consentono una maggiore fruibilità di altre fasi di lavorazione, quali stesura delle specifiche (test funzionali) e manutenzione del software (software inspection facilitato dal codice conciso).

5.2.1 Linguaggio funzionale: un linguaggio matematicamente puro

Andiamo più nel dettaglio per quanto riguarda le categorie dei linguaggi informatici. I linguaggi di programmazione più comuni e usati sono solitamente linguaggi *imperativi* . Si pensi ai più famosi, come C, Java e così via. Rappresentano la tecnica di programmazione più classica e "antica": ogni programma è composto da una serie di istruzioni eseguite sequenzialmente, una dopo l'altra. Le varie funzioni (in C) e metodi (in Java) che compongono i programmi sono di fatto *procedure* che lavorano in maniera iterativa.
Al contrario, Haskell è un linguaggio *funzionale*; appartiene, cioè, ad una categoria di linguaggi composta essenzialmente da funzioni matematiche pure, da cui il nome del paradigma. Una funzione in un linguaggio funzionale non esegue calcoli come nella sua controparte imperativa, ma, come in matematica, mappa elementi di un insieme (Dominio) in un altro (Codominio).
Ecco come si presenta una funzione matematica:

$$f(x) = y \tag{5.1}$$

Ad ogni elemento x appartenente al dominio X di definizione, la funzione f associa univocamente il valore y del Codominio Y. In questo caso si dice che y dipende da x in base alla relazione:

$$f : X \to Y \tag{5.2}$$

Tale concetto rappresenta la base dell'algebra e viene implementato elegantemente da tutti i linguaggi funzionali.

Osserviamo un esempio semplice in Haskell, come la funzione incremento:

```
module Esempio where

inc :: Int -> Int
inc x = x + 1
```

Il valore restituito da *inc* dipenderà dall'input (intero) in ingresso. In particolare (aperta la console ghci) otterremo:

```
Prelude Esempio> inc 2
3
Prelude Esempio> inc 4
5
Prelude Esempio> inc 100
101
```

Funzioni di ordine superiore

Ci addentriamo ora in uno dei punti chiave dei linguaggi funzionali. Come già specificato, in questo tipo di linguaggi ogni elemento che compone il programma è rappresentato da una funzione matematica. Ciò rende effettivamente potenti i linguaggi funzionali è lo sfruttamento di questa caratteristica per creare quelle che vengono definite funzioni **higher-order** o di ordine superiore.
Una funzione higher-order è un particolare tipo di funzioni che accetta altre funzioni come parametri o ne restituisce altre come risultato.
Facciamo un esempio aggiungendo al codice di prima una nuova funzione:

```
myOperator :: (Int -> Int) -> Int -> Int
myOperator f n = (f n) * 2
```

In questo caso la funzione *myOperator* accetta in ingresso una funzione da *Int* a *Int* e un valore *Int*. Il suo valore equivale al valore della funzione operata sull'intero in input moltiplicato per 2. Poiché la funzione *inc* corrisponde alle richieste, possiamo sfruttarla per fare qualche esempio di esecuzione:

```
Prelude Esempio> myOperator inc 2
6
Prelude Esempio> myOperator inc 4
10
Prelude Esempio>  myOperator inc 100
202
```

Questi sono esempi basilari, ma dovrebbero rendere conto delle potenzialità inerenti ad un linguaggio funzionale come Haskell. Le funzioni di ordine superiore non solo consentono un alto livello di modularità del codice, ma aprono le porte a numerose funzionalità, tra cui la capacità di incapsulare tra di loro varie funzioni per variarne meccanismi e risultati.

Monade

Altro elemento cardine di Haskell è la **Monade**. Una monade è sostanzialmente una classe di computazioni concatenabili, e consente ad Haskell di eseguire operazioni sequenziali anche nel suo contesto funzionale.
Per rendere più chiaro il concetto, definiamo una monade come una tripla `< M, return, >>= >`, in cui:

- `M` è un costruttore di tipi;
- `return` è una funzione che specifica come costruire i tipi di monadi a partire dal loro contenuto;
- `>>=` è l'operazione di bind; presi due parametri, passa ogni valore prodotto dal primo come argomento per il secondo; è ciò che rende sequenziali le operazioni per i linguaggi funzionali.

In Haskell la classe Monad si presenta così[1]:

```
class Monad m where
  (>>=)  :: m a    -> (a  -> m b) -> m b
  (>>)   :: m a    -> m b -> m b
  -- (>>) è come (>>=) ma scarta il risultato dell'operazione
  return :: a      -> m a
  fail   :: String -> m a
```

5.2.2 Astrazione

Il fatto che ogni elemento di Haskell sia una funzione matematica che può prenderne altre in ingresso o restituirne altre in uscita consente a una funzione di non avere una definizione esplicita di come funziona esattamente una funzione per poter essere sfruttata. Ogni elaborazione può essere sfruttata e intercambiata a piacimento e viene decisa solo in fase di esecuzione.
Tale caratteristica prende il nome di astrazione, la capacità di creare elementi

[1] In Haskell le lettere minuscole nella definizione dei tipi indicano sempre tipi generici; una funzione `f :: a -> b` indica una relazione tra un tipo generico `a` e un'altro tipo generico `b`, non necessariamente della stessa natura.

il cui funzionamento interno è definito in maniera implicita e non diretta.
L'astrazione è consente di modulare il codice a vantaggio della progettazione generale e permette la creazione di strutture ad alto livello intercambiabili e di semplice utilizzo. Un buon livello di astrazione è generalmente indice di una buona programmazione.

5.2.3 Parallelismo in Haskell

All'inizio del capitolo abbiamo distinto due categorie di parallelismo, implicito ed esplicito, stabilendo di voler concentrare la nostra attenzione sulla seconda. Nelle librerie che utilizzeremo Haskell garantisce una sorta di via di mezzo, una forma definita semi-esplicita: per l'esecuzione parallela sono richiesti dall'utente solo alcuni aspetti della coordinazione tra le varie partizioni del programma. Tuttavia ciò non è assolutamente un fatto negativo: cedendo al sistema la maggior parte del lavoro "sporco", ci consentirà di focalizzarci solo sulla stesura del codice.
Il nostro lavoro sarà solo ed esclusivamente partizionare il problema all'interno del codice in sezioni che possano essere eseguite parallelamente, senza precisazioni sulle caratteristiche della macchina che andrà ad eseguire il codice, nemmeno il numero di processori di cui dispone.
Ovviamente l'obiettivo è di mantenere i processori in attività sul problema limitando le interazioni tra gli stessi. Ovviamente i problemi derivanti da questa programmazione non mancano. In primis vi è la dipendenza che può intercorrere tra i dati: due sezioni possono condividere una stessa sezione di memoria o essere l'una in attesa di un risultato offerto dall'altra. Ciò porta ad una sorta di ritorno alla sequenzialità non accettabile visto il nostro obiettivo. In secondo luogo vi è il concetto di granularità dei threads, ovvero la dimensione delle partizioni che andremo a creare. Threads troppo piccoli rischiano di diventare un collo di bottiglia per l'esecuzione, che impegna i processori principalmente a coordinarsi piuttosto che a eseguire lavoro utile, vanificando (o anche rendendo obsoleti) gli sforzi della parallelizzazione.

Capitolo 6

Parallel Haskell

Abbiamo affermato che la finalità di sfruttare Haskell in parallelo (o come viene comunemente chiamato, Parallel Haskell), è di suddividere il lavoro e di eseguirlo su più processori contemporaneamente.
L'esecuzione in parallelo non è implicita nel codice e deve quindi passare con le corrette impostazioni prima a livello di compilatore, e successivamente a livello di esecuzione. Bisogna dunque assicurarsi di aver scaricato l'ultima versione di GHC, o di averlo aggiornato di conseguenza, e, una volta terminato il codice eseguire i seguenti passi:

- compilare il codice con l'opzione `-threaded`; esempio `ghc -threaded foo.hs -o foo`;
- una volta ottenuto il compilato, va mandato in esecuzione con l'opzione `+RTS -N4`; nello specifico `RTS` lega l'esecuzione con un sistema Real-Time (per la gestione autonoma di threads, memoria e così via), mentre l'opzione `-N4` specifica il numero di core da utilizzare, in questo caso quattro (omettere il numero di core da usare consente all'esecuzione di utilizzare tutti quelli presenti nella macchina); esempio `foo +RTS -N4`;
- per analizzare anche le statistiche temporali dell'esecuzione, così da confrontare la parallelizzazione con la controparte sequenziale, è anche utile compilare il codice aggiungendo l'opzione `-rtsopts` ed eseguirlo aggiungendo l'opzione `-sstderr`; l'esecuzione produrrà un file *stder* contenente le specifiche sul tempo di esecuzione e sulle risorse impiegate.

6.1 Parallelismo puro: Control.Parallel

Cominciamo a descrivere la parallelizzazione esplicita in Haskell partendo dalla libreria più importante in questo ambito, ciò che ci consentirà di trasformare il nostro codice sequenziale in codice parallelo.
La libreria Control.Parallel si compone essenzialmente di due funzioni, par e pseq, così definite:

```
par :: a -> b -> b
pseq :: a -> b -> b
```

La funzione par

La funzione `par f1 f2`, esprimibile anche come `f1 'par' f2`, si traduce sostanzialmente in: "l'esecuzione di f1 può essere parallelizzata con f2; il valore di ritorno è il risultato di f2".
Questa funzione viene usata per effettuare l'esecuzione parallela di due elementi, di cui il valore del primo non è richiesto immediatamente.

La funzione pseq

La funzione `pseq f1 f2`, esprimibile anche come `f1 'pseq' f2`, è la controparte parallela della funzione `seq`, della libreria standard di Haskell. Nel caso di `seq` si tratta di una funzione sfruttata per ottimizzare le prestazioni dell'esecuzione, mediante la gestione della lazy evaluation: `seq f1 f2` restituisce il valore di `f2` solo dopo che entrambe le funzioni in input hanno terminato il loro ciclo di esecuzione[1].
Analogamente la funzione `pseq f1 f2` si regge sullo stesso principio di base, ma con una particolarità in più: in genere, quando si tratta di parallelismo, vogliamo sfruttare il controllo sulla lazy evaluation in modo da valutare f1 prima di f2, magari perché abbiamo già specificato che i due vanno eseguiti in parallelo.
In tal caso `seq` non è sufficiente, poiché legata a entrambi gli argomenti. Con `pseq`, invece, ci assicuriamo che l'esecuzione avvenga nell'ordine corretto.

[1] Piccola nota: ciò non significa che `f2` viene eseguita dopo `f1`; si tratta solo di un modo per impedire inutili valutazioni pigre

6.1.1 Esempio: la serie di Fibonacci

Proviamo con l'"Hello World" degli algoritmi, uno semplici da implementare. Ricavare l'n-esimo elemento della serie di Fibonacci in Haskell diventa:

```
fib :: Int -> Int

fib 0 = 1
fib 1 = 1
fib n = (fib (n-1)) + (fib (n-2))
```

Parallelizziamo ora il codice usando pseq e par:

```
fibpar :: Int -> Int
fibpar 0 = 1
fibpar 1 = 1
fibpar n = (n1 `par` n2) `pseq` (n1 + n2)
           where
               n1 = fibpar (n-1)
               n2 = fibpar (n-2)
```

In questo caso, dato un generico intero `n`, viene parallelizzata l'esecuzione di Fibonacci su `n-1` insieme a quella di `n-2`, specificando che la loro somma deve attendere che restituiscano un risultato.
Notiamo che di per sé gli algoritmi sono concettualmente identici: effettuano la ricorsione due volte per chiamata e uniscono i risultati una volta terminate entrambe le esecuzioni.
Confrontiamone ora il tempo di esecuzione effettuato su input 46: il codice sequenziale ha restituito il risultato dopo 260 secondi, quello parallelo in 58. Ricaviamo quindi uno speedup di:

$$Speedup = \frac{260sec}{58sec} = 4.48 \simeq 450\% \tag{6.1}$$

6.2 Le strategie di calcolo

Un altro modo per parallelizzare l'esecuzione in Haskell è l'utilizzo delle **strategie**. Una strategia è essenzialmente una funzione in una particolare monade[2], Eval, che prende in ingresso il tipo da associare alla stessa. Il meccanismo in Haskell è il seguente:

```
type Strategy a = a -> Eval a
```

6.2.1 La monade Eval

La monade Eval è una struttura contenuta nel package Contol.Parallel.Strategies. Vediamo nel dettaglio come la sua composizione:

```
data Eval a
instance Monad Eval

rpar :: a -> Eval a
rseq :: a -> Eval a

runEval :: Eval a -> a
```

Si compone principalmente di due funzioni, **rpar** e **rseq**, semanticamente e funzionalmente identiche alle sopracitate **par** e **pseq** del package Control.Parallel (si veda il capitolo precedente). L'unica differenza sostanziale è il tipo che viene restituito: trattandosi di funzioni in seno a una monade, restituiranno, per l'appunto, un elemento Eval (con relativo tipo).
Vi è poi un'altra funzione, **runEval**, finalizzata ad effettuare la computazione della monade e restituirne il risultato.

6.2.2 Usare le strategie

Avviamoci ora verso l'utilizzo del tipo **Strategy**. Usare una strategia significa prendere in ingresso una struttura dati, utilizzare le funzioni **rpar** e **rseq** per creare parallelismo e restituire il valore del parametro in ingresso.
In questo caso le funzioni di Eval possono essere riscritte in funzione delle strategie:

```
rpar :: Strategy a
rseq :: Strategy a
```

[2]Si veda il capitolo "Perchè Haskell per un chiarimento sulle monadi."

Chiariamo il loro utilizzo con un esempio. Immaginiamo di aver creato una funzione f e di aver poi deciso di eseguirla sfruttando la strategia s. Ricavare il risultato di f diventa:

```
runEval (s f)
```

Spesso questa scrittura viene sostituita da una funzione più compatta, *using*:

```
using :: Strategy a -> a
using s x = runEval (s x)
```

Possiamo quindi riscrivere la funzione precedente come:

```
s `using` f
```

Un corretto utilizzo delle strategie non è solo un metodo stringente per definire parallelismo. Grazie alla potenza delle astrazioni di Haskell è possibile parallelizzare un algoritmo già esistente semplicemente applicandovi una strategia per costruire il risultato.

6.2.3 Esempio: la funzione map in parallelo

Vediamo ora come l'utilizzo delle strategie possa rendere la parallelizzazione concisa ed efficace. La funzione map prende in ingresso due parametri: una funzione (`f :: a -> b`) e una lista di elementi di tipo `a`. Il nostro intento è di rendere parallela questa operazione senza modificare l'algoritmo di partenza.
La funzione più semplice finalizzata a questo intento è:

```
parMapList :: (a -> b) -> [a] -> [b]
parMapList f ls = map f s `using` parList rseq
```

Osserviamo nel dettaglio le componenti della funzione:

- `map f s` è l'algoritmo a cui applicare la strategia
- `parList rseq` è la strategia da sfruttare; `rseq` l'abbiamo già menzionata, parList, invece, è una strategia già implementata in Control.Parallel.Strategies, che presa una strategia su valori `'a'` ne restituisce un'altra sulle liste di elementi di tipo `'a'`; in questo caso prende la strategia `rseq` e ne restituisce una che applica la suddetta ad ogni elemento di un lista

Osserviamo alcuni elementi fondamentali: prima di tutto non abbiamo creato nessuna funzione ex novo, abbiamo solo assemblato algoritmi preesistenti e vi abbiamo applicato un meccanismo di calcolo; in secondo luogo abbiamo messo in rilievo che è possibile creare strategie di livello superiore senza particolari assunzioni su tipi e strutture.

6.3 Limiti di Haskell

Come è stato affermato in questo capitolo, non esiste il linguaggio perfetto, e anche Haskell non è immune a questa regola. Pur essendo comodo e di facile utilizzo manca di una gestione diretta della memoria, almeno a livello virtuale, come invece consente l'utilizzo di C, grazie ai suoi puntatori. Anche per questo motivo è bene dosare bene i linguaggi da utilizzare: non è difficile immaginare che programmi relativamente semplici, che si poggiano su algoritmi minimali, trovino una migliore applicazione in un linguaggio imperativo come C piuttosto che in Haskell.
Analogamente un programma in Parallel Haskell eseguito su un problema di piccole dimensioni potrebbe creare colli di bottiglia tra overhead e comunicazione tra processori di gran lunga superiori all'effettiva esecuzione del problema rispetto alla sua controparte sequenziale (si ricordi il concetto di granularità).
Inoltre se l'efficienza degli algoritmi è l'obiettivo principale del programmatore, Haskell potrebbe non rivelarsi la scelta migliore quanto a gestione delle risorse della macchina (si pensi alle strategie, in cui la parallelizzazione avviene a livelli di astrazione troppo elevati per pensare di ottenere una efficienza ottimale in tutti i casi in cui viene utilizzata).
Tuttavia, tolti i problemi legati ad una cattiva progettazione, dare priorità all'ottimalità dell'esecuzione non sempre è la scelta migliore. Da questo punto di vista il linguaggio macchina sarebbe la scelta migliore per programmare, eppure è paradossale pensare che si programmi anche solo una semplice sommatoria in un linguaggio così a basso livello.
Come per molte cose, quindi, un buon compromesso è sempre la scelta migliore. Haskell non è stato scelto per essere il più ottimale dei linguaggi, ma perché consente al programmatore una gamma vastissima di scelte progettuali sposate a una buona implementazione della parallelizzazione esplicita.
Se la priorità del programmatore è una buona progettazione del software, un linguaggio funzionale come questo è quasi sempre la scelta migliore.

Parte IV

Implementazione in Haskell

Capitolo 7

Problemi implementati

7.1 Ordinamento

7.1.1 QuickSort

```
--Quicksort Sequenziale

quicksort :: Ord a => [a] -> [a]
quicksort [] = []
quicksort (x:xs) = left ++ [x] ++ right where
left = quicksort [a | a <- xs, a <= x]
right = quicksort [a | a <- xs, a > x]

--Quicksort Parallelo

parquicksort :: Ord a => [a] -> [a]
parquicksort [] = []
parquicksort (x:xs) = (left `par` right) `pseq` (left ++ [x] ++ right)
where
left = parquicksort [a | a <- xs, a <= x]
right = parquicksort [a | a <- xs, a > x]
```

7.1.2 Mergesort

```
--Funzioni ausiliarie

merge :: Ord a => [a] -> [a] -> [a]
merge xs [] = xs
merge [] ys = ys
merge (x:xs) (y:ys)
| x <= y = x:(merge xs (y:ys))
| otherwise = y:(merge (x:xs) ys)

forceList :: [a] -> ()
forceList [] = ()
forceList (x:xs) = x `pseq` (forceList xs)

--Mergesort Sequenziale

mergesort :: Ord a => [a] -> [a]
mergesort [] = []
mergesort [x] = [x]
mergesort xs = merge (mergesort left) (mergesort right)
where
(left, right) = splitAt l xs
l = div (length xs) 2

--Mergesort Parallelo

parmergesort :: Ord a => [a] -> [a]
parmergesort [] = []
parmergesort [x] = [x]
parmergesort xs = ((forceList left) `par` (forceList right))
                  `pseq` (merge left right)
        where
                              (left1, right1) = splitAt l xs
                              l = div (length xs) 2
                              left = parmergesort left1
                              right = parmergesort right1
```

7.2 Algebra Lineare

```
module Matrix where

import Data.List
import System.Random
import System.IO
import Control.Parallel
import Control.Parallel.Strategies

type Vector = [Double]
type Matrix = [Vector]

makeMat :: Int -> Matrix
makeMat n = replicate n [1.0..t] where t = fromIntegral n

zeroes n = replicate n (replicate n 0.0)

identity :: Int -> Matrix
identity n = identity2 n 0

identity2 :: Int -> Int -> Matrix
identity2 n m
    | n == m = []
    | otherwise = [((replicate m 0) ++ [1]
                   ++ (replicate (n-m-1) 0))] ++ (identity2 n (m+1))

--operazioni sequenziali tra matrici

sumVector :: Vector -> Vector -> Vector
sumVector v1 v2 = [a + b | a <- v1, b <- v2]

subVector :: Vector -> Vector -> Vector
subVector v1 v2 = [a - b | a <- v1, b <- v2]

scalar :: Vector -> Vector -> Double
scalar a b = sum (zipWith (*) a b)


sumMatrix :: Matrix -> Matrix -> Matrix
sumMatrix = zipWith (zipWith (+))
```

```
subMatrix :: Matrix -> Matrix -> Matrix
subMatrix = zipWith (zipWith (-))

prodMatrix :: Matrix -> Matrix -> Matrix
prodMatrix m1 m2 = [[scalar a b | b <- column] | a <- m1]
                   where column = transpose m2

powMatrix :: Matrix -> Int -> Matrix
powMatrix m 0 = (identity dim) where dim = length m
powMatrix m 1 = m
powMatrix m n = prodMatrix m (powMatrix m (n-1))


--operazioni parallele tra matrici

sumMatPar :: Matrix -> Matrix -> Matrix
sumMatPar a b = (sumMatrix a b) `using` parList rdeepseq

subMatPar :: Matrix -> Matrix -> Matrix
subMatPar a b = (subMatrix a b) `using` parList rdeepseq

prodMatPar :: Matrix -> Matrix -> Matrix
prodMatPar a b = (prodMatrix a b) `using` parList rdeepseq
```

```
powMatPar :: Matrix -> Int -> Matrix
powMatPar m 0 = (identity dim) where dim = length m
powMatPar m 1 = m
powMatPar m n = prodMatPar m ris
                    where
                         ris = powMatPar m (n-1)

--determinante

deleteColumn :: Matrix -> Int -> Matrix
deleteColumn [] _ = error "Error input Matrix"
deleteColumn m col = a ++ b where (a, _:b) = splitAt col m

deleteElement :: Vector -> Int -> Vector
deleteElement x index = left ++ right
where (left, _:right) = splitAt index x

deleteRow :: Matrix -> Int -> Matrix
deleteRow [] _ = error "Error input Matrix"
deleteRow m row = [deleteElement x row | x <- m]


minor :: Matrix -> Int -> Int -> Matrix
minor [] _ _ = error "Error input Matrix"
minor m row col = deleteRow m1 row where m1 = (deleteColumn m col)

det :: Matrix -> Double
det [] = error "Error input Matrix"
det [[x]] = x
det m = sum [a*s*(det m1) | i <- [0..dim-1],
            let a = (head m !! i), let m1 = (minor m i 0), let s = (-1)^i]
                where
                     dim = length m
```

```
--determinante parallelo


detList :: Matrix -> Vector
detList [] = error "Error input Matrix"
detList [[x]] = [x]
detList m = [a*s*(det m1) | i <- [0..dim-1],
            let a = (head m !! i), let m1 = (minor m i 0), let s = (-1)^i]
                  where
                      dim = length m

pardet :: Matrix -> Double
pardet m = sum ((detList m) `using` parList rpar)

--matrice inversa

invert :: Matrix -> Matrix
invert [[]] = [[]]
invert m = transpose [[s*(det m1)/d | i <- [0..dim-1],
               let m1 = (minor m i j), let s = (-1)^(i+j)] | j <- [0..dim-1]]
                         where
                              dim = length m
                              d = det m

--matrice inversa parallela

parinvert :: Matrix -> Matrix
parinvert m = (invert m) `using` parList rdeepseq
```

7.3 Grafi

```
module Graph where

import Matrix

type Graph = ([Node], [Edge])
type Node = Int
type Edge = (Node, Node)


adjMat :: Graph -> Matrix
adjMat (nodes, []) = zeroes dim
                  where
                        dim = length nodes
adjMat (nodes, e:edges) = setEdge e 1.0 (adjMat (nodes, edges))

setEdge :: Edge -> Double -> Matrix -> Matrix
setEdge _ _ [] = error "Error out of bound"
setEdge (x, y) v m = replacePosition x y v (replacePosition y x v m)

replacePosition :: Int -> Int -> Double -> Matrix -> Matrix
replacePosition x y v m = left ++ [new] ++ right
                      where
                         (left, old:right) = splitAt x m
                         (left2, _:right2) = splitAt y old
                         new = left2 ++ [v] ++ right2
```

```
normalize :: Matrix -> Matrix
normalize m = [[reduce x | x <- y] | y <- m]
               where reduce a
                     | a==0 = 0
                     | otherwise = 1

fullyConnected :: Matrix -> Bool
fullyConnected [] = True
fullyConnected (x:xs) = (all (/= 0.0) x) && fullyConnected xs

isConnected :: Matrix -> Bool
isConnected m = checkConnection m1 m1 where m1 = normalize m



checkConnection :: Matrix -> Matrix -> Bool
checkConnection m1 m2
      | fullyConnected m1 = True
      | m3 == m1 = False
      | otherwise = checkConnection m3 m2
         where m3 = normalize (prodMatrix m1 m2)
```

Bibliografia

[1] Io me medesimo. “Lor Anittal”. In: *Boh* 1 (1 2016), pp. 2–3.