UNIVERSITÀ DEGLI STUDI DI MILANO
FACOLTÀ DI SCIENZE E TECNOLOGIE

CORSO DI LAUREA TRIENNALE IN INFORMATICA

PARALLELISMO E HASKELL

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

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

ANNO ACCADEMICO 2015-2016

Prefazione

In campo informatico, soprattutto dal punto di vista della programmazione, lo studio degli algoritmi è alla base di qualsiasi tipo di esecuzione. Tuttavia, oltre al caso sequenziale, a cui tutti siamo generalmente abituati quando scriviamo un programma, esiste un'alternativa chiamata "parallelismo": si tratta della capacità di eseguire codice su un sistema dotato di più processori che possono lavorare, appunto, in parallelo.
Questo lavoro nasce in seguito ad un'analisi del concetto di algoritmi sia in ambito sequenziale, sia in ambito parallelo, ed alla necessità di trovare un linguaggio consono a questa esigenza. Tra tutti, Haskell ha rappresentato una delle scelte migliori in questo campo. Forte delle sue caratteristiche di linguaggio funzionale e di una vasta gamma di librerie che supportano vari livelli di programmazione parallela, Haskell consente di esprimere parallelismo sia esplicito che implicito già a livello di codice. Il tutto mediante una semantica di comodo utilizzo, a vantaggio del programmatore.
L'elaborato presenterà prima un'introduzione del calcolo sequenziale, per poi passare il concetto di calcolo parallelo (dal punto di vista teorico) e concluderà con una presentazione di alcuni modelli messi a disposizione da Haskell. La parte finale mostrerà, inoltre, una serie di algoritmi implementati e testati su una macchina multi-core.

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 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 f_A 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. In questo caso si parla di calcolo sequenziale, poiché le operazioni che definiscono la procedura desiderata vengono eseguite una alla volta (in sequenza).
Formalmente questo significa che, 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 $x \in I$.
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 appartenenti al dominio di A. 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, mentre un array di n elementi ha dimensione n.
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 sempre $n + (n-1) + .. + 1 = \frac{n(n-1)}{2}$ passi per ottenere un vettore ordinato di dimensione n.
Tuttavia in molti problemi 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 $T_A(x) \neq T_A(x')$.
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.
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

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 $n \to +\infty$. Questo rende possibile non solo ottenere un buon metodo di confronto, ma anche classificare i vari algoritmi rispetto a questa stima.
Nonostante questo meccanismo tenda a vacillare in caso di problemi relativamente piccoli, 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 pari a $f(n)$,

[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.

diciamo che l'algoritmo che lo risolve è **ottimo** se qualunque altro algoritmo che lo risolve ha complessità $\mathcal{O}(f(n))$.

Capitolo 2

Esempi di Problemi Risolti

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

L'ordinamento di un insieme di oggetti confrontabili è uno degli problemi tipici del calcolo. 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 e mergesort. In pseudocodice saranno:

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

merge(left, right)
    begin
    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)
    end
```

In questo caso, ad esempio i vari casi di esecuzione cambiano, anche a seconda del caso in analisi:

- 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.

2.2 Algebra Lineare

Facciamo degli esempi riguardanti le operazioni con matrici:

- prodotto tra matrici;
- calcolo del determinante;
- inversione di una matrice.

Nel primo caso l'algoritmo per calcolare il prodotto tra due matrici A e B è il seguente:

```
ProdottoMatrici(A : matrice MxN, B : matrice NxP)
    begin
        C = matrice di zeri MxP
        for i = 1,..,M do
            for j = 1,..,P do
                for k = 1,..,N do
                    C[i][j] = C[i,j] + A[i][k] + B[k][j]
                done
            done
        done
        return C
    end
```

In caso di prodotto di una matrice MxN con una NxP l'algoritmo avrà complessità asintotica $\mathcal{O}(N * M * P)$; nel caso peggiore questo si traduce in una complessità $\mathcal{O}(n^3)$.
Nel caso del calcolo del determinante di una matrice quadrata A di dimensione n, solitamente viene sfruttato il metodo di Laplace, mediante il seguente algoritmo ricorsivo:

$$det(A) = \begin{cases} a_{1,1} & se\ n\ =\ 1 \\ \sum_{i=1}^{n}(-1)^{i+k}a_{i,k}det(A_{i,k}) & altrimenti \end{cases}$$

in cui, preso un qualsiasi $i \in [1, .., n]$, $a_{i,k}$ è l'elemento (i, k) della matrice A e $A_{i,k}$ è il minore ottenuto eliminando da A la *i-esima* riga e la *k-esima* colonna.
In questo caso la complessità è evidentemente inefficiente: si tratta di calcolare ricorsivamente per ogni determinante di ordine n altri n determinanti di ordine $n - 1$. In altre parole possiede una complessità di $\mathcal{O}(n!)$. Si tratta ovviamente di un algoritmo altamente sconveniente da utilizzare, anche per problemi di piccole dimensioni. In questi casi è consigliabile cambiare strategia: una possibile alternativa è rappresentata dal *metodo di eliminazione di Gauss*. In sostanza si converte una matrice quadrata in una matrice triangolare (cioè formata da zeri al di sotto/sopra della sua diagonale) e calcolarne il determinante (operazione che si riduce a moltiplicare gli elementi sulla diagonale).

La procedura consiste in tre passaggi:

1. *Se la prima riga ha il primo elemento nullo, essa viene scambiata con una riga che ha il primo elemento non nullo. Se tutte le righe hanno il primo elemento nullo, vai al punto 3.*

2. *Per ogni riga A_i con primo elemento non nullo, eccetto la prima, moltiplica la prima riga per un coefficiente scelto in maniera tale che la somma tra la prima riga e A_i abbia il primo elemento nullo (quindi coefficiente $= -\frac{a_{i1}}{a_{11}}$. Sostituisci A_i con la somma appena ricavata.*

3. *Ripeti il punto 1 sulla sottomatrice ottenuta cancellando la prima riga e la prima colonna.*

In questo caso cambiare algoritmo offre un guadagno considerevole: da una complessità fattoriale si passa ad una polinomiale, pari a $\mathcal{O}(n^2)$.
L'ultimo esempio dell'algebra lineare che affronteremo è quello della matrice inversa. Il metodo più semplice è quel calcolo dei cofattori, in cui data una matrice A invertibile ($det(A) \neq 0$) allora:

$$A^{-1} = \frac{1}{det(A)}(cof(A))^T$$

In questo caso $cof(A)$ è la matrice dei cofattori o dei complementi algebrici di A, $a_{11}, .., a_{nn}$, ottenuti nel seguente modo:

$$a_{ij} = (-1)^{i+j} det(A_{ij})$$

Con $i, j \rightarrow [1, .., n]$, A_{ij} = A priva della $i - esima$ riga e della $j - esima$ colonna.
In questo caso la complessità dipende esclusivamente dall'algoritmo usato per ottenere il determinante: poiché in questo caso è un'operazione eseguita n^2 volte, l'efficienza/inefficienza si conserva.

2.3 Grafi

Un grafo è una struttura matematica composta da un insieme di elementi detti nodi che si possono collegare tra loro mediante oggetti archi. Formalmente un grafo G è una coppia di elementi $< N, E >$ in cui N è l'insieme dei nodi, ed E l'insieme degli archi; ogni arco è definito come $e = (n1, n2)$ in cui $n1, n2 \in N$ sono i nodi collegati. Definiamo inoltre "cammino" da un nodo n_1 a un nodo n_m una sequenza di archi $(n_1, n_2), (n_2, n_3), .., (n_{m-1}, n_m)$.

Esistono numerose proprietà e operazioni sui grafi, ma ci concentreremo su alcuni classici problemi riguardanti i grafi:

- grafo connesso;
- cammino minimo.

Molti di questi problemi sfrutteranno le leggi dell'algebra lineare, ovvero costruendo una matrice di adiacenza così definita: si crea una matrice quadrata di dimensione pari alla cardinalità dell'insieme dei nodi in cui ogni elemento m_{ij} della matrice rappresenta un possibile collegamento tra i nodi n_i e n_j; se l'arco (n_i, n_j) o (n_j, n_i) appartengono all'insieme degli archi del grafo, allora $m_{ij} = 1$, altrimenti $m_{ij} = 0$. Poiché semplicità studieremo i grafi non orientati (ovvero i grafi tali che ogni arco $i - j$ rappresenta anche un arco $j - i$). Analizziamo ora i vari problemi nel dettaglio. Nel primo caso diremo che un grafo $G =< N, E >$ è connesso se e solo se per ogni coppia di nodi $n_i, n_j \in N$ esiste un cammino che li unisce. Il metodo più semplice per verificare questa proprietà è quello sfruttare la matrice di adiacenza M relativa G.
L'algoritmo è il seguente:

```
n = numero di nodi
M = matrice di adiacenza
for k = 1,..,n do
    for j = 1,..,n do
        for i = 1,..,n do
            if Mij = 0 then Mij = Mik && Ckj
        done
    done
done
```

Se al termine dell'algoritmo la matrice contiene degli zeri il grafo non è connesso.
Si tratta di tre cicli innestati, proporzionali alla dimensione del grafo, dunque la complessità di calcolo equivale a $\mathcal{O}(n^3)$.
Il secondo caso che affronteremo è la ricerca di un cammino di costo minimo tra due nodi raggiungibili. Esistono molti algoritmi che risolvono questo problema, ma ci concentreremo su quello più utilizzato e conosciuto: l'algoritmo di Dijkstra. Si tratta di un algoritmo appartenete alla categoria degli *algoritmi greedy*, ovvero quelli che, partendo da una soluzione parziale, la estendono fino a che non è più possibile. A ogni iterazione l'algoritmo sceglie l'estensione giudicata più conveniente e ripete l'esecuzione.
L'istanza del problema è data da un grafo $G =< N, E >$, una sorgente $s \in N$ e una funzione $w : E \to \mathbb{Q}$ che definisce un costo (o "peso")[1] di ogni arco presente nel grafo. Definiamo inoltre $L(n)$ la lista dei nodi raggiungibili da n con un arco $e \in E$. Dati questi elementi l'algoritmo calcolerà i cammini di costo minimo che collegano s con ogni nodo $n \in N$.
Prima di eseguire l'algoritmo si definiscono tre insiemi: l'insieme S dei nodi per cui la soluzione è già stata calcolata, l'insieme D dei nodi non presenti in S e raggiungibili mediante un arco dai nodi presenti in S, e un insieme contenente tutti gli altri nodi rimanenti. A ogni nodo n appartenente a D sono inoltre associate due procedure $C(n)$ e $P(n)$: il primo indica il costo del cammino minimo da s a n trovato finora, il secondo indica l'ultimo nodo prima di n in tale cammino.

[1] Si considererà $w(e) \geq 0 \ \forall e \in E$

La procedura è la seguente:

```
Dijkstra(N, E, w, s)
    begin
        D = {s}
        for n in N do
            C(n) = infinity
            P(n) = NULL
        done
        C(s) = 0
        while D \= {} do
            begin
                seleziona il nodo v in D avente C(v) minore
                cancella n da D
                for u in L(u) do
                    if C(v) + w(v,u) < C(u) then
                        if C(u) = infinity then
                            aggiungi u a D
                        end if
                        C(u) = C(v) + w(v,u)
                        P(u) = v
                    end if
                done
            end
        return C, P
    end
```

Parte II

Calcolo Parallelo

Capitolo 3

Gli Algoritmi Paralleli

La definizione di calcolo sequenziale è legata al fatto che rappresenta procedure e funzioni eseguite da un singolo processore. Di conseguenza la velocità di esecuzione di un algoritmo è legata sia alla complessità che lo definisce sia alla velocità di calcolo del processore che lo esegue. In casi particolari, come i sistemi real time, questa caratteristica assume un ruolo critico e le limitazioni del calcolo sequenziale diventano evidenti.
Una possibile soluzione a questo problema è rappresentata dalla suddivisione del lavoro in sezioni da eseguire contemporaneamente da più processori, nel tentativo di ottenere una significativa riduzione al tempo di calcolo proporzionale all'aumento di risorse utilizzate. In questo modo a ogni ciclo di clock corrisponde un numero al più uguale al numero di processori in uso. I sistemi di questo tipo vengono detti *paralleli*.
Come si può facilmente intuire l'esecuzione parallela necessita di calcoli aggiuntivi per coordinare i vari processori, aggiunta che può diventare problematica se mal gestita (approfondiremo questo concetto in un secondo momento). La comunicazione tra i processori può avvenire in due modalità, a seconda della categoria del modello in uso: distinguiamo due famiglie di sistemi paralleli, ovvero i modelli a memoria condivisa e i modelli a memoria distribuita. Nel primo caso i processori utilizzano contemporaneamente una memoria globale, che viene usata anche per casi di coordinazione (i processori non comunicano direttamente tra loro). Nei sistemi distribuiti, invece, ogni processore ha una propria memoria locale, inaccessibile dagli altri, e la comunicazione avviene in maniera diretta. La maggior parte dei sistemi in uso sfruttano processori multicore a memoria condivisa, dunque ci concentreremo principalmente su questa categoria di modelli.

Il passaggio da sequenziale a parallelo offre sicuramente la possibilità di ottenere un guadagno considerevole in termini di esecuzione, ma al tempo stesso apre la porta a nuove sfide:

- sono necessarie ulteriori analisi del problema, in particolare se e come è possibile creare una procedura capace di parallelizzarlo (le parti che è possibile eseguire parallelamente);
- esistono problemi impossibili da parallelizzare;
- esistono problemi che in ambito parallelo sono meno performanti che in ambito sequenziale;
- è necessaria una coordinazione tra i vari processori;
- la programmazione parallela spesso risulta più complicata di quella sequenziale.

Nel corso di questo capitolo analizzeremo l'approccio a queste problematiche, servendoci di un modello di calcolo ad analizzare il parallelismo.

3.1 La macchina PRAM

Il modello che andremo ad analizzare è essenzialmente un'evoluzione di un particolare modello sequenziale. Nel capitolo precendente ci siamo serviti della Macchina di Turing come modello di calcolo ideale per analizzare le caratteristiche degli algoritmi sequenziali. Un altro esempio di modello sequenziale è la macchina RAM (Random Access Memory), formata essenzialmente da un singolo processore, capace di eseguire un set di operazioni elementari, e una memoria potenzialmente infinita di celle, accessibili dal processore in maniera casuale.
Questo modello nel calcolo parallelo definisce la base di quello che è il modello di calcolo che useremo: la macchina PRAM, un modello composto da un insieme di p processori RAM e una memoria globale.
Come abbiamo già affermato l'esecuzione dei processori avviene in maniera indipendente: l'esecuzione di un attività su un processore non influisce l'esecuzione di uno dei suoi "compagni". 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. In caso di esecuzione parallela ad un singolo ciclo di clock corrispondo molteplici istruzioni eseguite in contemporanea: in questo modo si ottiene il parallelismo desiderato.
Si distinguono i seguenti tipi di PRAM:

- 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.

Nel seguito illustreremo in particolare modelli EREW e CREW, sia dal punto di vista implementativo che da quello analitico.
In generale un'istruzione parallela ha questa forma:

```
for (i from 1 to NumProcessori) do
           operazioneParallela
```

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.
Indicheremo con $\mathrm{T}_A(n,p)$ il tempo di calcolo legato ad un algoritmo parallelo A eseguito su p processori su un generico input di dimensioni n. 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.
Poiché lo scopo della parallelizzazione è quello di ottenere un compromesso conveniente tra prestazioni e costo delle risorse aggiuntive ci serviremo di due misure particolari: lo *Speedup* e l'*Efficienza*, che indicheremo rispettivamente con $\mathrm{S}_A(n,p)$ e $\mathrm{E}_A(n,p)$.
Nel dettaglio:

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

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

Notiamo che lo Speedup rappresenta il guadagno in termini di tempo di esecuzione relativa all'utilizzo di p processori e l'efficienza risulta essere il rapporto tra il tempo dell'algoritmo sequenziale e il tempo totale consumato dai processori, come se fossero usati sequenzialmente. In altri termini lo Speedup misura la riduzione del tempo di calcolo, mentre l'efficienza definisce quanto l'algoritmo in uso sfrutta il parallelismo della macchina.
Dato un algoritmo A che lavora con p processori con una data efficienza E, è in generale possibile estendere l'algoritmo a lavorare con un numero inferiore di processori senza che l'efficienza diminuisca significativamente:

$$\text{Se } k \geq 1 \ \rightarrow \ E_A(n, \tfrac{p}{k}) \geq E_A(n, p)$$

Dato un algoritmo A che lavora con p processori, basta infatti costruire un algoritmo che utilizza $\frac{p}{k}$ processori. Ad ogni nuovo processore si fa corrispondere un blocco di k vecchi processori: ogni nuovo processore usa al più k passi per emulare il singolo passo parallelo dei k processori corrispondenti. Quindi vale che:

$$T_A(n, \frac{p}{k}) \leq kT_A(n, p)$$

Osservando che:

$$E_A(n, \frac{p}{k}) = \frac{T_A(n,1)}{\frac{p}{k}T_A(n, \frac{p}{k})} \geq \frac{T_A(n,1)}{pT_A(n, \frac{p}{k})} = E_A(n, p)$$

concludiamo che l'efficienza non diminuisce diminuendo i processori. In particolare, poiché $E_A(n, p) \leq E_A(n, 1) = 1$, l'efficienza non può superare 1, ovvero il caso ideale, in cui l'algoritmo utilizza tutti i processori per tutta l'esecuzione.
Ciò che limita lo speedup e di conseguenza l'efficienza degli algoritmi rappresenta l'*Overhead* di esecuzione, ovvero calcoli non necessari alla sola risoluzione del problema, e può essere definito come:

$$O_A(n) = pT_A(n, p) - T_A(n, 1)$$

Il tempo speso in Overhead è determinato da diversi fattori: momenti di idle, eventi di comunicazione, ecc.

3.2 Limiti della Parallelizzazione

Le leggi inerenti allo speedup e all'efficienza specificano implicitamente che non sempre l'aumento di più processori garantisce un guadagno in termini di prestazioni. Proviamo a dimostrarlo con un semplice esempio: immaginiamo di voler definire un algoritmo A che calcoli la somma di n numeri disposti in un array, dividendo quest'ultimo in p parti pari al numero di processori utilizzati. I processori sommeranno i valori contenuti nei loro array e restituiranno il risultato una volta finita l'esecuzione, dopodiché un singolo processore raggrupperà e sommerà tra loro tutti i risultati ottenuti. Il tempo di esecuzione sarà pari a:

$$\begin{cases} T_A(n,1) \simeq n \\ T_A(n,p) \simeq +\frac{n}{p}\log_2(p) \end{cases}$$

Il conseguente overhead risulta quindi:

$$O_A(n) \simeq pT_A(n,p) - T_A(n,1) = p(\tfrac{n}{p} + \log_2(p)) - n = p\log_2(p)$$

Deduciamo quindi che l'overhead aumenta al crescere del numero di processori (in questo caso con una crescita pari a $\mathcal{O}(p\log_2(p))$).
L'unico modo per limitare calcoli di overhead è cercare di ottenere uno speedup prossimo al caso ideale (ovvero avente crescita lineare). Per ottenere questo risultato occorre un'ulteriore analisi relativa al tempo di esecuzione. Dividiamo il tempo $T_A(n,1)$ in due parti: chiameremo T_{As} il tempo inerente alle operazioni eseguite esclusivamente in ambiente sequenziale e T_{Ap} quello inerente alle operazioni eseguite in parallelo; ovviamente $T_A(n,1) = T_{As}(n)+T_{Ap}(n)$. Ora possiamo ottenere $T_A(n,p)$ $p \geq 2$ in funzione di $T_A(n,1)$. Notiamo che, mentre la parte sequenziale rimane fissa al valore T_{As}, la componente parallela decresce in proporzione a p. Quindi otterremo:

$$T_A(n,p) = T_{As}(n) + \frac{T_{Ap}(n)}{p}$$

Se chiamiamo α la componente sequenziale T_{As} otteniamo che:

$$S_A(n,p) = \frac{1}{\alpha + \dfrac{1-\alpha}{p}}$$

La formula che abbiamo ottenuto non è altro che un caso particolare della *legge di Amdahl*, che afferma un concetto coerente con quello che abbiamo cercato di esprimere:

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

Intuitivamente è ragionevole che la miglioria legata all'aumento di processori perda il suo effetto nel momento in cui venga utilizzato esclusivamente il calcolo sequenziale. Infatti, poiché il miglioramento della componente parallela si riduce ad aumentare il numero di processori in uso, se facciamo tendere p a ∞ allora $S_A(n,p) \to \frac{1}{\alpha}$. In altre parole lo speedup ottenuto dalla parallelizzazione di qualsiasi algoritmo è sempre limitato dalla sua componente sequenziale.
Da questi risultati possiamo anche affermare che:

$$E_A(n,p) = \frac{S_A(n,p)}{p} = \frac{1}{\alpha + (1-\alpha)p} = \frac{1}{\alpha(p-1)+1}$$

$$O_A(n,p) = (p-1)\alpha$$

Capitolo 4

Esempi di Problemi Paralleli Risolti

Di seguito verranno presentati i casi di problemi presentati precedentemente nel caso parallelo.

4.1 Ordinamento

Abbiamo già mostrato che l'algoritmo Mergesort si compone essenzialmente di due fasi di calcolo: nel caso base di array di due elementi l'ordinamento è banale; in caso contrario l'algoritmo divide un array a metà, lo ordina ricorsivamente e effettua l'operazione di merge sui due risultati. L'operazione di merging, essendo efficiente (complessità $\mathcal{O}(n)$), può essere lasciata invariata ed eseguita sequenzialmente: avendo a disposizione p processori, il parallelismo si limiterà semplicemente ad assegnarne $\frac{p}{2}$ per le due parti in cui abbiamo diviso l'array. In altre parole il parallelismo si limita a fornire sotto-array ordinati, pronti per la fase merging.
Avendo a disposizione abbastanza processori questa procedura consente di ridurre la complessità di calcolo per la fase non merging a $\mathcal{O}(1)$, ottenendo una complessità totale di $\mathcal{O}(n)$. Ovviamente il vincolo per ottenere questo risultato è quello di avere un numero di processori pari a $n/2$.

4.2 Algebra Lineare

Riproponiamo di seguito i problemi che trattiamo nell'ambito dell'algebra lineare:

- prodotto tra matrici;
- calcolo del determinante;
- inversione di una matrice.

Il primo esempio presentato per le operazioni tra matrici è quello del prodotto. In questo caso osserviamo che dal punto di vista computazionale ogni elemento che compone la matrice risultante può essere calcolato in modo del tutto indipendente dagli altri: calcolare l'elemento in posizione (i, j) necessita esclusivamente dell'$i - esima$ riga della prima matrice e della $j - esima$ colonna della seconda. Possiamo quindi scrivere l'algoritmo parallelo in questo modo:

```
ProdottoMatriciParallelo(A : matrice MxN, B : matrice NxP)
    begin
        C = matrice di zeri MxP
        for i = 1,..,M do
            for j = 1,..,P do
                Esegui in parallelo la seguente operazione:
                    for k = 1,..,N do
                        C[i][j] = C[i,j] + A[i][k] + B[k][j]
                    done
            done
        done
        return C
    end
```

Avendo a disposizione $M * P$ processori otteniamo una complessità pari a $\mathcal{O}(N)$.
Il calcolo del determinante, come abbiamo visto, può essere implementato in modo inefficiente (metodo di Laplace) o efficiente (metodo di riduzione di Gauss). La parallelizzazione del primo caso è semplice: va effettuata sul calcolo dei vari determinanti per ogni minore ottenuto scorrendo una riga/colonna della matrice. Data una matrice A di dimensioni NxN possiamo quindi scrivere l'algoritmo in questo modo:

1. *Caso base: se $N = 1$ allora restituisci l'unico elemento presente in matrice.*

2. *Se $N > 1$, scegli una riga o una colonna i tale che $i \in [1, ..N]$ e calcola in parallelo i determinanti dei minori $A_{i,j}$ per ogni $j \in [1, .., N]$ e moltiplicali per $(-1)^{i+j}$.*

3. *Somma i risultati ottenuti.*

Anche in questo caso il parallelismo garantisce un buon guadagno di prestazioni. Concentriamoci ora sul caso efficiente: ricordiamo che si tratta di eseguire sottrazioni iterativamente tra le varie righe (prese due a due) per ottenere una matrice triangolare, dopodiché si moltiplicano i valori sulla diagonale. In questo caso l'unico modo per introdurre parallelismo è quello di eseguire in parallelo il prodotto finale: ricavare la matrice triangolare non può essere fatto in parallelo. Di fatto il calcolo va eseguito in larga parte in modo sequenziale, dunque può risultare poco conveniente introdurre risorse aggiuntive.
Vedremo nel capitolo successivo come questo si ripercuote concretamente in un'esecuzione parallela.
L'ultimo caso, l'operazione di inversione di matrici, è per certi versi legato al calcolo del determinante, ma fortunatamente qui possiamo introdurre un altro livello di parallelismo. Invertire una matrice NxN significa calcolare N^2 determinanti, ognuno in maniera indipendente dagli altri. Possiamo quindi scrivere, presa una matrice A di dimensioni NxN:

1. *Calcola la matrice dei cofattori, in cui ogni cofattore è calcolato in parallelo.*

2. *Moltiplica ogni elemento per $\frac{1}{det(A)}$.*

3. *La matrice A^{-1} è la matrice dei cofattori traslata.*

4.3 Grafi

Parte III

Programmazione Parallela e Haskell

Capitolo 5

Programmazione Parallela

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 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, 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; ovviamente il vincolo richiesto è di ottenere in parallelo lo stesso risultato che si otterrebbe nell'esecuzione sequenziale (in questo caso si parla anche di *parallelismo deterministico*).

5.1 Haskell

Tra tutti i linguaggi che consentono una gestione delle risorse interne delle architetture multiprocessore, 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.

5.1.1 Introduzione al linguaggio Haskell

Haskell è un linguaggio funzionale puro 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, le funzioni di ordine superiore e supporta il lambda calcolo, su cui si regge dal punto di vista matematico. Caratterizzato da una tipizzazione forte e statica, 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 al sito `https://www.schoolofhaskell.com/`.

5.1.2 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 utilizzato per questo lavoro sono 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 estensione *.hs* .

5.1.3 Perchè Haskell?

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

- 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 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 garantite 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).

5.1.4 Linguaggio funzionale: un linguaggio matematicamente puro

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 sequenziale.
Al contrario, Haskell è un linguaggio *funzionale*; appartiene, cioè, ad una categoria di linguaggi in cui ogni elemento è rappresentato esclusivamente 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:

$$f(x) = y$$

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

$$f : X \to Y$$

Tale concetto rappresenta la base dell'algebra e viene implementato elegantemente da tutti i linguaggi funzionali.
Nel caso di Haskell, la questione è molto simile, anche dal punto di vista semantico. Osserviamo un esempio semplice, come la funzione incremento:

```
module Esempio where

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

Analizziamo ora il codice riga per riga. In Haskell un programma è sostanzialmente una composizione di moduli (definiti appunto *modules*), una struttura a cui è associato un nome (*Esempio*) e definita da un insieme di valori, tipi di

dati e funzioni (in questo caso solo *inc*). Per utilizzare uno specifico modulo all'interno di un programma è sufficiente importarne il contenuto, mediante il comando `import`.
In questo caso osserviamo che *inc* viene prima definito dal punto di vista relazionale: è una funzione da elementi di tipo *Int* a elementi di tipo *Int*.
Il valore restituito da *inc* dipenderà dall'input (intero) in ingresso. In particolare (aperta la console ghci) otterremo:

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

Ovviamente, poiché abbiamo limitato la funzione a parametri interi, l'esecuzione di *inc* su valori in floating point solleverà un'eccezione (tipizzazione statica di Haskell).

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 che 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 funzione 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 del primo parametro, 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.

Astrazione

La capacità di creare funzioni di ordine superiore è uno strumento potente in Haskell, soprattutto se associato all'alto livello di astrazione di cui si compone il linguaggio Haskell. Si definisce astrazione la capacità di creare elementi il cui funzionamento interno è definito in maniera implicita e non diretta. Pensiamo alla funzione *myOperator* di prima: pur necessitando di un parametro avente una particolare struttura (*Int -> Int*) non è stato necessario fare alcuna assunzione sul calcolo che essa eseguiva. Potevamo creare qualsiasi funzione che rispettasse le caratteristiche richieste e il programma avrebbe agito di conseguenza in fase di esecuzione.
L'astrazione 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, facilita la progettazione software e, nel nostro caso, consente di implementare al meglio le funzioni che gestiscono il parallelismo.

Monade

Altro elemento cardine di Haskell è la **Monade**. Dal punto di vista matematico, preso un insieme X di definizione, una monade è una *un monoide di X nella categoria degli endofuntori di X* (def. Mac Lane[1]). Formalmente 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 binding; 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 informatica le monadi sono essenzialmente strutture che esprimono classi di computazioni concatenabili e risultano particolarmente utili nei linguaggi funzionali. Il meccanismo è il seguente: Haskell funziona mediante una tecnica di esecuzione detta valutazione pigra, o lazy evaluation, viste le sue caratteristiche poco performanti. In questo modo ogni elemento o funzione che compone i programmi vengono calcolati solo ed esclusivamente on-demand. Come conseguenza, da un lato una funzione può non venire calcolata se non viene richiesta, dall'altro un insieme di funzioni cui viene richiesta l'esecuzione viene trattato come un sistema di equazioni valutate simultaneamente. Le monadi aggirano questa caratteristica, consentendo una computazione più controllata e introducendo la sequenzialità nel linguaggio.
La classe che implementa le monadi in Haskell 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
```

Nel caso specifico le monadi devono rispettare le seguenti regole:

```
return a >>= k  =  k a                        <-- identità a sinistra
m >>= return  =  m                            <-- identità a destra
m >>= (\x -> k x >>= h)  =  (m >>= k) >>= h <-- associatività
```

Vediamo ora queste strutture in azione nella console GHCI:

```
##Definiamo per prima cosa la monade
Prelude> let m = return 3

##Ricaviamone il tipo associato
Prelude> :type m
m :: (Monad m, Num a) => m a
```

[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.

```
##Richiamare una monade ne restituisce il contenuto
Prelude> m
3

##Definiamo alcuni operatori per le monadi su interi
Prelude> let inc n = return (n + 1)
Prelude> let sub n = return (n - 1)


##Operiamo inc e sub in successione
Prelude> m >>= inc >>= sub >>=  inc
4
```

5.1.5 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", ciò consente di focalizzare l'attenzione 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. Tutte queste caratteritiche andranno definite solo in fase di esecuzione.
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'effettiva esecuzione su un sistema multiprocessore 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 f1 f2`, esprimibile anche come `f1 ‘par‘ f2`, si traduce sostanzialmente in: "l'esecuzione di f1 può essere parallelizzata con f2; il valore restituito è il risultato di f2". La funzione *par* quindi definisce *f1* come un thread che può essere eseguito in parallelo su un altro processore libero e restituisce 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.
Sembra una buona soluzione, tuttavia questo non basta per definire il parallelismo in ogni situazione: il fatto che il primo termine possa venire parallelizzato non implica che verrà effettivamente eseguito in parallelo su un altro processore, almeno dal punto di vista del compilatore. Osserviamo il seguente esempio per capirne il motivo:

```
parOp = f1 ‘par‘ (f1 + f2) where
            f1 = fib 20
            f2 = fib 20
```

In questo programma abbiamo definito una procedura che restituisce la somma di due funzioni, entrambe definite come l'elemento 20 della serie di Fibonacci (ipotizziamo di aver già creato una funzione che calcoli tale valore, in seguito ne vedremo un'implementazione effettiva). La prima componente di *par* definisce *f1* come thread parallelizzabile, dopodiché calcola $f1 + f2$. La funzione somma in Haskell funziona in modo da calcolare per primo l'elemento sinistro e poiché *f1* viene richiesto per primo[1] il sistema real time rimane sul processore che ha eseguito la richiesta, ne valuta il risultato e solo dopo aver terminato la valutazione di *f1* calcola *f2* e restituisce il risultato della somma. Di fatto in questo modo abbiamo vanificato gli sforzi per la parallelizzazione: l'esecuzione è rimasta fissa su un unico processore.
Una cattiva gestione di questa caratteristica può indurre facilmente a errori

[1] Ricordiamo che ogni elemento di Haskell viene valutato on demand.

concettuali, peraltro non facilmente tracciabili, poiché non sollevano eccezioni (eccezion fatta per altri errori di programmazione, l'esecuzione porterà sempre allo stesso risultato). Una soluzione immediata è di riscrivere il codice specificando che avvenga prima l'esecuzione di *f2*, nel seguente modo:

```
parOp2 = f1 `par` (f2 + f1) where
            f1 = fib 20
            f2 = fib 20
```

Il processore che ha eseguito la richiesta definirà *f1* sempre come un thread parallelizzabile, quindi calcolerà *f2*; un processore libero eseguirà in contemporanea il calcolo di *f1*. In questo caso abbiamo ottenuto il parallelismo richiesto.
Spesso però la gestione della parallelizzazione non è così immediata. Per questo motivo subentra la seconda funzione del package. 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: `f1 `seq` f2` restituisce il valore di `f2` solo dopo che entrambe le funzioni in input hanno terminato il loro ciclo di esecuzione. In questo modo ci assicuriamo che entrambe le funzioni abbiano terminato la loro esecuzione prima di restituire il risultato di *f2* le due funzioni possono essere eseguite in qualsiasi ordine senza problemi. In generale si dice che la funzione *seq* esegue il suo primo argomento in *weak head normal form*.
Analogamente la funzione `pseq f1 f2` si regge sullo stesso principio di base, ma con una particolarità in più: anziché attendere l'esecuzione di entrambe le funzioni, *pseq* vincola che il calcolo di *f2* avvenga solo dopo l'esecuzione di *(f1 + f2)*.
Nell'esempio di prima otteniamo lo stesso risultato parallelo scrivendo:

```
parOp3 = f1 `par` (f2 `pseq` (f1 + f2)) where
            f1 = fib 20
            f2 = fib 20
```

Osserviamo che, mentre *f1* viene sempre definito come parallelizzabile, viene forzata l'esecuzione di *f2* prima che venga effettivamente eseguita la somma. In questo modo il processore che ha eseguito *parOp3* calcola *f2*, mentre *f1* passa in esecuzione ad un altro processore libero.
I risultati ottenuti con *parOp2* e *parOp3* sono pressoché identici, in termini prestazionali: entrambi dimezzano il tempo di esecuzione rispetto a *parOp*.

6.1.1 Esempio: la serie di Fibonacci

Proviamo a mettere in pratica il parallelismo esplicito sulla serie di Fibonacci. Ricavare l'*n-esimo* elemento della serie in Haskell diventa:

```
fib :: Int -> Int

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

Per ora ignoriamo il fatto che si tratti di un'operazione altamente inefficiente per ottenere il valore richiesto. 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`, vengono creati due threads (o sparks come vengono chiamati in Parallel Haskell), ognuno associato al calcolo di Fibonacci su `n-1` e `n-2`, specificando che possono essere eseguiti parallelamente su due processori liberi (funzione *par*)e infine che la loro somma deve attendere che restituiscano un risultato (funzione *seq*).
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 ($n = \lfloor \log_2(46) \rfloor + 1 = 6$): il codice ha restituito il risultato dopo 260 secondi utilizzando un solo processore, quello parallelo in 58 secondi utilizzando, sulla stessa macchina, 8 processori. Ricaviamo quindi uno speedup di:

$$S(6,8) = \frac{260sec}{58sec} = 4.48 \simeq 450\%$$

e un'efficienza di:

$$E(6,8) = \frac{S(6,8)}{8} = 0.58 \simeq 58\%$$

6.2 Le strategie di calcolo

Un altro modo per parallelizzare l'esecuzione in Haskell è l'utilizzo delle **strategie di valutazione** o più semplicemente **strategie**. Si tratta di un meccanismo atto a definire parallelismo deterministico che suddivide il codice separando l'algoritmo in sé dal parallelismo. In questo modo, una volta definito un algoritmo, è possibile parallelizzarlo in modi diversi a seconda della strategia applicata.
Sostanzialmente una strategia è una funzione in *Eval*, una monade che prende come argomento un tipo generico e ne restituisce il valore.

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

L'idea alla base delle strategie è quella di prendere una struttura dati generica e di calcolarne le componenti mediante un'esecuzione sequenziale o parallela, a seconda di come è stata costruita la strategia.
Il vantaggio principale delle strategie consiste principalmente nell'essere componibili: trattando una monade, è possibile combinare sequenzialmente strategie semplici per gestire al meglio algoritmi più complessi (mediante l'operazione di binding).

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 identiche alle sopracitate **par** e **pseq** del package Control.Parallel (si veda il capitolo precedente). Tuttavia presentano differenze sostanziali: *rpar* crea il parallelismo, definendo il suo argomento in ingresso come una funzione da calcolare in parallelo (di fatto è una computazione priva di valutazione e se il suo argomento è già stato calcolato altrove la funzione non ha alcun effetto, vanificando il parallelismo); *rseq*, invece, forza il calcolo della funzione richiesta in (weak head normal form). Va fatto notare che entrambe le funzioni il

calcolo è eseguito in weak head normal form.
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
```

Chiariamo il loro utilizzo con un esempio. Immaginiamo di aver creato una funzione f e di aver poi deciso di eseguirla sfruttando una strategia s creata ad hoc. 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:

```
f `using` s
```

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 sequenziale già esistente semplicemente applicandovi una strategia per ottenere 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.

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] Saunders Mac Lane. *Categories for the Working Mathematician*. Springer Science e Business Media, 1971.