Parallelismo e Haskell

Jacopo Francesco Zemella

Università degli studi di Milano

jacopofrancesco.zemella@studenti.unimi.it

7 dicembre 2016

Algoritmi Sequenziali

Algoritmo: Sequenza finita di istruzioni interpretabili da un determinato agente, finalizzate a risolvere un problema

Calcolo Sequenziale: Insieme delle procedure in cui le istruzioni vengono eseguite in *sequenza*, una dopo l'altra. Nei sistemi informatici gli algoritmi sequenziali vengono eseguiti da una singola CPU: ad ogni istante t di tempo è in esecuzione una e una sola operazione

Complessità Sequenziale

La bontà di un algoritmo è legato al numero di operazioni che esegue per ottenere un risultato $\rightarrow$ **Complessità**

Complessità Asintotica: Stima asintotica della complessità di un algoritmo in funzione della dimensione n dell'input; esempi: n, $n \log n$, n^k, 2^n, $n!$

Caratteristiche degli Algoritmi

Efficienza: Un algoritmo si dice *efficiente* se la sua complessità asintotica è di ordine polinomiale ($\mathcal{O}(n^k),\ k \in \mathbb{N}$)

Ottimalità: Un algoritmo di complessità $f(n)$ si dice *ottimo* se ogni altro algoritmo che risolve lo stesso problema ha complessità pari *almeno* a $f(n)$

Algoritmi Paralleli

Calcolo Parallelo: Insieme delle procedure in cui parte delle istruzioni vengono eseguite su più processori contemporaneamente

Scopo della Parallelizzazione: Ottenere un compromesso conveniente tra prestazioni e costo delle risorse aggiunte

Come si presenta generico algoritmo parallelo:

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

Analisi degli Algoritmi Paralleli

Chiamiamo $\mathrm{T}_A(n, p)$ il tempo di esecuzione di un algoritmo parallelo A in funzione della dimensione dell'input n e del numero di processori p

- Speedup: $\mathrm{S}_A(n, p) = \dfrac{\mathrm{T}_A(n, 1)}{\mathrm{T}_A(n, p)}$
- Efficienza: $\mathrm{E}_A(n, p) = \dfrac{\mathrm{S}_A(p)}{p} = \dfrac{\mathrm{T}_A(n, 1)}{p\mathrm{T}_A(n, p)}$

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"

Analogamente il guadagno prestazionale ottenuto dal parallelismo è limitato dalla sua componente sequenziale

Se chiamiamo α la componente sequenziale del tempo di esecuzione otteniamo che:

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

$$S_A(n,p) = \frac{1}{\alpha} \text{ se } p \to \infty$$

Haskell

Caratteristiche di Haskell:

- Linguaggio funzionale
- Tipizzazione forte e statica
- Lazy Evaluation
- Alto livello di astrazioni
- Funzioni Higher-Order
- Pienamente avviato verso la programmazione parallela

Premesse nel parallelismo in Haskell

Haskell definisce i threads (o sparks) parallelizzabili in fase di compilazione

```
ghc -threaded foo.hs
```

In fase di esecuzione è inoltre possibile specificare il numero di processori da utilizzare mediante l'opzione `-N`

```
./foo +RTS -N2
```

Control.Paralell: par e pseq

Generiamo parallelismo a livello di codice mediante due funzioni del pacchetto Control.Paralell:

- *par*: definisce il parallelismo in modo esplicito; prende in ingresso due funzioni e definisce la prima come "parallelizzabile", ovvero eseguibile su un altro processore
- *pseq*: permette un controllo mirato sull'esecuzione delle varie funzioni che definiscono il programma

Perché controllare l'esecuzione?

Sia "`fib n`" la funzione che calcola l'elemento *n-esimo* della serie di Fibonacci. Eseguiamo il seguente programma:

```
pairFib = f1 + f2 where
     f1 = fib 36
     f2 = fib 36

pairFibPar = f1 `par` (f1 + f2) where
     f1 = fib 36
     f2 = fib 36
```

Tempo di esecuzione `pairFib` = 6.92 sec

Tempo di esecuzione `pairFibPar` = 6.92 sec

Abbiamo vainificato gli sforzi per il Parallelismo

Soluzione: pseq

```
pairFibPar2 =
    f1 `par` (f2 `pseq` (f1 + f2)) where
        f1 = fib 36
        f2 = fib 36
```

Tempo di esecuzione `pairFibPar2` = 3.48 sec

Control.Paralell.Strategies: strategie

Scopo = dividiamo l'algoritmo dalla componente parallela

Prendiamo un algoritmo "sequenziale" e vi applichiamo una strategia di calcolo che definisce implicitamente come deve essere parallelizzato

Possiamo sfruttare strategie già esistenti o crearne altre ex-novo adatte al nostro scopo

Esempio: Map mediante Strategies

```
-- Funzione Map sequenziale
map f [] = []
map f (x:xs) = (f x):(map f xs)

-- Funzione Map in parallelo
strat = parList rseq
parMap f xs = map f xs `using` strat
```

Grazie per l'attenzione!