Scalable and Cloud Programming Cheatsheet

Dynamic dispatch. Selezionare l'implementazione a runtime di un oggetto poliformo.

Trait vs Classe astratta	Gerarchia dei tipi in Scala
• Non hanno un costruttore. `trait Animal(name: string)` non è valido. • Quello selezionato è l'ultimo della lista di `with`. • `super` si riferisce alla classe finale che usa il trait. Si ha un concetto di legamento dinamico alla gerarchia della classe.	• `Nothing`. Sottotipo di tutti • `Any`. Supertipo di tutti • `AnyVal`. Supertipo di tutti i tipi base • `AnyRef`. Supertipo di tutte le collezioni

TypeBound. Vincolo per la definizione di tipi, upper `U >: T` o lower `U <: T`.

Covarianza e controvarianza.

```
trait Function[-T, +U] {
  def apply(input: T): U
}
```

```
trait List[+A] {
  def prepend[B >: A](elem: B): NonEmptyList[B] = NonEmptyList(elem, this)
}
```

Livelli di parallelismo:
- Bit: processa più bit consecutivamente.
- Istruzione: da stesso flusso di istruzioni da eseguire, ne esegue diverse in parallelo.
- Task: diversi flussi di istruzioni da eseguire, esegue flussi in parallelo. Girano in stessa macchina, condividono la memoria.

Parallelismo in JVM:
- Thread è l'unita di concorrenza. Condivide lo spazio di memoria con altri thread. Non può modificare lo stack di altri thread. Ha un suo stack di memoria e un program counter.

$$D(e) + \frac{W(e)}{P}$$

- $D(e)$ è la **depth complexity**: lunghezza attesa della più grande sequenza di operazioni elementari eseguita per input e.
- $W(e)$ è la **work complexity**: numero atteso di operazioni elementari per input e.
- $T(n)$ è **time complexity**: uguale a $W(p)$.
- P è numero massimo di thread paralleli.

Vi è un limite a ciò grazie alla **Legge di Amdhal**.

$$\text{Speedup} = \frac{1}{f + \frac{1-f}{P}} \qquad \text{ratio tra tempo di esecuzione sequenziale e parallelo}$$

- f è quello che *non* può essere speedup.
- $1 - f$ è quello che può essere speedup.

$$S(n) = \frac{T(1)}{T(n)} \qquad \text{(in contesto distribuito)}$$

- n numero di nodi. $T(n)$ tempo di esecuzione su n nodi. Idealmente $S(n) = n$.

Lo speedup può non aumentare a causa di continui accessi a memoria condivisa (diviene un bottleneck).

- **`.reduce` su Array**: convertire in albero bilanciato per ridurre la depth complexity, fare riduzione sull'albero. Funziona solo per operazioni associative.
- **`.foldLeft`** è sequenziale.
- **`.aggregate`** prende un metodo per operazioni paralleli piccoli (seqop) e un altro per ridurli in maniera singolare come aggregatore (combop).

Data Parallelism. Dati distribuiti su nodi che lavorano in parallelo; in ognuno si esegue un task parallelo. C'è uno scheduler che gestisce il parallelismo.
- I task paralleli hanno side-effect: le scritture per stesse locazioni necessitano di sync.

Partial failure. Crasha un nodo, non la completa computazione.

Network latency. Le interazioni inter-node sono più lente.

Pro/Cons di Hadoop. Fault tolerance buono per molti dati ma in ogni map-reduce scrive i dati su disco (performance lente).

Pro di Spark. Dataset immutabili, RDD e DataFrame. Dati intermediari in RAM. In caso di failure, le modifiche ai dataset sono calcolati di nuovo. Usa HOF.

- Ci si riferisce a tutti i dati di un RDD come se fosse una collezione unica, anche se distribuita fra i nodi.
- **Trasformazione**. Ritorna un RDD. Organizzati in stage dentro un DAG.
  - È **lazy** (valutate solo quando necessario). Ha il contro di essere re-computata se più azioni dipendono da esso.
  - Può essere **narrow**: eseguita localmente da ogni partitioner.
  - Può essere **wide**: inizia un nuovo stage. Fanno shuffling (cosa molto lenta perché scambio di dati inter-partition). Minimizzare shuffling con `reduceByKey` piuttosto di `groupByKey`.
  - La persistenza risolve il problema di laziness.
  - Valutato da **esecutori**.
- **Azione**. Ritorna qualcosa, che non è un RDD, al driver (processo con SparkContext). Triggera la valutazione di una trasformazione.
- **RDD KV**. Utile per computazione parallela su chiavi distinte e per riraggruppare i dati.
  - **Partizionatori**. Distribuzione delle coppie.
    - **Range**: divisione del dominio della chiave in range di stessa dimensione. **Hash**: si distribuisce in base alla chiave del RDD. Si applica un'hash function.
  - `mapValues, flatMapValues, filter, reduceByKey, groupByKey` preservano il partizionatore.

Strong scaling efficiency. Usa la legge di Amdhal. Aumentare nodi n ma con stessa dimensione del problema. Si aumenta n ma il lavoro totale si distribuisce fra i nodi.

$$E(n) = \frac{S(n)}{n} = \frac{T(1)}{nT(n)}$$

Tende sempre a 0 a causa di $1 - f$ nello speedup.

Weak scaling efficiency.

Usa la legge di Gustafson. Si basa sull'idea che la parte parallela scala linearmente all'aumentare delle risorse. La parte seriale non scala rispetto al problema.

Usato per risolvere problemi più grandi nello stesso tempo grazie all'aggiunta di potere computazionale. Si aumenta n ma il lavoro per nodo è uguale.

$$W(n) = \frac{T(1)}{T(n)} \qquad \text{avere } W(n) = 1 \text{ vuol dire usare al 100\% le risorse aggiunte}$$

$T(1)$ = tempo per completare 1 work unit con 1 nodo. $T(n)$ = tempo per completare n work unit con n nodi.

Se abbiamo 1 work unit per $W(m)$, per n work unit abbiamo $W(k \cdot m)$ t.c $n \cdot W(m)$.

Esempio. Per 1 w.u. abbiamo $W(m) = m^2$, per n abbiamo $W(k \cdot m) = n \cdot W(m)$ dunque $k^2 \cdot m^2 = n \cdot m^2$, dunque $k = n^{\frac{1}{2}}$.