Organizzazione della Lezione

- La legge di Amdahl
- Sincronizzazione di thread
  - metodi sincronizzati
  - lock intrinseci
  - accesso atomico
- Problemi di sincronizzazione
  - deadlock
  - starvation
- Conclusioni

Esempio del Pittore

- 5 amici che vogliono dipingere una casa con 5 stanze
- Se le 5 stanze sono uguali in dimensione (ed anche gli amici sono ugualmente capaci e produttivi!) allora finiscono in 1/5 del tempo che ci avrebbe impiegato una sola persona
  - lo speedup ottenuto è 5, pari al numero di amici
- Se 1 stanza è grande il doppio, però, il risultato è diverso
- Il tempo per fare la stanza grande "domina" il tempo delle altre
  - naturalmente non consideriamo la complicazione di aiutare il poveretto cui è toccata la stanza grande, per l'overhead del coordinamento necessario

Legge di Amdahl: Speedup

- Lo speedup `S` di un programma `X` è il rapporto tra il tempo impiegato da un processore per eseguire `X` rispetto al tempo impiegato da n processori per eseguire `X`
- Sia `p` la parte del programma `X` che è possibile parallelizzare
  - con `n` processori la parte parallela prende tempo `p/n` mentre la parte sequenziale prende tempo `(1-p)`

Lo speedup che si ottiene eseguendo il programma X su n processori, dove p è la parte di X che si può parallelizzare è: $S(n) = \frac{1}{(1-p)+\frac{p}{n}}$

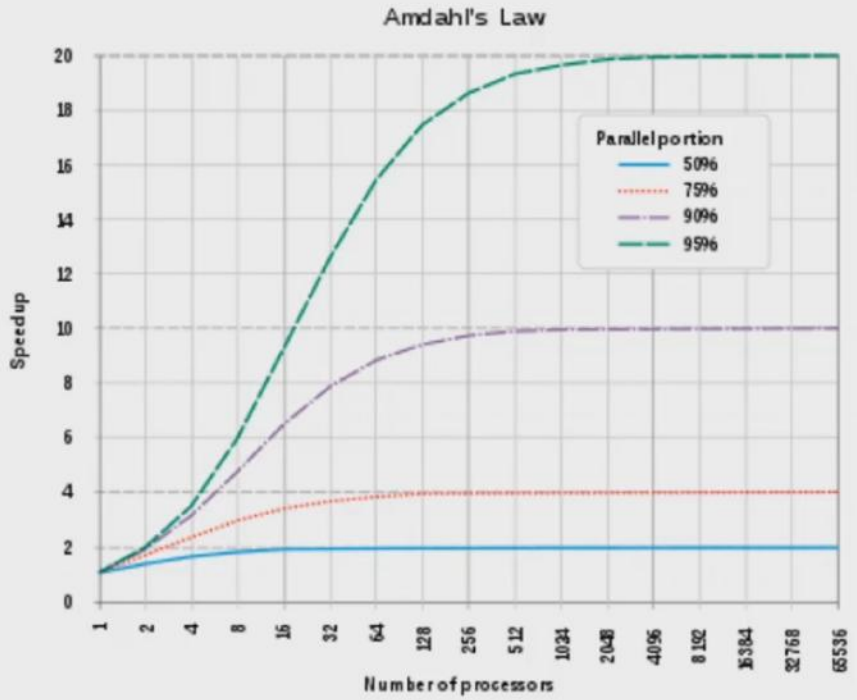
Legge di Amdahl

- La legge di Amdahl viene usata per predire l'aumento massimo teorico di velocità che si ottiene usando più processori

Lo speedup che si ottiene eseguendo il programma X su n processori, dove p è la parte di X che si può parallelizzare è:

$$S(n) = \frac{1}{(1-p) + \frac{p}{n}}$$

Legge di Amdahl: Esempio di speedup teorico



Lo speedup e limitatao dalla parte seriale del programma. Ad esempio, se il 95% del programma si puo paralelizzare, allora lo speedup massimo teorico e' 20x.

Esempi di Applicazione della Legge di Amdhal

- I 5 amici pittori: se le stanze sono 5, di cui 4 valgono 1, mentre 1stanza è grande il doppio (2*1=2), allora lo speedup che si ottiene è:

$$S(n) = \frac{1}{(1-p)+\frac{p}{n}} = \frac{1}{\left(1-\frac{5}{6}\right)+\frac{1}{6}} = 3$$

Legge di Amdahl per Multicore e Multiprocessori

- La legge di Amdahl ci dice che la parte sequenziale del programma rallenta significativamente qualsiasi speedup che possiamo pensare di ottenere
- Quindi, per velocizzare un programma non basta investire sull'hardware (più processori, più veloci, ..)
  - ma è assolutamente necessario e molto più cost-effective impegnarsi a rendere la parte parallela predominante rispetto alla parte sequenziale
- (fortunatamente per noi informatici!)

Sincronizzazione: Comunicazione tra Thread

- Comunicazione tra thread: tipicamente condividendo accesso a:
  - campi (tipi primitivi)
  - campi che contengono riferimenti ad oggetti
- Comunicazione molto efficiente
  - rispetto all'usare la rete
- Possibili due tipi di errori:
  - interferenza di thread
  - inconsistenza della memoria
- Per risolvere questi problemi, necessaria la sincronizzazione
  - che a sua volta genera problemi di contesa: quando più thread cercano di accedere alla stessa risorsa simultaneamente (deadlock e livelock)

Sincronizzazione: Metodi Sincronizzati

- I metodi sincronizzati (`synchronized`) sono un costrutto del linguaggio Java, che permette di risolvere semplicemente gli errori di concorrenza
  - al costo di inefficienza
- Per rendere un metodo sincronizzato, basta aggiungere `synchronized` alla sua dichiarazione:

```
public class SynchronizedCounter {
  private int c = 0;

  public synchronized void increment() {
    c++;
  }

  public synchronized void decrement() {
    c--;
  }

  public synchronized int value() {
    return c;
  }
}
```

Cosa comporta un metodo sincronizzato?

- Non è possibile che due esecuzioni dello stesso metodo sullo stesso oggetto siano interfogliate
- Quando un thread esegue un metodo sincronizzato per un oggetto, gli altri thread che invocano metodi sincronizzati dello stesso oggetto sono sospesi fino a quando il primo thread non ha finito
- Quando un thread esce da un metodo sincronizzato, allora si stabilisce una relazione happens-before con tutte le successive invocazioni dello stesso metodo sullo stesso oggetto
  - i cambi allo stato, effettuati dal thread appena uscito sono visibili a tutti i thread
- I costruttori non possono essere sincronizzati
  - solo il thread che crea dovrebbe avere accesso all'oggetto in costruzione

Lock Intrinseci

- Un lock intrinseco (o monitor lock) è una entità associata ad ogni oggetto
  - sia accesso esclusivo sia accesso consistente (relazione *happens-before*)
- Un thread deve
  - acquisire il lock di un oggetto
  - rilasciarlo quando ha terminato
- Quando il lock che possedeva viene rilasciato, viene stabilita la relazione happens-before
- Quando un thread esegue un metodo sincronizzato di un oggetto ne acquisisce il lock, e lo rilascia al termine (anche se c'è una eccezione)

Lock Intrinseci: Synchronized Statement

- Specificando di quale oggetto si usa il lock:

```
public void addName(String name) {
  synchronized(this) {
     lastName = name;
     nameCount++;
  }
  nameList.add(name);
}
```

- In questa maniera, si sincronizzano gli accessi solo durante la modifica, ma poi si provvede in maniera concorrente all'inserimento in lista

Sincronizzazione a Grana Fine

1. Due variabili
2. Dichiarazione di due lock
3. Accesso a `c1` con il `lock1`
4. Accesso a `c2` con il `lock2`
5. Con `synchronized` sul metodo si sequenzializza tutto (Amdahl!)
6. Con `synchronized` su `this` non sarebbero indipendenti!

```
public class MsLunch {
  private long c1 = 0;
  private long c2 = 0;
  private Object lock1 = new Object();
  private Object lock2 = new Object();

  public void inc1() {
    //molto codice
    synchronized(lock1) {
      c1++;
    }
    //molto codice
  }

  public void inc2() {
    //molto codice
    synchronized(lock2) {
      c2++;
    }
    //molto codice
  }
}
```

Sincronizzazione con Azioni Atomiche

- Azioni che non sono interrompibili e si completano (del tutto) oppure per niente
- Si possono specificare azioni atomiche in Java per:
  - read e write su variabili di riferimento e su tipi primitivi
  - read e write su tutte le variabili `volatile`
- Write a variabili `volatile` stabiliscono una relazione happens-before con le letture successive
- Tipi di dato definiti in `java.util.concurrent.atomic`

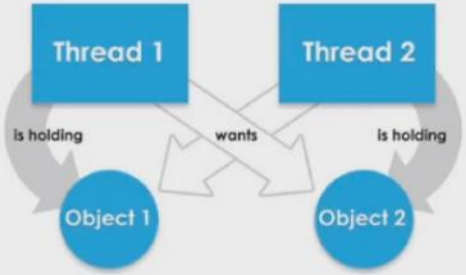
Esempio di Atomic

1. Package Classe
2. Variabile istanza
3. Metodo non sincronizzato
4. Uso di metodi atomici
5. Uso di metodi atomici
6. Lettura

```
import java.util.concurrent.atomic.AtomicInteger;

class AtomicCounter {
  private AtomicInteger c = new AtomicInteger(0);

  public void increment() {
    c.incrementAndGet();
  }

  public void decrement() {
    c.decrementAndGet();
  }

  public int value() {  return
    c.get();
  }
}
```

Problemi di Sincronizzazione: Deadlock

- Cosa è un deadlock?
- Quando due thread sono bloccati, ognuno in attesa dell'altro



- Ad esempio
  - un thread A ha il lock di una risorsa X e cerca di ottenere il lock di Y . . .
  - . . . mentre un thread B ha il lock della risorsa Y e cerca di ottenere il lock di X
- In questa maniera, il nostro programma concorrente si blocca e non c'è maniera di sbloccarlo

Esempio di Deadlock

- Alphonse & Gaston (COMICS DI INIZIO '900)

- Alfonso e Gastone sono due amici molto cortesi
- Rigida regola di cortesia:
  - quando ti inchini ad un amico, devi rimanere inchinato finchè il tuo amico non ha una possibilità di restituire l'inchino
- Ma se i metodi `bow` (inchino) vengono invocati assieme...
  - entrambi i thread si bloccheranno invocando `bowBack()`
  - nessuno dei due blocchi avrà fine, perchè ogni thread aspetta l'altro

Esempio di Deadlock: Gli inchini di Alphonse e Gaston (1)

1. Classe interna
2. Campo
3. Costruttore
4. "Inchino" . . .
5. . . . con risposta
6. La risposta

```
public class Deadlock {
 static class Friend {
  private final String name;

  public Friend(String name) {
   this.name = name;
  }

  public String getName() {
   return this.name;
  }

  public synchronized void bow(Friend bower) {
   System.out.format("%s:%s"+"has bowed to
   me!%n",this.name, bower.getName());
   bower.bowBack(this);
  }

  public synchronized void bowBack(Friend bower){
   System.out.format("%s:%s"+"has bowed back to
   me!%n",this.name, bower.getName());
  }
}
//...
```

Esempio di Deadlock: Gli inchini di Alphonse e Gaston (2)

1. Si creano Alphonse e Gaston
2. Classe anonima di tipo `Runnable` passata al costruttore di `Thread`
3. con il metodo da eseguire...
4. ...e si lancia

```
//...

public static void main(String[] args) {

  final Friend alphonse = new Friend("Alphonse");
  final Friend gaston = new Friend("Gaston");

  new Thread(new Runnable() {
    public void run() { alphonse.bow(gaston); }
  }).start();

  new Thread(new Runnable() {
    public void run() { gaston.bow(alphonse); }
  }).start();

 }//end main

}//end class
```

- Output:

```
Alphonse: Gaston has bowed to me!

Gaston: Alphonse has bowed to me!
```

- Nessun bows back!

Gli Inchini di Alphonse e Gaston

- Alphonse's thread

```
A: alphonse.bow(gaston) - acquires alphonse's lock
A: gaston.bowBack(alphonse) - acquires gaston's lock
A: both methods return, thus releasing both locks
```

- Gaston's thread

```
G: gaston.bow(alphonse) - acquires gaston's lock
G: alphonse.bowBack(gaston) - acquires alphonse's lock
G: both methods return, thus releasing both locks
```

- Possibile deadlock

```
A: alphonse.bow(gaston) - acquires alphonse's lock
G: gaston.bow(alphonse) - acquires gaston's lock
G: attempts to call alphonse.bowBack(gaston), but blocks waiting on alphonse's lock
A: attempts to call gaston.bowBack(alphonse), but blocks waiting on gaston's lock to
```

Perche il Deadlock?

- Riscriviamo i metodi `synchronized` in questa maniera

1. Lock esplicito sul lock dell'oggetto
   - idem
2. A questo punto Alphonse acquisisce il suo lock e cerca di acquisire quello di Gaston
   - che fa lo stesso: prima il suo e poi quello di Alphonse

```
static class Friend {
 //...
 public void bow(Friend bower) {
  synchronized(this) {
     System.out.format("%s:%s has bowed to
        me!%n",this.name, bower.getName());
     bower.bowBack(this);
  }
 }
 public void bowBack(Friend bower) {
  synchronized(this) {
     System.out.format("%s:%s has bowed back to
       me!%n",this.name, bower.getName());
  }
 }
}
```

- Funziona?

Problemi di Sincronizzazione: Starvation

- Quando un thread non riesce ad acquisire accesso ad una risorsa condivisa...
  - ...in maniera da non riuscire a fare progresso
  - risorsa è indisponibile per thread "ingordi"
- Esempio: un metodo sincronizzato che impiega molto tempo
  - se invocato spesso, altri thread possono essere prevenuti dall'accesso
- Arbitrarietà dello scheduler
  - attenzione: priorità dei thread nella JVM dipendente dal mapping effettuato sui thread dal S.O.!
  - priorità 3 e 4 in JVM possono essere mappate su stessa priorità del S.O.

Problemi di Sincronizzazione: Livelock

- Un thread A può reagire ad azioni di un altro thread B. . .
- . . . che reagisce con una risposta verso A
- I due thread non sono bloccati (non è un deadlock!) ma sono occupati a rispondere alle azioni dell'altro
- Anche se sono in esecuzione, non c'è progresso!
- Un esempio: due persone che si incontrano in un corridoio stretto, sullo stesso lato
  - attitudine belligerante: aspettare che l'altro si sposti
  - attitudine garbata: spostarsi di lato
- 2 belligeranti: deadlock!
- 2 garbati: livelock!

Conclusioni

- La legge di Amdahl
- Sincronizzazione di thread
  - metodi sincronizzati
  - lock intrinseci
  - accesso atomico
- Problemi di sincronizzazione
  - deadlock
  - starvation
- Conclusioni