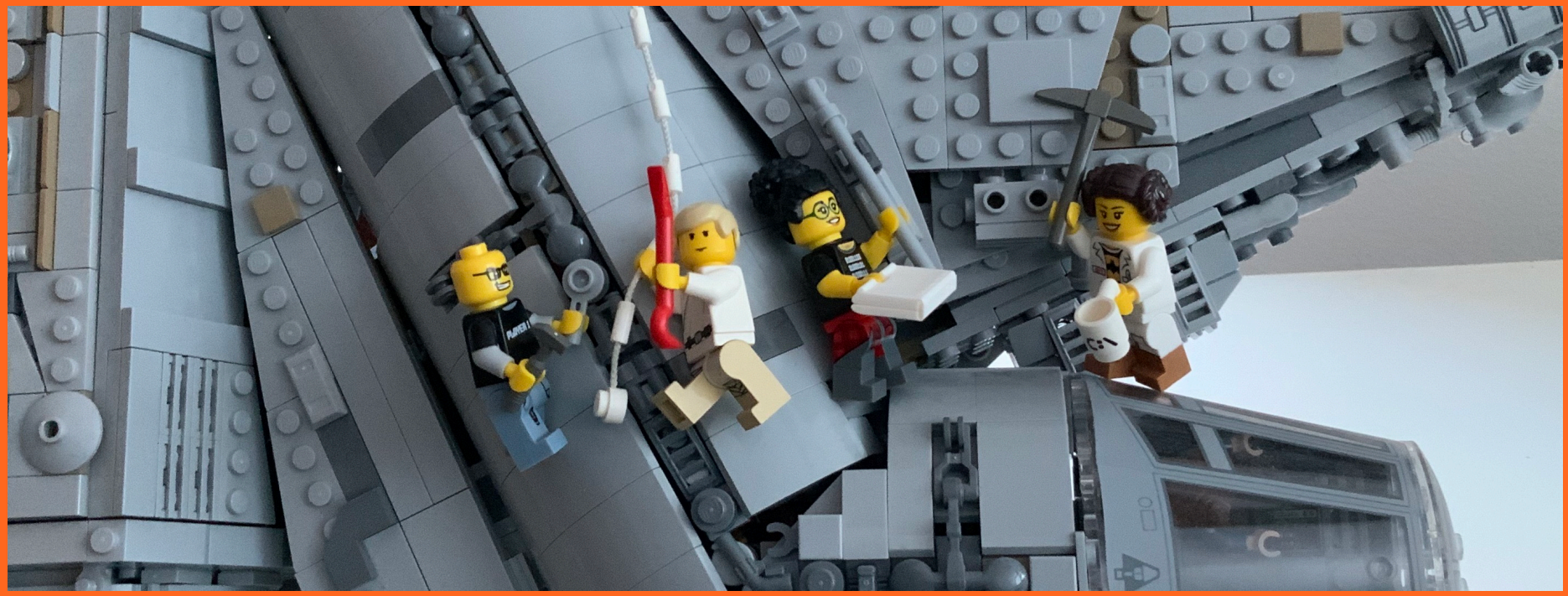
CS-A1120 Ohjelmointi 2



Aalto-yliopisto
Perustieteiden
korkeakoulu

Luento 9
kevät 2026

Sanna Suoranta (suomeksi, T1 klo 14) ja
Vesa Hirvisalo /Lukas Ahrenberg
(englanniksi, T1 klo 10)

4.5.2026 https://presemo.aalto.fi/o2fi2026

Tänään O2:ssa

Moduuli III: Uudempia juttuja

Aalto-yliopisto
Perustieteiden
korkeakoulu

https://presemo.aalto.fi/o2fi2026

Tänään O2:ssa

Samanaikaisuus ja rinnakkaisuus (concurrency and parallelism)

Aalto-yliopisto
Perustieteiden korkeakoulu

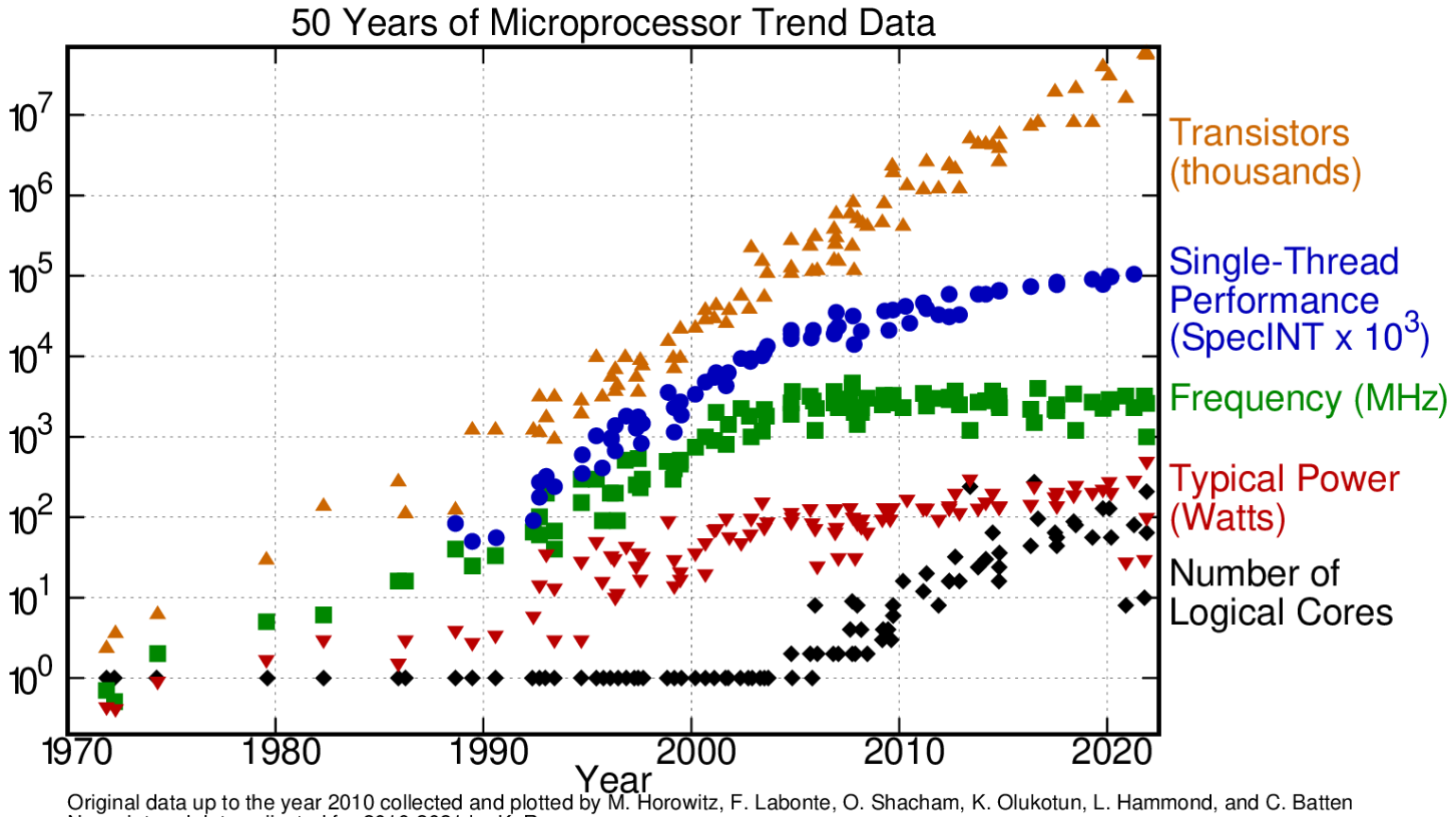
https://presemo.aalto.fi/o2fi2026

Kierroksen 9 oppimistavoitteet

- **Tämän viikon jälkeen,**
  - Osaat kertoa esimerkkejä rinnakkaisesta (parallel) ja samanaikaisesta (concurrent) laskennasta
  - Olet tietoinen säikeistä (computing threads) ja kuinka niitä voi luoda Scalassa
  - Olet tutustunut asynkroniseen laskentaan ja synkronointiin
  - Osaat käyttää korkean tason abstraktioita Future ja Promises Scalassa
  - Ostaa selittää puhtaat funktiot
  - Olet tietoinen riippuvuuksista (dependency) ja riippumattomuudesta (independency) rinnakkaisessa laskennasta sekä vähimmäiskestoajasta (minimum makespan)

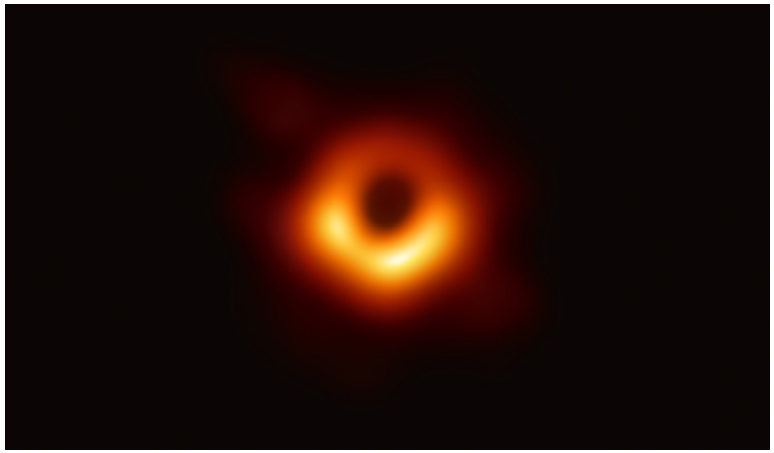
Mooren ‘laki’

- **Transistoreiden määrä piirissa tuplaantui joka 1,5-2 vuosi**



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2021 by K. Rupp

Entä jos yksi CPU ei riitä? Miten prosessoida?



- **Event Horizon—teleskooppi**
  - Radioteleskooppeja ympäri maailman
  - Jokainen generoi 350 TB/päivä *

* https://www.supermicro.com/white_paper/white_paper_Black_Hole_Event_Horizon_Imaging.pdf
* https://eventhorizontelescope.org/faq/how-much-data-recorded-during-observation-and-how-it-transferred-central-processing

- **Common crawl**: https://commoncrawl.org/**
  - Julkinen verkon läpikäyntiin perustuva data (web crawler)
  - Uusin kokoelma (crawl archive): 2,74 miljardia websivua (455 TiB)
  - Pääsy Amazon Open Data –rekisterin kautta***

** https://commoncrawl.org/
https://commoncrawl.org/blog/march-2025-crawl-archive-now-available
*** https://registry.opendata.aws/commoncrawl/

- **Amazon Snowmobile data truck**
  - 100 PB

https://aws.amazon.com/snowball/

Skaalautuva hajautettu laskentainfrastruktuuri

Laskentaklusterit (compute clusters)

- **Verkotettu joukko tietokoneita/noodeja (nodes)**
- **Samanlaisia koneita (homogeneous)**
- **Koneet lähellä toisiaan eli samassa räkissä (rack) / hallissa**
- **Jokainen noodi työskentelee saman tehtävän kanssa**

Skaalautuva hajautettu laskentainfrastruktuuri

Laskentaverkko (computing grids)

- **Verkotettu joukko tietokoneita/noodeja (nodes)**
- **Erilaisia koneita (heterogeneous)**
- **Maantieteellisesti hajautettu**
- **Noodit yleensä työskentelevät eri tehtävien kanssa**

Skaalautuva hajautettu laskentainfrastruktuuri

Talonkokoiset tietokoneet (Warehouse-scale computers)

- Erittäin homogeenisia ja hierarkisia
- Iso rakennus, jossa jäähdytys ja sähkönsyöttö
- Erikoistunut laitteisto ja ohjelmisto
- Isojen yritysten operoimia
- Palvelut on virtualisoitu

Skaalautuva hajautettu laskentainfrastruktuuri

Datakeskus (Data center)

- **Esim. Internet-palveluiden pyörittämiseen (hosting) – hakukoneet, videoiden suoratoistoon (streaming)**
- **Iso rakennus, jossa jäähdytys ja sähkönsyöttö**
- **Erikoistunut laitteisto ja ohjelmisto**
- **Erilaisia omistusmalleja**
- **Palvelut on virtualisoitu**
- **Paljon vaihtelevuutta (ja historiaa)**

Peräkkäinen (sequential) ja rinnakkainen (parallel)

- **Tähän asti: laskenta yksinkertaistettuna ilmiönä, jossa yksi prosessori (ydin) suorittaa yhtä konekielistä käskyä kerrallaan, peräkkäin (sequentially), eritettynä ympäristöstään**
  - Todellisuudessa, peräkkäinen suoritus on raaka yksinkertaistus
- **Semantiikka eli mitä ohjelma tarkoittaa**
  - Peräkkäinen (sequential): yhden asian saattaminen kerrallaan loppuun
  - Samanaikainen (concurrent): useamman tehtävän edistäminen yhtäaikaisesti (ei välttämättä samaan aikaan kaikkea tehden)
- **Suorittaminen (execution) eli miten laskenta tehdään**
  - Sarjallinen (serial): Yhden tehtävän tekeminen aikayksikön aikana
  - Rinnakkainen (parallel): Useamman tehtävän tekeminen yhtäaikaisesti

Raptor Lake CPU -arkkitehtuuri

- **Intel Core i9-14900F –prosessori**
- **24 ydintä, kaksi eri arkkitehtuuria, yhdistettynä samalle piirille**
- **Heterogeeninen arkkitehtuuri**
  - 8 suorituskykyistä: Raptor Core –arkkitehtuuri
  - 16 energiatehokasta: Gracemont –arkkitehtuuri
- **Prosessorien käskynsyöttö (processor pipelines) tukee vektorisoituja käskyjä**
  - Single Instruction Multiple Data (SIMD)

- https://www.intel.com/content/www/us/en/products/sku/236853/intel-core-i9-processor-14900f-36m-cache-up-to-5-80-ghz/specifications.html
- https://en.wikipedia.org/wiki/Golden_Cove#Raptor_Cove
- https://en.wikipedia.org/wiki/Gracemont_(microarchitecture)
- https://en.wikipedia.org/wiki/Single_instruction,_multiple_data

Lumi-C –supertietokone Kajaanissa

- **Lumi-C osio (pelkkä CPU)**
  - 1376 noodia, joissa kussakin
  - kaksi AMD EPYC 7763 -prosessoria, joissa 64 ydintä
  - CPU-ytimet tukevat AVX2 256-bittisiä vektorikäskyjä
  - Eli 16 kappaletta 16-bittisiä liukulukunumeroita rinnakkain
  - Yhteensä $5.63*10^{15}$ liukulukuoperaatiota sekunnissa (Floating point operations per second, FLOPS)

- **Erikseen GPU-pohjainen (Grapics Processing Unit) yksikkö, jossa 2978 noodia**
  - AMD Trento CPU ja neljä MI250X GPUta

https://www.lumi-supercomputer.eu/
https://en.wikipedia.org/wiki/LUMI

Rinnakkaisuus (parallelism) ja samanaikaisuus (concurrency)

- **Käytännössä, laskenta on ilmiö, jossa useita useita samanaikaisia (concurrent) ohjelmia suoritetaan rinnakkain (parallel), mutta toistensa kanssa vuorovaikutuksessa**
  - Mitä vähemmän vuorovaikutusta, sitä helpompaa suoritus on rinnakkaistaa
- **Hyötyjä:**
  - Kiihdytys (parallel speedup): jotkut tehtävät ovat luonnostaan rinnakkaistettavia: tehtävä voidaan jakaa itsenäisiin osiin
  - Läpimenoaika (throughput): saman tehtävän rinnakkaistus tarkoittaa että voimme käsitellä enemmän dataa
  - Saatavuus (availability): aikataulutus (scheduling) ja kuormantasaus (load balancing) niin, että järjestelmän osat ovat saatavissa tarvittaessa

Säikeet (threads)

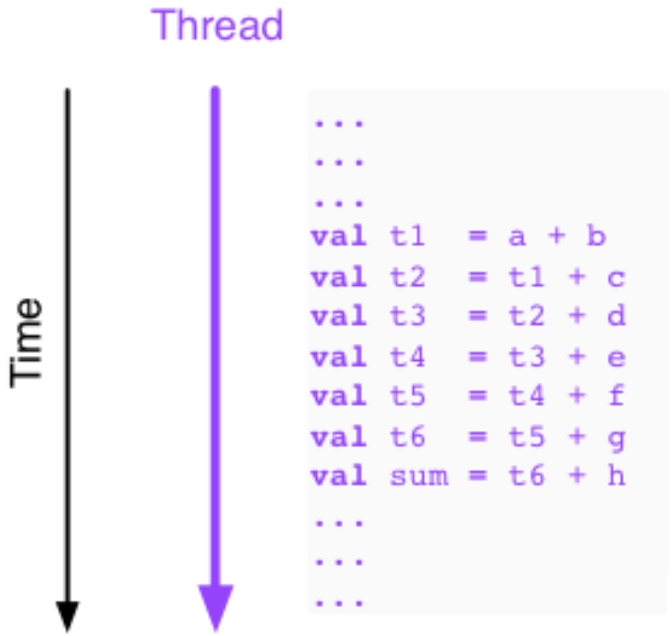
Aalto-yliopisto
Perustieteiden korkeakoulu

https://presemo.aalto.fi/o2fi2026

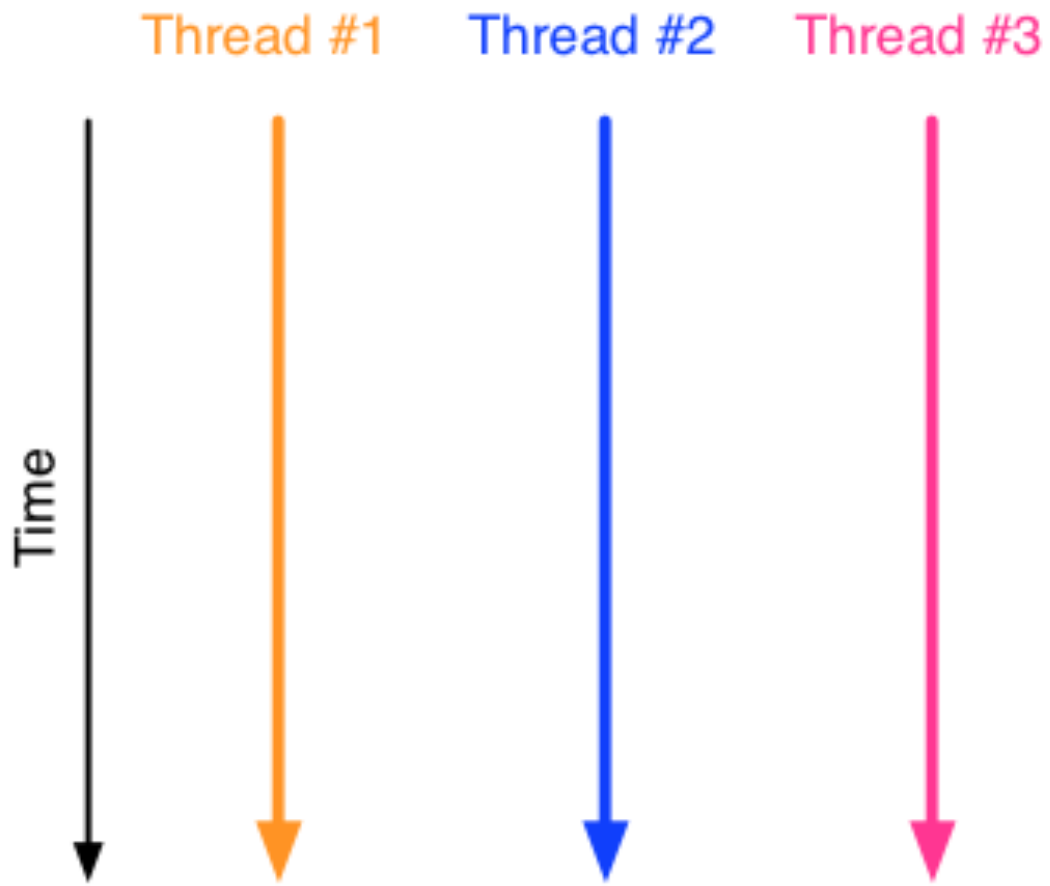
Yhden säikeen (thread) suoritus

- **Säie (thread) on itsenäinen sekvenssi käskyjä**
- **Ohjelma, joka suoritetaan yhdessä säikeessä, on peräkkäinen ohjelma (sequential program)**
  - Kuten tähän asti tällä kurssilla

```
val t1 = a + b
val t2 = t1 + c
val t3 = t2 + d
val t4 = t3 + e
val t5 = t4 + f
val t6 = t5 + g
val sum = t6 + h
```

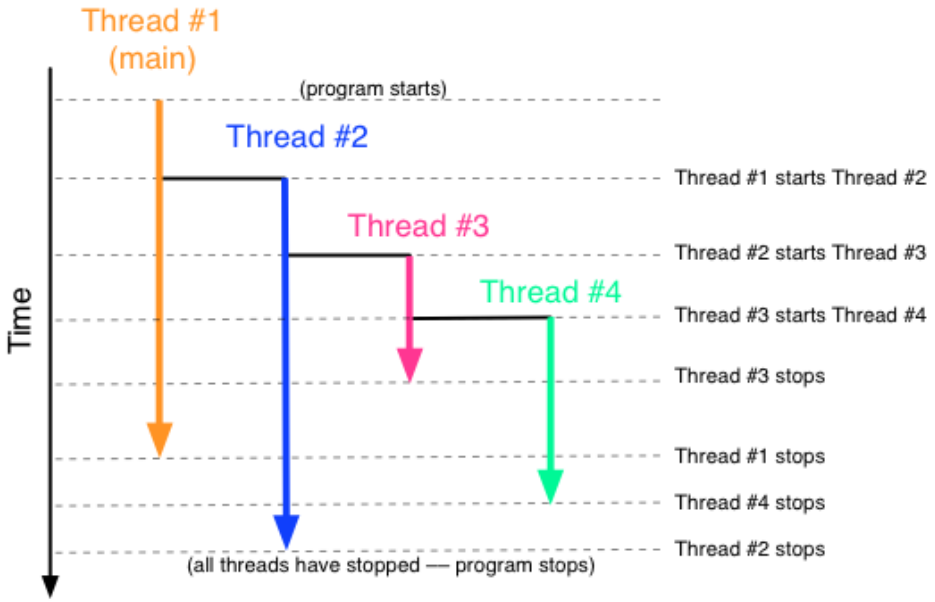


Useita säikeitä



Useita säikeitä

- **Jokaisella säikellä on oma sisäinen tila**
  - Ohjelmalaskurit, rekisterit,…
- **Lisäksi säikeillä on jaettu tila**
  - Keskusmuisti, käyttöjärjestelmän jakamat resurssit,…
- **Jokainen säie suoritetaan asynkronisesti (asynchronous)**
  - Käytännössä täysin itsenäisesti (fully independent) verrattuna muiden säikeiden tekemisiin
  - Paitsi jos ohjelmoija erikseen nimenomaisesti synkronoi (explicitly synchronize) säikeet



Säikeet ja asynkroninen suoritus Scalassa

- **`Runnable`-luokan perivä objekti voidaan ajaa asynkronisesti säikeenä `Thread` Scalassa**
  - Ajettava koodi laitetaan luokan `run`-metodiin

```
class HelloFrom(city:String)
    extends Runnable:
  def run() : Unit =
    print(s"Hello from $city")
end HelloFrom
```

```
val h1 = new HelloFrom("Rome")
val h2 = new HelloFrom("New Delhi")
val h3 = new HelloFrom("Canberra")

val t1 = new Thread(h1)
val t2 = new Thread(h2)
val t3 = new Thread(h3)
t1.start();t2.start();t3.start()
```

Säikeet ja asynkroninen suoritus Scalassa

- **`Runnable`-luokan perivä objekti voidaan ajaa asynkronisesti säikeenä `Thread` Scalassa**
  - Ajettava koodi laitetaan luokan `run`-metodiin

```
class HelloFrom(city:String)
    extends Runnable:
  def run() : Unit =
    print(s"Hello from $city")
end HelloFrom
```

```
val h1 = new HelloFrom("Rome")
val h2 = new HelloFrom("New Delhi")
val h3 = new HelloFrom("Canberra")

val t1 = new Thread(h1)
val t2 = new Thread(h2)
val t3 = new Thread(h3)
t1.start();t2.start();t3.start()
```

- **Mitä ohjelma tulostaa tässä tapauksessa?**

https://presemo.aalto.fi/o2fi2026

Säikeet ja asynkroninen suoritus Scalassa

- **`Runnable`-luokan perivä objekti voidaan ajaa asynkronisesti säikeenä `Thread` Scalassa**
  - Ajettava koodi laitetaan luokan `run`-metodiin

```
class HelloFrom(city:String)
    extends Runnable:
  def run() : Unit =
    print(s"Hello from $city")
end HelloFrom
```

```
val h1 = new HelloFrom("Rome")
val h2 = new HelloFrom("New Delhi")
val h3 = new HelloFrom("Canberra")

val t1 = new Thread(h1)
val t2 = new Thread(h2)
val t3 = new Thread(h3)
t1.start();t2.start();t3.start()
```

- **Emme tiedä mitä tulostuu!**
- **Ajojärjestys voi vaihdella!**

```
Hello from RomeHello from New DelhiHello from Canberra
Hello from RomeHello from CanberraHello from New Delhi
```


(Huom. Ensimmäinen näistä on vähän yleisempi, kun t1 pääsee alkuun ennen kuin t2 tai t3, eli se saattaa ehtiä varata konsolin ensimmäisenä.)

Säikeet ja asynkroninen suoritus Scalassa

- **Selkeämpää, jos teemme vielä näin**

```
class HelloFrom2(city:String)
    extends Runnable:
  def run() : Unit =
    val hs = s"Hello from $city"
    for(c <- hs) do
      print(c)
  end run
end HelloFrom2
```

```
val h1 = new HelloFrom2("Rome")
val h2 = new HelloFrom2("New Delhi")
val h3 = new HelloFrom2("Canberra")

val t1 = new Thread(h1)
val t2 = new Thread(h2)
val t3 = new Thread(h3)
t1.start();t2.start();t3.start()
```

Säikeet ja asynkroninen suoritus Scalassa

- **Selkeämpää, jos teemme vielä näin**

```
class HelloFrom2(city:String)
    extends Runnable:
  def run() : Unit =
    val hs = s"Hello from $city"
    for(c <- hs) do
      print(c)
  end run
end HelloFrom2
```

```
val h1 = new HelloFrom2("Rome")
val h2 = new HelloFrom2("New Delhi")
val h3 = new HelloFrom2("Canberra")

val t1 = new Thread(h1)
val t2 = new Thread(h2)
val t3 = new Thread(h3)
t1.start();t2.start();t3.start()
```

- **Kolme ajokertaa (kun tulostetaan kirjain kerrallaan):**

```
HHello from RomeHello from New Delhiello from Canberra
HellHo frello from Neow DeHlhim Romeello from Canberra
Hello from NewHHee Dlllo lelofrom hifrom Ro meCanberra
```

Synkronointi (synchronisation) ja riippuvuudet (dependencies)

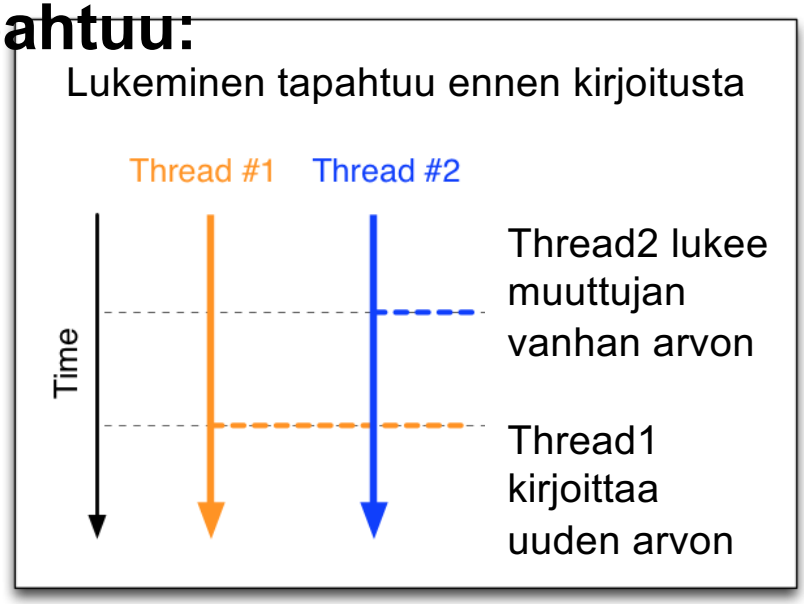
- **Säikeet suoritetaan asynkronisesti, jollei ohjelmoija synkroi niitä**
- **Säikeen käynnistys: `t.start()`**
  - Säikeen käynnistys ei ole itsenäinen verraten siihen säikeeseen, joka käynnistää sen, mutta suoritus on asynkronista käynnistyksen jälkeen
- **Säikeiden liittäminen (odottaminen): `t.join()`**
  - Liittäminen (joining) aiheuttaa sen, että säikeen kutsu keskeytetään (suspended, blocked) kunnes liitetyn säikeen `t` suoritus on loppunut. Sitten voidaan jatkaa (resume) suoritusta.

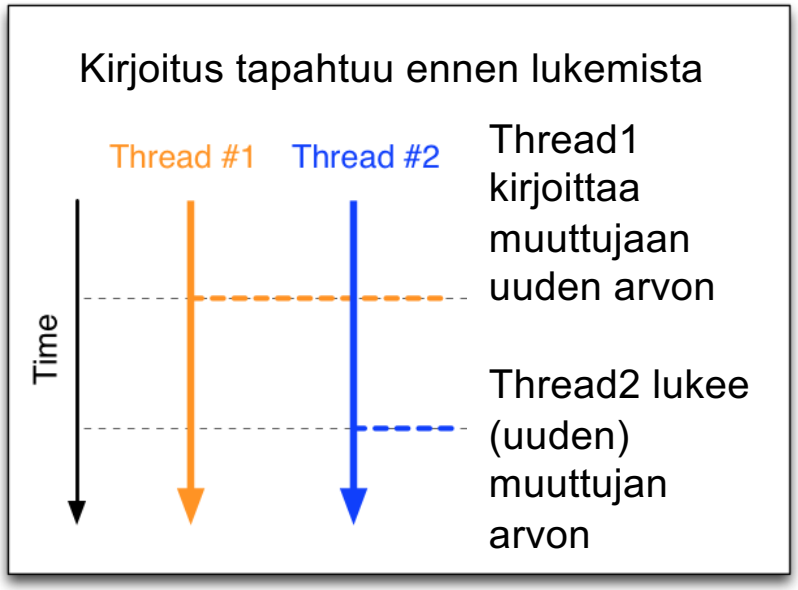
```
t1.start(); t1.join(); t2.start(); t2.join(); t3.start()

Hello from RomeHello from New DelhiHello from Canberra
```

Synkronointi

- **Ilman säikeiden välistä synkronointia emme tiedä resurssin käyttöjärjestystä! Ohjelmoijan vastuulla on tarkistaa, että**
  - Ohjelmaan ei jää lukkiutumia (deadlock), jossa säikeet odottavat toisiaan loputtomasti
  - Tai ei tapahdu nälkiintymistä (starvation) eli joku jää ilman resurssia
- **Esim. Jos kaksi säiettä käsittelee samaa muuttujaa, emme tiedä kumpi tapahtuu:**





Future ja Promise

Aalto-yliopisto
Perustieteiden korkeakoulu

https://presemo.aalto.fi/o2fi2026

“Helppo” rinnakkaistus Scalassa

- **Aina ei ole tarpeen käyttää suoraan säikeitä**
- **Ohjelmointitekniikat ja kirjastot tarjoavat abstraktioita rinnakkaisohjelmoinnille**
- **Scala tukee, esim.**
  - Rinnakkaisia kokoelmia (parallel collections) community-modulissa: https://docs.scala-lang.org/overviews/parallel-collections/overview.html
  - Futures ja Promises: https://docs.scala-lang.org/overviews/core/futures.html
- **CS-E4580 Programming Parallel Computers –kurssi* tarjoaa paljon tietoa, miten rinnakkaisuutta käytetään nyky-CPUissa**
  - Sen jatkoksi CS-E4690 Programming Parallel Supercomputers

* https://ppc.cs.aalto.fi/

`Future` ja `Promise`

- Ohjelmoinnin abstraktio, joka kuvaa rinnakkaista laskentaa ja arvoja
- Näppärä ketjuttaessa (chain) operaatioita käyttäen funktionaalistsa ohjelmointityyliä, jotta ohjelma ei jää jumiin

- `Future` tarkoittaa asynkronisen laskennan tulosta
- `Promise` on paikkamerkki objektille, joka tulee lopulta (tulevaisuudessa) saataville `Future`ssa

Esimerkki (alustusta myöhemmälle)

- **Haluamme hakea jonin websivun HTML-sisällön**
  - Onnistuu Javan http-kirjaston avulla:

```
import java.net.http._
import java.net.URI
import java.time.Duration
// Helper to make a request and get the response
def mkRequest(url : String) :HttpResponse[String] =
  // Construct a HTTP request
  val request = HttpRequest.newBuilder()
    .uri(URI.create(url))
    .build()
// Make a client
  val client = HttpClient.newBuilder()
    .followRedirects(HttpClient.Redirect.NORMAL)
    .connectTimeout(Duration.ofSeconds(30))
    .build()
// Send request, wait for the result, then return it
  client.send(request,
    HttpResponse.BodyHandlers.ofString())
end mkRequest
```

Esimerkki (jatkuu, ja palaamme vielä…)

- **Voimme esim. Konsolissa hakea satunnaisen artikkelin Wikipediasta:**

```
// Make request. Note the likely delay in the console
scala> val res = mkRequest("https://en.wikipedia.org/wiki/Special:Random")
val res: java.net.http.HttpResponse[String] =
  (GET https://en.wikipedia.org/wiki/Greg_Lloyd) 200

// We can use `body` to get the HTML as a string
scala> res.body
val res0: String =
<!DOCTYPE html>
<html class="client-nojs" lang="en" dir="ltr">
<head>
...
```

- **Viive `mkRequest`in suorittamisessa johtuu siitä, että kutsu on synkroninen ja siksi se estää (blocking) suorituksen etenemisen, eli mitään ei voi tehdä kunnes funktio on suoritettu**

`Future`

- **`Future` merkitsee asynkronisen laskennan tulosta**
  - Laskenta saattaa päättyä joskus tulevaisuudessa, tai se voi epäonnistua
  - Arvo saattaa olla saatavissa tai voi olla olematta tässä kohtaa koodia
  - Futuren laskeminen lopulta tuottaa joko
    - onnistumisen (Succeed) arvon kera tai
    - epäonnistumisen (Fail) eli heittää poikkeuksen
    - (tai sitten laskenta ei ole vielä valmis (not completed)
  - Scala APIn dokumentaatio:
    - https://scala-lang.org/api/3.2.1/scala/concurrent/Future.html

`Futuren` käyttäminen

- **`Future` tuottaa hyvin suoraviivaisen tavan asynkroniselle käytölle**
- **Ensin, meidän täytyy tehdä `ExecutionContext`**
  - Voidaan ilmoittaa suoraan konsolissa tai epäsuorasti funktion parametrina
- **Sitten, kutsu vaan paketoidaan `Futuren` sisään**

```
import scala.concurrent._
// Create execution context for Futures
implicit val ec: scala.concurrent.ExecutionContext =
  scala.concurrent.ExecutionContext.global
val res = Future {
  mkRequest("https://en.wikipedia.org/wiki/Special:Random")
}
```

`Future`en käyttäminen konsolissa

- **Enää ei odoteta vaan heti voi jatkaa käskyjen antamista**
- **Webisivu-esimerkin suorittamisen konsolissa pitäisi tuottaa heti (immediately) jotakuinkin seuraavaa**

```
val res: scala.concurrent.Future[java.net.http.HttpResponse[String]] =
  Future(<not completed>)
```

- **Kutsu on nyt asynkroninen ja `not completed` tarkoittaa, että pyyntöä yhä suoritetaan.**
- **Muutama sekunti myöhemmin saadaan tulos:**

```
scala> res
val res2: scala.concurrent.Future[java.net.http.HttpResponse[String]] =
  Future(Success((GET https://en.wikipedia.org/wiki/Hastak) 200))
```

`Future`n käyttäminen

- **Future on joko not completed**
- **tai completed arvonaan joko**
  - `Success` (arvo)
  - `Failure` (poikkeus)
- **(vertaa Scalan `Option`)**

- **Jos esimerkin kutsun tulos oli `Success`, voimme kirjoittaa saadaksemme HTML-sivun:**

```
scala> res.value.get.get.body
val res3: String =
<!DOCTYPE html>
<html class="client-nojs" lang="en” dir="ltr">
<head> ...
```

- **Vähän kömpelöä.. Pitäs odottaa, että olemme varmoja, että Future on suoritettu. Onko parempaa tapaa?**

`Future` – takaisinkutsu (callback)

- **Koska Futuren tuloksen laskenta ei välttämättä ole tällä hetkellä saatavissa, tarvitsemme funktion arvojen saamiseksi sitten kun ne on saatavissa: Tehdään takaisinkutsu (callback)**
- **Tätä voi käyttää `foreach`-käskyn avulla**
  - Älä hämäänny: `foreach`in syntaksi on sama kuin kokoelmille, Future vaan ”sisältää” vain yhden arvon
- **Funktio suoritetaan lopulta, jos `Future` onnistui**

```
val res = Future {
  mkRequest("https://en.wikipedia.org/wiki/Special:Random")
}
// Callback, print the HTML if the future is successful
res.foreach(h=>println(h.body))
```

  - Anonyymi funktio `h` on `Future`n tulos eli mitä tuo ylempi palauttaa

`Future` – takaisinkutsu (callback)

- **Voimme käyttää `onComplete`, jos `future` saattaa myös epäonnistua**

```
// Success and Failure
import scala.util.{Success, Failure}
val res = Future {
  // Note: invalid URL for example
  mkRequest("https://invalid.invalid.invalid")
}
// Will print html on success, error msg on fail
res.onComplete { // only completed, not those still under work
  case Success(h) => print(h.body)
  case Failure(e) => print(s"Failed due to $e")
}
```

- **Tulos, jos URL oli rikkinäinen:**

```
Failed due to java.net.ConnectException
```

`Future` – sovittaminen (composing)

- **Yksi `Futuren` etu on se, että voimme sovittaa uusia `futureja`**
- **Funktionaalista ohjelmointityyliä käyttämällä voidaan luoda ketjuja asynkornista laskentaa**
- **Perusesimerkki map**

```
val res = Future {
  mkRequest("https://en.wikipedia.org/wiki/Special:Random")
}
// Automatically get body in the Future, if res completes successfully
val bod = res.map(_.body)
```

- **Konsolissa ajaminen tuottaa:**

```
val res: scala.concurrent.Future[java.net.http.HttpResponse[String]] =
  Future(<not completed>)
val bod: scala.concurrent.Future[String] = Future(<not completed>)
```

- **Body on `Future[String]`, joka on riippuvainen `res`:stä**

`Future` – toipuminen (recovering)

- **Jos tulee `Failure`, siitä voi toipua käyttämällä esim. `recover` tai `recoverWith`**
  - `recover`:
  - Esim. Jos pyyntö epäonnistuu mistä tahansa syystä, korvataan body-merkkijono `Success`:lla, joka sisältää tyhjän merkkijonon

```
val res = Future {
  // Note: invalid URL for example
  mkRequest("https://invalid.invalid.invalid")
}
// When res fails, this is propagated to body
// this will recover any Failure (because we match _)
// and replace it with an empty string:
val bod = res.map(_.body).recover{case _ => ""}
```

`Future` – toipuminen (recovering)

- **Jos tulee `Failure`, siitä voi toipua käyttämällä esim. `recover` tai `recoverWith`**
  - `recoverWith`
  - Esim. Jos alkuperäinen pyyntö epäonnistuu, korvataan se toisen pyynnön tuloksella

```
val res = Future {
  // Note: invalid URL for example
  mkRequest("https://invalid.invalid.invalid")
}
// New future which will be same as res if res
// succeeds or the provided Future if res fails
val backup = res.recoverWith {
  case _ => Future{mkRequest("https://www.aalto.fi")}
}
```

Future – for-silmukkarakenne

- **map-käskyn sijaan voimme käyttää for-yield saavuttaaksemme saman asian**

```
val res = Future {
  mkRequest("https://en.wikipedia.org/wiki/Special:Random")
}
// Automatically get body in the Future, if res completes successfully
val bod = for x <- res yield x.body
```

- **Saattaa näyttää hämmentävältä aluksi, mutta ajattele, että Future on kokoelma, jossa on vain yksi elementti**

`Future` – `for`-silmukkarakenne

- **Tämä tapa on tehokas, kun yhdistetään useita `Futureja`**
- **Esim. Hae kaksi sivua (asynkronisesti) ja aseta body olemaan pienempi HTML-sivuista. Jätä toinen huomiotta:**

```
val resA = Future {
  mkRequest("https://en.wikipedia.org/wiki/Special:Random")
}
val resB = Future {
  mkRequest("https://en.wikipedia.org/wiki/Special:Random")
}

val bod = for (x <- resA; y <- resB) yield
  if (x.body.length < y.body.length) then x.body else y.body
```

  - Säikeillä olisi pitänyt varmistaa, että molemmat kutsut on tehty
  - Tässä ohjelma pitää itse huolen siitä, että kutsut on tehty

`Promise`

- **`Promise` on paikka objektille, joka tulee lopulta tarjolle (`Futuressa`)**
  - Luvattu objekti ei ole saatavissa silloin kuin se esitellään (declare)
  - Mutta sen pitäisi valmistua (completed) arvon kanssa tai epäonnistua (failed) poikkeuksen kera
- **`Promise`lla on `Future`**
- **`Future`a voidaan käyttää `Promise`n täyttämiseen**
- **Siinä missä Future joko onnistuu tai epäonnistuu riippuen laskennasta, Promise on objekti, jonka on suoritettavassa koodissa myöhemmin (ohjelmoijan toimesta tai `Future`ssa)**
- **Scala API:n dokumentaatio: https://www.scala-lang.org/api/3.2.1/scala/concurrent/Promise.html**

`Promise` – esimerkki

- **Pyydetään käyttäjää antamaan URL ja yritetään tehdä pyyntö, jollei annettu merkkijono ole tyhjä**
- **`Promise` voidaan täyttää**
  - Onnistuneesti käyttäen `success`-metodia
  - Epäonnistuen käyttäen `failure`-metodia

```
import scala.io.StdIn.readLine
// Promise an url, eventually
val url = Promise[String]()
// When it is delivered make a request
val res = url.future.map(s=>mkRequest(s))
// Ask the user for the URL
print("Enter an URL (e.g. https://www.aalto.fi),"+
  " empty input aborts: ")
val u = readLine()
// We manually complete the promise
if(u.length > 0) then url.success(u)
else url.failure(new RuntimeException("Empty URL"))
```

- **Luvattu `Future` saadaan käyttöön jossain vaiheessa**

`Promise` – täydentäminen (completion)

- **Yksi tai useampi `Future` voidaan rekisteröidä täydentämään (complete) `Promise`**
  - Mutta vain yksi niistä voi täyttää sen, vain yhden kerran
- **Esim. Useamman lähteen tuottaman asynkronisen kutsun lopputuloksen lupaaminen:**

```
// Promise a web page
val bod = Promise[String]()
// List if web sites containing the same resource
val mirrors = Seq(
  "http://mirror.linux.org.au/debian/",
  "http://debian.unnoba.edu.ar/debian/",
  "http://ftp.nz.debian.org/debian/")
// Query all of them concurrently
for url <- mirrors do
  // Make request, and get body
  val f = Future { mkRequest(url) }.map(_.body)
  // Complete the promise when any request finishes
  bod.completeWith(f)
end for
```

`Future`t ja `Promise`t ohjelmoijalle

- **`Future` ja `Promise` ovat funktionaalisen ohjelmoinnin konstruktioita**
- **Ne tuottavat meille mahdollisuuden – jossain määrin – eristää laskennan satunniaset suhteet lähdekoodin vastaavista**

- **Eli mikä odottaa mitä?**

https://presemo.aalto.fi/o2fi2026

Mitä voimme laskea rinnakkain?

Aalto-yliopisto
Perustieteiden
korkeakoulu

https://presemo.aalto.fi/o2fi2026

Yksinkertaistettu funktionaalinen näkökulma

- **Yksinkertaistetusti: laskenta on sarja arvoja, jotka saamme soveltamalla puhtaita funktioita sarjan edellisiin arvoihin**
  - Muistattehan: funktio on puhdas (pure), jos
  1. Sillä ei ole sivuvaikutuksia
  2. Funktion suorittaminen samoilla parametreilla tuottaa aina saman tuloksen

```
// ...
val y1 = f(x1)
val y2 = g(y1,x2)
val y3 = h(y1,y2)
// ... and so on
```

Riippumattomuus (independence)

- **Oletetaan, että `f` on funktio, jota käytetään johonkin datan `a`, `b`, `c`, `d`:**
- **Koska arvot ovat riippumattomia toisistaan, voimme rinnakkaistaa laskennan:**

```
// ...
val y1 = f(a)
val y2 = f(b)
val y3 = f(c)
val y4 = f(d)
//...
```

- **Tämä on esimerkki SIMD-rinnakkaisuudesta (Single instruction, Multiple Data), joka nopeuttaa suorittamista**
  - Alhaisella tasolla, esim. grafiikkakorteilla, tämä voi johtaa huomattavaan nopeuden lisäämiseen, jos dataa on riittävästi
  - Korkeammilla tasoilla vähemmän perustamiskustannusten vuoksi

Riippuvuus (dependence)

- **Riippumaton voidaan rinnakkaistaa, mutta riippuvaa ei voi**

```
//...
val y1 = g(y0)
val y2 = g(y1)
val y3 = g(y2)
val y4 = g(y3)
//...
```

- **Tulosten välillä on syy-riippuvuus (causal dependency). Suoritus pitää tehdä tietyssä järjestyksessä**

Riippuvuuksien selvittäminen

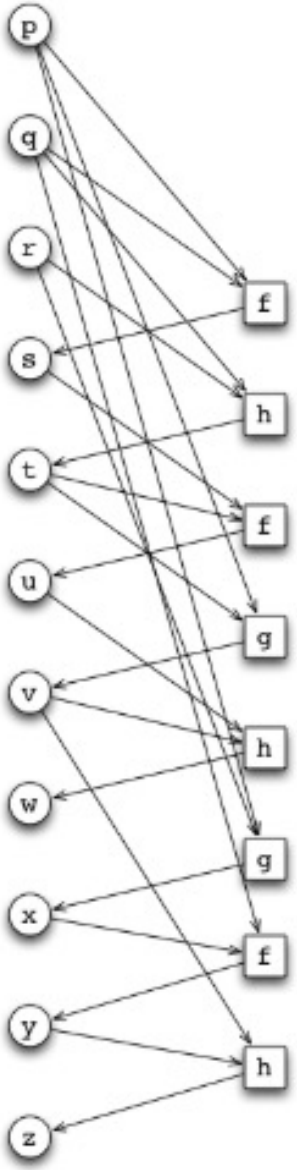
```
// ...
val s = f(p,q)
val t = h(q,r)
val u = f(s,t)
val v = g(p,t)
val w = h(u,v)
val x = g(p,r)
val y = f(q,x)
val z = h(v,y)
// ...
```

- **Yleisesti laskenta ei ole täysin riippuvaa eikä täysin riippumatonta**
- **Miten voidaan kartoittaa riippuvuuksia ja erottaa ne ohjelmakoodin peräkkäisestä luonteesta?**

Riippuvuuksien selvittäminen

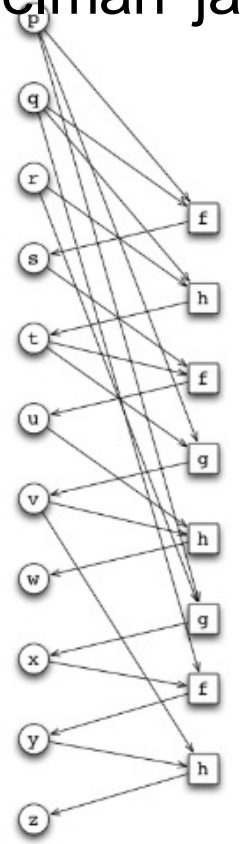
- **Solmut (ympyrät tai laatikot) esittävät muuttujia ja funktion kutsuja**
- **Reunat (nuolet) kertovat riippuvuuksista:**
  - a→b: b on riippuva a:sta
- **Tämä on suunnattu syklitön verkko (directed asyclic graph, DAG)**
  - Esittää samaa laskentaa (funktioiden parametrien järjestyksessä)

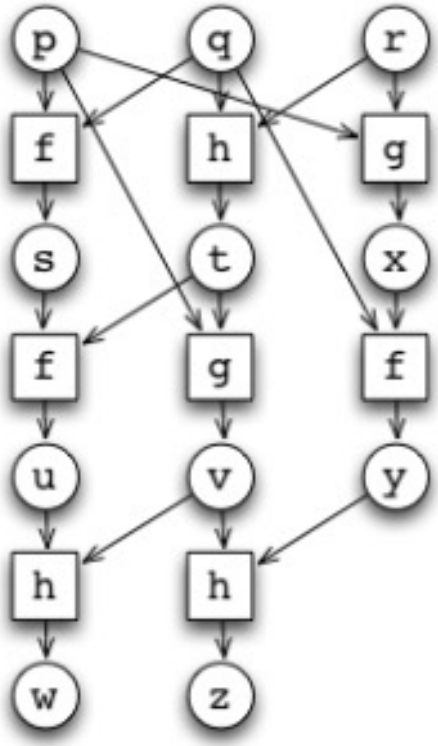
```
// ...
val s = f(p,q)
val t = h(q,r)
val u = f(s,t)
val v = g(p,t)
val w = h(u,v)
val x = g(p,r)
val y = f(q,x)
val z = h(v,y)
// ...
```



Riippuvuuksien selvittäminen

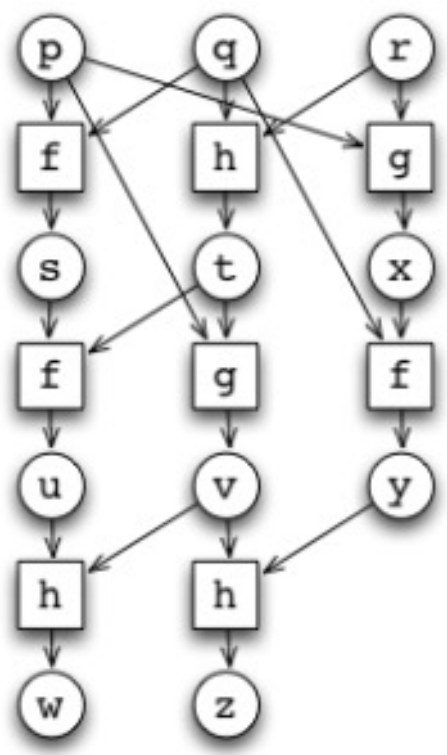
- **Graafien avulla riippuvuudet on helpompi ’nähdä’**
- **Sama graafi voidaan piirtää usealla tavalla:**
  ‘Ohjelman’ järjestys Yksi topologinen taso rivillä:





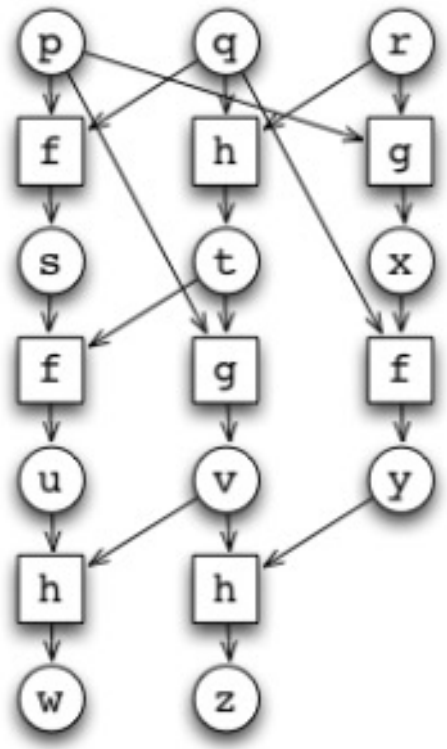
- solmut, joissa ei ole sisääntulevaa reunaa ensimmäisellä tasolla,
- sitten millä tahansa tasolla k, ainakin yksi sisääntuleva reuna tasolta k-1, eikä reunoja sitä ylemmiltä tasoilta

Riippuvuuksien selvittäminen



- Olettaen, että meillä on riittävät resurssit rinnakkaiselle laskennalle, voimme käyttää DAG-graafia tekemään laskennan ahnaammin (greedy manner)
- Aloitamme jokaisen funktion suorituksen ehti kun sille parametrina annettavat arvot ovat valmiit
- `Futures and promises` ovat abstraktioita, joiden avulla voimme rakentaa näitä riippuvuuksia ohjelmiin

Vähimmäissuoritusaika (minimum makespan)



- **Jos jokaisen funktion suoritusaika on tunnettu, vähimmäissuoritusaika (minimum makespan) on se polku läpi graafin ylhäältä alas, jossa on suurimmat kulut**
  - Ainakin tämän verran tarvitaan aikaa ohjelman suorittamiseen

Kierroksen tehtävistä

1. **Futures**
2. **Promises**
3. **Haastetehtävä: Säikeitä – alhaisen tason rinnakkaisuutta**

- **Koodin ja kommenttien lukeminen**
- **Lue Scalan Futures and promises –dokumentaatio**
  - Ja tämän esityksen esimerkit