CS-A1120 Programming 2

Aalto-yliopisto
Perustieteiden
korkeakoulu

Lecture 9
Spring 2026

Sanna Suoranta (Finnish, T1 at 2pm) and
Vesa Hirvisalo/Lukas Ahrenberg (English, T1 at 10am)

4.5.2026 https://presemo.aalto.fi/prog2

In O2 today

Module III: Frontiers

https://presemo.aalto.fi/prog2

Aalto-yliopisto
Perustieteiden
korkeakoulu

Today In O2

Concurrency and parallelism

Aalto-yliopisto
Perustieteiden korkeakoulu

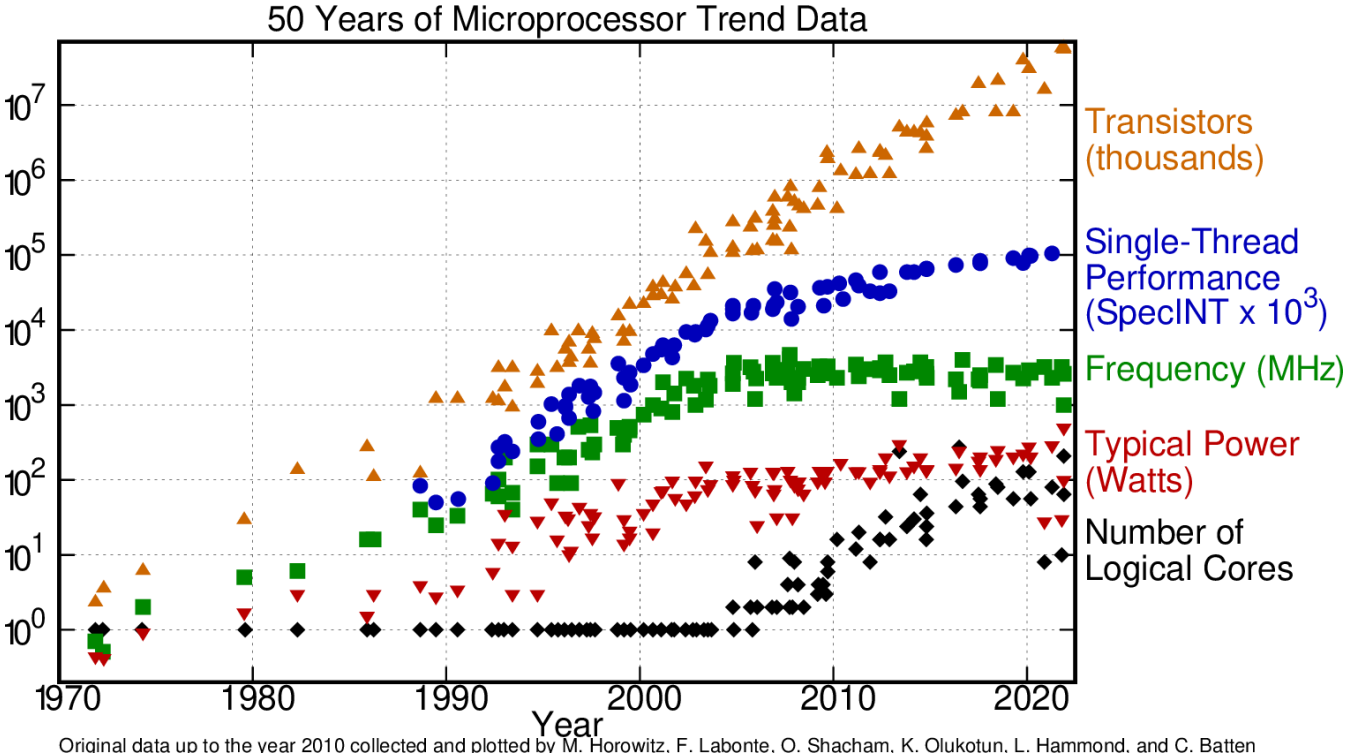
https://presemo.aalto.fi/prog2

Round 9 learning objectives

- **After this round, You**
  - can exemplify parallel and concurrent computation
  - are aware of computing threads and how to create them in Scala
  - are familiar with asynchronous computation and synchronisation
  - can use the high-level abstractions of Futures and Promises in Scala
  - can explain pure functions
  - are aware of (in)dependency in parallel computation, and minimum makespan

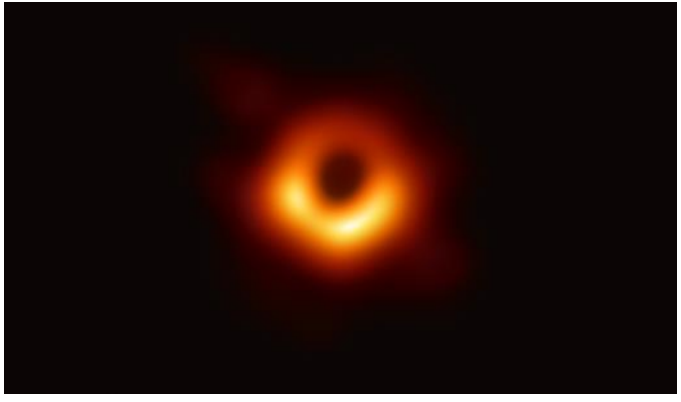
Moore's law

- **The number of transistors in an IC doubles every 1.5-2 years**



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

What is one CPU is not enough?



- **Event Horizon Telescope**
  - Radio telescopes distributed around the world
  - Each yields 350 TB/day*

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/**
  - Public web crawler data
  - January 2022 crawl set:
    - 2,95 billion web pages (320 TiB)
  - Access via Amazon Open Data registry***
  - Amazon Snowmobile data truck
    - 100PB pf data ****

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

Scalable distributed infrastructure (1/4)

Computer clusters

- Networked set of computers (nodes)
- Homogeneous
- Nodes are physically close (same rack/hall)
- Often, nodes work on the specific task(s)

Scalable distributed infrastructure (2/4)

Computing grids

- Networked set of computers (nodes)
- Heterogeneous
- Geographically dispersed
- Nodes usually work on different tasks

Scalable distributed infrastructure (3/4)

Warehouse-scale computers

- Extremely homogeneous and hierarchical
- One building with cooling and electricity
- Specialised hardware & software
- Operated by big companies
- Often for single specific purpose

Scalable distributed infrastructure (4/4)

Data centers

- Typically for hosting internet services (search engines, video streaming, …)
- One building with cooling and electricity
- Specialised hardware & software
- There are multiple tenancy models
- Services are virtualized (e.g., virtual machines)
- A lot of variance (and history)

On concurrent computations

- **So far, we have viewed computation as a simplified phenomenon**
- One processor (core) executes one machine instruction at a time
  - In the real world, sequential such execution is rare
  - Typically computing takes place in parallel on multiple scales
- **Semantics** (i.e., what programs mean)
  - Sequential: doing things step-by-step
  - Concurrent: having multiple tasks (steps) available at a point
- **Execution** (i.e., how computing is done)
  - Parallel: Progress on multiple tasks at the same time instant
  - Serial: Progress only one task at the same time instant

Raptor Lake CPU architecture

- **Intel Core i9-14900F processor**
  - 24 cores working in parallel on a chip
- **Heterogeneous architecture**
  - 8 "performance" Raptor Cove architecture
  - 16 "efficient" (energy) Gracemont architecture
  - The architecture differs but adhere the same memory model
- **Processors pipelines support vectorised instructions**
  - 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 supercomputer in Kajaani, Finland

- **Lumi-C partition (CPU only)**
  - 1376 nodes
  - two AMD EPYC 7763 – processor with 64 cores
  - CPU cores support AVX2 256-bit vector instructions
  - 16 16-bit floating point numbers in parallel
  - In total 5.63×1015 Floating point operations per second (FLOPS)

- **There is also a GPU (Graphics Processing Unit)-based partition**
  - 2978 nodes, each with an AMD Trento CPU and four MI250X GPUs.

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

Parallelism and concurrency in practice

- **In practice, computing is a phenomenon where multiple concurrent programs are executed in parallel but are interacting with each other**
  - The less interaction there is the easier it is to make use of parallel execution
- **Benefits:**
  - Parallel speedup: some tasks are parallelisable by nature
  - The full task (or part of it) can be divided into independent sub-tasks
  - Throughput: Running the same task in parallel means that we can handle more data
  - Availability: Makes scheduling and load balancing possible so that different parts of the system are available when needed

Threads

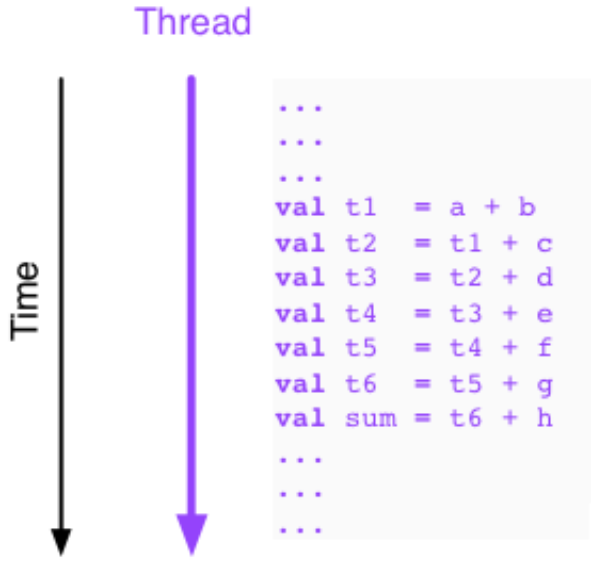
Aalto-yliopisto
Perustieteiden
korkeakoulu

https://presemo.aalto.fi/prog2

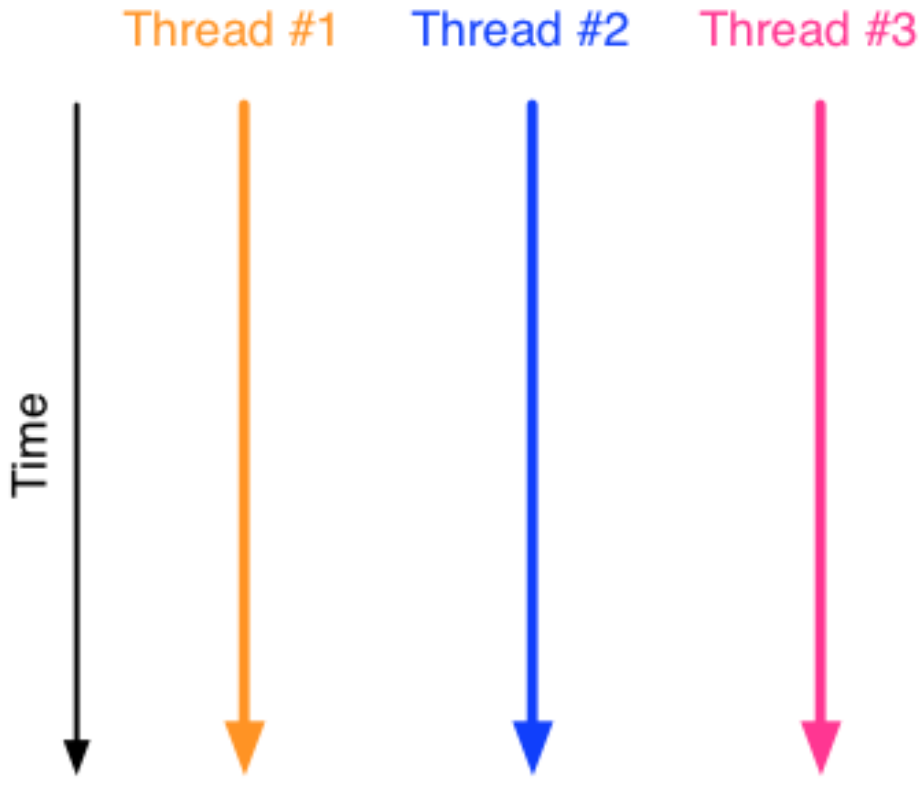
Single thread execution

- A thread is an independent sequence of instructions
- A program executed with a single thread is a sequential program (like we have done so far in this course)

```
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
```

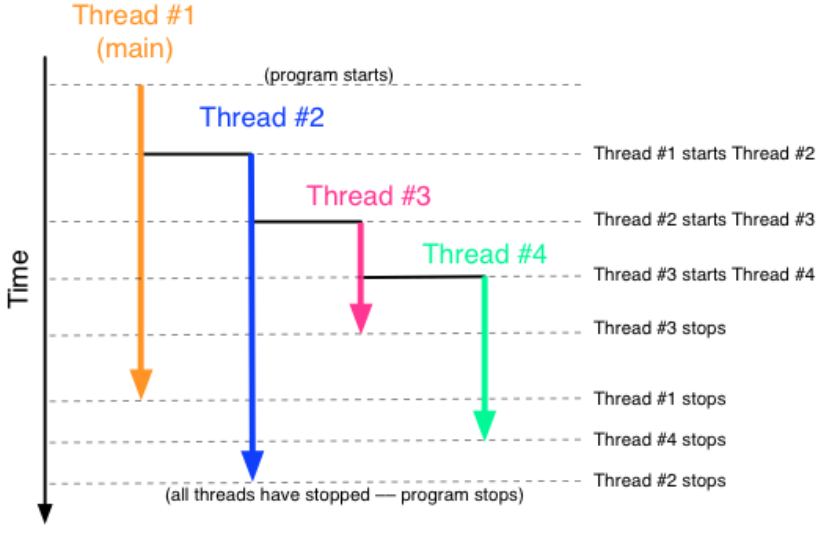


Multiple threads (1/2)



Multiple threads (2/2)

- **Each thread has its own internal state**
  - Program counter, registers,
- **In addition the threads have a shared state**
  - Main memory, resources accessed through OS, …
- **Each thread is executed asynchronously**
  - In effect, fully independent of what the other threads are doing
  - Unless the programmer explicitly synchronises the threads



Scala threads and asynchronous execution

- **An object of a class extending Runnable can be run asynchronously by a Thread in Scala**
  - The code we want to run in the tread is put in the run method

```
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()
```

Scala threads and asynchronous execution

- **An object of a class extending Runnable can be run asynchronously by a Thread in Scala**
  - The code we want to run in the tread is put in the run method

```
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()
```

- **What is the result?**

https://presemo.aalto.fi/prog2

Scala threads and asynchronous execution

- **An object of a class extending Runnable can be run asynchronously by a Thread in Scala**
  - The code we want to run in the tread is put in the run method

```
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()
```

- **Actually, we do not know!**
- **The execution order may vary!**

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

Scala threads and asynchronous execution

- **This is more visible**

```
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()
```

Scala threads and asynchronous execution

- **This is more visible**

```
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()
```

- **Three executions (printing one letter at a time):**

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

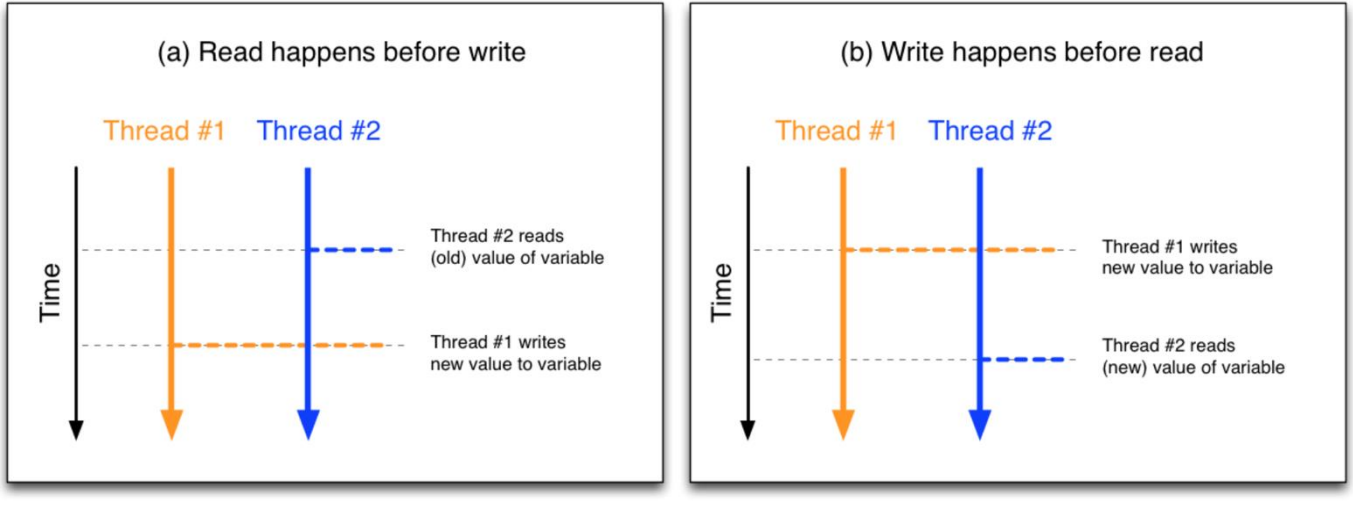
Synchronisation and dependencies

- **Threads executes asynchronously unless the programmer synchronises the threads**
  - Starting a thread: t.start() The starting thread is not independent of the thread that starts it, but execution is asynchronous after it starts
  - Joining with a thread (waiting): t.join() Causes the thread making the call to be suspended (blocked) until the execution of thread t is finished. Then execution resumes

```
val h1 = HelloFrom2("Rome")
val h2 = HelloFrom2("New Delhi")
val h3 = HelloFrom2("Canberra")
val t1 = Thread(h1)
val t2 = Thread(h2)
val t3 = Thread(h3)
t1.start(); t1.join(); t2.start(); t2.join(); t3.start()
```

Synchronisation

- **Without synchronisation between threads we cannot know in which order a common resource is accessed!**
  - It is the responsible of the programmer to make sure to avoid deadlocks where threads wait for each other indefinitely,
  - or starvation, where a thread gets no execution time.
- **E.g., two threads of a process access the same variable:**



Futures and Promises

Aalto-yliopisto
Perustieteiden
korkeakoulu

hthttps://presemo.aalto.fi/prog2

“Easy” concurrency in Scala

- **It is not always necessary to work directly with threads**
- **Programming techniques and libraries can offer some abstraction for parallel programming**
- **Scala supports, for example,**
  - Parallel collections (through a community module)
  - Futures and Promises
- **The course CS-E4580 Programming Parallel Computers has much more on how to utilize parallism in modern CPUs in general.**
  - Note also CS-E4690 Programming Parallel Supercomputers

Futures and Promises

- **Programming abstraction that describes concurrent computations and values.**
- **Useful to chain operations using a functional programming style in order to avoid blocking**
  - A **Future** denotes the result of an asynchronous computation
  - A **Promise** is a placeholder for an object which will be made available eventually (in the future)
- **See Scala docs: Futures and Promises**

Interlude – setting up an example (1/2)

- **Say that we wanted to retrieve the contents of a web page**
  - We could use Java's http library for this:

```
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
```

Interlude – setting up an example (2/2)

- **Now we can use this, e.g. in the console to get a random article from Wikipedia…**

```
// 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>
...
```

- **In console: note the delay when executing mkRequest, this is because the call is synchronous and therefore blocking.**

Future

- **A Future denotes the result of an asynchronous computation**
  - That is, the computation might complete in the future (or it might fail)
  - That is the value may or may not be available at that point in the code
  - A future computation will eventually Succeed with a value or Fail with an exception
- **Scala API doc: Future**
  - https://scala-lang.org/api/3.2.1/scala/concurrent/Future.html

Future – on using

- **A Future provides a very straight forward way to do this asynchronously**
  - First, we need to provide an ExecutionContext
  - Can be declared directly in console or provided as an implicit parameter to a function
- **Then we simply wrap the call in a Future**

```
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 – on using

- **Executing previous example in a console should immediately print something like**

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

- **The call is now asynchronous, and not completed means that the request is still ongoing.**
- **Checking it a few seconds later, however:**

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

Future – on using

- **Futures are either not completed,**
- **or, completed with one of**
  - Success(value)
  - Failure(exception)
- **(Compare to Scala's Option)**

- **If the http call was a Success we could do:**

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

- **A bit clumsy, and we would need to wait until we were sure the Future had completed - is there a better way?**

Future – callbacks

- **Future computation may not currently be available**
  - We can provide functions to apply to the values once they are – these are called callbacks
    - This can be done using foreach
    - Do not be confused by the syntax, although it is the same as for collections, a Future only 'contains' one value.
- **The provided function will be executed eventually, if the Future succeeds**

```
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))
```

Future – callbacks

- **We can also use onComplete for the option of doing something also when the future fails to complete.**

```
// 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")
}
```

- **The result (if the URL is invalid):**

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

`Future` – composing

- **A benefit of Futures: we can compose to create new futures**
  - Thus, 'chains' of asynchronous computing can be created by using a functional programming style
- **A basic basic example is 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)
```

- **In the console:**

```
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]`, which is dependent on `res`**

Future – recovering

- **It is also possible to recover from a Failure, for example using recover and recoverWith**
- `recover`:
  - If a the request fails for any reason replace the body string with a Success containing the empty string

```
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 – recovering

- **is also possible to recover from a Failure, for example using recover and recoverWith**
- `recoverWith`
  - If the original request fails, replace it with the result of another request

```
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 – the for loop construct

- **Instead of using map we could have used a for-yield to achieve the same thing**

```
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
```

- **Might look strange at first, but think of a Future as a collection of a single element.**

Future – the for loop construct

- The for construct is powerful when combining multiple Futures.
- Get two pages (asynchronously) and set bod to the smallest page HTML. Ignore the other:

```
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
```

Promise

- **A Promise is a placeholder for an object which will be made available eventually (in the future)**
  - The promised object is not available at the point when it is declared
  - But it should be completed with a value of failed with an exception
- **A Promise has a Future**
- **A Future can be used to complete a promise**
- **Scala API doc: Promise**
  - While a Future succeeds or fails depending on a computation, a Promise is an object that must be completed in the code (by the programmer, or by some Future)
    - see: https://www.scalalang.org/api/3.2.1/scala/concurrent/Promise.html

`Promise` – an example

- **Ask the user for an URL and try to make a request unless the provided string is empty**
- **`Promise` may be completed**
  - successfully using the success method
  - as failed, using the failure method

```
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"))
```

- **The promised Future can be accessed by the future method**

Promise – completion

- **One or more futures can be registered to complete a Promise**
  - But completed only once!
- **E.g., promise the result of a resource queried asynchronously from multiple sources:**

```
// 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` and `Promise` for a programmer

- **Futures and Promises are functional programming constructions.**
- **They allows us to some extent to isolate casual relationships of the computing from those in the source code.**

What can be computed in parallel?

Aalto-yliopisto
Perustieteiden
korkeakoulu

https://presemo.aalto.fi/prog2

A simplified functional perspective

- **Computing**
  - a sequence of values which we get by applying pure functions to previous values in the sequence
- **Definition: A function is pure, if**
  1. It does not have side effects
  2. Evaluating it on the same parameters always returns the same result

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

Independence

- **Assume f is a function applied to some data `a`, `b`, `c`, `d`:**
- **Because the values are independent of each other we can parallelise the computations:**

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

Thread #1	Thread #2	Thread #3	Thread #4
`val y1 = f(a)`	`val y2 = f(b)`	`val y3 = f(c)`	`val y4 = f(d)`

- **This is an example of SIMD (Single Instruction, Multiple Data) parallelism.**
  - SIMD parallelism can lead to huge speedups at low level (e.g. on a graphics card) or if sufficient data is available. At higher levels the speedup can be less substantial due to set-up costs.

Dependence

- **Independent can be made parallel, but not the dependent**

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

- **Causal dependency between results. Execution must be in order.**

Mapping out the dependencies

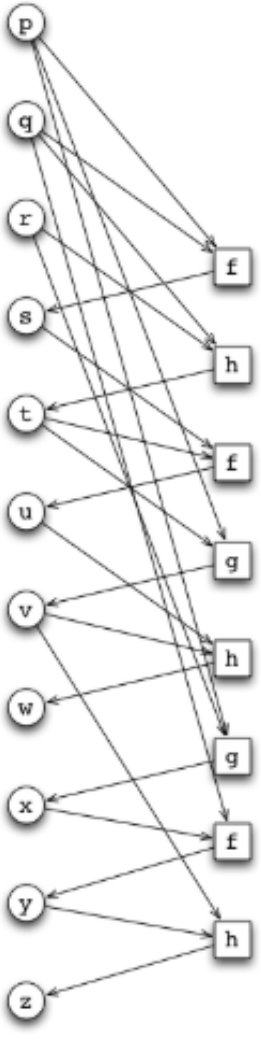
```
// ...
val s = f(p,q)
val t = h(q,r)
val u = f(s,t)
val v = g(p,t)
val w = h(u,v)
val x = g(p,r)
val y = f(q,x)
val z = h(v,y)
// ...
```

- **In general, neither complete dependence, nor complete independence.**
- **How can dependencies be mapped out, and separated from the sequential nature of the program code?**

Mapping out the dependencies

- **Nodes (circles or boxes) indicate vars and function calls**
- **Edges (arrows) indicate dependency:**
  - a→b: b depends on a
- **This graph is a Directed Acyclic Graph (DAG)**
  - Represents the same computation (up to the order of the function parameters)

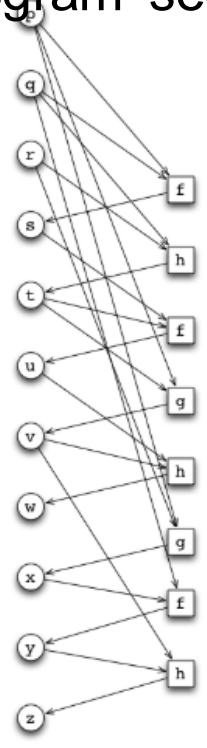
```
// ...
val s = f(p,q)
val t = h(q,r)
val u = f(s,t)
val v = g(p,t)
val w = h(u,v)
val x = g(p,r)
val y = f(q,x)
val z = h(v,y)
// ...
```



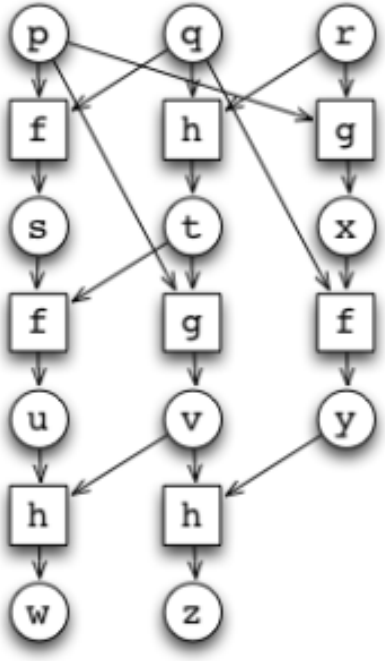
Mapping out the dependencies

- **We can draw the same graph with different layouts:**

‘Program’ sequence

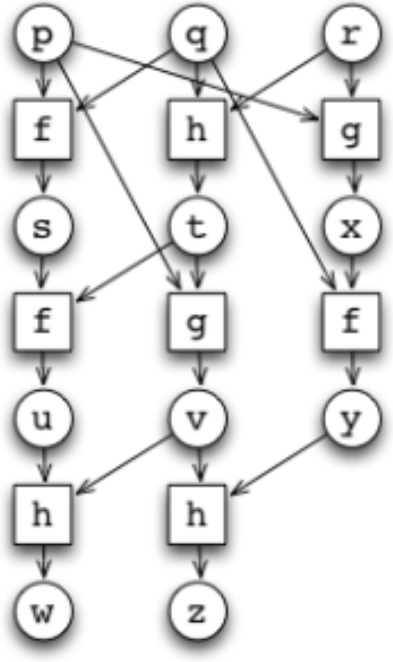


One topological level* at each row:



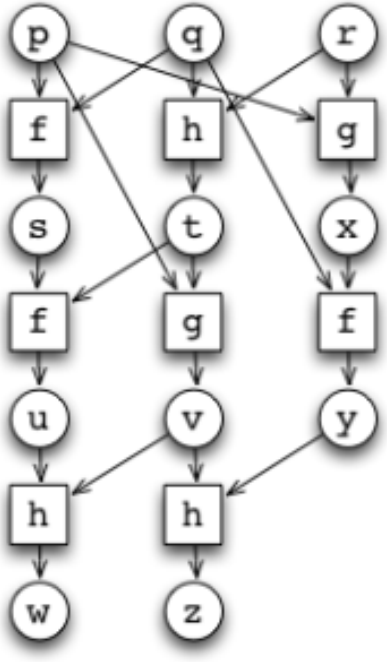
- * Nodes with no incoming edges on first level, then at any other level k, at least one incoming edge from level k−1 and no edges level higher than that.
- Now we can 'see' dependent/independent structure more clearly.

Mapping out the dependencies



- Assuming we have sufficient parallel computing resources, we can now use the DAG to do the computations in a greedy manner
- We start each function immediately when the values passed to that function as parameters are ready
- Futures and Promises are abstractions that let us build these dependencies when programming

Minimum makespan



- If the time to compute each function call is known, then the minimum makespan is that path from top to bottom with the greatest sum of costs
- We will need at least this much time to execute the program

Exercises

1. **Futures**
2. **Promises**
3. **Challenge:**
   Threads – low-level parallelism

- **Read the code and comments**
- **Check the Futures/Promises documentation**
  - And the examples in this presentation!