High-level parallel programming: Scala/Akka on the JVM

Contents of Lecture 2

- The Scala programming language
- Actors
- Parallel programming with Akka

Purpose of this lecture

- That you will understand why Scala may be interesting
- You will understand the key concepts of message passing using **actors**, introduced by Carl Hewitt at MIT 1973.
- You will understand enough about Scala actors that you can write a parallel version of the preflow-push algorithm in Scala

Scala and Akka

- Martin Odersky designed Generic Java and the Java compiler `javac` for Sun. He is a professor at EPFL in Lausanne.
- The Scala language produces Java byte code and Scala programs can use existing Java classes.
- When you run a Scala program, the JVM cannot see any difference between Java and Scala code.
- A good source to start with Akka: `https://developer.lightbend.com/start`
- Or start with the example at Tresorit
- Akka was created by Jonas Bonér from Sweden
- Lightbend is a company founded by Odersky, Bonér and another person.
- Download sbt: `https://www.scala-sbt.org/download.html`

Scala is a functional and object oriented language

- It is intended to be **scalable** and suitable to use from very small to very large programs.
- The Scala compiler, `scalac`, usually can infer the types of variables so you don't have to type them.

```
var capital = Map("Denmark" -> "Copenhagen", "France" -> "Paris", "Sweden" -> "Stockholm");

capital += ("Germany" -> "Berlin");

println(capital("Sweden"));
```

- There is no need to declare the type of the variable `capital` since `scalac` can do it for you.
- Less typing can potentially lead to faster programming — at least if the tedious part of the typing can be eliminated.

Shorter class declarations

- A short class declaration in Scala:

```scala
class Example(index: Int, name: String)
```

- The same class in Java:

```java
class Example {
    private int index;
    private String name;

    public Example(int index, String name) {
        this.index = index;
        this.name = name;
    }
}
```

Redefinable operators as in C++

```
def factorial(x: BigInt): BigInt = if (x == 0) 1 else x * factorial(x - 1)
```

- A function is defined using `def` and = and an expression.
- Or using the Java class `BigInteger`:

```
import java.math.BigInteger;

def factorial(x: BigInteger): BigInteger =
  if (x == BigInteger.ZERO)
    BigInteger.ONE
  else
    x.multiply(factorial(x.subtract(BigInteger.ONE)))
```

- It is obvious which is nicer.

Scala is statically typed

- Lisp, Smalltalk, Ruby, Python and many other languages are dynamically typed, which means type checking is performed at runtime.
- Scala and to a very large extent also C are statically typed.
- Of course, C is a very small language and much easier to type check.
- For C, if you use `<stdarg.h>` (which you usually shouldn't) or insane casts (which result in undefined behaviour = serious bug) the C compiler will not help you.
- Look at this program:

```
(defun sumlist (h)
        (if (null h) 0 (+ (car h) (sumlist (cdr h)))))

(setq b '(1 2 3 4))
(setq c '("x" "y" "z"))

(print (sumlist b))
(print (sumlist c))
```

- Which language is it?
- When is the error detected during dynamic type checking?

Answers

```
(defun sumlist (h)
        (if (null h) 0 (+ (car h) (sumlist (cdr h)))))

(setq b '(1 2 3 4))
(setq c '("x" "y" "z"))

(print (sumlist b))
(print (sumlist c))
```

- The language is Common Lisp.
- The error is detected when adding the string `"z"` to zero: Old measurement from several years back (also for C below)

```
> time clisp a.lisp

10
*** - +: "z" is not a number



real    0m0.062s
user    0m0.045s
sys     0m0.017s
```

A C Compiler must issue a diagnostic message

```
#include <stdlib.h>

typedef struct list_t   list_t;

struct list_t {
        list_t*         next;
        int             value;
};

list_t* cons(int value, list_t* list)
{
        list_t*         p;

        p = malloc(sizeof(list_t));
        if (p == NULL)
                abort();
        p->value = value;
        p->next = list;

        return p;
}

int sumlist(list_t* h)
{
        return h == NULL ? 0 : h->value + sumlist(h->next);
}

int main(void)
{
        list_t*         p;
        list_t*         q;

        p = cons(1, cons(2, cons(3, cons(4, NULL))));
        q = cons("x", cons("y", cons("z", NULL)));      // static type error
}
```

GCC output

```
> time gcc a.c
a.c: In function 'main':
a.c:35: warning: passing argument 1 of 'cons' makes integer from pointer without a cast
a.c:11: note: expected 'int' but argument is of type 'char *'
a.c:35: warning: passing argument 1 of 'cons' makes integer from pointer without a cast
a.c:11: note: expected 'int' but argument is of type 'char *'
a.c:35: warning: passing argument 1 of 'cons' makes integer from pointer without a cast
a.c:11: note: expected 'int' but argument is of type 'char *'

real    0m0.150s
user    0m0.103s
sys     0m0.047s
```

Clang output

```
> time clang -S a.c
a.c:35:31: warning: incompatible pointer to integer conversion passing
      'char [2]' to parameter of type 'int'
        q = cons("x", cons("y", cons("z", NULL)));
                                     ^~~
a.c:11:18: note: passing argument to parameter 'value' here
list_t* cons(int value, list_t* list)
                 ^
a.c:35:21: warning: incompatible pointer to integer conversion passing
      'char [2]' to parameter of type 'int'
        q = cons("x", cons("y", cons("z", NULL)));
                           ^~~
a.c:11:18: note: passing argument to parameter 'value' here
list_t* cons(int value, list_t* list)
                 ^
a.c:35:11: warning: incompatible pointer to integer conversion passing
      'char [2]' to parameter of type 'int'
        q = cons("x", cons("y", cons("z", NULL)));
                 ^~~
a.c:11:18: note: passing argument to parameter 'value' here
list_t* cons(int value, list_t* list)
                 ^
3 warnings generated.

real    0m0.050s
user    0m0.030s
sys     0m0.020s
```

A Scala compiler must issue a diagnostic message

```
class Test {
  def sumlist(h:List[Int]) : Int = if (h.isEmpty) 0 else h.head + sumlist(h.tail);
  var a = List(1,2,3,4);
  var b = sumlist(a);
  var c = List("x", "y", "z");
  var d = sumlist(c);
}

> time scalac a.scala
a.scala:6: error: type mismatch;
 found   : List[java.lang.String]
 required: List[Int]
  var d = sumlist(c);
                  ^
one error found

real    0m1.697s
user    0m4.384s
sys     0m0.088s
```

- Measurement made on `login.student.lth.se` September 12, 2019.
- The type analysis for Scala is of course more complex than for C.
- Avoid compilers with this compilation speed for large source code
- Compilation speed of Scala may improve significantly in the future.
- Previous measurement was 11 s.

The Fast Scala compiler

- There is a server program `fsc` which is faster than `scalac` because it avoids some initializations.
- Clang and Common Lisp were fastest.
- `clisp` was originally written in assembler and Lisp for Atari machines but has been rewritten in portable C and Lisp.

The Scala build tool: sbt

- sbt is a tool which downloads required libraries, starts the Scala compiler and runs the program
- It will try to make a Scala program by compiling all Scala files in the current directory — so keep only one version of your program there!
- In the lab it is sufficient to type `make`. Abbreviated output:

```
$ make
./sbt run < i
[info] loading settings for project lab1 from build.sbt
[info] running main
f = 9924
t = 3.85 s
```

- It takes about 10 s to compile the 474 lines of Scala code

Scala basics: val vs var

- Writing `var a = 1`, we declare an initialized `Int` variable that we can modify.
- With `val a = 1`, a becomes readonly instead.
- The following declares an array:

```
val a = new Array[String](2);
a(0)  = "hello";
a(1)  = "there";
```

- Note that it is a that is readonly, not its elements.
- We can iterate through an array like this, for example:

```
for (s <- a) println(s);
```

- We should not declare the variable `s`.

Numbers are objects

- Consider

```
for (i <- 0 to 9) println(i);
```

- Here the zero actually is an object with the method `to`.
- In many cases a method name can be written without the dot but rather as an operator.

Companion classes

- In Java you can have `static` attributes of a class which are shared by all objects of that class.
- In Scala, you instead create a **companion class** with the keyword **object** instead of **class**:

```
object A {
  var a = 44;
}

class A {
  println("a = " + A.a);
}

object Main {
  def main(args: Array[String]) {
    val a = new A;
  }
}
```

- By default attributes are `public` in all Scala classes, but an object may access a private attribute of its companion class.

Class declarations with attributes

- You can declare a class like this:

```
class B(u: Int, var v: Int) {
  def f = u + v;
}
```

- The parameters of a constructor by default become `val` attributes of the class.
- Therefore only `v` can be modified.
- Even if you only need the parameters in the constructor, they become attributes and cheerfully consume memory for you.

Code reuse in Scala using traits

- Scala uses single inheritance as Java with the same keyword `extends`.
- Instead of Java's interfaces which only provide abstract methods, Scala has the concept of a `trait`.
- Unlike an interface, a trait can contain attributes and code, however.
- A trait is similar to a class except that the constructor cannot have parameters.

```
object Main {
  def main(args: Array[String]) {

    val a = new C(44);

    a.hello;
    a.bye;
  }
}

class A(u: Int) {
  def bye { println("bye bye with u = " + u); }
}

trait B {
  def hi { println("hello"); }
}

class C(v: Int) extends A(v) with B {
  def hello { hi; }
}
```

Lists

- The standard class `List` is singly linked and consists of a pair of data and a pointer to the next element.
- An empty list is written either as `Nil` or `List()`.
- Five (of many) methods are:
  - `::` — create a list: `val h = 1 :: 2 :: 3 :: Nil`, which means: `val h = (1 :: (2 :: (3 :: Nil)))`.
    This can also be written as `val h = List(1, 2, 3)`
  - `:::` — create a new list by concatenating two lists.

    ```
    val a = List(1, 2, 3);
    val b = List(4, 5, 6);
    val c = a ::: b;
    ```

  - `isEmpty` — boolean
  - `head` — data in first element
  - `tail` — the rest of the list starting with the 2nd element.

Reversing a List 1(2)

```
def rev[T](h:List[T]) : List[T] = {
  if (h.isEmpty)
    h;
  else
    rev(h.tail) ::: List(h.head);
}
```

- This function is generic with element type `T`.
- It's not the most efficient way to reverse a list since list concatenation must traverse the left operand list.
- This version of reverse has quadratic time complexity.
- How can we do it in linear time?

Reversing a List 2(2)

```
def rev1[T](h : List[T], q : List[T]) : List[T] = {
  if (h.isEmpty)
    q;
  else
    rev1(h.tail, h.head :: q);
}

def rev[T](h : List[T]) : List[T] = {
  rev1(h, List());
}
```

- Better.

Pattern matching in Scala

- Pattern matching means we provide a sequence of cases against which the input data is matched.
- The first case that matches the input data is executed.

```
def rev[T](xs: List[T]) : List[T] = xs match {
  case List()    => xs;
  case x :: xs1 => rev(xs1) ::: List(x);
  }
```

- The reverse of the empty list is the parameter `xs`.
- The non-empty list matches a list with at least one element, as in the second case.
- Pattern matching is used extensively in functional programming.
- We will use pattern matching when receiving actor messages.

Programming with actors in Scala

- Programming with actors is in one sense just writing another multithreaded program.
- In another sense it's completely different because you should use no locks or condition variables or the like.
- An actor is like a thread which sits and waits for a message to arrive.

Sending a message

- With actors, messages are sent to an actor — and not to a mailbox or channel as in other systems.
- A message can be a variable or a type
- Assume an actor `A` has a reference to another actor `B`, then `A` can send messages of types `C` and `D` to `B` using:

```
val e = 124
B ! C;
B ! D(42);
B ! e;
```

- The sending actor immediately continues execution without waiting for the message to arrive.
- Without a parameter, the message type should be declared as: `case object C`
- With a parameter, instead use: `case class D(x:Int)`

Receiving a message

```
case object C
case class D(x: Int)
case object Thanks

def receive = {

case D(x: Int) => println("got a D with x = " + x)

case C         => { println("got a C"); sender ! Thanks }

case e:Int     => println("got an Int e = " + e)

}
```

- The sending actor is `sender`
- Case classes and objects must start with a capital otherwise they become variables which match everything
- When there are problems, use a variable in a last case and print it

Evaluating a message

- With pattern matching on the message, the action to perform is selected.
- If there is no match, the message is discarded.
- After performing it, the actor repeats the waiting for another message,
- It's not necessarily easier to program with actors than with locks.
- For instance, you can end up with a deadlock if two actors are waiting for messages from each other.
- A message never interrupts an actor — the actor processes a message to completion before processing the next message

Message arrival

- In some actor-based systems the arrival order of messages is not specified — even for messages from the same sender.
- For Scala actors, messages sent from one actor to another always arrive in the same order as they were sent.
- When an actor has created/modified data and sent a message to another actor, that data will be visible through the cache memory in the receiving thread so there is no need to use volatile as in Java
- As we will see, there is a volatile keyword in C/C++ as well but it means something else

Sharing mutable data and data-races

- If you share a message with data that both actors may want to modify, you can have a data-race
- For preflow push, for an edge (u, v) both nodes may at some point want to modify the flow of their edge
- If in your implementation both u and v can modify the flow at the same point in time, you have a data-race
- The nodes should be actors but although the edges could be actors as well, the program will be unnecessarily complicated then so use normal objects for the edges

Sharing data between actors in general

- This is not so much relevant for the lab but still very important
- Suppose actors have vectors or other larger data which they update and sometimes share with other actors
- When an actor X asks for the vector of another actor Y and:
  - Y is fine with giving the vector to X, but
  - Y wants to continue modifying the vector, then
- instead of creating a data-race by having both X and Y access the vector we can do as follows:
  - Y can make a copy of the vector and give it to X so X can do what it wants
- If X and possibly other actors only want to read the vector we can instead do:
  - Y can make a copy of the vector that it shares with any actor as immutable data (immutable = readonly)
  - Y can have one internal vector that it modifies itself and when sufficiently done it can copy it and let future actor requests see that instead

Declaring and creating an Actor

- An actor class can be declared as:

```
class Node(val index: Int) extends Actor {
   var     e = 0;  // excess preflow
   var     h = 0;  // height
}
```

- But we do not make an array *n* of Node objects:

```
var     node: Array[ActorRef] = null
node = new Array[ActorRef](n)

for (i <- 0 to n-1)
   node(i) = system.actorOf(Props(new Node(i)), name = "v" + i)
```

- So we create an array of `ActorRef`
- A "factory" creates each actor, i.e. not simply `new Node(i)`
- We will soon see what `system` is

A Preflow actor

- It is often convenient to have a central controller which determines when an algorithm is finished.
- Two disadvantages with this approach are
  1. A single controller can be a performance bottleneck — risk in lab 1
  2. A single point of failure can cause a service to fail — ignore in lab 1
- Usually when an algorithm is parallelized to be run on a multicore computer we assume crashes are software bugs, i.e., none in our code
- In a distributed system we need to be more careful
- Such care obviously creates overhead which we want to avoid in a multicore
- When only the sink has a positive excess preflow, we can stop

The Ceberg termination criterion

- The source starts with a negative excess preflow after it has pushed to each neighbor
- Then the source possibly gets some flow back
- The excess preflow of the source increases monotonically
- That is $|e_f(s)|$ decreases monotonically
- The sink gets more and more, ie $e_f(t)$ increases monotonically
- When the $|e_f(s)| = e_f(t)$ no other node can have any excess preflow
- Then we are finished
- This is easier to detect than counting the number of nodes with excess preflow
- Invented by Nils Ceberg (who took the course 2021).
- Although this may seem obvious, current published articles on distributed preflow push don't do this

Declaring and creating the controller actor

- The controller can be declared without parameters as:

```
class Preflow extends Actor
{
        var     s                         = 0;
        var     t                         = 0;
        var     n                         = 0;
        var     edge:Array[Edge]          = null
        var     node:Array[ActorRef]      = null
}
```

- Let the sink tell the source when it has received more preflow

```
val system = ActorSystem("Main")
val control = system.actorOf(Props[Preflow], name = "control")
```

- Without parameters, it is simpler to create an actor
- Note the different syntax compared to creating the node actors

self vs this

- Suppose we have

```
class A extends Actor { }
```

- Then `this` refers to `A` and `self` to `ActorRef`
- For the controller to send a reference to itself to a node `u`, use `self`:

```
u ! Control(self) // u is an ActorRef in the array node
```

- For a node to save the controller parameter, use `this`:

```
case Control(control:ActorRef)  => this.control = control
```

Waiting for an answer

- We can create a timeout and use ? when sending a message:

```
implicit val t = Timeout(4 seconds);
val flow = control ? Maxflow
val f = Await.result(flow, t.duration)
println("f = " + f)
```

- The type of the sender is `ActorRef`

```
var     ret:ActorRef               = null
case Maxflow => {
   ret = sender           // save sender for a future reply
   node(s) ! Source(n)    // tell s it is source and has h = n
   node(t) ! Sink         // tell t it is sink
   node(s) ! Start        // tell s to do initial pushes
}
```

Stopping the actors

- The stop method can be used:

```
system.stop(control);
for (i <- 0 to n-1)
   system.stop(node(i))
system.terminate()
```

Useful functions of a Node

```
def id: String = "@" + index; // easy to grep or search for

def status: Unit = {
   if (debug) println(id + " e = " + e + ", h = " + h);
}

def enter(func: String): Unit = {
   if (debug) { println(id + " enters " + func); status }
}

def exit(func: String): Unit = {
   if (debug) { println(id + " exits " + func); status }
}

def relabel : Unit = {
   enter("relabel")
   h += 1
   exit("relabel")
}
```

Increasing parallelism when doing push

- Think through what you need to do for a push
- Do you need to wait for a reply?
- What should the reply be in that case?
- If you want to wait for a reply, can you then do multiple pushes concurrently?
- By concurrently here is meant to have sent multiple "push" messages before waiting for a reply from each
- See pdf for lab 1 for requirements to pass the lab
- If this would not be about network flow but instead chat messages to a group, you probably would want to send multiple messages concurrently

The adjacency list of a Node

```
var     edge: List[Edge] = Nil          // at Node construction

def discharge: Unit = {                  // do pushes or relabel
   var     p: List[Edge]    = Nil        // pointer into edge
   var     a:Edge           = null       // edge to work with

   p = edge
   while (p != Nil) {
      a = p.head
      p = p.tail
   }
}
```

- This is one option to iterate through the adjacency list
- Note that you *may* want to wait for a reply before doing the next edge
- Try to increase parallelism by doing multiple pushes before waiting for a reply!

What to do in the lab

- Take the smallest input, draw the graph on paper, and write down all messages that should be sent.
- When you are happy with that, you may want to try bigger input or start thinking about source code.
- You will get an incomplete program which:
    - measures execution time
    - reads the input and creates the graph
    - asks the controller to compute the maximum flow
    - stops the actors
- You need to write code for:
    - iterate through the adjacency list and do push
    - if no push could be done due to heights of neighbors, do a relabel
    - tell controller when the sink has received more excess preflow

Hint: actor decision making

- When two or more actors are involved in a decision (such as whether it is the right time to do a push) it is important to figure out who can make the final decision so that no algorithm invariants are violated
- Assume a node u asks a neighbor v for its height,
- v replies with a height lower than the height of u, and
- u pushes to v but before the push arrives, v has done a relabel (so the push should not have been done).
- How can you avoid this problem?

Hint: debugging

- Let each node have a `debug` attribute that you can enable/disable easily without editing more than one line of source code
- Start with the smallest inputs and compare the output from your program with that from the sequential C program.
- Use grep to find out what a certain node is doing:

```
forsete> cat i0
3 2 0 0
0 1 10
1 2 2
forsete> sbt run < i0 >  x
forsete> grep @1 x
```

- This should print out all lines from x in which node 1 does something since node 1 is identified with `@1`