High-level Parallel Programming: Scala on the JVM

Contents of Lecture 4

- The Scala programming language
- Parallel programming using actors

Purpose of this Lecture

- That you will understand why Scala is interesting.
- You will see examples of what is different between Scala and Java.
- You will understand how and why Scala and Java can be used in the same program.
- You will understand the key concepts of message passing using **actors**, which were first described by Carl Hewitt at MIT 1973. Essentially, an actor is a thread — however, there are certain differences as we will see.
- You will understand enough about Scala actors that you can write a parallel version of the dataflow program in Scala (to save you time for the lab, you can download an almost complete version).

Scala and the JVM

- 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.
- You can download Scala from `www.scala-lang.org`.
- It's easy to get started.
- Scala is not tied to Java byte code however and can produce output also for the Dalvik VM (for Android) and .Net.
- So what is Scala then?

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:

```
class MyClass(index: Int, name: String)
```

- The same class in Java:

```
class MyClass {
    private int index;
    private String name;

    public MyClass(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)))
```

- Which version do you prefer?

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:

```
> 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

```c
#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    0m11.884s
user    0m11.653s
sys     0m0.196s
```

- The type analysis for Scala is of course more complex than for C.
- Don't use a compiler with this compilation speed for multi million lines software — probably you don't have to because Scala programs are shorter...
- Compilation speed of Scala will probably improve significantly in the future.

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.

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 = new List(1, 2, 3)`
  - `:::` — create a new list by concatenating two lists.

```
val a = new List(1, 2, 3);
val b = new 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.
- In Scala you can have two kinds of actors which differ in how they wait for messages:
  - using `receive` syntax: one JVM thread per actor, or
  - using `react` syntax: essentially one JVM thread per CPU which handles numerous actors.

React vs Receive

- With `receive` the response time should be faster.
- With `react` you can have hundreds of thousands actors without too much overhead.
- For instance, in my version of the dataflow program, I have two actors per vertex using `react`.
- There are certain limitations when using `react` such as when waiting using `react` the actor's stack frame is discarded when it starts waiting — you cannot return to anywhere since no return address is remembered.
- Therefore no value can be returned either.
- These limitations don't make it noticably difficult to use `react`.

Sending a Message

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

```
B ! new C();
```

- The sending actor immediately continues execution without waiting for the message to arrive.
- The receiving actor may live in the same machine or in another country.

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, or does nothing which terminates it.
- 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 is not an interrupt: only when the actor has finished one message, it will evaluate the next.

Message Arrival Order

- 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.
- Consider the liveness algorithm and assume each vertex is an actor.
- Assume a vertex/actor sends a message to each of its successor vertices requesting the successor's in-set.
- Obviously, the actor cannot expect the replies will arrive in any particular order. Instead the replies can be counted to see whether all have arrived.

Copying Data or Immutable Messages

- By copying instead of sharing data you can avoid data races.
- Since copying degrades performance, and our machine has shared memory, you probably want to write your Scala program without copying data.
- Instead of copying the same data **for each message** you send, you can do as follows:
    - Create the new data.
    - When another actor wants your data, send a reference to it.
    - Forbid the receiving actors from modifying the data.
- If you need to recompute your data, do that in a different object (e.g. a newly created one).
- By having the messages immutable, there cannot be any data race.
- One of the research projects on Scala in Lausanne is to try to do static analysis to determine whether a Scala program certainly is free from data races.
- Thus, it's not guaranteed by the language but by your design — and in the future checked by the Scala compiler.

Declaring an Actor

- An actor can be declared as follows:

```
case class A();
case class B();

class Controller extends Actor {
  def act() {
    react() {
      case A() => {
        println("recieved an A " + sender);
        act();
      }

      case B() => {
        println("recieved a B and says goodbye");
      }
    }
  }
}
```

- To send two messages to a controller, we can do as follows:

```
val controller = new Controller();

controller ! new A();
controller ! new B();
```

- The messages can have parameters as any other class constructor.
- Simple and elegant syntax.
- The sending actor is available as `sender`.

Receiving a Message

- When an actor has determined which case matched, it will execute undisturbed by other incoming messages.
- The execution of a message is any Scala code.
- If the actor wants to proceed and wait for new messages, it should end with a recursive call to `act()`.
- The Scala compiler eliminates this tail recursion so new stack space will not be used.

Implementation of Actors in Scala

- Scala actors are implemented using Java threads.
- Recall, there are two constructs to receive messages:
    - `react`
    - `receive`
- An actor waiting with `receive` uses its own JVM thread.
- Instead, with `react` Java threads are shared between different actors.
- In a `receive` the actor calls the Java `wait` method and is resumed when either `notify` or `notifyAll` is called.
- At the `react` statement, the actor registers an event handler (the matching code) and the JVM thread is free to work for some other actor.
- When a message arrives to an actor waiting with `react`, an existing thread can process the message. Recall:
    - Using `react` the Scala runtime library creates as many Java threads as the machine has processors.
    - If you use `react` you can create hundreds of thousands of actors without problems.

Actor Loop

- Instead of a tail recursive call to `act()` for waiting on a new message, we can write:

```
def act() {
  loop {
    react {
      case A() => println("got A");

      case B() => println("got B");
    }
  }
}
```

- A problem with this, however, is that the loop is infinte so your actors will never terminate.
- Instead of `loop`, we can use `loopWhile` which terminates, when its condition becomes false.

Combining React Actors

- The loops we just saw are examples of **combinators** which essentially let us continue execution after a `react` without a recursive call to `act()`.
- Another combinator is `andThen` which also let's us do something after we wake up:

```
{
  react { /* ... */ }
} andThen {
  react { /* ... */ }
}
```

Syntax for Synchronous Messages

- As mentioned, the sending actor does not wait for a reply, i.e. normal messages sent with `!` are asynchronous.
- We can send synchronous message using: `!?`

```
server !? new Request("hello") match {
  case response: String => println(response);
}
```

Futures

- There is a concept of so called **futures** which simply mean you don't wait until you need to — similar to data prefetching or properly used DMA in the Cell:

```
val future = server !! new Request("hello");

/* do other things while server is working */

/* invoking the future waits until it has arrived. */
println(future());
```

- An example of using futures is to request data from multiple sources in parallel and then wait for them to arrive before working on them.

Message Timeouts

- With `receiveWithin` we can specify a maximum waiting time.
- If no other message is received before that, a `TIMEOUT` is created and received.

```
receiveWithin(1000) {
  case msg: String => /* ... */
  case TIMEOUT:    => /* ... */
}
```

Programming Hints

- You can send the actor as part of a message, using `this`.
  - The receiving actor should use it for sending a reply only.
  - If the receiving actor would invoke methods in the received thread, you can create data races which you need to avoid.
- If an actor has created some data and then sends it to an actor A, then actor A should not modify that data.
- Communicate only through messages.
- The Scala program terminates when the last actor terminates.

- You can download a subset of the file I will go through next

Akka

- You will notice Scala actors are slow
- A faster implementation of the actor model is Akka
- It has some disadvantages though, which makes it less convenient to work with
- As you saw in `Dataflow.scala` you create and work with actor objects directly in the old model
- In Akka, you essentially use `void*` and don't know which type the reference to the actor object really is. It is called `ActorRef`

Creating Actors in Akka

```
class Vertex extends Actor { /* ... */ }
object Lab2 extends App {
  val system: ActorSystem = ActorSystem("Lab2")

  /* ... */

  val cfg    = new Array[ActorRef](nvertex);
  val nsucc  = new Array[Int](nvertex);
  val system = ActorSystem("Lab2");

  val controller = system.actorOf(Props[Controller], "controller");

  for (i <- 0 until nvertex) {
    nsucc(i) = (rand.nextInt() % maxsucc).abs;
    cfg(i) = system.actorOf(Props[Vertex], "vertex"+i);
  }

  /* ... */

}
```

- You cannot refer to attributes of e.g. a Vertex actor — due to the use of `ActorRef`
- Instead send a message and request the data
- I will give you a small Akka program and related project files.
- In Lab 2 you should do at least one of the actor implementations (either the old or Akka)