SIGCSE 2013 Workshop on Scala

Release 1.0

Lewis, Läufer, and Thiruvathukal

06-May-2013 13:37:48

CONTENTS

Contents:

CHAPTER

ONE

DOWNLOADING

This tutorial is in *alpha* status. We hope it will be *beta* by the time we give our workshop!

1.1 Code Examples

- Our Mercurial repository (hosted at https://bitbucket.org/sigcse2013scala/notes) contains the source used to build this tutorial.
- Instructions for how to build this tutorial and get *all* of the code examples can be found via this page.
- Our build scripts automatically pull down all needed code from various repositories maintained by the authors. We are a *distributed* group, so our code examples are everywhere. This project is actually an attempt to aggregate the various examples in a pedagogically suitable package.

1.2 Viewing Offline

We also provide the following formats for offline reading:

http://scalaworkshop.cs.luc.edu/latex/sigcse-scala.pdf (for offline desktop/tablet viewing)

http://scalaworkshop.cs.luc.edu/epub/sigcse-scala.epub

http://scalaworkshop.cs.luc.edu/

1.3 Comments?

Please don't hesitate to contact the authors.

CHAPTER

TWO

SCALA IN CS1 AND CS2

This part of our presentation is on Google Documents: http://goo.gl/Q68fA.

CHAPTER

THREE

BUILD TOOLS FOR SCALA

In this section, we provide a brief overview of build tools for Scala. In general, build tools support the build process in several ways:

- structured representation of the project dependency graph
- management of the build lifecycle (compile, test, run)
- management of external dependencies

3.1 Why Use a Build Tool?

When using the Java or Scala command-line tools, the developer is responsible for setting the dreaded classpath. This can quickly become unwieldy when dependencies even as simple as JUnit are involved, so this is not something you would usually want to do manually.

3.2 Brief History of Build Tools

Unix make, Apache ant These tools manage the build lifecycle but not external dependencies.

Apache maven This tool also manages external dependencies but requires a lot of XML-based configuraration.

```
<dependency>
        <groupId>org.restlet</groupId>
        <artifactId>org.restlet.ext.spring</artifactId>
        <version>${restlet.version}</version>
</dependency>
```

Apache ivy, Gradle, Scala's Simple Build Tool (sbt), etc. These tools emphasize convention over configuration in support of agile development processes. sbt is compatible with ivy and designed primarily for Scala development. For example, ivy uses a structured but lighter-weight format:

```
<dependency org="junit" name="junit" rev="4.11"/>
```

3.3 Sbt

in the simplest case, sbt does not require any configuration and will use reasonable defaults.

SBT supports two configuration styles, one based on a simple subset of Scala, and one based on the full Scala language for configuring all aspects of a project.

3.3.1 build.sbt format

A minimal SBT `build.sbt` file would look like this. The empty lines are required, and the file must be placed in the project root folder.

```
name := "integration-scala"

version := "0.0.2"

scalaVersion := "2.10.1-RC1"
```

Additional dependencies can be specified either one at a time

```
libraryDependencies += "junit" % "junit" % "4.11"
```

or as a group

```
libraryDependencies ++= Seq(
  "junit" % "junit" % "4.11",
  "com.novocode" % "junit-interface" % "0.10-M2" % "test"
)
```

3.3.2 Build.scala format

Examples of more complex Scala-based project configurations can be found in these examples:

- Android click counter app
- Prime checker web service

3.4 Plugin Ecosystem

sbt includes a growing plugin ecosystem. Key examples include

sbteclipse automatically generates an Eclipse project configuration from an sbt one.

sbt-start-script generates a start script for running a Scala application outside of sbt.

3.5 Starting from Scratch

A remaining question is how to start new projects from scratch. One can start with a skeleton and modify it, or one can use maven archetypes, which are somewhat configuration-heavy and a bit hard to use.

Alternatively, Giter8 is a command-line tool that instantiates templates stored in Git repositories. Giter8 itself is based on Scala but handles templates in any language(s). For example:

```
$ g8 fxthomas/android-app

Template for Android apps in Scala

package [my.android.project]:
name [My Android Project]: my-android-project
main_activity [MainActivity]:
min_api_level [8]:
scala_version [2.10.0]:
```

```
  api_level [16]: 17
  useProguard [true]:
  scalatest_version [1.9.1]:

  Applied fxthomas/android-app.g8 in my-android-project
```

Now we have a hello world app that is ready to run.

```
  $ sbt android:package-debug
  $ sbt android:start-emulator
```

CHAPTER

FOUR

WEB APPLICATION AND SERVICES

4.1 Professional Context

Disclaimer: I am not affiliated in any form with Typesafe.

In this section, we discuss how Scala can enhance the pedagogy in certain foundational and applied advanced courses. Given that the majority of students coming out of our undergraduate and graduate programs find jobs in local and global industry, we place considerable emphasis on professional practice.

To this end, we draw on several (local and global) resources:

- Uncle Bob
- ThoughtWorks Technology Radar
- IEEE Software's Software Engineering Radio

We attempt to strike a balance among solid foundational knowledge, state-of-the-art technology, and job market demands. In particular, we have adopted the following rule for our advanced courses and research projects: *new code in Java (the language, as opposed to the platform)*.

We also strive to choose technologies that scale properly. Here, the typical startup story goes like this: Implement in RoR, oops, it doesn't scale, then redo in guess what language? There are an increasing number of good language and platform choices here, and in our experience, Scala is one of them.

The job market is also increasingly interested in Scala. Here is an interesting local internship ad.

4.2 Curricular Context

Starting in 2010, we have been incorporating Scala into several of these courses, taught mostly by Konstantin and experienced professionals serving as adjuncts.

- COMP 373/473: Advanced Object-Oriented Development, using Scala since spring 2010, planning to add Android in spring 2014
- COMP 372/471: Theory (and Practice) of Programming Languages, using Haskell (and some F#) since fall 2010
- COMP 338/442: Server-Side Software Development (focusing on web applications), using Scala with the Play! framework since fall 2010
- COMP 388/433: Web Services Development, using Scala with the spray toolkit since spring 2011
- COMP 313/413: Intermediate Object-Oriented Development (focusing on software design and architecture), using Java with Android since fall 2012, considering the addition of Scala down the road

4.3 Why Scala?

In our experience, Scala can help with the pedagogy of advanced applied courses in multiple ways:

- less boilerplate
- focus on deep concepts
- focus on good practices

In the remainder of this section, we will give an overview of web application and web service development in Scala and contrast the experience with the corresponding Java-based techniques.

4.4 Web Applications

The predominant web application frameworks targeting Scala are Lift and Play!. Typesafe has adopted the latter as part of its official stack, and we had actually started to use Play! in our course before Typesafe's decision was known.

The primary concerns in the development of web application for human users—as opposed to a web service for programmatic consumption—are

- **views**
  - presentation
  - visual styles
  - layout
  - navigation
  - i18n
- **controllers**
  - validation
  - authentication
  - dynamic behavior/interaction/state
- **models**
  - services
  - domain model
- **persistence**
  - database technologies
  - mapping domain object to persistent storage
- **testing**
  - unit testing
  - integration testing
  - functional testing
  - acceptance testing
  - performance/load testing

When using a Java-based stack, we typically address these concerns with a stack tied together by a dependency-injection framework such as Spring and an object-relational mapper (ORM) such as Hibernate, along with a MVC framework for the upper layers.

When using a Scala-based stack, we an express an equivalent architecture much more concisely and using language mechanisms instead of requiring a DI framework.

4.4.1 Examples

- Linear regression in Java with Spring, Stripes, and maven
- To-do list in Scala with Play!
- Live version of the to-do list deployed to the cloud

4.5 Web Services

As opposed to the Simple Object Access Protocol (SOAP), we will focus on representational state transfer (REST), which has emerged as the preferred approach of the broader agile community.

The implementing-rest community has put together a helpful language matrix of REST libraries, toolkits, and frameworks. Typesafe's stack does not yet include strong support for RESTful web services in the sense of a high-level DSL for request routing, which is where some of the choices in the language matrix come into the picture.

We have picked spray not only because the author is from Konstantin's home town but also because it is:

- concise
- flexible
- type-safe
- focus on HTTP and request routing
- more and more widely used and supported

Scala and Spray are supported by Heroku and several other newer APaaS cloud providers. Deploying a service to the cloud requires a simple Git commit; this makes it possible to achieve continuous delivery.

4.5.1 Examples

- Social bookmarking example based on Java with the Restlet framework
- Prime number checker based on Scala with spray

CHAPTER

FIVE

MOBILE APPLICATION DEVELOPMENT WITH ANDROID

Mobile applications backed by cloud-based RESTful services have emerged as the primary face of computing in terms of massive consumer participation. Jason Christensen described this system architecture in his OOPSLA 2009 presentation. Therefore, not only do we find it important to cover this system architecture in the curriculum, but we also see this architecture as a very effective context for teaching various important computer science topics:

- **(real-world) software architecture**
  - dependency injection principle (DIP)
  - model-view-adapter architectural pattern (MVA)
  - testability
  - etc.
- **concurrent, parallel, and distributed computing topics (PDC/TCPP/EduPar)**
  - events
  - timers
  - background threads
  - offloading tasks to the cloud
- **embedded/resource-conscious computing**
  - limitations of the device
  - capabilities of the device (numerous sensors)

Konstantin drew the inspiration to use Android instead of Swing as a context for teaching these topics from the mobile computing session at SIGCSE 2012 in Raleigh.

Furthermore, we have found the cost of switching from, say, Java Swing to Android minimal. Besides, Android matters in the real world: it is a widely used technology, and mobile app development skills are in increasing demand.

While our overall goals are similar to those of the Sofia framework project, we discuss here a language-based approach but are planning to enhance the practice of Android development in Scala through additional support classes.

As mentioned above, current and future focus has been on these courses:

- COMP 313/413: Intermediate Object-Oriented Development (focusing on software design and architecture), using Java with Android since fall 2012, considering the addition of Scala down the road
- COMP 373/473: Advanced Object-Oriented Development, using Scala since spring 2010, planning to add Android in spring 2014

5.1 Examples

The learning objectives of each example are stated in the example's readme.

- Clickcounter app
- Prime checker app
- Prime checker web service

5.2 Lab Assignment

Format Pair project

Time 10 minutes

Deliverable An enhancement of this clickcounter app that addresses at least one following additional functional requirements:

- New user story: a max (^) button as the analogous dual to the reset (0) button.
- Retaining application state during rotation (see here to find out how to rotate the emulator).

Nonfunctional requirements

- You should update the tests and the rest of the existing code accordingly.
- You should implement the `onSaveInstanceState` and `onRestoreInstanceState` application lifecycle methods (see for details. The system passes this method a Bundle in which you can save state information about the activity as name-value pairs, using methods such as `putString()` and `putInt()`.

Prize The first pair to add both new requirements wins a pair of T-shirts. If there is no such pair, the first pair to complete one of the requirements wins the T-shirts. Otherwise we will raffle off the T-shirts by throwing them into the crowd.

Memorabilia Every participant receives stickers and pins. Yay!

Acknowledgments We thank Gary Brown, sales director at Typesafe, for having provided T-shirts, stickers, and pins on such short notice!

CHAPTER

SIX

BASIC PARALLELISM USING PAR

Using parallelism solves problems more quickly than using a single-processor machine, as is the case where groups of people solve problems larger than one person can solve. But just as with groups of people, there are additional costs and problems involved in coordinating parallel processors:

- We need to have more than one processor work on the problem at the same time. Our machine must have more than one processor, and the operating system must be able to give more than one processor to our program at the same time. Kernel threads allow this in Java. An alternative approach is to have several networked computers work on parts of the problem; this is discussed in Chapters 11 and 12, "Networking" and "Coordination." of the HPJPC book by Christopher and Thiruvathukal.
- We need to assign parts of the problem to threads. This at least requires rewriting a sequential program. It usually requires rethinking the algorithm as well.
- We need to coordinate the threads so they perform their operations in the proper order, as well as avoid race conditions and deadlocks. A number of useful facilities are not provided by the standard Java language package. We provide a good collection for your use in our thread package.
- We need to maintain a reasonable grain size. Grain size refers to the amount of work a thread does between communications or synchronizations. Fine grain uses very few instructions between synchronizations; coarse grain uses a large amount of work. Too fine a grain wastes too much overhead creating and synchronizing threads. Too coarse a grain results in load imbalance and the underutilization of processors.

Two easy, practical approaches to dividing the work among several processors are executing functions in parallel and executing iterations of loops in parallel. Parallelizing loops will be presented in the next chapter. In this chapter we will discuss running subroutines in parallel.

Executing functions in parallel is an easy way to speed up computation. The chunks of code are already packaged for you in methods; you merely need to wrap runnable classes around them. Of course, there are certain requirements:

- The function must be able to run in parallel with some other computation. This usually means that there are several function calls that can run independently.
- The function must have a reasonable grain size. It costs a lot to get a thread running, and it doesn't pay off for only a few instructions.

Two kinds of algorithms particularly adaptable to parallel execution of functions are the divide-and-conquer and branch-and-bound algorithms. Divide-and-conquer algorithms break large problems into parts and solve the parts independently. Parts that are small enough are solved simply as special cases. You must know how to break a large problem into parts that can be solved independently and whose solutions can be reassembled into a solution of the overall problem. The algorithm may undergo some cost in breaking the problem into subparts or in assembling the solutions.

6.1 Example: Trapezoidal Numeric Integration

Sometimes, a program needs to integrate a function (i.e., calculate the area under a curve). It might be able to use a formula for the integral, but doing so isn't always convenient, or even possible. An easy alternative approach is to approximate the curve with straight line segments and calculate an estimate of the area from them.

This visual (courtesy of HPJPC) shows the trapezoidal method:

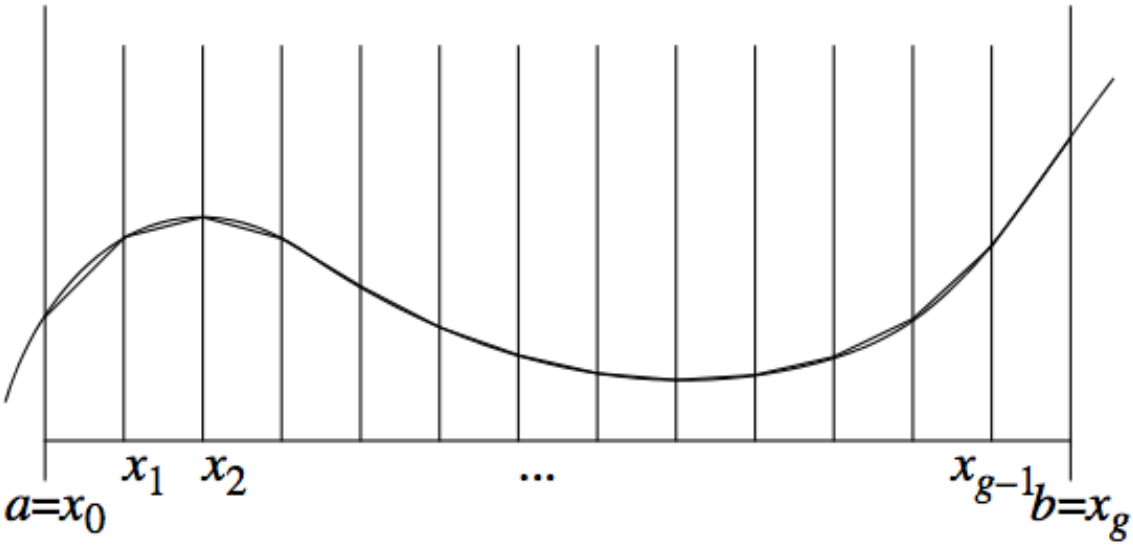

Figure 5–1 Approximating an integral with trapezoids.

This equation shows how to calculate the area.

We wish to find the area under the curve from a to b. We approximate the function by dividing the domain from a to b into g equally sized segments. Each segment is $(b-a)/g$ long. Let the boundaries of these segments be $x_0 = a$, x_1, x_1, ... , $x_g = b$. The polyline approximating the curve will have coordinates $(x_0, f(x_0))$, $(x_1, f(x_1))$, $(x_2, f(x_2))$, ..., $(x_g, f(x_g))$.

The area is then given by this formula, which sums the area of all trapezoids:

$$A = \sum_{i=1}^{g} \frac{1}{2} \cdot \frac{(b-a)}{g} \cdot (f(x_{i-1}) + f(x_i))$$

If we apply that formula unthinkingly, we will evaluate the function twice for each value of x, except the first and the last values. While the correct result would be obtained, it is inefficient and kind of defeats the purpose of going parallel. A little manipulation gives us the following:

$$A = \frac{b-a}{g} \cdot \left(\frac{f(a)}{2} + \frac{f(b)}{2} + \sum_{i=1}^{g-1} f(x_i) \right)$$

We've now reduced the problem to computing the parallel sum term, which is represented nicely in Scala.

We present three solutions in Scala:

- `integrateSequential()`: The sequential (i.e. not parallel) solution is aimed at showcasing one of the key benefits of functional programming in general. In many cases, the code closely follows the mathematics you write. Because the code is clear in terms of its intentions, we can adapt the solution for parallel execution.
- `integrateParallel()`: The parallel version shows how to get parallel speedup in Scala from the sequential one, simply by adding `par`.
- `integrateParallelGranular()`: This version shows how to combine parallel and serial execution. By allowing the user to

specify the grain size (number of rectangles to do sequentially at a time), it is possible to determine how Scala itself might be chunking the work in the `integrateParallel()` solution.

6.2 Download the Code

ZIP File https://bitbucket.org/loyolachicagocs_plsystems/integration-scala/get/default.zip

via Mercurial hg clone https://bitbucket.org/loyolachicagocs_plsystems/integration-scala

Build and Run Instructions https://bitbucket.org/loyolachicagocs_plsystems/integration-scala/overview

6.3 Going Scala!

Let's start by looking at `integrateSequential()`.

```
1   def integrateSequential(a: Double, b: Double, rectangles: Int, f: Fx): Double = {
2     val interval = (b - a) / rectangles
3     val fxValues = (1 until rectangles).view map { n => f(a + n * interval) }
```

```
4       interval * (f(a) / 2 + f(b) / 2 + fxValues.sum)
5     }
```

As would be expected, we should be write this the way we think of the problem mathematically:

- `a` and `b`: The endpoints of the integration interval
- `rectangles`: The number of rectangles to use to approximate the integral
- `f`: The function to be integrated.

In the last case, this is where Scala makes our work particularly easy by allowing us to define a proper function type as shown below:

```
1   type Fx = Double => Double
```

In our previous Java work (in the book), we had to use Java *interfaces* for this same task. While seemingly just *syntactic sugar*, Java's boilerplate is offputting to computational scientists, who would rather use C and FORTRAN where function parameters are possible. Unlike those choices, however, Scala gives us the compelling aspect of *full type checking*, which means that we can be assured of excellent performance without the complexity–and sometimes unsafe behavior–that is found in other languages.

Because it is essential to understand the sequential version (the core algorithm) before proceeding, we offer a brief explanation of each line of code, even when obvious, and how we are taking advantage of Scala (when less obvious!)

- In line 2, we calculate `interval` using the formula that was presented earlier. Scala, being pragmatic, is able to do the right thing and treat the entire expression as Double, resulting in a Double value. Scala really shines here by not requiring us to declare every val type, owing to its innate type inferencing mechanism. This results in code that is much easier to comprehend (at least we like to think so).
- In line 3, this requires a bit more explanation. Working from our equation, recall that our summation term goes from 1 to the number of rectangles minus 0. We use Scala's until to get the indices. When `rectangles` is small, this is fine. What happens when it is large? The answer depends on whether we are using eager or lazy evaluation. In mathematical/scientific computing, we often need to do a large number of iterations to get a better answer. This is where the `view` comes in. It gives us a lazy sequence that can then be mapped (also lazily) using the user-supplied function, `f`.
- In line 4, we are able to plug everything into the derived formular for calculating the trapezoidal integration. Technically,

we could have put the `fxValues` val definition in the same line of code, but having it separate makes it easier to understand for new Scala users (one of the goals for our SIGCSE workshop). More importantly, you should be able to write the code this way and not have to worry about losing performance. By setting up this lazy computation, we're able to compute the sum *on demand*. Aside from this split, the code here exactly matches the formula we derived for performing trapezoidal integration.

The sequential version of integration presented here is completely side-effect free. That is, all of the work is being done without mutating state. This means that it can immmediately be turned into something parallel in Scala, provided we know where the actual work is being done. Let's continue this exploration!

6.4 Going Parallel

The immediately preceding discussion was presented with great care, because what we are about to demonstrate illustrates how one needs to do very little work to take what is sometimes known as an *embarrassingly parallel* algorithm and make it run in parallel. The term *embarrassing* is a tad misleading. As we'll see, the results don't always follow your intuition. Furthermore, while the results can be gotten quickly, it doesn't always mean that you are getting the best results possible. For example, as well as Scala does, it still doesn't do nearly as well as our hand-coded multithreaded Java example from HPJPC. We'll say more about this later.

Let's look at the parallel version.

```
  def integrateParallel(a: Double, b: Double, rectangles: Int, f: Fx): Double = {
    val interval = (b - a) / rectangles
    val fxValues = (1 until rectangles).par.view map { n => f(a + n * interval) }
    interval * (f(a) / 2 + f(b) / 2 + fxValues.sum)
  }
```

In this version, observe that we have added the `par` method call just before generating the lazy `view`. This is the only sensible place to add `par`, because in mathematical/scientific computing, we know that most of the parallel potential is found *where the loops are*. The `1 until rectangles` is where the actual workload is being generated, so it is a natural place to suggest `par`.

6.5 Testing

The following code shows the unit tests for our various *integration* examples.

```
package edu.luc.etl.sigcse13.scala.integration

import org.junit.Test
import org.junit.Assert._
import Integration._
import Fixtures._

/**
 * Simple JUnit-based tests.
 */
class Tests {

  @Test def testSequential() {
    assertEquals(333.3, integrateSequential(0, 10, 1000, sqr), 0.1)
  }

  @Test def testParallel() {
    assertEquals(333.3, integrateParallel(0, 10, 1000, sqr), 0.1)
  }

  @Test def testParallelGranular() {
    assertEquals(333.3, integrateParallelGranular(10)(0, 10, 1000, sqr), 0.1)
  }
}
```

For the purpose of testing, we set up $f(x) = x^2$ and integrated it from 0 to 10. The value of this integral should be 333.333333333....

We can't stress enough the importance of unit tests, especially when working a sequential algorithm into a parallel one. While you are less likely to make mistakes in Scala, it is very easy when trying certain strategies to get the wrong answer. (In fact, one of them, the third, gave an incorrect answer during testing, simply because of a division error I made when computing the number of workers!)

Using the notion of a test fixture, it is possible to specify what function we wish to test without contaminating the general-purpose code we wrote with a specific function to be integrated. See below.

```
package edu.luc.etl.sigcse13.scala.integration

// begin-object-Fixtures
object Fixtures {
```

```
  def sqr(x: Double): Double = x * x
}
// end-object-Fixtures
```

6.6 Running

The tradition of scientific computing is one where users *want* and *need* to be able to run it from the command-line, often in an unattended fashion (say, on a computing cluster or network of workstations or cloud resources). The following is the main program we put together to run the integration of $f(x) = x^2$ manually. You can specify the number of rectangles, the number of times to run each of the experiments, and a grain size for testing the combined parallel/sequential version, `integrateParallelGranular()`. We'll say more about this function in our performance discussion.

```
package edu.luc.etl.sigcse13.scala.integration

import Integration._
import Fixtures.sqr

object Main extends {

  def main(args: Array[String]) {
    try {
      require { 2 <= args.length }
      val rectangles = math.max(args(0).toInt, 1000)
      val n = math.max(args(1).toInt, 1)
      val grainSize = if (args.length == 3) math.min(args(2).toInt, rectangles) else rectangles

      timedRun(rectangles, n, "sequentially", integrateSequential)
      timedRun(rectangles, n, "in parallel", integrateParallel)
      timedRun(rectangles, n, "in parallel with " + grainSize +
        " rectangles per serial worker", integrateParallelGranular(grainSize))

    } catch {
      case _: NumberFormatException => usage()
      case _: IllegalArgumentException => usage()
    }
  }

  def usage() {
    Console.err.println("usage: rectangles (>= 1000) " +
      "numberOfRuns (>= 1) [ grainSize (rectangles % grainSize == 0) ]")
  }

  def timeThis[A](s: String)(block: => A): A = {
    val time0 = System.currentTimeMillis
    val b = block
    val time1 = System.currentTimeMillis - time0
    println("Timing " + s + " = " + time1)
    b
  }

  // begin-timedRun
  def timedRun(rectangles: Int, n: Int, how: String,
    integrationStrategy: (Double, Double, Int, Fx) => Double) {
    timeThis(how) {
      print("Computing area " + how + "; now timing " + n + " iterations")
      val area: Double = (1 to n).map { _ => integrationStrategy(0, 10, rectangles, sqr) }.head
```

```
45          println("; area = " + area)
46        }
47      }
48      // end-timedRun
49    }
```

6.7 Initial Experiments with Performance

This is still being written up but will be demonstrated live.

```
1     def integrateParallelGranular(grainSize: Int)(a: Double, b: Double, rectangles: Int, f: Fx): Double
2       require { rectangles % grainSize == 0 } // can relax this later
3       val workers = rectangles / grainSize
4       val interval = (b - a) / workers
5       val fullIntegration = (0 until workers).par.view map { n =>
6         val c = a + n * interval
7         integrateSequential(c, c + interval, grainSize, f)
8       }
9       fullIntegration.sum
10    }
```

6.8 Previous Work

This example was developed as part of High-Performance Java Platform Computing by Thomas W. Christopher and George K. Thiruvathukal.

PDF of Book https://hpjpc.googlecode.com/files/HPJPC%20Christopher%20and%20Thiruvathukal.pdf


bitbucket

ZIP File https://bitbucket.org/loyolachicagocs_books/hpjpc-source-java/get/default.zip


bitbucket

via Mercurial hg clone https://bitbucket.org/loyolachicagocs_books/hpjpc-source-java

Build and Run Instructions https://bitbucket.org/loyolachicagocs_plsystems/integration-scala/overview

CHAPTER

SEVEN

PARALLELISM USING ACTORS

In this section, we're going to take a look at the Java vs. Scala way of doing things. We'll look at a guiding example that is focused on concurrent/parallel computing. This example appeared in *High Performance Java Platform Computing* by Thomas W. Christopher and George K. Thiruvathukal. We'll show how to organize a previously worked out solution that uses more explicit concurrency mechanisms from Java and how it can be reworked into a side-effect free Scala version by taking advantage of Scala's innate support for basic actor-style parallelism.

7.1 Guiding Example: Longest Common Subsequence

A longest common subsequence (LCS) of two strings is a longest sequence of characters that occurs in order in the two strings. It differs from the *longest common* substring in that the characters in the longest common subsequence need not be contiguous. There may, of course, be more than one LCS, since there may be several subsequences with the same length.

There is a folk algorithm to find the *length* of the LCS of two strings. The algorithm uses a form of dynamic programming. In divide-and-conquer algorithms, recall that the overall problem is broken into parts, the parts are solved individually, and the solutions are assembled into a solution to the overall problem. Dynamic programming is similar, except that the best way to divide the overall problem into parts is not known before the subproblems are solved, so dynamic programming solves all subproblems and then finds the best way to assemble them.

The algorithm works as follows: Let the two strings be `c0` and `c1`. Create a two-dimensional array `a`:

```
int [][] a=new int[c0.length()+1] [c1.length()+1]
```

Initialize `a[i][0]` to 0 for all `i` and `a[0][j]` to 0 for all `j`, since there are no characters in an empty substring. The other elements, `a[i][j]` , are filled in as follows:

```
for (int i=0; i <= c0.length(); i++)
   a[i][0] = 0;

for (int j=0; j <= c1.length(); j++)
   a[0][j] = 0;
```

We will fill in the array so that `a[i][j]` is the length of the LCS of `c0.substring(0,i)` and `c1.substring(0,j)`. Recall that `s.substring(m,n)` is the substring of `s` from position `m` up to, but not including, position `n`.:

```
for (i=1; i <= c0.length(); i++)
   for (j=1; j <= c1.length(); j++)
      if (c0.charAt(i-1) == c1.charAt(j-1))
         a[i][j]=a[i-1][j-1]+1;
      else
         a[i][j]=Math.max(a[i][j-1],a[i-1][j]);
```

The above shows a *traditional* imperative solution that constructs a result matrix comprising the results of the LCS.

So how exactly does this method work?

Element `a[i-1][j-1]` has the length of the LCS of string `c0.substring(0,i-1)` and `c1.substring(0,j-1)`. If elements c0.charAt(i-1) and c1.charAt(j-1) are found to be equal, then the LCS can be extended by one to length `a[i-1] [j-1]+1`. If these characters don't match, then what? In that case, we ignore the last character in one or the other of the strings. The LCS is either `a[i][j-1]` or `a[i-1][j]`, representing the maximum length of the LCS for all but the last character of `c1.substring(0,j-1)` or `c0.substring(0,i-1)`, respectively.

The chore graph from [HPJPC] for calculation of the LCS is shown in the following figure.

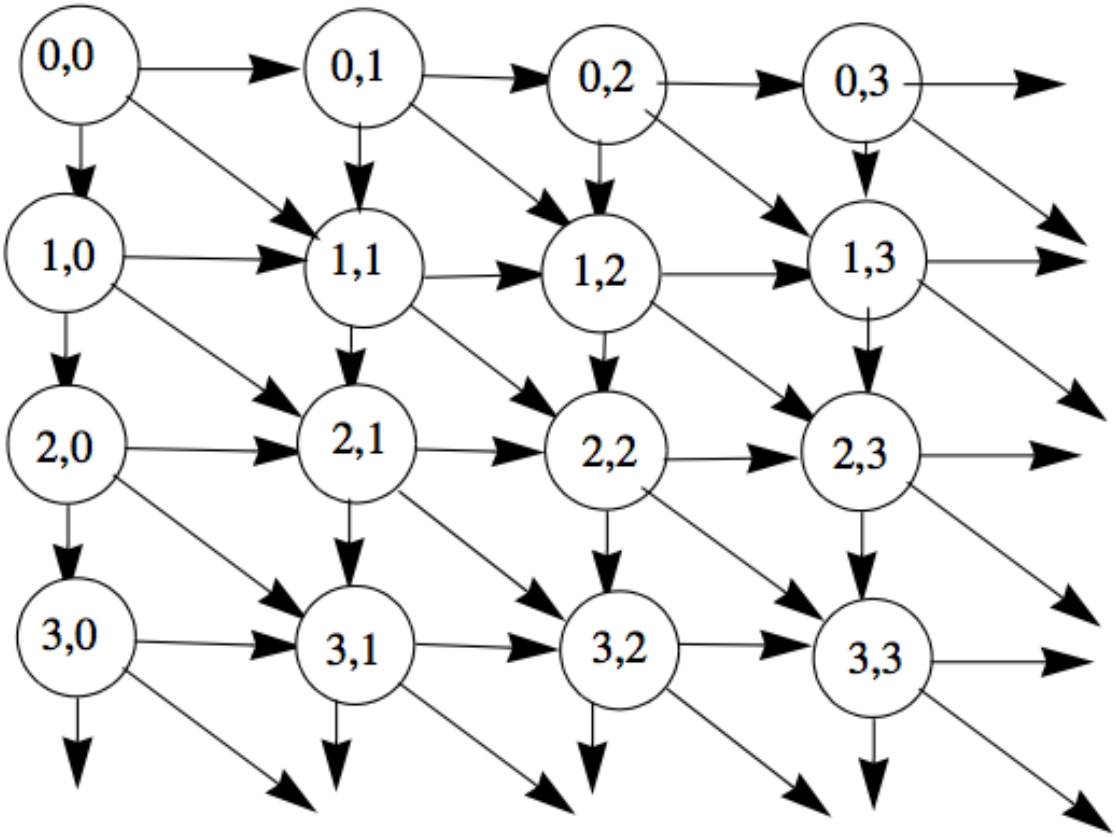

Figure 6–8 Chore graph for LCS.

Any order of calculation that is consistent with the dependencies is permissible. Two are fairly obvious: (1) by rows, top to bottom, and (2) by columns, left to right.

Another possibility is along diagonals. All `a[i][j]`, where `i+j==m` can be calculated at the same time, for `m` stepping from 2 to `c0.length()+c1.length()`. . Visualizing waves of computation passing across arrays is a good technique for designing parallel array algorithms. It has been researched under the names systolic arrays and wavefront arrays [Wavefront].

The following figure shows how a wavefront computation progresses.

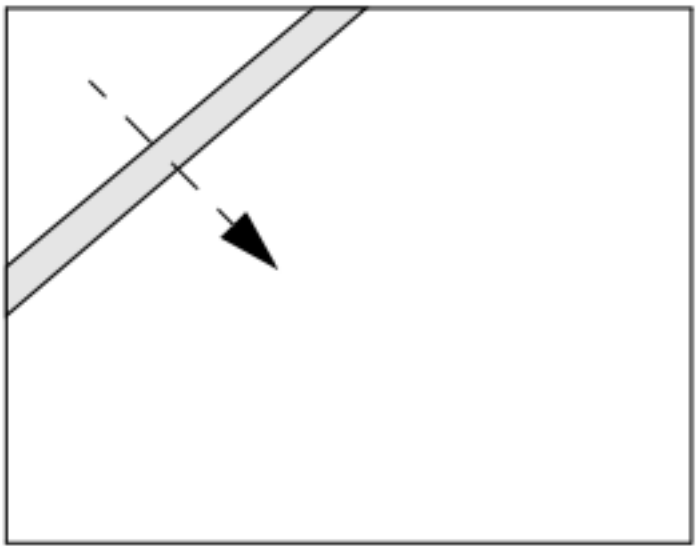

Figure 6–9 Wavefront calculation of LCS. Shading represents elements being calculated.

7.2 Java Threads Implementation

ZIP File https://bitbucket.org/loyolachicagocs_books/hpjpc-source-java/get/default.zip

via Mercurial hg clone https://bitbucket.org/loyolachicagocs_books/hpjpc-source-java

Build and Run Instructions https://bitbucket.org/loyolachicagocs_books/hpjpc-source-java

Our Java implementation (see LCS.java) of the LCS algorithm divides the array into vertical bands and is pictured in Each band is filled in row by row from top to bottom. Each band (except the leftmost) must wait for the band to its left to fill in the last element of a row before it can start can start filling in that row. This is an instance of the producer-consumer releationship.

The following figure shows how our Java solution organizes the work in bands:

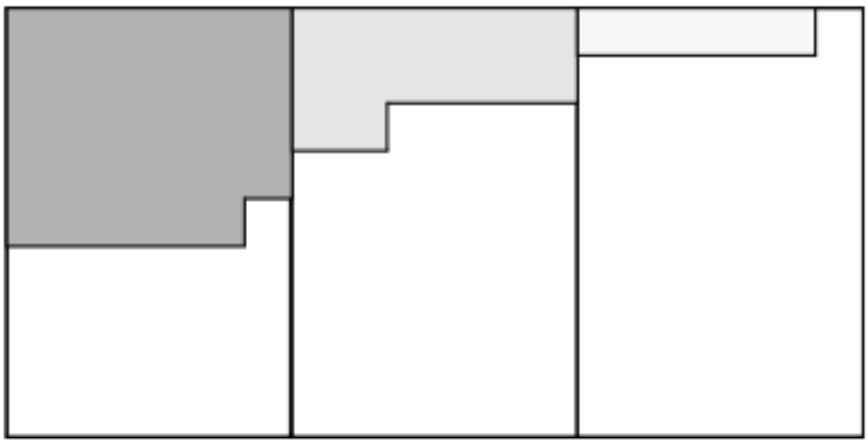

Figure 6–10 LCS calculated in vertical bands. Shading represents elements that have been calculated.

LCS class

```
    int numThreads;
    char[] c0;
    char[] c1;
    int[][] a;
    Accumulator done;
```

```
    public LCS(char[] c0, char[] c1, int numThreads) {
       this.numThreads = numThreads;
       this.c0 = c0;
       this.c1 = c1;
       int i;
       done = new Accumulator(numThreads);

       a = new int[c0.length + 1][c1.length + 1];

       Semaphore left = new Semaphore(c0.length), right;
       for (i = 0; i < numThreads; i++) {
          right = new Semaphore();
          new Band(startOfBand(i, numThreads, c1.length), startOfBand(i + 1,
                numThreads, c1.length) - 1, left, right).start();
          left = right;
       }
    }
```

```
    public LCS(String s0, String s1, int numThreads) {
       this(s0.toCharArray(), s1.toCharArray(), numThreads);
    }
```

```
    int startOfBand(int i, int nb, int N) {
       return 1 + i * (N / nb) + Math.min(i, N % nb);
    }
```

```
   public int getLength() {
      try {
         done.getFuture().getValue();
      } catch (InterruptedException ex) {
      }
      return a[c0.length][c1.length];
   }
```

Band internal class (does the work)

```
      int low;
      int high;
      Semaphore left, right;

      Band(int low, int high, Semaphore left, Semaphore right) {
         this.low = low;
         this.high = high;
         this.left = left;
         this.right = right;
      }
```

Actual Runnable body

```
      public void run() {
         try {
            int i, j, k;
            for (i = 1; i < a.length; i++) {
               left.down();
               for (j = low; j <= high; j++) {
                  if (c0[i - 1] == c1[j - 1])
                     a[i][j] = a[i - 1][j - 1] + 1;
                  else
                     a[i][j] = Math.max(a[i - 1][j], a[i][j - 1]);
               }
               right.up();
            }
            done.signal();
         } catch (InterruptedException ex) {
         }
      }
```

Main

```
      public static void main(String[] args) {
         if (args.length < 2) {
            System.out.println("Usage: java LCS$Test1 string0 string1");
            System.exit(0);
         }
         int nt = 3;
         int i;

         String s0 = args[0];
         String s1 = args[1];
         System.out.println(s0);
         System.out.println(s1);
         long t0 = System.currentTimeMillis();
         LCS w = new LCS(s0, s1, nt);
         long t1 = System.currentTimeMillis() - t0;
         System.out.println(w.getLength());
```

```
17            System.out.println("Elapsed time " + t1 + " milliseconds");
18        }
```

7.3 Scala Actors Implementation

ZIP File https://bitbucket.org/loyolachicagocs_plsystems/lcs-systolicarray-scala/get/default.zip

via Mercurial hg clone https://bitbucket.org/loyolachicagocs_plsystems/lcs-systolicarray-scala

Build and Run Instructions https://bitbucket.org/loyolachicagocs_plsystems/lcs-systolicarray-scala

Trait

```
1  trait SystolicArray[T] {
2    def start(): Unit
3    def put(v: T): Unit
4    def take(): T
5    def stop(): Unit
6  }
```

The entire SystolicArray implementation is here:

```
1  trait SystolicArray[T] {
2    def start(): Unit
3    def put(v: T): Unit
4    def take(): T
5    def stop(): Unit
6  }
```

Logging

// begin-object-logger

```
1    private object logger {
2      private val DEBUG = false
3      // use call-by-name to ensure the argument is evaluated on demand only
4      def debug(msg: => String) { if (DEBUG) println("debug: " + msg) }
5      // add other log levels as needed
6    }
```

Apply

```
1    def apply[T](rows: Int, cols: Int, f: Acc[T]): SystolicArray[T] = {
2      require { 0 < rows }
3      require { 0 < cols }
4      val result = new SyncVar[T]
5      lazy val a: LazyArray[T] = Stream.tabulate(rows, cols) {
```

```
      (i, j) => new Cell(i, j, rows, cols, a, f, result)
    }
    val root = a(0)(0)
    new SystolicArray[T] {
      override def start() = root.start()
      override def put(v: T) { root ! ((-1, -1) -> v) }
      override def take() = result.take()
      override def stop() { root ! Stop }
    }
  }
```

The internal Cell class, used to represent the cells of the Systolic Array (generally).

```
  protected class Cell[T](row: Int, col: Int, rows: Int, cols: Int, a: => LazyArray[T],
      f: Acc[T], result: SyncVar[T]) extends Actor { self =>

    require { 0 <= row && row < rows }
    require { 0 <= col && col < cols }

    logger.debug("creating (" + row + ", " + col + ")")

    override def act() {
      logger.debug("starting (" + row + ", " + col + ")")
      var start = true
      loop {
        logger.debug("waiting  (" + row + ", " + col + ")")
        barrier(if (row == 0 || col == 0) 1 else 3) { ms =>
          if (start) { startNeighbors() ; start = false }
          propagate(ms)
        }
        // one-way message: anything below here is skipped!
      }
    }

    protected def barrier(n: Int)(f: Map[Pos, T] => Unit): Unit =
      barrier1(n)(f)(Map.empty)

    protected def barrier1(n: Int)(f: Map[Pos, T] => Unit)(ms: Map[Pos, T]): Unit = {
      if (n <= 0)
        f(ms)
      else
        react {
          case Stop => stopNeighbors() ; exit()
          case (p: Pos, v: T) => barrier1(n - 1)(f)(ms + (p -> v))
        }
      // one-way message: anything after react is skipped!
    }

    protected def applyToNeighbors(f: Cell[T] => Unit) {
      if (row < rows - 1)                   f(a(row + 1)(col    ))
      if (col < cols - 1)                   f(a(row    )(col + 1))
      if (row < rows - 1 && col < cols - 1) f(a(row + 1)(col + 1))
    }

    protected def startNeighbors() { applyToNeighbors { _.start() } }

    protected def propagate(ms: Map[Pos, T]) {
      val r = f((row, col), ms)
      val m = (row, col) -> r
```

```
47        logger.debug("firing   " + m)
48        if (row < rows - 1)                     a(row + 1)(col    ) ! m
49        if (col < cols - 1)                     a(row    )(col + 1) ! m
50        if (row < rows - 1 && col < cols - 1)   a(row + 1)(col + 1) ! m
51        if (row >= rows - 1 && col >= cols - 1) result.put(r)
52      }
53
54      protected def stopNeighbors() { applyToNeighbors { _ ! Stop } }
55    }
```

This is used for autowiring the quadrant from where messages are being fired (from). It is an example of how Scala can help us avoid making mistakes. In scientific computations, subscript problems are common.

```
1   implicit class Helper[T](ms: Map[Pos, T]) {
2     def north     (implicit current: (Pos, T)): T = ms.get((current._1._1 - 1, current._1._2    )).get
3     def west      (implicit current: (Pos, T)): T = ms.get((current._1._1    , current._1._2 - 1)).get
4     def northwest(implicit current: (Pos, T)): T = ms.get((current._1._1 - 1, current._1._2 - 1)).get
5   }
```

This is used to autowire the left and top edges of the array. Although easy enough to check, it can be difficult to remember which subscript is row or column. Scala again makes this very easy for us. As we'll see, it also helps to make the user function self-documenting (literate).

```
1   implicit class PosHelper(p: Pos) {
2     def north = p._1 - 1
3     def west = p._2 - 1
4     def isOnEdge = p._1 == 0 || p._2 == 0
5   }
```

Wrapping it up with object lcs...

```
1   object lcs {
2     import SystolicArray._
3
4     def f(c0: String, c1: String)(p: Pos, ms: Map[Pos, Int]) = {
5       implicit val currentPosAndDefaultValue = (p, 0)
6       if (p.isOnEdge)
7         0
8       else if (c0(p.north) == c1(p.west))
9         ms.northwest + 1
10      else
11        math.max(ms.west, ms.north)
12    }
13
14    def apply(c0: String, c1: String): Int = {
15      val root = SystolicArray(c0.length + 1, c1.length + 1, f(c0, c1))
16      root.start()
17      root.put(1)
18      root.take
19    }
20  }
```

Setting up the text fixtures...

```
1   object Fixtures {
2     import SystolicArray._
3
4     val c0 = "Now is the time for all great women to come to the aid of their country"
5
```

```
  val c1 = "Now all great women will come to the aid of their country"

  val f1 = (p: Pos, ms: Map[Pos, Int]) => ms.values.sum

  val f2 = (p: Pos, ms: Map[Pos, Int]) => {
    implicit val currentPosAndDefaultValue = (p, 0)
    ms.north + ms.northwest + ms.west
  }

  val f3 = lcs.f(c0, c1) _

  // "bare-metal" version of lcs, does not run significantly faster
  val f4 = (p: Pos, ms: Map[Pos, Int]) => {
    if (p._1 == 0 || p._2 == 0)
      0
    else if (c0(p._1 - 1) == c1(p._2 - 1))
      ms.get((p._1 - 1, p._2 - 1)).getOrElse(0) + 1
    else
      math.max(
        ms.get((p._1 - 1, p._2)).getOrElse(0),
        ms.get((p._1, p._2 - 1)).getOrElse(0))
  }
}
```

Testing...

```
class Tests {

  @Test def testSum() {
    val root = SystolicArray(3, 3, f1)
    root.start()
    root.put(1)
    assertEquals(13, root.take())
  }

  @Test def testSample() {
    assertEquals(53, lcs(c0, c1))
  }
}
```

CHAPTER

EIGHT

INDICES AND TABLES

- *genindex*
- *search*

BIBLIOGRAPHY

[Wavefront] 8. (a) Kung, C. E. Leiserson: Algorithms for VLSI processor arrays; in: C. Mead, L. Conway (eds.): Introduction to VLSI Systems; Addison-Wesley, 1979

[HPJPC] Thomas W. Christopher and George K. Thiruvathukal, *High Performance Java Platform Computing*, Prentice Hall PTR and Sun Microsystems Press, 2000.

INDEX