Гуляев Г.М.

Современные технологии программирования (часть 2)

Лекция 7. Конкурентное программирование

Курс лекций для студентов АлтГТУ

Параллеллизм в Scala

- Конкурентное (параллельное) программирование в тех языках программирования, где такое вообще возможно в полной мере, как правило, достаточно сложно.
- В java для этого используется класс **Thread** (потоки). В языке Erlang используется иная модель - акторы (или актеры). Scala поддерживает обе эти модели.
- Использовать параллельные потоки в Scala просто:

```
import scala.concurrent.ops._
spawn { println("Привет из параллельного потока") }
println("Привет из основного потока")
```

- **spawn** создает параллельный основному поток. Не требуется создавать новый объект **Thread** и в нем реализовывать метод **run** как это было в java.
- Для параллельных вычислений есть фьючеры (**future**). Это зарезервированная программная конструкция для еще неизвестного результата конкурентного вычисления.

Параллеллизм в Scala

- **Пример использования фьючеров:**

```
import concurrent.future
import concurrent.ExecutionContext.Implicits.global
var a = future {10 * 20}
var b = future {12 * 99}
a.value.get.get * b.value.get.get // 237600
```

- **Еще пример:**

```
import concurrent.duration.Duration
import concurrent.{Await, ExecutionContext, Future, future}
import ExecutionContext.Implicits.global
object Test {
  def main(args: Array[String]) {
    println("До вызова future")
    val s = "Привет из"
    val f = future {s + " future!"}
    f onSuccess {case v => println(v)}
    Await.ready(f, Duration.Inf) // Thread.sleep(500)
    println("После вызова future")
  }
}
```

Параллеллизм в коллекциях

- **Параллельные вычисления в коллекциях (метод par):**

```
case class Data(val a: Int = 0, val b: Int = 0)

object Test {
  def f(data: Data) = { // функция задержки и вычисления
    Thread.sleep(10)
    data.a + data.b
  }

  def genData(n: Int) = { // генерация списка List[Data]
    val r = scala.util.Random
    (for { i <- 0 to n } yield Data(r.nextInt(10000), r.nextInt(10000))).toList
  }

  def computeSeq(list: List[Data]) = { // последовательное вычисление
    list.map{f}.max
  }
```

Параллелизм в коллекциях

- **Продолжение примера:**

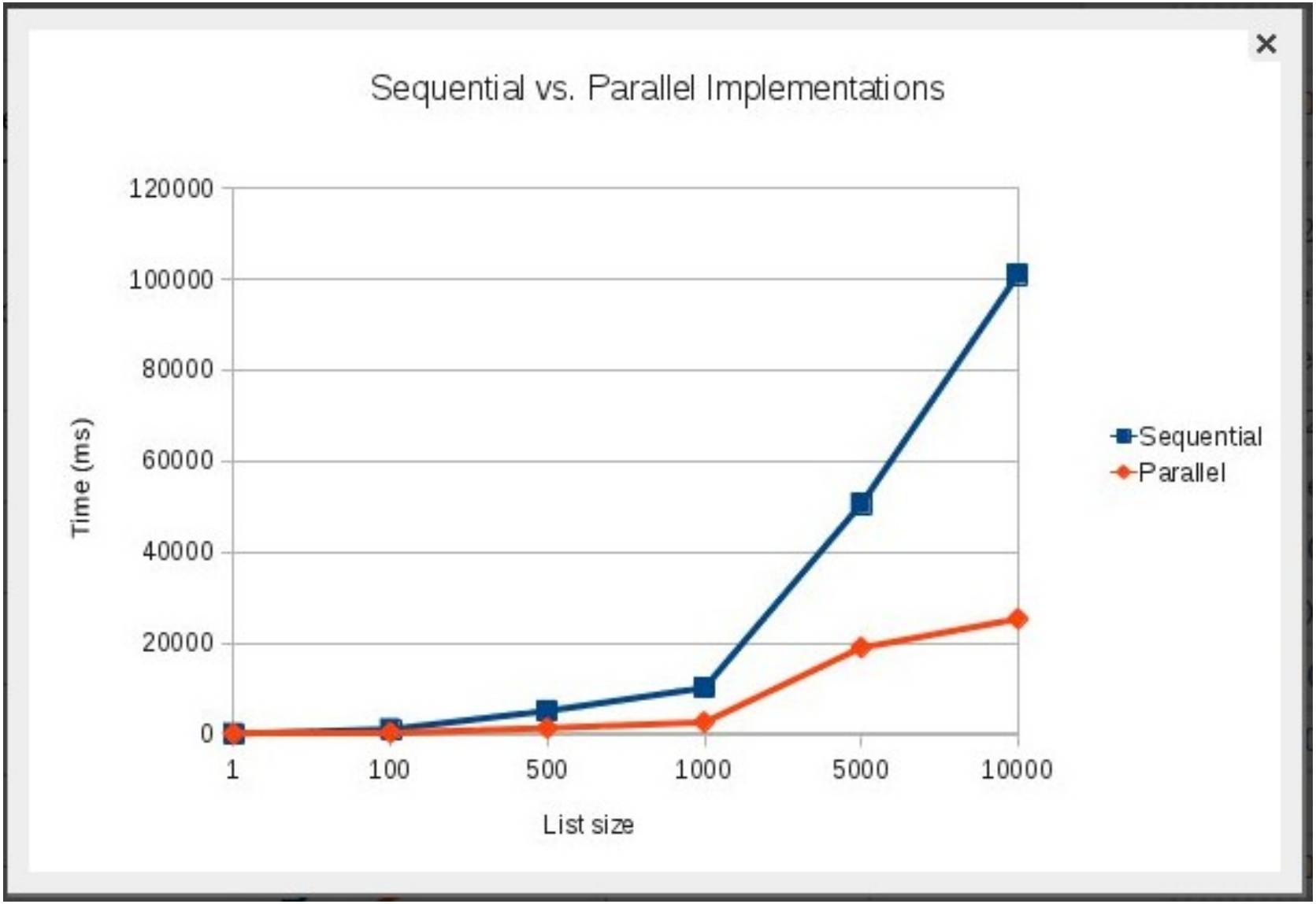
```
def computePar(list: List[Data]) = { // параллельное вычисление
  list.par.map{f}.par.max
}

def main(args: Array[String]) {
  val list = genData(args(0).toInt)
  def time = System.currentTimeMillis
  var t = time
  computeSeq(list)
  println("Последовательное: " + (time-t) + " (ms)")
  t = time
  computePar(list)
  println("Параллельное: " + (time-t) + " (ms)")
 }
}
```

- **Вычисления (sbt): > run 100**

  Последовательное: 1019 (ms)

  Параллельное: 267 (ms)

Параллеллизм в коллекциях



Параллеллизм в коллекциях

- В основе параллельности лежат специальные классы коллекций:

```
List(1,2,3,4,5).par.filter {x => x % 2 == 0 } // ParVector(2, 4)
```

- Использование параллельности в коллекциях дает существенное преимущество уже на 4 ядрах. Для большего числа процессоров преимущество будет только возрастать.
- Большинство методов коллекций (**map**, **count**, **filter**, …) поддерживают параллельность.
- Исключение - агрегатные функции (**foldLeft**, …). Они не параллельны по определению.

Акторы

- Акторы появились в работах по искусственному интеллекту в 1973 году. Позднее они пришли в языки программирования (Erlang, Io, …).
- Актор - это объект, который принимает и обрабатывает сообщения.
- Порядок сообщений не имеет значения для актора, хотя в некоторых реализациях (например, в Scala) есть понятие очереди сообщений.
- Отсутствие предпочтения в выборе сообщения для обработки дает возможность Актору работать с ними параллельно.
- Акторы получают сообщения, обрабатывают их, отправляют сообщения другим актерам а также могут запускать других акторов.
- Программу на Scala можно написать в виде множества акторов. Они легковесны и, по определению, обеспечивают многопоточность приложения.

Акторы в Scala

- Создать актора - это унаследовать свой класс от трейта **Actor** и реализовать в нем метод **act**:

```
import scala.actors.Actor
class MyActor extends Actor {
  def act() {
    println("Работаем!")
  }
}
val act = new MyActor
act.start
```

- А можно то же сделать короче, используя метод **actor**:

```
import scala.actors.Actor
import scala.actors.Actor._
val act = actor { println("Работаем!") }
```

- Во втором случае не требуется создавать класс и метод **act**, создавать объект и стартовать - все это уже встроено в метод **actor**.

Акторы в Scala

- **Получение и обработка сообщений:**

```
import scala.actors.Actor
import scala.actors.Actor._
val act = actor {
  loop {
    receive {
      case s: String => println("Получили String: " + s)
      case i: Int => println("Получили Int: " + i)
      case _ => println("Получили что-то еще")
    }
  }
}
```

- **В бесконечном цикле loop ведется прием и обработка сообщений. Передача сообщения актору происходит при помощи оператора !:**

```
act ! "привет вам" // Получили String: привет вам
act ! 235 // Получили Int: 235
act ! 2.35 // Получили что-то еще
```

Акторы в Scala

- **Обмен сообщениями:**

```
import scala.actors.Actor
import scala.actors.Actor._
var visitor = actor {}
val guard = actor { // охранник
  loop{
    react {
      case "свои" => {println("пароль"); visitor ! "пароль"}
      case "гроза" => {println("залив. Проходите"); visitor ! "залив. Проходите"}
      case _ => {println("Стой! Кто идет?"); visitor ! "Стой! Кто идет?"}
    }
  }
}
visitor = actor { { // посетитель
  loop{
    react {
      case "Стой! Кто идет?" => {println("свои"); guard ! "свои"}
      case "пароль" => {println("гроза"); guard ! "гроза"}
      case "залив. Проходите" => {println("ок"); exit}
    }
}}
```

Акторы в Scala

- **Обмен сообщениями:**

```
guard ! "" // шорох в кустах

Стой! Кто идет?
свои
пароль
гроза
залив. Проходите
ок
```

- **Вместо встроенной в Scala реализации акторов, можно использовать библиотеку akka:**

```
import akka.actor.Actor
import akka.actor.ActorSystem
import akka.actor.Props
```

Спасибо за внимание !

www.altailand.ru