Гуляев Г.М.
Современные технологии программирования

Лекция 8. Конкурентное программирование

Центр компетенции СПО Алтайского края

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.
- До версии 2.10 параллельные потоки создавались так:

```
import scala.concurrent.ops._
def test = { 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.
- Теперь пакет **ops** и **spawn**, в частности, являются устаревшими (deprecated) и вместо них рекомендовано использовать **scala.concurrent.Future**

Параллелизм в Scala

- Для параллельных вычислений используются фьючеры (**scala.concurrent.future**). Это зарезервированная программная конструкция для еще неизвестного результата конкурентного вычисления.
- **future { ... }** - выполняет асинхронное вычисление в другом потоке и возвращает значение. Результаты могут обрабатываться через функции обратного вызова **OnSuccess**, **OnFailure**, **OnComplete**.
- Примеры:

```
import concurrent.future
import concurrent.ExecutionContext.Implicits.global
val a = future { 10 * 20 }
a.value // Some(Success(200))
a.value.get // Success(200)
a.value.get.get // 200
val b = future { 1/0 }
b.value // Some(Failure(java.lang.ArithmeticException: / by zero))
b.value.get // Failure(java.lang.ArithmeticException: / by zero)
```

Параллеллизм в Scala

- **Примеры:**

```
 val hi = future {
   Thread.sleep(10000)
   "Привет!"
 }
 hi onSuccess {
   case mes => println(mes)
 }

val zero = future {
   Thread.sleep(10000)
   1 / 0
}
zero onFailure {
  case e: ArithmeticException => println("Ошибка "+e)
}
```

Параллеллизм в Scala

- Пример использования одного future в другом:

```
val first = future {
  Thread.sleep(10000)
  "\nЯ Вас люблю"
}
val then = first map {
  case mes => {
    println(mes)
    Thread.sleep(1000)
    "Я пошутил"
  }
}

then onSuccess {
  case s => println(s)
}
```

Параллеллизм в Scala

- Пример зависимости от нескольких future:

```
val first = future {
  Thread.sleep(500)
  scala.util.Random.nextInt(10)
}
val second = future {
  Thread.sleep(500)
  scala.util.Random.nextInt(10)
}
val less = for {
   f <- first
   s <- second
} yield f < s
less onSuccess {
  case b => println("Первый "+(if (b) "меньше" else "больше"))
}
```

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
    (1 to n).map(i => 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)

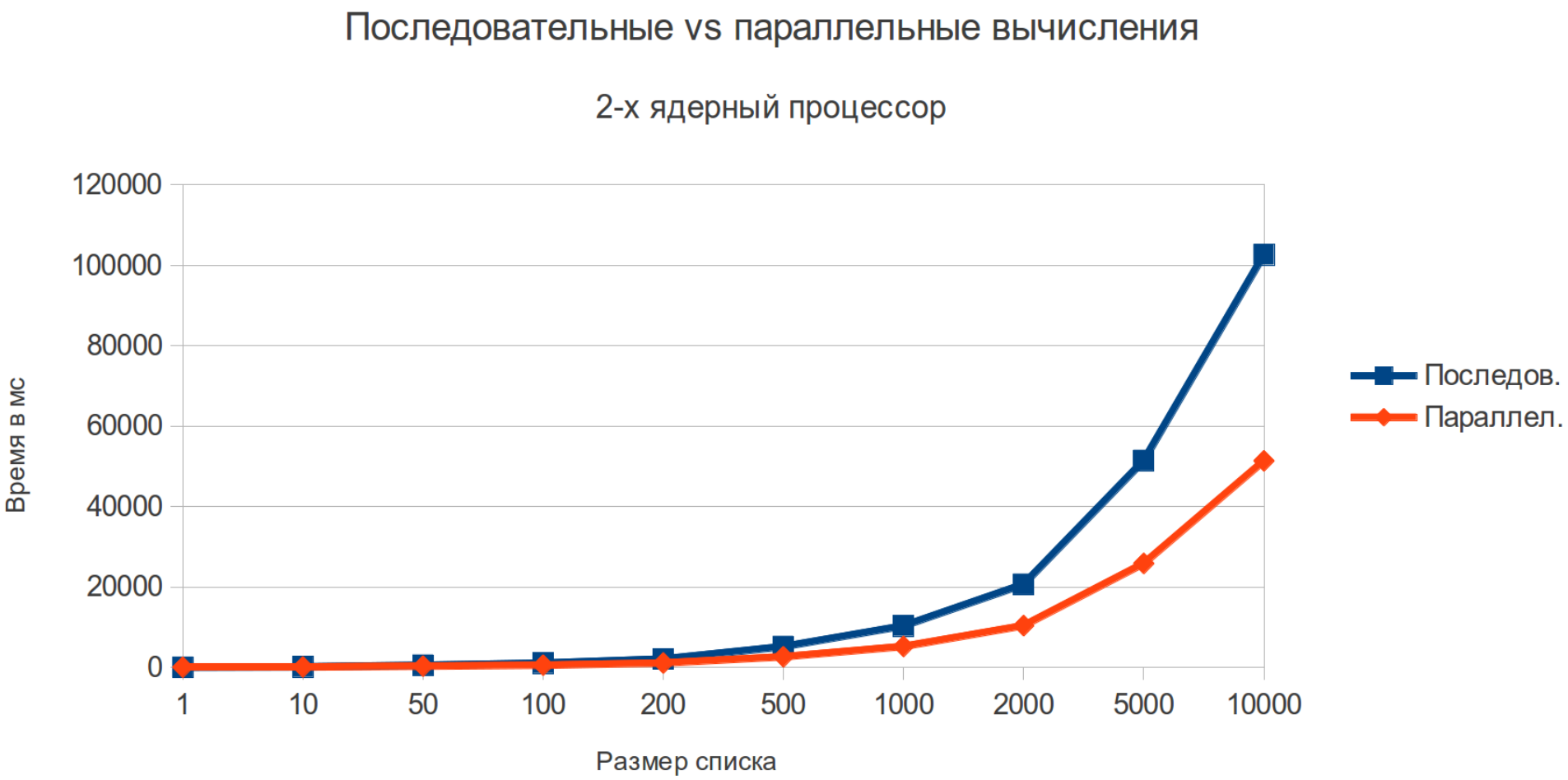
Параллеллизм в коллекциях

- **Результаты:**

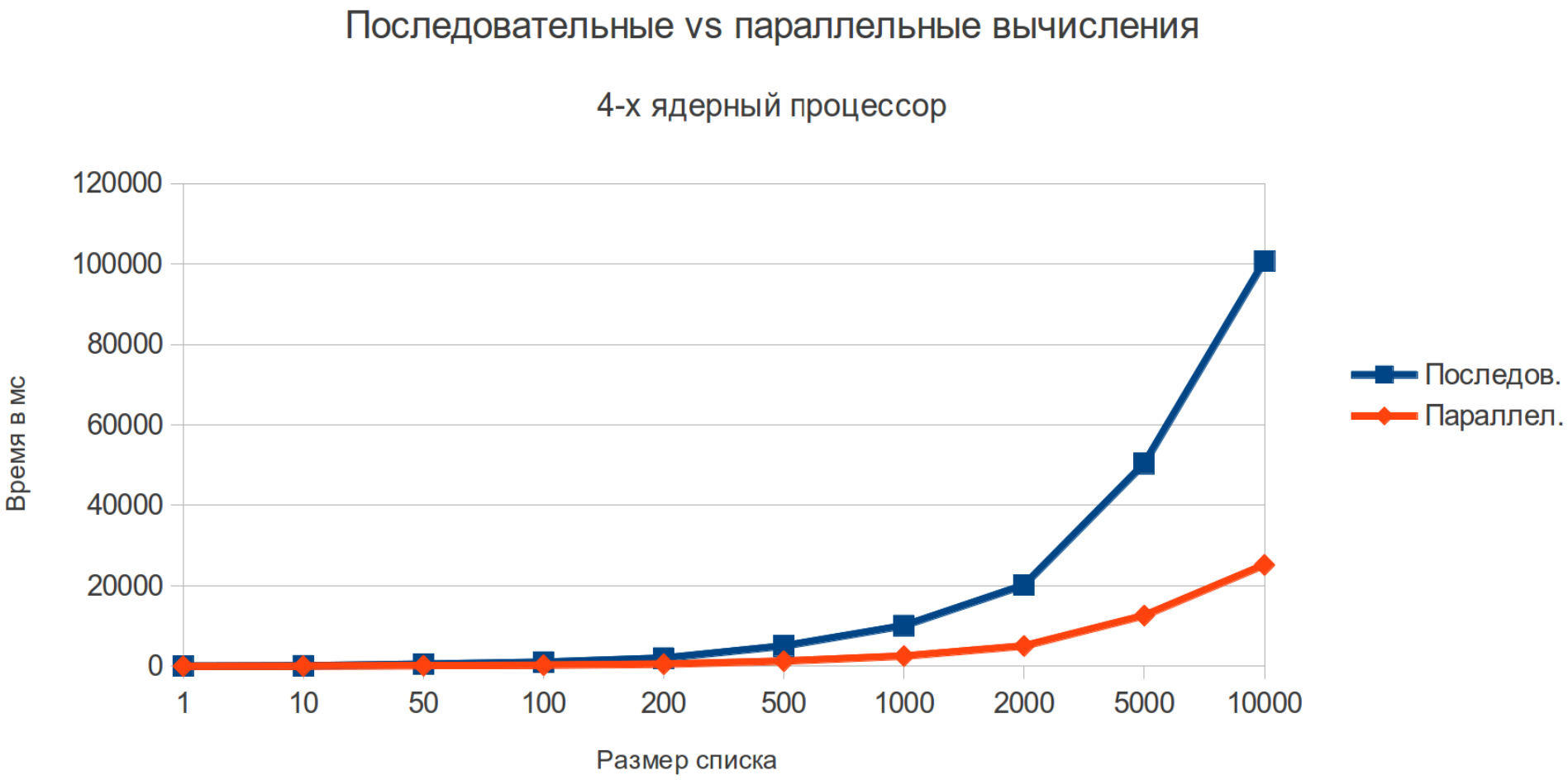
	2-х ядерный процессор									
n	1	10	50	100	200	500	1000	2000	5000	10000
Последов.	12	105	520	1032	2064	5154	10308	20643	51437	102626
Параллел.	40	90	300	556	1080	2627	5209	10362	25834	51353
	4-х ядерный процессор									
n	1	10	50	100	200	500	1000	2000	5000	10000
Последов.	11	101	504	1008	2015	5035	10072	20141	50352	100698
Параллел.	11	32	134	265	507	1268	2524	5039	12593	25179

- **Скорость последовательных вычислений практически не зависит от числа ядер**
- **Параллельные вычисления практически в k раз быстрее последовательных, где k — число ядер**

Параллеллизм в коллекциях



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 акторам 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
```

Задания для самостоятельной работы

- **Напишите программу генерирующую массив из n случайных вещественных чисел и вычисляющую среднее значение и дисперсию. Используя коллекции scala, реализовать параллельные вычисления среднего и дисперсии. Реализовать вычисления тех же параметров при помощи нескольких акторов (разбивая данные на части и обмениваясь уже вычисленными результатами). Провести сравнение быстродействия этих трех решений для различных значений n.**

Спасибо за внимание !

www.altailand.ru