Reading: Chapter 07 of [Chiusano and Bjarnason 2014]

Purely Functional Parallelism

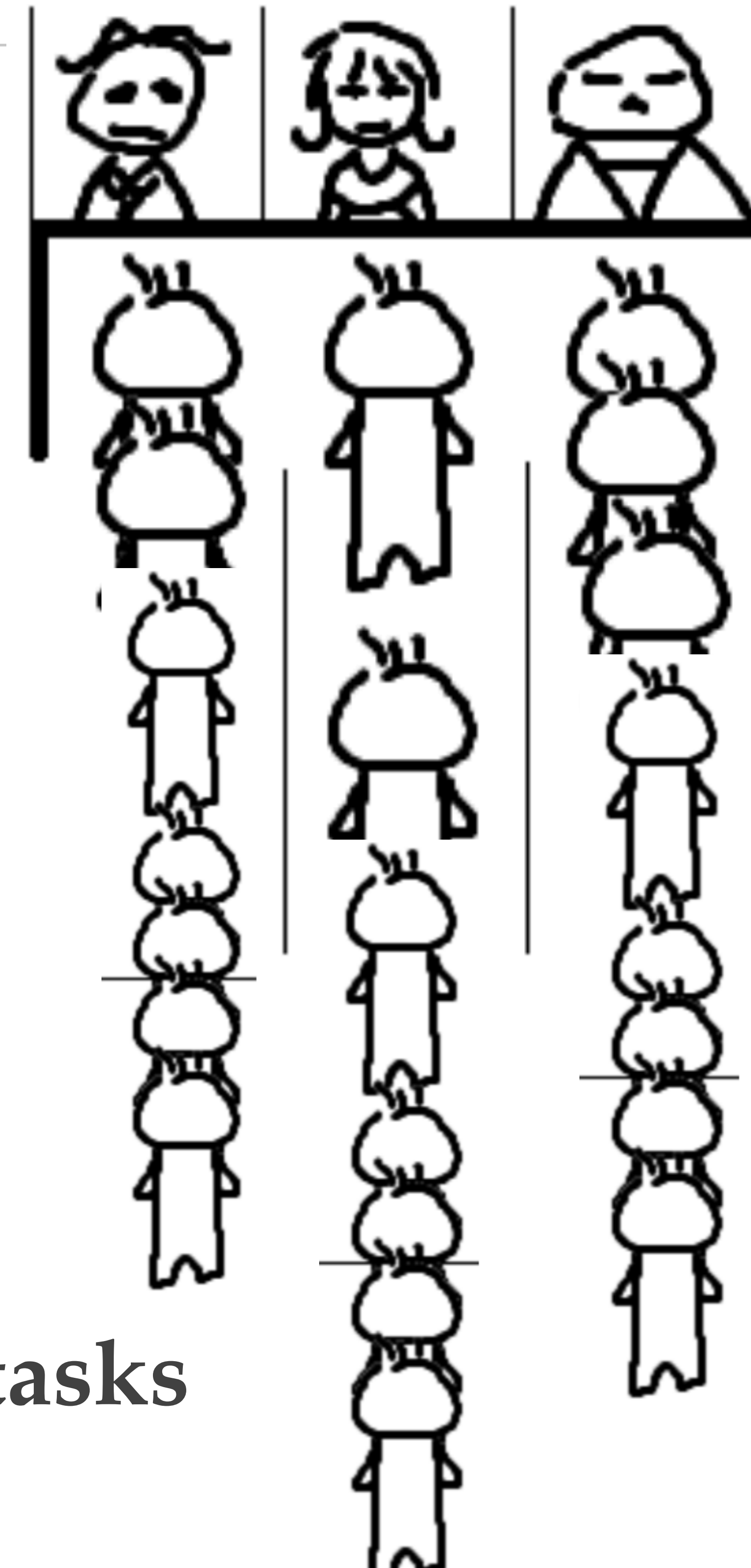
Advanced Programming

Oct 31, 2019

Andrzej Wasowski & Zhoulai Fu

IT University of Copenhagen

Why parallelism



Because of:

(1)significantly more computational tasks

(2)Significantly more CPU

Why functional parallelism

- Many parallelism models involve the using of a shared variable, and side effects
- And side effects can be hard to reason, easy to get wrong

blue thread	other threads
`x = …;` `done = true;`	`while (!done) {}` `… = x;`

The code works reliably with a dumb compiler, but can cause deadlock for many modern compilers!

Reason: Modern compilers compile “While (! done){}” to tmp=done; while (!tmp){}

Clarification 1 : concurrency vs parallelism:

- A minor note about concurrency vs parallelism:
    - Concurrency describes a *problem* — that things needs to happen together
    - Parallelism describes a *solution* — *that is* based on multiple threads/CPUs
- Today's class is about *designing* API for parallelism, under the functional paradigm

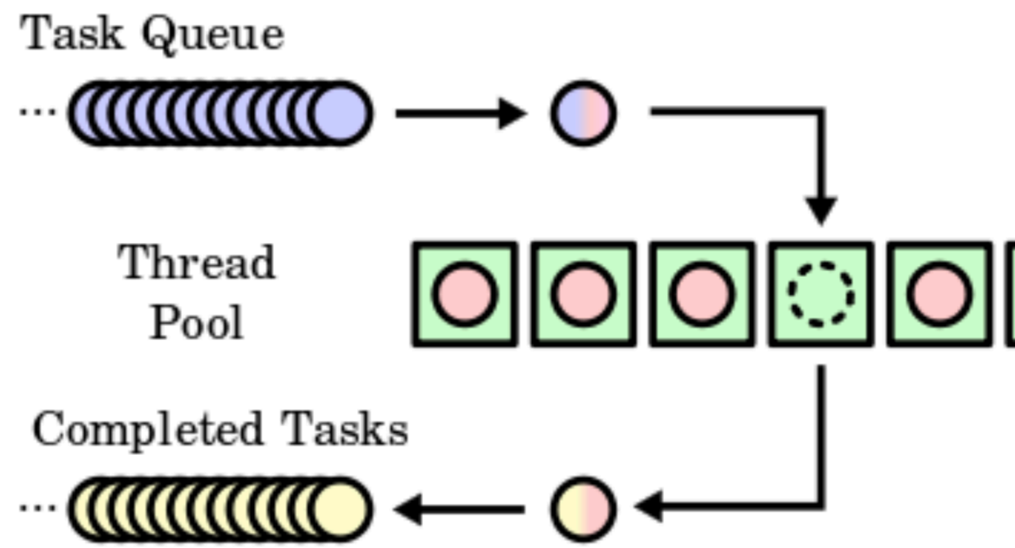
Clarification2: Parallelism is not always applicable

- You can calculate 1+2+3+…+N via parallelism
- But, you may not calculate 0.1+0.2+0.3 … via parallelism

Demo

Background: Java's ExecutorService and Future API

```
class ExecutorService {
  def submit[A](a: Callable[A]): Future[A]
}
trait Future[A] {
  def get: A
}
```



https://www.slideshare.net/afkham_azeez/java-colombo-developing-highly-scalable-apps

```
public interface ExecutorService
extends Executor
```

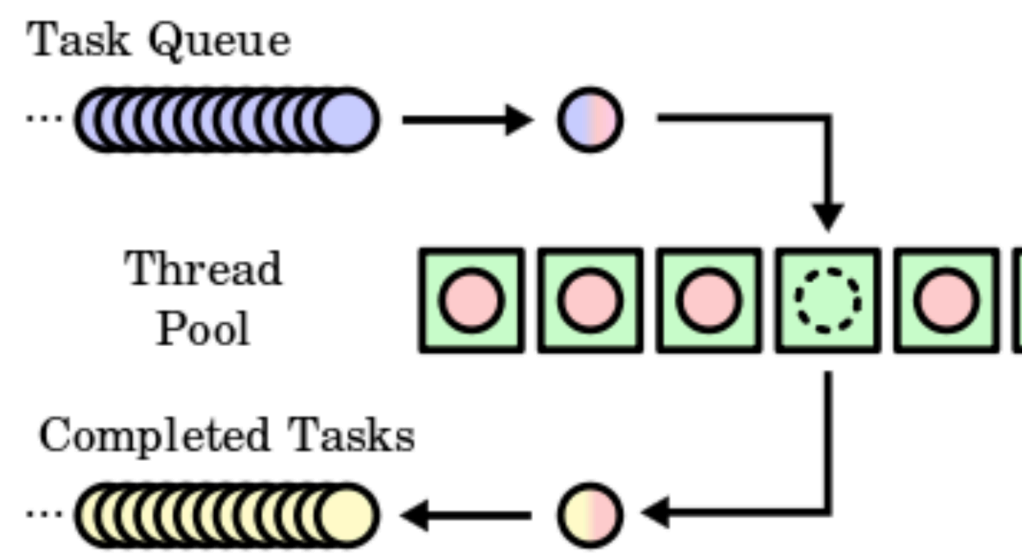
An `Executor` that provides methods to manage termination and methods that can produce a `Future` for tracking progress of one or more asynchronous tasks.

The thread pool execution uses a blocking queue. It keeps storing all the tasks that you have submitted to the **executor service**, and all threads (workers) are always running and performing the same steps:

- Take the Task from the queue
- Execute it
- Take the next or wait until a task will be added to the queue

Typical usage of ExecutorService

```scala
class ExecutorService {
  def submit[A](a: Callable[A]): Future[A]
}
trait Future[A] {
  def get: A
}
```



Typical usage1

```scala
val service: ExecutorService =
Executors.newFixedThreadPool(2)
service.submit(t1)
service.submit(t2)
service.submit(t3)
```

Typical usage2

```scala
val es=Executors.newWorkStealingPool()
val fut=es.submit(new Callable[Int] {
  override def call(): Int = (1 to 10).sum
})
val res=fut.get
```

Background: Strict and Lazy Functions

- Strict Evaluation (by-value): function argument evaluated before entering the function
- Lazy evaluation, (by-name, by-need)

```
val x = {println ("eager"); 5}
```

Earger

```
lazy val y = {println ("lazy"); 4}
```

Lazy

What would be x+x+y+y ?

DEMO

Example of using a lazy function (recall)

```
def time[A](a: => A) = {
  val now = System.nanoTime
  val result = a
  val micros = (System.nanoTime - now) / 1000
  println("%d microseconds".format(micros))
  result
}
```

DEMO

API for functional parallelism

- No right answers in design
- You will see a collection of design choices
- You are to understand their trade-offs, and think critically.

Why not use Java Thread

```
trait Runnable { def run: Unit }

class Thread(r: Runnable) {
  def start: Unit
  def join: Unit
}
```

Begins running `r` in a separate thread.

Blocks the calling thread until `r` finishes running.

- **Side-effects are evil when it comes to functional program and reasoning:** if we want to get any information out of a `Runnable`, it has to have some side effect, like mutating some state that we can inspect.

Design Goals

- No right answers in design.
- **Pure**: function return the same value for the same input, without observable side effects
- **High-level**: having the capability to write something like foldleft, as in sequential programs.
-

```
def sum(ints: Seq[Int]): Int =
  ints.foldLeft(0)((a,b) => a + b)
```

Design Methodologies

- Start from a very simple use case
- Try - Challenge - Refine

Example: Summing a list with divide-and-conquer

```
def sum(ints: IndexedSeq[Int]): Int =
  if (ints.size <= 1)
    ints.headOption getOrElse 0
  else {
    val (l,r) = ints.splitAt(ints.length/2)
    sum(l) + sum(r)
  }
```

Divides the sequence in half using the `splitAt` function.

`IndexedSeq` is a superclass of random-access sequences like `Vector` in the standard library. Unlike lists, these sequences provide an efficient `splitAt` method for dividing them into two parts at a particular index.

`headOption` is a method defined on all collections in Scala. We saw this function in chapter 4.

Recursively sums both halves and adds the results together.

- Listing 7.1, [Chiusano et al]

The Making of a Parallel Sum (1 - try)

```
def sum(ints: IndexedSeq[Int]): Int =
  if (ints.size <= 1)
    ints headOption getOrElse 0
  else {
    val (l,r) = ints.splitAt(ints.length/2)
    val sumL: Par[Int] = Par.unit(sum(l))
    val sumR: Par[Int] = Par.unit(sum(r))
    Par.get(sumL) + Par.get(sumR)
  }
```

Computes the left half in parallel.

Computes the right half in parallel.

Extracts both results and sums them.

- Need a data type to contain parallel computation results: **Par[A]**
- Need a function to evaluate a computation in a separate thread
  - **Par.unit** (a: =>A): Par[A]
- Need another function to extract a result from a Par[A]:
  - **Par.get [A] (a: Par[A]):A**

The Making of a Parallel Sum (1 - problem)

```
def sum(ints: IndexedSeq[Int]): Int =
  if (ints.size <= 1)
    ints headOption getOrElse 0
  else {
    val (l,r) = ints.splitAt(ints.length/2)
    val sumL: Par[Int] = Par.unit(sum(l))
    val sumR: Par[Int] = Par.unit(sum(r))
    Par.get(sumL) + Par.get(sumR)
  }
```

Computes the left half in parallel.

Computes the right half in parallel.

Extracts both results and sums them.

- For the sake of parallelization, Par.unit has to delay the computation until Par.get
- Problem: The whole computation is still sequential because "+" is strict

The Making of a Parallel Sum: (2 - try)

```
def sum(ints: IndexedSeq[Int]): Par[Int] =
  if (ints.size <= 1)
    Par.unit(ints.headOption getOrElse 0)
  else {
    val (l,r) = ints.splitAt(ints.length/2)
    Par.map2(sum(l), sum(r))(_ + _)
  }
```

- Par.map2 is a new higher-order function for combining the result of two parallel computations.
- Q: What is its signature?
- A: Par.map2[A,B,C] (a: Par[A], b: Par[B]) (f: (A,B) => C): Par[C]
- Q: Should Par.map2 be lazy or strict?
- A: :If it is strict, we'll strictly construct the entire left half of the tree of summations first before moving on to (strictly) constructing the right half ==> Let Par.map2 be lazy

The Making of a Parallel Sum: (2 - problem)

```
def sum(ints: IndexedSeq[Int]): Par[Int] =
  if (ints.size <= 1)
    Par.unit(ints.headOption getOrElse 0)
  else {
    val (l,r) = ints.splitAt(ints.length/2)
    Par.map2(sum(l), sum(r))(_ + _)
  }
```

- Q: Do we always want to evaluate the two arguments to Par.map2 in parallel?
- A: : Probably not. Consider Par.map2(Par.unit(1), Par.unit(2))(_+_). The overhead for thread creation/management is swamping any tiny gains from parallelization.
- Problem: This API is ver inexplicit about when computations gets forked off the main thread — the programmer cannot specify where this forking should occur.

The Making of a Parallel Sum: (3 - try)

```
def sum(ints: IndexedSeq[Int]): Par[Int] =
  if (ints.length <= 1)
    Par.unit(ints.headOption getOrElse 0)
  else {
    val (l,r) = ints.splitAt(ints.length/2)
    Par.map2(Par.fork(sum(l)), Par.fork(sum(r)))(_ + _)
  }
```

- Par.fork[A](a: => Par[A]): Par[Int] runs *a* in a separate logical thread
- With *Par.fork*, we can make *Par.map2* strict, leaving it up to the programmer to wrap arguments if they want

The Making of a Parallel Sum: (final)

- We let *fork* hold on to its unevaluated argument until later. It takes an unevaluated *Par[A]* and marks it for concurrent evaluation later
- In this model, *Par[A]* holds a *description* of a parallel computation that gets *interpreted* at a later time by something like the *get* function

Implementation: reused API from Java

```
class ExecutorService {
  def submit[A](a: Callable[A]): Future[A]
}
trait Callable[A] { def call: A }
trait Future[A] {
  def get: A
  def get(timeout: Long, unit: TimeUnit): A
  def cancel(evenIfRunning: Boolean): Boolean
  def isDone: Boolean
  def isCancelled: Boolean
}
```

- *Future* is a handle for running a computation in a separate thread
- *ExecutorService* allows us submit a *Callable* value and get back a corresponding *Future*
- When *Future* obtain a value from *get,* it blocks the current thread until the value is available.
- *Future* has extra features for cancellation, e.g., throwing an exception after blocking for a certain amount of time.

Implementation: Other API

- Type alias: *type Par[A] = ExecutorService => Future[A]*
- Object Par that holds three primitive operations: *unit, map2, and fork*

Demo

```scala
type Par[A] = ExecutorService => Future[A]

object Par {
  def unit[A](a:A):Par[A] = (es:ExecutorService)=> UnitFuture(a)

  private case class UnitFuture[A] (get: A) extends Future[A] {
    override def cancel(mayInterruptIfRunning: Boolean): Boolean = false

    override def isCancelled: Boolean = false

    override def isDone: Boolean = true

    override def get(timeout: Long, unit: TimeUnit): A = get
  }

  def map2[A,B,C] (a:Par[A],b:Par[B])(f:(A,B)=>C):Par[C] =
    (es:concurrent.ExecutorService) =>{
    val af = a(es)
    val bf = b(es)
    UnitFuture(f(af.get,bf.get))
  }

  def fork[A](a: => Par[A]) : Par[A] =
    es=>es.submit(new Callable[A] {
      override def call = a(es).get
    })
}
```

- unit is represented as a function that returns a UnitFuture, which is a simple implementation of Future that just wraps a constant value.

- map2 doesn’t evaluate the call to f in a separate logical thread, in accord with our design choice of having fork be the sole function in the API for controlling parallelism.

DEMO

Quiz

- Define map[A,B](pa: Par[A])(f: A => B): Par[B] in terms of:
    - map2[A,B,C] (a: Par[A], b: Par[B]) (f: (A,B) => C): Par[C]

- The fact that we can implement map in terms of map2 but not inversely, shows that map2 is strictly more powerful than map.
- This sort of thing happens a lot when we're designing libraries—often, a function that seems to be primitive will turn out to be expressible using some more powerful primitive.

Answer

- Define map[A,B](pa: Par[A])(f: A => B): Par[B] in terms of:
    - map2[A,B,C] (a: Par[A], b: Par[B]) (f: (A,B) => C): Par[C]
-

```
def map[A,B](pa: Par[A])(f: A => B): Par[B] =
  map2(pa, unit(()))((a,_) => f(a))
```

Laws and Properties

- Consider a unit test (incidentally of the unit function):
  - map(unit(1)) (_ + 1) == unit(2)
- One can define “==“ on the Par[Int] as follows:
  - def equal[A] (e: ExecutorService) (p: Par[A], p2: Par[A]) :Boolean = p(e).get == p2(e).get
- Such laws are useful as they can be turned into tests systematically

Conclusion

- Motivation for functional parallelism
- Designing the APIs for pure functional parallelism
- Re-used Java components