

Lecture 1

Introduction to Parallel Programming

Course Logistics

Starting point for information:

<http://lamp.epfl.ch/page-118166-en.html>

▶ <http://tinyurl.com/epflpar>

Also reachable from <http://lamp.epfl.ch/teaching>

Course Logistics

Starting point for information:

<http://lamp.epfl.ch/page-118166-en.html>

- ▶ <http://tinyurl.com/epflpar>

Also reachable from <http://lamp.epfl.ch/teaching>

Please follow instructions there to sign up for Coursera and use the form to which we link to tell us your coursera email.

- ▶ First part of course uses classical lectures, but we need your coursera accounts because grading of assignments is done through Coursera platform

Course Logistics

Starting point for information:

<http://lamp.epfl.ch/page-118166-en.html>

- ▶ <http://tinyurl.com/epflpar>

Also reachable from <http://lamp.epfl.ch/teaching>

Please follow instructions there to sign up for Coursera and use the form to which we link to tell us your coursera email.

- ▶ First part of course uses classical lectures, but we need your coursera accounts because grading of assignments is done through Coursera platform

Grading:

- ▶ 30% project assignments during the semester
- ▶ 30% mid-term after Spring break
- ▶ 40% final quiz on the last day (Friday) of the semester

Course Outline

Part I: Parallel programming in Scala:

- ▶ Week 1: Introduction, basic constructs, analysis
- ▶ Week 2: Reductions, operator associativity
- ▶ Week 3: Data-parallel programming
- ▶ Week 4: Parallel data structures and algorithms

Part II: Data Processing with Spark:

<https://spark.apache.org>

- ▶ Week 5: Spark and futures
- ▶ Week 6: Spark in-depth
- ▶ Week 7: Data mining and data analysis

Part III: Reactive Programming (7 weeks):

<https://www.coursera.org/course/reactive>

Lecturers

Part I: Parallel programming in Scala:

- ▶ Week 1: Viktor Kuncak
- ▶ Week 2: Viktor Kuncak
- ▶ Week 3: Aleksandar Prokopec
- ▶ Week 4: Aleksandar Prokopec

Part II: Data Processing with Spark:

<https://spark.apache.org>

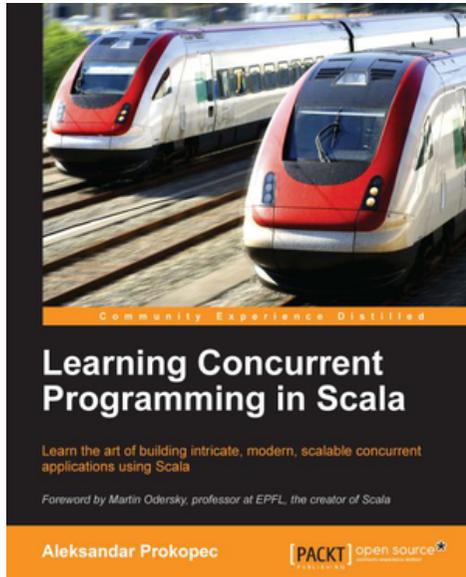
- ▶ Week 5: Heather Miller
- ▶ Week 6: Heather Miller
- ▶ Week 7: Heather Miller

Part III: Reactive Programming (7 weeks):

<https://www.coursera.org/course/reactive>

Martin Odersky, Roland Kuhn, and Erik Meijer

Course Textbook



Learning Concurrent Programming in Scala,
by Aleksandar Prokopec. PACKT Publishing,
November 2014

- ▶ Covers many of the concepts we teach in the first part of the course, and more

[https://www.packtpub.com/application-development/
learning-concurrent-programming-scala](https://www.packtpub.com/application-development/learning-concurrent-programming-scala)

Parallel Programming

Sequential programming: at every point in time, one part of program executing:

Parallel Programming

Sequential programming: at every point in time, one part of program executing:



Parallel Programming

Sequential programming: at every point in time, one part of program executing:



Parallel Programming

Sequential programming: at every point in time, one part of program executing:



Parallel programming: multiple parts of program execute at once:



In both cases, we compute the same functionality

Benefit: in parallel case we wait less until the work is completed

- ▶ it's all about **performance**

Parallel Programming: Infrastructure vs Applications

Two separate problems:

Parallel Programming: Infrastructure vs Applications

Two separate problems:

- ▶ **infrastructure:** build hardware, libraries, frameworks for parallel programming

Parallel Programming: Infrastructure vs Applications

Two separate problems:

- ▶ **infrastructure:** build hardware, libraries, frameworks for parallel programming
- ▶ **applications:** design your software (algorithms, data structures) in a way that exploits parallelism

For the start, we will look at **applications**

- ▶ we learn to **use** parallel programming primitives: **parallel**, **task**, parallel collections

Parallelism vs Concurrency (My Interpretation, Informal)

Concurrent programming is a general concept: structure program into concurrent tasks that **communicate**, both among each other and with the external environment.

- ▶ communication can happen in the middle of task execution
- ▶ must ensure safe accesses to shared resources (use e.g. locks)
- ▶ relevant even if we have no parallel hardware; OS and JVM provide threads even on a single CPU core
- ▶ can be used to ensure responsiveness in interactive apps

Parallelism vs Concurrency (My Interpretation, Informal)

Concurrent programming is a general concept: structure program into concurrent tasks that **communicate**, both among each other and with the external environment.

- ▶ communication can happen in the middle of task execution
- ▶ must ensure safe accesses to shared resources (use e.g. locks)
- ▶ relevant even if we have no parallel hardware; OS and JVM provide threads even on a single CPU core
- ▶ can be used to ensure responsiveness in interactive apps

Parallel programming: speed up computation through parallel hardware resources without solving all difficulties of concurrency (use frameworks to keep things simple). We often use this pattern:

Parallelism vs Concurrency (My Interpretation, Informal)

Concurrent programming is a general concept: structure program into concurrent tasks that **communicate**, both among each other and with the external environment.

- ▶ communication can happen in the middle of task execution
- ▶ must ensure safe accesses to shared resources (use e.g. locks)
- ▶ relevant even if we have no parallel hardware; OS and JVM provide threads even on a single CPU core
- ▶ can be used to ensure responsiveness in interactive apps

Parallel programming: speed up computation through parallel hardware resources without solving all difficulties of concurrency (use frameworks to keep things simple). We often use this pattern:

- ▶ get some part of the problem
- ▶ work on the problem in isolation, possibly creating other tasks
- ▶ return the result (or store it to a dedicated place)

We Write Programs in Scala

See the Coursera online course

Functional Programming Principles in Scala

- ▶ <https://www.coursera.org/course/progfun>

We benefit from an ecosystem of frameworks for parallel and concurrent programming in Scala

We Write Programs in Scala

See the Coursera online course

Functional Programming Principles in Scala

- ▶ <https://www.coursera.org/course/progfun>

We benefit from an ecosystem of frameworks for parallel and concurrent programming in Scala

We follow a functional programming style, try to avoid side effects. In practice, we may use some simple side effects, for efficiency, but important property of our programs in any case is:

If one parallel task writes to a variable (or array entry), no other task may read or write this variable at the same time.

Parallel programming primitives help us enforce this goal without relying on difficult general-purpose concurrent programming techniques.

A Minimal Construct for Parallelism

A Small Program: p -norm

First, solve sequentially the following problem, call it `sumSegment`

Given

- ▶ an integer array a
- ▶ a positive double floating point number p
- ▶ two valid indices $s \leq t$ into array a

compute

$$\sum_{i=s}^{t-1} [|a_i|^p]$$

Assume we have available the power function, defined e.g.

```
def power(x: Int, p: Double): Int = math.exp(p * math.log(abs(x))).toInt
```



Write the `sumSegment` function in Scala

Solution for sumSegment

```
def sumSegment(a: Array[Int], p: Double, s: Int, t: Int): Int = {  
  var i = s; var sum: Int = 0  
  while (i < t) {  
    sum = sum + power(a(i), p)  
    i = i + 1  
  }  
  sum  
}
```

Local **var**-s above are harmless; code above does same as:

```
def sumSegment(a: Array[Int], p: Double, s: Int, t: Int): Int = {  
  def sumFrom(i: Int, acc: Int): Int = {  
    if (i < t) sumFrom(i + 1, acc + power(a(i), p))  
    else acc  
  }  
  sumFrom(s, 0)  
}
```

The p -norm of an Array

def sumSegment(a: Array[Int], p: Double, s: Int, t: Int): Int

Use it to compute p -norm, ($p = 2$ - Euclidean norm). $N = a.length$

$$\|a\|_p := \left(\sum_{i=0}^{N-1} [|a_i|^p] \right)^{1/p}$$



Write expression for p -norm using sumSegment

The p -norm of an Array

```
def sumSegment(a: Array[Int], p: Double, s: Int, t: Int): Int
```

Use it to compute p -norm, ($p = 2$ - Euclidean norm). $N = a.length$

$$\|a\|_p := \left(\sum_{i=0}^{N-1} [|a_i|^p] \right)^{1/p}$$



Write expression for p -norm using `sumSegment`

```
def pNorm(a: Array[Int], p: Real): Int =  
  power(sumSegment(a, p, 0, a.length), 1/p)
```

The p -norm of an Array

```
def sumSegment(a: Array[Int], p: Double, s: Int, t: Int): Int
```

Use it to compute p -norm, ($p = 2$ - Euclidean norm). $N = a.length$

$$\|a\|_p := \left(\sum_{i=0}^{N-1} \|a_i\|^p \right)^{1/p}$$



Write expression for p -norm using sumSegment

```
def pNorm(a: Array[Int], p: Real): Int =  
  power(sumSegment(a, p, 0, a.length), 1/p)
```

Next use instead the following alternative formula (split the sum):

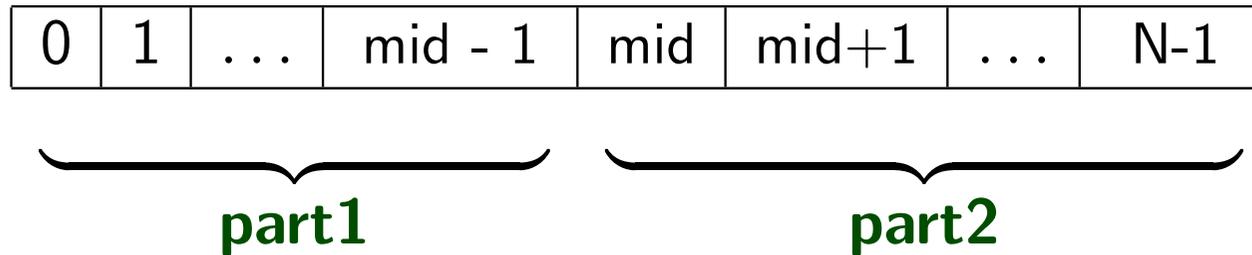
$$\|a\|_p := \left(\sum_{i=0}^{\lfloor N/2 \rfloor - 1} \|a_i\|^p + \sum_{i=\lfloor N/2 \rfloor}^{N-1} \|a_i\|^p \right)^{1/p}$$



What is the new expression?

Computing p -norm in Two Parts

Let $\text{mid} = N/2$



Computing p -norm in Two Parts

Let $\text{mid} = N/2$



```
def pNormTwoPart(a: Array[Int], p: Real): Int = {  
  val mid = a.length / 2  
  val (part1, part2) = (sumSegment(a, p, 0, mid),  
                        sumSegment(a, p, mid, a.length))  
  power(part1 + part2, 1/p)  
}
```

Computed a pair with two invocations, stored into two values, then summed them. Essentially same computation

Computing Two Parts in Parallel

This was two parts sequentially - nothing new here:

```
def pNormTwoPart(a: Array[Int], p: Real): Int = {  
  val mid = a.length / 2  
  val (part1, part2) = (sumSegment(a, p, 0, mid),  
                       sumSegment(a, p, mid, a.length))  
  power(part1 + part2, 1/p)  
}
```

Two parts in parallel:

Computing Two Parts in Parallel

This was two parts sequentially - nothing new here:

```
def pNormTwoPart(a: Array[Int], p: Real): Int = {  
  val mid = a.length / 2  
  val (part1, part2) = (sumSegment(a, p, 0, mid),  
                       sumSegment(a, p, mid, a.length))  
  power(part1 + part2, 1/p)  
}
```

Two parts in parallel:

```
def pNormTwoPart(a: Array[Int], p: Real): Int = {  
  val mid = a.length / 2  
  val (part1, part2) = parallel(sumSegment(a, p, 0, mid),  
                                 sumSegment(a, p, mid, a.length))  
  power(part1 + part2, 1/p) }
```

Computing Two Parts in Parallel

This was two parts sequentially - nothing new here:

```
def pNormTwoPart(a: Array[Int], p: Real): Int = {  
  val mid = a.length / 2  
  val (part1, part2) = (sumSegment(a, p, 0, mid),  
                       sumSegment(a, p, mid, a.length))  
  power(part1 + part2, 1/p)  
}
```

Two parts in parallel:

```
def pNormTwoPart(a: Array[Int], p: Real): Int = {  
  val mid = a.length / 2  
  val (part1, part2) = parallel(sumSegment(a, p, 0, mid),  
                                sumSegment(a, p, mid, a.length))  
  power(part1 + part2, 1/p) }
```

The only difference: invoke **parallel** on the pair of computations!

A Minimal Interface for Parallel Programming

```
def parallel[A, B](taskA: => A, taskB: => B): (A, B) = { ... }
```

A Minimal Interface for Parallel Programming

```
def parallel[A, B](taskA: => A, taskB: => B): (A, B) = { ... }
```

- ▶ type of result is the same as the type of its arguments

A Minimal Interface for Parallel Programming

```
def parallel[A, B](taskA: => A, taskB: => B): (A, B) = { ... }
```

- ▶ type of result is the same as the type of its arguments
- ▶ running time can be maximum of times (instead of sum)

A Minimal Interface for Parallel Programming

```
def parallel[A, B](taskA: => A, taskB: => B): (A, B) = { ... }
```

- ▶ type of result is the same as the type of its arguments
- ▶ running time can be maximum of times (instead of sum)
 - ▶ if one task is short, we need to wait for the second

A Minimal Interface for Parallel Programming

```
def parallel[A, B](taskA: => A, taskB: => B): (A, B) = { ... }
```

- ▶ type of result is the same as the type of its arguments
- ▶ running time can be maximum of times (instead of sum)
 - ▶ if one task is short, we need to wait for the second
- ▶ if there are no available parallel resources, it executes tasks sequentially (sum)

A Minimal Interface for Parallel Programming

```
def parallel[A, B](taskA: => A, taskB: => B): (A, B) = { ... }
```

- ▶ type of result is the same as the type of its arguments
- ▶ running time can be maximum of times (instead of sum)
 - ▶ if one task is short, we need to wait for the second
- ▶ if there are no available parallel resources, it executes tasks sequentially (sum)
- ▶ invoking **parallel** always carries a significant constant overhead (do not do it if not needed)

A Minimal Interface for Parallel Programming

```
def parallel[A, B](taskA: => A, taskB: => B): (A, B) = { ... }
```

- ▶ type of result is the same as the type of its arguments
- ▶ running time can be maximum of times (instead of sum)
 - ▶ if one task is short, we need to wait for the second
- ▶ if there are no available parallel resources, it executes tasks sequentially (sum)
- ▶ invoking **parallel** always carries a significant constant overhead (do not do it if not needed)
- ▶ arguments **taskA**, **taskB** are call by name

A Minimal Interface for Parallel Programming

```
def parallel[A, B](taskA: => A, taskB: => B): (A, B) = { ... }
```

- ▶ type of result is the same as the type of its arguments
- ▶ running time can be maximum of times (instead of sum)
 - ▶ if one task is short, we need to wait for the second
- ▶ if there are no available parallel resources, it executes tasks sequentially (sum)
- ▶ invoking **parallel** always carries a significant constant overhead (do not do it if not needed)
- ▶ arguments `taskA`, `taskB` are call by name
 - ▶  **What would happen if they were call by value (without `=>` before their types)?**

A Minimal Interface for Parallel Programming

```
def parallel[A, B](taskA: => A, taskB: => B): (A, B) = { ... }
```

- ▶ type of result is the same as the type of its arguments
- ▶ running time can be maximum of times (instead of sum)
 - ▶ if one task is short, we need to wait for the second
- ▶ if there are no available parallel resources, it executes tasks sequentially (sum)
- ▶ invoking **parallel** always carries a significant constant overhead (do not do it if not needed)
- ▶ arguments `taskA`, `taskB` are call by name
 - ▶  **What would happen if they were call by value (without `=>` before their types)?**

The implementation of **parallel** is not important for now (on JVM they mostly reduce to Java threads, which reduce typically to OS threads, which can run on different CPU cores).

Can p -norm Use More Parallel Resources When Available?

Two parts in parallel:

```
def pNormTwoPart(a: Array[Int], p: Real): Int = {  
  val mid = a.length / 2  
  val (part1, part2) = parallel(sumSegment(a, p, 0, mid),  
                                sumSegment(a, p, mid, a.length))  
  power(part1 + part2, 1/p) }
```



How to do 4 parts in parallel?

Can p -norm Use More Parallel Resources When Available?

Two parts in parallel:

```
def pNormTwoPart(a: Array[Int], p: Real): Int = {  
  val mid = a.length / 2  
  val (part1, part2) = parallel(sumSegment(a, p, 0, mid),  
                                sumSegment(a, p, mid, a.length))  
  power(part1 + part2, 1/p) }  
}
```



How to do 4 parts in parallel?

```
def pNormFourPart(a: Array[Int], p: Real): Int = {  
  val mid1 = a.length / 4  
  val mid2 = a.length / 2  
  val mid3 = 3 * a.length / 4  
  val ((part1, part2), (part3, part4)) =  
    parallel(parallel(sumSegment(a, p, 0, mid1),  
                    sumSegment(a, p, mid1, mid2)),  
            parallel(sumSegment(a, p, mid2, mid3),  
                    sumSegment(a, p, mid3, a.length)))  
  power(part1 + part2 + part3 + part4, 1/p) }  
}
```

Divide and Conquer Recursive p -norm

```
def pNormRec(a: Array[Int], p: Real): Int =  
    power(segmentRec(a, p, 0, a.length), 1/p)
```

```
def segmentRec(a: Array[Int], p: Real, s: Int, t: Int) = {  
    if (t - s < threshold)  
        sumSegment(xs, p, s, t) // for small segments: faster to do sequentially  
    else {  
        val mid = s + (t - s)/2  
        val (leftSum, rightSum) =  
            parallel(segmentRec(a, p, s, mid),  
                    segmentRec(a, p, mid, t))  
        leftSum + rightSum  
    }  
}
```

Analyzing Parallel Programs

Notions of Work and Span

How long does our computation take?

```
parallel(parallel(sumSegment(a, p, 0, mid1),  
                sumSegment(a, p, mid1, mid2)),  
        parallel(sumSegment(a, p, mid2, mid3),  
                sumSegment(a, p, mid3, a.length)))
```

Assume that we have enough parallel threads and memory bandwidth and that the “costs”, $S(e)$, of different program parts are as follows:

- ▶ $S(\text{sumSegment}(a, p, s, t)) = c_1(t - s) + c_2$
- ▶ $S(\text{parallel}(t1, t2)) = c_P + \max(S(t1), S(t2))$
Note: if $S(t1) = S(t2) = c$, this reduces to $c_P + c$

How long does our computation take?

```
parallel(parallel(sumSegment(a, p, 0, mid1),  
                sumSegment(a, p, mid1, mid2)),  
        parallel(sumSegment(a, p, mid2, mid3),  
                sumSegment(a, p, mid3, a.length)))
```

Assume that we have enough parallel threads and memory bandwidth and that the “costs”, $S(e)$, of different program parts are as follows:

- ▶ $S(\text{sumSegment}(a, p, s, t)) = c_1(t - s) + c_2$

- ▶ $S(\text{parallel}(t1, t2)) = c_P + \max(S(t1), S(t2))$

Note: if $S(t1) = S(t2) = c$, this reduces to $c_P + c$

For each `sumSegment` above,

$$S(\text{sumSegment}(a, p, s, t)) = c_1 N/4 + c_2 \quad (N = a.length)$$

How long does our computation take?

```
parallel(parallel(sumSegment(a, p, 0, mid1),  
                sumSegment(a, p, mid1, mid2)),  
        parallel(sumSegment(a, p, mid2, mid3),  
                sumSegment(a, p, mid3, a.length)))
```

Assume that we have enough parallel threads and memory bandwidth and that the “costs”, $S(e)$, of different program parts are as follows:

- ▶ $S(\text{sumSegment}(a, p, s, t)) = c_1(t - s) + c_2$

- ▶ $S(\text{parallel}(t1, t2)) = c_P + \max(S(t1), S(t2))$

Note: if $S(t1) = S(t2) = c$, this reduces to $c_P + c$

For each `sumSegment` above,

$$S(\text{sumSegment}(a, p, s, t)) = c_1 N/4 + c_2 \quad (N = a.length)$$

$$S(\text{parallel}(\text{sumSegment}, \text{sumSegment})) =$$

How long does our computation take?

```
parallel(parallel(sumSegment(a, p, 0, mid1),
                  sumSegment(a, p, mid1, mid2)),
         parallel(sumSegment(a, p, mid2, mid3),
                  sumSegment(a, p, mid3, a.length)))
```

Assume that we have enough parallel threads and memory bandwidth and that the “costs”, $S(e)$, of different program parts are as follows:

- ▶ $S(\text{sumSegment}(a, p, s, t)) = c_1(t - s) + c_2$

- ▶ $S(\text{parallel}(t1, t2)) = c_P + \max(S(t1), S(t2))$

Note: if $S(t1) = S(t2) = c$, this reduces to $c_P + c$

For each `sumSegment` above,

$$S(\text{sumSegment}(a, p, s, t)) = c_1 N/4 + c_2 \quad (N = a.length)$$

$$S(\text{parallel}(\text{sumSegment}, \text{sumSegment})) = c_P + c_1 N/4 + c_2$$

How long does our computation take?

```
parallel(parallel(sumSegment(a, p, 0, mid1),
                  sumSegment(a, p, mid1, mid2)),
         parallel(sumSegment(a, p, mid2, mid3),
                  sumSegment(a, p, mid3, a.length)))
```

Assume that we have enough parallel threads and memory bandwidth and that the “costs”, $S(e)$, of different program parts are as follows:

- ▶ $S(\text{sumSegment}(a, p, s, t)) = c_1(t - s) + c_2$

- ▶ $S(\text{parallel}(t1, t2)) = c_P + \max(S(t1), S(t2))$

Note: if $S(t1) = S(t2) = c$, this reduces to $c_P + c$

For each `sumSegment` above,

$$S(\text{sumSegment}(a, p, s, t)) = c_1 N/4 + c_2 \quad (N = a.length)$$

$$S(\text{parallel}(\text{sumSegment}, \text{sumSegment})) = c_P + c_1 N/4 + c_2$$

$$S(\text{parallel}(\text{parallel}, \text{parallel})) =$$

How long does our computation take?

```
parallel(parallel(sumSegment(a, p, 0, mid1),
                  sumSegment(a, p, mid1, mid2)),
         parallel(sumSegment(a, p, mid2, mid3),
                  sumSegment(a, p, mid3, a.length)))
```

Assume that we have enough parallel threads and memory bandwidth and that the “costs”, $S(e)$, of different program parts are as follows:

- ▶ $S(\text{sumSegment}(a, p, s, t)) = c_1(t - s) + c_2$

- ▶ $S(\text{parallel}(t1, t2)) = c_P + \max(S(t1), S(t2))$

Note: if $S(t1) = S(t2) = c$, this reduces to $c_P + c$

For each `sumSegment` above,

$$S(\text{sumSegment}(a, p, s, t)) = c_1 N/4 + c_2 \quad (N = a.length)$$

$$S(\text{parallel}(\text{sumSegment}, \text{sumSegment})) = c_P + c_1 N/4 + c_2$$

$$S(\text{parallel}(\text{parallel}, \text{parallel})) = c_P + c_P + \mathbf{c_1 N/4} + c_2$$

How long does our computation take?

```
parallel(parallel(sumSegment(a, p, 0, mid1),
                  sumSegment(a, p, mid1, mid2)),
         parallel(sumSegment(a, p, mid2, mid3),
                  sumSegment(a, p, mid3, a.length)))
```

Assume that we have enough parallel threads and memory bandwidth and that the “costs”, $S(e)$, of different program parts are as follows:

- ▶ $S(\text{sumSegment}(a, p, s, t)) = c_1(t - s) + c_2$

- ▶ $S(\text{parallel}(t1, t2)) = c_P + \max(S(t1), S(t2))$

Note: if $S(t1) = S(t2) = c$, this reduces to $c_P + c$

For each `sumSegment` above,

$$S(\text{sumSegment}(a, p, s, t)) = c_1 N/4 + c_2 \quad (N = a.length)$$

$$S(\text{parallel}(\text{sumSegment}, \text{sumSegment})) = c_P + c_1 N/4 + c_2$$

$$S(\text{parallel}(\text{parallel}, \text{parallel})) = c_P + c_P + \mathbf{c_1 N/4} + c_2$$

For large N , this is a win compared to sequential: $\mathbf{c_1 N} + c_2$

Estimating Cost of Our Programs

You have previously learned how to concisely characterize behavior of sequential programs using the number of operations they perform as a function of arguments.

- ▶ inserting an integer into a sorted linear list takes time $O(n)$, for list storing n integers
- ▶ inserting an integer into a balanced binary tree of n integers takes time $O(\log n)$, for tree storing n integers

Estimating Cost of Our Programs

You have previously learned how to concisely characterize behavior of sequential programs using the number of operations they perform as a function of arguments.

- ▶ inserting an integer into a sorted linear list takes time $O(n)$, for list storing n integers
- ▶ inserting an integer into a balanced binary tree of n integers takes time $O(\log n)$, for tree storing n integers

We would now similarly like to speak about the complexity of parallel code

Estimating Cost of Our Programs

You have previously learned how to concisely characterize behavior of sequential programs using the number of operations they perform as a function of arguments.

- ▶ inserting an integer into a sorted linear list takes time $O(n)$, for list storing n integers
- ▶ inserting an integer into a balanced binary tree of n integers takes time $O(\log n)$, for tree storing n integers

We would now similarly like to speak about the complexity of parallel code

- ▶ but this depends on available parallel resources

Estimating Cost of Our Programs

You have previously learned how to concisely characterize behavior of sequential programs using the number of operations they perform as a function of arguments.

- ▶ inserting an integer into a sorted linear list takes time $O(n)$, for list storing n integers
- ▶ inserting an integer into a balanced binary tree of n integers takes time $O(\log n)$, for tree storing n integers

We would now similarly like to speak about the complexity of parallel code

- ▶ but this depends on available parallel resources
- ▶ introduce two measures for a program: **span** and **work**

Estimating Cost of Our Programs

You have previously learned how to concisely characterize behavior of sequential programs using the number of operations they perform as a function of arguments.

- ▶ inserting an integer into a sorted linear list takes time $O(n)$, for list storing n integers
- ▶ inserting an integer into a balanced binary tree of n integers takes time $O(\log n)$, for tree storing n integers

We would now similarly like to speak about the complexity of parallel code

- ▶ but this depends on available parallel resources
- ▶ introduce two measures for a program: **span** and **work**

Span, $S(e)$: number of steps if we had unbounded parallelism

Estimating Cost of Our Programs

You have previously learned how to concisely characterize behavior of sequential programs using the number of operations they perform as a function of arguments.

- ▶ inserting an integer into a sorted linear list takes time $O(n)$, for list storing n integers
- ▶ inserting an integer into a balanced binary tree of n integers takes time $O(\log n)$, for tree storing n integers

We would now similarly like to speak about the complexity of parallel code

- ▶ but this depends on available parallel resources
- ▶ introduce two measures for a program: **span** and **work**

Span, $S(e)$: number of steps if we had unbounded parallelism

Work, $W(e)$: number of steps e would take if there was no parallelism (this is simply the sequential execution time)

Rules for Span and Work

Key rules are for

- ▶ $S(\text{parallel}(e_1, e_2)) = c_P + \max(S(e_1), S(e_2))$
- ▶ $W(\text{parallel}(e_1, e_2)) = c_Q + W(e_1) + W(e_2)$

If we divide work in equal parts, for S it counts only once!

Rules for Span and Work

Key rules are for

- ▶ $S(\text{parallel}(e_1, e_2)) = c_P + \max(S(e_1), S(e_2))$
- ▶ $W(\text{parallel}(e_1, e_2)) = c_Q + W(e_1) + W(e_2)$

If we divide work in equal parts, for S it counts only once!

For parts of code where we do not use **parallel** explicitly, we must add up costs. For function call or operation $f(e_1, \dots, e_n)$:

- ▶ $S(f(e_1, \dots, e_n)) = S(e_1) + \dots + S(e_n) + S(f)(v_1, \dots, v_n)$
- ▶ $W(f(e_1, \dots, e_n)) = W(e_1) + \dots + W(e_n) + W(f)(v_1, \dots, v_n)$

Here v_i denotes values of e_i . If f is primitive operation on integers, then $S(f)$ and $W(f)$ are constant functions, regardless of v_i .

Rules for Span and Work

Key rules are for

- ▶ $S(\text{parallel}(e_1, e_2)) = c_P + \max(S(e_1), S(e_2))$
- ▶ $W(\text{parallel}(e_1, e_2)) = c_Q + W(e_1) + W(e_2)$

If we divide work in equal parts, for S it counts only once!

For parts of code where we do not use **parallel** explicitly, we must add up costs. For function call or operation $f(e_1, \dots, e_n)$:

- ▶ $S(f(e_1, \dots, e_n)) = S(e_1) + \dots + S(e_n) + S(f)(v_1, \dots, v_n)$
- ▶ $W(f(e_1, \dots, e_n)) = W(e_1) + \dots + W(e_n) + W(f)(v_1, \dots, v_n)$

Here v_i denotes values of e_i . If f is primitive operation on integers, then $S(f)$ and $W(f)$ are constant functions, regardless of v_i .

Rules for (non-lazy) **val** are similar to those for nested expressions:

$$S(\{\mathbf{val} v_1 = e_1; \dots \mathbf{val} v_p = e_p; f(v_{i_1}, \dots, v_{i_n})\}) = \\ S(e_1) + \dots + S(e_n) + S(f)(v_{i_1}, \dots, v_{i_n})$$

Using Span and Work for Estimates

Suppose we know $S(e)$ and $W(e)$ and we consider platform with at most K parallel threads.

Using Span and Work for Estimates

Suppose we know $S(e)$ and $W(e)$ and we consider platform with at most K parallel threads.

Regardless of K , we cannot finish sooner than $S(e)$ because of dependencies

Using Span and Work for Estimates

Suppose we know $S(e)$ and $W(e)$ and we consider platform with at most K parallel threads.

Regardless of K , we cannot finish sooner than $S(e)$ because of dependencies

Regardless of $S(e)$, we cannot finish sooner than $W(e)/K$, since every piece of work needs to be done by some parallel thread

Thus we need to wait at least

$$\max \left(S(e), \frac{W(e)}{K} \right)$$

Parallelism is “just” a way to improve constant factors:

- ▶ If K is constant but inputs grow, parallel programs have same asymptotic time complexity as sequential ones
- ▶ Even if we have infinite resources, ($K \rightarrow \infty$), we have non-zero complexity given by $S(e)$

Parallelism and Amdahl's Law

Suppose that we have two parts of a sequential computation:

- ▶ part1 takes fraction f (e.g. 40%) of the computation time
- ▶ part2 take the remaining $1 - f$ (60%)

Suppose we can only speed up part2. If we manage to make part2 K times faster (say 100), if the original running time was one unit, the new running time will be

$$f + \frac{1 - f}{K}$$

So the speedup compared to the original time is

$$\frac{1}{f + \frac{1-f}{K}}$$

In our example, we obtain

$$\frac{1}{0.4 + \frac{0.6}{100}} = 2.46$$

Even if we speed the second part infinitely, we can obtain at most $1/0.4 = 2.5$ speed up.

Span and Work for Recursive p -norm

```
def segmentRec(a: Array[Int], p: Real, s: Int, t: Int) = {  
  if (t - s < thr)  
    sumSegment(xs, p, s, t)  
  else {  
    val mid = s + (t - s)/2  
    val (leftSum, rightSum) =  
      parallel(segmentRec(a, p, s, mid),  
              segmentRec(a, p, mid, t))  
    leftSum + rightSum  
  } }  
}
```

Span and Work for Recursive p -norm

```
def segmentRec(a: Array[Int], p: Real, s: Int, t: Int) = {  
  if (t - s < thr)  
    sumSegment(xs, p, s, t)  
  else {  
    val mid = s + (t - s)/2  
    val (leftSum, rightSum) =  
      parallel(segmentRec(a, p, s, mid),  
              segmentRec(a, p, mid, t))  
    leftSum + rightSum  
  } }  
}
```

Let S_k denote $S(\text{segmentRec}(a, p, s, t))$ for $t - s = 2^k \text{thr}$.

Span and Work for Recursive p -norm

```
def segmentRec(a: Array[Int], p: Real, s: Int, t: Int) = {  
  if (t - s < thr)  
    sumSegment(xs, p, s, t)  
  else {  
    val mid = s + (t - s)/2  
    val (leftSum, rightSum) =  
      parallel(segmentRec(a, p, s, mid),  
              segmentRec(a, p, mid, t))  
    leftSum + rightSum  
  } }  
}
```

Let S_k denote $S(\text{segmentRec}(a, p, s, t))$ for $t - s = 2^k \text{thr}$. Then $S_k = c_1 + S_{k-1}$ for some c_1 .

Span and Work for Recursive p -norm

```
def segmentRec(a: Array[Int], p: Real, s: Int, t: Int) = {  
  if (t - s < thr)  
    sumSegment(xs, p, s, t)  
  else {  
    val mid = s + (t - s)/2  
    val (leftSum, rightSum) =  
      parallel(segmentRec(a, p, s, mid),  
              segmentRec(a, p, mid, t))  
    leftSum + rightSum  
  } }  
}
```

Let S_k denote $S(\text{segmentRec}(a, p, s, t))$ for $t - s = 2^k \text{thr}$. Then $S_k = c_1 + S_{k-1}$ for some c_1 . By induction $S_k = kc_1 S_0 = O(k)$

Span and Work for Recursive p -norm

```
def segmentRec(a: Array[Int], p: Real, s: Int, t: Int) = {  
  if (t - s < thr)  
    sumSegment(xs, p, s, t)  
  else {  
    val mid = s + (t - s)/2  
    val (leftSum, rightSum) =  
      parallel(segmentRec(a, p, s, mid),  
              segmentRec(a, p, mid, t))  
    leftSum + rightSum  
  } }  
}
```

Let S_k denote $S(\text{segmentRec}(a, p, s, t))$ for $t - s = 2^k \text{thr}$. Then $S_k = c_1 + S_{k-1}$ for some c_1 . By induction $S_k = kc_1 S_0 = O(k)$

- ▶ span S is logarithmic in $t - s$

Span and Work for Recursive p -norm

```
def segmentRec(a: Array[Int], p: Real, s: Int, t: Int) = {  
  if (t - s < thr)  
    sumSegment(xs, p, s, t)  
  else {  
    val mid = s + (t - s)/2  
    val (leftSum, rightSum) =  
      parallel(segmentRec(a, p, s, mid),  
              segmentRec(a, p, mid, t))  
    leftSum + rightSum  
  } }  
}
```

Let S_k denote $S(\text{segmentRec}(a, p, s, t))$ for $t - s = 2^k \text{thr}$. Then $S_k = c_1 + S_{k-1}$ for some c_1 . By induction $S_k = kc_1 S_0 = O(k)$

► span S is logarithmic in $t - s$

W_k denote $W(\text{segmentRec}(a, p, s, t))$ for $t - s = 2^k \text{thr}$. Then $W_k = c_2 + 2W_{k-1}$.

Span and Work for Recursive p -norm

```
def segmentRec(a: Array[Int], p: Real, s: Int, t: Int) = {  
  if (t - s < thr)  
    sumSegment(xs, p, s, t)  
  else {  
    val mid = s + (t - s)/2  
    val (leftSum, rightSum) =  
      parallel(segmentRec(a, p, s, mid),  
              segmentRec(a, p, mid, t))  
    leftSum + rightSum  
  } }  
}
```

Let S_k denote $S(\text{segmentRec}(a, p, s, t))$ for $t - s = 2^k \text{thr}$. Then $S_k = c_1 + S_{k-1}$ for some c_1 . By induction $S_k = kc_1 S_0 = O(k)$

► span S is logarithmic in $t - s$

W_k denote $W(\text{segmentRec}(a, p, s, t))$ for $t - s = 2^k \text{thr}$. Then $W_k = c_2 + 2W_{k-1}$. $W_k = c_2(1 + 2 + \dots + 2^{k-1}) + 2^k W_0 = O(2^k)$

Span and Work for Recursive p -norm

```
def segmentRec(a: Array[Int], p: Real, s: Int, t: Int) = {  
  if (t - s < thr)  
    sumSegment(xs, p, s, t)  
  else {  
    val mid = s + (t - s)/2  
    val (leftSum, rightSum) =  
      parallel(segmentRec(a, p, s, mid),  
              segmentRec(a, p, mid, t))  
    leftSum + rightSum  
  } }  
}
```

Let S_k denote $S(\text{segmentRec}(a, p, s, t))$ for $t - s = 2^k \text{thr}$. Then $S_k = c_1 + S_{k-1}$ for some c_1 . By induction $S_k = kc_1 S_0 = O(k)$

- ▶ span S is logarithmic in $t - s$

W_k denote $W(\text{segmentRec}(a, p, s, t))$ for $t - s = 2^k \text{thr}$. Then $W_k = c_2 + 2W_{k-1}$. $W_k = c_2(1 + 2 + \dots + 2^{k-1}) + 2^k W_0 = O(2^k)$

- ▶ work W is linear in $t - s$

Week 1

ScalaMeter



Testing

- unit testing - verifies that functionality of a specific section of code is correct
- benchmarks - used to assess relative performance of a specific section of code

We rely on parallel programming to improve program performance. Therefore, we almost always want to benchmark the code.

ScalaMeter

Benchmarking and performance regression testing framework for the JVM.

- performance regression testing - comparing the performance of the current program run against the previous runs
- benchmarking - measuring performance of the current program run, or part of the program  **our focus**

Using ScalaMeter

Add ScalaMeter as a dependency:

```
libraryDependencies +=  
  "com.storm-enroute" %% "scalameter-core" % "0.6"
```

Import the contents of the ScalaMeter package, and measure:

```
import org.scalameter._  
  
val time = measure {  
  (0 until 1000000).toArray  
}  
  
println(s"Array initialization time: $time ms")
```

```
Array initialization time: 233.897224 ms
```

Using ScalaMeter

The previous program outputs the running time:

```
Array initialization time: 233.897224 ms
```

However, if we run the program again:

```
Array initialization time: 110.354206 ms
```

Why is that?

JVM warmup

When a JVM program starts, it undergoes a period of *warmup*, after which it achieves its maximum performance.

- first, the program is *interpreted*
- then, parts of the program are compiled into machine code
- the JVM may apply additional dynamic optimizations during the run of the program
- and eventually, the program reaches *steady-state* performance

ScalaMeter Warmers

Usually, we want to measure the steady-state performance of the program.

ScalaMeter Warmer objects run the benchmarked code until they detect steady state.

```
import org.scalameter._

val time = withWarmer(new Warmer.Default) measure {
  (0 until 1000000).toArray
}

println(s"Array initialization time: $time ms")
```

```
Array initialization time: 3.458489 ms
```

ScalaMeter Configuration

`Warmer.Default` runs at least `minWarmupRuns` times, and at most `maxWarmupRuns`, and until it detects stability.

ScalaMeter allows configuring these values:

```
val time = config(
  Key.exec.minWarmupRuns -> 20,
  Key.exec.maxWarmupRuns -> 60,
  Key.verbose -> true
) withWarmer(new Warmer.Default) measure {
  (0 until 1000000).toArray
}
```

```
Starting warmup.
0. warmup run running time: 18.991815 (covNoGC: NaN, covGC: NaN)
1. warmup run running time: 14.542149 (covNoGC: 0.1877, covGC: 0.1877)
2. warmup run running time: 3.448592 (covNoGC: 0.6364, covGC: 0.6364)
...
59. warmup run running time: 3.498078 (covNoGC: 0.1606, covGC: 0.2349)
Steady-state not detected.
measurements: 3.439795, 3.446759, 3.429897, ...
```

ScalaMeter Measurers

We can measure more than just running time!

- `Default` - plain running time
- `IgnoringGC` - running time without GC pauses
- `OutlierElimination` - removes statistical outliers
- `MemoryFootprint` - memory footprint of an object or a data structure created by the code snippet
- `GarbageCollectionCycles` - total number of GC cycles during the execution of a code snippet

Happy measuring!

A Note on Memory Bandwidth

What If We Compute Just Array Sum?

```
def sumSegment(a: Array[Int], p: Double, s: Int, t: Int): Int = {  
  var i = s; var sum: Int = 0  
  while (i < t) {  
    sum = sum + a(i) // no exponentiation  
    i = i + 1  
  }  
  sum }
```

```
val ((part1, part2), (part3, part4)) =  
  parallel(parallel(sumSegment(a, p, 0, mid1),  
                    sumSegment(a, p, mid1, mid2)),  
           parallel(sumSegment(a, p, mid2, mid3),  
                    sumSegment(a, p, mid3, a.length)))  
power(part1 + part2 + part3 + part4, 1/p) }
```

On a consumer desktop of today you may find it difficult to get any speedup, even if you have 4 cores.



Why?

RAM as Bottleneck

```
def sumSegment(a: Array[Int], p: Double, s: Int, t: Int): Int = {  
  var i = s; var sum: Int = 0  
  while (i < t) {  
    sum = sum + a(i) // no exponentiation  
    i = i + 1  
  }  
  sum }
```

The computation is **memory bound**: almost all time is spent waiting for data to arrive from memory, because computation itself (integer sum) is so simple. Caches do not help because we traverse all elements once.

Even if we parallelize to multiple cores in the same socket, we end up using same memory channel, so we are again bound by the same memory bandwidth.

Example: Parallel Monte Carlo Calculation (of π)

A Methods to Estimate π (3.14159...)

First, bake a perfectly round pie



A Methods to Estimate π (3.14159...)

First, bake a perfectly round pie



Put it into a square that touches its borders.

A Methods to Estimate π (3.14159...)

First, bake a perfectly round pie



Put it into a square that touches its borders.

Sample: take a toothpick, uniformly poke inside the square.

A Methods to Estimate π (3.14159...)

First, bake a perfectly round pie



Put it into a square that touches its borders.

Sample: take a toothpick, uniformly poke inside the square.

Compute the percentage λ of times you picked into the pie.

A Methods to Estimate π (3.14159...)

First, bake a perfectly round pie



Put it into a square that touches its borders.

Sample: take a toothpick, uniformly poke inside the square.

Compute the percentage λ of times you picked into the pie.

This should approach ration of the circle and the square:

$$\lambda \approx \frac{r^2 \pi}{(2r)^2} = \frac{\pi}{4}$$

A Methods to Estimate π (3.14159...)

First, bake a perfectly round pie



Put it into a square that touches its borders.

Sample: take a toothpick, uniformly poke inside the square.

Compute the percentage λ of times you picked into the pie.

This should approach ration of the circle and the square:

$$\lambda \approx \frac{r^2 \pi}{(2r)^2} = \frac{\pi}{4}$$

So, compute π approximately as 4λ .

Code for Sampling Pie

Sequential:

```
import scala.util.Random
def monteCarloPi(iterations: Int): Double = {
  val randomX = new Random; val randomY = new Random
  var hits = 0
  for (i <- 0 until iterations) {
    val x = randomX.nextDouble(); val y = randomY.nextDouble()
    if (x*x + y*y < 1) hits = hits + 1
  }
  4.0*hits/iterations
}
```

Code for Sampling Pie

Sequential:

```
import scala.util.Random
def monteCarloPi(iterations: Int): Double = {
  val randomX = new Random; val randomY = new Random
  var hits = 0
  for (i ← 0 until iterations) {
    val x = randomX.nextDouble(); val y = randomY.nextDouble()
    if (x*x + y*y < 1) hits = hits + 1
  }
  4.0*hits/iterations
}
```

Speeding up using parallelization:

```
def parMonteCarloPi(iterations: Int): Double = {
  val ((pi1, pi2), (pi3, pi4)) = parallel(
    parallel(monteCarloPi(iterations/4), monteCarloPi(iterations/4)),
    parallel(monteCarloPi(iterations/4), monteCarloPi(iterations/4)))
  (pi1 + pi2 + pi3 + pi4)/4
}
```

A More Flexible Way to Start Tasks

Starting Four Tasks. The Execution Graph

We have seen the **parallel** construct:

```
val ((part1, part2),(part3,part4)) =  
    parallel(parallel(sumSegment(a, p, 0, mid1),  
                      sumSegment(a, p, mid1, mid2)),  
            parallel(sumSegment(a, p, mid2, mid3),  
                      sumSegment(a, p, mid3, a.length)))  
power(part1 + part2 + part3 + part4, 1/p)
```

Here is an alternative construct: **task**. Same example:

```
val task1 = task {sumSegment(a, p, 0, mid1)}  
val task2 = task {sumSegment(a, p, mid1, mid2)}  
val task3 = task {sumSegment(a, p, mid2, mid3)}  
val part4 = sumSegment(a, p, mid3, a.length)  
power(part1.join + part2.join + part3.join + part4, 1/p)
```

A More Flexible Interface: task

```
def task(c: => A) : ForkJoinTask[A]
```

```
trait ForkJoinTask[A] {  
  def join: A  
}
```

No need to always do binary splits.

Need not wait for both computations to finish

- ▶ you should not call join on a task sooner than needed

Still type safe and gives back the value

Can we define 'parallel' using 'task'?



Define parallel using task

Can we define 'parallel' using 'task'?



Define parallel using task

```
def parallel[A, B](ca: => A, cb: => B): (A, B) = {  
  val right = task { cb }  
  val left = ca  
  (left, right.join())  
}
```

What is the type of `left` and what of `right`?

Can we define 'parallel' using 'task'?



Define parallel using task

```
def parallel[A, B](ca: => A, cb: => B): (A, B) = {  
  val right = task { cb }  
  val left = ca  
  (left, right.join())  
}
```

What is the type of `left` and what of `right`?

What is wrong with the following attempt at parallel?

```
def parallelWrong[A, B](ca: => A, cb: => B): (A, B) = {  
  val right = (task { cb }).join()  
  val left = ca  
  (left, right)  
}
```

- ▶ Does it type check? What are the types of `left`, `right`?
- ▶ How does it behave?