



ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Parallel Programming Week 1

Parallel Programming in Scala

Aleksandar Prokopec and Viktor Kuncak

What is Parallel Computing?

Parallel computing is a type of computation in which many calculations are performed at the same time.

Basic principle: computation can be divided into smaller subproblems, each of which can be solved simultaneously.

Assumption: we have parallel hardware at our disposal, which is capable of executing these computations in parallel.

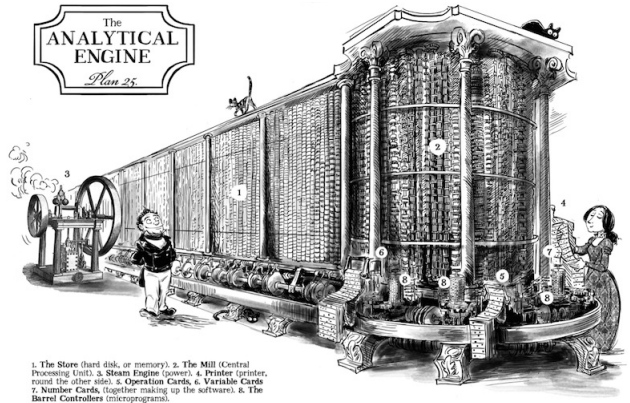
History

Parallel computing was present since the early days of computing.



History

Parallel computing was present since the early days of computing.



History

Parallel computing was present since the early days of computing.



History

Parallel computing was present since the early days of computing.

In the 20th century, researchers from IBM built some of the first commercial parallel computers.

For over a decade prophets have voiced the contention that the organization of a single computer has reached its limits and that truly significant advances can be made only by interconnection of a multiplicity of computers.

Gene Amdahl, 1967.

At the time, parallel computing was confined to niche communities and used in high performance computing.

Recent History

At the beginning of the 21st century processor frequency scaling hit the *power wall*.

Processor vendors decided to provide multiple CPU cores on the same processor chip, each capable of executing separate instruction streams.

Common theme: parallel computing provides computational power when sequential computing cannot do so.

Why Parallel Computing?

Parallel programming is much harder than sequential programming.

- ▶ Separating sequential computations into parallel subcomputations can be challenging, or even impossible.
- ▶ Ensuring program correctness is more difficult, due to new types of errors.

Speedup is the only reason why we bother paying for this complexity.

Parallel Programming vs. Concurrent Programming

Parallelism and concurrency are closely related concepts.

Parallel program – uses parallel hardware to execute computation more quickly. Efficiency is its main concern.

Concurrent program – *may or may not* execute multiple executions at the same time. Improves modularity, responsiveness or maintainability.

Parallelism Granularity

Parallelism manifests itself at different granularity levels.

- ▶ bit-level parallelism – processing multiple bits of data in parallel
- ▶ instruction-level parallelism – executing different instructions from the same instruction stream in parallel
- ▶ task-level parallelism – executing separate instruction streams in parallel

In this course, we focus on task-level parallelism.

Classes of Parallel Computers

Many different forms of parallel hardware.

- ▶ multi-core processors
- ▶ symmetric multiprocessors
- ▶ general purpose graphics processing unit
- ▶ field-programmable gate arrays
- ▶ computer clusters

Our focus will be programming for multi-cores and SMPs.

Summary

Course structure:

- ▶ week 1 – basics of parallel computing and parallel program analysis
- ▶ week 2 – task-parallelism, basic parallel algorithms
- ▶ week 3 – data-parallelism, Scala parallel collections
- ▶ week 4 – data structures for parallel computing

JVM and parallelism

There are many forms of parallelism.

Our parallel programming model assumption – multicore or multiprocessor systems with shared memory.

Operating system and the JVM as the underlying runtime environments.

Processes

Operating system – software that manages hardware and software resources, and schedules program execution.

Process – an instance of a program that is executing in the OS.

The same program can be started as a process more than once, or even simultaneously in the same OS.

The operating system multiplexes many different processes and a limited number of CPUs, so that they get *time slices* of execution. This mechanism is called *multitasking*.

Two different processes cannot access each other's memory directly – they are isolated.

Threads

Each process can contain multiple independent concurrency units called *threads*.

Threads can be started from within the same program, and they share the same memory address space.

Each thread has a program counter and a program stack.

JVM threads cannot modify each other's stack memory. They can only modify the heap memory.

Creating and starting threads

Each JVM process starts with a **main thread**.

To start additional threads:

1. Define a Thread subclass.
2. Instantiate a new Thread object.
3. Call start on the Thread object.

The Thread subclass defines the code that the thread will execute. The same custom Thread subclass can be used to start multiple threads.

Example: starting threads

```
class HelloThread extends Thread {  
  override def run() {  
    println("Hello world!")  
  }  
}
```

```
val t = new HelloThread
```

```
t.start()
```

```
t.join()
```

Time for a demo!

Atomicity

The previous demo showed that separate statements in two threads can overlap.

In some cases, we want to ensure that a sequence of statements in a specific thread executes at once.

An operation is *atomic* if it appears as if it occurred instantaneously from the point of view of other threads.

Let's see a demo:

```
private var uidCount = 0L
def getUniqueId(): Long = {
  uidCount = uidCount + 1
  uidCount
}
```

The synchronized block

The synchronized block is used to achieve atomicity. Code block after a synchronized call on an object *x* is never executed by two threads at the same time.

```
private val x = new AnyRef {}  
private var uidCount = 0L  
def getUniqueId(): Long = x.synchronized {  
    uidCount = uidCount + 1  
    uidCount  
}
```

Different threads use the synchronized block to agree on unique values. The synchronized block is an example of a *synchronization primitive*.

Composition with the synchronized block

Invocations of the synchronized block can nest.

```
class Account(private var amount: Int = 0) {  
  def transfer(target: Account, n: Int) =  
    this.synchronized {  
      target.synchronized {  
        this.amount -= n  
        target.amount += n  
      }  
    }  
}
```

Time for a demo!

Deadlocks

Deadlock is a scenario in which two or more threads compete for resources (such as monitor ownership), and wait for each to finish without releasing the already acquired resources.

```
val a = new Account(50)
```

```
val b = new Account(70)
```

```
// thread T1
```

```
a.transfer(b, 10)
```

```
// thread T2
```

```
b.transfer(a, 10)
```

Resolving deadlocks

One approach is to always acquire resources in the same order.

This assumes an ordering relationship on the resources.

```
val uid = getUniqueId()
private def lockAndTransfer(target: Account, n: Int) =
  this.synchronized {
    target.synchronized {
      this.amount -= n
      target.amount += n
    }
  }
def transfer(target: Account, n: Int) =
  if (this.uid < target.uid) this.lockAndTransfer(target, n)
  else target.lockAndTransfer(this, -n)
```

Memory model

Memory model is a set of rules that describes how threads interact when accessing shared memory.

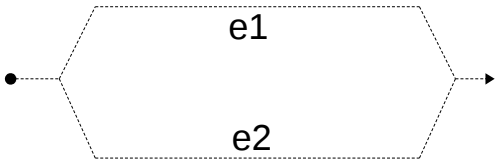
Java Memory Model – the memory model for the JVM.

1. Two threads writing to separate locations in memory do not need synchronization.
2. A thread X that calls join on another thread Y is guaranteed to observe all the writes by thread Y after join returns.

Basic parallel construct

Given expressions e_1 and e_2 , compute them in parallel and return the pair of results

`parallel(e_1 , e_2)`

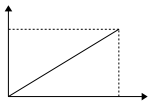


Example: computing p-norm

Given a vector as an array (of integers), compute its p-norm

A *p-norm* is a generalization of the notion of *length* from geometry

2-norm of a two-dimensional vector (a_1, a_2) is $(a_1^2 + a_2^2)^{1/2}$



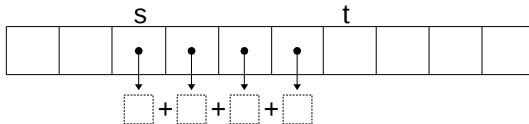
The p-norm of a vector (a_1, \dots, a_n) is $\left(\sum_{i=1}^n |a_i|^p \right)^{1/p}$

Main step: sum of powers of array segment

First, solve *sequentially* the following sumSegment problem: given

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

compute $\sum_{i=s}^{t-1} \lfloor |a_i|^p \rfloor$ where $\lfloor y \rfloor$ rounds down to an integer



Sum of powers of array segment: solution

The main function is

```
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  
}
```

Here power computes $\lfloor |x|^p \rfloor$:

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

Given `sumSegment(a,p,s,t)`, how to compute p-norm?

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

where `N = a.length`

Given `sumSegment(a,p,s,t)`, how to compute p-norm?

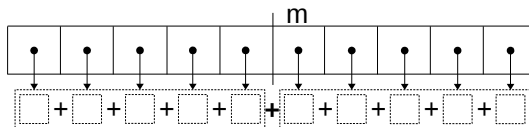
$$\|a\|_p := \left(\sum_{i=0}^{N-1} [|a_i|^p] \right)^{1/p}$$

where `N = a.length`

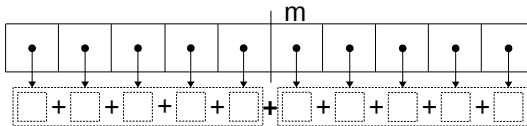
```
def pNorm(a: Array[Int], p: Double): Int =  
  power(sumSegment(a, p, 0, a.length), 1/p)
```

Observe that we can split this sum into two

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



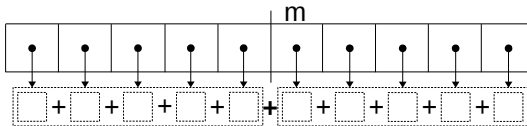
Using sumSegment twice



The resulting function is:

```
def pNormTwoPart(a: Array[Int], p: Double): Int = {  
  val m = a.length / 2  
  val (sum1, sum2) = (sumSegment(a, p, 0, m),  
                     sumSegment(a, p, m, a.length))  
  power(sum1 + sum2, 1/p) }
```

Making two sumSegment invocations parallel

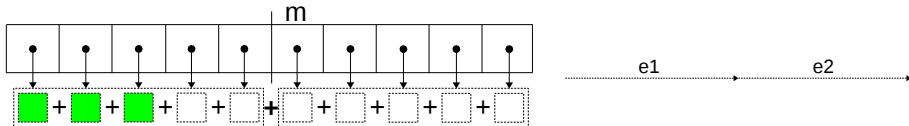


The resulting function is:

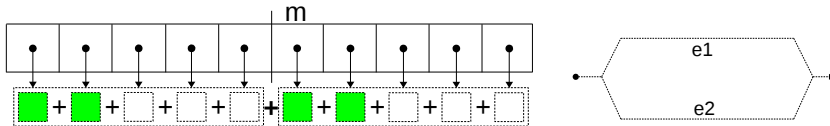
```
def pNormTwoPart(a: Array[Int], p: Double): Int = {  
  val m = a.length / 2  
  val (sum1, sum2) = parallel(sumSegment(a, p, 0, m),  
                              sumSegment(a, p, m, a.length))  
  power(sum1 + sum2, 1/p) }
```


Comparing execution of two versions

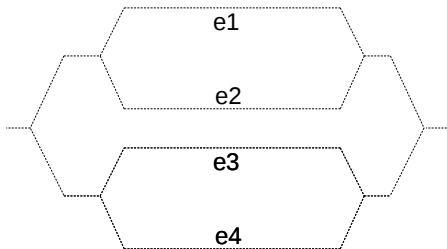
```
val (sum1, sum2) = (sumSegment(a, p, 0, m),  
                    sumSegment(a, p, m, a.length))
```



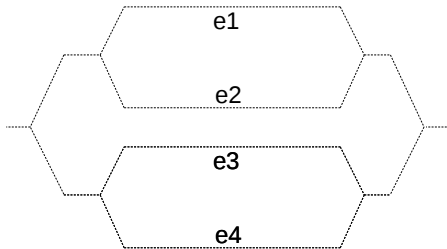
```
val (sum1, sum2) = parallel(sumSegment(a, p, 0, m),  
                             sumSegment(a, p, m, a.length))
```



How to process four array segments in parallel?



How to process four array segments in parallel?



```
val m1 = a.length/4; val m2 = a.length/2; val m3 = 3*a.length/4
val ((sum1, sum2),(sum3,sum4)) =
  parallel(parallel(sumSegment(a, p, 0, m1), sumSegment(a, p, m1, m2)),
    parallel(sumSegment(a, p, m2, m3), sumSegment(a, p, m3, a.length)))
```

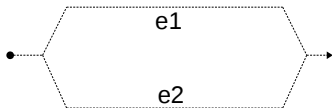
Is there a recursive algorithm for an unbounded number of threads?

Is there a recursive algorithm for an unbounded number of threads?

```
def pNormRec(a: Array[Int], p: Double): Int =  
    power(segmentRec(a, p, 0, a.length), 1/p)  
  
// like sumSegment but parallel  
def segmentRec(a: Array[Int], p: Double, s: Int, t: Int) = {  
    if (t - s < threshold)  
        sumSegment(a, p, s, t) // small segment: do it sequentially  
    else {  
        val m = s + (t - s)/2  
        val (sum1, sum2) = parallel(segmentRec(a, p, s, m),  
                                    segmentRec(a, p, m, t))  
        sum1 + sum2 } }
```

Signature of parallel

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



- ▶ returns the same value as given
- ▶ benefit: `parallel(a,b)` can be faster than `(a,b)`
- ▶ it takes its arguments as *by name*, indicated with `=> A` and `=> B`

parallel is a control structure

Suppose that both `parallel` and `parallel1` contain the same body:

```
def parallel [A, B](taskA: => A, taskB: => B): (A, B) = { ... }  
def parallel1[A, B](taskA:    A, taskB:    B): (A, B) = { ... }
```

If `a` and `b` are some expressions what is the difference between

1. `val (va, vb) = parallel(a, b)`
2. `val (va, vb) = parallel1(a, b)`

parallel is a control structure

Suppose that both `parallel` and `parallel1` contain the same body:

```
def parallel [A, B](taskA: => A, taskB: => B): (A, B) = { ... }  
def parallel1[A, B](taskA:    A, taskB:    B): (A, B) = { ... }
```

If `a` and `b` are some expressions what is the difference between

1. `val (va, vb) = parallel(a, b)`
2. `val (va, vb) = parallel1(a, b)`

The second computation evaluates sequentially, as in `val (va,vb) = (a,b)`

For parallelism, need to pass *unevaluated computations* (call by name).

What happens inside a system when we use parallel?

Efficient parallelism requires support from

- ▶ language and libraries
- ▶ virtual machine
- ▶ operating system
- ▶ hardware

One implementation of parallel uses Java Virtual Machine threads

- ▶ those typically map to operating system threads
- ▶ operating system can schedule different threads on multiple cores

Given sufficient resources, a parallel program can run faster

Underlying Hardware Architecture Affects Performance

Consider code that sums up array elements instead of their powers:

```
def sum1(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 ((sum1, sum2), (sum3, sum4)) = parallel(  
  parallel(sum1(a, p, 0, m1), sum1(a, p, m1, m2)),  
  parallel(sum1(a, p, m2, m3), sum1(a, p, m3, a.length)))
```

Underlying Hardware Architecture Affects Performance

Consider code that sums up array elements instead of their powers:

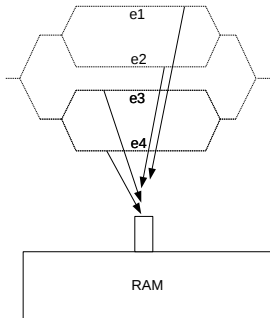
```
def sum1(a: Array[Int], p: Double, s: Int, t: Int): Int = {  
  var i= s; var sum: Int = 0  
  while (i < t) {  
    sum= sum + a(i) // no exponentiation!  
    i= i + 1  
  }  
  sum  
}  
  
val ((sum1, sum2),(sum3,sum4)) = parallel(  
  parallel(sum1(a, p, 0, m1), sum1(a, p, m1, m2)),  
  parallel(sum1(a, p, m2, m3), sum1(a, p, m3, a.length)))
```

Unlike with 'sumSegment', difficult to get speedup for 'sum1'. Why?

Underlying Hardware Architecture Affects Performance

Consider code that sums up array elements instead of their powers:

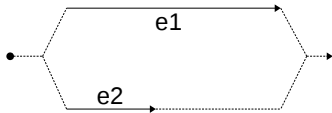
```
def sum1(a: Array[Int], p: Double, s: Int, t: Int): Int = {  
  var i= s; var sum: Int = 0  
  while (i < t) {  
    sum= sum + a(i) // no exponentiation!  
    i= i + 1  
  }  
  sum }  
val ((sum1, sum2),(sum3,sum4)) = parallel(  
  parallel(sum1(a, p, 0, m1), sum1(a, p, m1, m2)),  
  parallel(sum1(a, p, m2, m3), sum1(a, p, m3, a.length)))
```



Memory is a bottleneck

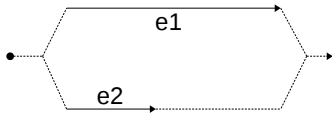
Combining computations of different length with parallel

```
val (v1, v2) = parallel(e1, e2)
```



Combining computations of different length with parallel

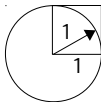
```
val (v1, v2) = parallel(e1, e2)
```



The running time of `parallel(e1, e2)` is the maximum of two running times

A Method to Estimate π (3.14...)

Consider a square and a circle of radius one inside a square:



Ratio between the surfaces of $1/4$ of a circle and $1/4$ of a square:

$$\lambda = \frac{(1^2)\pi/4}{2^2/4} = \frac{\pi}{4}$$

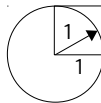
Estimating λ : randomly sample points inside the square

Count how many fall inside the circle

Multiply this ratio by 4 for an estimate of π

Sequential Code for Sampling Pi

```
import scala.util.Random
def mcCount(iter: Int): Int = {
  val randomX = new Random
  val randomY = new Random
  var hits = 0
  for (i <- 0 until iter) {
    val x = randomX.nextDouble // in [0,1]
    val y = randomY.nextDouble // in [0,1]
    if (x*x + y*y < 1) hits = hits + 1
  }
  hits
}
def monteCarloPiSeq(iter: Int): Double = 4.0 * mcCount(iter) / iter
```



Four-Way Parallel Code for Sampling Pi

```
def monteCarloPiPar(iter: Int): Double = {  
  val ((pi1, pi2), (pi3, pi4)) = parallel(  
    parallel(mcCount(iter/4), mcCount(iter/4)),  
    parallel(mcCount(iter/4), mcCount(iter - 3*(iter/4))))  
  4.0 * (pi1 + pi2 + pi3 + pi4) / iter  
}
```

More flexible construct for parallel computation

```
val (v1, v2) = parallel(e1, e2)
```

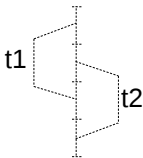
we can write alternatively using the task construct:

```
val t1 = task(e1)
```

```
val t2 = task(e2)
```

```
val v1 = t1.join
```

```
val v2 = t2.join
```



`t = task(e)` starts computation `e` “in the background”

- ▶ `t` is a *task*, which performs computation of `e`
- ▶ current computation proceeds in parallel with `t`
- ▶ to obtain the result of `e`, use `t.join`
- ▶ `t.join` blocks and waits until the result is computed
- ▶ subsequent `t.join` calls quickly return the same result

Task interface

Here is a minimal interface for tasks:

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

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

task and join establish maps between computations and tasks

In terms of the value computed the equation `task(e).join==e` holds

We can omit writing `.join` if we also define an implicit conversion:

```
implicit def getJoin[T](x:Task[T]): T = x.join
```

Example: Starting Four Tasks

We have seen four-way parallel p -norm:

```
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 essentially the same computation expressed using task:

```
val t1 = task {sumSegment(a, p, 0, mid1)}  
val t2 = task {sumSegment(a, p, mid1, mid2)}  
val t3 = task {sumSegment(a, p, mid2, mid3)}  
val t4 = task {sumSegment(a, p, mid3, a.length)}  
power(t1 + t2 + t3 + t4, 1/p)
```

Can we define parallel using task?

Suppose you are allowed to use task

Implement parallel construct as a method using task

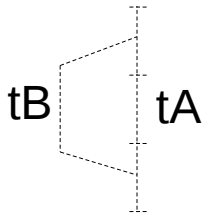
```
def parallel[A, B](cA: => A, cB: => B): (A, B) = {  
  ...  
}
```

Can we define parallel using task?

Suppose you are allowed to use task

Implement parallel construct as a method using task

```
def parallel[A, B](cA: => A, cB: => B): (A, B) = {  
  val tB: Task[B] = task { cB }  
  val tA: A = cA  
  (tA, tB.join)  
}
```



What is wrong with parallelWrong definition?

// CORRECT

```
def parallel[A, B](cA: => A, cB: => B): (A, B) = {  
  val tB: Task[B] = task { cB }  
  val tA: A = cA  
  (tA, tB.join)  
}
```

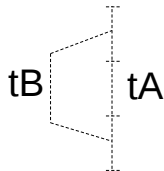
// WRONG

```
def parallelWrong[A, B](cA: => A, cB: => B): (A, B) = {  
  val tB: B = (task { cB }).join  
  val tA: A = cA  
  (tA, tB.join)  
}
```

What is wrong with parallelWrong definition?

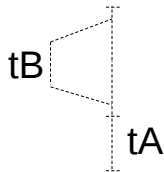
// CORRECT

```
def parallel[A, B](cA: => A, cB: => B): (A, B) = {  
  val tB: Task[B] = task { cB }  
  val tA: A = cA  
  (tA, tB.join)  
}
```



// WRONG

```
def parallelWrong[A, B](cA: => A, cB: => B): (A, B) = {  
  val tB: B = (task { cB }).join  
  val tA: A = cA  
  (tA, tB.join)  
}
```



How long does our computation take?

Performance: a key motivation for parallelism

How to estimate it?

- ▶ empirical measurement
- ▶ asymptotic analysis

Asymptotic analysis is important to understand how algorithms scale when:

- ▶ inputs get larger
- ▶ we have more hardware parallelism available

We examine worst-case (as opposed to average) bounds

Asymptotic analysis of sequential running time

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

Let us review these techniques by applying them to our sum segment example

Asymptotic analysis of sequential running time

Find time bound on sequential `sumSegment` as a function of `s` and `t`

```
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 }
```

Asymptotic analysis of sequential running time

Find time bound on sequential `sumSegment` as a function of `s` and `t`

```
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 }
```

The answer is: $W(s, t) = O(t - s)$, a function of the form: $c_1(t - s) + c_2$

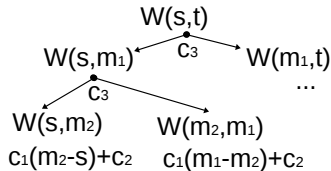
- ▶ $t - s$ loop iterations
- ▶ a constant amount of work in each iteration

Analysis of recursive functions

```
def segmentRec(a: Array[Int], p: Double, s: Int, t: Int) = {  
  if (t - s < threshold)  
    sumSegment(a, p, s, t)  
  else {  
    val m = s + (t - s) / 2  
    val (sum1, sum2) = (segmentRec(a, p, s, m),  
                       segmentRec(a, p, m, t))  
    sum1 + sum2 } }
```

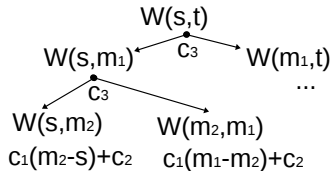
Analysis of recursive functions

```
def segmentRec(a: Array[Int], p: Double, s: Int, t: Int) = {  
  if (t - s < threshold)  
    sumSegment(a, p, s, t)  
  else {  
    val m = s + (t - s) / 2  
    val (sum1, sum2) = (segmentRec(a, p, s, m),  
                       segmentRec(a, p, m, t))  
    sum1 + sum2 } }
```



Analysis of recursive functions

```
def segmentRec(a: Array[Int], p: Double, s: Int, t: Int) = {  
  if (t - s < threshold)  
    sumSegment(a, p, s, t)  
  else {  
    val m = s + (t - s)/2  
    val (sum1, sum2) = (segmentRec(a, p, s, m),  
                        segmentRec(a, p, m, t))  
    sum1 + sum2 } }
```



$$W(s, t) = \begin{cases} c_1(t - s) + c_2, & \text{if } t - s < \text{threshold} \\ W(s, m) + W(m, t) + c_3 & \text{otherwise, for } m = \lfloor (s + t)/2 \rfloor \end{cases}$$

Bounding solution of recurrence equation

$$W(s, t) = \begin{cases} c_1(t - s) + c_2, & \text{if } t - s < \text{threshold} \\ W(s, m) + W(m, t) + c_3 & \text{otherwise, for } m = \lfloor (s + t)/2 \rfloor \end{cases}$$

Assume $t - s = 2^N(\text{threshold} - 1)$, where N is the depth of the tree

Computation tree has 2^N leaves and $2^N - 1$ internal nodes

$$W(s, t) = 2^N(c_1(\text{threshold} - 1) + c_2) + (2^N - 1)c_3 = 2^N c_4 + c_5$$

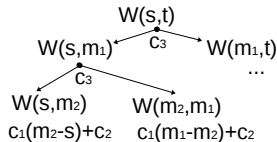
If $2^{N-1} < (t - s)/(\text{threshold} - 1) \leq 2^N$, we have

$$W(s, t) \leq 2^N c_4 + c_5 < (t - s) \cdot 2/(\text{threshold} - 1) + c_5$$

$W(s, t)$ is in $O(t - s)$. Sequential segmentRec is linear in $t - s$

Recursive functions with unbounded parallelism

```
def segmentRec(a: Array[Int], p: Double, s: Int, t: Int) = {  
  if (t - s < threshold)  
    sumSegment(a, p, s, t)  
  else {  
    val m = s + (t - s)/2  
    val (sum1, sum2) = parallel(segmentRec(a, p, s, m),  
                                segmentRec(a, p, m, t))  
    sum1 + sum2 } }
```



$$D(s, t) = \begin{cases} c_1(t - s) + c_2, & \text{if } t - s < \text{threshold} \\ \max(D(s, m), D(m, t)) + c_3 & \text{otherwise, for } m = \lfloor (s + t)/2 \rfloor \end{cases}$$

Solving recurrence with unbounded parallelism

$$D(s, t) = \begin{cases} c_1(t - s) + c_2, & \text{if } t - s < \text{threshold} \\ \max(D(s, m), D(m, t)) + c_3 & \text{otherwise, for } m = \lfloor (s + t)/2 \rfloor \end{cases}$$

Assume $t - s = 2^N(\text{threshold} - 1)$, where N is the depth of the tree

Computation tree has 2^N leaves and $2^N - 1$ internal nodes

The value of $D(s, t)$ in leaves of computation tree: $c_1(\text{threshold} - 1) + c_2$

One level above: $c_1(\text{threshold} - 1) + c_2 + c_3$

Root: $c_1(\text{threshold} - 1) + c_2 + (N - 1)c_3$

Solution bounded by $O(N)$. Also, running time is monotonic in $t - s$

If $2^{N-1} < (t - s)/(\text{threshold} - 1) \leq 2^N$, we have $N < \log(t - s) + c_6$

$D(s, t)$ is in $O(\log(t - s))$

Work and depth

We would like to speak about the asymptotic complexity of parallel code

- ▶ but this depends on available parallel resources
- ▶ we introduce *two measures* for a program

Work $W(e)$: number of steps e would take if there was no parallelism

- ▶ this is simply the sequential execution time
- ▶ treat all $\text{parallel}(e1, e2)$ as $(e1, e2)$

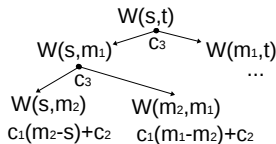
Depth $D(e)$: number of steps if we had unbounded parallelism

- ▶ we take maximum of running times for arguments of parallel

Rules for depth (span) and work

Key rules are:

- ▶ $W(\text{parallel}(e_1, e_2)) = W(e_1) + W(e_2) + c_2$
- ▶ $D(\text{parallel}(e_1, e_2)) = \max(D(e_1), D(e_2)) + c_1$



If we divide work in equal parts, for depth 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 $f(e_1, \dots, e_n)$:

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

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

Note: we assume (reasonably) that constants are such that $D \leq W$

Computing time bound for given parallelism

Suppose we know $W(e)$ and $D(e)$ and our platform has P parallel threads

Regardless of P , cannot finish sooner than $D(e)$ because of dependencies

Regardless of $D(e)$, cannot finish sooner than $W(e)/P$: every piece of work needs to be done

So it is reasonable to use this estimate for running time:

$$D(e) + \frac{W(e)}{P}$$

Given W and D , we can estimate how programs behave for different P

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

Consequences for segmentRec

The call to parallel function `segmentRec` had:

- ▶ work W : $O(t - s)$
- ▶ depth D : $O(\log(t - s))$

On a platform with P parallel threads the running time is, for some constants b_1, b_2, b_3, b_4 :

$$b_1 \log(t - s) + b_2 + \frac{b_3(t - s) + b_4}{P}$$

- ▶ if P is bounded, we have linear behavior in $t - s$
 - ▶ possibly faster than sequential, depending on constants
- ▶ if P grows, the depth starts to dominate the cost and the running time becomes logarithmic in $t - s$

Parallelism and Amdahl's Law

Suppose that we have two parts of a sequential computation:

- ▶ part1 takes fraction f of the computation time (e.g. 40%)
- ▶ part2 take the remaining $1 - f$ fraction of time (e.g. 60%) and we can speed it up

If we make part2 P times faster the speedup is

$$1 / \left(f + \frac{1 - f}{P} \right)$$

For $P = 100$ and $f = 0.4$ we obtain 2.46

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

Testing and Benchmarking

- ▶ testing – ensures that parts of the program are behaving according to the intended behavior
- ▶ benchmarking – computes performance metrics for parts of the program

Typically, *testing* yields a binary output – a program or its part is either correct or it is not.

Benchmarking usually yields a continuous value, which denotes the extent to which the program is correct.

Benchmarking Parallel Programs

Why do we benchmark parallel programs?

Performance benefits are the main reason why we are writing parallel programs in the first place.

Benchmarking parallel programs is even more important than benchmarking sequential programs.

Performance Factors

Performance (specifically, running time) is subject to many factors:

- ▶ processor speed
- ▶ number of processors
- ▶ memory access latency and throughput (affects contention)
- ▶ cache behavior (e.g. false sharing, associativity effects)
- ▶ runtime behavior (e.g. garbage collection, JIT compilation, thread scheduling)

To learn more, see [What Every Programmer Should Know About Memory](#), by Ulrich Drepper.

Measurement Methodologies

Measuring performance is difficult – usually, the a performance metric is a random variable.

- ▶ multiple repetitions
- ▶ statistical treatment – computing mean and variance
- ▶ eliminating outliers
- ▶ ensuring steady state (warm-up)
- ▶ preventing anomalies (GC, JIT compilation, aggressive optimizations)

To learn more, see Statistically Rigorous Java Performance Evaluation, by Georges, Buytaert, and Eeckhout.

ScalaMeter

ScalaMeter is a benchmarking and performance regression testing framework for the JVM.

- ▶ performance regression testing – comparing performance of the current program run against known previous runs
- ▶ benchmarking – measuring performance of the current (part of the) program

We will focus on benchmarking.

Using ScalaMeter

First, add ScalaMeter as a dependency.

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

Then, 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")
```

Demo

Measuring the running time.

JVM Warmup

The demo showed two very different running times on two consecutive runs of the program.

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
- ▶ later, the JVM may choose to apply additional dynamic optimizations
- ▶ eventually, the program reaches *steady state*

ScalaMeter Warmers

Usually, we want to measure steady state program performance.

ScalaMeter Warmer objects run the benchmarked code until detecting steady state.

```
import org.scalameter._

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


Demo

Measuring the stable running time.

ScalaMeter Configuration

ScalaMeter configuration clause allows specifying various parameters, such as the minimum and maximum number of warmup runs.

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

Demo

Measuring the stable running time with verbose output.

ScalaMeter Measurers

Finally, ScalaMeter can measure more than just the running time.

- ▶ `Measurer.Default` – plain running time
- ▶ `IgnoringGC` – running time without GC pauses
- ▶ `OutlierElimination` – removes statistical outliers
- ▶ `MemoryFootprint` – memory footprint of an object
- ▶ `GarbageCollectionCycles` – total number of GC pauses
- ▶ newer ScalaMeter versions can also measure method invocation counts and boxing counts

Demo

Measuring the memory footprint.