CS-206 (EPFL)
Parallelism and Concurrency

David Duan

Spring 2020

The course introduces parallel programming models, algorithms, and data structures, map-reduce frameworks and their use for data analysis, as well as shared-memory concurrency.

Math Prereq: Associativity

Definition 1. An operation f is **associative** iff for every x, y, z, $f(x, f(y, z)) = f(f(x, y), z)$. Alternatively, a binary operation $\otimes$ is associative iff $(a \otimes (b \otimes c)) = ((a \otimes b) \otimes c)$ for every a, b, c.

Definition 2. Define $E(x, y, z) = f(f(x, y), z)$. We say arguments of E can **rotate** if $E(x, y, z) = f(f(x, y), z) = f(f(y, z), x) = E(y, z, x)$.

Theorem 1. If f is commutative and arguments of E can rotate, then f is associative.

Theroem 2. Define binary operation on sets A and B by $f(A, B) = (A \cup B)^*$ where * is any operator on sets. Every f satisfying the following three conditions is associative:

- **Expansion**: $A \subset A^*$.
- **Monotonicity**: $A \subseteq B \implies A^* \subseteq B^*$.
- **Idempotence**: $(A^*)^* = A^*$.

Math Prereq: Monoid

Definition 3. Let S be a set and $f : S \times S \to S$ a binary operation. S with f is a **monoid** if it satisfies the following two axioms:

- **Associativity**: $\forall a, b, c \in S : f(f(a, b), c) = f(a, f(b, c))$.
- **Unique identity element**: $\exists z \in S : \forall a \in S : f(a, z) = f(z, a) = a$; we call z the **identity element**.

1. Parallelism

Parallel computing is a type of computation in which many calculations or the execution of processes are carried out simultaneously. Large problems can often be divided into smaller ones, which can then be solved at the same time.

1.1. Task Parallelism

Task parallelism is a form of parallelization that distributes tasks—concurrently performed by processes or threads—across different processors.

1.1.1. The Parallel Construct

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

The `parallel` construct allows us to run two tasks simultaneously. This behaves like an identity function, but the arguments are ***taken by name*** instead of by value (indicated with `=>`) as the arguments would otherwise get evaluated sequentially. The running time of `parallel(e1, e2)` is the maximum of running times of `e1` and `e2`.

1.1.2. The Task Construct

```
trait Task[A] {
   def join: A // task(e).join == e
}
def task[A](c: => A): Task[A]
```

The statement `t = task(e)` starts computation `e` in the background. The variable `t` is a task that performs the computation of `e`; current computations proceed in parallel with `t`. To obtain the result of `e`, we call `t.join`, which blocks and waits until the result is computed. Subsequent `t.join` calls return immediately with the same result.

1.1.3. Parallel Computing Template

The basic template for parallel computing using the `parallel` construct is as follows.

```
def threshold = { ... } // Spawn more parallel tasks only if this condition is satisfied.

def parFunc(arg: argType): ReturnType = {
   if (not(threshold)) {
      complete_the_task_sequentially()
   } else {
      val (arg1, arg2) = split_computation_into_parts()
      val (res1, res2) = parallel(parFunc(arg1), parFunc(arg2))
      combine_results_to_produce_return_values(res1, res2)
   }
}
```

1.1.4. Asymptotic Analysis for Parallel Functions

Let e be some computational task.

- The **work** of e, $W(e)$, is the number of steps e would take if there is no paralellism.
- The **depth** of e, $D(e)$, is the number of steps e would take if we had unbounded parallelism.

Given two computations e_1 and e_2, we have

- $W(\texttt{parallel}(e_1, e_2)) = W(e_1) + W(e_2) + c$.
- $D(\texttt{parallel}(e_1, e_2)) = \max(D(e_1), D(e_2)) + c$.

The estimated running time for a parallel function with P threads available is given by

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

Amdahl's Law Suppose a sequential computation consists of two parts, each taking fraction f and $(1 - f)$ of the total computation time, respectively. If we can make the second part P times faster (e.g., by using P threads), the total speed up is given by

$$\frac{1}{f + \frac{1-f}{P}}.$$

1.1.5. Parallel Operations

We prefer **arrays** and **immutable trees** over lists and other inherently sequential data structures.

- **Advantages/Disadvantages of Arrays:**
  - \+ Random access to elements and good memory locality.
  - – Expensive concatenation and require disjointness.
- **Advantages/Disadvantages of Immutable Trees:**
  - \+ Combining trees is efficient and no need to worry about disjointness of writes.
  - – High memory allocation due to immutability and high memory access overhead due to bad locality.

See appendix for parallel implementations of **map**, **reduce**, and **scanLeft** on arrays and immutable trees.

- **Fold** takes among others a binary operation, but variants differ:
  - whether it takes an inital element (`fold`) or assumes non-empty list (`reduce`);
  - in which order they combine operations of collection (`left` or `right`).

  Note that for parallelism to work, our operation for fold/reduce must be ***associative***.
- **ScanLeft** produces a list of folds of all list prefixes.

  $$List(a_1, \ldots, a_n).scanLeft(a_0)(f) = List(b_0, b_1, \ldots, b_n)$$
  $$\text{where } b_0 = a_0 \text{ and } b_i = f(b_{i-1}, a_i) \text{ for } i \leq i \leq n.$$

  We use two helpers:
  - **Upsweep**: implement a parallel reduce that preserves the computation tree.
  - **Downsweep**: given a tree of intermediate results, produce values for `scanLeft`.

1.2. Data Parallelism

Data parallelism is a form of parallelization that distributes data across computing nodes, which operate on the data in parallel.

1.2.1. From Fold to Aggregate

```
def foldLeft[B](z: B)(f: (B, A) => B): B
```

Operations `foldLeft/Right`, `reduceLeft/Right`, and `scanLeft/Right` must process the elements sequentially as the first argument for function `f` (of type `B`) comes from the last iteration of `f`.

```
def fold(z: A)(f: (A, A) => A): A
```

The `fold` operation can process the elements in a reduction tree, so it can execute in parallel. However, it can only produce values of the same type as the collection that it is called on. Also, for `fold(z: A)(f: (A, A) => A)` to work correctly, `A` and `f` must form a **monoid**.

```
def aggregate[B](z: B)(f: (B, A) => B, g: (B, B) => B): B
```

The `aggregate` operation can be viewed as a combination of `foldLeft` and `fold`.

- `f`: Function for `foldLeft`, used to process segment of a collection sequentially.
- `g`: Function for `fold`, used to combine individual results from `f`.

1.2.2. Scala Parallel Collections

Scala **parallel collections** are meant to be used in exactly the same way as sequential collections.

- From sequential to parallel [*]: `val parCollection = seqCollection.par`.
- From parallel to sequential: `val seqCollection = parCollection.seq`.

Conceptually, Scala's parallel collections framework parallelizes an operation on a parallel collection by recursively ***splitting*** the collection, applying the operation on each partition of the collection in parallel, and ***re-combining*** all of the results (that were completed in parallel). Because of these concurrent, "out-of-order" semantics of parallel collections, results of ***side-effecting operations*** and ***non-associative operations*** are *non-deterministic*.

For these reasons, we recommend the following best practices:

- Avoid mutations to the same memory locations without proper synchronization. [†]
- Side-effects can be avoided by using the correct combinators.
- Never modify a parallel collection on which a data-parallel operation is in progress.

[*] Collections that are inherently sequential (in the sense that elements must be accessed one after the other), like lists, queues, and streams, are converted to their parallel counterparts by copying the elements into the closest parallel collection. For example, a `List` is converted to a `ParVector`.

[†] Solution: Use a concurrent collection (`import java.util.concurrent._`), which can be mutated by multiple threads.

2. Concurrency

Concurrent programming expresses a program as a set of concurrent computations that execute during overlapping time intervals and that need to coordinate in some way.

2.1. Key Concepts

Deadlock A situation where no thread can make progress because each thread waits on some lock is called a **deadlock**. To prevent the deadlock, we can enforce that locks are always taken in the same order by all threads.

Atomic Variable A **linearizable** operation is one that appears instantaneously with the rest of the system. We also say the operation is performed **atomically**. An **atomic variable** is a memory location that supports linearizable operations.

CAS Atomic operations are usually based on the **compare-and-swap** primitive, which is available as `compareAndSet(oldValue, newValue)` method on atomic variable.

Lock-freedom An operation op is **lock-free** if whenever there is a set of threads executing `op` at least one thread completes the operation after a finite number of steps, regardless of the speed in which the different threads progress.

2.2. Concurrency on JVM

- Use `thread(...)` to spawn a thread.
- Call `t.join()` to wait for t to finish.
- Use `synchronized{...}` for atomic exeecution.
- Use `wait`, `notify`, t for signalling inside monitors.
- Use `@volatile` for safe publication of fields.

2.3. Futures

A **future** is an object holding a value which may become available at some point. This value is usually the result of some other computation:

- If the computation has not yet completed, we say that the `Future` is **not completed**.
- If the computation has completed with a value, we say that the `Future` is **successfully completed** with that value.
- If the computation has completed with an exception, we say that the `Future` **failed** with that exception.

Note that once a `Future` object is given a value or an exception, it becomes in effect immutable – it can never be overwritten.

2.3.1. Defining a Future

- A future: `val future: Future[Any] = Future { body }`
- An async function returning a future: `def future(): Future[Any] = Future { body }`

2.3.2. Accessing Return Values

- `f onComplete { case Success(res) => ...; case Failure(t) => ... }`
- `f foreach { res => for (r <- res) ... }`

2.3.3. Operations

- `def map[B](f: A => B): Future[B]`
- `def flatMap[B](f: A => Future[B]): Future[B]`
- `def zip[B](other: Future[B]): Future[(A, B)]`

2.3.4. Handling Failures

- `def recover[B >: A](pf: PartialFunction[Throwable, B]): Future[B]`
- `def recoverWith[B >: A](pf: PartialFunction[Throwable, Future[B]]): Future[B]`

2.3.5. A Note on For-Comprehensions

A **for-comprehension** is a syntactic sugar for `map`, `flatMap`, and `filter` operations on collections. The general form is `for (s) yield e` where

- `s` is a sequence of generators (e..g, `p <- e`) and filters (e.g., `if f`);
- if there are several generators (equivalent of a nested loop), the last generator (equivalent to the innermost loop variable) varies faster than the first;
- use `{ s }` instead of `( s )` if you want to use multiple lines without using semicolons;
- `e` is an element of the resulting collection.

In general,

```
for (x <- e1) yield e2 <=> e1.map(x => e2)
for (x <- e1 if f) yield e2 <=> for (x <- e1.filter(x => f)) yield e2
for (x <- e1; y <- e2) yield e <=> e1.flatMap(x => for (y <- e2) yield e)
```

The following are equivalent:

```
for {
   i <- 1 until n
   j <- 1 until i
   if isPrime(i + j)
} yield(i, j)

for (i <- 1 until n; j <- 1 until i if is Prime(i + j)) yield (i, j)

(1 until n)
   .flatMap(i => (1 until i).filter(j => isPrime(i+j)))
   .map(j => (i, j))
```

3. Actors

3.1. The Actor Model

The **actor model** is a conceptual model of concurrent computation that treats **actors** as the universal primitive of concurrent computation. In response to a message it receives, an actor can do the following three fundamental actions:

- send a finite number of messages to actors it knows;
- create a finite number of new actors;
- designate the behavior to be applied to the next message.

Looking from the outside, actors are completely encapsulated and isolated from each other. Looking from the inside, actors are effectively single-threaded.

3.2. Key Concepts

State Actor objects will typically contain some variables which reflect possible **states** the actor may be in. Each actor (in Akka) has its own light-weight thread, so you can write your actor code without worrying about concurrency at all.

Behavior Every time a message is processed, it is matched against the current **behavior** of the actor, which is a function that defines the actions to be taken in reaction to the message at that point in time. The behavior may change over time, but it is not visible on the outside.

Mailbox Each actor has exactly one **mailbox** to which all senders enqueue their messages. The default mailbox implementation is FIFO/queue. Enqueuing happens in the time-order of send operations, which means that messages sent from different actors may not have a defined order at runtime due to the apparent randomness of distributing actors across threads. Sending multiple messages to the send target from the same actor, on the other hand, will enqueue them in the same order.

Parent-child relationship Each actor is potentially a **parent**: if it creates **children** for delegating sub-tasks, it will automatically supervise them. The list of children is maintained within the actor's context and the actor has access to it. Modifications to the list are done by spawning or stopping children and these actions are reflected immediately. The actual creation and termination actions happen behind the scenes in an asynchronous, non-blocking way.

Actor reference For encapsulation purposes, it is not possible to look inside an actor and get hold of its state from the outside. Instead, actors are represented to the outside using actor references. An **actor reference** is a subtype of `ActorRef` and is used to support sending messages to the actor it represents. Each actor can access its own reference through the `ActorContext.self` field and can include it in messages to other actors to get replies back.

Message delivery Akka follows two general rules: **at-most-once delivery** (no guaranteed delivery) and **message ordering per sender-receiver pair**. The first rule is typically found also in other actor implementations while the second is specific to Akka.

3.3. Key Methods in Akka

```
object Counter {  // define permitted message types in the companion object
   case class Change(newCount: Int)
   case object Incr
   case object Get
   case object Stop
}

class Counter extends Actor {
   import Counter._  // import permitted message types from the companion object
   def counter(n: Int): Receive = {
      case Change(newCount) => context.become(newCount)
      case Incr => context.become(counter(n+1))
      case Get => sender ! n
      case Stop => context.stop(self)
   }
   def receive = counter(0)
}

val system = ActorSystem("Counter Example")
val counter = system.actorOf(Props[Counter], "counter")
counter != Counter.Change(5); counter != Counter.Incr
counter != Counter.Get; counter != Counter.Stop
```

receive The `receive` function in the `Actor` trait describes the action of an actor upon receiving a message. It is a partial function of type `[Any, Unit]`, indicating that the *any message could come* and that *the actor does not return anything to the sender of the message* at the end.

!, tell The exclamation mark (`!`), also known as `tell`, means "fire-and-forget", e.g. send a message asynchronously and return immediately. It will pick up an implicit argument `sender` from the surroundings. Within the receiving actor, the field `sender` stores the `ActorRef` which sent the message that is currently being processed.

actorOf Actors are created by passing a `Props` instance—descriptions on how to create an actor—and a name string into the `actorOf` factory method which is available on `ActorSystem` and `ActorContext`. Using the `ActorSystem` will create top-level actors, supervised by the actor system's provided guardian actor while using an actor's context will create a child actor. Note the method `actorOf` returns an `ActorRef` of the newly-create actor.

stop Each actor can stop another actor by passing in an `ActorRef` to the `stop` method. This method is often applied to `self`, i.e., an actor wants to stop itself.

become Akka supports hotswapping the actor's message loop (e.g. its implementation) at runtime: invoke the `context.become` method from within the actor. This function takes a `PartialFunction[Any, Unit]` that implements the new message handler. The hotswapped code is kept in a stack which can be pushed and popped.

4. Spark

Apache Spark is an open-source distributed general-purpose cluster-computing framework. Spark provides an interface for programming entire clusters with implicit data parallelism and fault tolerance.

4.1. Overview

SparkContext/SparkSession The `SparkContext` object (renamed `SparkSession`) can be thought of as your ***handle*** to the Spark cluster, which represents the connection between the Spark cluster and your running application.

Resilient Distributed Dataset (RDD) The basic abstraction in Spark. Features include in-memory computation, lazy-evaluation, fault tolerant, immutability, partitioning, persistence, and coarse-grained operations.

4.2. Creating RDDs

RDDs can be created in two ways:

- Transforming an existing RDD (more on this later);
- From a `SparkContext` or `SparkSession` object, which provides useful methods include
  - `parallelize[T](seq: Seq[T]): RDD[T]` - convert a local Scala collection to an RDD;
  - `textFile(path: String)` - construct an RDD based on a (remote or local) text file.

4.3. RDD Operations

RDDs support two types of operations: **transformations**, which create a new dataset from an existing one, and **actions**, which return a value to the driver program after running a computation on the dataset. Most common operations include:

- Tranformations:
  - `map[B](f: A => B): RDD[B]`
  - `flatMap[B](f: A => TraversableOnce[B]): RDD[B]`
  - `filter(pred: A => Boolean): RDD[A]`
  - `distinct(): RDD[B]`
- Actions:
  - `collect(): Array[T]`
  - `count(): Long`
  - `take(num: Int): Array[T]`
  - `reduce(op: (A, A) => A): A`
  - `foreach(f: T => Unit): Unit`

All transformations in Spark are ***lazy***, in that they do not compute their results right away. Instead, they just remember the transformations applied to some base dataset (e.g. a file). The transformations are only computed when an action requires a result to be returned to the driver program. Lazy evaluation of transformations allow Spark to *stage* computations,that is, Spark can make important optimization to the chain of operations before evaluation.

4.4. RDD Operations on Pair RDDs

In Spark, distributed key-value pairs are called **pair RDDs**, which are useful as they allow you to act on each key in parallel or regroup data across the network. Pair RDDs are parameterized by a pair, e.g., `RDD[(K,V)]`, and have additional, specialized methods for working wtih data associated with keys:

- `groupByKey(): RDD[(K, Iterable[A])]`
- `reduceByKey(func: (V, V) => V): RDD[(K, V)]`
- `countByKey(): Map[K, Long]`
- `mapValues[U](f: (V) => U): RDD[(K, U)]`

The **join** transformation combines two pair RDDs into one. The key difference between **inner join** and **outer join** is what happens to the keys when RDDs don't contain the same key. Inner joins return a new RDD combining pairs whose keys are present in both inputs RDDs whereas outer join returns a new RDD containing combined pairs whose keys don't have to present in both input RDDs:

- `join[W](other: RDD[(K, W)]): RDD[(K, (V, W)]`
- `leftOuterJoin[W](other: RDD[(K, W)]): RDD[(K, (V, Option[W])]`
- `rightOuterJoin[W](other: RDD[(K, W)]): RDD[(K, (Option[V], W)]`

4.5. Persistence

By default, RDDs are recomputed each time you run an action on them. This can be expensive (in time) if you use a dataset more than once. Good news is, Spark allows you to control what is cached in memory. To cache an RDD in memory, simply call `persist()` or `cache()` on it. There are many ways to configure how your data is persisted. Possible options include:

1. (default option) in memory as regular Java objects;
2. on disk as regular Java objects;
3. in meory as serialized Java objects (more compact);
4. on disk as serialized Java objects (more compact);
5. both in memory and on disk (spill over to disk to avoid re-computation).

4.6. Partitioning

Data within an RDD is split into several **partitions**, which have the following properties:

- Data in the same partition are guaranteed to be on the same machine.
- Each machine in the cluster has one or more partitions.
- The number of partitions to use is configurable.

There are two kinds of partitioning available in Spark: **hash partitioning** and **range partitioning**. To create RDDs with specific partitionings, you can

- Call `partitionBy` on an RDD, providing an explicit `Partitioner`.
- Using transformations that return RDDs with specific partitioners.

Note the result of `partitionBy` should be persisted. Otherwise, the partitioning is repeatedly applied each time the partitioned RDD is used.

4.7. Shuffling

Certain operations within Spark trigger an event known as the **shuffle**. This is Spark's mechanism for re-distributing data so that it's grouped differently across partitions. This typically involves copying data across executors and machines, making the shuffle a complex and costly operation. In general, a shuffle can occur when the resulting RDD depends on other elements from the same RDD or another RDD. Thus, it is recommended to have your data organized on the cluster and apply operations in a clever order. For example, range partitioning can be used to group together tuples with the same keys. Some methods, such as `reduceByKey` or `join`, involve way less shuffling twhen data is partitioned this way.

4.8. Execution and Cluster Topology

Spark jobs are executed in a **master-worker** model. Intuitively, you interact with the master node when you are writing Spark programs, but the actual jobs are done by worker nodes. The two layers are communicated via a **cluster manager** (e.g., Yarn or Mesos) which allocates resources across clusters and manages scheduling.

More formally, a Spark application is a set of *processes* running on a *cluster*. All processes are coordinated by the **driver program**, which is the process where the `main()` method of your program runs, and is also the one running the code that creates a `SparkContext`, creates RDDs, and stages up or sends off transformations and actions. **Executors**, on the other hand, run the tasks that represent the applications, return comptued results to the driver, and provide in-memory storage for cached RDDs. Let us now look at the execution of a Spark program:

1. The driver program runs the Spark application which creates a `SparkContext` upon start-up.
2. The `SparkContext` connects to a cluster manager which allocates resources.
3. Spark acquires executors on nodes in the cluster, which are processes that run computations and store data for your application.
4. Driver programs sends your application code to the executors.
5. `SparkContext` sends tasks for the executors to run.

4.9. Wide vs Narrow Dependencies

A **lineage graph** is a DAG that represents the computations done on the RDD. In fact, Scala RDDs recover from failures by recompuitng lost partitions from lineage graphs.

Each RDD is amde up of 4 parts in total: ***partitions***, ***dependencies***, ***function*** (for computing the dataset based on its parent RDDs), and ***metadata*** (about its partitioning scheme and data placement). We can classify the dependency between children and parents as follows:

- **Narrow** dependency: each partition of the parent RDD is used by at most ***one*** partition of the child RDD; fast as no shuffling is required.
- **Wide** dependency: each partition of the parent RDD may be used by ***multiple*** child partitions; slow, require some or all data to be shuffled over the network.

A. Parallel Operations

A.1. Parallel Map

A.1.1. Parallel Map on Arrays

```
// write to out(i) for left <= i <= right - 1
def mapASegSeq[A, B](inp: Array[B], left: Int, right: Int, f: A => B, out: Array[B]) = {
   var i = left
   while (i < left) { out(i) = f(inp(i)); i += 1 }
}

def mapASegPar[A, B](inp: Array[A], left: Int, right: Int, f: A => B, out: Array[B]) = {
   if (right - left < threshold) {
      mapASegSeq(inp, left, right, f, out) // sequential base case
   } else {
      val mid = left + (right - left) / 2
      parallel(mapASegPar(inp, left, mid, f, out), mapASegPar(inp, mid, right, f, out))
   }
}
```

A.1.2. Parallel Map on Immutable Trees

```
sealed abstract class Tree[A] { val size: Int }

// Leaves store array segments
case class Leaf[A](a: Array[A]) extends Tree[A] {
   override val size = a.size
}

// Internal nodes store only two subtrees
case class Node[A](l: Tree[A], r: Tree[A]) extends Tree[A] {
   override val size = l.size + r.size
}

def mapTreePar[A: Manifest, B: Manifest](t: Tree[A], f: A => B): Tree[B] = t match {
      case Leaf(a) =>
         val len = a.length; val b = new Array[B](len); var i = 0
         while (i < len) { b(i) = f(a(i)); i += 1 }
         Leaf(b)
      case Node(l, r) =>
         val (lb, rb) = parallel(mapTreePar(l, f), mapTreePar(r, f))
         Node(lb, rb)
}
```

A.2. Parallel Reduce

A.2.1. Parallel Reduce on Arrays

```
def reduceSeg[A](inp: Array[A], left: Int, right: Int, f: (A, A) => A): A = {
   if (right - left < threshold) {
      var res = inp(left); var i = left + 1
      while (i < right) { res = f(res, inp(i)); i += 1 }
      res
   } else {
      val mid = left + (right - left) / 2
      val (a1, a2) = parallel(reduceSeg(inp, left, mid, f),
                              reduceSeg(inp, mid, right, f))
      f(a1, a2)
   }
}

def reduce[A](inp: Array[A], f: (A, A) => A): A = {
   reduceSeg(inp, 0, inp.length, f)
}
```

A.2.2. Parallel Reduce on Trees

```
sealed abstract class Tree[A]

// Leaves store values
case class Leaf[A](values: A) extends Tree[A]

// Internal nodes store two subtrees
case class Node[A](left: Tree[A], right: Tree[A]) extends Tree[A]

def reduce[A](t: Tree[A], f: (A, A) => A): A = t match {
   case Leaf(v) => v
   case Node(l, r) =>
      val (lV, rV) = parallel(reduce[A](l , f), reduce[A](r, f))
      f(lV, rV)
}
```

A.3. Parallel ScanLeft

A.3.1. Parallel ScanLeft on Trees

To reuse intermediate values, we modify the tree data structure so that internal nodes can also store values.

```
sealed abstract class TreeRes[A] { val res: A }
case class LeafRes[A](override val res: A) extends TreeRes[A]
case class NodeRes[A](l: TreeRes[A], override val res: A, r: TreeRes[A])
   extends TreeRes[A]

// A parallel reduce that preserves the computation tree
def upsweep[A](t: Tree[A], f: (A, A) => A): TreeRes[A] = t match {
   case Leaf(v) => LeafRes(v)
   case Node(l, r) =>
      val (tL, tR) = parallel(upsweep(l, f), upsweep(r, f))
      NodeRes(tL, f(tL.res, tR.res), tR)
}

// Produce results; a0 = the result from reduce of all elements left to the tree t
def downsweep[A](t: TreeRes[A], a0: A, f: (A, A) => A): Tree[A] = t match {
   case LeafRes(a) = Leaf(f(a0, a))
   case NodeRes(l, _, r) =>
      val (tL, tR) = parallel(downsweep[A](l, a0, f), downsweep[A](r, f(a0, l.res), f))
      Node(tL, tR)
}

// Main function
def scanLeft(t: Tree[A], a0: A, f: (A, A) => A): Tree[A] = {
   val tRes = upsweep(t, f)
   val scanRes = downsweep(tRes, a0, f)
   prepend(a0, scanRes)
}
```

A.3.2. Parallel ScanLeft on Arrays

We use a tree to store intermediate values. Each leaf keeps track of the array segment `(from, to)` from which `res` is computed, but not the actual array elements. Instead, we pass around a reference to the input array.

```
sealed abstract class TreeResA[A] { val res: A }
case class Leaf[A](from: Int, to: Int, override val res: A) extends TreeResA[A]
case class Node[A](l: TreeRes[A], override val res: A, r: TreeRes[A]) extends TreeRes[A]

def scanLeftSeg[A](inp: Array[A], from: Int, to: Int, a0: A,
                   f: (A, A) => A, out: Array[A]): Unit = {
   out(from) = a0; var a = a0; var i = from
   while (i < to) { a = f(a, inp(i)); i = i + 1; out(i) = a }
}

```

```
11    def upsweep[A](inp: Array[A], from: Int, to: Int, f: (A, A) => A): TreeResA[A] = {
12       if (to - from < threshold) {
13          Leaf(from, to, reduceSeg1(inp, from+1, to, inp(from), f))
14       } else {
15          val mid = from + (to - from) / 2
16          val (tL, tR) = parallel(upsweep(inp, from, mid, f),
17          upsweep(inp, mid, to, f))
18          Node(tL, f(tL.res, tR.res), tR)
19       }
20    }
21
22    def downsweep[A](inp: Array[A], a0: A, f: (A, A) => A, t: TreeResA[A],
23                        out: Array[A]): Unit = t match {
24       case Leaf(from, to, res) => scanLeftSeg(inp, from, to, a0, f, out)
25       case Node(l, _, r) => {
26          val (_, _) = parallel(downsweep(inp, a0, f, l, out),
27          downsweep(inp, f(a0, l.res), f, r, out))
28       }
29    }
30
31    def scanLeft[A](inp: Array[A], a0: A, f: (A, A) => A, out: Array[A]) = {
32       val t = upsweep(inp, 0, inp.length, f)
33       downsweep(inp, a0, f, t, out) // fills [1..inp.length]
34       out(0) = a0 // prepend
35    }
```

B. Synchronization

Implement a thread-safe read/write lock with two methods:

- `read(op)`: performs operation `op` as a reader;
- `write(op)`: performs operation `op` as a writer.

The methods should satisfy the following constraints:

- Only one writer may own the lock at any point in time.
- Multiple readers can own the lock concurrently.
- A writer may own the lock only if no reader owns it.
- Any new writer takes precedence over any new reader.

```
1  class ReadWrite extends Monitor {
2
3     var pendingWriters: Int = 0
4     var readers: Int = 0
5
6     def read[T](op: => T): T = {
7        synchronized {
8           while(pendingWriters > 0) {
9              wait()
10          }
11          readers += 1
12       }
13       try {
14          op
15       } finally {
16          synchronized {
17             readers -= 1
18             if (readers == 0)
19                notifyAll()
20          }
21       }
22    }
23
24    def write[T](op: => Unit): Unit = synchronized {
25       pendingWriters += 1
26       while (readers > 0) {
27          wait()
28       }
29       try {
30          op
31       } finally {
32          pendingWriters -= 1
33          if (pendingWriters == 0)
34             notifyAll()
35          }
36       }
37    }
```