ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Implementing Combiners

Parallel Programming in Scala

Aleksandar Prokopec

Builders

Builders are used in sequential collection methods:

Builders

Builders are used in sequential collection methods:

```
trait Builder[T, Repr] {
  def +=(elem: T): this.type
  def result: Repr
}
```

Combiners

```scala
trait Combiner[T, Repr] extends Builder[T, Repr] {
  def combine(that: Combiner[T, Repr]): Combiner[T, Repr]
}
```

Combiners

```
trait Combiner[T, Repr] extends Builder[T, Repr] {
  def combine(that: Combiner[T, Repr]): Combiner[T, Repr]
}
```

How can we implement the `combine` method *efficiently*?

Combiners

- when `Repr` is a set or a map, `combine` represents union

Combiners

- when `Repr` is a set or a map, `combine` represents union
- when `Repr` is a sequence, `combine` represents concatenation

Combiners

The `combine` operation must be efficient, i.e. execute in $O(\log n + \log m)$ time, where n and m are the sizes of two input combiners.

Combiners

The `combine` operation must be efficient, i.e. execute in $O(\log n + \log m)$ time, where n and m are the sizes of two input combiners.

Question: Is the method `combine` *efficient*?

```
def combine(xs: Array[Int], ys: Array[Int]): Array[Int] = {
  val r = new Array[Int](xs.length + ys.length)
  Array.copy(xs, 0, r, 0, xs.length)
  Array.copy(ys, 0, r, xs.length, ys.length)
  r
}
```

- Yes.
- No.

Array Concatenation

Arrays cannot be efficiently concatenated.

Sets

Typically, set data structures have efficient lookup, insertion and deletion.

Sets

Typically, set data structures have efficient lookup, insertion and deletion.

- hash tables – expected $O(1)$

Sets

Typically, set data structures have efficient lookup, insertion and deletion.

- hash tables – expected $O(1)$
- balanced trees – $O(\log n)$

Sets

Typically, set data structures have efficient lookup, insertion and deletion.

- hash tables – expected $O(1)$
- balanced trees – $O(\log n)$
- linked lists – $O(n)$

Sets

Typically, set data structures have efficient lookup, insertion and deletion.

- hash tables – expected $O(1)$
- balanced trees – $O(\log n)$
- linked lists – $O(n)$

Most set implementations do not have efficient union operation.

Sequences

Operation complexity for sequences can vary.

Sequences

Operation complexity for sequences can vary.

- mutable linked lists – $O(1)$ prepend and append, $O(n)$ insertion

Sequences

Operation complexity for sequences can vary.

- mutable linked lists – $O(1)$ prepend and append, $O(n)$ insertion
- functional (cons) lists – $O(1)$ prepend operations, everything else $O(n)$

Sequences

Operation complexity for sequences can vary.

- mutable linked lists – $O(1)$ prepend and append, $O(n)$ insertion
- functional (cons) lists – $O(1)$ prepend operations, everything else $O(n)$
- array lists – amortized $O(1)$ append, $O(1)$ random accesss, otherwise $O(n)$

Sequences

Operation complexity for sequences can vary.

- mutable linked lists – $O(1)$ prepend and append, $O(n)$ insertion
- functional (cons) lists – $O(1)$ prepend operations, everything else $O(n)$
- array lists – amortized $O(1)$ append, $O(1)$ random accesss, otherwise $O(n)$

Mutable linked list can have $O(1)$ concatenation, but for most sequences, concatenation is $O(n)$.

Parallel Two-Phase Construction

Parallel Programming in Scala

Aleksandar Prokopec

Two-Phase Construction

Most data structures can be constructed in parallel using *two-phase construction*.

Two-Phase Construction

Most data structures can be constructed in parallel using *two-phase construction*.

The *intermediate data structure* is a data structure that:

- has an efficient `combine` method – $O(\log n + \log m)$ or better

Two-Phase Construction

Most data structures can be constructed in parallel using *two-phase construction*.

The *intermediate data structure* is a data structure that:

- has an efficient `combine` method – $O(\log n + \log m)$ or better
- has an efficient `+=` method

Two-Phase Construction

Most data structures can be constructed in parallel using *two-phase construction*.

The *intermediate data structure* is a data structure that:

- has an efficient `combine` method – $O(\log n + \log m)$ or better
- has an efficient `+=` method
- can be converted to the resulting data structure in $O(n/P)$ time

Example: Array Combiner

Let's implement a combiner for arrays.

Two arrays cannot be efficiently concatenated, so we will do a *two-phase construction*.

Example: Array Combiner

Let's implement a combiner for arrays.

Two arrays cannot be efficiently concatenated, so we will do a *two-phase construction*.

```
class ArrayCombiner[T <: AnyRef: ClassTag](val parallelism: Int) {
  private var numElems = 0
  private val buffers = new ArrayBuffer[ArrayBuffer[T]]
  buffers += new ArrayBuffer[T]
```

Example: Array Combiner

First, we implement the += method:

```
def +=(x: T) = {
  buffers.last += x
  numElems += 1
  this
}
```

Example: Array Combiner

First, we implement the += method:

```
def +=(x: T) = {
  buffers.last += x
  numElems += 1
  this
}
```

Amortized $O(1)$, low constant factors – as efficient as an array buffer.

Example: Array Combiner

Next, we implement the `combine` method:

```
def combine(that: ArrayCombiner[T]) = {
  buffers ++= that.buffers
  numElems += that.numElems
  this
}
```

Example: Array Combiner

Next, we implement the `combine` method:

```
def combine(that: ArrayCombiner[T]) = {
  buffers ++= that.buffers
  numElems += that.numElems
  this
}
```

$O(P)$, assuming that `buffers` contains no more than $O(P)$ nested array buffers.

Example: Array Combiner

Finally, we implement the `result` method:

```
def result: Array[T] = {
  val array = new Array[T](numElems)
  val step = math.max(1, numElems / parallelism)
  val starts = (0 until numElems by step) :+ numElems
  val chunks = starts.zip(starts.tail)
  val tasks = for ((from, end) <- chunks) yield task {
    copyTo(array, from, end)
  }
  tasks.foreach(_.join())
  array
}
```

Benchmark

Demo – we will test the performance of the `aggregate` method:

```
xs.par.aggregate(newCombiner)(_ += _, _ combine _).result
```

Two-Phase Construction for Arrays

Two-phase construction works for in a similar way for other data structures. First, partition the elements, then construct parts of the final data structure in parallel:

1. partition the indices into subintervals
2. initialize the array in parallel

Two-Phase Construction for Hash Tables

1. partition the hash codes into buckets
2. allocate the table, and map hash codes from different buckets into different regions

Two-Phase Construction for Search Trees

1. partition the elements into non-overlapping intervals according to their ordering
2. construct search trees in parallel, and link non-overlapping trees

Two-Phase Construction for Spatial Data Structures

1. spatially partition the elements
2. construct non-overlapping subsets and link them

Implementing combiners

How can we implement combiners?

Implementing combiners

How can we implement combiners?

1. Two-phase construction – the combiner uses an intermediate data structure with an efficient `combine` method to partition the elements. When `result` is called, the final data structure is constructed in parallel from the intermediate data structure.

Implementing combiners

How can we implement combiners?

1. Two-phase construction – the combiner uses an intermediate data structure with an efficient `combine` method to partition the elements. When `result` is called, the final data structure is constructed in parallel from the intermediate data structure.
2. An efficient concatenation or union operation – a preferred way when the resulting data structure allows this.

Implementing combiners

How can we implement combiners?

1. Two-phase construction – the combiner uses an intermediate data structure with an efficient `combine` method to partition the elements. When `result` is called, the final data structure is constructed in parallel from the intermediate data structure.
2. An efficient concatenation or union operation – a preferred way when the resulting data structure allows this.
3. Concurrent data structure – different combiners share the same underlying data structure, and rely on *synchronization* to correctly update the data structure when `+=` is called.

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Conc-Trees

Parallel Programming in Scala

Aleksandar Prokopec

List Data Type

Let's recall the list data type in functional programming.

```
sealed trait List[+T] {
  def head: T
  def tail: List[T]
}

case class ::[T](head: T, tail: List[T])
extends List[T]

case object Nil extends List[Nothing] {
  def head = sys.error(”empty list”)
  def tail = sys.error(”empty list”)
}
```

List Data Type

How do we implement a `filter` method on lists?

List Data Type

How do we implement a `filter` method on lists?

```scala
def filter[T](lst: List[T])(p: T => Boolean): List[T] = lst match {
  case x :: xs if p(x) => x :: filter(xs)(p)
  case x :: xs => filter(xs)(p)
  case Nil => Nil
}
```

Trees

Lists are built for sequential computations – they are traversed from left to right.

Trees

Lists are built for sequential computations – they are traversed from left to right.

Trees allow parallel computations – their subtrees can be traversed in parallel.

Trees

Lists are built for sequential computations – they are traversed from left to right.

Trees allow parallel computations – their subtrees can be traversed in parallel.

```
sealed trait Tree[+T]

case class Node[T](left: Tree[T], right: Tree[T])
extends Tree[T]

case class Leaf[T](elem: T) extends Tree[T]

case object Empty extends Tree[Nothing]
```

Filter On Trees

How do we implement a filter method on trees?

Filter On Trees

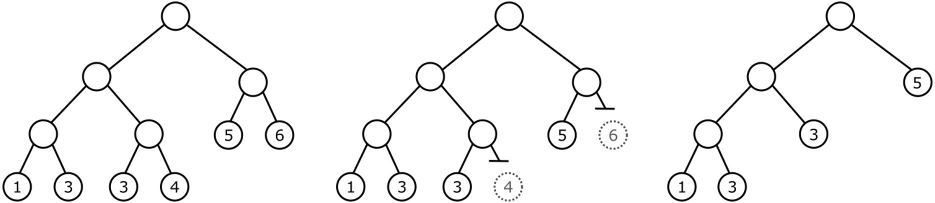
How do we implement a filter method on trees?

```
def filter[T](t: Tree[T])(p: T => Boolean): Tree[T] = t match {
  case Node(left, right) => Node(parallel(filter(left)(p), filter(right)(p)))
  case Leaf(elem) => if (p(elem)) t else Empty
  case Empty => Empty
}
```

Filter On Trees

How do we implement a filter method on trees?

```
def filter[T](t: Tree[T])(p: T => Boolean): Tree[T] = t match {
  case Node(left, right) => Node(parallel(filter(left)(p), filter(right)(p)))
  case Leaf(elem) => if (p(elem)) t else Empty
  case Empty => Empty
}
```



Conc Data Type

Trees are not good for parallelism unless they are balanced.

Conc Data Type

Trees are not good for parallelism unless they are balanced.

Let's devise a data type called `Conc`, which represents balanced trees:

```
sealed trait Conc[+T] {
  def level: Int
  def size: Int
  def left: Conc[T]
  def right: Conc[T]
}
```

In parallel programming, this data type is known as the *conc-list* (introduced in the Fortress language).

Conc Data Type

Concrete implementations of the `Conc` data type:

```
case object Empty extends Conc[Nothing] {
  def level = 0
  def size = 0
}
class Single[T](val x: T) extends Conc[T] {
  def level = 0
  def size = 1
}
case class <>[T](left: Conc[T], right: Conc[T]) extends Conc[T] {
  val level = 1 + math.max(left.level, right.level)
  val size = left.size + right.size
}
```

Conc Data Type Invariants

In addition, we will define the following *invariants* for Conc-trees:

1. A <> node can never contain `Empty` as its subtree.
2. The `level` difference between the left and the right subtree of a <> node is always 1 or less.

Conc Data Type Invariants

In addition, we will define the following *invariants* for Conc-trees:

1. A <> node can never contain `Empty` as its subtree.
2. The `level` difference between the left and the right subtree of a <> node is always 1 or less.

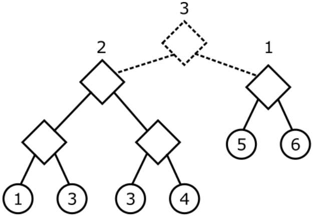
We will rely on these invariants to implement concatenation:

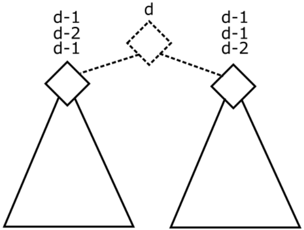
```
def <>(that: Conc[T]): Conc[T] = {
  if (this == Empty) that
  else if (that == Empty) this
  else concat(this, that)
}
```

Concatenation with the Conc Data Type

Concatenation needs to consider several cases.

First, the two trees could have height difference 1 or less:

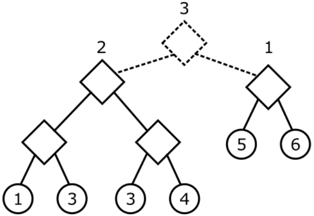


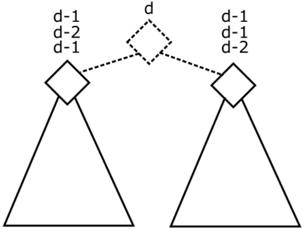


Concatenation with the Conc Data Type

Concatenation needs to consider several cases.

First, the two trees could have height difference 1 or less:





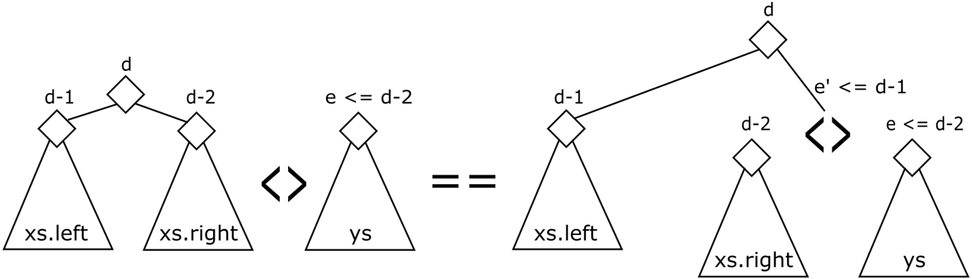

```
def concat[T](xs: Conc[T], ys: Conc[T]): Conc[T] = {
  val diff = ys.level - xs.level
  if (diff >= -1 && diff <= 1) new <>(xs, ys)
  else if (diff < -1) {
```

Concatenation with the Conc Data Type

Otherwise, let's assume that the left tree is higher than the right one.

Concatenation with the Conc Data Type

Otherwise, let's assume that the left tree is higher than the right one.

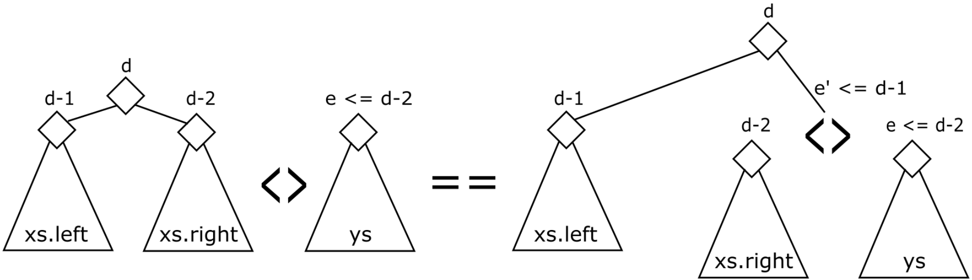


Case 1: The left tree is left-leaning.

Recursively concatenate the right subtree.

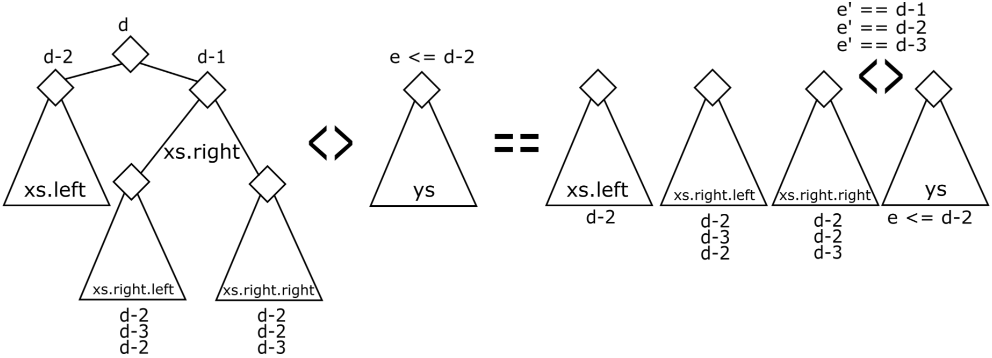
Concatenation with the Conc Data Type

```
if (xs.left.level >= xs.right.level) {
  val nr = concat(xs.right, ys)
  new <>(xs.left, nr)
} else {
```



Concatenation with the Conc Data Type

Case 2: The left tree is right-leaning.



Concatenation with the Conc Data Type

```
} else {
  val nrr = concat(xs.right.right, ys)
  if (nrr.level == xs.level - 3) {
    val nl = xs.left
    val nr = new <>(xs.right.left, nrr)
    new <>(nl, nr)
  } else {
    val nl = new <>(xs.left, xs.right.left)
    val nr = nrr
    new <>(nl, nr)
  }
}
```

Summary

Question: What is the complexity of <> method?

- $O(\log n)$
- $O(h_1 - h_2)$
- $O(n)$
- $O(1)$

Summary

Question: What is the complexity of <> method?

- $O(\log n)$
- $O(h_1 - h_2)$
- $O(n)$
- $O(1)$

Concatenation takes $O(h_1 - h_2)$ time, where h_1 and h_2 are the heights of the two trees.

Amortized Conc-Tree Appends

Parallel Programming in Scala

Aleksandar Prokopec

Constant Time Appends in Conc-Trees

Let's use Conc-Trees to implement a `Combiner`.

How could we implement `+=` method?

```
var xs: Conc[T] = Empty
def +=(elem: T) {
  xs = xs <> Single(elem)
}
```

Constant Time Appends in Conc-Trees

Let's use Conc-Trees to implement a `Combiner`.

How could we implement `+=` method?

```
var xs: Conc[T] = Empty
def +=(elem: T) {
  xs = xs <> Single(elem)
}
```

This takes $O(\log n)$ time – can we do better than that?

Constant Time Appends in Conc-Trees

To achieve $O(1)$ appends with low constant factors, we need to extend the Conc-Tree data structure.

We will introduce a new `Append` node with different semantics:

```
case class Append[T](left: Conc[T], right: Conc[T]) extends Conc[T] {
  val level = 1 + math.max(left.level, right.level)
  val size = left.size + right.size
}
```

Constant Time Appends in Conc-Trees

One possible `appendLeaf` implementation:

```
def appendLeaf[T](xs: Conc[T], y: T): Conc[T] = Append(xs, new Single(y))
```

Constant Time Appends in Conc-Trees

One possible `appendLeaf` implementation:

```
def appendLeaf[T](xs: Conc[T], y: T): Conc[T] = Append(xs, new Single(y))
```

Can we still do $O(\log n)$ concatenation? I.e. can we eliminate `Append` nodes in $O(\log n)$ time?

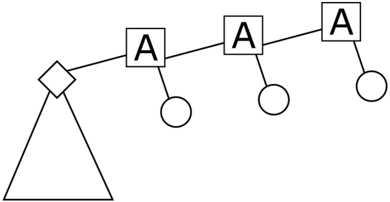
Constant Time Appends in Conc-Trees

One possible `appendLeaf` implementation:

```
def appendLeaf[T](xs: Conc[T], y: T): Conc[T] = Append(xs, new Single(y))
```

Can we still do $O(\log n)$ concatenation? I.e. can we eliminate `Append` nodes in $O(\log n)$ time?

This implementation breaks the $O(\log n)$ bound on the concatenation.



Counting in a Binary Number System

0

W=2^0

Counting in a Binary Number System

1

W=2^0

Counting in a Binary Number System

$$\underset{W=2^0}{1} \;+\; 1$$

$$\underset{W=2^1}{1}\;\;\underset{W=2^0}{0}$$

Counting in a Binary Number System

$$\underset{W=2^0}{1} + 1$$

$$\underset{W=2^1}{1}\ \underset{W=2^0}{0} + 1$$

$$\underset{W=2^1}{1}\ \underset{W=2^0}{1}$$

Counting in a Binary Number System

		1	+	1
		$W=2^0$		
	1	0	+	1
	$W=2^1$	$W=2^0$		
	1	1	+	1
	$W=2^1$	$W=2^0$		
1	0	0		
$W=2^2$	$W=2^1$	$W=2^0$		

Counting in a Binary Number System

$$\underset{W=2^0}{1} + 1$$

$$\underset{W=2^1}{1}\ \underset{W=2^0}{0} + 1$$

$$\underset{W=2^1}{1}\ \underset{W=2^0}{1} + 1$$

$$\underset{W=2^2}{1}\ \underset{W=2^1}{0}\ \underset{W=2^0}{0}$$

- To count up to n in the binary number system, we need $O(n)$ work.

Counting in a Binary Number System

		1	+	1
		W=2^0		
	1	0	+	1
	W=2^1	W=2^0		
	1	1	+	1
	W=2^1	W=2^0		
1	0	0		
W=2^2	W=2^1	W=2^0		

- To count up to n in the binary number system, we need $O(n)$ work.
- A number n requires $O(\log n)$ digits.

Counting in a Binary Number System

$1 + 1$

$W=2^0$

$1 \quad 0 + 1$

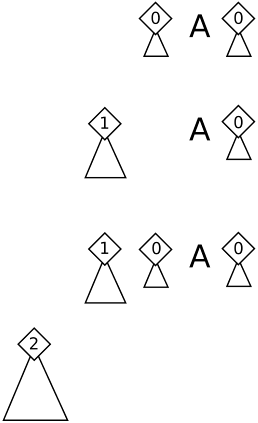
$W=2^1 \quad W=2^0$

$1 \quad 1 + 1$

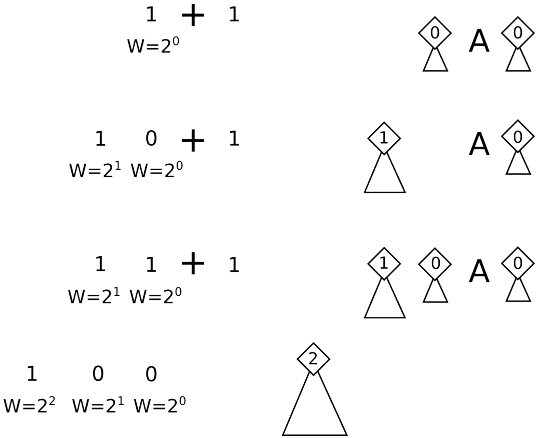
$W=2^1 \quad W=2^0$

$1 \quad 0 \quad 0$

$W=2^2 \quad W=2^1 \quad W=2^0$



Counting in a Binary Number System



- To add n leaves to an Append list, we need $O(n)$ work.
- Storing n leaves requires $O(\log n)$ Append nodes.

Binary Number Representation

- 0 digit corresponds to a missing tree
- 1 digit corresponds to an existing tree

Constant Time Appends in Conc-Trees

```
def appendLeaf[T](xs: Conc[T], ys: Single[T]): Conc[T] = xs match {
  case Empty => ys
  case xs: Single[T] => new <>(xs, ys)
  case _ <> _ => new Append(xs, ys)
  case xs: Append[T] => append(xs, ys)
}
```

Constant Time Appends in Conc-Trees

```
@tailrec private def append[T](xs: Append[T], ys: Conc[T]): Conc[T] = {
  if (xs.right.level > ys.level) new Append(xs, ys)
  else {
    val zs = new <>(xs.right, ys)
    xs.left match {
      case ws @ Append(_, _) => append(ws, zs)
      case ws if ws.level <= zs.level => ws <> zs
      case ws => new Append(ws, zs)
    }
  }
}
```

Constant Time Appends in Conc-Trees

We have implemented an *immutable* data structure with:

- $O(1)$ appends
- $O(\log n)$ concatenation

Next, we will see if we can implement a more efficient, *mutable* data Conc-tree variant, which can implement a `Combiner`.

Conc-Tree Combiners

Parallel Programming in Scala

Aleksandar Prokopec

Conc Buffers

The `ConcBuffer` appends elements into an array of size k.

When the array gets full, it is stored into a `Chunk` node and added into the Conc-tree.

```
class ConcBuffer[T: ClassTag](val k: Int, private var conc: Conc[T]) {
  private var chunk: Array[T] = new Array(k)
  private var chunkSize: Int = 0
```

Conc Buffers

The += operation in most cases just adds an element to the chunk array:

```
final def +=(elem: T): Unit = {
  if (chunkSize >= k) expand()
  chunk(chunkSize) = elem
  chunkSize += 1
}
```

Occasionally, the chunk array becomes full, and needs to be expanded.

Chunk Nodes

`Chunk` nodes are similar to `Single` nodes, but instead of a single element, they hold an array of elements.

```
class Chunk[T](val array: Array[T], val size: Int) extends Conc[T] {
  def level = 0
}
```

Expanding the Conc Buffer

The `expand` method inserts the chunk into the Conc-tree, and allocates a new chunk:

```
private def expand() {
  conc = appendLeaf(conc, new Chunk(chunk, chunkSize))
  chunk = new Array(k)
  chunkSize = 0
}
```

Combine Method

The `combine` method is straightforward:

```
final def combine(that: ConcBuffer[T]): ConcBuffer[T] = {
  val combinedConc = this.result <> that.result
  new ConcBuffer(k, combinedConc)
}
```

Above, the `combine` method relies on the `result` method to obtain the Conc-trees from both buffers.

Result Method

The `result` method packs `chunk` array into the tree and returns the resulting tree:

```
def result: Conc[T] = {
  conc = appendLeaf(conc, new Chunk(chunk, chunkSize))
  conc
}
```

Result Method

The `result` method packs `chunk` array into the tree and returns the resulting tree:

```
def result: Conc[T] = {
  conc = appendLeaf(conc, new Chunk(chunk, chunkSize))
  conc
}
```

Summary:

- $O(\log n)$ `combine` concatenation
- fast $O(1)$ `+=` operation
- $O(1)$ `result` operation

Conc Buffer Demo

Demo – run the same benchmark as we did for the `ArrayCombiner`:

```
xs.par.aggregate(new ConcBuffer[String])(_ += _, _ combine _).result
```