Implementing Combiners

Parallel Programming and Data Analysis

Aleksandar Prokopec
Let's recall combiners from the previous lecture:

```scala
trait Combiner[T, Repr] extends Builder[T, Repr] {
  def combine(that: Combiner[T, Repr]): Combiner[T, Repr]
}
```
Let's recall combiners from the previous lecture:

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

```scala
trait Builder[T, Repr] {
  def +=(elem: T): this.type
  def result: Repr
}
```
Let's recall combiners from the previous lecture:

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

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

- when \( \text{Repr} \) is a set or a map, combine represents union
- when \( \text{Repr} \) is a sequence, combine represents concatenation

The combine operation must be efficient, i.e. execute in \( O(P \cdot \log n) \) time, where \( n \) is the number of elements, and \( P \) is the number of processors.
Combiners

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

The combine operation must be efficient, i.e. execute in $O(P \cdot \log n)$ time, where $n$ is the number of elements, and $P$ is the number of processors.

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

```scala
def concat(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
}
```
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 – $O(\log n)$
Typically, set data structures have efficient lookup, insertion and deletion.

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

Let’s see the operation complexity for sequences.
Sequences

Let’s see the operation complexity for sequences.

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

Let’s see the operation complexity for sequences.

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

Let’s see the operation complexity for sequences.

- 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

Let’s see the operation complexity for sequences.

- 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 access, otherwise $O(n)$

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

Most data structures can be constructed in parallel with \textit{two-phase construction}, which uses an intermediate data structure.
Two-Phase Construction

Most data structures can be constructed in parallel with *two-phase construction*, which uses an intermediate data structure.

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

- has efficient combine method – $O(P \cdot \log n)$ or better
Two-Phase Construction

Most data structures can be constructed in parallel with *two-phase construction*, which uses an intermediate data structure.

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

- has efficient combine method – $O(P \cdot \log n)$ or better
- has efficient $+=\text{ method}$
Two-Phase Construction

Most data structures can be constructed in parallel with *two-phase construction*, which uses an intermediate data structure.

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

- has efficient combine method – $O(P \cdot \log n)$ or better
- has efficient $+=$ method
- the result method is allowed to be $O(n)$, but can be parallelized
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*.

```scala
class ArrayCombiner[T <: AnyRef: ClassTag](val parallelism: Int) {
  private var numElems = 0
  private val buffers = new ArrayBuffer[ArrayBuffer[T]]
  buffers += new ArrayBuffer[T]
```
First, we implement the \(+=\) method:

```python
def +=(elem: T) = {
    buffers.last += elem
    numElems += 1
    this
}
```
Example: Array Combiner

First, we implement the += method:

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

Amortized $O(1)$, low constant factors – as efficient as an array buffer.
Next, we implement the combine method:

```python
def combine(that: ArrayCombiner[T]) = {
    buffers += that.buffers
    numElems += that.numElems
    this
}
```
Next, we implement the combine method:

```python
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.
Finally, we implement the result method:

```scala
def result: Array[T] = {
  val step = math.max(1, numElems / parallelism)
  val array = new Array[T](numElems)
  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
}
```
Demo – we will test the performance of the aggregate method:

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