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.