Conc-Trees for Functional and Parallel Programming

Aleksandar Prokopec, Martin Odersky

École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland

Abstract. Parallel algorithms can be expressed more concisely in a functional programming style. This task is made easier through the use of proper sequence data structures, which allow splitting the data structure between the processors as easily as concatenating several data structures together. Efficient update, split and concatenation operations are essential for declarative-style parallel programs.
This paper shows a functional data structure that can improve the efficiency of parallel programs. The paper introduces two Conc-Tree variants: the Conc-Tree list, which provides worst-case $O(\log n)$ time lookup, update, split and concatenation operations, and the Conc-Tree rope, which additionally provides amortized $O(1)$ time append and prepend operations. The paper demonstrates how Conc-Trees implement efficient mutable sequences, evaluates them against similar persistent and mutable data structures, and shows up to 3× performance improvements when applying Conc-Trees to data-parallel operations.

1 Introduction

Balanced trees are good for *data-parallelism*. They can be easily split between CPUs, so that their subsets are processed independently. Providing efficient concatenation and retaining these properties is challenging, but essential for efficient declarative data-parallel operations. The following data-parallel program maps numbers in the given range by incrementing them:

```
(0 until 1000000).toPar.map(x => x + 1)
```

When the per-element workload is minimal, as is the case with addition, the overwhelming factor of the data-parallel computation is copying the data. Tree data structures can avoid the need for copying results from different processors by providing efficient concatentation. Another use case for trees is efficient parallelization of *task-parallel* functional programs. In the following we compare a *cons-list*-based functional implemenation of the sum method against the *conc-list*-based parallel implementation [16]:

```
def sum(xs: List[Int]) =        def sum(xs: Conc[Int]) =
  xs match {                      xs match {
    case head :: tail =>            case ls <> rs =>
      head + sum(tail)                sum(ls) + sum(rs)
    case Nil => 0 }                 case Single(x) => x }
```

The first `sum` implementation decomposes the data structure `xs` into the first element `head` and the remaining elements `tail`. Sum is computed by recursively adding `head` to the sum of `tail`. This implementation cannot be efficiently parallelized. The second `sum` implementation splits `xs` into two subtrees `ls` and `rs`, and recursively computes their partial sums before adding them together. If `xs` is a balanced tree, the second `sum` implementation can be efficiently parallelized.

In this paper, we describe several variants of the binary tree data-structure called *Conc-Tree*, used to store sequences of elements. The basic variant is *persistent* [11], but we use Conc-Trees to design efficient mutable data structures. Traditionally, persistent data structures are perceived as slower and less efficient than imperative data structures. This paper shows that Conc-Trees are the basis for efficient *mutable* data structures for parallel computing. Data-parallel combiners [12] [13] based on Conc-Trees improve performance of data-parallel operations. Functional task-parallel programming abstractions, such as Fortress Conc-lists [2], can be implemented using Conc-Trees directly. Concretely, the paper describes:

– Conc-Tree lists, with *worst-case* $O(\log n)$ time *persistent* insert, remove and lookup, and *worst-case* $O(\log n)$ *persistent* split and concatenation.
– Conc-Tree ropes, which additionally introduce *amortized* $O(1)$ time *ephemeral* append and prepend operations, and have optimal memory usage.
– Mutable buffers based on Conc-Trees, used to improve data-parallel operation performance by up to $3\times$ compared to previous approaches.

In Section 2, we introduce Conc-Tree lists. We discuss Conc-Tree ropes in Section 3. In Section 4, we apply Conc-Trees to mutable data structures, and in Section 5, we experimentally validate our Conc-Tree implementation. Finally, we give an overview of related work in Section 6.

2 Conc-Tree List

Trees with relaxed invariants are typically more efficient to maintain in terms of asymptotic running time. Although they provide less guarantees on their balance, the impact is small in practice – most trees break the perfect balance by at most a constant factor. Conc-Trees use a classic relaxed invariant seen in red-black and AVL trees [1] – the longest path from the root to a leaf is never more than twice as long than the shortest path from the root to a leaf.

The Conc-Tree data structure consists of several node types. We refer to Conc-Tree nodes with the `Conc` type. This abstract data type has several concrete data types, similar to how the functional `List` data type is either an empty list `Nil` or a `::` (pronounced *cons*) – element and another list. The `Conc` may either be an `Empty`, denoting an empty tree, a `Single`, denoting a tree with a single element, or a `<>` (pronounced *conc*), denoting two separate subtrees.

We show these basic data types in Figure 1. Any `Conc` has an associated `level`, which denotes the longest path from the root to some leaf in that tree. The `level` is defined to be 0 for the `Empty` and `Single` tree, and 1 plus the level of the deeper subtree for the `<>` tree. The `size` of a `Conc` denotes the

```
abstract class Conc[+T] {
  def level: Int
  def size: Int
  def left: Conc[T]
  def right: Conc[T]
  def normalized = this }

abstract class Leaf[T]
extends Conc[T] {
  def left = error()
  def right = error() }

case object Empty
extends Leaf[Nothing] {
  def level = 0
  def size = 0 }
```

```
case class Single[T](x: T)
extends Leaf[T] {
  def level = 0
  def size = 1
}

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

```

Fig. 1. Basic Conc-Tree Data Types

total number of elements contained in the Conc-Tree. The `size` and `level` are cached as fields in the <> type to prevent traversing the tree to compute them each time they are requested. `Conc` trees are persistent like cons-lists – they are never modified after construction. We defer the explanation of the `normalized` method until Section 3 – for now `normalized` just returns the tree.

It is easy to see that the data types described so far can yield imbalanced trees. We can construct arbitrarily large empty trees by combining the `Empty` tree instances with <>. We thus enforce the following invariant – the `Empty` tree can never be a part of <>. However, this restriction is still not sufficient – imbalanced trees can be constructed by iteratively adding elements to the right:

```
(0 until n).foldLeft(Empty: Conc[Int]) {
  (tree, x) => new <>(tree, new Single(x))
}
```

To ensure that the Conc-Trees are balanced, we require that the difference in `levels` of the left subtree and the right subtree is less than or equal to 1. This relaxed invariant imposes bounds on the number of elements. If the tree is completely balanced, i.e. every <> node has two children with equal levels, then the subtree size is $S(level) = 2^{level}$. If we denote the number of elements as $n = S(level)$, it follows that the level of this tree is $level = \log_2 n$.

Next, if the tree is sparse and every <> node at a specific $level$ has two subtrees such that $|left.level - right.level| = 1$, the size of a node at $level$ is:

$$S(level) = S(level-1) + S(level-2), S(0) = 1 \tag{1}$$

This is the familiar Fibonacci recurrence with the solution:

$$S(level) = \frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^{level} - \frac{1}{\sqrt{5}}(\frac{1-\sqrt{5}}{2})^{level} \tag{2}$$

The second addend in the previous equation quickly becomes insignificant, and the level of such a tree is $level = \log_{\frac{1+\sqrt{5}}{2}} n + \log_{\frac{1+\sqrt{5}}{2}} \sqrt{5}$.

From the monotonicity of these recurrences, it follows that $O(\log n)$ is both an upper and a lower bound for the Conc-Tree depth. The bounds also ensure that Conc-Trees have $O(\log n)$ lookup and update operations.

```
def apply(xs: Conc[T], i: Int) = xs match {
  case Single(x) => x
  case left <> right =>
    if (i < left.size) apply(left, i)
    else apply(right, i - left.size) }
def update(xs: Conc[T], i: Int, y: T) =
  xs match {
    case Single(x) => Single(y)
    case left <> right if i < left.size =>
      new <>(update(left, i, y), right)
    case left <> right =>
      val ni = i - left.size
      new <>(left, update(right, ni, y)) }
```

The `update` operation produces a new Conc-Tree such that the element at index `i` is replaced with a new element `y`. This operation only allows replacing existing elements, and we want to insert elements as well. Before showing an $O(\log n)$ `insert` operation, we show how to concatenate two Conc-Trees.

Conc-Tree concatenation is shown in Figure 2. The <> method allows nicer concatenation syntax – the expression `xs <> ys` concatenates two trees together. Note that this is different than the expression `new <>(xs, ys)` that simply links two trees together with one <> node – invoking the constructor directly can violate the balance invariant. We refer to composing two trees together with a <> node as *linking*. Creating a Conc-Tree that respects the invariants and that is the concatenated sequence of the two input trees we call *concatenation*.

The bulk of the concatenation logic is in the `concat` method in Figure 2. This method assumes that the trees are *normalized*, i.e. composed from the basic data types from Figure 1. In explaining the code in Figure 2 we will make an assumption that concatenating two Conc-Trees can yield a tree whose `level` is either equal to the larger input Conc-Tree or greater by exactly 1. In other words, concatenation never increases the Conc-Tree `level` by more than 1. We call this the *height-increase assumption*. We will inductively show that the height-increase assumption is correct while explaining the recursive `concat` method in Figure 2. We skip the trivial base case of merging `Single` trees.

The trees `xs` and `ys` may be in several different relationships with respect to their `levels`. First of all, the absolute difference between the `levels` of `xs` and `ys` could differ by one or less. This is an ideal case – the two trees can be linked directly by creating a <> node that connects them. Otherwise, one tree has a greater `level` than the other one. Without the loss of generality we assume that the left Conc-Tree `xs` is higher than the right Conc-Tree `ys`. To concatenate `xs` and `ys` we need to break `xs` into parts.

```
56 def <>[T](xs: Conc[T], ys: Conc[T]) = {
57   if (xs == Empty) ys
58   else if (ys == Empty) xs
59   else concat(xs.normalized, ys.normalized) }
60 def concat[T](xs: Conc[T], ys: Conc[T]) = {
61   val diff = ys.level - xs.level
62   if (abs(diff) <= 1) new <>(xs, ys)
63   else if (diff < -1) {
64     if (xs.left.level >= xs.right.level) {
65       val nr = concat(xs.right, ys)
66       new <>(xs.left, nr)
67     } else {
68       val nrr = concat(xs.right.right, ys)
69       if (nrr.level == xs.level - 3) {
70         val nr = new <>(xs.right.left, nrr)
71         new <>(xs.left, nr)
72       } else {
73         val nl = new <>(xs.left, xs.right.left)
74         new <>(nl, nrr)
75       } }
76   } else {
77     if (ys.right.level >= ys.left.level) {
78       val nl = concat(xs, ys.left)
79       new <>(nl, ys.right)
80     } else {
81       val nll = concat(xs, ys.left.left)
82       if (nll.level == ys.level - 3) {
83         val nl = new <>(nll, ys.left.right)
84         new <>(nl, ys.right)
85       } else {
86         val nr = new <>(ys.left.right, ys.right)
87         new <>(nll, nr)
88       } } } }
```

Fig. 2. Conc-Tree Concatenation Operation

Assume that `xs.left.level >= xs.right.level`, in other words, that `xs` is left-leaning. The concatenation `xs.right <> ys` in line 65 increases the height of the right subtree by at most 1. This means that the difference in `level`s between `xs.left` and `xs.right <> ys` is 1 or less, so we can link them directly in line 66. Under the height-increase assumption, the resulting tree increases its height by at most 1, which inductively proves the assumption for left-leaning trees.

We next assume that `xs.left.level < xs.right.level`. The subtree `xs.right.right` is recursively concatenated with *ys* in line 68. Its level may be equal to either `xs.level - 2` or `xs.level - 3`. After concatenation we obtain a new tree `nrr` with the level anywhere between `xs.level - 3` and `xs.level - 1`. Note that, if the `nrr.level` is equal to `xs.level - 3`, then the tree `xs.right.left` level is `xs.level - 2`, by the balance invariant. Depending on the level of `nrr` we either link it with `xs.right.left`, or we link `xs.left` with `xs.right.left`, and link the resulting trees once more. Again, the resulting tree does not increase its height by more than 1. This turns the height-increase assumption into the following theorem.

Theorem 1 (Height Increase). *Concatenating two Conc-Tree lists of heights h_1 and h_2 yields a tree with height h such that $|h - max(h_1, h_2)| \leq 1$.*

The bound on the concatenation running time follows directly from the previous theorem and the implementation in Figure 2:

Theorem 2 (Concatenation Time). *Concatenation of two Conc-Tree lists with heights h_1 and h_2 is an $O(|h_1 - h_2|)$ asymptotic running time operation.*

Proof. Direct linking in the concatenation operation is always an $O(1)$ operation. Recursively invoking `concat` occurs at most once on any control path in `concat`. Each time `concat` is called recursively, the height of the higher Conc-Tree is decreased by 1, 2 or 3. Method `concat` will not be called recursively if the absolute difference in Conc-Tree heights is less than or equal to 1. Thus, `concat` can only be called at most $O(|xs_{level} - ys_{level}|)$ times. □

These theorems will be important in proving the running times of the data structures shown later. We now turn to the `insert` operation to show the importance of concatenation on a simple example. The concatenation operation makes expressing the `insert` operation straightforward:

```
def insert[T](xs: Conc[T], i: Int, y: T) =
  xs match {
    case Single(x) =>
      if (i == 0) new <>(Single(y), xs)
      else new <>(xs, Single(y))
    case left <> right if i < left.size =>
      insert(left, i, y) <> right
    case left <> right =>
      left <> insert(right, i - left.size, y) }
```

Insert unzips the tree along a certain path by dividing it into two subtrees and inserting the element into one of the subtrees. That subtree will increase its height by at most one by Theorem 1, making the height difference with its sibling at most two. Merging the two new siblings is thus $O(1)$ by Theorem 2. Since the length of the path from the root to any leaf is $O(\log n)$, the total amount of work done becomes $O(\log n)$. The `split` operation is similar to `insert`, and has $O(\log n)$ complexity by the same argument.

Appending to a Conc-Tree list amounts to merging it with a `Single` tree:

```
def <>[T](xs: Conc[T], x: T) = xs <> Single(x)
```

The downside of appending elements this way is that it takes $O(\log n)$ time. If most of the computation involves appending or prepending elements, this is not satisfactory. We see how to improve this bound in the next section.

3 Conc-Tree Rope

In this section, we modify the Conc-Tree to support an amortized $O(1)$ time ephemeral append operation. The reason that `append` from the last section takes $O(\log n)$ time is that it has to traverse a path from the root to a leaf. Note that the append position is always the same – the rightmost leaf. Even if we could expose that rightmost position by defining the Conc-Tree as a pair of the root and the rightmost leaf, updating the path from the leaf to the root would take $O(\log n)$ time. We instead relax the Conc-Tree invariants.

We introduce a new Conc-Tree node called `Append`, which has a structure isomorphic to the `<>` node. The difference is that the `Append` node does not have the balance invariant – the heights of its `left` and `right` subtrees are not constrained. Instead, we impose the ***append invariant*** on `Append` nodes: the `right` subtree of an `Append` node is never another `Append` node. Furthermore, the `Append` tree cannot contain `Empty` nodes. Finally, only an `Append` node may point to another `Append` node. The `Append` tree is thus isomorphic to a cons-list with the difference that the last node is not `Nil`, but another Conc-Tree.

This data type is transparent to clients and can alternatively be encoded as a special bit in `<>` nodes – clients never observe nor can construct `Append` nodes.

```
case class Append[T](left: Conc[T], right: Conc[T])
extends Conc[T] {
  val level = 1 + left.level.max(right.level)
  val size = left.size + right.size
  override def normalized = wrap(left, right)
}
def wrap[T](xs: Conc[T], ys: Conc[T]) =
  xs match {
    case Append(ws, zs) => wrap(ws, zs <> ys)
    case xs => xs <> ys
  }
```

We implement `normalized` so that it returns the Conc-Tree that contains the same sequence of elements as the original Conc-Tree, but is composed only of the basic Conc-Tree data types in Figure 1. We call this process *normalization*. The method `normalized` in `Append` calls the recursive method `wrap`, which folds the trees in the linked list induced by `Append`.

We postpone claims about the normalization running time, but note that the previously defined `concat` method invokes `normalized` twice and is expected to run in $O(\log n)$ time – `normalized` should not be worse than $O(\log n)$.

We turn to the append operation, which adds a single element at the end of the Conc-Tree. Recall that by using `concat` directly this operation has $O(\log n)$ running time. We now implement a more efficient append operation. The invariant for the `Append` nodes allows appending as follows:

```
def append[T](xs: Conc[T], ys: Single[T]) = new Append(xs, ys)
```

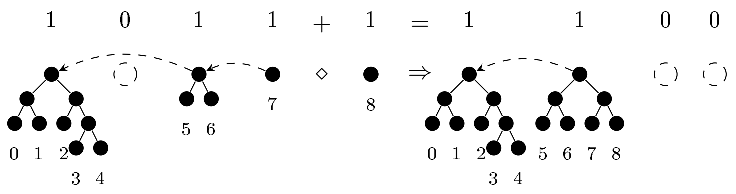


Fig. 3. Correspondence Between the Binary Number System and Append-Lists

Defined like this, `append` is a worst-case constant-time operation, but it has a negative impact on the `normalized` method. Appending n elements results in a long list-like Conc-Tree on which `normalized` takes $O(n \log n)$ time. This `append` implementation illustrates that the more time `append` spends organizing the relaxed Conc-Tree, the less time a `concat` spends later.

Before attempting a different `append` implementation, note the correspondence between a linked list of trees of different `level`s and the digits of different weights in a standard binary number representation. This correspondence is induced by linking two Conc-Tree nodes of the same `level` with a new `<>` node, and adding two binary digits of the same weight. With binary numbers, counting up to n takes $O(n)$ computation steps, where one computation step is rewriting a single digit in the binary representation. Adding 1 is usually an $O(1)$ operation, but the carries chain-react and occasionally require up to $O(\log n)$ rewrites. It follows that adding n `Single` trees in the same way requires $O(n)$ computation steps, where a computation step is linking two trees with the same `level` together – by Theorem 2, an $O(1)$ operation.

We augment the append invariant – if an `Append` node `a` has another `Append` node `b` as the left child, then `a.right.level < b.right.level`. If we now interpret the Conc-Trees under `Append` nodes as binary digits with the weight 2^{level}, we end up with the sparse binary number representation [11]. In this representation, zero digits (missing Conc-Tree levels) are not a part of the physical structure in memory. This correspondence is illustrated in Figure 3, where the binary digits are shown above the corresponding Conc-Trees and the dashed line represents the linked list formed by the `Append` nodes.

Figure 4 shows the `append` operation that executes in $O(1)$ amortized time. The link operation in line 118, which corresponds to adding binary digits, occurs only for adjacent trees that happen to have the same `level`. The trees in the append list are in a form that is friendly to normalization. This list of trees of increasing size is such that the height of the largest tree is $O(\log n)$, and no two trees have the same height. It follows that there are no more than $O(\log n)$ such trees. Furthermore, the sum of the height differences between adjacent trees is $O(\log n)$. By Theorem 1 concatenating any two adjacent trees y and z in the strictly decreasing sequence t^*xyzs^* yields a tree with a height no larger than the height of x. By Theorem 2, the total amount of work required to merge $O(\log n)$ such trees is $O(\log n)$. Thus, appending in a way analogous to incrementing binary numbers ensures $O(\log n)$ normalization.

```
def append[T](xs: Conc[T], ys: Leaf[T]) =
  xs match {
    case Empty => ys
    case xs: Leaf[T] => new <>(xs, ys)
    case _ <> _ => new Append(xs, ys)
    case xs: Append[T] => append(xs, ys) }
private def append[T](xs: Append[T], ys: 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 <= xs.level) ws <> zs
        else new Append(ws, zs) } }
```

Fig. 4. Append Operation

Note that the public `append` method takes a `Leaf` node instead of a `Single` node. The conc-lists from Section 2 and their variant from this section have a high memory footprint. Using a separate leaf to represent each element is inefficient. Traversing the elements in such a data structure is also suboptimal. Conc-Tree travesal (i.e. a `foreach`) must have the same running time as array traversal, and memory consumption should correspond to the memory footprint of an array. We therefore introduce a new type of a `Leaf` node, called a `Chunk`, that packs the elements more tightly together. As we will see in Section 4, this also ensures an efficient imperative += operation.

```
case class Chunk[T](xs: Array[T], size: Int, k: Int)
extends Leaf[T] { def level = 0 }
```

The `Chunk` node contains an array `xs` with `size` elements. The additional argument `k` denotes the maximum size that a `Chunk` can have. The `insert` operation from Section 2 must copy the target `Chunk` when updating the Conc-Tree, and divides the `Chunk` into two if `size` exceeds `k`. Similarly, a `remove` operation fuses two adjacent `Chunks` if their total size is below a threshold.

The *Conc-Tree rope* has one limitation. When used persistently, it is possible that we obtain an instance of the Conc-Tree whose next `append` triggers a chain of linking operations. If we repetitively use that instance of the tree for appending, we lose the amortized $O(1)$ running time. Thus, when used persistently, the Conc-Tree rope has $O(\log n)$ appends. This limitation is overcome by another Conc-Tree variant called a *conqueue*, described in related work [12]. Conc-Tree ropes are nonetheless useful, since their simplicity ensures good constant factors and $O(1)$ ephemeral use. In fact, many applications, such as data-parallel *combiners* [13], always use the most recent version of the data structure.

```
class ConcBuffer[T](val k: Int) {
  private var conc: Conc[T] = Empty
  private var ch: Array[T] = new Array(k)
  private var lastSize: Int = 0
  def +=(elem: T) {
    if (lastSize >= k) expand()
    ch(lastSize) = elem
    lastSize += 1 }
  private def expand() {
    conc = append(conc, new Chunk(ch, lastSize, k))
    ch = new Array(k)
    lastSize = 0 } }
```

Fig. 5. Conc-Buffer Implementation

4 Mutable Conc-Trees

Most of the data structures shown so far were persistent. This persistence comes at a cost – while adding a single node has an $O(1)$ running time, the constant factors involved with allocating objects are still large. In Figure 5, we show the `ConcBuffer` data structure, which uses Conc-Tree ropes as basic building blocks. This mutable data structure maintains an array segment to which it writes appended elements. Once the array segment becomes full, it is pushed into the Conc-Tree as a `Chunk` node, and a new array segment is allocated.

Although combiners based on growing arrays have $O(1)$ appends [13], resizing requires writing an element to memory twice on average. Conc-ropes with `Chunk` leaves ensure that every element is written only once. The larger the maximum chunk size `k` is, the less often is a `Conc` operation invoked in the method `expand` – this amortizes Conc-rope append cost, while retaining fast traversal. The `ConcBuffer` shown above is much faster than Java `ArrayList` or C++ `vector` when appending elements, and at the same time supports efficient concatenation. The underlying persistent Conc-rope allows an efficient copy-on-write snapshot operation.

5 Evaluation

In this section, we compare Conc-Trees against fundamental sequences in the Scala standard library – functional cons-lists, array buffers and Scala Vectors. In a cons-list, prepending an element is highly efficient, but indexing, updating or appending an elements are $O(n)$ time operations. Scala `ArrayBuffer` is a resizeable array known as the `ArrayList` in Java and as `vector` in C++. Array buffers are mutable random access sequences that can index or update elements with a simple memory read or write. Appending is amortized $O(1)$, as it occasionally resizes the array, and rewrites all the elements. An important limitation is that append takes up to 2 memory writes on average. Scala (and Clojure) Vectors are efficient trees that can implement mutable and persistent sequences. Their defining features are low memory consumption and efficient prepending and appending. Current implementations do not have concatenation.

We compare different Conc-Tree variants: lists, ropes, mutable Conc-Buffers, as well as conqueues, described in related work [12].

We execute the benchmarks on an Intel i7 3.4 GHz quad-core processor. We start with *traversal* – we evaluate the `foreach` on persistent Conc-Tree lists from Section 2 and compare it to the `foreach` on the functional cons-list in Figure 6A. Traversing the cons-list is tail recursive and does not use the call stack. Furthermore, Conc-Tree list traversal visits more nodes compared to cons-lists. Therefore, traversing the basic Conc-Tree list is slower than traversing a cons-list. On the other hand, the `Chunk` nodes ensure efficient traversal, as shown in Figure 6B. For $k = 128$, Conc-Tree traversal running time is $2\times$ faster than that of Scala Vector. In subsequent benchmarks we set k to 128.

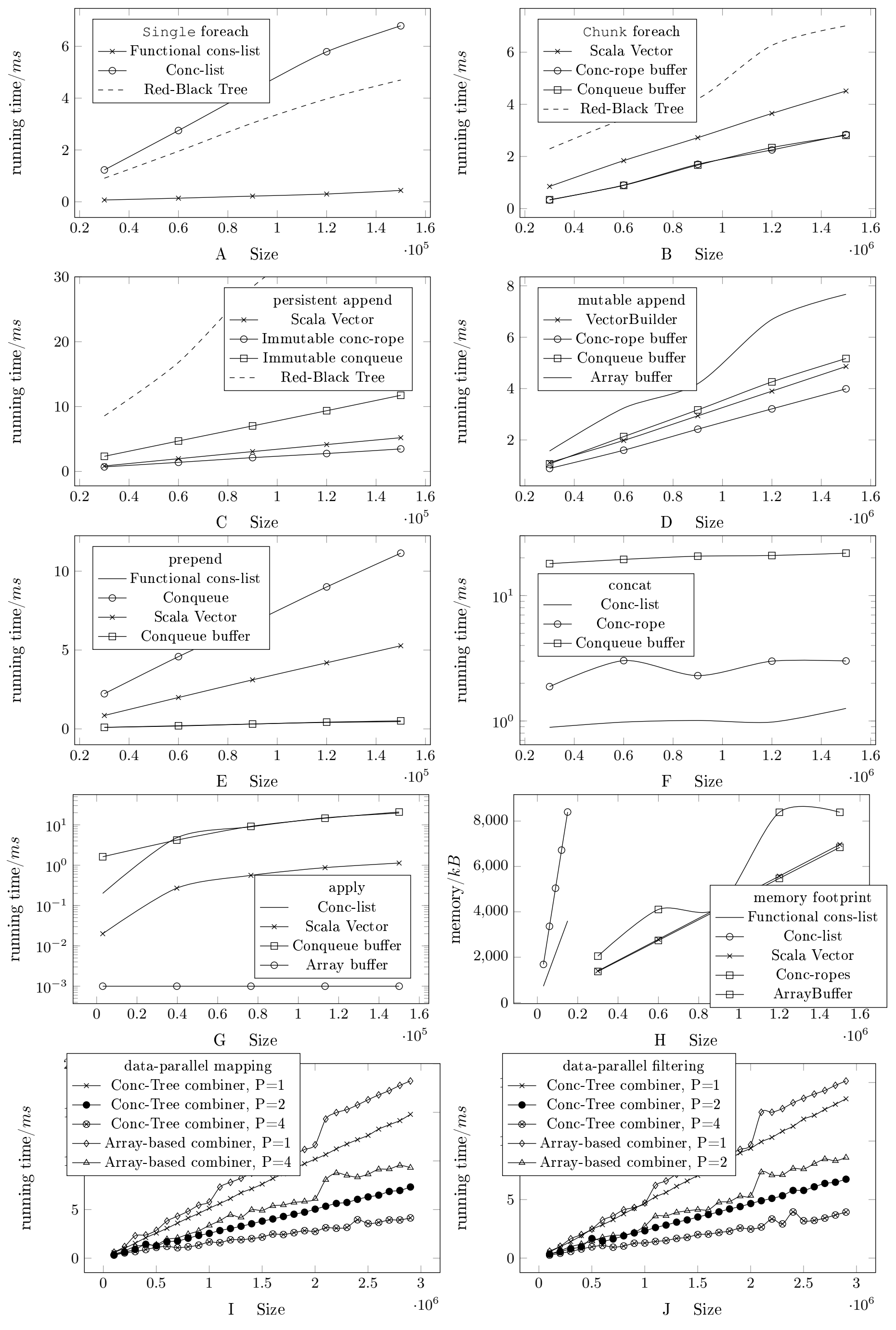
Appending is important for data-parallel transformations. While higher constant factors result in $2\times$ slower conqueue appends compared to *persistent* Vectors, *persistent* Conc-Tree rope append is faster (Figure 6C). For comparison, inserting into a red-black tree is approximately $4\times$ slower than appending to a conqueue. In Figure 6D, we compare Conc-Tree buffers against *mutable* Scala Vectors. Resizeable array appends are outperformed by all other data structures.

When it comes to *prepending* elements, cons-lists are very fast – prepending amounts to creating a single node. Cons-list have the same performance as *mutable* conqueue buffers, even though cons-lists are persistent. Both Scala Vectors and persistent conqueues are an order of magnitude slower.

Concatenation has the same performance for both persistent and mutable Conc-Tree variants. Concatenating mutable variants requires taking a snapshot, which can be done lazily in constant-time [14]. We show concatenation performance in Figure 6F, where we repeat concatenation 10^4 times. Concatenating Conc-ropes is slightly more expensive than conc-list concatenation because of the normalization, and it varies with size because the number of trees (that is, non-zeros) in the append list fluctuates. Conqueue concatenation is slower (note the log axis) due to the longer normalization process. Concatenating lists, array buffers and Scala Vectors is not shown here, as it is a linear time operation, and thousands of times slower for the same number of elements.

Random access is an operation where Scala Vectors have a clear upper hand over the other persistent sequences. Although indexing a Scala Vector is faster than indexing Conc-Trees, both are orders of magnitudes slower than array random access. We note that applications that really need random-access performance must use arrays for indexing operations, and avoid `Vector` altogether.

We show *memory consumption* in Figure 6H. While a Conc-Tree list occupies twice as much memory as a functional cons-list, using `Chunk` nodes has a clear impact on the memory footprint – arrays, Scala Vectors and Conc-Trees with `Chunk` nodes occupy an almost optimal amount of memory, where optimal is the number of elements in the data structure multiplied by the pointer size. Resizeable arrays waste up to 50% of space due to their resizing policy.



A - `foreach` w/o `Chunks`, B - `foreach` with `Chunks`, C - `append` w/o `Chunks`, D - `append` with `Chunks`, E - `prepend` w/o `Chunks`, F - Concatenation, G - Random Access, H - Memory Footprint, I - Data-parallel Mapping with Identity, J - Data-parallel Filtering

Fig. 6. Conc-Tree Benchmarks (smaller is better)

Data-parallel operations are the main use-case for Conc-Trees. Scala collection framework defines high-level collection combinators, such as filtering, grouping, mapping and scanning. This API is similar to high-level data-processing APIs such as FlumeJava and Apache Spark. The example from Section 1 shows how to map numbers from a parallel range of numbers using the `map` operation. This `map` operation works by parts of the parallel range across different processors, and producing parts of the resulting collection in parallel. The lambda function `x => x + 1` is used on each input element to produce an output element. After independent processors produce intermediate collections, their results must be merged into a new collection. When the resulting collection is an array, intermediate array chunks cannot be simply linked together – instead, a new array must be allocated, and intermediate results must be copied into it. The array cannot be preallocated, because in general the number of output elements is not known in advance – in most data-parallel operations, a single input element can map into any number of output elements, determined after the lambda is run.

In the ScalaBlitz parallel collection framework [13] [15], the unifying abstraction that allows expressing different parallel operations on Scala collections generically, is called a *combiner*. The combiner defines three generic operations: adding a new element to the combiner (invoked every time a new output element is created), merging two combiners (invoked when combiners from two different processors are merged), and producing the final collection (which is invoked once at the end of the operation). The arrays created from the parallel ranges in the `map` operation use a special array-based combiner, as described above.

We replaced the standard array-based combiner implementation in ScalaBlitz with Conc-Tree-based combiners, and compared *data-parallel map operation* performance with and without Conc-Trees in Figure 6I, and *data-parallel filter operation* performance in Figure 6J.

With Conc-Trees, performance of the data-parallel mapping is improved by $2 - 3\times$. The reason for this improvement is two-fold. First, array chunks stored inside Conc-Trees do not need bulk resizes, which array-based combiners periodically do. This is visible in Figure 6I,J, where the array-based combiner has spikes at certain input collection sizes. Second, Conc-Tree-based combiners avoid copying each element twice, since intermediate Conc-Trees from different processors can be efficiently merged without copying.

6 Related Work

Standard programming language libraries come with resizeable array implementations, e.g. the `ArrayList` in the JDK or the `vector` in C++ standard template library. These are mutable data structures that provide $O(1)$ worst case time indexing and update operations, with $O(1)$ amortized time append operation. Although appending is amortized $O(1)$, each append on average requires two writes to memory, and each memory location is allocated twice. Concatenation is an $O(n)$ operation. Cons-lists have an efficient push-head and pop-head, but other operations are $O(n)$.

Ropes are heavily relied upon in the Xerox Cedar environment [5], where bulk rebalancing is done after the rope becomes particularly skewed. These ropes have an amortized $O(\log n)$ operation complexity. VList [3] is a functional sequence, with logarithmic time lookup operations. Scala Vector [4] is a persistent sequence implementation. Its dequeue operation has low constant factors, but requires $O(\log n)$ time. Scala Vector does not support concatentation, since concatenation support slows down other operations.

The idea of Conc lists was proposed in the Fortress language [2], where parallel programs are expressed as recursion and pattern matching on three types of nodes – empty, single element or conc nodes [16]. All Conc-Tree variants from this paper provide the same programming model as conc-lists from Fortress.

Relaxing the balancing requirements to allow efficient updates was first proposed by Adelson-Velsky and Landis, in the AVL tree data structure [1]. Okasaki was one of the first to bridge the gap between amortization and persistence through the use of lazy evaluation [9]. While persistent random access lists rely on binary number representations to achieve efficient append operations, they are composed from complete trees of different heights, and do not support concatenation as a consequence [11].

The recursive slowdown techniques were worked on by Kaplan and Tarjan [7]. Previously, persistent sequence data structures were proposed that achieve constant time prepend and append operations, and asymptotic constant time concatenation [8]. Although asymptotic bounds of these data structures are better than that of Conc-Trees, their operations have higher constant factors, and increased implementation complexity. The catenable real-time queues due to Okasaki allow efficient concatenation but do not have the balanced tree structure required for parallelization, nor support logarithmic random access [10]. Hinze and Paterson describe a lazy finger tree data structure [6] with *amortized* constant time deque and concatenation operations.

7 Conclusion

This paper introduces Conc-Tree data structures for functional parallel programming with worst-case $O(\log n)$ time splitting and concatenation. The Conc-Tree list comes with a worst-case $O(\log \frac{n_1}{n_2})$ time concatenation with low constant factors. The Conc-Tree rope provides an amortized $O(1)$ time append and prepend operations. In terms of absolute performance, persistent Conc-Trees outperform existing persistent data structures such as AVL trees and red-black trees by a factor of $3-4\times$, and mutable Conc-Trees outperform mutable sequence data structures such as mutable Vectors and resizeable arrays by $20-50\%$, additionally providing efficient concatenation. Data-parallel operation running time can be improved by up to $3\times$, depending on the workload characteristic.

When choosing between different Conc-Tree variants, we advise the use of ropes for most applications. Although Conc-Tree ropes achieve amortized bounds, ephemeral use is typically sufficient.

Besides serving as a catenable data-type for functional task-parallel programs, and improving the efficiency of data-parallel operations, the immutable nature of Conc-Trees makes them amenable to linearizable concurrent snapshot operations [12]. Inefficiencies associated with persistent data can be amortized to a near-optimal degree, so we expect Conc-Trees to find their applications in future concurrent data structures.

References

1. G. M. Adelson-Velsky and E. M. Landis. An algorithm for the organization of information. *Doklady Akademii Nauk SSSR*, 146:263–266, 1962.
2. Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy Steele, and Sam Tobin-Hochstadt. The Fortress Language Specification. Technical report, Sun Microsystems, Inc., 2007.
3. Phil Bagwell. Fast Functional Lists, Hash-Lists, Deques, and Variable Length Arrays. Technical report, 2002.
4. Philip Bagwell and Tiark Rompf. RRB-Trees: Efficient Immutable Vectors. Technical report, 2011.
5. Hans-J. Boehm, Russ Atkinson, and Michael Plass. Ropes: An alternative to strings. *Softw. Pract. Exper.*, 25(12):1315–1330, December 1995.
6. Ralf Hinze and Ross Paterson. Finger trees: A simple general-purpose data structure. *J. Funct. Program.*, 16(2):197–217, March 2006.
7. Haim Kaplan and Robert E. Tarjan. Persistent lists with catenation via recursive slow-down. STOC '95, pages 93–102, New York, NY, USA, 1995. ACM.
8. Haim Kaplan and Robert E. Tarjan. Purely functional representations of catenable sorted lists. In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pages 202–211, New York, NY, USA, 1996. ACM.
9. Chris Okasaki. *Purely Functional Data Structures.* PhD thesis, Pittsburgh, PA, USA, 1996. AAI9813847.
10. Chris Okasaki. Catenable double-ended queues. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pages 66–74. ACM Press, 1997.
11. Chris Okasaki. *Purely Functional Data Structures.* Cambridge University Press, NY, USA, 1998.
12. Aleksandar Prokopec. *Data Structures and Algorithms for Data-Parallel Computing in a Managed Runtime.* PhD thesis, EPFL, 2014.
13. Aleksandar Prokopec, Phil Bagwell, Tiark Rompf, and Martin Odersky. A generic parallel collection framework. In *Proceedings of the 17th international conference on Parallel processing - Volume Part II*, Euro-Par'11, pages 136–147, Berlin, Heidelberg, 2011. Springer-Verlag.
14. Aleksandar Prokopec, Nathan Grasso Bronson, Phil Bagwell, and Martin Odersky. Concurrent tries with efficient non-blocking snapshots. PPoPP '12, pages 151–160, New York, NY, USA, 2012. ACM.
15. Aleksandar Prokopec, Dmitry Petrashko, and Martin Odersky. On Lock-Free Work-stealing Iterators for Parallel Data Structures. Technical report, 2014.
16. Guy Steele. Organizing functional code for parallel execution; or, foldl and foldr considered slightly harmful. International Conference on Functional Programming (ICFP), 2009.