

A New Approach for Parallel Functional Arrays

Ananya Kumar

Carnegie Mellon University
ananyak@andrew.cmu.edu

Guy Blelloch

Carnegie Mellon University
blelloch@cs.cmu.edu

Abstract

In this paper we introduce a $O(1)$ wait-free, parallel, functional array, that allows $O(1)$ reads and writes to the most recent version of the array. We describe the cost dynamics and sketch out a provable implementation. We show favorable benchmarks comparing our functional arrays with regular arrays in Java.

Categories and Subject Descriptors CR-number [subcategory]: third-level

Keywords array, parallel, cost semantics

1. Introduction

Arrays are very important in functional programming languages because they allow work-efficient implementations of algorithms like depth-first search. Accessing old versions of arrays can be useful for efficient checkpointing, logging, and event handling. In this paper we introduce an efficient, parallel, functional array that allows $O(1)$ reads and writes to the most recent version of the array.

2. Previous Approaches

Many functional programming languages use monads to build arrays. However, monads do not allow accessing old versions of an array.

Another approach is to use compiler based reference counting. If the number of references to an array is provably one, the compiler allows the array to mutate, otherwise the compiler copies the array before writing to the array. This approach makes it difficult for programmers to reason about the time complexity of array operations because it depends on the compiler. Furthermore, if multiple variables reference the same array, the array is copied even if all writes are to the most recent version. This could make the time complexity of programs significantly higher than they need to be.

A third approach is to use linear types to ensure that programmers can only access and mutate arrays in valid ways. This approach is very difficult to implement in existing programming languages.

O’Neill’s functional arrays guarantee $O(1)$ access and insertions when used like an imperative array. However, the arrays use version trees which are inefficient in practice. Additionally, the reads and writes to the most recent version of the array might take $O(\log n)$ work if the array is not used like an imperative array.

Chuang’s approach supports $O(1)$ accesses and insertions to the most recent version of arrays, however accessing old versions of the array takes $O(n)$ work which is often impractical.

None of the existing approaches support concurrent operations on arrays. O’Neill suggests having a lock for each element of the array. However, when many threads contend for the same element, this would serialize accesses and make accesses not $O(1)$. Additionally, per-element locks add significant memory overhead. Separation logic could be used to parallelize divide and conquer algorithms on arrays however they would severely restrict the way functional arrays can be used.

3. Dynamics

We use a standard applicative-order language defined as follows:

$$L = x \mid c \mid \lambda x.e \mid f \ x \mid e_1 \mid e_2 \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3$$

L contains the usual arithmetic types, such as the natural numbers, and numerical operations such as sums and products. Dynamics for function application and fork-join are given as examples.

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{e_1 \mid e_2 \Downarrow (v_1, v_2)} \text{ (fork-join)}$$

$$\frac{f \Downarrow \lambda x.e \quad y \Downarrow v}{f \ y \Downarrow [v/x]e} \text{ (function-app)}$$

The language also includes 3 functions for working with arrays: tabulate, get, and set. The dynamics for tabulate is given below:

$$\frac{f(1) \Downarrow v_1, \dots, f(n) \Downarrow v_n}{\text{tabulate } f \ n \Downarrow [1 \mapsto v_1, \dots, n \mapsto v_n]}$$

The dynamics for the other two array functions are standard.

4. Approach

An array A is NEW if the set function has not been called with A as an argument, otherwise A is OLD. While the dynamics of our arrays are functional, the costs are different for OLD and NEW arrays.

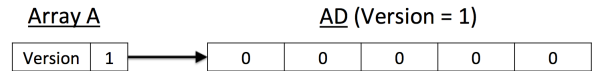


Figure 1. New functional array filled with 0s

Suppose that A is a NEW functional array (see figure 1). A has a version number V , and a pointer to an ArrayData object AD . AD keeps a regular (mutable) array of values, which corresponds to the values in A . AD has a version number, which is the same as A ’s (V) to indicate that A is NEW. For each element of the array in

AD , keep a log of values that used to be at that index. The logs are initially empty.

Get on NEW arrays: to get the i^{th} element in A , simply access $\text{array}[i]$ in AD .

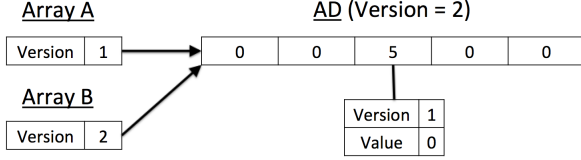


Figure 2. $B = \text{set}(A, 3, 5)$ changes the array in AD and adds a log entry.

Set on NEW arrays: Suppose that $A[i]$ is v_{old} and we want to do $B = \text{set}(A, i, v_{\text{new}})$ (see figure 2). Add an entry (V, v_{old}) to the log at index i , expressing that $A[i]$ was v_{old} at version V . Increment the version of AD to $V + 1$. Set $\text{array}[i]$ in AD to v_{new} . Create a new functional array B , with version $V + 1$, which points to AD . Notice that both A and B point to AD . However, the version of A is V and the version of AD is $V + 1$, which indicates that A is OLD. The version of B is $V + 1$ which indicates that B is NEW.

Get on OLD array: Suppose we do $\text{get}(A, i)$ where A is OLD and has version V . We binary search the log at index i to find what value was stored in the array at version V .

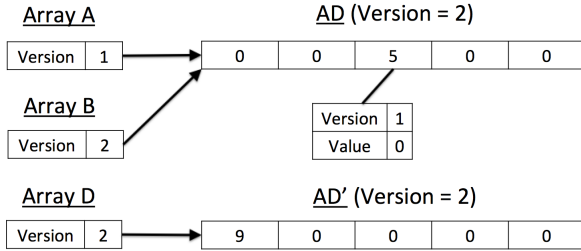


Figure 3. $D = \text{set}(A, 1, 9)$ copies AD because A is OLD

Suppose we do $D = \text{set}(A, i, v)$ where A is old (see figure 3). Create a new ArrayData object AD' (with a new array). Copy the values from the array in AD to the array in AD' , and then set $\text{array}[i]$ to v in AD' .

The logs are stored in a doubling array which doubles in size when full. This guarantees amortized $O(1)$ insertion of a log entry, and allows us to binary search with $O(\log m)$ work, where m is the number of log entries.

Suppose that the size of the array is n . Every n times an ArrayData object is modified, we create a new ArrayData object, and copy the values over to the new ArrayData object. This ensures that $m \leq n$ i.e. we don't have too many log entries in an ArrayData object. Copying is amortized $O(1)$.

It follows that in the RAM model the work of set and get is $O(1)$ for a NEW array. Get in an OLD array involves a binary search and has work $O(\log n)$. Set in an OLD array involves copying the array, and has work $O(n)$.

5. Cost Dynamics

Arrays are represented by (l, D) where l is a label that uniquely identifies the array, and D contains the values in the array. $D[i]$ is

the i^{th} value in the array. If the set method has been applied on an array, the array is OLD (represented by $-$), otherwise the array is NEW (represented by $+$).

Array methods have different costs depending on whether the array is NEW or OLD. Let $C(l)$ be the number of get operations applied on array $A = (l, D)$ that had $O(1)$ work. We thread a store δ through each operation. δ maps array labels l to $(+/-, c)$, where $c = C(l)$.

Our cost dynamics are defined by the following judgement, where δ is the store, f is the array function being applied, δ' is the new store, v is the value returned by applying f on $args$, w is the work, and s is the span.

$$\delta; f \text{ args} \Downarrow \delta'; v; w; s$$

The cost dynamics for the methods on a functional array of size n are:

$$\frac{l = \text{new label} \quad f(1) \Downarrow v_1; w_1; s_1; \dots f(n) \Downarrow v_n; w_n; s_n}{\delta; \text{tabulate } f \ n \Downarrow \delta[l \mapsto (+, 0)]; (l, \bigcup_{i=1}^n [i \mapsto v_i]); 1 + \sum w_i; 1 + \max s_i}$$

$$\frac{\delta[l \mapsto (+, c)] \quad A = (l, D) \quad l' = \text{new label}}{\delta; \text{set } A \ i \ v \Downarrow \delta[l \mapsto (-, c), l' \mapsto (+, 0)]; (l', D[i \mapsto v]); 1; 1} \text{ (set-new)}$$

$$\frac{\delta[l \mapsto (-, c)] \quad A = (l, D) \quad l' = \text{new label}}{\delta; \text{set } A \ i \ v \Downarrow \delta[l \mapsto (-, c), l' \mapsto (+, 0)]; (l', D[i \mapsto v]); n; 1} \text{ (set-old)}$$

$$\frac{\delta[l \mapsto (+, c)] \quad A = (l, D)}{\delta; \text{get } A \ i \Downarrow \delta[l \mapsto (+, c + 1)]; D[i]; 1; 1} \text{ (get-new)}$$

$$\frac{\delta[l \mapsto (-, c)] \quad A = (l, D)}{\delta; \text{get } A \ i \Downarrow \delta; D[i]; \log n; 1} \text{ (get-old)}$$

The fork-join cost semantics are the most interesting. We give the cost dynamic using 2 helper functions. Let $L(\delta)$ denote the set of labels in the store δ . Let $f(n)$ be the cost of calling get on an OLD array and $g(n)$ be the cost of calling set on an OLD array.

def NEW-MAP: $L(\delta_1) \cup L(\delta_2) \rightarrow (+/-, \text{int}) =$

Case $\delta[l \mapsto (-, c)] : (-, c)$

Case $\delta[l \mapsto (+, c)] :$

Case $\delta_1[l \mapsto (+, c + x)], \delta_2[l \mapsto (+, c + y)] : (+, c + x + y)$

Case $\delta_1[l \mapsto (+, c + x)], \delta_2[l \mapsto (-, c + y)] : (-, c + y)$

Case $\delta_1[l \mapsto (-, c + x)], \delta_2[l \mapsto (+, c + y)] : (-, c + x)$

Case $\delta_1[l \mapsto (-, c + x)], \delta_2[l \mapsto (-, c + y)] : (-, c)$

Else:

Case $\delta_1[l \mapsto (s, c)] : (s, c)$

Case $\delta_2[l \mapsto (s, c)] : (s, c)$

def EXTRA-WORK: $L(\delta_1) \cup L(\delta_2) \rightarrow \text{int} =$

Case $\delta[l \mapsto (+, c)] :$

Case $\delta_1[l \mapsto (+, c + x)], \delta_2[l \mapsto (+, c + y)] : 0$

Case $\delta_1[l \mapsto (+, c + x)], \delta_2[l \mapsto (-, c + y)] : xf(n)$

Case $\delta_1[l \mapsto (-, c + x)], \delta_2[l \mapsto (+, c + y)] : yf(n)$

Case $\delta_1[l \mapsto (-, c + x)], \delta_2[l \mapsto (-, c + y)] : (x + y)f(n) + g(n)$

Else: 0

$$\delta' = \bigcup_{l \in L(\delta_1) \cup L(\delta_2)} \text{NEW-MAP}(l)$$

$$w' = \sum_{l \in L(\delta_1) \cup L(\delta_2)} \text{EXTRA-WORK}(l)$$

Then, the fork-join cost semantics are:

$$\delta; e_1 || e_2 \Downarrow \delta'; (v_1, v_2); 1 + w_1 + w_2 + w'; 1 + \max(s_1, s_2)$$

6. Machine Model

Our target language is a Java-like language augmented with tabulate, fork-join, lambda functions, and link load store conditional (LLSC). Fork-join spawns two threads, one thread executing each sub-expression. Link load store conditional (LLSC) takes an (address, old value, new value) tuple as argument. LLSC first checks that the value at address is the same as old value, if it is different then LLSC returns false. Otherwise LLSC atomically stores new value at address and returns true, however if the value at address was modified between the load and store, LLSC returns false.

Our target language is run on a P processor machine. In a single time step the machine takes at most P runnable instructions and executes them. The effect of executing the ($\leq P$) instructions in a time step is the same as some (arbitrary) sequential ordering of the instructions. We assume that the machine does not reorder instructions in the target language. In reality, languages like Java have very relaxed consistency models, and we deal with this in our real implementation using memory fences.

The link load store conditional architecture deserves special mention. Call an LLSC operation pending if the load operation has completed but the store operation has not. Multiple threads can load the value at a particular address and compare it with old value in a single time step. Only a single store operation to a particular address can be executed at a time step. However, after the store operation, in each time step the bus arbiter notifies all pending LLSC operations to the same memory address that they have failed. This assumption is reasonable because all LLSC instructions are typically processed by the bus arbiter, so the bus arbiter can keep track of pending LLSC instructions to each memory address.

7. Proof of Correctness

We begin by proving that functional arrays work correctly when executed sequentially. Specifically, in any fork join, we assume that the left fork is evaluated before the right fork.

Lemma 7.1. *The implementation in section 4 correctly implements the dynamics of functional arrays when executed sequentially.*

Proof. We define a data structure invariant, prove the invariant by structural induction on the array functions, and finally use the invariant to show that get always returns required value.

Data structure invariant: consider arbitrary array A in the environment. Let $A[i]$ denote the actual value of A at index i . A points to a structure AD and has version v . If v is the same as the version of AD , then $AD.array[i]$ and $A[i]$ are the same. Otherwise, consider arbitrary index i and let L be the log array of AD at index i . If the version of all log entries in L is less than v , then $A[i]$ is the same as $AD.array[i]$. Otherwise, let (v, e) be the log entry with the lowest version v s.t. $v \geq V$. Then $A[i] = e$.

We prove that this invariant holds by structural induction. (Add proof outline)

This then trivially gives us that get returns the correct value. \square

Lemma 7.2. *Functional arrays work correctly when evaluated sequentially.*

Proof. \square