Fast Arrays: Atomic Arrays with Constant Time Initialization

Siddhartha Jayanti
సిద్ధార్థ జయంతి
MIT CSAIL, Cambridge, MA, USA

Julian Shun
MIT CSAIL, Cambridge, MA, USA

Abstract

Some algorithms require a large array, but only operate on a small fraction of its indices. Examples include adjacency matrices for sparse graphs, hash tables, and van Emde Boas trees. For such algorithms, array initialization can be the most time-consuming operation. *Fast arrays* were invented to avoid this costly initialization. A *fast array* is a software implementation of an array, such that the entire array can be initialized in just constant time.

While algorithms for *sequential* fast arrays have been known for a long time, to the best of our knowledge, there are no previous algorithms for *concurrent* fast arrays. We present the first such algorithms in this paper. Our first algorithm is linearizable and wait-free, uses only linear space, and supports all operations – initialize, read, and write – in constant time. Our second algorithm enhances the first to additionally support all the read-modify-write operations available in hardware (such as compare-and-swap) in constant time.

2012 ACM Subject Classification Theory of computation → Concurrent algorithms; Theory of computation → Data structures design and analysis

Keywords and phrases fast array, linearizable, wait-free, asynchronous, multiprocessor, constant time, space efficient, data structure

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.25

Funding *Siddhartha Jayanti*: NDSEG Fellowship from the US Department of Defense
Julian Shun: DOE Early Career Award #DESC0018947, NSF CAREER Award #CCF-1845763, a Google Faculty Research Award, a Google Research Scholar Award, DARPA SDH Award #HR0011-18-3-0007, and Applications Driving Architectures (ADA) Research Center, and a JUMP Center co-sponsored by SRC and DARPA

1 Introduction

Arrays are the most fundamental data structure in computer science. Semantically, an *array* of length m is an object that supports the following interface:

- INITIALIZE(m, f): return an array $\mathcal{O}$ initialized to $\mathcal{O}[i] = f(i)$ for each $i \in [m]$.[1]
- $\mathcal{O}$.READ(i): return $\mathcal{O}[i]$, if $i \in [m]$.
- $\mathcal{O}$.WRITE(i, v): update $\mathcal{O}[i]$'s value to v, if $i \in [m]$.

Here, INITIALIZE() is the *constructor* method that creates the object, and READ() and WRITE() are the regular operations an array supports. Ordinarily, initialization is achieved by allocating an array of length m and looping through to initialize its entries, while reads and writes simply use the hardware load and store instructions. This standard implementation achieves a space complexity of $O(m)$, and time complexities of $O(m)$ for initialization and $O(1)$ for reads and writes. These time complexities are good for applications that eventually access most of the entries of the array. But, some applications – such as adjacency matrix

[1] For a positive integer m, we use the notation $[m] \triangleq \{0, 1, \ldots, m-1\}$.

representations of sparse graphs, van Emde Boas trees, and certain hash tables – need to allocate a large array when only a small fraction of the array will eventually be accessed. The time complexities of such algorithms would improve drastically if we had *fast arrays*: arrays that support all three operations – READ(), WRITE(), and *even* INITIALIZE() – in just $O(1)$ worst-case time. Perhaps surprisingly, sequential fast array implementations have been known for decades, but, to the best of our knowledge, concurrent implementations do not exist. We design the first algorithms for concurrent fast arrays in this paper.

1.1 Sequential fast arrays: history and applications

Sequential algorithms for fast arrays date back to at least the 1970s [1, 4, 25]. In fact, the well known *folklore algorithm* for the problem (which we will revisit in Section 3) was alluded to in an exercise of the celebrated text by Aho, Hopcroft, and Ullman [1] and further described by Mehlhorn [25] and Bentley [4]; it achieves a deterministic worst-case time complexity of $O(1)$ for each of the three operations, while using only $3m + 1$ memory words. Fast arrays have been important to the efficiency of several algorithms. Notably:

- Fast arrays are used in implementations of *van Emde Boas trees* [6, 30] – associative arrays that store keys from a universe $\{1, 2, \ldots, u\}$ and support *insert*, *get*, and *delete* with a time complexity of just $O(\log \log u)$.
- Katoh et al. [21] note that Knuth employs fast arrays in the implementation of the hash table in his *Simpath* algorithm [22], which enumerates all simple paths between two vertices in a graph. Knuth uses the hash table to efficiently implement a certain data structure called ZDD (Zero-suppressed binary Decision Diagram) [27], which has many applications besides Simpath [18, 22, 28, 29, 34, 35].
- When a sparse graph of n vertices and $m \ll n^2$ edges is represented using an adjacency matrix, mere initialization can take $\Theta(n^2)$ time with a traditional array. With a fast array however, the graph can be stored in just $O(m)$ time. Consequently, for a constant degree graph, storing the graph takes $O(n)$ time instead of $\Theta(n^2)$ time.

More generally, fast arrays can increase the asymptotic efficiency of algorithms that have higher space complexity than time complexity – just allocate all the space in one huge block in $O(1)$ time.

This range of applications has spurred a lot of research into fast arrays in recent years. A string of papers, starting with Navarro's work in 2012 and culminating in three back to back papers in 2017 by Hagerup and Kammer, Loong et al., and Katoh and Goto, have brought down the space complexity from $3m + 1$ to $m + 1$ using complex bit-packing and chaining techniques [13, 21, 23, 31, 32]. Fredriksson and Kilpeläinen recently studied the empirical running times of the more practical implementations of these sequential fast arrays [9].

1.2 Concurrent fast arrays

In contrast to sequential fast arrays, which have been well studied, there has been no prior work on concurrent fast arrays, to the best of our knowledge. In this paper, we propose and design algorithms for two variants of concurrent fast arrays:

- **Fast Array:** This is an implementation of an array which supports the standard operations – INITIALIZE(m, f), READ(i), and WRITE(i, v) – and satisfies two conditions. First, each operation is *linearizable*, i.e., it appears to take effect at some instant between its invocation and response [15]. Second, each operation is not only *wait-free* [14], but the process that executes the operation completes it in a constant number of its steps. The first condition ensures *atomicity*, and the second condition ensures *efficiency*.

- **Fast Generalized Array:** Besides *load* and *store*, modern architectures like x86 commonly support read-modify-write (RMW) primitives, such as Compare-and-Swap (CAS), Fetch-and-Add (FAA), and Fetch-and-Store (FAS) [17]. In fact, some of these primitives are indispensible for efficiency and even solvability of problems that arise in concurrent systems. For instance, implementing a wait-free queue is impossible using only loads and stores [14]. Mutex locks can be implemented using loads and stores, but constant RMR (remote memory reference) complexity implementations are impossible using only loads and stores [3, 7, 26].

  Since RMW primitives are supported by hardware and are essential for concurrent algorithms, it would be ideal if the components of the fast array can be manipulated using these primitives. For instance, when implementing a fast array $\mathcal{O}$ on a multiprocessor that supports CAS and FAS in hardware, a process π should not only be able to read $\mathcal{O}[i]$ and write to $\mathcal{O}[i]$, but should also be able to CAS $\mathcal{O}[i]$ and FAS $\mathcal{O}[i]$. We term such an array, which allows all hardware-supported operations to be applied to its components, a *generalized array*.

  Let $\mathcal{S}$ be the set of hardware-supported RMW primitives. A *fast generalized array* is an implementation that not only supports $O(1)$-time linearizable $\textsc{Initialize}(m, f)$, $\textsc{Read}(i)$, and $\textsc{Write}(i, v)$ operations, but also supports $O(1)$-time linearizable operations from the set $\mathcal{S}$.

1.3 Our contributions

In addition to defining the two types of concurrent fast arrays, our paper makes the following two principal contributions:

- We design an algorithm for the (standard) fast array. If p processes share a fast array of length m, our algorithm uses only $O(m + p)$ space. More generally, to instantiate and use k fast arrays (for any k) of lengths $m_1, \ldots, m_k$, our algorithm uses only $O(M + p)$ space, where $M = \sum_{j=1}^{k} m_j$.
- We enhance the above algorithm to design a fast generalized array. Its space complexity is the same as the previous algorithm's – $O(m + p)$ for a single array of length m, and $O(M + p)$ for multiple arrays of combined length M.

Both of the above algorithms require hardware support for read, write, and CAS.

2 Model

We work in the standard asynchronous shared memory multiprocessor model where p processes, numbered $0, \ldots, p - 1$ run concurrently but asynchronously, and each process is either performing an initializable array operation or is idle. The computation proceeds in steps, where an *adversarial scheduler* decides which process π takes the next step.

To provide synchronization, we assume the hardware compare-and-swap (CAS) synchronization primitive. The CAS operation on a memory word X with arguments *old* and *new* is called as follows: $\textsc{Cas}(X, old, new)$. The operation is atomic and has the following behavior. If $X = old$, then X's value is updated to *new* and *true* is returned to indicate that the operation successfully changed the value; otherwise, if $X \neq old$, the value of X is not changed and *false* is returned. On modern x86 architectures, individual memory words are 64-bits, and so any hardware primitive can be applied on a standard 64-bit word. Usefully however, CAS can also be executed on 128-bit double-words, i.e., two adjacent words of memory [17]. We will make use of this feature. (Note that this 128-bit CAS operation is *not* DCAS – a primitive that does CAS on two non-adjacent memory locations, which is not supported in modern architectures.)

A data structure is *linearizable* if each operation can be assigned a unique *linearization point* between its invocation and return, and the return values of the operations are consistent with those of a sequential execution in which operations are executed in the order of linearization points [15]. Operations are *bounded wait-free* if there exists a bound b such that every invocation by a process π returns within b of π's own steps [14]. In the literature, data structures that are both linearizable and wait-free are called *atomic*.

We measure the efficiency of an algorithm by its worst-case work and space complexities. The *space complexity* of an algorithm is the total number of memory words that the algorithm uses. The *work complexity* of an operation by a process π, is the total number of steps executed by π between the invocation and return of that operation. Since work complexity is the natural generalization of time complexity to multiprocessors, it is often called *time complexity* in the literature; we adopt this convention and use the two terms interchangeably. Furthermore, as is standard [1, 4, 25, 30–33], we assume that it takes constant time to allocate an *uninitialized* array of any size n. We call an object implementation *fast* if every operation on that object takes only $O(1)$ time to execute in the worst-case. Our paper focuses on fast algorithms for arrays and generalized arrays.

3 Folklore Sequential Algorithm

Our concurrent algorithms are inspired by the folklore sequential algorithm, and so we present the pseudo-code for a folkore fast array object $\mathcal{O}$ in Algorithm 1, and describe it below.

Algorithm 1 The folklore algorithm for a sequential fast array.

```
    procedure INITIALIZE(m, f)
1:      A ← new array[m]
2:      B ← new array[m]
3:      C ← new array[m]
4:      f_init ← f
5:      X ← 0

    procedure READ(i)
6:      if 0 ≤ B[i] < X and C[B[i]] = i then return A[i] else return f_init(i)

    procedure WRITE(i, v)
7:      A[i] ← v
8:      if B[i] < 0 or B[i] ≥ X or C[B[i]] ≠ i then
9:          C[X] ← i
10:         B[i] ← X
11:         X ← X + 1
```

The method INITIALIZE(m, f) instantiates a new fast array $\mathcal{O}$ of length m. The implementation of the fast array uses three un-initialized arrays A, B, and C, each of length m, an integer X, and stores a pointer f_{init} to the function f. We call A the *principal array* and use $A[i]$ to hold the current value of the abstract element $\mathcal{O}[i]$ for each index i that has been *initialized*, i.e., written to at least once. The elements of $A[i]$ corresponding to uninitialized indices of $\mathcal{O}[i]$ hold their initial, arbitrary values. B, C, and X are used to keep track of which indices i have already been initialized (as we describe later).

Using the mechanism described above, implementing read and write becomes simple. READ(i) just returns $A[i]$ if i has been initialized, and $f(i)$ otherwise. Correspondingly, WRITE(i, v) simply writes $A[i] \leftarrow v$, and ensures that index i is marked as initialized.

The main difficulty of the algorithm lies in efficiently remembering which set of indices i have been initialized. We use the array C and the integer X to maintain this set as follows. If k indices have already been initialized, then we ensure that $X = k$ and that the sub-array $C[0, \ldots, X-1]$ holds the values of these initialized indices. Correspondingly, we spend constant time in the INITIALIZE() method to set $X \leftarrow 0$. Terminologically, we call C the *certification array*, call the elements of the array *certificates*, call the elements of the sub-array $C[0, \ldots, X-1]$ *valid*, and say that an index i is *certified* when it appears in the valid sub-array.

Maintaining A, C, and X is sufficient to get a *correct* implementation of an array, but not an *efficient* one. For efficiency, we need to distinguish between certified and un-certified indices in constant time. We use the array B for this purpose. In particular, whenever we certify a new index i in an element $C[j]$, we set $B[i] \leftarrow j$ to maintain the invariant that

$$\mathcal{I} \equiv \ \forall i \in [m],\ (0 \leq B[i] < X \text{ and } C[B[i]] = i \ \ \textit{if and only if} \text{ index } i \text{ is initialized}).$$

The check that $0 \leq B[i] < X$ ensures that $C[B[i]]$ is valid, while the check that $C[B[i]] = i$ ensures that this valid element of the certification array, indeed, certifies that index i is initialized.

4 Our Concurrent Fast-Array

The goal of this section is to design a linearizable wait-free fast-array that is both time and space efficient. We do so by building on the ideas of the folklore algorithm.

The folklore algorithm is built on two pillars: (1) the principal array A, which stores the values of initialized indices, and (2) the *certification mechanism* constituted by B, C, and X, which ensures that initialized indices can be identified in constant time by invariant $\mathcal{I}$. The principal array can easily be maintained in the concurrent setting, however the certification mechanism, which is the main workhorse of Algorithm 1, must be redesigned to cope with concurrency.

The difficulty of using the sequential certification mechanism with multiple processors stems from the contention on the variable X, and on the next available slot in the certification array, $C[X]$. In particular, if all p processors are concurrently performing different write operations on different un-initialized indices $i_0, \ldots, i_{p-1}$, then the old certification mechanism will direct all of them to the same location $C[X]$ in the certification array. Regardless of how the contention is resolved, only one index can fit into $C[X]$, meaning that $p-1$ processes will fail to certify their index by placing it in $C[X]$ and will thereby need to find an alternate location in the certification array. So, in the worst-case, only one process will finish its operation after all p processes do one unit of work each, meaning the algorithm will do $O(p)$ work per operation rather than $O(1)$.

We overcome this difficulty of adapting the certifying mechanism posed by contention on C and X by introducing four ideas that we detail below. Our first idea will eliminate the contention, thereby enabling constant time certifications and look-ups; however, it bloats the space complexity to $\Omega(m \cdot p)$. Our second and third ideas, in combination, eliminate this space overhead and bring the space complexity down to just $O(m + p)$, while ensuring that the time complexities of operations remain at just $O(1)$. Our fourth idea describes how to share resources in order to minimize the space complexity when multiple fast arrays are instantiated.

Individual certification arrays. Our first idea is to eliminate the universal C and X, and instead equip each process π with its own certification array c_π and a corresponding *control variable* X$[\pi]$. Here, X is a one-dimensional array of length p that is indexed by process

ids, and each c_π is an array of length m. Thus, process π certifies a new index i by performing three steps: (1) writing $c_\pi[\mathrm{X}[\pi]] \leftarrow i$, (2) setting $\mathrm{X}[\pi] \leftarrow \mathrm{X}[\pi] + 1$, and (3) writing $B[i] \leftarrow (\pi, \mathrm{X}[\pi] - 1)$. (While it will not yet be clear to the reader at this stage, the relative order of steps 2 and 3 is very important for the correctness of later ideas. We expand on this thought in the forthcoming Remark 1.) Unlike the act of certifying a new index which involves modifying certification arrays and control variables, the act of checking whether a given index i is certified only requires reading. We allow process π to freely read the arrays of other processes while checking if an index is certified. This idea of individual certification arrays by itself would lead to a concurrent fast-array algorithm; however, the space complexity of this algorithm is inherently super-linear. In particular, if all p processes concurrently write to a previously un-initialized location i, an adversarial scheduler can force each of them to certify that location in its own certification array; if this happens for each of the m indices, then each c_π must store m indices, leading to a total space complexity of $\Theta(mp)$.

Synchronization and walk-back. To reduce the space complexity induced by individual certification arrays, we must ensure that each index i is certified by at most one process, even if multiple processes perform concurrent writes to the same un-initialized index. To do this, we introduce two related ideas: *synchronization on B* and *walk-back*. That is, each processor π that wishes to certify an index i attempts to CAS (rather than write) the pair $(\pi, \mathrm{X}[\pi] - 1)$ – indicating the location in its certification array where i is certified – into $B[i]$. We orchestrate the update to $B[i]$ using CAS to ensure that at most one process gets a return value of *true*, indicating that *it* is the process that succeeded in certifying i. Each other process π, whose CAS to $B[i]$ fails, "walks back", i.e., it reclaims the location $c_\pi[\mathrm{X}[\pi] - 1]$ that it was going to use to certify i by decrementing $\mathrm{X}[\pi]$. Since each index is certified by at most one process, and each process has at most one certification location that it will walk back on at any given time, the total space used across all c_π arrays is $O(m + p)$.

Array doubling. Our synchronization and walk-back scheme guarantees that at most $O(m + p)$ space is *used* across all of the c_π arrays, but we do not *a priori* know how many locations each process π will use in its c_π array. To ensure that we allocate only as much space as we use, we employ the classic idea of *array-doubling* from sequential algorithms. We initially allocate constant sized c_π arrays. Each time c_π fills up, we replace it with a newly allocated array c'_π of length $2 \cdot c_\pi.len$ and copy over the old $c_\pi.len$ elements from c_π to c'_π. Note that we *do not* de-allocate the old array c_π when we switch to c'_π, since other processes could be accessing it; yet, since the sum of a geometric series is proportional to the largest term in the series, our total memory allocation for the c_π arrays is proportional to the amount of space we end up using. (Note that it is important to have a mechanism by which other processes can get access to the current array c_π, since the location of the array is changing whenever we double. We describe this detail when we discuss the pseudo-code in a later sub-section. We will also describe how to implement array doubling with worst-case, rather than amortized, constant time per operation in the same sub-section.)

Sharing the certification mechanism. Array doubling allows us to share a single certification mechanism across all fast-array objects that we initialize. In particular, if we have multiple fast-arrays $\mathcal{O}_1, \ldots, \mathcal{O}_k$, each $\mathcal{O}_j$ can simply maintain the two instance variables $\mathcal{O}_j.A$ and $\mathcal{O}_j.B$, and *share* the certification mechanism – $\forall \pi \in [p], (\mathrm{X}[\pi], c_\pi)$. All we need to do to enable this sharing is store a pointer $\&(\mathcal{O}_j.A[i])$ as the certificate that i is initialized in

fast-array $\mathcal{O}_j$, rather than just store the index i. Since all fast array objects can share one certification mechanism, the space complexity of maintaining k fast-arrays $\mathcal{O}_1, \ldots, \mathcal{O}_k$ of sizes $m_1, \ldots, m_k$ is just $O(M + p)$, where $M \triangleq \sum_{j=1}^{k} m_j$.

▶ Remark 1 (the relative order of synchronizing and incrementing). When a process π, with next available certification location $x_\pi = \mathrm{X}[\pi]$, is certifying a new index i, we described that our algorithm follows three logical steps (not including potential walk-back): (1) *writing the certificate*: $c_\pi[x_\pi] \leftarrow i$, (2) *incrementing* $\mathrm{X}[\pi]$: $\mathrm{X}[\pi] \leftarrow x_\pi + 1$, and (3) *synchronizing* on $B[i]$: attempting to CAS the value (π, x_π) into $B[i]$. At first glance, it may appear that step (3) can be executed before step (2). Indeed, if this were possible, then we could simplify our algorithm by avoiding walk-backs altogether, since a process π that fails the CAS could simply not increment $\mathrm{X}[\pi]$. However, as we mentioned earlier, the relative order of steps (2) and (3) is pivotal to correctness. We now explain why.

Consider a scenario where two processes π and τ are each performing the operation $\textsc{Write}(i, new)$ on a previously un-initialized location $\mathcal{O}[i]$ whose initial value is $\mathcal{O}[i] = f(i) = old$, with the order of steps (2) and (3) swapped. Then the following sequence of events can occur:

1. Both process π and τ write $A[i] \leftarrow new$, read the same old value $b \leftarrow B[i]$ (in anticipation of having to $\textsc{Cas}$ $B[i]$ in step (3)), and start the certification process.
2. π completes steps (1) and (3) and thereby successfully changes $B[i]$'s value to (π, x_π). However, i is still not certified since step (2) is yet to be done, and $\mathrm{X}[\pi] \not> x_\pi$.
3. τ completes steps (1) and (3), but because τ's $\textsc{Cas}$ in step (3) fails, it does not need to execute step (2), and it returns from its write operation.
4. Having finished its write operation, τ performs $\textsc{Read}(i)$, but sees that i is not yet certified (since π has not yet finished step (2)), and returns $f(i) = old$.

- This execution is not linearizable, since τ reads the value old in $\mathcal{O}[i]$ even after it finishes writing new.

Remark 1 establishes that a process π must increment $\mathrm{X}[\pi]$ before it synchronizes at $B[i]$ in the certification process. This means that we must indeed implement walk-back to achieve space efficiency. However, if walk-back is not implemented very carefully, it can lead to a nasty race condition. We describe this possible race condition, and how to overcome it, in the next subsection.

4.1 A tricky race condition that must be avoided

Until now, we have described the main ideas that propel our space-efficient fast-array implementation at a high-level. Of these ideas, individual certification arrays are straightforward to implement as suggested, and array-doubling requires only mild adaptation to work in the face of asynchronous concurrency. The idea of walk-back however can lead to a nasty race-condition if it is not implemented correctly. We describe this potential race, and how we overcome it below.

In order to understand the race condition, let us consider the following set-up. We have a freshly initialized fast-array $\mathcal{O}$ with just two locations $\mathcal{O}[0, 1]$, and initialization function $f(i) = old$. There are three processors π, τ, and ρ with: $\mathrm{X}[\pi] = 0$, $\mathrm{X}[\tau] = 0$, and $\mathrm{X}[\rho] = 0$. The processes will perform the following operations:

- π will perform $\mathcal{O}.\textsc{Write}(0, new)$ followed by $\mathcal{O}.\textsc{Write}(1, new)$
- τ will perform $\mathcal{O}.\textsc{Write}(0, new)$
- ρ will perform $\mathcal{O}.\textsc{Read}(0)$ followed by another $\mathcal{O}.\textsc{Read}(0)$

By design, both locations initially hold the value *old* and at some point in time will take on the value *new* and hold that value forever. However, the race condition will be that ρ's first read of index 0 will return *new*, while its second read will return *old*. The initial value of $B[0]$ – which can be arbitrary by design of the algorithm – is pivotal to achieving the race. In particular, we consider the initial value $B[0] = (\pi, 0)$. The initial values of $A[0, 1]$ and $B[1]$ can be arbitrary.

We describe the offending run below in a sequence of bullet points. When the relative order of certain operations do not matter, we may describe them all in the same bullet point.

- Recall that $B[0] = (\pi, 0)$, π is performing $\textsc{Write}(0, new)$, and τ is performing $\textsc{Write}(0, new)$.

1. π and τ both write $A[0] \leftarrow new$, and both read the initial value b of $B[0]$. (They will need this value b when they attempt to certify index 0 and do a Cas on $B[0]$ later.)

2. π and τ both conclude that index 0 is not certified yet, and thus wish to certify the location.

- Recall that $\mathrm{X}[\pi] = 0$.

3. π's next open certification location is 0, thus π writes $c_\pi[0] \leftarrow 0$, and increments $\mathrm{X}[\pi] \leftarrow 1$. π now stalls (before attempting to Cas its certification location $(\pi, 0)$ into $B[0]$).

- Notice that while π is not finished with its certification process, index 0 is already certified, since $B[0] = (\pi, 0)$ initially, and location $c_\pi[0]$ is valid and holds the value 0.

4. ρ does its first $\textsc{Read}(0)$ operation. That is, it reads $B[0] = (\pi, 0)$, checks that $c_\pi[0]$ is valid and that $c_\pi[0] = 0$ and thereby returns the value $A[0] = new$.

5. ρ starts its second $\textsc{Read}(0)$ operation. It starts its verification by reading $B[0] = (\pi, 0)$, and then stalls.

6. τ's next open certification location is 0, thus τ writes $c_\tau[0] \leftarrow 0$, increments $\mathrm{X}[\tau] \leftarrow 1$, and successfully CAses its certification location $(\tau, 0)$ into $B[0]$.

- Notice that index 0 is now certified by two certificates $c_\pi[0]$ and $c_\tau[0]$. $B[0] = (\tau, 0)$ identifies only the new certificate, but process ρ is about to check for the old certificate $c_\pi[0]$.

7. π attempts to finish its certification process by CAsing $(\pi, 0)$ into $B[0]$. However, its Cas fails. Thus, π walks-back, and resets $\mathrm{X}[\pi] \leftarrow 0$. This completes π's write operation to index 0.

8. π does its entire $\textsc{Write}(1, new)$ operation. That is, it writes $A[1] \leftarrow new$, writes $c_\pi[0] \leftarrow 1$, increments $\mathrm{X}[\pi]$ to 1, successfully CAses $(\pi, 0)$ into $B[1]$, and returns.

9. ρ now finishes its operation. Since it had previously read $B[0] = (\pi, 0)$ and $\mathrm{X}[\pi] = 1 > 0$, it checks $c_\pi[0]$, finds the value 1 there, concludes that index 0 is not certified, and thereby returns $f(0) = old$.

- This run cannot be linearized since $\mathcal{O}[0]$ was initially *old* and became *new*, but ρ reads its value to be *new* and subsequently re-reads the value to be *old*.

We observe that the cause of this race condition is the coincidental initial value of $B[0]$. In particular, $B[0]$'s (potentially arbitrary) initial value, happened to coincide with the exact location that process π would use to certify index 0 and later have to walk-back on. This coincidence, in turn, caused $B[0]$ to become certified during step (3), before π finished its certification operation by updating $B[0]$ with a Cas.

Tombstoning. Since we have only constant time to initialize $\mathcal{O}$, we *cannot* control the initial values of all the elements $B[i]$. However, we *can* control which location in the certification array π uses to certify $B[i]$. Therefore, we eliminate this nasty race condition as follows. If the initial value of $B[i]$ is (π, k), then we ensure that that particular process π does not

attempt to use its certificate $c_\pi[k]$ to certify index i. If it so happens that $c_\pi[k]$ is the next available certificate for process π when process π is attempting to certify i, then we simply *tombstone* that location by writing a special null-value $c_\pi[k] \leftarrow \perp$, and use the next location $c_\pi[k+1]$ to certify i. This ensures correctness.

Furthermore, observe that for each location i, there is exactly one initial value $B[i] = (\pi, k)$ that references exactly one process π and one specific location k. Thus, at most m locations get tombstoned by our method across all processes, and the space complexity bound of $O(M + p)$ continues to hold true even in the worst-case. (We expect that tombstoning will occur only very rarely in practice.)

4.2 The pseudo-code and its description

Having described individual certification arrays, synchronization and walk-back, array-doubling, and tombstoning, we are ready to describe our fast-array algorithm. We present the pseudo-code as Algorithm 2, and describe it below.

Naming conventions. In order to distinguish between variables of different processes and operations that are performed by a particular process π, we use subscripts. For example, we denote a local variable x of process π by x_π, and denote a $\textsc{Read}()$ operation by process π as $\textsc{Read}_\pi()$. We use captial letters, such as A and X, to refer to arrays that all processes have the address of by default. Importantly, note that the pointer to the *current* certification arrays, c_π, follows the above convention, and by default only process π has access to the array. In order to allow other processes to access these arrays, our implementation stores a pair in the control variable $\mathrm{X}[\pi]$. So, initially $\mathrm{X}[\pi] = (0, c_\pi)$ (rather than $\mathrm{X}[\pi] = 0$).

In order to implement array doubling, we maintain a *next* certification array c'_π alongside the current array c_π. As such, initially $\mathrm{X}[\pi] = (0, c_\pi)$ and no process other than π has access to the array pointed to by c'_π. When c_π fills up entirely, we maintain the invariant that $c'_\pi[0, \ldots, c_\pi.len - 1] = c_\pi[0, \ldots, c_\pi.len - 1]$ and thus, we can simply replace $\mathrm{X}[\pi] = (x_\pi, c_\pi)$ by $\mathrm{X}[\pi] = (x_\pi, c'_\pi)$; we also rename c'_π as c_π because it has become the current array, and allocate a new (un-initialized) c'_π that is twice the length of the new current array. In order to ensure that $c'_\pi[0, \ldots, c_\pi.len - 1] = c_\pi[0, \ldots, c_\pi.len - 1]$ by the time c_π gets entirely full, we *transfer* two values from c_π to c'_π each time a new value is appended to c_π. We note that we *do not* de-allocate the old certification arrays because other processes could potentially be reading from them – this does not change the asymptotic space complexity. However, each process can store pointers to all of its arrays so that they can be de-allocated when the fast array is no longer needed. In our algorithm, we choose to start with a c_π of length 2. (Any other constant length would have sufficed.)

A line-by-line description of the code is as follows. $\mathcal{O}.\textsc{Initialize}_\pi(m_\pi, f_\pi)$ simply allocates the unintialized arrays A and B of length m_π (Lines 1 and 2), and stores the initialization function f_π as f_{init} for future use (Line 3). The elements of $A[i]$ will be used to hold the values of the abstract elements $\mathcal{O}[i]$. The elements of $B[i]$ will be used to hold process-index pairs (π, j); as a matter of convention, we call the first element in the pair $B[i].pid$ (process id) and the second element $B[i].loc$ (location).

$\textsc{Write}_\pi(i_\pi, v_\pi)$ first updates $A[i_\pi]$ to the new value v_π (Line 4). Since index i_π may not yet be certified, it calls $\textsc{Certify}_\pi(i_\pi)$ (Line 5). As we describe below, $\textsc{Certify}_\pi(i_\pi)$ only creates a new certificate for i_π if the index is not already certified.

Since $\textsc{Certify}_\pi(i_\pi)$ creates a new certificate (if necessary), and must perform the synchronization-CAS on $B[i_\pi]$ after creating a certificate, it must read the old value of $B[i_\pi]$. Thus, $\textsc{Certify}_\pi(i_\pi)$ starts by reading $b_\pi^{old} \leftarrow B[i]$ (Line 16). Certification should

Algorithm 2 Atomic fast array for p processes. Pseudo-code shown for an arbitrary process π.

Variables:

For each process $\pi \in [p]$ the following variables are shared across *all* fast-arrays $\mathcal{O}$:

- $c_\pi[0,1]$ is a pointer to an allocated un-initialized array of length 2.
- $c'_\pi[0,\ldots,3]$ is a pointer to an allocated un-initialized array of length 4.
- k_π is a non-negative integer that is initialized to 0.
- $\mathrm{X}[\pi]$ is a pair that is initialized to $(0, c_\pi)$.

Each object $\mathcal{O}$ has three instance variables instantiated by $\textsc{Initialize}_\pi(m_\pi, f_\pi)$:

- A and B are arrays of length m_π.
- f_{init} stores the initial value function.

Each process $\pi \in [p]$ uses the following arbitrarily initialized temporary local variables:

- b_π, b_π^{old}: hold (process id, array index) pairs.
- x_π: holds an array index.
- c_π^{other}: holds an array pointer.

procedure $\mathcal{O}.\textsc{Initialize}_\pi(m_\pi, f_\pi)$
1: $A \leftarrow$ **new** array$[m_\pi]$
2: $B \leftarrow$ **new** array$[m_\pi]$
3: $f_{init} \leftarrow f_\pi$

procedure $\mathcal{O}.\textsc{Write}_\pi(i_\pi, v_\pi)$
4: $A[i_\pi] \leftarrow v_\pi$
5: $\textsc{Certify}_\pi(i_\pi)$

procedure $\mathcal{O}.\textsc{Read}_\pi(i_\pi)$
6: **if** $\textsc{IsCertified}_\pi(i_\pi)$ **then**
7: **return** $A[i_\pi]$
8: **else return** $f_{init}(i_\pi)$

9: **procedure** $\mathcal{O}.\textsc{IsCertified}_\pi(i_\pi)$
10: $b_\pi \leftarrow B[i_\pi]$
11: **if** $0 \le b_\pi.pid < p$ **then**
12: $(x_\pi, c_\pi^{other}) \leftarrow \mathrm{X}[b_\pi.pid]$
13: **if** $0 \le b_\pi.loc < x_\pi$ **and** $c_\pi^{other}[b_\pi.loc] = \&A[i_\pi]$ **then return** *true*
14: **return** *false*

15: **procedure** $\mathcal{O}.\textsc{Certify}_\pi(i_\pi)$
16: $b_\pi^{old} \leftarrow B[i_\pi]$
17: **if** $\textsc{IsCertified}_\pi(i_\pi)$ **then return**
18: $(x_\pi, -) \leftarrow \mathrm{X}[\pi]$
19: **if** $x_\pi \ge c_\pi.len$ **then**
20: $c_\pi \leftarrow c'_\pi$
21: $c'_\pi \leftarrow$ **new** array$[2 \cdot c_\pi.len]$
22: $k_\pi \leftarrow 0$
23: **if** $b_\pi^{old}.pid = \pi$ **and** $b_\pi^{old}.loc = x_\pi$ **then**
24: $c_\pi[x_\pi] = \perp$
25: $x_\pi \leftarrow x_\pi + 1$
26: $c_\pi[x_\pi] \leftarrow \&A[i_\pi]$
27: $\mathrm{X}[\pi] \leftarrow (x_\pi + 1, c_\pi)$
28: **while** $k_\pi < 2x_\pi - c_\pi.len$ **do**
29: $c'_\pi[k_\pi] \leftarrow c_\pi[k_\pi]$
30: $k_\pi \leftarrow k_\pi + 1$
31: **if not** $\textsc{Cas}(B[i_\pi], b_\pi^{old}, (\pi, x_\pi))$ **then**
32: $\mathrm{X}[\pi] \leftarrow (x_\pi, c_\pi)$

only be done if i_π is not already certified, and so it calls IsCertified$_\pi(i_\pi)$ and returns immediately if location i_π is already certified (Line 17). Otherwise, it fetches the next location x_π that is free for creating a certificate (Line 18). Certification proceeds in five steps:

1. If the current certification array is full (check on Line 19), then the next array becomes the current one (Line 20), and a new (un-initialized) next array is allocated (Line 21). Finally, the local variable k_π – which is used to keep track of how many elements in the current array have already been transferred to the next array – is reset to 0 (Line 22).
2. At this point, we are sure that location $c_\pi[x_\pi]$ exists, and is free to certify a new index. Lines 23–25 tombstone this location and increment x_π if $B[i]$'s initial value happens to already hold the value (π, x_π). (This step eliminates the nasty race condition described earlier.)
3. Lines 26–27 certify location $A[i_\pi]$, by writing a pointer to $\&A[i_\pi]$ in $c_\pi[x_\pi]$ and updating the current array pointer and the length of the valid sub-array values in X$[\pi]$.
4. Since a new location has been filled up in c_π, we must transfer values from c_π to c'_π. If there were no walking-back (described in the next step), then we would need to transfer exactly two values. However, because of potential walk-backs (in previous operations), it is possible that no values need to be transferred in this operation. Lines 28–30 orchestrate this transfer by maintaining k_π's progress relative to x_π.
5. Finally, π performs the synchronization step: it tries to finish its certification process with a Cas on Line 31. If the Cas fails, then it walks-back on Line 32.

That completes the description of Certify$_\pi(i_\pi)$.

Read$_\pi(i_\pi)$ simply checks whether element i_π is certified (Line 6), and returns $A[i_\pi]$ or $f_{init}(i_\pi)$ accordingly (Lines 7–8).

Like the read operation, IsCertified$_\pi(i_\pi)$ is also short. However, its code highlights an important point. The operation starts by reading $b_\pi \leftarrow B[i_\pi]$ which would hold the location of a certificate if i_π was initialized (Line 10). This is where it must be careful. The values in the pair $b_\pi = (b_\pi.pid, b_\pi.loc)$ are directly read from $B[i_\pi]$, and are thus potentially un-initialized ill-formed values. Thus, before the operation proceeds, it must check that $b_\pi.pid$ is indeed a real process id (Line 11). If so, it reads the control information in X$[b_\pi.pid]$ to get a pointer to $c_\pi^{other} = c_{b_\pi.pid}$ and the length of its valid sub-array x_π. *After checking that* $b_\pi.loc$ is indeed in the valid portion of c_π^{other}, it verifies that the location $c_\pi^{other}[b_\pi.loc]$ certifies $A[i_\pi]$ (Line 13). The careful well-formedness checks are necessary to avoid accessing un-allocated portions of memory. Of course, the operation returns *true* only if all of the checks pass (Line 13); if *any* checks fail (well-formedness or otherwise), it simply returns *false* (Line 14).

▶ Remark 2 (old certification arrays). A process π that is performing IsCertified$_\pi(i_\pi)$ may hold a reference c_π^{other} that is no longer the current array of any process. That is, after π reads $(x_\pi, c_\pi^{other}) \leftarrow$ X$[b_\pi.pid]$, the process $b_\pi.pid$ may have certified more elements and updated its current certification array. However, our algorithm remains correct, since the old certification arrays are not de-allocated, and the value of $c_\pi^{other}[b_\pi.loc]$ is guaranteed to be equal to $c_{b_\pi.pid}[b_\pi.loc]$ – the corresponding value in the current array.

The preceding discussion is summarized in the theorem below.

▶ **Theorem 3.** *Algorithm 2 is a linearizable wait-free fast array implementation for p processes. That is, it supports* Initialize$_\pi(m_\pi, f_\pi)$, Read$_\pi(i_\pi)$, *and* Write$_\pi(i_\pi, v_\pi)$ *each with a time complexity of $O(1)$. The total space complexity of the algorithm for supporting k fast-arrays of sizes $m_1, \ldots, m_k$ is $O(M + p)$, where $M = \sum_{j=1}^{k} m_j$.*

5 A Concurrent Fast Generalized Array

In this section, we implement fast generalized arrays, which we motivated in Section 1.2.

Recall that, if $\mathcal{S}$ is the set of hardware-supported RMW primitives, then a *fast generalized array* is an implementation that not only supports $O(1)$-time linearizable $\textsc{Initialize}(m, f)$, $\textsc{Read}(i)$, and $\textsc{Write}(i, v)$ operations, but also supports $O(1)$-time linearizable operations from the set $\mathcal{S}$. To this end, we consider the operation:

- $\mathcal{O}.\textsc{Apply}(i, op, args)$: perform operation op with arguments $args$ on $\mathcal{O}[i]$, and return the response.

Here op can be any RMW operation – such as Write, CAS, FAA, or FAS – that is supported in hardware, and $args$ are the arguments that the primitive requires. For example, if $\mathcal{O}[5] = 17$, then a call to $\mathcal{O}.\textsc{Apply}(5, \textsc{Cas}, (17, 35))$ changes the value of $\mathcal{O}[5]$ to 35 and returns *true*. We term an array that supports $\textsc{Initialize}(m, f)$, $\textsc{Read}(i)$, and $\textsc{Apply}(i, op, args)$, a *generalized array*, and an implementation that runs each operation in $O(1)$ time, a *fast generalized array*. Note that a $\textsc{Write}(i, v)$ can be executed as $\textsc{Apply}(i, \textsc{Write}, v)$. While $\textsc{Read}(i)$ can similarly be executed using $\textsc{Apply}(i, \textsc{Read})$, we design a simpler read method that circumvents the certification overhead for locations that are only read and never updated.

The goal of this section is to design a fast generalized array. We will achieve this goal by building on our ideas from Algorithm 2 for concurrent fast arrays. Therefore, we will continue to use the ideas of individual certification arrays, synchronization and walk-back, array doubling, sharing of certification arrays, and tombstoning. Even so, supporting arbitrary RMW operations poses yet new challenges. We first describe these challenges, and then explain how we overcome them.

Recall that at a high level, our fast array algorithm represents each abstract array element $\mathcal{O}[i]$ by the value of $A[i]$, along with the certification mechanism which keeps track of whether i has been initialized. Thus, $\textsc{Read}(i)$ simply returns $A[i]$ if i is initialized and $f(i)$ otherwise. Write operations, on the other hand, follow an *apply-then-certify* scheme. That is, $\textsc{Write}(i, v)$ blindly *applies* its operation by writing $A[i] \leftarrow v$, and subsequently *certifies* i if necessary.

A natural idea for implementing an RMW operation on $\mathcal{O}[i]$ would be to mimic the apply-then-certify scheme used by writes in Algorithm 2. For example, $\mathcal{O}.\textsc{Apply}(i, \textsc{Cas}, (old, new))$ would blindly apply $r \leftarrow \textsc{Cas}(A[i], old, new)$, and then certify i if necessary, and finally return r. Indeed, this idea would work if i were already initialized, since, in that case, $A[i]$ would hold the value of $\mathcal{O}[i]$. However, if i were not already initialized, then the abstract element $\mathcal{O}[i]$ has the value $f(i)$, while $A[i]$ has some arbitrary value. In particular, if $f(i) = old$, but $A[i] = some\text{-}other\text{-}value$ (not equal to old), then $\mathcal{O}.\textsc{Apply}(i, \textsc{Cas}, (old, new))$ should change $\mathcal{O}[i]$'s value to new and return *true*, but the proposed scheme would keep the value the same (at $\mathcal{O}[i] = A[i] = some\text{-}other\text{-}value$), certify index i, and return *false*. Thus, both the final value of $\mathcal{O}[i]$ and the return value of the operation would be incorrect.

From the example above, we see that RMW operations are difficult to apply before index i is initialized and certified, but easy to apply after the certification process. So, our idea is to reverse the scheme, rather than *apply-then-certify*, we will implement *certify-then-apply*. Since this new scheme will ensure that i is always certified first, the actual application of the RMW primitive can be realized as a hardware primitive applied directly to $A[i]$. Consequently, we can apply *any* primitive operation that hardware supports, not just the select few that we listed at the beginning of the section.

Certifying first poses a new challenge. When we applied writes to $A[i]$ before certifying, we were guaranteed that $A[i]$ would hold a valid (linearizable) value by the time i was certified. Since a reader will return $A[i]$ as the value of $\mathcal{O}[i]$ any time after i is certified,

we still need to guarantee that $A[i] = \mathcal{O}[i]$ at the time of certification. This seems to be a difficult requirement with our current setup, since our previous certification process did not touch $A[i]$, but rather linearized at the time that a CAS was performed on $B[i]$. To overcome this challenge, we introduce the idea of *fusing* as described below.

Fusing. The values stored in each $A[i]$ correspond to the values stored in the corresponding abstract element $\mathcal{O}[i]$. So, it is important to allow these values to take up a full-pointer sized word, e.g., a 64-bit full-word in a modern 64-bit architecture. The pairs (π, j) that we are storing in $B[i]$ however, are just an internal representation used by our algorithm. Furthermore, it is entirely reasonable to assume that this pair can be stored in a single full-word. For example, allocating 14-bits for the process id π would allow for over 16,000 processors, and the remaining 50-bits would be enough to index an array with a thousand-trillion indices (i.e., an array taking up 8000 terabytes of memory). Therefore, we modify our representation by, intuitively, "absorbing the array B into A". Now, each element of our array $A[i]$ will hold a triple $(A[i].val, A[i].pid, A[i].loc)$, where the *value* $A[i].val$ is stored in the first word and the *process id* $A[i].pid$ and the *location* $A[i].loc$ are packed into the second word of a *double-width word*. Modern architectures, such as x86-64, allow us to perform double-width CAS operations on the full double-word $A[i]$, while also allowing all the standard single-width hardware primitives (CAS, FAA, FAS, WRITE, etc.) on the first word $A[i].val$. Using this feature of hardware, we can safely implement the "certify" portion of the certify-then-apply scheme. In particular, if process π reads $a_0 \leftarrow A[i]$ in its "un-initialized" state, and creates a certificate for it in $c_\pi[j]$, it can perform the certify step via: $\text{CAS}(A[i], a_0, (f(i), \pi, j))$.

5.1 The pseudo-code and its description

The pseudo-code for a process π's operations on our fast generalized array is presented as Algorithm 3. The algorithm is built on all of the ideas from the previous section – individual certification arrays, synchronization and walk-back, concurrent array-doubling, tombstoning, and certification mechanism sharing – along with the ideas introduced above – fusing, and the certify-then-apply scheme. We proceed to briefly describe the pseudo-code below.

The code of the three operations in the interface is simple to understand. $\mathcal{O}.\text{INITIALIZE}_\pi(m_\pi, f_\pi)$ simply instantiates a single new un-initialized array A (Line 1), and stores the initialization function (Line 2). $\mathcal{O}.\text{APPLY}_\pi(i_\pi, op_\pi, args_\pi)$ executes certify-then-apply by simply certifying at Line 3 and applying (and returning) at Line 4. $\mathcal{O}.\text{READ}_\pi(i_\pi)$ simply returns $A[i_\pi]$'s value field val if i_π is certified and $f_{init}(i_\pi)$ otherwise (Line 5).

Once again, the main workhorse of the algorithm is the certification mechanism. $\mathcal{O}.\text{CERTIFY}_\pi(i_\pi)$ returns early if i_π is already certified (Lines 13–14). Otherwise, it loads the next available certification location x_π from $\text{X}[\pi]$ at Line 15, and follows the same logical steps as our earlier certification method: (1) update current arrays if necessary (Lines 16–19), (2) tombstone the location if the nasty race condition might arise (Lines 20–22), (3) create a certificate for $A[i_\pi]$ (Lines 23–24), (4) transfer values to the next array (Lines 25–27), and (5) synchronize, and walk-back if necessary (Lines 28–29). The most noteworthy difference from Algorithm 2 is Line 28, where we perform the double-width CAS operation to simultaneously update $A[i_\pi].val$ to $f_{init}(i_\pi)$ and $(A[i].pid, A[i_\pi].loc)$ to (π, x_π).

$\mathcal{O}.\text{ISCERTIFIED}_\pi(i_\pi)$ now reads a triple a_π (rather than the pair b_π) at Line 7, but performs the same logical function as in the standard fast array. It returns *true* only if $a_\pi.pid$ and $c_{a_\pi.pid}[a_\pi.loc]$ are valid, and if so certifies $A[i_\pi]$ (Lines 8–10). Otherwise, it returns *false* (Line 11).

The preceding discussion is summarized in the theorem below.

Algorithm 3 Atomic fast generalized array for p processes. Pseudo-code shown for an arbitrary process π.

Variables:

For each process $\pi \in [p]$ the following variables are shared across *all* fast-arrays $\mathcal{O}$:

- $c_\pi[0,1]$ is a pointer to an allocated un-initialized array of length 2.
- $c'_\pi[0,\ldots,3]$ is a pointer to an allocated un-initialized array of length 4.
- k_π is a non-negative integer that is initialized to 0.
- X is an array, where each X$[\pi]$ stores pair that is initialized to $(0, c_\pi)$.

Each object $\mathcal{O}$ has two instance variables instantiated by $\textsc{Initialize}_\pi(m_\pi, f_\pi)$:

- A is an array of double words.
- f_{init} stores the initial value function.

Each process $\pi \in [p]$ uses the following arbitrarily initialized temporary local variables:

- a_π, a_π^{old}: hold (value, process id, array index) triples.
- x_π: holds an array index.
- c_π^{other}: holds an array pointer.

procedure $\mathcal{O}.\textsc{Initialize}_\pi(m_\pi, f_\pi)$
1: $A \leftarrow$ **new** double-width-array$[m_\pi]$
2: $f_{init} \leftarrow f_\pi$

procedure $\mathcal{O}.\textsc{Apply}_\pi(i_\pi, op_\pi, args_\pi)$
3: $\textsc{Certify}_\pi(i_\pi)$
4: **return** $op_\pi(A[i_\pi].val, args_\pi)$

procedure $\mathcal{O}.\textsc{Read}_\pi(i_\pi)$
5: **if** $\textsc{IsCertified}_\pi(i_\pi)$ **then return** $A[i_\pi].val$ **else return** $f_{init}(i_\pi)$

6: **procedure** $\mathcal{O}.\textsc{IsCertified}_\pi(i_\pi)$
7: $a_\pi \leftarrow A[i_\pi]$
8: **if** $0 \le a_\pi.pid < p$ **then**
9: $(x_\pi, c_\pi^{other}) \leftarrow$ X$[a_\pi.pid]$
10: **if** $0 \le a_\pi.loc < x_\pi$ **and** $c_p^{other}[a_\pi.loc] = \&A[i_\pi]$ **then return** *true*
11: **return** *false*

12: **procedure** $\mathcal{O}.\textsc{Certify}_\pi(i_\pi)$
13: $a_\pi^{old} \leftarrow A[i_\pi]$
14: **if** $\textsc{IsCertified}_\pi(i_\pi)$ **then return**
15: $(x_\pi, -) \leftarrow$ X$[\pi]$
16: **if** $x_\pi \ge c_\pi.len$ **then**
17: $c_\pi \leftarrow c'_\pi$
18: $c'_\pi \leftarrow$ **new** array$[2 \cdot c_\pi.len]$
19: $k_\pi \leftarrow 0$
20: **if** $a_\pi^{old}.pid = p$ **and** $a_\pi^{old}.loc = x_\pi$ **then**
21: $c_\pi[x_\pi] = \perp$
22: $x_\pi \leftarrow x_\pi + 1$
23: $c_\pi[x_\pi] \leftarrow \&A[i_\pi]$
24: X$[\pi] \leftarrow (x_\pi + 1, c_\pi)$
25: **while** $k_\pi < 2x_\pi - c_\pi.len$ **do**
26: $c'_\pi[k_\pi] \leftarrow c_\pi[k_\pi]$
27: $k_\pi \leftarrow k_\pi + 1$
28: **if not** $\textsc{Cas}(A[i_\pi], a_\pi^{old}, (f_{init}(i_\pi), \pi, x_\pi))$ **then**
29: X$[\pi] \leftarrow (x_\pi, c_\pi)$

▶ **Theorem 4.** *Algorithm 3 is a linearizable wait-free fast generalized array implementation for p processes. That is, for each process $\pi \in [p]$, it supports* $\textsc{Initialize}_\pi(m_\pi, f_\pi)$, $\textsc{Read}_\pi(i_\pi)$, *and* $\textsc{Apply}_\pi(i_\pi, op_\pi, args_\pi)$ *each with a time complexity of $O(1)$. The total space complexity of the algorithm for supporting k fast generalized arrays of sizes $m_1, \ldots, m_k$ is $O(M + p)$, where $M = \sum_{j=1}^{k} m_j$, given that each memory word has at least $\lceil \log_2 M \rceil + \lceil \log_2 p \rceil$ bits.*

6 Discussion and Future Work

In this paper, we designed the first algorithms for concurrent fast arrays and fast generalized arrays. Just as sequential fast arrays have found several applications, we envisage future work that explores applications of these concurrent fast arrays. The following directions seem promising.

- The concurrent union-find data structure of Jayanti and Tarjan [19], which is used in the fastest parallel algorithms for computing connected components and spanning forests on CPUs and GPUs [8, 16], requires a generalized array of n nodes, with each node initially pointing to itself, i.e., $f(i) = i$. So, any concurrent union-find object on which only $o(n)$ operations are performed benefits from the use of our fast generalized array.
- A concurrent (standard) fast array is useful for implementing an adjacency matrix, E, of a mutable sparse graph. In particular, adding or removing an edge (i, j) is implemented by writing 1 or 0 (respectively) in $E[i, j]$, and querying an edge (i, j) is a simple read of $E[i, j]$. The real saving lies in storing the graph initially. To store a sparse graph of $m \ll n^2$ edges, we initialize the matrix E with all-zero entries in just $O(1)$ time, and then add the m edges, one at a time. Thus, the entire graph is stored in just $O(m)$ time, instead of the usual $\Theta(n^2)$ time.
- Kanellakis and Shvartsman introduced the *write-all* problem, a version of which is stated as follows: given an array A of length m such that each entry $A[i]$ has an arbitrary initial value, devise an algorithm for p asynchronous processes to initialize each entry $A[i]$ to 0, such that no process returns before the initialization is complete. This problem has attracted a lot of research [2, 5, 12, 20, 24], especially since a write-all solution is a critical subroutine in some implementations of concurrent hash tables [10, 11, 33].

  Although the two problems are different, fast arrays and write-all share the quest to achieve "fast initialization". The difference is that write-all insists on physically initializing each array element, whereas a fast array promises only to create the illusion of initializing each element. Thus, initialization takes only $O(1)$ time with fast arrays, while it takes at least linear time in any solution of write-all. Consequently, if an algorithm that uses a write-all solution can instead be satisfied with a fast array, then the algorithm's speed can potentially improve.
- Allocating a hash table of size n requires $\Theta(n)$ time using conventional arrays (because of initialization). As a result, it has been difficult to implement efficient re-sizable lock-free hash tables [10, 11, 33]. Using fast arrays however, a new table of any size can be allocated in just $O(1)$ time. Exploiting this feature, we are in the process of designing a re-sizeable wait-free hash table that guarantees $O(1)$ average time for *find* and *insert* operations.

We present some brief experiments to measure the empirical efficiency of our standard fast array algorithm in the appendix (Appendix A). We look forward to the further development and deployment of these ideas by algorithmists and practioners alike.

References

1. Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
2. Richard J. Anderson and Heather Woll. Algorithms for the certified write-all problem. *SIAM J. Comput.*, 26(5):1277–1283, 1997.
3. Hagit Attiya, Danny Hendler, and Philipp Woelfel. Tight RMR lower bounds for mutual exclusion and other problems. In Rida A. Bazzi and Boaz Patt-Shamir, editors, *Proceedings of the Twenty-Seventh Annual ACM Symposium on Principles of Distributed Computing, PODC 2008, Toronto, Canada, August 18-21, 2008*, page 447. ACM, 2008. doi:10.1145/1400751.1400843.
4. Jon Louis Bentley. *Programming pearls*. Addison-Wesley, 1986.
5. Jonathan F. Buss, Paris C. Kanellakis, Prabhakar L. Ragde, and Alex Allister Shvartsman. Parallel algorithms with processor failures and delays. *Journal of Algorithms*, 20(1):45–86, 1996.
6. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.
7. Robert Cypher. The communication requirements of mutual exclusion. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 147–156, 1995.
8. Laxman Dhulipala, Changwan Hong, and Julian Shun. Connectit: A framework for static and incremental parallel graph connectivity algorithms. *Proc. VLDB Endow.*, 14(4):653–667, 2020.
9. Kimmo Fredriksson and Pekka Kilpeläinen. Practically efficient array initialization. *Softw. Pract. Exp.*, 46(4):435–467, 2016.
10. Hui Gao, Jan Friso Groote, and Wim H. Hesselink. Almost wait-free resizable hashtable. In *International Parallel and Distributed Processing Symposium*, 2004.
11. Hui Gao, Jan Friso Groote, and Wim H. Hesselink. Lock-free dynamic hash tables with open addressing. *Distributed Comput.*, 18(1):21–42, 2005.
12. Jan Groote, Wim Hesselink, and Sjouke Mauw. An algorithm for the asynchronous write-all problem based on process collision. *Distributed Computing*, 14:75–81, April 2001.
13. Torben Hagerup and Frank Kammer. On-the-fly array initialization in less space. In *International Symposium on Algorithms and Computation*, volume 92, pages 44:1–44:12, 2017.
14. Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.
15. Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
16. Changwan Hong, Laxman Dhulipala, and Julian Shun. Exploring the design space of static and incremental graph connectivity algorithms on GPUs. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, pages 55–69, 2020.
17. Intel. Intel 64 and IA-32 architectures software developer manuals, 2020. URL: https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html.
18. Masakazu Ishihata, Shan Gao, and Shin-ichi Minato. Fast message passing algorithm using ZDD-based local structure compilation. In *Proceedings of the Workshop on Advanced Methodologies for Bayesian Networks*, volume 73, pages 117–128, 2017.
19. Siddhartha V. Jayanti and Robert E. Tarjan. A randomized concurrent algorithm for disjoint set union. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 75–82, 2016.
20. Paris C. Kanellakis and Alex A. Shvartsman. Efficient parallel algorithms can be made robust. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, page 211–219, 1989.
21. Takashi Katoh and Keisuke Goto. In-place initializable arrays. *CoRR*, abs/1709.08900, 2017.
22. Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams*. Addison-Wesley Professional, 12th edition, 2009.

23 Jacob Teo Por Loong, Jelani Nelson, and Huacheng Yu. Fillable arrays with constant time operations and a single bit of redundancy. *CoRR*, abs/1709.09574, 2017. `arXiv:1709.09574`.

24 C. Martel, R. Subramonian, and A. Part. Asynchronous PRAMs are (almost) as good as synchronous PRAMs. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, pages 590–599, 1990.

25 Kurt Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*, volume 1 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1984.

26 John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.

27 Shin-ichi Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proceedings of the International Design Automation Conference*, page 272–277, 1993.

28 Shin-ichi Minato. Counting by ZDD. In *Encyclopedia of Algorithms*, pages 454–458. Springer Publishing Company, 2016.

29 Shin-ichi Minato. Power of enumeration—recent topics on BDD/ZDD-based techniques for discrete structure manipulation. *IEICE Trans. Inf. Syst.*, 100-D(8):1556–1562, 2017.

30 Dana Moshkovitz and Bruce Tidor. Lecture notes 15: van Emde Boas data structure. URL: `https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-design-and-analysis-of-algorithms-spring-2012/lecture-notes/MIT6_046JS12_lec15.pdf`.

31 Gonzalo Navarro. Constant-time array initialization in little space. In *Proceedings of the International Conference of the Chilean Computer Science Society (SCCC)*, 2012.

32 Gonzalo Navarro. Spaces, trees, and colors: The algorithmic landscape of document retrieval on sequences. *ACM Comput. Surv.*, 46(4):52:1–52:47, 2013.

33 Ori Shalev and Nir Shavit. Split-ordered lists: Lock-free extensible hash tables. *J. ACM*, 53(3):379–405, 2006.

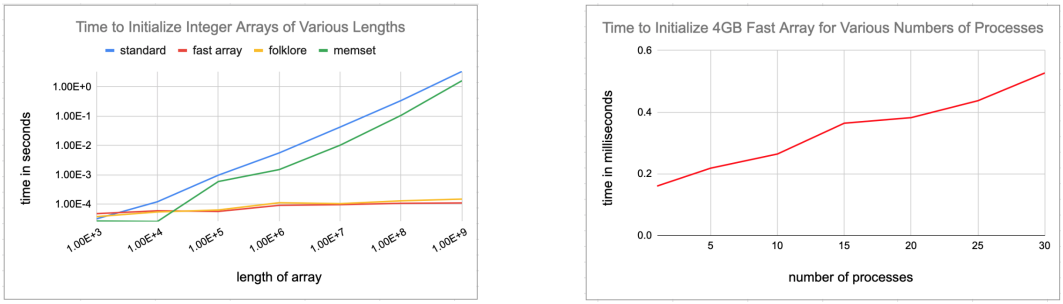
34 Hirofumi Suzuki and Shin-ichi Minato. Fast enumeration of all pareto-optimal solutions for 0-1 multi-objective knapsack problems using ZDDs. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, 101-A(9):1375–1382, 2018.

35 Ryo Yoshinaka, Toshiki Saitoh, Jun Kawahara, Koji Tsuruma, Hiroaki Iwashita, and Shin-ichi Minato. Finding all solutions and instances of numberlink and slitherlink by ZDDs. *Algorithms*, 5(2):176–213, 2012.

A Experiments

We perform experiments using two 8-core Intel Xeon E5-2670 CPUs with two-way hyper-threading. The machine has 64GB of DRAM. Our machine ran 64-bit Ubuntu 12.04 with Linux kernel 3.13.0-143. All algorithms were coded in C++ and without any optimizations or specific algorithmic engineering to increase the speeds from the pseudo-code presented in the body of the paper. We used `std::threads` to implement our concurrent fast array, and we compiled our code with g++ version 4.8.4 with the `-std=c++11`, `-pthread`, and `-mcx16` options set. We experiment with four different algorithms:

1. *standard:* a classic array, where initialization is performed by a linear-time for-loop through the indices of the array.
2. *memset:* a classic array, where initialization is performed by the C++ `memset` primitive. The `memset` operation can only be used to initialize an array to all 0s. In particular, it cannot be used to initialize it to an arbitrary function f.
3. *folklore:* the sequential folklore fast array algorithm (i.e., Algorithm 1). This algorithm can only be used by a single process; it is not a concurrent algorithm, but it can serve as a baseline.
4. *fast array:* our concurrent fast array (i.e., Algorithm 2).



(a) **(b)**

Figure 1 Figure 1a is a log-log plot charting the time to initialize arrays of various lengths for a single process. Figure 1b is a plot charting the times to initialize fast arrays of length one billion for various numbers of processes.

Our experiments focus on measuring the speeds of the three operations – INITIALIZE(), READ(), and WRITE(). We present the results below.

1. To compare the speed of INITIALIZE() across all four algorithms, we measure the time to initialize arrays to all zeroes. The array lengths we test are $m = 10^k$ for $k \in \{3, 4, 5, 6, 7, 8, 9\}$ with each entry being 4 bytes, i.e., arrays of size 4KB to 4GB. As predicted by the algorithmic analysis, the folklore and fast array algorithms take only constant time to initialize, while initializing by for-loop and `memset` take linear time.
   As shown in Figure 1a, initializing an array with `memset` is 1.2–4.7 times faster than initializing with a for-loop, however initializing a fast array of length one billion is more than 14,000 times faster than initializing with `memset`. As the length of the array gets smaller, the initialization time advantage of fast arrays reduces. Fast array initialization and `memset` initialization become equally fast at an array length between 10^4 and 10^5, and fast array initialization and for-loop initialization become equally fast at an array length between 10^3 and 10^4. We consistently observe that initializing a fast array takes only as much time as initializing a folklore array. As shown in Figure 1b, it takes only 3.3 times more time to initialize a 4GB fast array for 30 processes rather than one for a single process.
2. While fast arrays are much faster to initialize than standard arrays, read and write operations are slower on these arrays at all levels of concurrency. In order to measure exactly how much slower fast array reads are, we measure the cumulative time to perform one million reads using each type of array. Since the fast array READ() algorithm is different for indices that have been "initialized" – i.e., written to at least once after initialization – versus those that are "uninitialized", we measure the two types of reads separately (see Figure 2).
   The main takeaways of the experiment are as follows: (1) the two different types of reads – initialized and uninitialized – on fast arrays are of comparable speed; (2) fast array reads are 2.1–4.3 times slower than reads to standard arrays.
3. Algorithm 2 suggests that writes to "initialized" indices should be faster than those to "un-initialized" indices. This is also confirmed by our experiment shown in Figure 3. In particular, fast array writes to un-initialized indices are 6–21 times slower than standard writes, but writes to initialized indices are only 2.2–4.9 times slower. It is noteworthy here that the slower speed only occurs once per index, and all subsequent writes to that index happen at the faster speed.

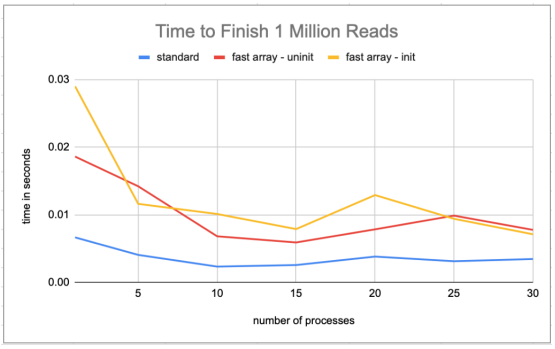


Figure 2 This plot compares the time to read from one million indices of a standard array versus a fast array for various numbers of concurrent processes. For the fast array, both reads from uninitialized and initialized indices are compared.

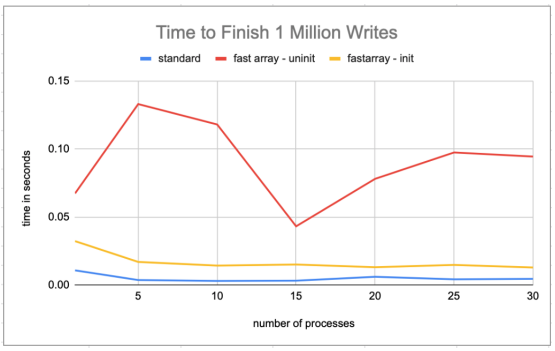


Figure 3 This plot compares the time to write to one million indices of a standard array versus a fast array for various numbers of concurrent processes. For the fast array, both writes to uninitialized and initialized indices are compared.

4. Writing to a new index in a fast array is 6–21× slower than writing to a new index in a naive array. So, fast arrays maintain their initial advantage over standard arrays as long as only 4–17% of the array indices are written to.