RCUArray: An RCU-like Parallel-Safe Distributed Resizable Array

Louis Jenkins
Bloomsburg University
lpj11535@huskies.bloomu.edu

***Abstract*—Presented in this work is RCUArray, a parallel-safe distributed array that allows for read and update operations to occur concurrently with a resize via Read-Copy-Update. Also presented is a novel extension to Epoch-Based Reclamation (EBR) that functions without the requirement for either Task-Local or Thread-Local storage, as the Chapel language currently lacks a notion of either. Also presented is an extension to Quiescent State-Based Reclamation (QSBR) that is implemented in Chapel's runtime and allows for parallel-safe memory reclamation of arbitrary data. At 32-nodes with 44-cores per node, the RCUArray with EBR provides only 20% of the performance of an unsynchronized Chapel block distributed array for read and update operations but near-equivalent with QSBR; in both cases RCUArray is up to 40x faster for resize operations.**

I. Introduction and Background

Chapel's arrays and distributions have a robust and complex design with generality at its core and while they host a wide variety of operations they are not parallel-safe while being resized. Mutual exclusion provides an easy solution but inhibits scalability and introduces problems such as deadlock, priority inversion, and convoying [6], [8]. Reader-writer locks take a step in the right direction by allowing concurrent readers, but have the drawback of enforcing mutual exclusion with a single writer. Scalability is only half the battle as the root of many problems in the design of high-performance data structures is memory reclamation; caution must be used in the reclamation of memory that may be accessed concurrently in a language without garbage collection. Mechanisms such as Hazard Pointers [14] can provide a safe non-blocking approach for memory reclamation with a balanced but noticeable overhead to both read and write operations. These mechanisms ensure high throughput for most non-blocking data structures but are unsuitable when the performance of reads is far more important than the performance of writes. Furthermore, the mechanisms require some notion of task-local or thread-local storage, which the Chapel language currently lacks.

Read-Copy-Update (RCU) [10] and in particular the userspace variant [4] is a more recent type of synchronization strategy which allows parallel-safe reads during a write, offering significant improvements in performance over locking [12]. It is not without drawbacks, as writers must perform the task of memory reclamation by waiting for all readers to *evacuate* by finishing their operation. RCU can come in two flavors: *Epoch-Based Reclamation* (EBR) [5] which requires readers to enter *read-side critical sections* in which they indicate that they are accessing the protected data, and *Quiescent State-Based Reclamation* (QSBR) [15] in which readers must periodically invoke *checkpoints* to explicitly notify that they no longer have access to the protected data. QSBR comes with the benefit of ensuring that readers may proceed without overhead, but it is entirely application-dependent as strategic placement of checkpoints is required. EBR comes with a small overhead of forcing readers to make use of memory barriers, but can be implemented in a much wider variety of applications [7]. Unfortunately both known variants of RCU require the usage of thread-local or task-local storage (TLS).

In this work I present RCUArray, a distributed array that may be used in place of Chapel's arrays and distributions that provides an additional feature: parallel-safe resizing. I also present a novel extension to RCU based on EBR that does not require thread-local or task-local storage and provides scalable perfomance but at the cost of additional overhead for read and update operations. Finally I also present an implementation of QSBR in Chapel's runtime that can be used to perform memory reclamation on arbitrary data and comes without overhead either for read and update operations.

II. Related Work

Applications of RCU can be seen in various data structures such as linked lists, balanced trees [2] and hash tables [17]. To allow greater concurrency for write operations, an extension to RCU called Read-Log-Update [11] provides an interesting solution by borrowing concepts from software transactional memory [16] to allow for multiple concurrent writers via means of write logs to provide isolation, conflict detection and resolution. Another extension is Predicate RCU [1] which makes use of a user-supplied predicate to determine whether a writer should wait for a concurrent reader. Another related work that provides a resizable array is from Damian et al. [3] who presented a lock-free resizable array used as a vector that makes use of a helping algorithm and operation descriptors. While there are related efforts on the application of RCU in data structures, in the research and development of thread-safe resizable arrays, and in the deployment of RCU in a distributed context [13], there are none to the author's knowledge that combine the application of all three.

III. Design

RCUArray is simple in design but overcomes three core challenges: (1) parallel-safe memory reclamation; (2) concurrency of read and update operations even while the data

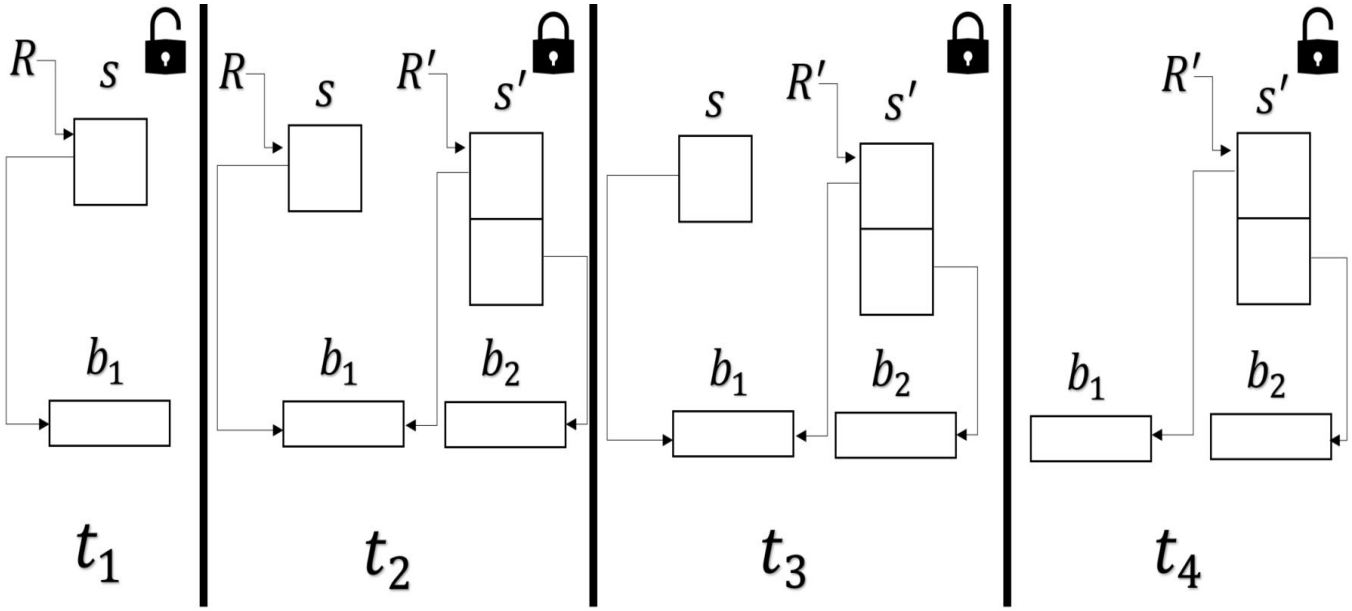


Fig. 1: Example RCUArray resize operation. At t_1 there is a reader R that is using the current GlobalSnapshot s to access the block b_1. At t_2 a writer has acquired the cluster WriteLock, has performed its clone operation on s to produce s', appended the new block b_2 to s', and has set the current GlobalSnapshot to s'. While the writer was waiting for R to finish its operation on s, a new reader R' begins its operation on s'. R finishes its operation at t_3 and the writer safely reclaims s and releases the WriteLock at t_4.

structure is in the process of being resized; and (3) distribution across multiple nodes in a cluster.

Listing 1 displays the two data types and their fields that are used in the array but only the ones that are used in both implementations will be covered in this section. Both data types are privatized[1] and many of their fields are mutated in a manner that is *node-local* and independent of other privatized copies. The privatization id, PID, is a descriptor used to access the privatized instance allocated on each node. GlobalSnapshot is the current *snapshot*[2] of metadata; Each snapshot of metadata is an RCUArraySnapshot, equivalent to an array of blocks where each block is an array with a capacity of BlockSize. WriteLock is a cluster-wide lock, in this case a lock that is wrapped in some class allocated on a single node, used to provide mutual exclusion with respect to all nodes during resize operations. NextLocaleId is used as a naive counter to handle distributing the allocation of blocks across multiple nodes in a block distributed fashion.

Listing 1: Data Structure Types

```
Constants:
  BlockSize :     uint
RCUArrayMetaData:
  PID :           int
  EpochReaders :  [0..1] atomic uint
  GlobalEpoch :   atomic uint
  GlobalSnapshot : RCUArraySnapshot
  WriteLock :     GlobalLock
  NextLocaleId :  int
RCUArraySnapshot:
  Blocks :  [0..-1] Block
```

A. Epoch-Based Reclamation of Snapshots

Epoch-Based Reclamation (EBR) is a strategy where concurrent readers must pass through a barrier to enter what is called a *read-side critical section* which ensures that a concurrent writer does not reclaim the memory we are interested in until appropriate, detailed in Algorithm 1. An *epoch* is a version number that corresponds to a snapshot, and each node maintains the current epoch, GlobalEpoch, which is an atomic monotonically increasing counter. A reader must notify potential writers of the epoch they are using to ensure that reclamation of the respective snapshot is safely deferred. Due to the lack of TLS, readers are unable to broadcast such notifications individually and instead do so *collectively* using a set of two atomic counters, EpochReaders. The parity of the epoch determines which of the EpochReaders to use to *record* the operation as *in-progress*, done by performing an atomic increment, and later to record as finished, done by performing an atomic decrement; a writer must wait until all recorded in-progress operations are finished before it may reclaim the corresponding snapshot. The EpochReaders become the point of linearizability [9] where readers can ensure that they are appropriately seen by a concurrent writer, ergo safe to proceed.

A parallel-safe write operation λ can be performed via RCU_Write. [3] As a writer W must ensure that each snapshot is immutable, a clone of the current GlobalSnapshot s is created as s', the λ function is applied on s', and s' becomes the new GlobalSnapshot (lines 1 – 4). To ensure s' will become immediately visible as the new GlobalSnapshot, W performs

[1] A shallow copy of the object is allocated on each node to eliminate inter-node communication.

[2] An immutable version of data.

[3] The WriteLock should be acquired prior to invoking RCU_Write.

Algorithm 1: RCU Pseudocode

```
   // Applies a side-effect inducing function λ to protected data
   proc RCU_Write (λ)
1      oldSnapshot ← GlobalSnapshot;
2      newSnapshot ← clone(oldSnapshot);
       // Update performed on clone, clone becomes new snapshot
3      λ(newSnapshot);
4      GlobalSnapshot ← newSnapshot;
5      epoch ← GlobalEpoch.fetchAdd(1);
       // Wait for readers...
6      readIdx ← epoch % 2;
7      waitForReaders(readIdx);
       // Safe to delete...
8      delete(oldSnapshot);

   // Applies a function λ to protected data with a result
   proc RCU_Read (λ)
9      while true do
           // Attempt to record our read
10         epoch ← GlobalEpoch.read();
11         readIdx ← epoch % 2;
12         EpochReaders[readIdx].add(1);
           // Did snapshot possibly change before we recorded?
13         if epoch = GlobalEpoch.read() then
               // Safe to apply user function
14             retval ← λ(GlobalSnapshot);
15             EpochReaders[readIdx].sub(1);
16             return retval;
           // Try again
17         EpochReaders[readIdx].sub(1);
```

an atomic fetchAdd to update the current GlobalEpoch from e to $e' = e + 1$ and waits for all readers that recorded their operation for e (lines 5 – 7). Only after all recorded operations have evacuated can W reclaim s (line 8).

A parallel-safe read operation λ can be performed via RCU_Read. As operations must be performed collectively, the act of recording the operation is divided into two steps: incrementing and verification. A reader R first reads the current GlobalEpoch e and increments the EpochReaders counter based on the parity of e (lines 10 – 12). It is possible that a concurrent writer W will change the GlobalEpoch from e to e' after $R's$ read but prior to $R's$ increment, which may cause W to not see nor wait for $R's$ operation to finish before performing memory reclamation. While R will see the snapshot s set by W, a future writer W' will also fail to see $R's$ operation as W' will be waiting for readers who recorded based on the parity of e' and may end up reclaiming s while R is applying its λ operation. To remedy this R performs a verification check to determine whether the GlobalEpoch has changed values between our read and increment (line 13). If there has been a change in the GlobalEpoch, such as the above scenario where GlobalEpoch has changed from e to e', R would see that $e \neq e'$ and would undo the operation (line 17) and loop again (line 9). If there has not been a change in the GlobalEpoch, then R has *linearized*. R applies its λ operation to the current GlobalSnapshot, decrements the appropriate EpochReaders counter, and returns the result obtained from the λ (line 14 – 16).

Correctness of the algorithm can be proven further by means of the following 3 lemmas:

Lemma 1. *There will be at most two active snapshots at any given time.*

Proof Sketch: Given a writer W that has acquired the WriteLock, if W updates the GlobalSnapshot from s to s' at time t, a concurrent reader R that linearized prior to t will see s but a concurrent reader R' that linearized after t will see s', hence there are the two active snapshots: s and s'. W must wait for R to evacuate before it may reclaim s, and only then can W release the WriteLock, leaving only one active snapshot: s'. ■

Lemma 2. *Two EpochReaders are sufficient for ensuring safe memory reclamation of snapshots, even in the event of integer overflow of the GlobalEpoch.*

Proof Sketch: As there can be only two active snapshots at any given time, s and s', we can associate to them their respective epochs, e and e'. As the GlobalEpoch is monotonically increasing, $e' = e + 1$, hence e and e' are of different parity. If we represent epochs as N-bit integers, we can then represent them as the binary string $B = (b_1, b_2, ..., b_N)$ where $\forall b \in B, b \in \{0, 1\}$ and where b_1 is the least significant bit. If we have $e = (1, 1, ..., 1)$ being the largest possible value, and $e' = e + 1 = (0, 0, ..., 0)$ overflowing to the smallest possible value, the parity is still preserved and so is the correctness of the EpochReaders.

Given the event of epoch overflow where a preempted reader R reads the GlobalEpoch e at some time t and the GlobalEpoch overflows back to the value of e at some time t' where $t' > t$, correctness is still preserved. If we represent epochs as 1-bit integers, given a scenario where a writer W updates the GlobalEpoch $e = 0$ to $e' = 1$, and another writer W' updates GlobalEpoch $e' = 1$ to $e'' = 0$, if R increments the EpochReaders associated with e and performs its verification check, it will succeed since $e = e'' = 0$. As W' will only wait on readers that have recorded for $e' = 1$, it will not wait for R, but in this case R will see the snapshot s set by W' which is safe since a future writer W'' that updates GlobalEpoch $e'' = 0$ to $e''' = 1$ will wait on R as $e = e'' = 0$ before reclaiming s. ■

Lemma 3. *After a reader R has recorded and verified its operation, it may safely access the current GlobalSnapshot without it being reclaimed.*

Proof Sketch: Given a writer W that acquires the WriteLock at a time t_{acq}, releases the WriteLock at a time t_{rel}, updates the GlobalSnapshot from s to s' at a time $t_s \in (t_{acq}, t_{rel})$ and updates the GlobalEpoch from e to e' at a time $t_e \in (t_s, t_{rel})$, and a reader R that linearizes at some time $t \in [t_{acq}, t_{rel}]$: if $t \in [t_{acq}, t_s)$ then R will see s and e; if $t \in [t_s, t_e)$ then R will see s' and e; if $t \in [t_e, t_{rel}]$ then R will see s' and e'. Note that it is safe for R to operate on s when it has recorded for e, and safe to operate on s' when it has recorded for e', but it may not be so clear that it is safe for R to operate on s' when it has recorded for e. This is safe as W will not reclaim s' nor s, and while it does result in W waiting on R unnecessarily it has no impact on safety. ■

B. Runtime Support for Quiescent State-Based Reclamation

Quiescent State-Based Reclamation (QSBR) is a strategy in which all participants, whether reader, writer, or updater, must

Algorithm 2: QSBR Pseudocode

```
   // Defers memory reclamation of objs until safe
   proc QSBR_Defer (objs)
1      tls ← getTLS();
       // Update and observe the new global state.
2      tls.ObservedEpoch ← StateEpoch.fetchAdd(1) + 1;
3      tls.DeferList.push(objs, tls.ObservedEpoch);

   // Handle memory reclamation for DeferList if eligible
   proc QSBR_Checkpoint ()
4      tls ← getTLS();
       // Observe the current state.
5      tls.ObservedEpoch ← StateEpoch.read();
       // Find smallest (safest) epoch
6      minEpoch ← tls.ObservedEpoch;
7      for tls' in TLSList
8          minEpoch ← min(tls'.ObservedEpoch, minEpoch)
       // Split DeferList where the safe epoch ≤ minimum epoch.
9      head ← tls.DeferList.popLessEqual(minEpoch);
10     while head ≠ nil
11         tmp ← head;
12         head ← head.next;
13         delete tmp;
```

periodically invoke *checkpoints* to ensure eventual memory reclamation. To generalize this concept, QSBR is decoupled from RCU, is extended to make use of epochs in a manner similar to EBR, and is implemented in Chapel's runtime which provides access to thread-local storage. An atomic monotonically increasing counter is maintained that denotes the epoch as a state of the entire system, StateEpoch; whenever memory is to be reclaimed the StateEpoch must be incremented to reflect this state change, and during checkpoints all participants must notify that they are seeing the newest state. All *threads* act as participants and keep track of their own thread-specific[4] metadata, which is also accessible via a linked list, TLSList. Each time memory reclamation is desired, instead of waiting for all other threads to invoke a checkpoint and risk entering deadlock, we append the memory to be reclaimed to a list, DeferList. To determine when it is safe to reclaim memory we couple the *safe epoch*, the minimum epoch that all threads need to *observe* for safe memory reclamation, to defer processing at checkpoints. As the DeferList is thread-specific, memory reclamation can be performed in a parallel-safe manner and because it holds the safe epoch it can be traversed to determine which objects are safe for memory reclaimation in a lockless manner. A feature that is supported but not discussed in detail in this work is the support for parking and unparking of threads which occurs when a thread is idle without a task and is used to cleanup its own DeferList, notify of its quiescence, and to provide assistance with bookkeeping.

QSBR, formally described in Algorithm 2, can be used as a general-purpose memory reclamation device with negligible overhead, but does come with its share of downsides. For example, it is not safe to dereference any memory managed by QSBR if it has been acquired prior to a checkpoint or deferal of memory reclamation, as this *QSBR-protected* memory could have been marked for deletion by another thread. As well, since Chapel tasks can be multiplexed on the same thread, they can share the same TLS and it is not recommended that tasks yield while intending to dereference memory that is QSBR-protected, nor should it be used in any future tasking layers that are preemptible. Lastly it is unclear whether checkpoints should be injected by the compiler, placed at strategic points in the runtime, or invoked manually by the user.

When memory is to be reclaimed via QSBR_Defer, the StateEpoch is atomically updated from e to $e' = e + 1$[5], notifying that the old state described by e is being discarded in favor of the newer state described by e'. The current thread T observes the new state e', making the promise that it has become entirely quiescent of the state described by e or of any prior state (lines 1 – 2). The memory to be reclaimed m is coupled with e' as the safe epoch and is pushed in Last-In-First-Out order on $T's$ DeferList, deferring further processing to $T's$ next checkpoint (line 3). For future convenience, DeferList entries are represented as the triple (m, e, t) where m is the memory to be reclaimed, e is the safe epoch, and t is the time of insertion into the DeferList[6].

When a checkpoint is to be invoked via QSBR_Checkpoint by a thread T, T will observe the current StateEpoch e, making a promise of quiescence of any state prior to e. (lines 4 – 5). T will then find the minimum observed epoch e_{min} of all threads (lines 7 – 8). We then split the DeferList at the first entry with a safe epoch less than or equal to e_{min} and handle deletion (lines 9 – 13).

Correctness of the algorithm can be proven further by means of the following 2 lemmas:

Lemma 4. *If StateEpoch does not overflow, DeferList is sorted by safe epoch in descending order.*

Proof Sketch: Given that StateEpoch is monotonically increasing, insertions are handled sequentially on the same thread, and that the previous head of the DeferList is (m, e, t), if another entry (m', e', t') is inserted into the list, then $t' > t$ and therefore $e' > e$ as the safe epoch is always derived from the StateEpoch. Since entries are inserted at the head in Last-In-First-Out order, and since each successive insertion has a larger safe epoch than its predecessor, the list is sorted in descending order. ■

Lemma 5. *Given a DeferList entry with safe epoch e, memory reclamation is safe if $e_{min} \geq e$ where e_{min} is the minimum observed epoch of all threads so long as StateEpoch does not overflow.*

Proof Sketch: Assume the opposite is true that it is not safe to reclaim the DeferList entry. Given a DeferList entry (m, e, t), if any thread T has invoked a checkpoint or deferred memory for safe reclamation at some time t_T, if $t_T > t$ then T has observed some epoch e_T such that $e_T \geq e_{min}$ and can no longer access m after becoming quiescent. However for the reclamation of m to be unsafe it would need to be accessible by some thread T' such that it has observed some epoch $e_{T'}$ such that $e_{min} > e_{T'}$, but e_{min} is the minimum observed epoch of all threads, hence this is a contradiction. ■

[4]Using thread-local storage to keep track of data that is owned by the thread.

[5]If $e' = e + 1$ were to result in overflow, the algorithm would be subject to undefined behavior.

[6]The time t is only used to prove correctness of the design and is not required in the actual implementation.

C. Concurrent Updates and Resizing

To allow for *update* operations, which are assignments to some indexable portion of the array,[7] to attain performance equivalent to that of a read operation, the λ can return a reference[8] to the desired portion of the array to be written to later. This does not come without its own set of problems, as it is possible for updates to a previous snapshot to be *lost*. Given some updater U with some function that returns by reference λ running concurrently with a writer W with some function λ', consider the scenario where U has appropriately linearized and returned the reference r obtained by applying λ to the current snapshot s at some time t_{ret} and performs a non-zero amount of assignment through r at some time t_{fin}. If W clones s to create s' at some time $t_{cln} \in [t_{ret}, t_{fin}]$, then $U's$ assignment through r will be lost to s', lost to λ' as it is applied to s', and finally it will be lost to all future writers, updater, and readers once s' is set as the new GlobalSnapshot.

To prevent the loss of these updates, a clone of a snapshot s will *recycle* the blocks in s when creating s', depicted in Figure 1. During the cloning process for a writer, each block is recycled by the newer snapshot to ensure that any updates to the older snapshot is visible via the indirection. This indirection not only comes with very little cost to performance, it also allows updates to share the same performance as reads. Furthermore, recycling blocks of memory proves to be significantly faster than copying by value into larger memory.

Correctness of the algorithm can be proven further by means of the following lemma:

Lemma 6. *Given a writer W, an updater U, and the GlobalSnapshot s, if W has started its clone on s to produce s' and U performs a non-zero number of assignments through its reference r to s, those assignments will be immediately visible to s'.*

Proof Sketch: Given that s is a snapshot with N blocks represented by the sequence $(b_1, b_2, ..., b_N)$, cloning s to create a larger snapshot s' with M blocks can be represented as the sequence $(b_1, b_2, ..., b_N, b_{N+1}, ..., b_M)$; that is s becomes a subsequence of s' where $\forall i \in [1..N] : s(i) = s'(i)$. Hence any block that r refers to in s is also recycled in s', and any assignment that U performs through r will be visible to both s and s'. ∎

D. Distribution

Blocks of the array are distributed in a round-robin fashion similar to a block-cyclic distribution.[9] If a writer W performs its function λ to change the GlobalSnapshot from s to s', this change can be propogated by replicating the operation across all nodes in parallel. As a benefit of replicating these operations across all nodes, both read and update operations act mostly on node-local metadata, significantly improving their locality; their only required communication being PUT and GET operations to distributed blocks of the array.[10]

Algorithm 3: Implementation Pseudocode

```
   // Indexes into array
   proc Index (idx) ref
       proc Helper (snapshot) ref
1          blockIdx ← idx / BlockSize;
2          elemIdx ← idx % BlockSize;
3          return snapshot.blocks[blockIdx][elemIdx];
4      pThis ← chpl_getPrivatizedCopy(PID);
5      if isQSBR then
6          return Helper(pThis.GlobalSnapshot);
7      else
8          return pThis.RCU_Read(Helper);

   // Expands the size of the array
   proc Resize (size)
9      newBlocks : [1..0]Block;
10     WriteLock.acquire();
11     locId ← NextLocaleId;
       // Allocate and distribute new blocks
12     while size > 0
13         on Locales[locId] do
14             newBlocks.push_back(newBlock());
15         locId ← (locId + 1) % numLocales;
16         size ← size − BlockSize;
       // Function to append blocks to snapshot
       proc Helper (snapshot)
17         snapshot.blocks.push_back(newBlocks);
       // Update performed on each node
18     coforall loc in Locales do on loc
19         pThis ← chpl_getPrivatizedCopy(PID);
20         if isQSBR then
               // Handle RCU directly with QSBR...
21             oldSnapshot ← pThis.GlobalSnapshot;
22             newSnapshot ← clone(oldSnapshot);
23             Helper(newSnapshot);
24             pThis.GlobalSnapshot ← newSnapshot;
25             QSBR_Defer(oldSnapshot);
26         else
27             pThis.RCU_Write(Helper);
28         pThis.NextLocaleId ← locId;
29     WriteLock.release();
```

IV. Implementation

The implementation of RCUArray makes use of either EBR or QSBR, and the required changes in implementation are minor and can be contained within a single conditional using the compile-time parameter, $isQSBR$. As displayed in Algorithm 3, the implementation makes use of Chapel-specific constructs such as nested procedures which have access to local variables declared in the scope of their parents, and the combination of the 'coforall' and 'on' statements which spawn a task on each node to run in parallel.

A. Indexing

Both read and update operations can be performed through the reference returned via Index.[11] The nested procedure Helper is defined and used to identify both the block and the offset being requested and return it by reference (lines 1 – 3). After obtaining the privatized copy via the runtime function

[7] All assignments are performed on the blocks of memory the array is composed of.

[8] In languages that do not support references, this can be accomplished by returning a pointer instead.

[9] More complex distribution patterns are beyond the scope of this work.

[10] In Chapel, these PUT/GET operations are performed behind-the-scenes, and so both readers and updaters are completely oblivious of all communication.

[11] For brevity, no checks for out-of-bounds are performed.

chpl_getPrivatizedCopy for the current task C (line 4), a check is performed to determine the configuration of the RCUArray (line 5). If configured to use QSBR, C will perform the Helper operation directly on the node-local GlobalSnapshot as it will not be reclaimed until C later invokes a checkpoint (line 6); if not configured, C will instead invoke RCU_Read on the privatized copy with Helper as the λ function (line 8).

B. Resizing

Resizing is performed through Resize, which takes as argument the amount to expand the RCUArray.[12] Making use of Chapel's syntax for defining arrays, an empty array B is created to hold blocks used as temporary storage (line 9). After the current task C acquires mutual exclusion (line 10), C performs round-robin allocation of blocks and appends each to B (lines 11 – 16). The nested procedure Helper is defined and used to append B to the current snapshot (line 17). C then spawns a task C' on each node in the cluster (line 18), C' obtains its privatized copy (line 19), and a check is performed to determine the configuration of the RCUArray (line 20). If configured to use QSBR, C' will clone the old snapshot s of the privatized copy to create s', C' will apply the Helper function directly on s' and set the GlobalSnapshot of the privatized copy to s', and finally C' will defer the memory reclamation of s to QSBR_Defer(lines 21 – 25). If not configured for QSBR, C' will instead invoke RCU_Update on the privatized copy with Helper as the λ function (line 27). Finally C' will update the counter used for round-robin allocation before completing (line 28). After C' completes, C has also completed and will release mutual exclusion (line 29).

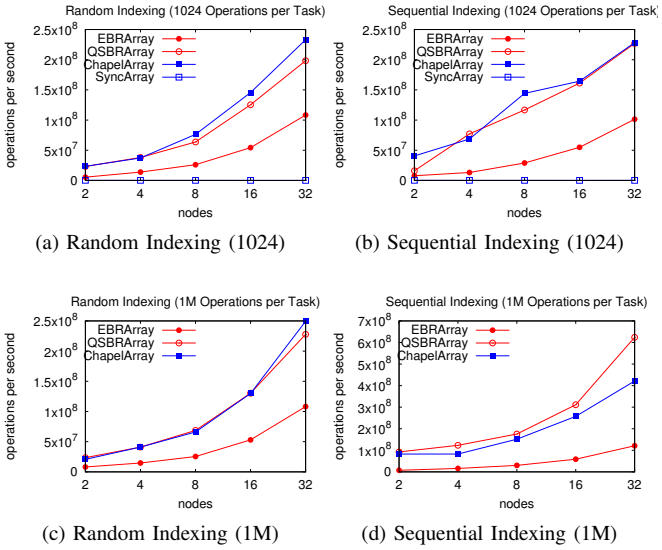
V. Performance Evaluation

Variants of RCUArray using QSBR (QSBRArray)[13] and EBR (EBRArray) were tested against each other, and when appropriate against an unsynchronized naive block distributed array using Chapel's standard BlockDist distribution (UnsafeArray). Compared are the performance of read, update, and resizing operations. While UnsafeArray allows for concurrent read and update operations, it is unable to allow concurrent resize operations and so a safer variant is defined that uses mutual exclusion via sync variables (SyncArray). All benchmarks were performed using a subset of a Cray XC50 cluster totaling 32-nodes, each node running Intel Xeon Broadwell 44-core processors; unless stated otherwise, results obtained from a single-node will be excluded due to the regressions in performance caused by introducing communications. All benchmarks are compiled under a fork of Chapel 1.17 pre-release,[14] optimized via the "--fast" flag, built with the qthread tasking layer, and under the Cray Compiler Environment. For maximum performance, the following relevant Cray modules were loaded: cray-hugepages16M, craype, craype-network-aries, and craype-broadwell.

[12]Only expansion by multiples of BlockSize will be covered in this work.

[13]QSBRArray does not make use of checkpoints and represents the best-case.

[14]Forked after SHA 60bb637fa16772400ee702be4374427154080345



(a) Random Indexing (1024) (b) Sequential Indexing (1024)

(c) Random Indexing (1M) (d) Sequential Indexing (1M)

Fig. 2: Random and Sequential Access

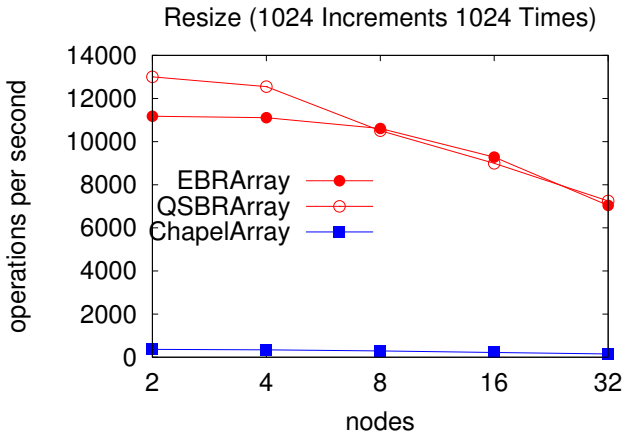
A. Indexing & Resizing

For the first and second benchmarks, ChapelArray, QSBRArray, EBRArray, and SyncArray all perform 1024 update operations per task, with 44 tasks per locale, on randomized and sequential indices of the array, shown in Figure 2a and Figure 2b respectively. These benchmarks choose a smaller number of operations to allow for SyncArray to finish within a reasonable amount of time. As expected, SyncArray is the slowest of all where not only does it not scale due to mutual exclusion, but also degrades in performance due to the increasing number of remote tasks that must contest for the same lock. QSBRArray offers competitive performance to the unsynchronized ChapelArray, slightly losing for random-access patterns but offers near-equivalent performance for more predictable access patterns. EBRArray proves to scale relatively well but only offers approximately 40% of the performance of ChapelArray and QSBRArray.

For the third and fourth benchmarks, ChapelArray, QSBRArray, and EBRArray[15] all perform 1M update operations per task, with 44 tasks per locale, on randomized and sequential indices of the array, shown in Figure 2c and Figure 2d respectively. Unlike the former benchmarks, a larger number of operations can be performed to obtain more precise and accurate data. QSBRArray loses slightly to ChapelArray under random-access patterns like before but exceeds ChapelArray in performance when it comes to sequential-access patterns by approximately 1.5x, likely due to the simplicity in design. EBRArray this time offers less than 20% of the performance of ChapelArray and QSBRArray.

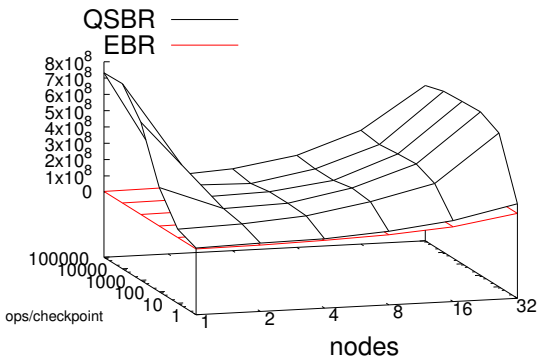
For the fifth benchmark, ChapelArray, EBRArray, and QSBRArray perform a total of 1024 resize operations in increments of 1024, starting with zero-capacity and increasing to a total capacity of 1M, shown in Figure 3. ChapelArray prove to perform the slowest, with QSBRArray and EBRArray offering

[15]SyncArray is excluded due to required runtime



(a) Resizing to 1M in increments of 1024

Fig. 3: Resizing to 1M in 1024 increments



(a) QSBR Checkpoint Overhead

Fig. 4: Overhead of checkpoints

near-equivalent performance, exceeding ChapelArray by over 40x. Inherent in RCUArray's novel design, both QSBRArray and EBRArray can avoid the extra work required to deep-copy blocks of memory from one smaller storage into a larger storage, avoiding the risk of cache pollution.

B. QSBR Checkpoints

Checkpoints, invoked via QSBR_Checkpoint, and their strategic placement are crucial for not only correctness but overall performance. To demonstrate the latter, a benchmark is prepared that invokes a checkpoint after a fixed number of RCUArray update operations, shown in Figure 4. In the benchmark, we spawn 44 tasks per locale that each perform 1M operations with checkpoints invoked after a certain number of operations. Contrary to other benchmarks, the performance at one locale is shown as QSBR is more general-purpose and is suitable for use for single and multiple locale applications. The performance gathered from previous benchmarks for EBRArray in Figure 2d are reused here and inserted as a baseline of performance. As shown, QSBRArray and QSBR in general exceeds the performance of the extension of the EBR algorithm presented in this work, even in cases where a checkpoint is invoked after each operation. This is likely due to the contention and sequential consistency memory ordering of the Fetch-And-Add and Fetch-And-Sub atomic operations on the EpochReaders counters. Checkpoints can have very little overhead by themselves, but when called with enough frequency can become a bottleneck. Careful profiling is required for determining the appropriate frequency; if too few checkpoints are used, memory consumption may become an issue; if too many checkpoints are used, performance may become an issue.

VI. Conclusions and Future Work

Presented in this work is the RCUArray, a parallel-safe distributed array that allows concurrent read and update operations while being resized. Also presented is an extension to Epoch-Based Reclamation that does not rely on thread-local or task-local storage and provides a guarantee on a constant space overhead. Also presented is an extension to Quiescent State-Based Reclamation that is introduced into Chapel's runtime that makes use of thread-local metadata, epochs, and checkpoints to determine the safe reclamation of arbitrary data. The RCUArray allocates memory in blocks of a predetermined size that can be distributed across multiple nodes, enabling the recycling of memory. RCUArray relaxes RCU reads to return by reference to allow for updates, and uses the indirection of using blocks of memory to allow for proper privatization of data and to ensure visibility of updates across different nodes and snapshots. The RCUArray under EBR suffers from the lack of thread-local and task-local storage and as such can offer as little as 20% of the read and update performance of an unsynchronized Chapel block distributed array, but under QSBR it can offer near-equivalent or slightly superior performance; RCUArray under both memory reclamation algorithms can offer as much as 40x performance for resizing.

While the EBR algorithm demonstrated in this work is slower than the QSBR algorithm, it may work independent of changes to the runtime and establishes correctness even under integer overflow. In future work, the decoupling of EBR from RCUArray can be performed easily, and future improvements to the decoupled EBR algorithm are planned and can even be used in other languages that lack official support for TLS, such as Golang. In the meantime, RCUArray can serve as the ideal backbone for a random-access data structure such as a distributed vector or table which both benefit from the ability to be resized and indexed with parallel-safety. The official integration of the QSBR algorithm into the Chapel project is nearing completion and planned for Chapel release 1.18. Lastly, compatibility of RCUArray and Chapel's Domain map Standard Interface is being explored with hopes to provide users with a parallel-safe resizable distribution.

ACKNOWLEDGEMENTS

The paper nor the research would have been possible without the encouragement of Chapel's development team members, Michael Ferguson and Brad Chamberlain. Special thanks to Cray, as benchmark results could not have been gathered without access to the Cray XC-50 supercomputer. Special thanks to William Calhoun of Bloomsburg University and another anonymous individual for proofreading.

REFERENCES

[1] M. Arbel and A. Morrison. Predicate rcu: an rcu for scalable concurrent updates. In *ACM SIGPLAN Notices*, volume 50, pages 21–30. ACM, 2015.

[2] A. T. Clements, M. F. Kaashoek, and N. Zeldovich. Scalable address spaces using rcu balanced trees. *ACM SIGPLAN Notices*, 47(4):199–210, 2012.

[3] D. Dechev, P. Pirkelbauer, and B. Stroustrup. Lock-free dynamically resizable arrays. In *International Conference On Principles Of Distributed Systems*, pages 142–156. Springer, 2006.

[4] M. Desnoyers, P. E. McKenney, A. S. Stern, M. R. Dagenais, and J. Walpole. User-level implementations of read-copy update. *IEEE Transaction on Parallel and Distributed Systems*, pages 375–382, 2012.

[5] K. Fraser. Practical lock freedom. *Technical Report UCAM-CL-TR-579*, 2004.

[6] K. Fraser and T. Harris. Concurrent programming without locks. *ACM Transactions on Computer Systems*, 2007.

[7] T. E. Hart. *Comparative Performance of Memory Reclamation Strategies for Lock-free and Concurrently-readable Data Structures*. PhD thesis, University of Toronto, 2005.

[8] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[9] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, pages 463–492, 1990.

[10] Z. Liu, J. Chen, and Z. Shen. *Read-Copy Update and Its Use in Linux Kernel*. New York University, 2011.

[11] A. Matveev, N. Shavit, P. Felber, and P. Marlier. Read-log-update: a lightweight synchronization mechanism for concurrent programming. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 168–183. ACM, 2015.

[12] P. E. McKenney. Rcu vs. locking performance on different cpus. In *linux. conf. au*, 2004.

[13] P. E. McKenney and J. Satran. Cluster-wide read-copy update system and method, Feb. 28 2012. US Patent 8,126,843.

[14] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, pages 491–504, 2004.

[15] J. D. S. Paul E. McKenney. Read-copy update: Using execution history to solve concurrent problems. *Parallel and Distributed Computing and Systems*, pages 509–518, 1998.

[16] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[17] J. Triplett, P. E. McKenney, and J. Walpole. Resizable, scalable, concurrent hash tables via relativistic programming. In *USENIX Annual Technical Conference*, page 11, 2011.