

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 [3], [5]. 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 [8] 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) [7] and in particular the userspace variant [1] is a more recent type of synchronization strategy which allows parallel-safe reads during a write. 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) [2] 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) [9] in which readers must periodically

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
  
```

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 [4]. 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 performance 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. 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 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 privatized¹ 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

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

Algorithm 1: RCU Pseudocode

```

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

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

```

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 which is our actual data, 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.

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 [6] 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 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: 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: 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 monoton-

²An immutable version of data.

³The WriteLock should be acquired prior to invoking RCU_Write.

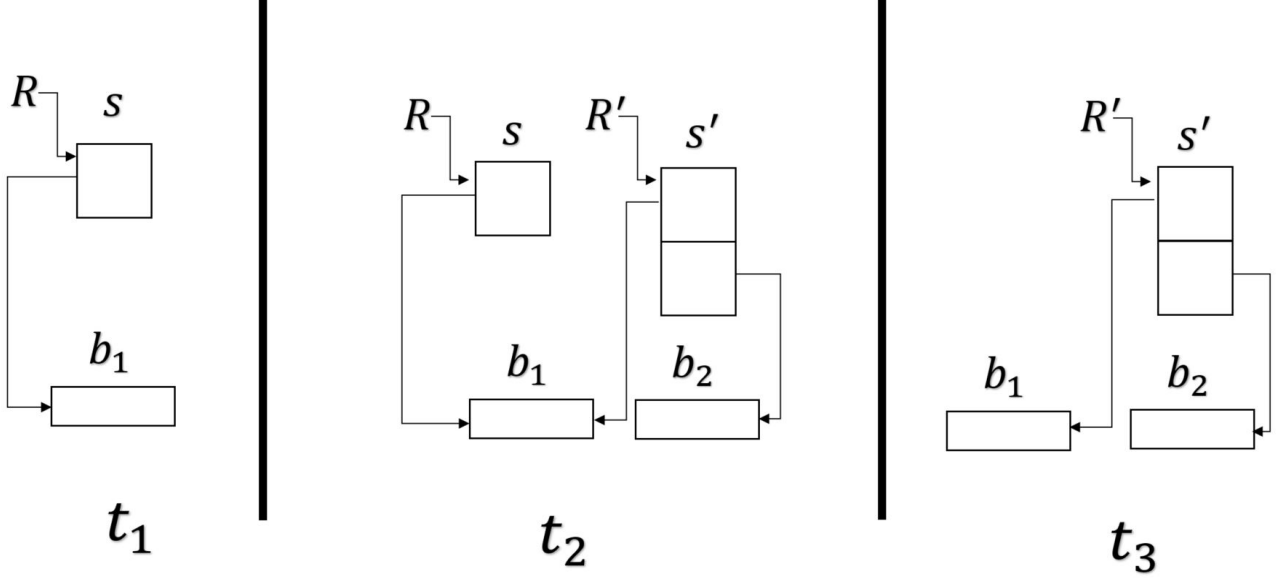


Fig. 1: Parallel-safe resizing of RCUArray at 3 distinct points in time: t_1 which is prior to a write where a reader R is accessing the current snapshot s ; t_2 which is after a concurrent writer updates the current GlobalSnapshot to s' , the clone of s , and when a new reader R' accesses s' ; and t_3 which is after the write has safely reclaimed s once R had finished its operation.

ically 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 $B = (b_1, b_2, \dots, b_N)$ where $\forall b \in B, b \in \{0, 1\}$ and where b_1 is the least significant bit. If we have $e = (1, 1, \dots, 1)$ being the largest possible value, and $e' = e + 1 = (0, 0, \dots, 0)$ 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: 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' .

It is clear 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 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 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-specific⁴ 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

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

Algorithm 2: QSBR Pseudocode

```

1 // Defers memory reclamation of objs until safe
2 proc QSBRR_Defer (objs)
3   | tls ← getTLS();
4   | // Update and observe the new global state.
5   | tls.ObservedEpoch ← StateEpoch.fetchAdd(1) + 1;
6   | tls.DeferList.push(objs, tls.ObservedEpoch);
7
8 // Handle memory reclamation for DeferList if eligible
9 proc QSBRR_Checkpoint ()
10  | tls ← getTLS();
11  | // Observe the current state.
12  | tls.ObservedEpoch ← StateEpoch.read();
13  | // Find smallest (safest) epoch
14  | minEpoch ← tls.ObservedEpoch;
15  | for tls' in TLSSLList
16  |   | minEpoch ← min(tls'.ObservedEpoch, minEpoch);
17  | // Split DeferList where the safe epoch  $\leq$  minimum epoch.
18  | head ← tls.DeferList.popLessEqual(minEpoch);
19  | while head  $\neq$  nil
20  |   | tmp ← head;
21  |   | head ← head.next;
22  |   | delete tmp;

```

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: Given that s is a snapshot with N blocks represented by the sequence (b_1, b_2, \dots, b_N) , cloning s to create a larger snapshot s' with M blocks can be represented as the sequence $(b_1, b_2, \dots, b_N, b_{N+1}, \dots, b_M)$; 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 propagated 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

III. 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 (line 1 - 3).

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

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

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

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
  17 proc Helper (snapshot)
  18   snapshot.blocks.push_back(newBlocks);
  // Update performed on each node
  19 coforall loc in Locales do on loc
  20   pThis ← chpl_getPrivatizedCopy(PID);
  21   if isQSBR then
  22     // Handle RCU directly with QSBR...
  23     oldSnapshot ← pThis.GlobalSnapshot;
  24     newSnapshot ← clone(oldSnapshot);
  25     Helper(newSnapshot);
  26     pThis.GlobalSnapshot ← newSnapshot;
  27     QSBR_Defer(oldSnapshot);
  28   else
  29     pThis.RCU_Update(Helper);
  30   pThis.NextLocaleId ← locId;
  WriteLock.release();

```

After obtaining the privatized copy via the runtime function `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 (line 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

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

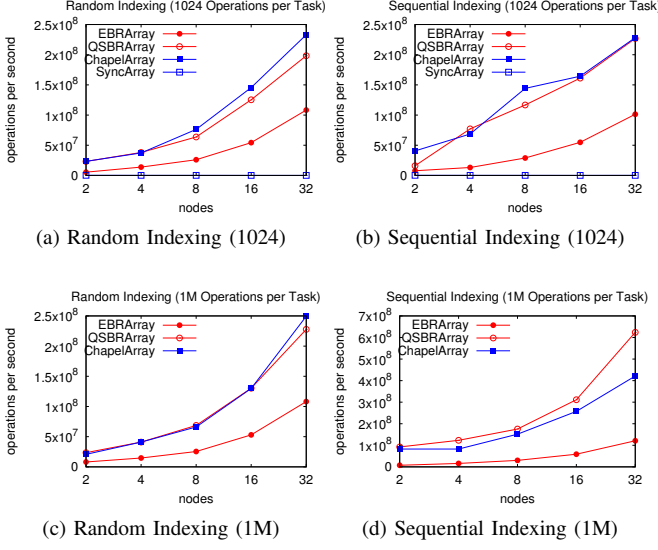


Fig. 2: Random and Sequential Access

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 $QSBRArarray$ (line 21 - 25). If not configured for QSBRArarray, 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).

IV. PERFORMANCE EVALUATION

Extensions to QSBRArarray (QSBRArarray)¹³ and EBR (EBRArarray) were tested against each other, and when appropriate against a naive block distributed array using Chapel's standard BlockDist distribution (ChapelArray). Compared are the performance of read, update, resizing, and mixed operations. While ChapelArray allows for concurrent read and update operations, a variant is defined that uses mutual exclusion to allow for safe resize operations via sync variables (SyncArray). All benchmarks were performed on a Cray-XC50 cluster running Intel Xeon Broadwell 44-core processors. 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 proprietary modules were loaded: cray-hugepages16M, craype, craype-network-aries, and craype-broadwell.

A. Indexing & Resizing

For the first and second benchmarks, ChapelArray, QSBRArarray, EBRArray, and SyncArray all perform 1024 update

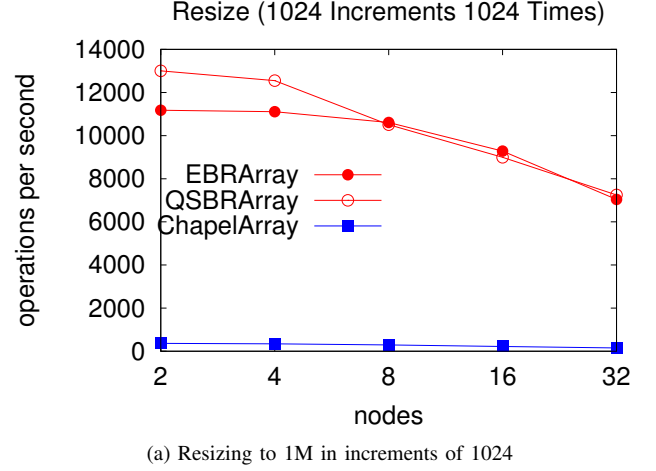


Fig. 3: Resizing to 1M in 1024 increments

operations per task, with 44 tasks per locale, on randomized and sequential indices of the array, shown in Figure 2a and Figure 2b respectively. This benchmark chooses 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. QSBRArarray 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 QSBRArarray.

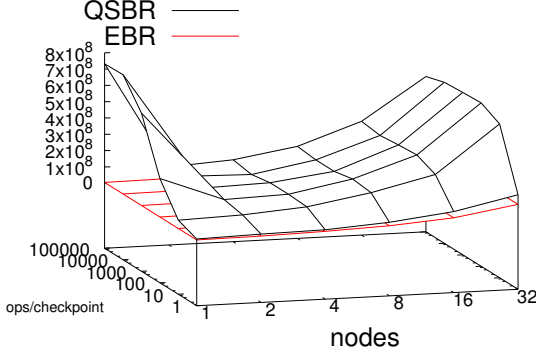
For the third and fourth benchmarks, ChapelArray, QSBRArarray, and EBRArray¹⁵ all perform 1M update operations per task, with 44 tasks per locale, on randomized and sequential indices of the array, shown in Figure 2c and Figure 2d respectively. Unlike the former benchmarks, a larger number of operations can be performed to obtain more precise and accurate data. QSBRArarray 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 QSBRArarray.

For the fifth benchmark, ChapelArray, EBRArray, and QSBRArarray 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 QSBRArarray and EBRArray offering near-equivalent performance, exceeding ChapelArray by over 40x. Inherent in RCUArray's novel design, both QSBRArarray 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.

¹³QSBRArarray does not make use of checkpoints and represents the best-case.

¹⁴Forked after SHA 60bb637fa16772400e702be4374427154080345

¹⁵SyncArray is excluded due to required runtime



(a) QSBR Checkpoint Overhead

Fig. 4: Overhead of checkpoints

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.

V. 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.

VI. 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. 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.
- [2] K. Fraser. Practical lock freedom. *Technical Report UCAM-CL-TR-579*, 2004.
- [3] K. Fraser and T. Harris. Concurrent programming without locks. *ACM Transactions on Computer Systems*, 2007.
- [4] T. E. Hart. *Comparative Performance of Memory Reclamation Strategies for Lock-free and Concurrently-readable Data Structures*. PhD thesis, University of Toronto, 2005.
- [5] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.
- [6] 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.
- [7] Z. Liu, J. Chen, and Z. Shen. *Read-Copy Update and Its Use in Linux Kernel*. New York University, 2011.
- [8] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, pages 491–504, 2004.
- [9] 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.