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

Louis Jenkins
Bloomsburg University
lpj11535@huskies.bloomu.edu

***Abstract*—I present RCUArray, a parallel-safe distributed array that allows for read and update operations to occur concurrently with a resize. As Chapel lacks thread-local and task-local storage, I also present a novel extension to the Read-Copy-Update synchronization strategy that functions without the need for either. At 32-nodes with 44-cores per node the RCUArray's relative performance to an unsynchronized Chapel block distributed array is as little as 20% for read and update operations, but with runtime support for zero-overhead RCU and thread-local or task-local storage it has the potential to be near-equivalent; relative performance for resize operations is as much as 3600% due to the novel design.**

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 is infeasible as it inhibits scalability and introduces problems such as deadlock, priority inversion, and convoying [3], [6]. Reader-writer locks are 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 as caution must be used in the reclamation of memory that may be accessed concurrently in a language without garbage collection. Mechanisms such as Pass The Buck [5] and Hazard Pointers [9] can provide a safe non-blocking approach for memory reclamation, but impose a noticeable overhead that is inadequate when the importance in the performance of reads far outweigh the performance for writes.

Read-Copy-Update (RCU) [8] and in particular the userspace variant [1] is a more recent type of synchronization strategy which allows parallel-safe reads during a write. It does not come without drawbacks, as writers must perform the task of memory reclamation by waiting for all readers to *evacuate*[1]. RCU can come in two flavors: *Epoch-Based Reclamation* (EBR) [2] which require readers to enter *read-side critical sections* to safely access the protected data, and *Quiescent State-Based Reclamation* (QSBR) [10] where readers must explicitly notify writers at checkpoints that they are no longer accessing 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 variants of RCU require the usage of thread-local or task-local storage, of which Chapel lacks any notion of.

In this work, I present RCUArray, a distributed array that may be used in place of Chapel's arrays and distributions that allow for a crucial feature they lack: parallel-safe resizing. I also present a novel extension to RCU based on EBR that functions without the need for either thread-local or task-local storage. RCUArray, while scalable, comes with a significant cost to read and update performance under this new extension, but yields significant improvements in resize performance due to its design and is an adequate base for future works.

II. Design

The array is simple in design but overcomes three core challenges: (1) parallel-safe memory reclamation; (2) allowing concurrent read and write operations even while the data structure is in the process of being resized; and (3) ensuring that the data structure can be properly maintained and distributed across multiple nodes in a cluster.

Listing 1 displays the two data types used in the array: RCUArrayMetaData and RCUArraySnapshot, both of which are privatized[2] and *node-local*[3]. The RCUArrayMetaData maintains information required for privatization such as the privatization id, PID, and node-local RCU related data that will be explained shortly. The WriteLock is simply a reference to some global lock that all nodes share. The RCUArraySnapshot is our actual data, equivalent to an array of Blocks, where each block is simply another array of data with a predetermined capacity.

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. Each node maintains the current *epoch*[4],

[1] To no longer have access to.

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

[3] Abuse of privatization to update the privatized copy that are only visible to the node it is allocated on.

[4] Version number.

Listing 1: Data Structure Types

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

Algorithm 1: RCU Pseudocode for Readers

```
// Applies a function λ to protected data with a result
proc RCU_Read (λ)
1   while true do
        // Attempt to record our read
2       epoch ← GlobalEpoch;
3       readIdx ← epoch % 2;
4       EpochReaders[readIdx].add(1);
        // Did snapshot possibly changed before we recorded?
5       if epoch = GlobalEpoch then
            // Safe to apply user function
6           retval ← λ(GlobalSnapshot);
7           EpochReaders[readIdx].sub(1);
8           return retval;
        // Try again
9       EpochReaders[readIdx].sub(1);
```

Algorithm 2: RCU Pseudocode for Writers

```
// Applies a side-effect inducing function λ to protected data
proc RCU_Update (λ)
1   newSnapshot ← clone(GlobalSnapshot);
    // Update performed on clone, clone becomes new snapshot
2   λ(newSnapshot);
3   GlobalSnapshot ← newSnapshot;
    // Reminder: We already have mutual exclusion
4   epoch ← GlobalEpoch;
5   GlobalEpoch ← epoch + 1;
    // Wait for readers...
6   readIdx ← epoch % 2;
7   waitForReaders(readIdx);
    // Safe to delete...
8   delete(old);
```

GlobalEpoch, which corresponds to the current *snapshot*[5], GlobalSnapshot. To make up for the lack of thread-local or task-local storage, readers must compensate by *recording* their reads so that a concurrent writer does not free the snapshot they are using. Recording is performed using a set of two atomic counters, called EpochReaders, which exploits the parity of the current epoch to determine which counter to use, demonstrated in Algorithm 1.

A single writer waits for all recorded readers to evacuate after updating the GlobalSnapshot and GlobalEpoch *in that order*, and promptly frees the older snapshot ensuring that there can be at most two snapshots alive at any given time. To ensure that there can only be a single writer, mutual exclusion with respect to other potential writers is required. Algorithm 2 demonstrates the algorithm that writers must follow.

As readers may run concurrently with a single writer, we must ensure that there is a point of linearizability [7] where recorded reads are appropriately seen by a concurrent writer, ergo safe. We can see that given a scenario where we have a reader, *R*, adhering to Algorithm 1 running concurrently with a writer, *W*, adhering to Algorithm 2 that *R* must first observe the current GlobalEpoch, but it is possible for *W* to update the GlobalEpoch and to finish waiting for readers before *R* can finish recording its read. In this case, *R* will see the new snapshot set by *W* when it returns even if *R*'s read is not properly recorded, but it is possible for another writer, *W'*, to complete before *R* resulting in *W'* freeing the memory while *R* is using it. To remedy this we must perform a check after attempting to record our read. If *R* determines that its observed epoch is outdated, we know that it is possible that *W* will not see our recorded write; if *R* determines that its observed epoch is not outdated, then we know that *W* has not yet started waiting for readers and will see our recorded read.

B. Concurrent Updates and Resizing

It would be prohibitively expensive if something as simple as an update[6] had the same overhead as a resize. To allow for updates to attain performance equivalent to a read, we simply extend Algorithm 1 to return by reference. This solution does not come without downsides, as updates become non-linearizable. If some updater, *U*, adhering to the extension of Algorithm 1 runs concurrently with a writer, *W*, adhering to Algorithm 2 then we can see that if *U* has successfully obtained its reference, while it is guaranteed to be safe, it is possible that *W* has cloned the GlobalSnapshot before *U* has completed its update. This results in the loss of *U*'s update, which will be unseen during the application of the λ function and to all future readers, updaters, and writers once the incorrect snapshot has been globally committed by *W*.

To prevent the loss of these updates, multiple blocks of memory of some predetermined size are used in place of a single contiguous block. 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, but it allows updates to share the same performance as reads. Furthermore, recycling blocks of memory proves to be significantly faster than copying by value into a larger memory.

C. Distribution

The array is distributed simply by distributing the allocation of the blocks in a way that resembles a block-cyclic distribution, where we round-robin the node on which we allocate each block. As both RCUArrayMetaData and RCUArraySnapshot are privatized data structures with only the WriteLock and the blocks themselves being distributed, readers and updaters operate oblivious to distribution and communication. To allow this, writes must be replicated across all nodes with respect to their own node-local data[7].

[5] An immutable version of data.

[6] An update is classified as an assignment to some indexable portion of the array.

[7] A writer must *for each node* update the GlobalEpoch, GlobalSnapshot, wait for the appropriate EpochReaders, and then delete their older snapshot.

Algorithm 3: Index 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      return pThis.RCU_Reader(Helper);
```

Algorithm 4: Resize Pseudocode (Expand Only)

```
   // Expands the size of the array
   proc Resize (size)
1      newBlocks : [1..0]Block;
2      GlobalLock.acquire();
3      locId ← NextLocaleId;
       // Allocate and distribute new blocks
4      while size > 0
           on Locales[locId] do
5              newBlocks.push_back(newBlock());
6          locId ← (locId + 1) % numLocales;
7          size ← size / BlockSize;
       // Function to append blocks to snapshot
       proc Helper (snapshot)
8          snapshot.blocks.push_back(newBlocks);
       // Update performed on each node
9      coforall loc in Locales do on loc
10         pThis ← chpl_getPrivatizedCopy(PID);
11         pThis.RCU_Update(Helper);
12         pThis.NextLocaleId ← locId;
13     releaseGlobalLock();
```

III. Implementation

In this section I describe the algorithm for parallel-safe indexing and resizing by utilizing the RCU abstractions portrayed in Algorithm 1 and Algorithm 2.

A. Indexing

Both read and update operations can be performed through the reference returned via indexing as seen in Algorithm 3.[8] We utilize a *Helper* which is shown as a nested procedure that allows accessing local variables within the same scope. The Helper translates the idx and returns a reference to the part of the block that it corresponds to. We use the runtime function chpl_getPrivatizedCopy to obtain the privatized copy to apply the RCU read operation. [9]

B. Resizing

As seen in Algorithm 4, this operation is mutually exclusive with respect to not only the node but the entire cluster. We allocate an empty array using Chapel's array notation and begin filling it with round-robin distributed blocks[10]. Our Helper function also has access to local variables within the same scope and will handle pushing back the allocated blocks to each snapshot[11]. We then replicate this update across all locales, ensuring that we also invoke the RCU update on the returned privatized copy. Once all remote tasks have finished, we can safely give up our lock, completing the operation.

IV. Performance Evaluation

A core feature of RCUArray is the ability to perform concurrent-safe reads and updates and so the benchmarks provided will solely test for scalability and comparative performance to Chapel's built-in arrays and distributions. All benchmarks were performed on a Cray-XC50 cluster running Intel Xeon Broadwell 44-core processors per node, optimized and compiled under Chapel 1.17 pre-release using the QThread tasking layer. For maximum performance, the following relevant proprietary modules were loaded for maximized performance: cray-mpich, cray-hugepages16M, craype, craype-network-aries, craype-broadwell, and cce.

All benchmarks will only run in a distributed context where there is more than a single node. For our benchmarks we test the performance of random and sequential access as well as resizing for both RCUArray and Chapel's array using a Block distribution, ChapelArray. As RCUArray uses synchronization of some kind we add a variant of each: ChapelArray_Sync[12] which is a naive synchronized array that requires the user to acquire and release a lock before and after accessing, and RCUArray_NoBarrier which is RCUArray without read-side critical sections to resemble performance under the zero-overhead QSBR.

A. Indexing

We test RCUArray, RCUArray_NoBarrier, ChapelArray, and ChapelArray_Sync in a benchmark that performs both sequential and random access 1024 times per task per node. As can be seen in Figure 1 the RCUArray_NoBarrier offers competitive performance. RCUArray suffers from the performance loss of read-side critical sections and the lack of thread-local or task-local storage, but it does demonstrate scalability. ChapelArray_Sync does not scale at all due to mutual exclusion.

Next we test RCUArray, RCUArray_NoBarrier, and ChapelArray again and raise the number of operations per task per node to one million.[13] This time we adjust the units from operations per second to overall run-time as seen in Figure 2. Both RCUArray_NoBarrier and ChapelArray both offer nearly equivalent performance with RCUArray offering only 20% of the performance due to the read-side critical sections.

B. Resizing

Finally I present a benchmark between RCUArray and ChapelArray for resizing the array itself, which begins with a zero-capacity array and resizes in increments of 1024 until reaching a capacity of 1M elements. The benchmark tests the relative overhead of writes in a write-heavy workload

[8]For brevity we ignore checks for indexes that are out-of-bounds.

[9]We explicitly use the returned privatized copy in the case that the method is invoked on a remote instance.

[10]For simplicity, we only handle expansion with a size that is some multiple of the BlockSize.

[11]Reminder: This is a clone of the snapshot.

[12]Note that Chapel only offers sync variables, which are allocated on a single node and result in excessive communication on top of total mutual exclusion.

[13]We exclude ChapelArray_Sync as it is unable to complete within a reasonable time frame.

against the advantages of being able to recycle memory, seen in Figure 3. As can be observed, RCUArray's novel design allows it to avoid the redundant work of allocating an even larger storage and performing a cache-polluting memcpy into it. RCUArray offers a significant resize performance improvement over ChapelArray at just over 3,600% at 32 nodes.

V. Conclusions and Future Work

In this paper I introduced the RCUArray, a parallel-safe distributed array that allows concurrent read and update operations while being resized. Also introduced is an extension to Epoch-Based Reclamation that does not rely on thread-local or task-local storage which is currently used by RCUArray. 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 suffers from the lack of thread-local and task-local storage and as such currently offers merely 20% the read and update performance of an unsynchronized Chapel block distributed array, but as much as 3600% for resizing. Experiments show that RCUArray with read barriers disabled, which resembles the zero-overhead Quiescent State-Based Reclamation, can offer nearly equivalent read and update performance. The RCUArray can serve as the ideal backbone for a random-access data structures such as a distributed vector which requires the ability to be resized and indexed and would benefit from the parallel-safety.

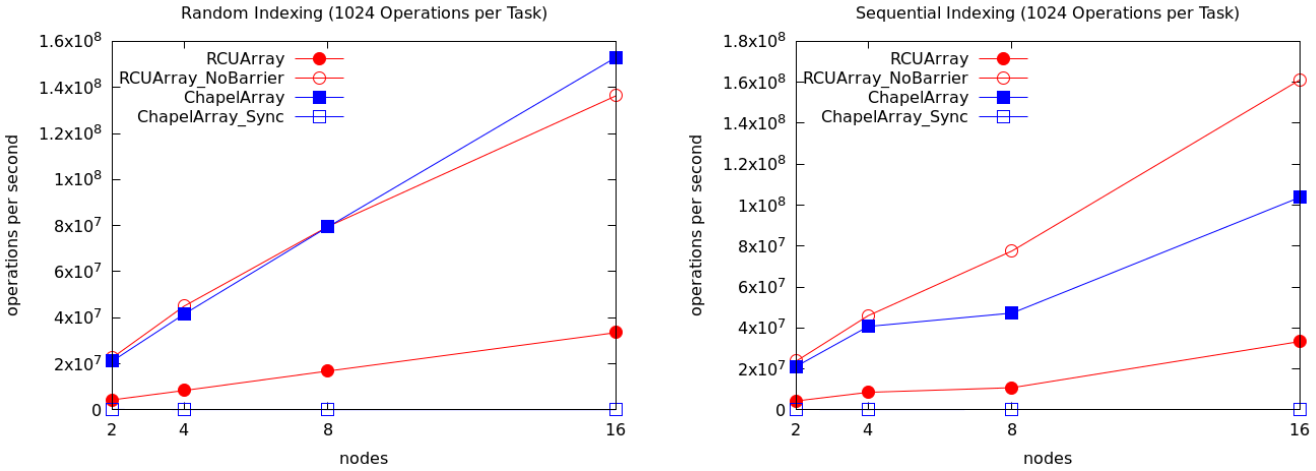

Fig. 1: Random and Sequential Access, 1024 per Task per Node

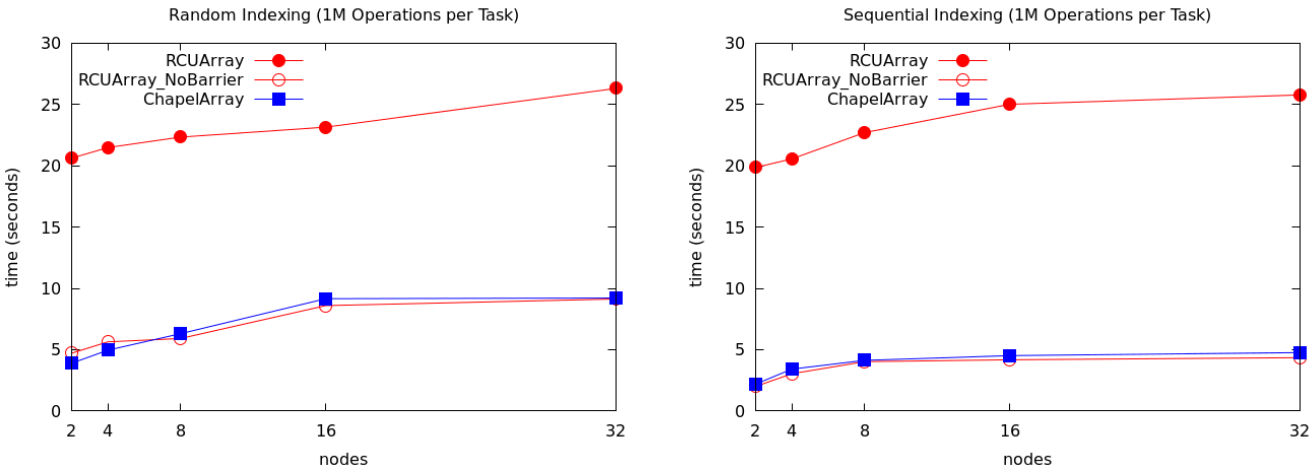

Fig. 2: Random and Sequential Access, 1M per Task per Node

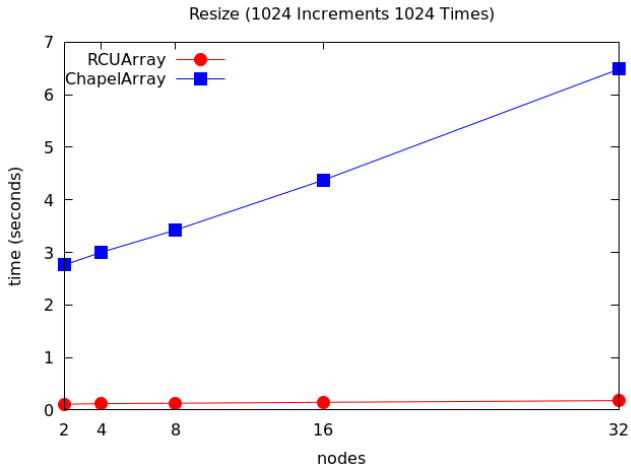

Fig. 3: Resizing to 1M in 1024 increments

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, V. Luchangco, P. Martin, and M. Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Transactions on Computer Systems*, pages 146 – 196, 2005.

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

[7] 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.

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

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

[10] 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.