Paving the way for Distributed Non-Blocking Algorithms and Data Structures in the Partitioned Global Address Space model

Garvit Dewan, Indian Institute of Technology Roorkee, gdewan@cs.iitr.ac.in
Louis Jenkins, University of Rochester, louis.jenkins@rochester.edu

Abstract— **The partitioned global address space memory model has bridged the gap between shared and distributed memory, and with this bridge comes the ability to adapt shared memory concepts, such as non-blocking programming, to distributed systems such as supercomputers. To enable non-blocking algorithms, we present ways to perform scalable atomic operations on objects in remote memory via remote direct memory access and pointer compression. As a solution to the problem of concurrent-safe reclamation of memory in a distributed system, we adapt Epoch-Based Memory Reclamation to distributed memory and implement it such that it supports global-view programming. This construct is designed and implemented for the Chapel programming language but can be adapted and generalized to work on other languages and libraries.**

I. INTRODUCTION

In synchronized data structures and algorithms, there are many pitfalls that programmers can fall into, such as deadlock, livelock, and priority inversion [1]. Non-blocking data structures and algorithms provide many benefits over their synchronized counterparts, such as providing guarantees on liveness, which is a property on how threads make progress throughout a system. One such liveness property is *obstruction-freedom* [2], which states that threads are guaranteed to complete their operation so long as they are not obstructed by some other thread, such as by executing in isolation; in *lock-freedom* [3], at least one thread is guaranteed to progress and succeed in a bounded number of steps; in *wait-freedom* [4], all threads are guaranteed to progress and succeed in a bounded number of steps. These properties are attainable in shared-memory, with decades of research available on non-blocking data structures for shared-memory. However, to the authors' knowledge, there are not many, if at all, for distributed memory.

The *Partitioned Global Address Space (PGAS)* memory model offers an abstraction of distributed memory systems in a way that allows them to have very similar semantics to that of shared-memory. For example, PUTs and GETs, which are *remote-direct memory access (RDMA)* operations that remotely write and read values in memory without the intervention of the CPU, are analogous to shared-memory load and store operations, and can even be modeled as such [5]. RDMA atomic operations, which are entirely handled by the network interface controller (NIC) in high-performance computing networks such as InfiniBand and Gemini/Aries, allow for extremely low latency atomic operations that are in the ballpark of mere microseconds. The exploration of the application of non-blocking algorithms and data structures to PGAS is enticing, but there are roadblocks and hurdles that must be dealt with first. For example, the Chapel programming language lacks native support for atomics on arbitrary objects such as class instances, which is necessary for any non-blocking algorithm. As well, the issue of concurrent-safe memory-reclamation, that is, the reclamation of memory when arbitrarily many threads could be accessing said memory at any given time, is a real problem in shared-memory where multiple solutions exist [6], [7], [8], [9].

This work provides `EpochManager` and `LocalEpochManager`, which are based on Epoch-Based Reclamation (EBR) [10]. Both serve as pseudo garbage collection mechanisms that scale not only in shared-memory but in distributed memory as well. Also provided is `AtomicObject`, and the local-optimized variant `LocalAtomicObject`. Both provide atomic operations on class instances and provide optional ABA-protection. The former is designed to support RDMA atomics on class instances, making it possible for some truly scalable algorithms and data structures.

This work was supported by the US Department of Energy Computational Science Graduate Fellowship (grant DE-SC0020347). Special thanks to Cray, a Hewlett-Packard Enterprise Company, for providing access to the compute resources utilized during development.

II. DESIGN & IMPLEMENTATION

In the development of the `EpochManager`, there were prerequisites that needed to be addressed, such as the need for atomic operations on class instances. Only after overcoming these hurdles is it possible to create the infrastructure and building blocks necessary for creating non-blocking algorithms in both shared and distributed memory.

A. Atomic Objects

In Chapel, *atomic* operations, which are operations that appear to take place all at once from any other task's point of view, are defined only on *bool*, *int*, *uint* and *real* primitive types. As of today, there is no official support for atomic operations on class instances, which are represented as *widened* pointers that contain not only the 64-bit virtual address but 64 bits of locality information, comprising a 128-bit structure. Chapel has not implemented support for atomics on class instances due to portability challenges, creating significant obstacles to create even the most primitive of non-blocking data structures, such as queues, stacks, and linked lists. Furthermore, in Chapel, atomic operations over the network rely upon *Remote Direct Memory Access (RDMA)*, which currently only supports atomic operations up to 64-bit.

In the initial prototype, which has been adapted into its independent module, called the `LocalAtomicObject`, the locality information is ignored, and it maintains an atomic holding only the 64-bit virtual address. As `LocalAtomicObject` will only work in a shared-memory context, the `GlobalAtomicObject` offers *pointer compression*, which is designed to take advantage of the fact that currently, processors only use the lowest 48-bits for the virtual address, enabling the encoding of 16-bits of locality information in the 64-bit pointers. This approach will only work in distributed setups with fewer than 2^{16} compute nodes, which consequently enables RDMA atomics on Cray Aries.[1] In the event that more than 2^{16} compute nodes are used, the implementation will fall back to using x86 *CMPXCHG16B* instruction[2], also known as the 'Double-word-Compare-and-Swap' (DCAS) operation, which can atomically update both the virtual address as well as the 64-bits of locality information. Unfortunately, this demotes atomic operations on remote memory from RDMA atomics, which take around a microsecond to complete and do not require the intervention of the CPU, to using active messages, which are entirely handled by the progress thread of the recipient compute node. As `shared` type is already wrapped in a record and is larger than 64-bits, `owned` type is statically managed and cannot be tracked without significant rework to the type, and `borrowed` types are explicitly tracked by the compiler making it difficult to track without some significant rework, support is currently restricted to `unmanaged` class instances. Support for `owned` and `borrowed` types is planned as a future work.

Another problem that had to be overcome was the *ABA* problem. The ABA problem occurs in scenarios where one has at least two threads, and typically arises when performing a compare-and-swap operation. In one such scenario, consider an atomic linked list where one has multiple threads, where a thread τ_1 reads from the head of the list and receives the node with virtual address α. Imagine that τ_1 gets preempted, and some thread τ_2 also reads the head of the list, atomically moves the head of the list forward, and deletes the node such that α is put back on some free-list. Later, some other thread τ_3 allocates a new node which happens to have the same address α and atomically inserts this at the head of the list. Now, τ_1 wakes up and incorrectly succeeds in its atomic exchange, despite the fact that the head of the list has changed. There are two known ways to solve the ABA problem, and they are to either use a concurrent memory reclamation system, in which is currently being built and leads to a chicken-and-egg paradox, and using a DCAS, where a 64-bit counter is held adjacent to the 64-bit word being atomically updated. In the DCAS approach, the counter gets incremented after each ABA-dependent operation, which causes a DCAS to fail even in the event of the ABA problem, since the 64-bit counter will have changed. The `AtomicObject` and `LocalAtomicObject` provide a 128-bit wrapper for 64-bit types, called `ABA` where such a 64-bit counter is held adjacent to the 64-bit virtual address, which in conjunction with pointer compression can provide both ABA-free atomic operations on remote objects, albeit using remote execution rather than RDMA. Each operation has an ABA variant, which includes the suffix 'ABA', that will take into account the 64-bit counter, but the advanced user is free to use both ABA and normal variants interchangeably. Due to Chapel's `forwarding` decorator, it is possible to use the `ABA` in a seamless manner as if it were the type it is wrapping, as all methods and field accesses will forward to the underlying instance. Example usage of `AtomicObject`, implementing a `push` operation of the Trieber Stack [11], can be seen in Listing 1.

```
proc LockFreeStack.push(newObj : T) {
    var node = new unmanaged Node(newObj);
    do {
      var oldHead = head.readABA();
      node.next = oldHead.getObject();
    } while(!head.compareAndSwapABA(oldHead, node));
}
```

Listing 1: Example usage of AtomicObject

B. Epoch Based Reclamation (EBR)

It was essential to make the `EpochManager` non-blocking so to not weaken the non-blocking guarantees of the data structures that employ it, or at least not too much. The three non-blocking guarantees from weakest to strongest are as follows: Obstruction-Freedom, which ensures that if a thread runs in isolation, that is the thread does not have its progress obstructed by any other thread, it will complete in a bounded number of steps; Lock-Freedom, which ensures that at least one thread must complete within a bounded number of steps even when obstructed; Wait-Freedom, which ensures that *all* threads must complete within a bounded number of steps, regardless of obstruction. The `EpochManager` has been made *lock-free*.

Epoch-Based Reclamation (EBR) is a concurrent-safe memory reclamation system that utilizes *epochs*, which are descriptors for a specific period of time, to determine the quiescence of objects and determine when they are safe to be reclaimed. Concurrent-safe memory reclamation is a non-trivial problem and is at the very root of non-blocking algorithms and data structures. The problem presented by concurrent access is that it is not easy to know whether or not a thread is accessing data we are interested in deleting, and naively deallocating data can result in undefined behavior from a use-after-free error. That is, once an object is freed, it is normally placed on some type of free-list where it can be used in some future allocation, which can cause data corruption in the case of arbitrary writes or segmentation faults in the case of dereferencing pointers. Epoch-Based

[1]RDMA atomics are not yet possible on InfiniBand networks due to a lack of current support in Chapel's implementation.

[2]The equivalent load-linked/store-conditional instructions can be used on ARM.

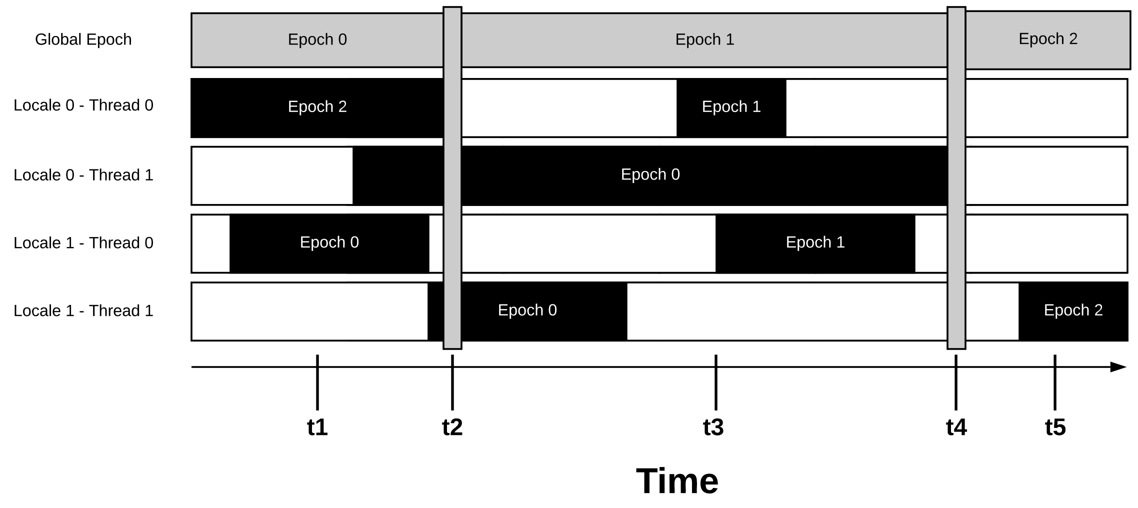


Fig. 1. Illustration of Epoch-Based Reclamation, adapted to distributed memory. Given two locales with two threads each, the illustration begins in epoch 0. At time t_1, the global epoch is 0 but Locale 0 – Thread 0 is still in epoch 2, preventing an advancement to epoch 1. At time t_2, the global epoch is safely advanced to epoch 1 after Locale 0 – Thread 0 becomes quiescent; even though Locale 1 – Thread 0 and Locale 0 – Thread 1 are in Epoch 0, it is still safe to advance the epoch as epoch 0 is only one epoch behind. Locale 0 – Thread 1 remains in epoch 0 from t_1 through t_3, preventing the epoch from advancing until t_4, where the global epoch is then advanced to epoch 2. At t_5 and onward, the algorithm continues much the same.

Reclamation combats this issue by utilizing epochs. EBR tracks the epochs that participating threads are in, where each participating thread must enter an epoch before accessing data, and must leave the epoch afterwards. Generally, if a thread is not in an epoch, it is considered quiescent in that it no longer has access to the objects we are interested in at that given moment. A thread inside of an epoch may or may not be accessing the object at that given time, but out of safety, the deallocation of said objects is deferred until later.

To delete an object, it is first *logically* removed from the data structure from which it is accessible. An example of logical removal would be the removal of a node from a linked list. The logically removed object is then put in a *limbo list*, which is a list of objects to be reclaimed, associated with a given epoch. More formally, an object o that is associated with an epoch e must not be deleted until it is certain that no thread is in epoch e. The only *hazard* in concurrent memory reclamation occurs when another thread is accessing it while it is being deleted, but the logical removal of o entirely removes it from the data structure, and so only threads that have had access prior to the removal can access o. Eventually, once the epoch has been advanced to $e{+}1$, which occurs after all threads are guaranteed to be either inactive or in at least epoch e and *not* epoch $e-1$, it is safe to delete the objects in the limbo list for $e-1$. Note that o is not reclaimed at this point. Instead, the epoch must advance once more, after which there is utmost certainty that o can safely be reclaimed as all participating threads were quiescent after the logical removal of o, and since o is no longer accessible from the current epoch. An illustration of Epoch-Based Reclamation can be seen in Figure 1.

C. Epoch Manager

The `EpochManager` is built on top of the notion of epoch-based reclamation and limbo lists, in that objects that are marked for deletion during an epoch are held in limbo until they are safe to be deleted. To implement the limbo lists, it was necessary to implement a non-blocking data structure that was optimized not only for concurrent insertion, but bulk removal, as all objects in the limbo list are deleted at once, and not incrementally. The limbo list can be viewed as having two phases— an insertion phase, which is entirely concurrent, and a deletion phase, which both occur at disjoint times. A somewhat novel but simple data structure has been designed to significantly reduce overall latency to the point that deferring an object for deletion has been made entirely wait-free during the insertion phase and during the deletion phase, and both are handled in one atomic exchange, shown in Listing 2. Nodes are recycled using a lock-free stack [11] and the ABA-protection provided by the `AtomicObject`.

```
proc push(obj : unmanaged object?) {
    var node = recycleNode(obj);
    var oldHead = _head.exchange(node);
    node.next = oldHead;
}
proc pop() {
    return _head.exchange(nil);
}
```

Listing 2: Wait-Free Limbo List.

The `EpochManager` is *privatized*, in that an instance of the `EpochManager` is created and maintained on each locale, and all accesses the `EpochManager` are *forwarded*, such as the case for field accesses or method invocations, to the instance that is local to that locale. That is, even though

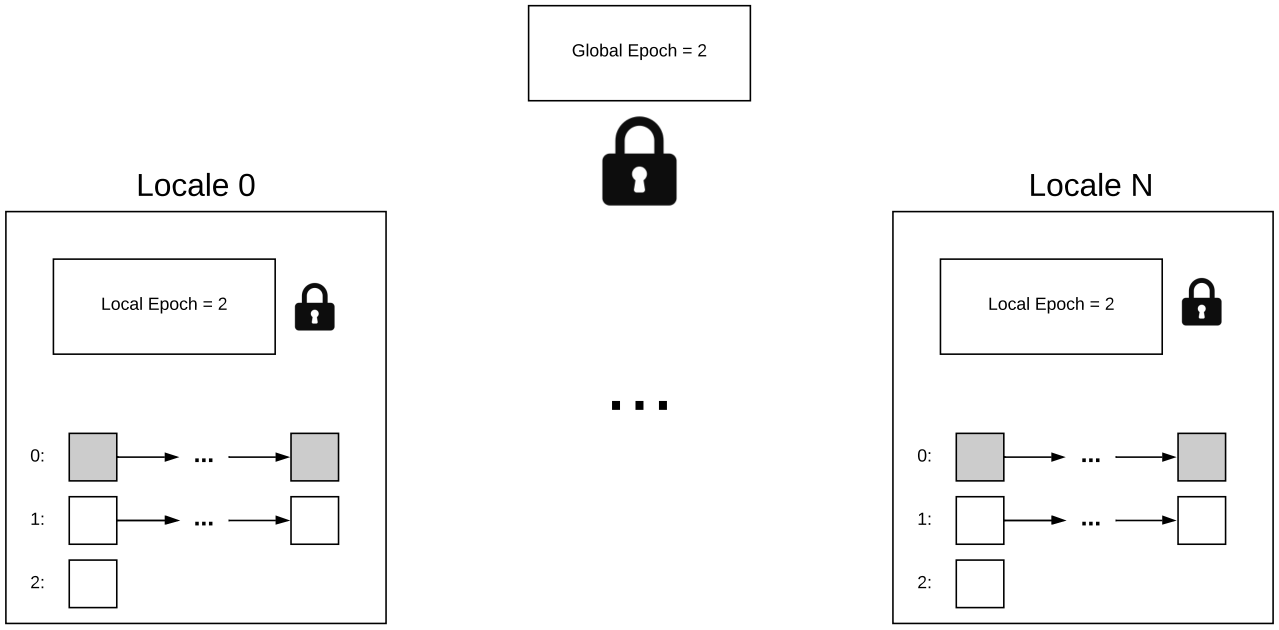


Fig. 2. Illustration of EpochManager when global and local epoch is 2. Each locale manages its own privatized instance, where all accesses are directed to. Limbo list 0, shaded gray, is free to be reclaimed as there is a guarantee that no active task will be in epoch $e - 2$; limbo list 1 is not yet reclaimed, and limbo list 2 becomes the current that all new reclaimed objects will be added to. Local Epoch acts as a locale-private cache for the current epoch, reducing necessary communication. An atomic flag must also be acquired to advance the epoch, first locally and then globally, where in failing to do so will cause the attempting thread to back out (non-blocking).

the `EpochManager` can be used in distributed contexts, such as in distributed parallel `forall` loops, or inside of remote-procedure call (RPC) `on` statements, all accesses are guaranteed to respect locality. This is achieved by *remote-value forwarding* and *record-wrapping*, where a record which holds data required to lookup the instance itself is based by-value, and not by-reference as is the default in Chapel when it comes to `forall` statements. This allows for zero-communication when acquiring the privatized instance. This results in a massive speedup, since replication across locales cuts down all unnecessary communication and allows the caching of data or even keeping locale-specific instances of data, and the record-wrapping eliminates an additional round-trip communication required to obtain the metadata needed to find the privatized instance. In practice, this has been observed by the authors to allow distributed objects to no longer be communication bound, that is bound the available bandwidth and latency of the network, and allows for some truly scalable algorithms. This technology is not new, either, as it has been used in previous works to create distributed data structures [12], [13], [14], [15], and is used as the backbone for Chapel's arrays, domains, and distributions. An illustration is provided in Figure 2.

An array of three limbo lists are maintained on each locale in the privatized instance, representing the possible epochs that any given thread can be in, which are $e-1$, e, and $e+1$. Each locale caches its epoch, which is used when deciding which limbo list to defer deletion of objects to. When it is time to update the global epoch, a task gets elected. In this case, the election is handled in a first-come-first-serve nature via a local atomic flag `is_setting_epoch` for their locale, and then for the locales that the `EpochManager` is distributed over. This has the effect of stemming off unnecessary amounts of communication that would arise if multiple tasks across multiple locales attempted to update the global epoch at the same time, as only one of them can succeed in doing so. As each locale has its own individual instance, a class instance wraps the global epoch itself so that there is a single centralized and coherent epoch that all locales can come to a consensus on.

The `EpochManager` creates a set of *tokens*, which are class instances that keep track of the epoch that a task is currently engaged in. Before a task is free to access a data structure that is protected by the epoch-based reclamation provided by the `EpochManager`, it must first *register* and obtain one of these tokens. When they are finished, they must *unregister* and relinquish them. Two separate lists are maintained for tokens— one which keeps track of free tokens, used when registering and unregistering, and one which is a list of all allocated tokens, which is used to scan the minimum epoch. Once registered, the token is not yet in an epoch, and in fact can be used to perform multiple operations in the same task as an optimization. A token must be *pinned* and *unpinned* just like it must be registered and unregistered, where pinning enters the current epoch, and unpinning exits the current epoch. When an object is to be deleted, it is always added to the current epoch associated with the token. The token is itself wrapped in a managed class so that when it goes out of scope, the token can automatically be unregistered. This is particularly useful while using task-private variable intents on `forall` loops, as shown below.

```
var em = new EpochManager();
// Serial and Shared Memory
var tok = em.register();
tok.pin();
tok.unpin();
tok.unregister();

// Parallel and Distributed (forall)...
forall x in X with (var tok = em.register()) {
   tok.pin();
   tok.deferDelete(x);
   tok.unpin();
} // automatic unregister
em.clear(); // Reclaim everything at once.
```

Listing 3: Example usage of `EpochManager`.

The `EpochManager` will not advance the epoch on its own, and requires user intervention to do so. The user is free to `tryReclaim`, which attempts to advance the epoch if and only if no token on any other locale is in a previous epoch. As well, since the objects to be deleted can be remote, and since remote deallocation would result in RPC, a scatter list is constructed that sorts objects by the locales they are allocated on, significantly cutting down unnecessary communication. The `tryReclaim` method is intended to be invoked on the token or `EpochManager`. It is a global operation, optimized such that if another task is attempting to update the epoch on the current locale, other tasks will swiftly return, without much wasted effort; if another task is attempting to update the global epoch, it will also return after clearing the local flag. The `clear` method is intended to be invoked directly on the `EpochManager` and performs the same action as `tryReclaim` with the exception that it will always reclaim all objects across all epochs, and should be called when there is a guarantee that no other thread is interacting with the `EpochManager`.

The `LocalEpochManager` is a shared-memory optimized variant that functions in a similar way to the `EpochManager`, but differs in that it lacks global epoch and does not take remote objects into consideration when being used, speeding up computations that do not require epoch-based reclamation support across multiple locales.

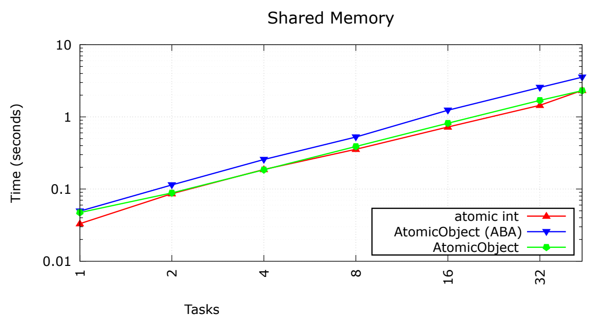
```
proc tryReclaim() {
  if (is_setting_epoch.testAndSet()) then return;
  if (global_epoch.is_setting_epoch.testAndSet()) {
    is_setting_epoch.clear();
    return;
  }

  // Is it safe to reclaim across all locales?
  const this_epoch = global_epoch.read();
  var safeToReclaim = true;
  coforall loc in Locales with (&& reduce safeToReclaim)
    do on loc {
    var _this = getPrivatizedInstance();
    for tok in _this.allocated_list {
      var local_epoch = tok.local_epoch.read();
      if (local_epoch != 0 && local_epoch != this_epoch) {
        safeToReclaim = false;
        break;
      }
    }
  }

  if safeToReclaim {
    const new_epoch = (current_global_epoch % 3) + 1;
    global_epoch.write(new_epoch);
    coforall loc in Locales do on loc {
      // Update each locale's epoch
      var _this = getPrivatizedInstance();
      _this.locale_epoch.write(new_epoch);

      const reclaim_epoch = _this.getReclaimEpoch();
      var reclaim_limbo_list =
          _this.limbo_list[reclaim_epoch];
      var head = reclaim_limbo_list.pop();

      while (head != nil) {
        var obj = head.val;
        var next = head.next;
        // Scatter objects to their locale
        _this.objsToDelete[obj.locale.id].append(obj);
        delete head;
        head = next;
      }
      coforall loc in Locales do on loc {
        // Bulk transfer and delete
        var ourObjs =
            _this.objsToDelete[here.id].getArray();
        delete ourObjs;
      }

      // Clear scatter list
      forall i in LocaleSpace do
        _this.objsToDelete[i].clear();
    }
  }
  global_epoch.is_setting_epoch.clear();
  is_setting_epoch.clear();
}
```

Listing 4: Implementation of `tryReclaim`.

III. Performance Evaluation

All experiments were conducted on a 64-node Cray XC-50 with 44 core Broadwell CPUs per node, compiled using Chapel 1.20 with the 'fast' flag to enable all compiler and backend optimizations. The experiments were conducted both in the presence and absence of `CHPL_NETWORK_ATOMICS`, which is RDMA atomic operations. These RDMA atomics are not coherent, and so all atomic operations, including those that are performed locally on the same system, must go through the Gemini or Aries NIC. This overhead of using network atomics for local operation has been measured to be as much as an order of magnitude by the authors. While in the development of both the `EpochManager` and `AtomicObject`, care has been taken to eliminate the usage of RDMA atomics when they are unnecessary by 'opting out', it is still useful to compare performance with and without the support for RDMA atomics. Since RDMA atomics are only available on Gemini and Aries, the performance observed with RDMA atomics disabled would be relatively close to the performance seen when using InfiniBand, as Chapel does not utilize InfiniBand RDMA atomics even when present on the system.

The performance of both `AtomicObject` and `EpochManager` is measured to show the raw overhead of both constructs with the goal of proving them both scalable. Such microbenchmarks are important in that it is exceedingly difficult to build scalable non-blocking algorithms without scalable building blocks, beginning



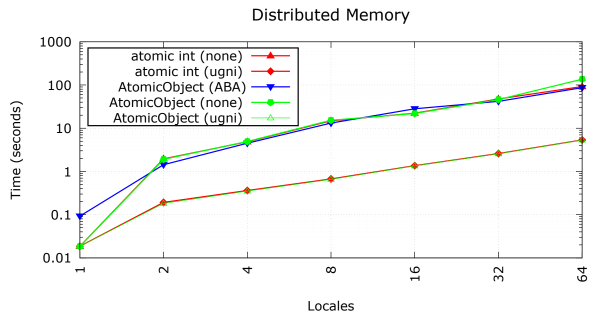


Fig. 3. `AtomicObject` vs `atomic int`

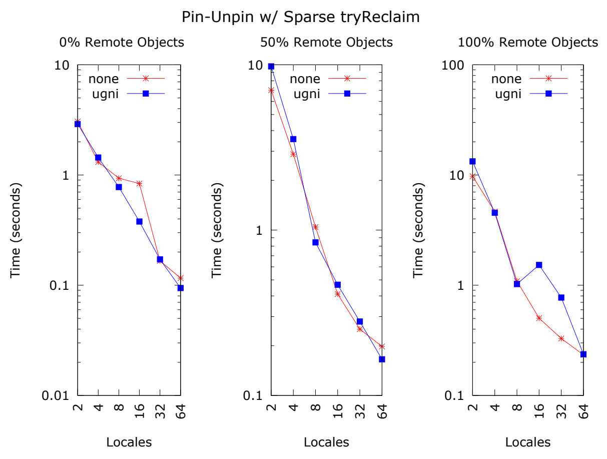


Fig. 4. Deletion with `tryReclaim` called once per 1024 iterations.

with *AtomicObject*, which is a fundamental building block, including for `EpochManager`. The `AtomicObject` is compared to one of the only types that atomics are natively supported in Chapel, the `atomic int`. The `atomic int` is also a sibling of the `atomic uint`, which the `AtomicObject` is built on top of. Microbenchmarks involving the `AtomicObject` test the overhead injected by the abstraction. Microbenchmarks involving the `EpochManager` focus on different use-cases and workloads.

In this section, we explore two performance criteria. First, we evaluate the performance of Atomic Objects against Chapel's atomic variables. We use Chapel's `atomic int`. The experiments focus on the common set of operations

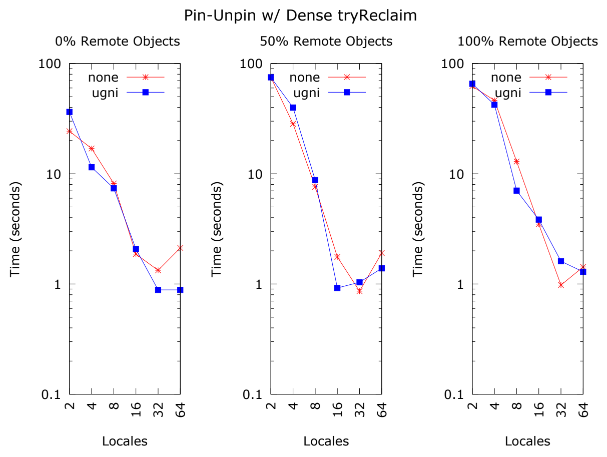


Fig. 5. Deletion with `tryReclaim` called every iteration.

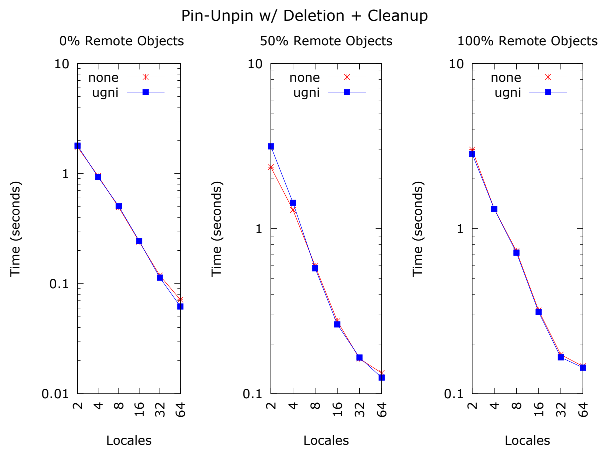


Fig. 6. Deletion with reclamation only performed at end.

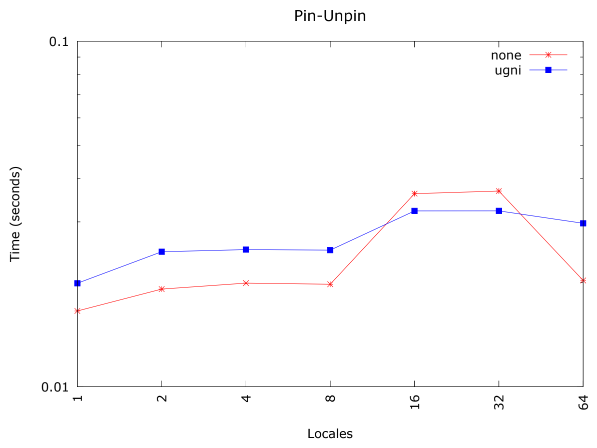


Fig. 7. Read-only workload without deletion.

available between Chapel's atomic variables and Atomic Objects: read, write, compare and swap, and exchange. Second, we evaluate the scalability of `EpochManager`, testing raw acquire/release, memory reclamation with remote objects, and manual garbage collection every fixed number of iterations.

A. Atomic Objects Performance Evaluation

We compare Atomic Objects performance with and without ABA protection against Chapel's `atomic int` in shared memory and distributed memory, as shown in Figure 3. The experiment evaluates strong scaling, with each task performing the same number of operations, comprising 25% read, 25% write, 25% compare-and-swap, and 25% exchange operations. Shared memory experiments show that all three of atomic int, AtomicObject (ABA) and AtomicObject scale linearly with an increasing numbers of tasks. AtomicObject without ABA protection performs equivalently to Chapel's atomic int, and AtomicObject (ABA) takes the highest amount of time, with a constant overhead. In distributed memory, the performance of AtomicObject without ABA protection is equivalent to Chapel's atomic int. This shows that even in distributed memory, there is no noticeable overhead, and it scales linearly with the number of locales, whether RDMA atomics are used or not. AtomicObject (ABA) scales linearly with an increasing numbers of locales. It performs equivalently to Chapel's atomic int without network atomics.

B. Epoch Manager Performance Evaluation

```
// Create manager instance
var manager = new EpochManager();
var objsDom = {0..#numObjects} dmapped Cyclic(startIdx=0);
var objs : [objsDom] unmanaged C();
// Randomize locale that each object is allocated on
randomizeObjs(objs);
forall obj in objs with (
  var tok = manager.register(),
  var M : int
) {
  tok.pin();
  // If we are deleting...
  tok.deferDelete(obj);
  tok.unpin();
  M += 1;
  // If we are tryReclaim'ing...
  if M % perIteration == 0 {
    tok.tryReclaim();
  }
}
// Reclaim all objects at end
manager.clear();
```

Listing 5: Microbenchmark of `EpochManager`.

The microbenchmarks for `EpochManager` are similar to Listing 5.

We evaluate the scalability of `EpochManager` under various workloads, which should be representative of the different use-cases. In a read-only workload, such as for a read-often write-rarely data structure, such as when performing a lookup in a hash table or a linked list, it may be suitable to just pin at the beginning of the operation, and then unpin at the end. Demonstrated in Figure 7, performance is essentially stable across multiple locales, demonstrating that even in distributed contexts it can scale reasonably well as all locales forward their accesses to their privatized instances despite being in a parallel and distributed `forall` loop. In Figure 6, another typical workload is analyzed where no reclamation is performed until the very end, which is typical when the number of objects is bound and can fit within memory without running out of memory. The number of remote objects to be reclaimed varies by 0%, 50%, and 100%, which measures the overhead of reclaiming remote objects. The `EpochManager` scales even in the case where `tryReclaim` is invoked with increasing frequency, as demonstrated by the results displayed in Figure 4. When reclamation is performed, not even the locale where the global epoch is allocated is bogged down by redundant requests thanks to the first-come-first-serve election of tasks, and scales equally both with and without RDMA atomics. In the case where the user does not want to take any chances and attempts to `tryReclaim` on every iteration, there is still scalability, as shown in Figure 5.

IV. Conclusion

The `AtomicObject` is a solution to the problem of a lack of language support atomic operations on objects. The implementation not only provides the operations in shared-memory but distributed memory, utilizing pointer compression that enables RDMA atomic operations, which are on-par with the performance for atomic operations on integers, while also providing protection from the ABA problem with memory reclamation via usage of double-word compare-and-swap. The `EpochManager` is a non-blocking epoch-based reclamation garbage collection system that allows for concurrent-safe reclamation even in distributed-memory contexts. Both of these are essential building blocks for developing non-blocking algorithms in both shared-memory and distributed-memory. In future works, it is planned to allow more than 2^{16} locales while still allowing RDMA atomic operations, by introducing another level of indirection and utilizing an descriptor index into a separate table of objects in place of the pointer itself. As well, there is planned exploration of allowing atomics on owned and borrow types. Also in future works, an application of both the constructs in the porting of the Interlocked Hash Table [16] is complete and awaiting release; their applications in the creation of other distributed algorithms are also planned.

References

[1] M.-C. M and S. L, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Transactions on Computer Systems (TOCS)*, Feb. 1991.

[2] M. Herlihy, V. Luchangco, and M. Moir, "Obstruction-free synchronization: double-ended queues as an example," in *23rd International Conference on Distributed Computing Systems, 2003. Proceedings.*, (Providence, Rhode Island, USA), pp. 522–529, IEEE, 2003.

[3] H. Massalin and C. Pu, "A lock-free multiprocessor os kernel," *ACM SIGOPS Operating Systems Review*, vol. 26, no. 2, p. 108, 1992.

[4] M. Herlihy, "Wait-free synchronization," *ACM Trans. Program. Lang. Syst.*, vol. 13, p. 124149, Jan. 1991.

[5] A. Hayashi, J. Zhao, M. Ferguson, and V. Sarkar, “LLVM-based communication optimizations for PGAS programs,” in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC - LLVM ’15*, (Austin, Texas), pp. 1–11, ACM Press, 2015.

[6] D. L. Detlefs, P. A. Martin, M. Moir, and G. L. Steele, “Lock-free reference counting,” in *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing*, PODC 01, (New York, NY, USA), p. 190199, Association for Computing Machinery, 2001.

[7] M. M. Michael, “Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, pp. 491–504, June 2004.

[8] H. Wen, J. Izraelevitz, W. Cai, H. A. Beadle, and M. L. Scott, “Interval-based Memory Reclamation,” in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP ’18, (New York, NY, USA), pp. 1–13, ACM, 2018. event-place: Vienna, Austria.

[9] T. E. Hart, P. E. McKenney, A. D. Brown, and J. Walpole, “Performance of memory reclamation for lockless synchronization,” *Journal of Parallel and Distributed Computing*, vol. 67, pp. 1270–1285, Dec. 2007.

[10] K. Fraser, “Practical lock-freedom,” tech. rep., University of Cambridge, Computer Laboratory, 2004.

[11] D. Hendler, N. Shavit, and L. Yerushalmi, “A scalable lock-free stack algorithm,” in *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA ’04, (Barcelona, Spain), pp. 206–215, Association for Computing Machinery, June 2004.

[12] L. Jenkins, M. Zalewski, and M. Ferguson, “Chapel Aggregation Library (CAL),” in *2018 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI (PAW-ATM)*, pp. 34–43, Nov. 2018.

[13] L. Jenkins and M. Zalewski, “Chapel Graph Library (CGL),” in *Proceedings of the ACM SIGPLAN 6th on Chapel Implementers and Users Workshop*, CHIUW 2019, (New York, NY, USA), pp. 29–30, ACM, 2019. event-place: Phoenix, AZ, USA.

[14] L. Jenkins, T. Bhuiyan, S. Harun, C. Lightsey, D. Mentgen, S. Aksoy, T. Stavcnger, M. Zalewski, H. Medal, and C. Joslyn, “Chapel HyperGraph Library (CHGL),” in *2018 IEEE High Performance extreme Computing Conference (HPEC)*, (Waltham, MA), pp. 1–6, IEEE, Sept. 2018.

[15] L. Jenkins, “RCUArray: An RCU-Like Parallel-Safe Distributed Resizable Array,” in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 925–933, May 2018.

[16] L. Jenkins, T. Zhou, and M. Spear, “Redesigning gos built-in map to support concurrent operations,” in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 14–26, IEEE, 2017.