Distributed Non-Blocking Data Structures for the Partitioned Global Address Space

Garvit Dewan, Louis Jenkins

Background

Why ‘Non-Blocking’?

- Disadvantage of using locks (mutexes, semaphores, spinlocks, etc.)
  - Potential to introduce Deadlock, Livelock, Priority Inversion, etc.
  - Coarse-grained synchronization eliminates scalability
  - Fine-grained synchronization can scale but increases chance of deadlock/livelock
- Advantages of Non-Blocking algorithms
  - *Liveness* - Property of the progress of threads in a system
    - Obstruction-Free - Threads finish in finite number of steps if not obstructed
    - Lock-Free - At least one thread finishes in a bounded number of steps
    - Wait-Free - All threads finish in a bounded number of steps
  - Provide scalability while also providing guarantees on liveness, although difficult to create

Concurrent-Memory Reclamation

- Reclamation of objects that may be accessed by other threads
  - Reclaim too early and you have a use-after-free and undefined behavior
  - Never reclaim and you leak memory
  - Not all systems have built-in garbage collection (C,C++,Chapel)
  - Should be fast enough to not induce too much overhead on operations
- Different approaches towards the solution
  - Pointer-Based - Objects are not reclaimed if any thread is explicitly tracking it
  - Quiescent-Based - Objects are not reclaimed until all threads become quiescent (epochs)
  - Reference-Counting - Objects are not reclaimed until reference count is 0
  - Garbage-Collection - Objects are not reclaimed if reachable by any thread

Partitioned Global Address Space (PGAS)

- Distributed “Shared” Memory Model
  - Global address space composed of virtual address of individual processing elements (PE)
  - Remote Direct Memory Access (RDMA) are handled entirely by NIC (no CPU intervention)
  - PEs can access memory on remote PEs via RDMA PUT/GET (analog of store/load)
    - Low Latency Atomics (μs) via RDMA also available on most NICs
- Chapel, the PGAS programming language
  - Transparent conversion of loads and stores to GETs and PUTs respectively
  - Provides features that enable (pseudo) first-class distributed objects
    - Accesses are redirected to *privatized* instance of object; added layer of transparency
  - Enable writing algorithms in both distributed- and shared-memory
    - Write-Once, Run-Anywhere (w.r.t shared-memory and distributed systems)

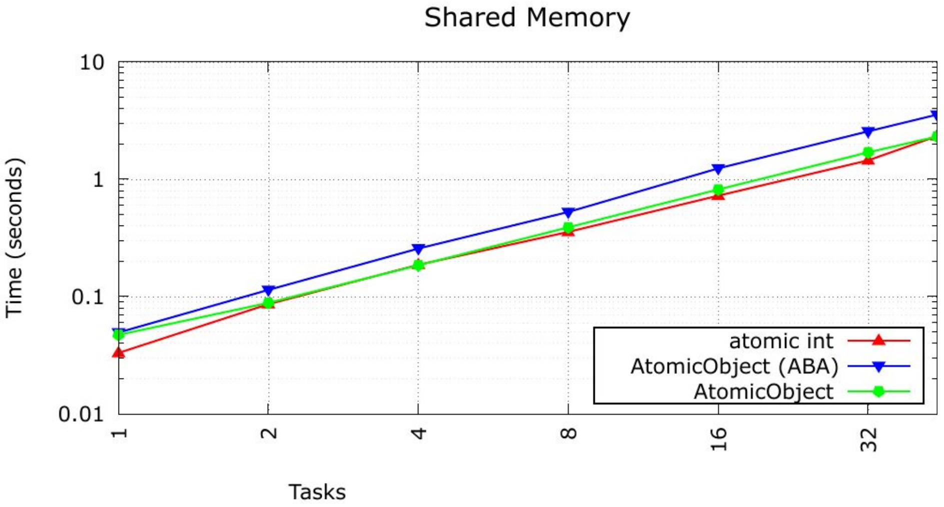
Distributed Non-Blocking Algorithms and Data Structures

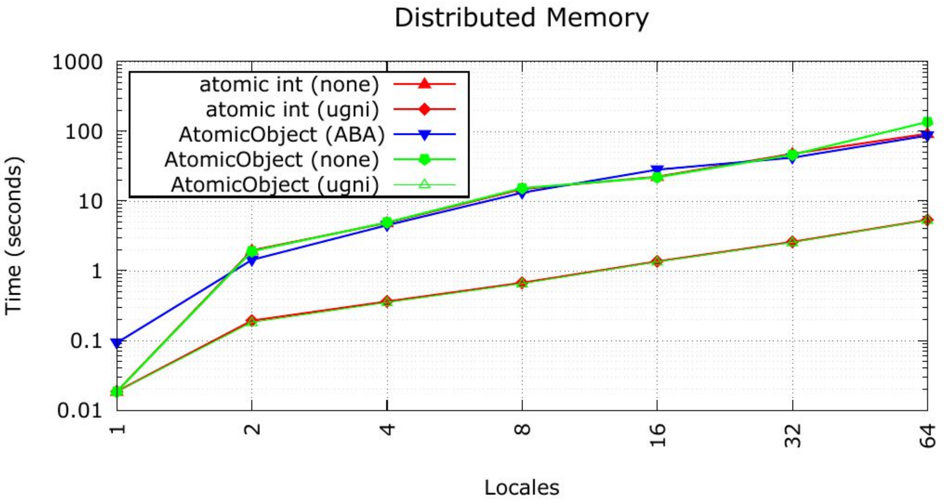
Atomics on Distributed Objects

- Missing prerequisite for Non-Blocking Algorithms and Data Structures
  - Pointers are represented as 128-bit structs (64-bit virtual address, 64-bit locality info.)
  - NICs only support 64-bit atomic operations
    - Require ‘remote-execution’ atomics (Active Message)
    - RDMA atomics outperform ‘remote-execution’ atomics by an order of magnitude
- Implemented AtomicObject
  - Compress 48-bit virtual address with 16-bit locality information (64-bit total for RDMA)
  - Extended to provide solution to ABA problem (ABAWrapper)
    - Attached 64-bit sequence number to compressed 64-bit pointer
    - Requires 128-bit Compare-and-Swap primitives via remote execution
    - Used internally to implement other more scalable solutions

AtomicObject Performance

- Compare AtomicObject to native atomic implementation
  - Strong scaling, w/ vs. w/o ABA, w/ vs w/o RDMA, distributed- and shared-memory
  - Equal ratio of reads, writes, compare-and-swap, and exchange operations





AtomicObject - Future works

- Support more than 16-bits worth of locality information (> 2^{16} PEs)
  - Introduce distributed table[1] of objects where 64-bit index serves as pointer to object
    - "All problems in computer science can be solved by another level of indirection"
- Explore possibility of handling managed types (`owned` and `shared`)
  - Currently only supports `unmanaged` types.

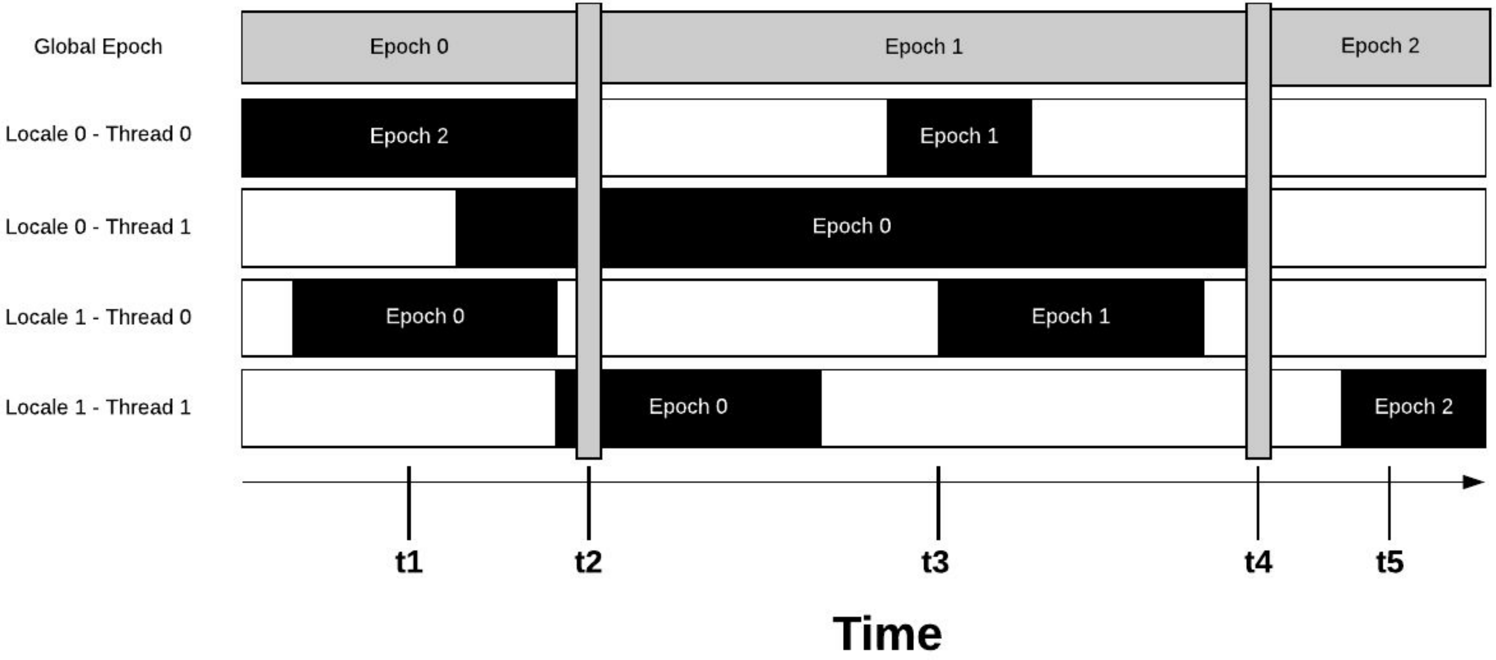
[1] L. Jenkins, "RCUArray: An RCU-Like Parallel-Safe Distributed Resizable Array," 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Vancouver, BC, 2018, pp. 925-933, doi: 10.1109/IPDPSW.2018.00146.

EpochManager

- EpochManager is built on the notion of
  - Epoch-Based Reclamation
  - Limbo Lists

- Epoch-Based Reclamation (EBR)
  - Is a concurrent-safe memory reclamation system
  - Utilizes epochs, which are descriptors for a specific period of time, to determine
    - The quiescence of objects
    - When they are safe to be reclaimed

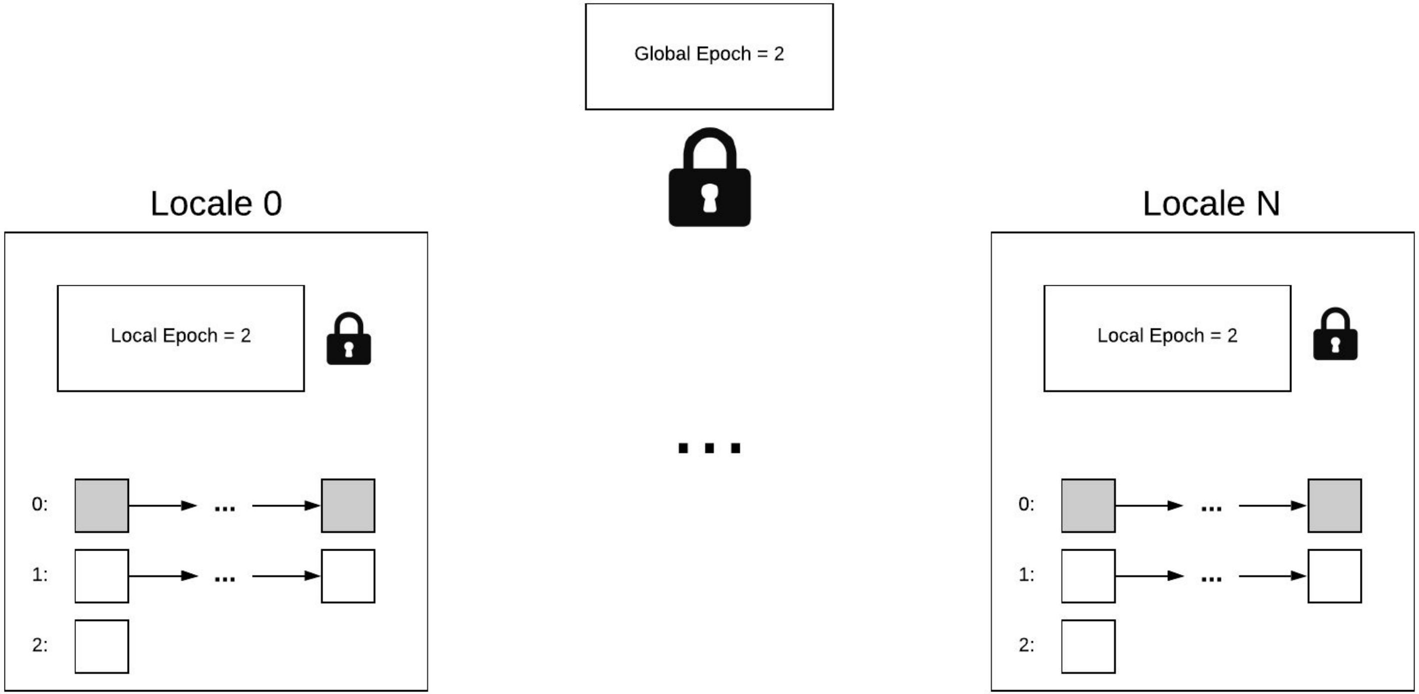
Epoch-Based Reclamation



EpochManager

- Limbo Lists
    - Objects marked for deletion during an epoch are held in limbo until they are safe to be deleted
    - 2 phases:
        - concurrent insertion
        - Bulk deletion

EpochManager



EpochManager

- Limbo Lists
  - Implemented using push-exchange stack
  - Both insertion and deletion operations are wait-free

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

EpochManager

- EpochManager is *privatized*
  - An instance of EpochManager is created and maintained on each locale
  - 3 limbo lists per locale, corresponding to epochs $e - 1$, e, $e + 1$
  - One global epoch; locale-local epoch

EpochManager

- Tokens
  - Issued to each participating task
  - Keeps track of status of a registered task (active/inactive)
  - In case of active task, keeps track of the epoch in which the task is engaged in
  - Each task must *register* to obtain a token; *unregister* to free a token
    - automatically triggered when token goes out of scope

- Tokens are managed using two lists
  - allocated_list: List of all allocated tokens
  - free_list: List of free tokens ready to be recycled

EpochManager

- To enter critical section, a task must *pin* its token
    - Pinning marks the task as “active”
    - Task is always pinned to the current locale epoch
    - While a task is pinned, its local epoch cannot be updated
    - Objects deleted go into the corresponding epoch’s limbo list

- To exit critical section, a task must *unpin* its token
    - Unpinning marks the task as “inactive”

Example usage of EpochManager

```
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.
```

EpochManager

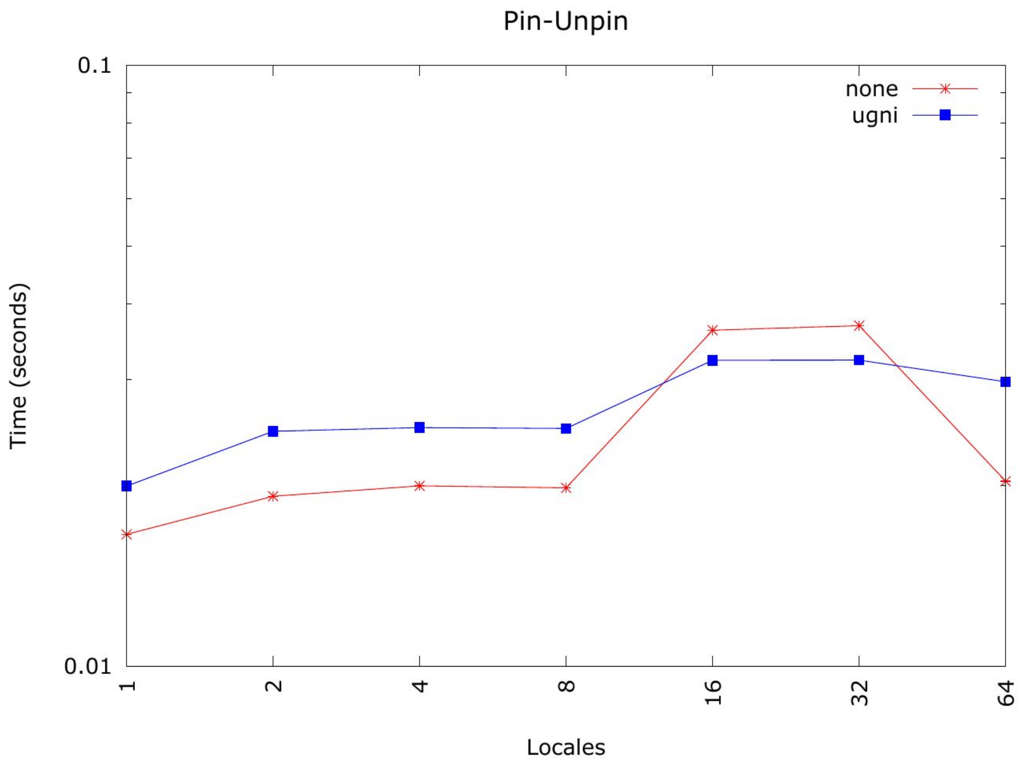
- Global epoch can be advanced by calling tryReclaim on the token
  - Advances the epoch only if no active task is in an epoch previous to the global epoch
  - On successful advancement, reclaims objects safe to be reclaimed
  - Constructs a scatter list of remote objects

EpochManager

- LocalEpochManager
    - Shared-memory optimized variant
    - Lacks locale-local epochs
    - Does not take remote objects into consideration

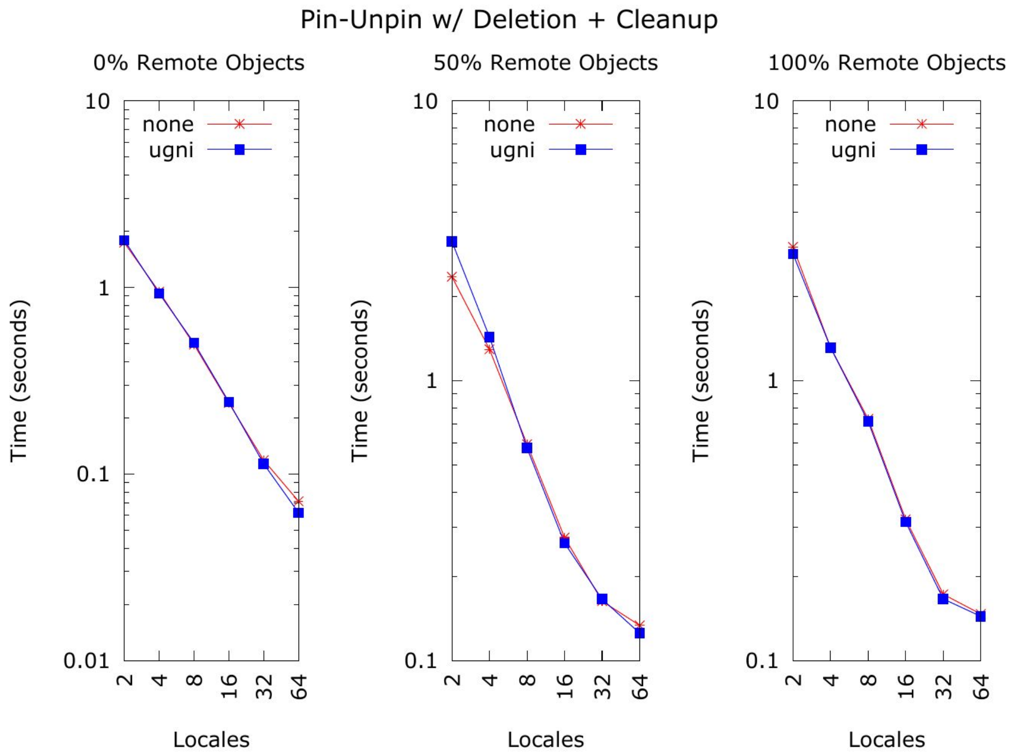
EpochManager Performance

- Read-Only (no deletion)
  - Thread Enters
  - Thread Read/Writes Data
  - Thread Exits
- Relatively stable perf.
  - Even when distributed
  - Shows privatization benefit
    - Locale-private epochs



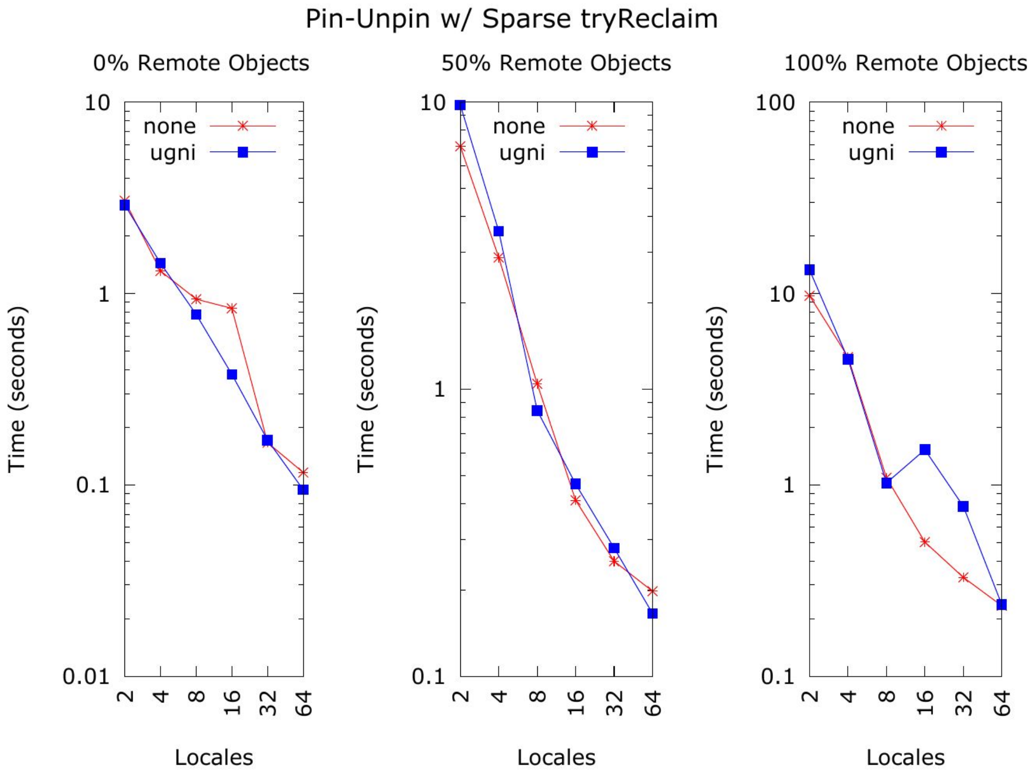
EpochManager Performance

- Small Static Workload
  - Thread Enters
  - Thread Reads/Writes Data
  - Thread Deletes Data
  - Thread Exits
  - Reclaim Memory when finished
- Embarrassingly Parallel
  - Scales in distributed memory
  - Highlights privatization benefit
    - Locale-private limbo lists



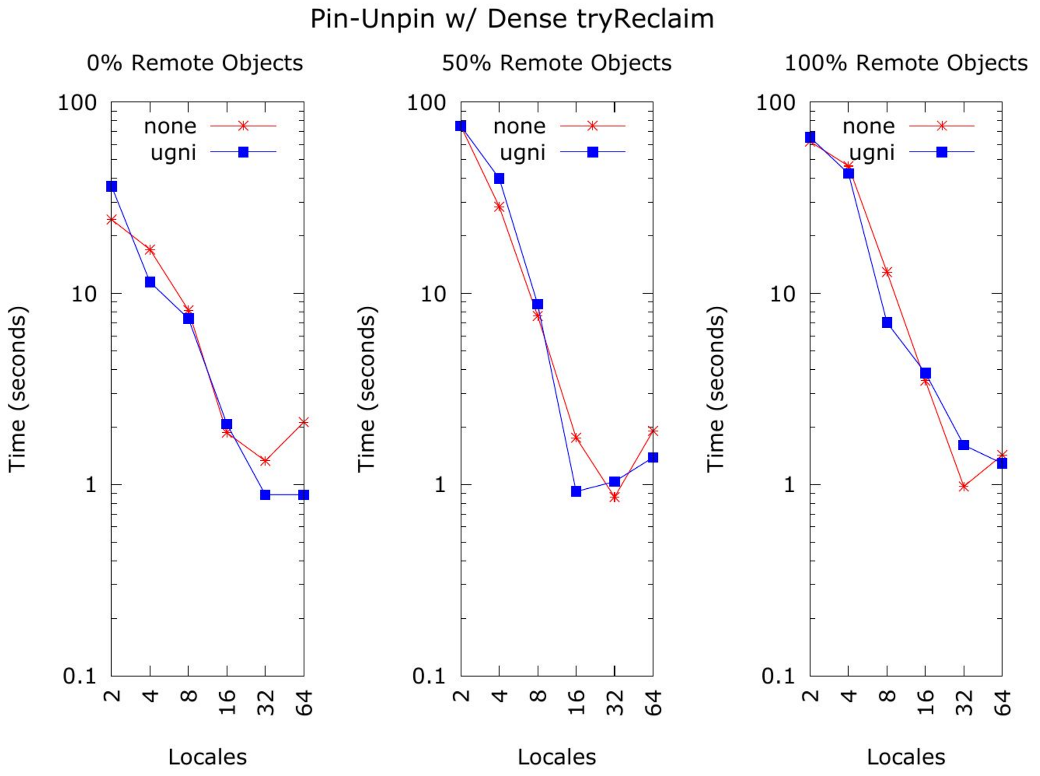
EpochManager Performance

- Typical Workload
  - Thread Enters
  - Thread Read/Writes Data
  - Thread Deletes Data
  - Thread Exits
  - Thread periodically reclaims
  - Reclaim Memory when finished
- Embarrassingly Parallel
  - Scales in distributed memory
  - Further highlights privatization
    - Scatter lists for reclamation



EpochManager Performance

- Worst-Case Workload
  - Thread Enters
  - Thread Read/Writes Data
  - Thread Deletes Data
  - Thread Exits
  - Thread *always* reclaims
  - Reclaim Memory when finished
- Embarrassingly Parallel
  - Scales in distributed memory
  - Further highlights privatization
    - Scatter lists for reclamation



EpochManager - Future Work

- Interlocked Hash Table[2]
  - 80% find, 10% insert, 10% delete
    - Shared-Memory ------>
    - 60x faster than standard...
  - 1.5B Op/Sec at 64 locales!!!
    - Map not dist. data structure
- More data structures to come

Tasks	Map (ns/op)	ConcurrentMap (ns/op)
1	106.038	207.047
2	165.431	126.87
4	219.872	68.0732
8	268.668	35.6337
16	421.874	19.3164
32	577.605	11.1499
44	621.96	10.99

[2] L. Jenkins, T. Zhou and M. Spear, "Redesigning Go's Built-In Map to Support Concurrent Operations," 2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT), Portland, OR, 2017, pp. 14-26, doi: 10.1109/PACT.2017.45.

Conclusion

- The AtomicObject is a solution to the problem of a lack of language support atomic operations on objects
  - Works in both shared and distributed memory
  - Provides protection from the ABA problem
- The EpochManager is a non-blocking epoch-based reclamation garbage collection system
  - 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.
- Contact:

  - Garvit Dewan - gdewan@cs.iitr.ac.in
  - Louis Jenkins - ljenkin4@ur.rochester.edu