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

By Louis Jenkins
The Problem
Parallel-Safe Resizing

• Not inherently thread-safe to access memory while it is being resized
  • Memory has to be ‘moved’ from the smaller storage into larger storage
The Problem
Parallel-Safe Resizing

• Not inherently thread-safe to access memory while it is being resized
  • Memory has to be ‘moved’ from the smaller storage into larger storage
  • Concurrent loads and stores can result in undefined behavior
    • Stores after memory is moved can be lost entirely
The Problem
Parallel-Safe Resizing

• Not inherently thread-safe to access memory while it is being resized
  • Memory has to be ‘moved’ from the smaller storage into larger storage
  • Concurrent loads and stores can result in undefined behavior
    • Stores after memory is moved can be lost entirely
    • Loads and Stores after the smaller storage is reclaimed can produce undefined behavior
The Problem
Parallel-Safe Resizing

• Not inherently thread-safe to access memory while it is being resized
  • Memory has to be ‘moved’ from the smaller storage into larger storage
  • Concurrent loads and stores can result in undefined behavior
    • Stores after memory is moved can be lost entirely
    • Loads and Stores after the smaller storage is reclaimed can produce undefined behavior
• Why not just synchronize access?  
  • Not scalable
The Problem
Parallel-Safe Resizing

• Not inherently thread-safe to access memory while it is being resized
  • Memory has to be ‘moved’ from the smaller storage into larger storage
  • Concurrent loads and stores can result in undefined behavior
    • Stores after memory is moved can be lost entirely
    • Loads and Stores after the smaller storage is reclaimed can produce undefined behavior

• Why not just synchronize access?
  • Not scalable

• What do we need?
The Problem
Parallel-Safe Resizing

• Not inherently thread-safe to access memory while it is being resized
  • Memory has to be ‘moved’ from the smaller storage into larger storage
  • Concurrent loads and stores can result in undefined behavior
    • Stores after memory is moved can be lost entirely
    • Loads and Stores after the smaller storage is reclaimed can produce undefined behavior

• Why not just synchronize access?
  • Not scalable

• What do we need?
  1. Allow concurrent access to both smaller and larger storage
The Problem
Parallel-Safe Resizing

• Not inherently thread-safe to access memory while it is being resized
  • Memory has to be ‘moved’ from the smaller storage into larger storage
  • Concurrent loads and stores can result in undefined behavior
    • Stores after memory is moved can be lost entirely
    • Loads and Stores after the smaller storage is reclaimed can produce undefined behavior

• Why not just synchronize access?
  • Not scalable

• What do we need?
  1. Allow concurrent access to both smaller and larger storage
  2. Ensure safe memory management of smaller storage
The Problem
Parallel-Safe Resizing

• Not inherently thread-safe to access memory while it is being resized
  • Memory has to be ‘moved’ from the smaller storage into larger storage
  • Concurrent loads and stores can result in undefined behavior
    • Stores after memory is moved can be lost entirely
    • Loads and Stores after the smaller storage is reclaimed can produce undefined behavior

• Why not just synchronize access?
  • Not scalable

• What do we need?
  1. Allow concurrent access to both smaller and larger storage
  2. Ensure safe memory management of smaller storage
  3. Ensure that stores to old memory are visible in larger storage
Read-Copy-Update (RCU)

- Synchronization strategy that favors performance of readers over writers
  - **Read** the current snapshot $s$

\[
S = (b_1)
\]
Read-Copy-Update (RCU)

• Synchronization strategy that favors performance of readers over writers
  • **Read** the current snapshot $s$
  • **Copy** $s$ to create $s'$

$$S = (b_1) \quad S' = (b_1)$$
Read-Copy-Update (RCU)

- Synchronization strategy that favors performance of readers over writers
  - **Read** the current snapshot \( s \)
  - **Copy** \( s \) to create \( s' \)
  - **Update** applied to \( s' \)…

\[
S = (b_1) \\
S' = (b_1, b_2)
\]
Read-Copy-Update (RCU)

• Synchronization strategy that favors performance of readers over writers
  • **Read** the current snapshot \( s \)
  • **Copy** \( s \) to create \( s' \)
  • **Update** applied to \( s' \), \( s' \) becomes new current snapshot

\[
S = (b_1) \\
S' = (b_1, b_2)
\]
Read-Copy-Update (RCU)

- Synchronization strategy that favors performance of readers over writers
  - **Read** the current snapshot $s$
  - **Copy** $s$ to create $s'$
  - **Update** applied to $s'$, $s'$ becomes new current snapshot
- Not always applicable in all situations
  - Must be safe to access at least two different snapshots of the same data

$S = b_1$
$S' = (b_1, b_2)$
Read-Copy-Update (RCU)

- Synchronization strategy that favors performance of readers over writers
  - **Read** the current snapshot $s$
  - **Copy** $s$ to create $s'$
  - **Update** applied to $s'$, $s'$ becomes new current snapshot
- Not always applicable in all situations
  - Must be safe to access *at least* two different snapshots of the same data

**Read-Copy-Update**
- Readers Concurrent with Readers

**Reader-Writer Locks**
- Readers Concurrent With Readers
Read-Copy-Update (RCU)

- Synchronization strategy that favors performance of readers over writers
  - **Read** the current snapshot $s$
  - **Copy** $s$ to create $s'$
  - **Update** applied to $s'$, $s'$ becomes new current snapshot
- Not always applicable in all situations
  - Must be safe to access *at least* two different snapshots of the same data

<table>
<thead>
<tr>
<th>Read-Copy-Update</th>
<th>Reader-Writer Locks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Readers Concurrent with Readers</td>
<td>Readers Concurrent With Readers</td>
</tr>
<tr>
<td>Writers Mutually Exclusive with Writers</td>
<td>Writers Mutually Exclusive with Writers</td>
</tr>
</tbody>
</table>
Read-Copy-Update (RCU)

- Synchronization strategy that favors performance of readers over writers
  - **Read** the current snapshot $s$
  - **Copy** $s$ to create $s'$
  - **Update** applied to $s'$, $s'$ becomes new current snapshot
- Not always applicable in all situations
  - Must be safe to access *at least* two different snapshots of the same data

**Read-Copy-Update**
- Readers Concurrent with Readers
- Writers Mutually Exclusive with Writers
- Readers Concurrent with Writers

**Reader-Writer Locks**
- Readers Concurrent With Readers
- Writers Mutually Exclusive with Writers
- Readers Mutually Exclusive with Writers
Distributed RCU

- Privatization and Snapshots
  - Each node in the cluster has its own local snapshot

Locale #0

\[
S = (b_1)
\]

Locale #1

\[
S = (b_1)
\]

Locale #2

\[
S = (b_1)
\]

Locale #3

\[
S = (b_1)
\]
Distributed RCU

- Privatization and Snapshots
  - Each node in the cluster has its own local snapshot
  - All local snapshots point to the same block

\[ S = b_1 \]
Distributed RCU

- Privatization and Snapshots
  - Each node in the cluster has its own local snapshot
  - All local snapshots point to the same block

- Reader Concurrency
  - Readers will read from local snapshot only
  - All readers regardless of node will see same block
  - All stores to $b_1$ are seen by any snapshot or node
Distributed RCU

• Privatization and Snapshots
  • Each node in the cluster has its own local snapshot
  • All local snapshots point to the same block

• Reader Concurrency
  • Readers will read from local snapshot only
  • All readers regardless of node will see same block
  • All stores to $b_1$ are seen by any snapshot or node

• Writer Mutual Exclusion
  • Use a distributed lock
Distributed RCU

- Privatization and Snapshots
  - Each node in the cluster has its own local snapshot
  - All local snapshots point to the same block

- Reader Concurrency
  - Readers will read from local snapshot only
  - All readers regardless of node will see same block
  - All stores to $b_1$ are seen by any snapshot or node

- Writer Mutual Exclusion
  - Use a distributed lock
  - Perform each update local to each node
Distributed RCU

• Privatization and Snapshots
  • Each node in the cluster has its own local snapshot
  • All local snapshots point to the same block

• Reader Concurrency
  • Readers will read from local snapshot only
  • All readers regardless of node will see same block
  • All stores to $b_1$ are seen by any snapshot or node

• Writer Mutual Exclusion
  • Use a distributed lock
  • Perform each update local to each node

• Results
  • Fast and parallel-safe loads/stores across multiple nodes
  • Allow for loads and stores to be immediately visible
  • 40x faster resizing than naïve Block Distribution at 32-nodes
RCUArray – Resizing Example

Set of readers $R$ begin using snapshot $s$
RCUArray – Resizing

Writer acquires Cluster Lock

$s \rightarrow R \rightarrow b_1$

$s \rightarrow R \rightarrow b_1$
RCUArray – Resizing

Writer clones $s$ to create $s'$
RCUArray – Resizing

Writer appends block $b_2$ to $s'$
RCUArray – Resizing

Writer updates current snapshot to $s'$
RCUArray – Resizing

Set of readers $R'$ begin accessing $s'$
RCUArray – Resizing

Readers $R$ finish using $s$
RCUArray – Resizing
RCUArray – Resizing

Writer releases cluster lock
Network Atomics vs Remote Execution Atomics

- In Chapel, pointers to potentially remote memory are widened to 128-bits
  - 64-bit Address, 32-bit Locale id, 32-bit Sub-locale id (NUMA)
- Cray’s Aeries NIC only supports 64-bit network atomic operations
  - Atomics via remote execution proves to be significantly slower than network atomics
  - Distributed wait-free algorithms can scale with network atomics
    - Must have a low constant bounds in inter-node communications

Network Execution 26x faster (32 Nodes)

Network Execution 20x faster (32 Nodes)
• Replacing Wide Pointers
  • Blocks have locality information
  • 64-bits vs 128-bits
  • Network Atomics

• Recycling Memory
  • Each node recycles indices to local blocks

• Dynamic Heap
  • Parallel-Safe and Fast Resizing
  • Distributed across multiple locales
  • Great as a per data-structure heap
Conclusion

• Chapel makes RCU easier...
  • Lot of abstraction and language constructs
    • Privatization
    • Parallel remote tasks
  • Including Distributed RCU...

• RCUArray as a distribution
  • Exploring implementation under Domain map Standard Interface (DSI)

• Memory Management Related Efforts
  • Current efforts to add Quiescent State-Based “Garbage Collector” into language
    • 75% finished runtime changes... but on hold
  • Plans to introduce a Epoch-Based “Garbage Collector” as a Chapel module...