Distributed Global-View Data Structures in the Partitioned Global Address Space

Garvit Dewan

Under the guidance of Dr. Pradumn K. Pandey

Indian Institute of Technology Roorkee

Abstract

The Partitioned Global Address Space (PGAS), a memory model in which the global address space is explicitly partitioned across compute nodes in a cluster, strives to bridge the gap between shared-memory and distributed-memory programming. To further bridge this gap, there has been an adoption of global-view distributed data structures, such as `global arrays' or `distributed arrays'. This project focuses on taking global-view programming to the next step by enabling and creating more complex distributed data structures.

Introduction

The **Partitioned Global Address Space** (**PGAS**) is a parallel programming model. PGAS assumes a logically partitioned global memory address space. A portion of the partition is local to each process, thread, or processing element. The novelty of PGAS is that the portions of the shared memory space may have an affinity for a particular process, thereby exploiting locality of reference. The PGAS model forms the basis of many programming languages, such as Coarray Fortran, DASH, Fortress, Split-C, Unified Parallel C, **Chapel**, SHMEM, Coarray C++, Global Arrays, X10 and UPC++.

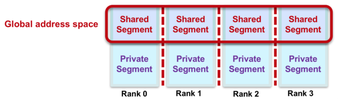

Figure 1:PGAS Memory Model. Source: [8]

This project focuses on taking global-view programming to the next step by creating more complex distributed data structures. The first step is enabling scalable memory reclamation techniques, either by the direct application or derived from shared-memory counterparts such as **Epoch-Based Reclamation**, Quiescent-State Based Reclamation, or Hazard Pointers. The next step is to create distributed adaptations from **non-blocking shared-memory data structures** such as queues and stacks, Lastly, as a future-work, an exploration for creating fault-tolerant global-view distributed data structures via checkpointing.

The Chapel Programming Language

Chapel is a programming language designed for productive parallel computing on large-scale systems. Chapel's design and implementation have been undertaken with **portability** in mind, permitting Chapel to run on multicore desktops and laptops, commodity clusters, and the cloud, in addition to the high-end supercomputers for which it was designed. Chapel supports a multithreaded execution model via high-level abstractions for **data parallelism**, **task parallelism**, **concurrency**, and **nested parallelism**.

Non-Blocking Algorithms

An algorithm is called non-blocking if failure or suspension of any thread cannot cause failure or suspension of another thread. A non-blocking algorithm is **lock-free** if there is guaranteed system-wide progress, and **wait-free** if there is also guaranteed per-thread progress. These algorithms use **atomic read-modify-write** primitives that the hardware must provide, the most notable of which is **compare and swap** (**CAS**). These algorithms, especially lock-free algorithms can prove to be great alternatives to our traditional blocking algorithms. Lock-free algorithms however, are prone to the **ABA problem**, which occurs during synchronization, when a location is read twice, has the same value for both reads, and 'value is the same' is used to indicate 'nothing has changed'. Hence, this needs to be taken care of while implementing any lock-free algorithms.

AtomicObjects Library

The Chapel programming language lacks an internal garbage collector. The first step, hence, is enabling scalable memory reclamation techniques, such as Epoch Based Memory Reclamation System. This requires atomic objects support. Chapel however only has atomic support for booleans and integers. Hence, as the first step, I created an Atomic Objects library [2] for Chapel, which **enables atomic operations** on any *unmanaged* lifetime class object. Atomic operations both with and without **ABA protection** have been implemented. ABA protection is implemented using tagged state reference counting using x86-64's **CMPXCH16B** instruction. The library also supports supports global atomics via **Network RDMA Atomics** on Cray XC systems on Cray's Aries network. Atomic functions implemented are: read, readABA, write, writeABA, compareAndSwap, compareAndSwapABA, exchange, and exchangeABA.

Epoch Based Memory Reclamation

Epoch Based Reclamation [1] is a lock-free memory reclamation algorithm. It uses grace periods, fuzzy barriers that it maintains to prevent reclamation of objects. There are 3 epochs. Each thread maintains its own **local epoch** while there is a system wide **global epoch**. The difference between the global epoch and an active thread's local epoch cannot exceed 1.

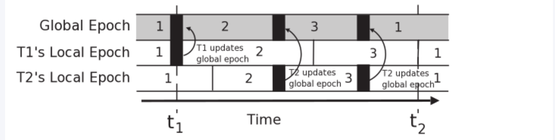

Figure 2:Epoch Based reclamation. Thin solid lines show updates to a thread's local epoch, and thick solid lines show updates to both a local epoch and the global epoch.

Figure 2 shows how epoch based reclamation allows for safe memory reclamation by tracking epochs. When a thread wants to enter a critical section, it **pins** to the epoch manager. By doing so, it is marked as active/executing and its local epoch is updated to the global epoch. Similarly, when a thread exits a critical section it unpins, and it is marked inactive. The manager maintains **limbo lists** for each epoch, which store the deleted objects. To reclaim objects, the manager first tries to announce a new epoch by **advancing the global epoch**. A new epoch can be announced only when all the pinned tasks are in the same epoch as the global epoch, else the attempt is aborted. If advancement is successful, then it is safe to reclaim objects deleted in **e-2** epoch, as there is no thread in that epoch. Hence, the objects from that limbo list are reclaimed.

Some alternatives to Epoch Based Reclamation are Quiescent State Based Reclamation, Hazard Pointer Based Reclamation, Lock-free Reference Counting, which was earlier used to manage shared memory objects in Chapel. However, none of these schemes match the performance and scalability of Epoch Based Reclamation [6].

Treiber's Stack, Michael & Scott's Queue

To prove the working of epoch manager, I implemented a modified version of **Treiber's Lock-free Stack** [7] and **Michael & Scott's Lock-free Queue** [9], which makes use of Epoch Manager. Both these algorithms predominantly requires ABA protection to perform operations. Without ABA protection, data race conditions may arise. However, ABA protection requires 128 bit atomic operations, which can be highly **expensive**. This is where Epoch Manager comes in to play. It defers the deletion of objects, hence preventing data race conditions. As a result, a data structure employing Epoch Manager does not require ABA protection. This results in significant improvement in performance, which is clearly visible from benchmarks.

Results

The implementation so far has been highly successful. All the libraries (AtomicObjects, EpochManager) and data structures (LockFreeStack, LockFreeQueue) have been **rigorously tested for correctness**. All the libraries and data structures implemented have been **merged into the Chapel language's repository** [2][3][4][5]. The modules are available for use from Chapel 1.20.

To test the performance, I implemented and optimised various versions of Michael & Scott's Queue and ran the same **performance benchmark** on each version. The benchmarks were run on a **Cray XC-50 supercomputer** with **Intel's Broadwell (BW) processors**. The processor supports **44 chapel tasks parallely**. In each benchmark there were 44 tasks, which performed 16M enqueue and dequeue operations per task. Following are the benchmark results:

Data Structure	Time Taken
Epoch Managed MS-Queue	37.7992s
Two-Lock (TATAS) Queue	69.8408s
Two-Lock (TAS) Queue	148.661s
Recycled (ABA) MS-Queue	154.375s

Table 1:Performance Benchmark of various implementations of locked and lock-free Queues

It is clear from the above benchmark results that the Epoch Managed queue is **atleast 2 times faster** than any other optimised locked version. It is clear that ABA protection comes with a huge toll; the performance speedup from a recycled queue is almost **5 times**. These benchmarks prove the performance viability and scalability of lock free data structures and Epoch Manager.

Future Work

Now that EpochManager has been implemented and merged successfully, its viability proven, the next step will be to create an efficient **distributed adaptation** of the memory management system. This is a challenging task as epoch manager was predominantly designed for shared memory. Finally, as a proof of concept, I plan to implement a **distributed data structure** which employs the distributed memory manager. This would mark the completion of the project.

References

[1] Keir Fraser.
Practical lock-freedom.
Technical report, University of Cambridge, Computer Laboratory, 2004.

[2] Garvit Dewan, Louis Jenkins, Cray Inc.
Atomicobjects - chapel documentation 1.20, url: https://chapel-lang.org/docs/modules/packages/atomicobjects.html, 2019.

[3] Garvit Dewan, Louis Jenkins, Cray Inc.
Epochmanager - chapel documentation 1.20, url: https://chapel-lang.org/docs/modules/packages/epochmanager.html, 2019.

[4] Garvit Dewan, Louis Jenkins, Cray Inc.
Lockfreequeue - chapel documentation 1.20, url: https://chapel-lang.org/docs/modules/packages/lockfreequeue.html, 2019.

[5] Garvit Dewan, Louis Jenkins, Cray Inc.
Lockfreestack - chapel documentation 1.20, url: https://chapel-lang.org/docs/modules/packages/lockfreestack.html, 2019.

[6] Thomas E Hart, Paul E McKenney, Angela Demke Brown, and Jonathan Walpole.
Performance of memory reclamation for lockless synchronization.
Journal of Parallel and Distributed Computing, 67(12):1270--1285, 2007.

[7] Danny Hendler, Nir Shavit, and Lena Yerushalmi.
A scalable lock-free stack algorithm.
In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 206--215. ACM, 2004.

[8] Lawrence Berkeley National Laboratory.
Upc++ v1.0 programmer's guide, url: https://upcxx.lbl.gov/docs/html/guide.html, 2019.

[9] Maged M Michael and Michael L Scott.
Simple, fast, and practical non-blocking and blocking concurrent queue algorithms.
Technical report, ROCHESTER UNIV NY DEPT OF COMPUTER SCIENCE, 1995.