Heterogeneous Parallelism Mini Project

Title: Implementation of High Performance, Lock Free and Concurrent Data Structures

Team 1
Madhav Jivrajani (PES1201800028)
M S Akshatha Laxmi (PES1201800130)
Sparsh Temani (PES1201800284)

Introduction/Background

Spin Locks: Reloops till the CAS operations returns True

- Equivalent of acquire lock: **while(!lock.CAS(0, 1));**
- Equivalent of lock release: **lock = 0**
- While the thread is in the critical section, lock is set to 1, once it exists, one of the waiting threads sets the lock to 1 again using the CAS operation and enters the critical section.
- CAS (Compare and Swap) : Executed as a single instruction on the CPU (atomic)
- Can have severe performance implications as only thread can enter the critical section at once.

Issues with Spin Lock based Concurrent Data Structures

Reloops on Progress and Non Progress

- Uses additional CPU cycles even when no progress is made
  - When a thread with an acquired lock gets preempted, another thread which gets scheduled will waste CPU cycles

- If the thread with an acquired lock dies, there is no progress made
  - If a thread holding a lock dies in the middle, other threads waiting for the lock cannot proceed

- Priority inversion
  - A process with a lower priority is holding a lock that is required by a process of a higher priority

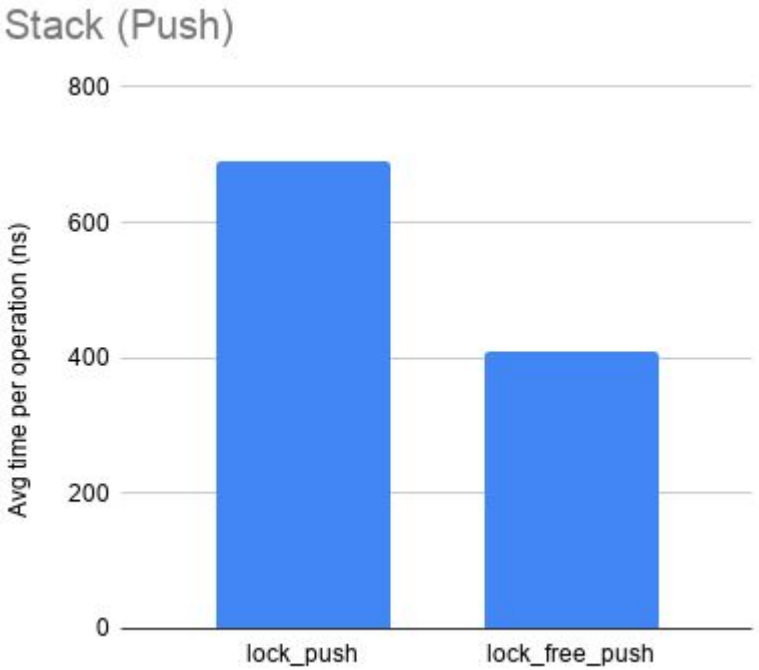
How Lockless Programming is useful ?

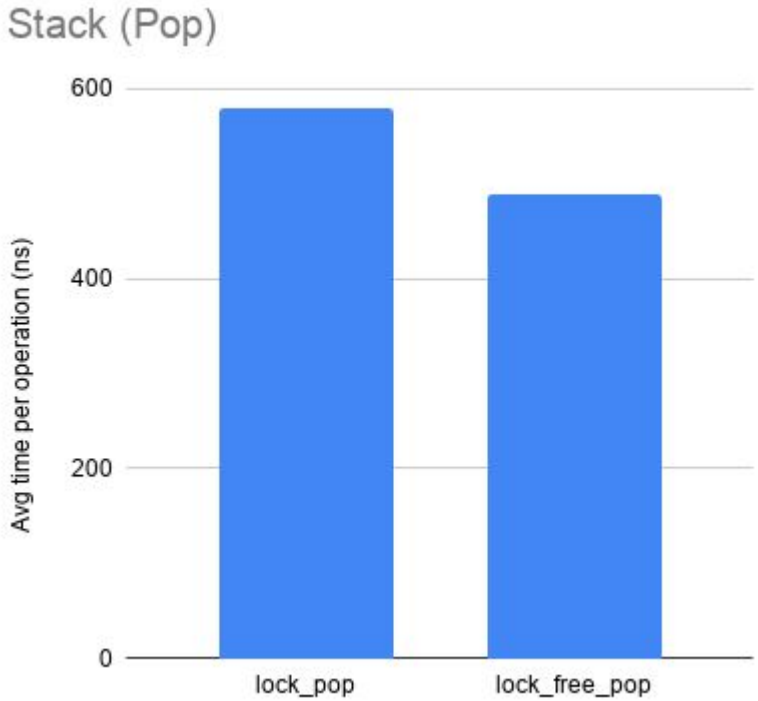
- Lockless programming makes sure at least one thread is making progress at any given instant (assuming it is scheduled by the OS).
- When the thread is preempted, another process can make progress since no clock is acquired
- Since no lock is held in any of the threads, if a thread dies, only the operations assigned to the thread do not execute where as the rest of the system can proceed as it is
- Lockless Programming guarantees system wide progress

Implementation

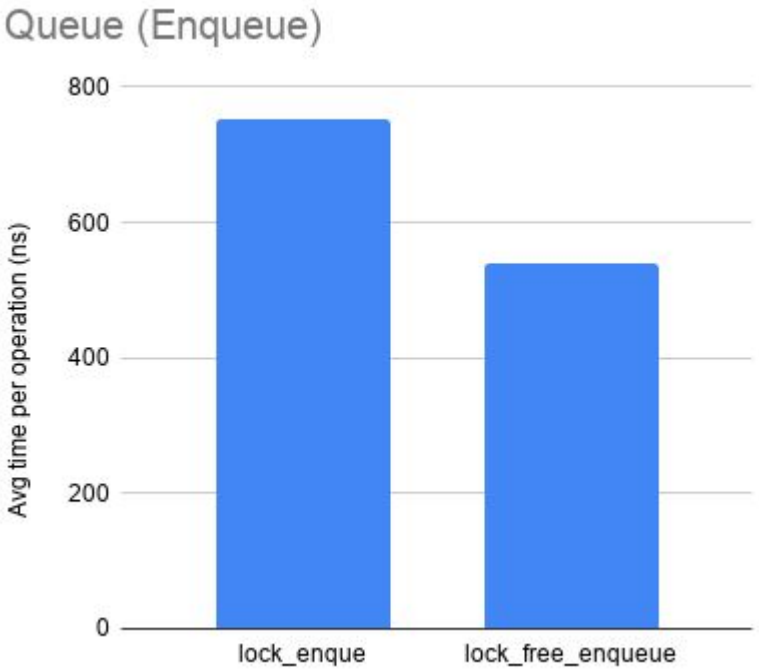
- Language used: Go
- Lock and equivalent lock-free implementations benchmarked using Go testing package, and checked for race conditions using Go's race detector.
- Call Graphs generated using `pprof` for the lock-free and its respective lock-based counterparts
- Data Structures Implemented so far:
    - **Stack**: Push, Pop, Pee
    - **Queue**: Enqueue, Dequeue
    - **List**: Insert, Delete

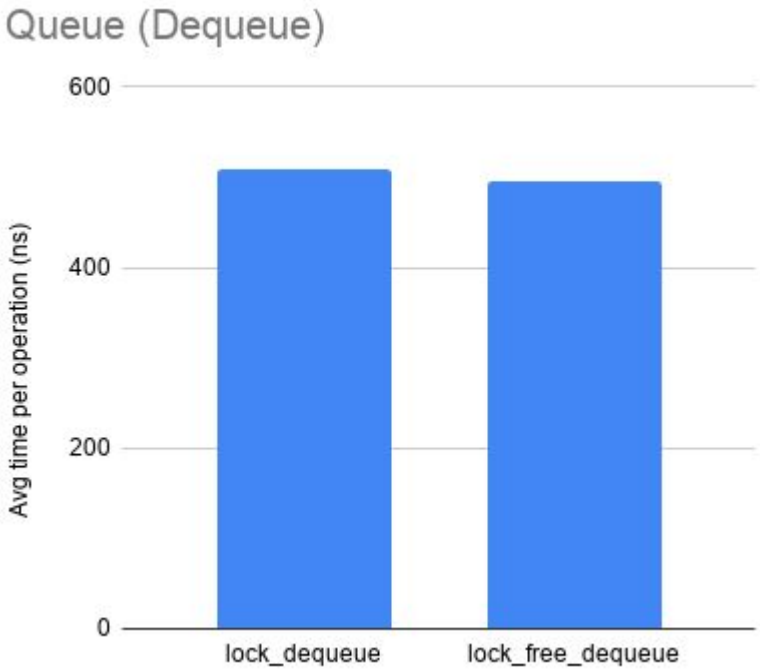
Results - Stack



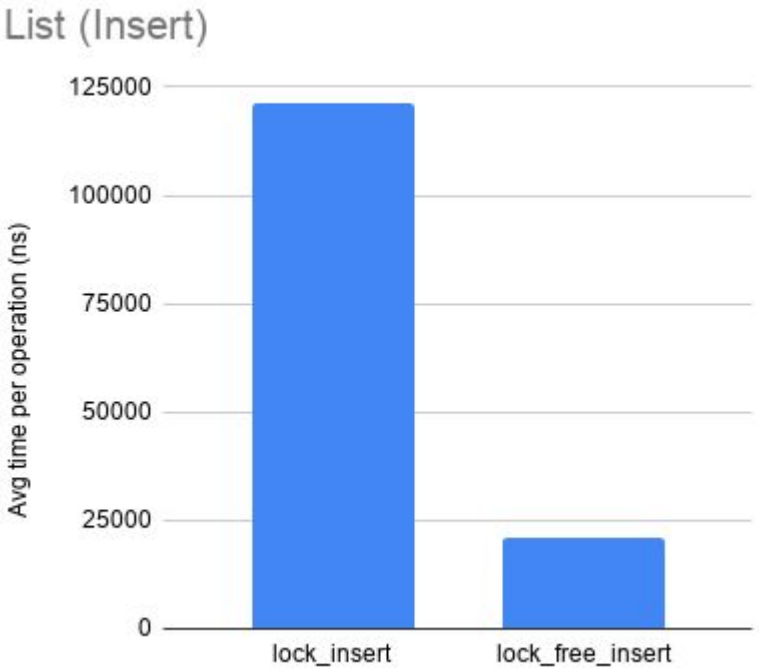


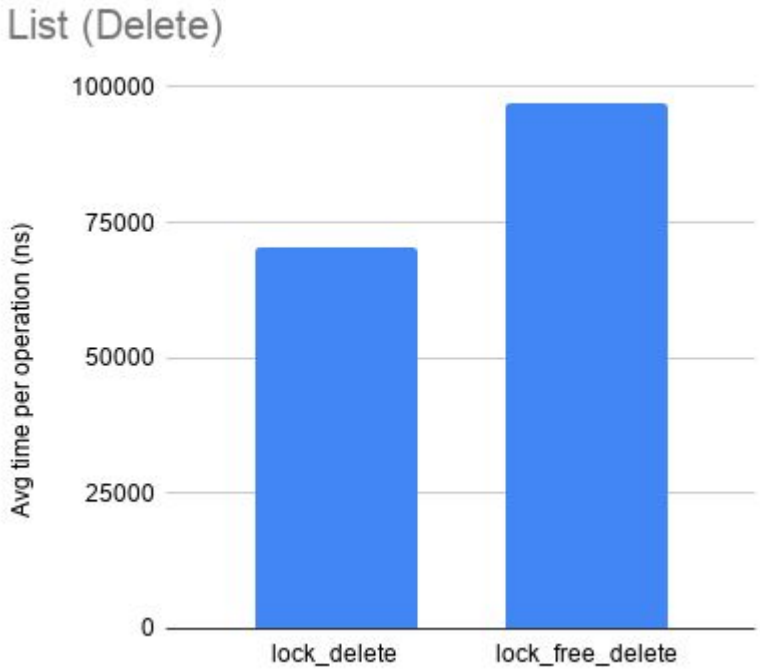
Results - Queue





Results - List





Issues with current Implementation

ABA problem

- The Lockless implementation relies on using the Compare and Swap Operation and making sure the node values have not changed before and after the CAS. This can be an issue when a pointer address gets deallocated and again allocated by the program.

Solution:

- Add a counter to keep track of the number of references to the pointer
  - Still can cause issues if a new pointer created reaches the same number of references
- Deallocate the pointer only when the active reference count reaches zero

References

Papers:

Implementing Lock Free Queues, *J. D. Valois, Dept. of CSE, Rensselaer Polytechnic Institute.*

A Pragmatic Implementation of Non-Blocking Linked-Lists, *Timothy L. Harris, University of Cambridge.*

Videos:

Introduction to Lock-free Programming *- Tony van Eerd, NDC Techtown*

Lock Free Programming *- Herb Slutter, CppCon 2014*

Future Scope

- Try and implement 2 stage deletion in lists
- Perform further profiling to check if improvements via avoiding false-sharing can be achieved.
- Further implementation:
  - Maps
  - Skiplists
- Benchmark for read + write

Thank You