CS 6301: Special Topics is Computer Science- Introduction to Multicore Programming Section 001
Programming Assignment 3 Report

Submitted By: Rohit Bhattacharjee (rxb151030) and Suraj Poojary (ssp151830)

a) Experiment details:

We used the TACC stampede cluster to conduct our experiments on the different lock implementations. Here is the system configuration of the system:

- OS: Linux
- Architecture: x86_64
- CPU op-mode(s): 32-bit, 64-bit
- Byte Order: Little Endian
- CPU(s): 16
- Thread(s) per core: 1
- Core(s) per socket: 8
- Socket(s): 2
- NUMA node(s): 2
- Vendor ID: GenuineIntel
- CPU family: 6
- Model: 45
- Stepping: 7
- CPU MHz: 2701.000
- BogoMIPS: 5399.22
- Virtualization: VT-x
- L1d cache: 32K
- L1i cache: 32K
- L2 cache: 256K
- L3 cache: 20480K
- NUMA node0 CPU(s): 0-7
- NUMA node1 CPU(s): 8-15

We used Java (SE 8) as the programming language.

Range of values

Number of Threads: 1 to 32
Key space size: 1,000 to 1,000,000

Distributions:
(a) read-dominated: 90% search, 9% insert and 1% delete.
(b) Mixed: 70% search, 20% insert and 10% delete.
(c) write-dominated: 0% search, 50% insert and 50% delete.

Number of runs for each data point: 10

For every run, the tree is initially populated to half the key space size before performing the operations and recording the experimental results.

b) Correctness Validation:

In order traversal of the BST was used to validate the tree. During the traversal, the below 2 checks were performed at each node.
Check 1:
Left child's key is less than or equal to its parent's key.
Right child's key is greater than its parent's key.

Check 2:
Each external node has no children.

The algorithm exits with a message "Invalid BST" if any one of these checks fail.

We have also run both the implementations for the below input scenario as suggested by the professor. A small key space size and a large number of instructions ensure that there is large contention generated. Both implementations passed this check.

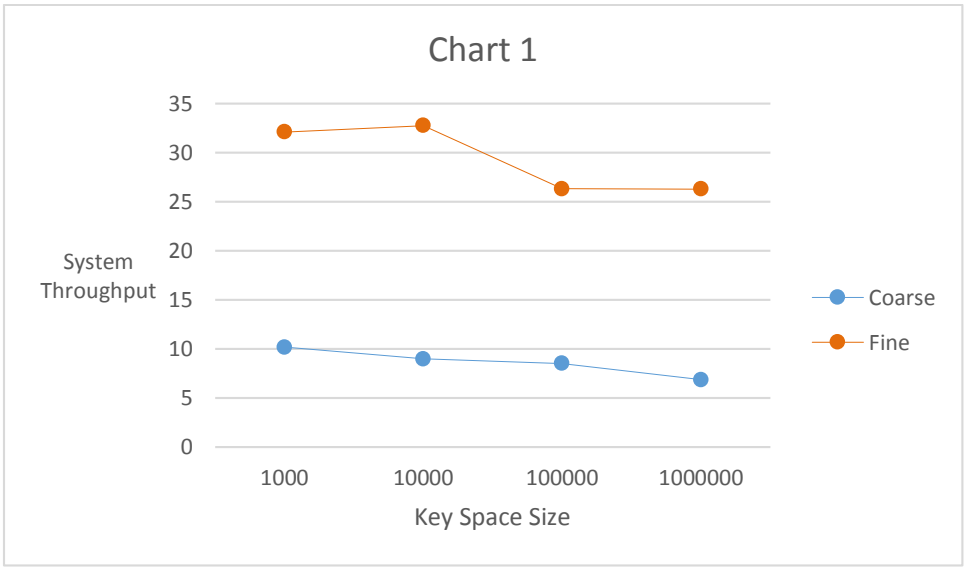
#instructions/thread: 10,000,000
Key space size: 10
Distribution: mixed
#threads: 32

c) Graphical Data d) Discussion of Results:
Three graphs have been plotted for the three system parameters. Each graph compares the 2 implementations for a specific setup.

As expected, fine grained lazy synchronization approach performs way better than coarse grained approach in terms of system throughput.

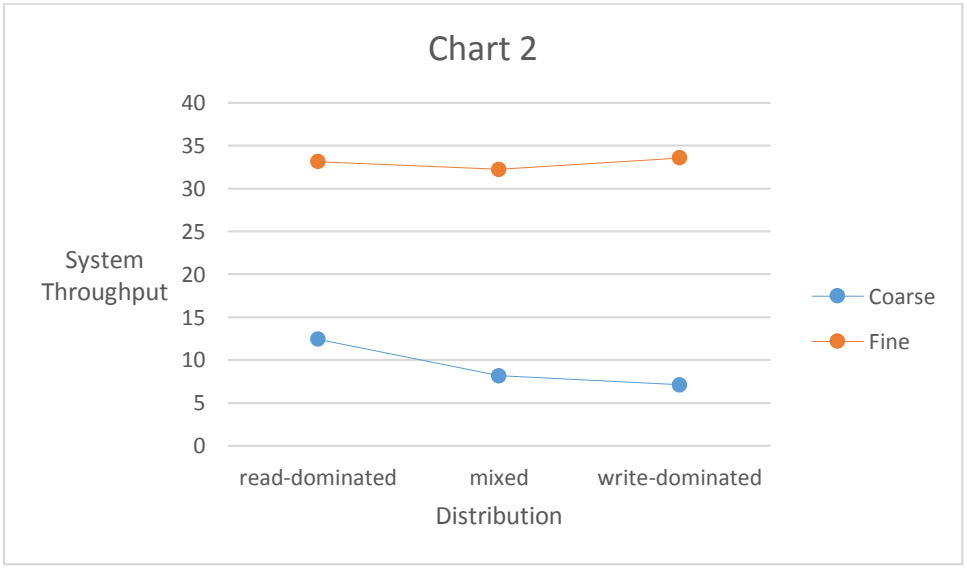
Note: Throughputs have been scaled down by 100 to represent appropriately in the graphs.

Graph 1: (For fixed distribution and #threads)
Distribution: Mixed (70% search, 20% insert and 10% delete.)
#Threads: 16



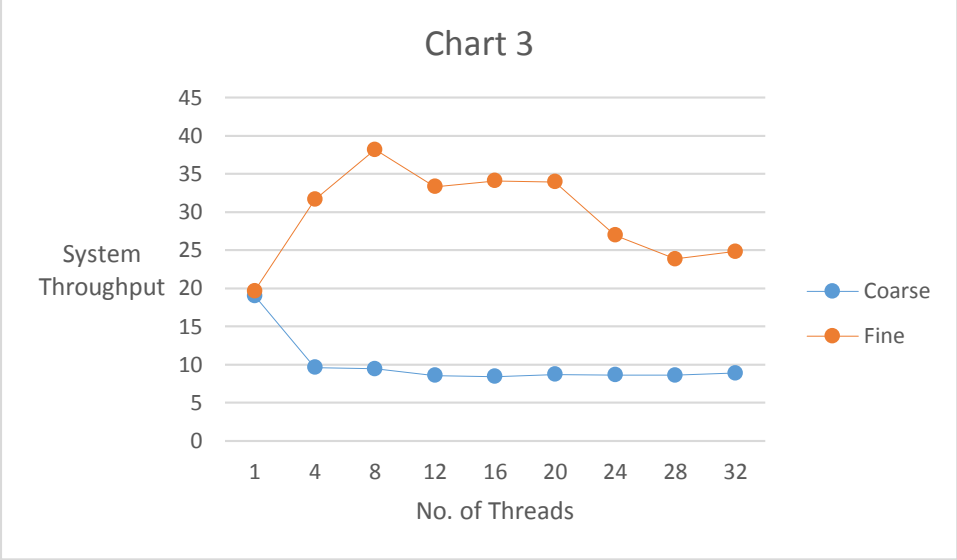
For both fine and coarse grained approach, the system throughput decreases with increase in key space size. With more key space size, the tree grows larger and searches (insert and delete also perform search) take lot more time, hence the drop in throughput.

Graph 2: (For fixed key space size and #threads)
Key Space Size: 100,000
#Threads: 16



Going from read-dominated to mixed to write-dominated distribution, the probability of read-write and write-write conflict increases. This causes the throughput to decrease in case of coarse grained approach but does not cause much variation in case of fine grained lazy approach.

Graph 3: (For fixed key space size and distribution)
Key Space Size: 100,000
Distribution: Mixed (70% search, 20% insert and 10% delete.)



The throughput in case of coarse grained approach remains fairly constant with increase in threads after the initial expected drop for 4 threads. (There is no contention for 1 thread hence both fine and coarse grained have almost the same throughput). Fine grained lazy approach on the other hand has increasing throughput up to a certain number of threads. After that, the throughput gradually decreases for further increase in threads.

4 thread data point:
Coarse grained approach: Throughput decreases as threads have to wait on other threads for acquiring lock on the tree even if they are operating on different sections of the tree.
Fine grained lazy synchronization approach: Lazy approach is capable of accommodating multiple requests together, hence the increase in throughput.