Siddharth Kannan, Jayanth Bharadwaj
Andrew ID: siddhark, jbharadw

Parallel Computer Architecture and Programming (15-418/618)

Project Checkpoint

So far, we have went ahead and implemented a fine-grained implementation of the hash table, by locking on each individual bucket within the hash table instead of on the entire table itself. We associated a regular pthread lock with each linked list within the hash table and are locking on all reads and writes. This effectively ensures that we never traverse a broken linked list or find an element in the linked list that doesn't exist. We also went ahead and implemented a lock free implementation mentioned in the paper here. A brief summary of the method follows.

The main problem with implementing a deletion operation in a lock-free linked list is that the successor of the node being deleted must not change while the delete is happening. This may happen for instance while inserting a node after the node to be deleted. What happens is that the predecessor of the node to be deleted ends up pointing to a stale successor. In order to avoid this, Harris[1] proposed a two-step approach wherein they mark the node to be deleted in the first step. At this stage the node is said to be logically deleted and no other nodes can be inserted after this node. In the second step, they physically remove the node from the list. While this method works well, Fomitchev[2] pointed out their implementation can be inefficient for the following reason. Suppose a node has to be inserted after node X. At the same time X has been marked for deletion. This means that the node cannot be inserted after X now. So in the implementation of Harris the algorithm would restart the search for a suitable position to insert the node from the start of the list. Fomitchev suggests that each node have an additional attribute called 'backlink' which will point to the predecessor of a node if the node is marked for deletion. By doing so, we do not have to restart the search from the start of the list every time. We can traverse the backlink pointers till we reach an unmarked node and resume search from there. But the pitfall here, as highlighted by Fomitchev is that we can end up repeatedly traversing chains of backlinks from left to right while finding a position to insert. In order to avoid this, they introduce an additional flag bit which ensures that long chains of backlinks do not form. How this works is that when a node is to be deleted the predecessor is first flagged. Flagging means that the next pointer of the flagged node cannot be changed and the node cannot be marked for deletion. Once flagging is successful, the node is marked and deleted as before in two steps. Because the predecessor cannot be marked for deletion, chains of backlinks cannot form. In our implementation, we embed the flag and mark bits in the next

[1] https://timharris.uk/papers/2001-disc.pdf
[2] http://www.cse.yorku.ca/~ruppert/papers/lfll.pdf

pointer of the node and use simple bit masking to extract these bits. We are able to do this because the dynamic memory allocated to nodes is at least 4-byte aligned.

We are currently doing fine with regard to our planned schedule. We have finished both the fine grained implementation and the lock free implementation as outlined on our schedule. I'm not sure we'll be able to write a resizing implementation of the hash tables (nice to have), but we should still definitely be able to finish the transactional model of the hash table. We may need access to machines with hardware support for transactions to continue down this route, since the software overhead associated with the model may just not be worth it. We aim to have finished the transactional model within the next 2 weeks and then going on to gather results on the performance and comparing them.

At this time we have no preliminary results to show. We plan to show some graphs outlining the performance differences in these various implementations of a concurrent hash tables at the poster session. We still haven't researched to implement the transactional version of the concurrent hash table and need to do some literature review. I'm mainly worried about the implementation of resizing within the hash table, which may prove to be very difficult. We also don't know how to approach memory reclamation in the lock free implementation, since this is bound to be very problematic. We know that hazard pointers would help, but we still need to worry about any race conditions that arise from this.

Schedule:-

19 Nov - 21 Nov	Transactional memory - Jayanth, Siddharth
22 Nov - 25 Nov	Transactional memory - Jayanth Testing Suite - Siddharth
26 Nov - 29 Nov	If TM complete, incorporate memory reclamation (125%) and expand table functionality (150%), Testing - Jayanth, Siddharth
30 Nov - 3 Dec	Performance testing - Jayanth, Siddharth
3 Dec - 6 Dec	Performance testing, Report - Jayanth, Siddharth
7 Dec - 10 Dec	Report - Jayanth, Siddharth
11 Dec - 14 Dec	Buffer in case something goes wrong