Parallelizing DSW Trees

Willard Stanley
Department of Computer Science
University of Central Florida
COP6616

Abstract—A parallelization of the Day-Stout-Warren algorithm for balancing binary trees. As its input, this algorithm takes an arbitrary binary tree and returns an equivalent tree which is balanced so as to preserve the $\theta(\log(n))$ lookup time for elements of the tree. The sequential Day-Stout-Warren algorithm has a linear runtime and uses constant space. This new parallelization of the Day-Stout-Warren algorithm attempts to do the same while providing a speedup which is as near as possible to linear to the number of processing elements. Also, ideally it should do so in an online fashion, without blocking new reads, inserts, and deletes.

I. INTRODUCTION

A. Motivation

The rapid adoption of multicore CPU architectures in the commodity computer market has dramatically changed the landscape of available CPU resources available to modern programmers. The year over year increase in sequential performance is beginning to cool. No longer can programmers rely on sequential implementations simply getting faster on subsequent generations of hardware. Since we are no longer able to rely on sequential speedup, we must exploit the now nearly ubiquitous multicore machines in order to ensure performance continues to increase as time progresses.

In order to ensure that programming parallel for parallel architectures is feasible we must be able to parallelize existing algorithms and data structures so that programmers can use familiar constructs. There is a large wealth of well known sequential algorithms that are ripe for parallelizations.

If we can successfully encapsulate the complexity introduced by the common pitfalls of parallel data structures such as race conditions, deadlock, starvation, and read-modify-write interleavings, we can ensure creating programs which take advantage of multicore architectures remains within reach.

B. Problem Definition

One such well known data structure is the binary tree. A binary tree allows lookups and insertions in $\theta(\log(n))$ provided that the structure remains balanced. There are many known sequential algorithms for ensuring that a tree remains balanced. Some of these sequential algorithms, such as AVL trees [1], ensure that a tree is balanced after every insert or delete operation. In doing so, the cost of ensuring balance is added to every insert and delete operation. It follows that this increased cost also increases the likelihood of contention in a parallel system with many inserts and deletes. For this reason, this approach focus on a balancing algorithm which is called periodically to amortize the cost of rebalancing over many inserts and deletes and thus limit the contention increase. Specifically, this implementation will parallelize the Day-Stout-Warren (DSW) tree balancing algorithm. [11]

C. Related Work

We will begin with a brief overview of the algorithm which is going to be parallelized. The DSW algorithm provides balancing of a binary tree by only balancing periodically. This allows the cost of rebalancing to be amortized across many insertions and deletions. During rebalancing, the algorithm will perform a tree_to_vine operation. This operation will convert the unbalanced tree into a series of linked nodes whose left child is null. It will do so using a series of right rotations. Once the unbalanced tree has been fully converted into a vine, the algorithm will perform a vine_to_tree operation. This is a series of left rotations which will convert the vine into a balanced tree.

The existing literature on parallelizing the DSW algorithm specifically is a bit sparse, however, there are many other parallel balancing algorithms which can be drawn from. One such balancing algorithm alluded to earlier uses AVL trees and read/write locks [3]. When a writer comes to insert or delete, it will lock the node it is inserting into along with its parent. This is primarily to ensure that other writers are excluded. Readers are allowed through for inserts and most delete scenarios. If a rotation is needed, then the writer must lock out other readers as well to avoid changing the tree structure out from underneath a reader. Given that rotations can occur after any insert or delete operation, and that a deletion of the root element requires a recheck and possibly rotation of, it is easy to see the potential for contention in this approach.

A later approach by Larsen [7] attempts to address this requirement by relaxing the balance requirement of the AVL tree. In this approach rather than rebalance immediately, a node will be tagged when it is out of balance. As inserts and deletes accumulate, the tags will continue to be updated to indicate how far out of balance the tree is. Once the tags cross a certain threshold, the rebalancing operation will be performed. This approach shares the advantage of amortization of balance operations with the DSW algorithm. However, a successful parallelization of the DSW algorithm will provide the additional advantage of an improved tree height due to a more restrictive balance definition [9].

II. APPROACH OVERVIEW

One of the primary goals of this project is to ensure that we end up with a usable data structure. Before even beginning the DSW algorithm this implementation I will need a method to ensure that the base binary tree operations (insert, delete, search) complete safely. In order to ensure correctness in a parallel implementation of a binary tree, we must ensure that each operation has an atomic view of the tree. One traditional approach to atomicity guarantees is to use a lock to ensure that only one operation may execute at a time. With a course
grained locking approach, the entire data structure is locked for each operation. This has the effect of serializing all access to the data structure and can be quite detrimental to performance. To avoid this serialization one may opt for a fine grained locking approach. Since there are many existing implementation and the basic binary tree operations are outside of the scope of this work, this work makes use of techniques from an existing implementation from Bronson, et al. [2] To simplify implementation, the optimistic locking and versioning of nodes was dropped in favor of a simpler hand over hand approach.

A. Insert Concerns

Even with insert, delete, search, and rotate properly accounted for, there is still the concern of interrupting the vine_to_tree and tree_to_vine operations. During the tree_to_vine operation, the algorithm begins at the root and through a series of rotations converts the tree in a top down fashion. The potential hazard comes in when an insert is allowed to occur into the vine section that has already been completed. If an insert of value X occurs where vine_tail > X > vine_root it will end up being a left child of one of the intermediate nodes, thus converting the vine back into a tree.

1) Do nothing: This solution is only possible if we can prove that vine_to_tree will still produce a valid tree if it is given a tree as an input. This as it turns out is quite straightforward. Given that a tree rotation is atomic, that rotation will always produce a valid binary tree [10]. Since both the vine_to_tree and tree_to_vine operations are merely a series of rotations, the final output of both these operations must also be a tree, as long as each individual rotation is atomic. Just because we can get away with this, does not mean that this is the best approach. We will also need to prove that the resulting tree is no more unbalanced than if vine_to_tree had completed before the insert came along.

2) Delayed Inserts: One possible way around this insert hazard is to hold inserts while a tree_to_vine operation is in progress. Inserts would be queued into an intermediate data structure (possibly another tree) to be replayed after the tree_to_vine operation is complete. This approach would guarantee the safety of the vine, however there may be some drawbacks as well. First of all, this intermediate data structure would need to be read on any subsequent read, potentially doubling traversal time. Secondly, if the inserts were frequent enough, it might be possible to accrue enough unbalance in the tree so as to need to rebalance it again immediately.

3) Modified Inserts: Presuming that we could accurately detect if we’re inserting into a vine, we could modify the insert behavior to preserve the vine property. At this point it would essentially be doing a linked list insert along the right child path of the vine. The advantage of this approach is that we will not accrue any further unbalance, and we will not have violated the vine property. We must be able to detect if the vine_to_tree has converted the node we are on back into a tree.

B. Delete Concerns

While performing tree_to_vine, deletes do not share the same concern as inserts. When deleting an item on a vine, there is no chance of unintentionally converting the vine back to a tree. Thus when a delete occurs on a vine, we can simply perform a normal binary tree delete. The deleted node, by definition of vine, will not have a left child and thus no promotion is necessary. As for vine_to_tree, the concern is that we may end up modifying the size of the tree during the operation which may effect the balance of the tree. Also, if the delete occurs on a section of the structure which has already been converted from vine_to_tree, there is the potential that we may need to promote the leftmost right child, or the rightmost left child, which is quite difficult to ensure atomicity for. To get around these hazards, while using the vine property to our advantage, we can use a virtual deletion strategy and let the tree_to_vine perform the actual deletion as it goes along.

III. IMPLEMENTATION DETAILS

To facilitate atomicity of operations, each node element contains the following items in addition to the value and children pointers:

- isVine - A flag to indicate that an in progress tree_to_vine operation has already completed for this node.
- isDeleted - A flag to facilitate virtual deletion. If this flag is set, the value is virtually deleted.
- nodeLock - A lock to allow for hand over hand locking during tree traversal and manipulation. Since lock type is outside of the scope of this project, an off the shelf spin lock is used from Intel’s Thread Building Block library.

These extra variables will allow us to construct the following atomic operations.

A. Atomic Operations

1) search: The search operation will use hand over hand locking to traverse the tree and find the appropriate value. If a value is found, the isDeleted flag will be checked.

2) insert: The insert operation is similar to a traditional binary tree insert with hand over hand locking. The insert will traverse the tree using hand over hand locking to ensure it safely reaches the appropriate parent node for the new element. For the normal case once the parent node has been found and locked, insert will create a new node and insert it into the tree. In the case that it finds the parent node to have the isVine flag set, insert will only place the new node as a right child of the parent node. The operation will have to look ahead one element during the traversal of the vine as the parent will be the node such that parent → value < newValue < parent → rightChild → value. It must lock both the parent and the right child to ensure that another insert does not invalidate the above check. By adding this modification, we can ensure that another insert to a vine will not transform the vine back into a tree.

3) delete: The delete operation uses hand over hand locking to traverse the tree as well. When the element to be deleted is found, delete will not actually remove the item. It will merely set the isDeleted flag and leave the node in the tree. If it were to try to remove the node from the tree it may interfere with in progress vine_to_tree operations. Furthermore, if the
deleted node has both left and right children, it would need to promote the leftmost right child, or the rightmost left child. In order to do that it would need to lock all intermediate nodes to ensure that it was picking up a valid value. By marking a node as deleted, we can push the actual removal operation onto the *tree_to_vine* operation.

4) rotate: The *rotate* operation will lock the parent, the rotation node, and the rotation node’s right or left child depending on which direction that the node is being rotated. It will then rotate the nodes such that the rotation node becomes the child of the child node and the child node becomes the child of the parent. It will then release all three locks.

B. Aggregate Operations

1) *tree_to_vine*: The *tree_to_vine* operation will do a locked traversal of the tree, as with all other operations. At each node, it will perform a right rotation and repeat the process until the current node has no left child. Once there is no left child, it will check the current node for deletion. If it finds that the current node has the *isDeleted* flag set, it will remove the node. If the current node is not deleted, it will set the *isVine* flag. It will then lock the next right child and continue down the tree.

2) *vine_to_tree*: The *tree_to_vine* operation will do a locked traversal of the vine, performing a left rotate on each node. After the rotation is complete, it will set *isVine* for each left child as it goes.

C. Parallelization Strategy

With each operation properly protected, all that remains is to divide up the work of the balance operation in such a way as to facilitate parallel execution. In this implementation, a pool of worker thread waits on a concurrent queue. Once again, this is an off the shelf concurrent queue from Intel’s Thread Building Block library. Each parallel operation will divide up the work to be done and place it in the queue to be executed by the worker threads. Each batch job contains a pointer to the node to be operated on, the operation to be performed, and a pointer to the lock of the parent node. In the case of a *vine_to_tree* operation, the batch job will also contain the number of nodes to run the compress operation on.

1) *tree_to_vine*: When allocating work for the *tree_to_vine* operation, we can take advantage of the existing tree structure to ensure that work is divided evenly among the worker threads. Since the *tree_to_vine* will perform a series of right rotations, splitting up work down the right side of the tree will allow threads to perform their rotations without colliding with one another. This strategy has the hazard of poor performance on a tree which is poorly balanced at the root. To combat this possibility, we also divide the work on all the right children of root’s left child. This provides chunks of work arranged from large to small.

2) *vine_to_tree*: When splitting up the work for the *vine_to_tree* operation, we can allow the threads to follow each other in a cascade down the vine. The moment that a thread has acquired the lock on the root node, it can signal another thread to begin compression on the root. Since we are using hand over hand locking, we do not need to fear threads overtaking one another as they walk the vine.

IV. EXPERIMENTAL RESULTS

A. Methodology

In order to test this parallel implementation, a series of randomized input trees were created. Each of these trees were then balanced using the sequential DSW algorithm [11]. Since the input sequence is random, this process was then repeated 5 times to ensure accuracy for the average case. This process was then repeated using the parallel DSW algorithm with the same randomized sequence of inputs for each tree. Given that this implementation allows for parallel inserts, these trees will be equivalent, if not identical. Also, the size of the tree was varied to demonstrate the performance of this algorithm across different tree sizes. Trees of size 1 million, 10 million, and 50 million were used to properly exercise all threads. Each test of the parallel DSW Tree was run with 1, 2, 4, and 8 threads. A synthetic benchmark was used which performed the following sequence of operations:

- \( N \) random inserts to build the tree.
- \( N \) searches on the randomized tree to serve as comparison with the balanced tree.
- A *balance* operation to exercise both *tree_to_vine* and *vine_to_tree*.
- \( N \) searches on the randomized tree to serve as comparison with the balanced tree.

This implementation was compiled with GCC 4.4.4 against the Intel Thread Building Blocks library v4.2. It was run on a Quad Core Intel E5620 with hyperthreading enabled.

V. CONCLUSION

First of all, you will notice a slight decrease in performance from the sequential DSW algorithm to a single threaded execution of the parallel DSW algorithm. This is to be expected given the overhead of the hand over hand locking used in the parallel implementation. From the run data for two or more threads we can see right away a performance increase over the sequential version of DSW Tree. As to be expected, the simple operations of *search* and *insert* see the largest performance
impact. You will also notice that the performance improves with eight threads, but not with the same margin as did with two or four threads. This is likely due to the fact that this processor uses hyperthreading and thus with eight threads, two threads will be running on each core. A machine with eight or more cores may perform better with eight threads.

As for the DSW balancing algorithm itself, we can see that vine_to_tree was the better performing of the two components. It is also the longer running of the two, so my implementation targeted it for performance improvement first. The multithreaded parallel implementation achieved as high as 38% increase over the corresponding sequential version when using all eight threads. The tree_to_vine operation achieved a more modest 15% improvement in the best case. This is likely due to the random balance given to these trees in the test scenario. There may be room for improvement for the tree_to_vine if a better workload distribution could be found. Over all balance speedup, including setup and teardown of the work queue was 21%.

This approach does indeed improve the runtime of the DSW tree algorithm. Also, importantly, it provides thread safety for the other operations that may execute during a balance operation.

VI. Future Work

Given the few existing parallel implementations of the DSW algorithm, there is substantial room for future study of this topic. One potential area for improvement over the implementation presented here would be to replace the locking mechanism with an optimistic approach like the one described in [2] and the use of lock-free and wait-free synchronization techniques [4]–[6], [12], [13]. The use and performance of the presented DWS tree algorithm in a task-parallel environment [8] will also be explored in future work.

REFERENCES