A Symmetric Concurrent B-Tree Algorithm: Java Implementation

Elise Cormie

Department of Computer Science and Engineering, York University
4700 Keele Street, Toronto, Ontario, Canada, M3J 1P3

Abstract. In their paper entitled *A Symmetric Concurrent B-Tree Algorithm*, Vladimir Lanin and Dennis Shasha introduce a deadlock-free concurrent B-tree algorithm which preserves the symmetric properties of the B-tree and requires minimal locking. This paper presents a Java implementation of the algorithm. Its performance is compared to a sequential version, and a version similar to the earlier algorithm it was based on. It is tested for errors using randomized tests and Java PathFinder.

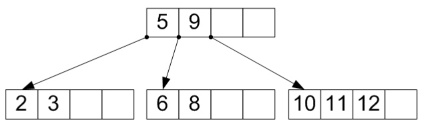
1 Introduction

1.1 About B-trees

B-trees are common tree-based data structures. They are normally used when large amounts of data must be stored and efficiently searched. They can be searched in logarithmic time, like other types of tree. However, unlike trees with only one item per node, B-trees have many items in each node which can be loaded into memory at the same time. This reduces the number of hard disc accesses required to search the tree.

A B-tree has an order, t, and each non-root node contains between t and $2t$ keys. Each key can have a child to its right and to its left. Children are shared between neighboring keys. Generally, left children contain keys smaller than the parent key, and right children contain larger keys. The tree should be symmetric, with all leaves on the same level.

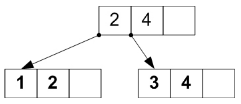
Example:



1.2 B-tree Variants

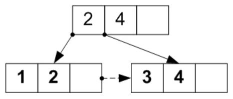
A *B+ tree* only stores keys in leaf nodes. Internal nodes contain numbers to organize and navigate to the required keys, but do not store the keys or data themselves.

Example:



A *B-link* tree is a concurrent tree based on the B+ tree. The idea was introduced by Lehman and Yao in [1]. It adds additional links between the nodes of a B+ tree, to allow processes to retrieve data that has been moved during write operations. This reduces the amount of locking required. The B-link tree is the basis of the algorithm discussed in this paper.

Example:



1.3 Restructuring the Tree

To use the language of the readers-writers problem, *search* will be referred to as a read operation, and *insert* and *delete* as write operations. After adding or removing a key, write operations may have to restructure the tree to preserve its symmetry and ensure the correct number of keys in each node. This restructuring can be done while descending the tree, as described in [2, Ch. 18]. In this case, the nodes must have an odd number of keys, $t - 1$ to $2t - 1$. More commonly, restructuring is done by ascending the tree and propagating changes upwards after the node that needs to be inserted to or deleted from is found. Trees that restructure in this manner have between t and $2t$ keys per node.

1.4 Previous Concurrent B-trees

Prior to Lanin and Shasha's work, there were a few different approaches to B-tree concurrency. The intuitive approach, first discussed by Bayer and Schkolnik [3] and Samadi [4], is to have writers exclusively lock entire subtrees starting at the highest node that might be modified. This allows modifications to be done safely, but limits the amount of concurrency by preventing access to large parts of the tree.

In [3], Bayer and Schkolnik describe an approach in which optimistic writers place only read-locks on the subtrees they descend, assuming they will be able to insert or delete without propagating change up the tree. They place an exclusive lock only on the one node they need to insert into or delete from. If the node becomes too large or small and the tree must be re-balanced, they repeat the descent using exclusive locks. This provides improved performance compared to the approach in [3], since most of the time the tree will not need to be re-balanced after a write operation.

In [5], Mond and Raz used the descending approach to insertion and deletion, described in [2, Ch. 18]. In a concurrent setting, writers can exclusively lock nodes they need to modify and then release them as they descend further, as they will not need to modify them again. This allows less of the tree to be locked at any particular time.

Lehman and Yao, in [1], added extra links to the B+ tree to create the B-link tree. Their algorithm allows data to be found in this tree even when it is in an invalid or changing state, meaning fewer nodes need to be locked. They did not include concurrent operations to re-balance the tree after deletion.

It should also be noted that shortly before Lanin and Shasha published their algorithm, a very similar one was independently created and published by Sagiv [6].

2 Lanin and Shasha's Algorithm

The concurrent B-tree algorithm by Lanin and Shasha improves upon the B-link tree in [1] by adding a concurrent merge operation to balance the tree after deletion.

Unlike Lehman and Yao's algorithm, which assumes atomic node access, locking in Lanin and Shasha's algorithm is done using *read-locks* and *write-locks* on individual nodes. Read-locks can be held by multiple processes at the same time, while write-locks are exclusive. A process will not need to hold more than one read-lock or write-lock at a time, except deletion which may hold two write-locks while restructuring the tree. It is also shown to be deadlock-free in [7, p. 383-385].

Lanin and Shasha implemented their algorithm and tested it against the algorithm in [5] with and without optimistic descents, and two algorithms from [3]. It was shown to be faster than these other algorithms. The speedup became less pronounced as the size and number of the tree's nodes increased, though it still performed slightly better on all trees [7, p. 385-386]. After this algorithm was published, the relative performance of it and others was analyzed more thoroughly by Shasha and Johnson [8] and by Srinivasan and Carey [9]. Both studies found that B-link-based algorithms, such as this one, performed significantly better than other types of concurrent algorithms.

In the remainder of this section, the algorithm will be summarized with select pseudocode. More complete pseudocode is available in [7, Appendix].

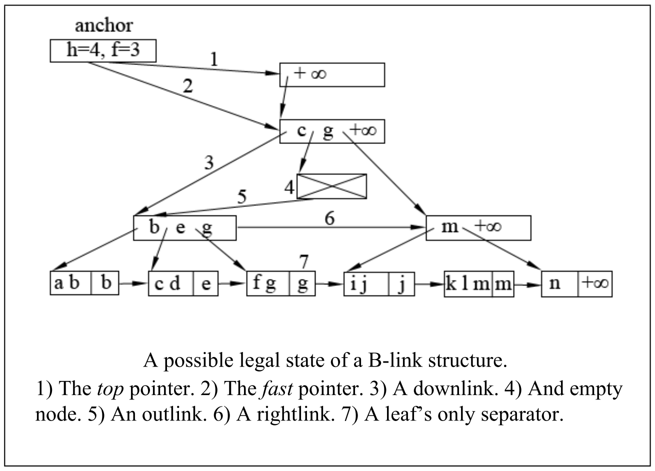
2.1 Definitions

The list below shows the definitions needed to understand the algorithm and pseudocode. They are mostly summarized from [7, p. 382]. Figure 1 shows an example from the article, which illustrates many of these definitions.

d: a downlink, n: a node

- *internal node* : Consists of a *rightlink* to the node to its right on the same level, and a sequence of *downlinks.*
- *separator* :

  - Internal nodes: Numbers (called *keys* in most other sources) which divide the range of keys located within the downlinks to either side of them.
  - Leaf nodes: A leaf node has only one separator. It corresponds to the rightmost separator of its internal parent node, which marks the upper limit of keys stored in the leaf.
- *downlink* : Link from an internal node to a node in the level below.
- *rightsep(d)* : Separator to the immediate right of *d*.
- *rightsep(n)* : The rightmost separator in *n*.
- *leftsep(d)* : Separator to the immediate left of *d*. If *d* is the leftmost downlink in its node *n*, then this is defined as the rightmost separator in *n*'s left neighbor node, or $-\infty$ if *n* is the leftmost node and has no left neighbor.
- *leftsep(n)* : For the leftmost downlink *d* in *n*: *leftsep(n) = leftsep(d)*. If *n* is empty and has an outlink pointing to n', then *leftsep(n) = leftsep(n')*
- *coverset(n)* : The range of key values covered by the node *n*: $\{x|leftsep(n) < x \leq rightsep(n)\}$.



A possible legal state of a B-link structure.
1) The *top* pointer. 2) The *fast* pointer. 3) A downlink. 4) And empty node. 5) An outlink. 6) A rightlink. 7) A leaf's only separator.

Fig. 1. A B-link tree, of the type created by Lanin and Shasha's algorithm, which illustrates several components of the structre. From [7, p. 382].

2.2 Locating Keys

Search, *insert*, and *delete* all start by using *locate* to find the leaf node where the key to be found/inserted/deleted should be. Once the node is found, *insert* and *delete* place a write-lock on the node, while *search* places a read-lock instead.

Lanin and Shasha refer to the point where search, insert and delete actually read or modify the keys in question as their *decisive operations*. For search, the decisive operation simply checks if a key exists within the leaf that has been read-locked. For insert and delete, the decisive operation is adding or removing, respectively, the desired key in the leaf that the operation has write-locked.

2.3 Normalization

When keys are inserted or deleted, nodes may end up having too many or too few keys. *Normalization* of a node, part of the process that is elsewhere reffered to as restructuring or rebalancing of the tree, is used to remedy the situation.

Nodes are split when they have too many keys, and merged when they have too few keys. These operations can cause changes that propagate up the tree, and could possibly affect the entire subtree containing the initial node up to the root. The easiest way to avoid problems with these operations in a concurrent setting is to lock the entire subtree of the node being modified.

This algorithm avoids locking such a large portion of the tree, in part, by performing splits and merges in two separate stages:

Stage 1: Half-Split or Half-Merge *Half-merge* and *half-split* are called by the *normalize* function, which obtains a write-lock on a node before calling them, and releases the lock afterward.

A half-split, where a node n is split into two nodes n and n_{right}, moves data from n but does not modify n's parent. After a half-split, n and n_{right} can still be considered part of the same node, though their structure has been changed so that they are actually two separate nodes linked together.

A half-merge is where nodes n_{left} and n_{right} are merged into n_{left}. The data from n_{right} is moved into n_{left}, and the downlink that previously pointed to n_{right} is changed to point to n_{left}. n_{right} is left empty at the end of this step, and is given an *outlink* that points to n_{left}.

Once a half-split or half-merge is performed, the locks on the nodes can be released, as the structure at this point will be operational.

Pseudocode (derived from the description in [7, Appendix]):

```
half-split($n$, $n_{right}$: node pointers){
    $n_{right}$.rightlink = $n$.rightlink;
    $n$.rightlink = $n_{right}$;
    move right half of keys in $n$ to $n_{right}$;
    if($n$ and $n_{right}$ are leaves){
        $n$.rightsep = largest key in $n$;
        $n_{right}$.rightsep = largest key in $n_{right}$;
    }
    return $n$.rightsep;
}
```

```
half-merge($n_{left}$, $n_{right}$: node pointers){
    $temp$ = $n_{left}$.rightsep;
    move all keys from $n_{right}$ to end of $n_{left}$;
    $n_{left}$.rightlink = $n_{right}$.rightlink;
    $n_{right}$.empty = true;
    $n_{right}$.outlink = $n_{left}$;
    if($n_{left}$ is a leaf){
        $n_{left}$.rightsep = largest key in $n_{left}$;
    }
    return $temp$;
}
```

Stage 2: Add-Link or Remove-Link This step completes the process begun by half-split or half-merge. Locks must already be held on the nodes being modified before these methods are used.

Add-link is done after the half-split of node n into n and n_{right}. Let $s = rightsep(n)$. Find the node p for which $s \in coverset(p)$. (This is usually known due to the earlier *locate* operation. Another *locate* can be done if the tree has changed.) Insert a downlink to the new node n into the correct (sorted) location in p, and insert s to the left of this downlink.

Remove-link is done along with the half-merge of nodes n_{left} and n_{right} into n_{left}. Before half-merge, take the rightmost separator s from n_{left}. The node p in which $s \in coverset(p)$ is the parent. After half-merge, remove s, and the downlink to the newly-empty n_{right}, from p.

2.4 Changing The Root Level

New top levels can be created by splitting the root node. However, when a root node becomes too small or empty, Lanin and Shasha found that removing it was difficult. It is left in the tree, and may be re-used later. A process called the *critic* continually runs in the background. It keeps track of, and stores in the *anchor*, a pointer to the *fast* level (see Figure 1), the highest level with more than one downlink. This is the highest useful level to start searching. Because this tree contains rightlinks between the nodes of each level, searches do not need to start at the actual root of the tree. In fact, they can start at any level and achieve the correct results, albeit less efficiently if they start at a lower level.

3 Java Implementation

3.1 Program Structure

One challenge of implementing this algorithm in Java is that the original is written in a non-modular, non-object-oriented way, with many static methods which must be called in the correct sequence for locking to work. It also contains a finely-grained locking system, and many methods assume a lock is held when they are called (see Figure 2).

In this Java implementation, some object-oriented techniques are used, but many things remain from the original algorithm which would normally be considered bad Java programming practice. For example, some methods assume locks are held before they are called, and others use parameters to determine method behavior instead of polymorphism.

Figure 2 shows a sequence diagram of the original *search* method, which is similar to *insert* and *delete*. From the method calls shown in Figure 2, it is evident that many of the original methods take only a single node, or a node and a lock type, as parameters. So in this Java version, these static methods have been replaced by instance methods in Node objects. Most of the remaining methods have been grouped into a BLinkTree object. Figure 3 shows a UML class diagram of the Java objects used in this implementation.

The algorithm also requires some methods to spawn new threads. These threads ascend the tree, starting at a leaf node and working their way up, to re-balance it. For this purpose an AscendThread class has been created, which extends Thread. These AscendThreads are launched by the BLinkTree class.

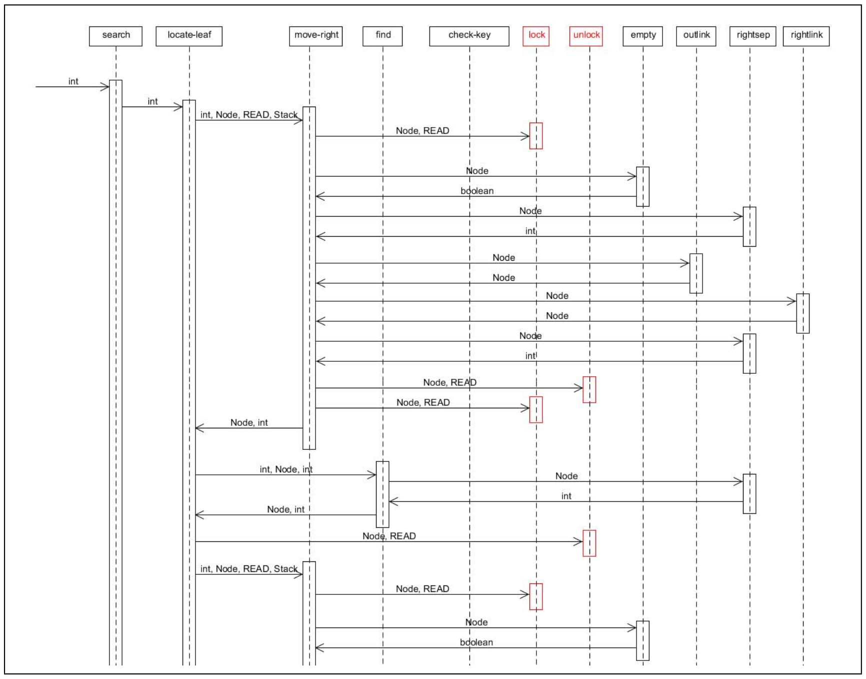


Fig. 2. A pseudo-sequence diagram of Lanin and Shasha's *search* method from [7, Appendix]. The boxes in the top row represent static methods rather than objects.

The Critic, which runs continuously in the background and keeps track of the highest useful node of the tree, has been similarly implemented as a class that extends Thread. The Critic is started when a BLinkTree is created. It can be terminated manually by a method in the BLinkTree class, so that the program does not run forever.

3.2 Data Structures

The most important Java structure that was created is the Node class, which contains all the static methods from the original algorithm that relate to an individual node. Within Node, the built-in Java Vector is used to store data. Using an array might be more efficient. However, Vector allows for easy insertion/deletion from the middle of the sequence without shifting the remaining keys. Java also has built-in methods to sort and count the contents of Vectors, both of which are frequently used by the program. Furthermore, in this particular B-tree algorithm it is possible for a Node to have many extra keys added to it, which will afterwards be removed when the tree is balanced. If arrays were used, allocation of a new array and copying of old contents, or some equivalent solution, would be required for the arrays to expand beyond their maximum capacity.

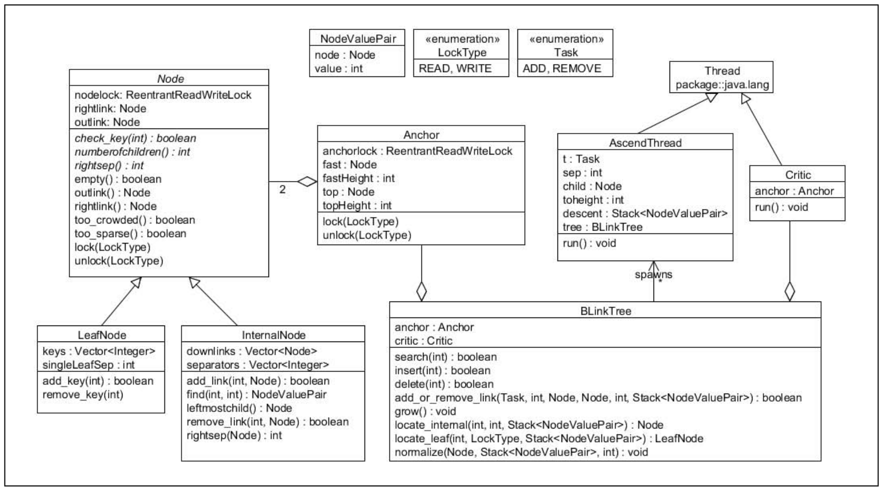


Fig. 3. A UML diagram showing the class structure of this Java implementation. This diagrams shows only methods that are part of the original algorithm.

3.3 Concurrency

To achieve concurrency, the original algorithm used only two types of lock: read-locks and write-locks. The original pseudocode contains the static methods *lock(Node, LockType)* and *unlock(Node, LockType)*, which take a node and the type of lock (*read* or *write*) as parameters. In the Java program, these have been replaced by the instance methods *lock(LockType)* and *unlock(LockType)* of the Node class. The Node objects each contain their own instance of the built-in Java lock class java.util.concurrent.locks.ReentrantReadWriteLock to produce the correct locking behavior. The *lock-anchor(LockType)* and *unlock-anchor(LockType)* methods were treated similarly.

To ensure that all threads started at the same time during testing, CyclicBarrier was used.

4 Randomized Testing

4.1 Comparison Algorithms

For comparison, a sequential version of this Java program was created. It is like the concurrent version except all locks were removed, and all instances of threads being launched were replaced by calls to sequential methods.

A version without the *merge* operation, similar to the algorithm in [1], was also created. Though it is certain that at some point a B-tree without any type of merge will become very sparse and will be less efficiently searched than a balanced B-tree, it is interesting to see how much overhead Lanin and Shasha's addition of the *merge* operation creates before this occurs.

4.2 Correctness Testing

The structure of the B-link tree is less strict than a normal B-tree, and many characteristics that would be incorrect in a standard B-tree are allowed in the B-link tree. The most important properties that must be true at all times are: all leaves are on the same level; all keys are sorted; and no thread can end up to the right of the location it is searching for.

The essential determinant of correctness is that every key in the tree can be found at any point in the tree's life, including all stages of modification. It is thus important to test the correctness of the tree while other threads are changing it. Correctness testing was therefore integrated into the performance tests, as follows:

Whenever the program was run, a number of threads did nothing but search for a list of randomly-generated odd numbers, which were added to the tree beforehand. Threads that were assigned to insert or delete only used even numbers, so the odd numbers sought would be unaffected. Thus, if the algorithm was correct, every time a search thread tried to find a key it should have been successful. Any unsuccessful search would have meant an error in the algorithm or implementation.

After initial debugging was completed, no test runs of the program had any search threads fail to find their keys. Since many tests were done, including tests on a 32-core machine, this shows that the algorithm and implementation are quite reliable.

4.3 Performance Testing

Hardware Used Tests that used varying numbers of cores were done on the Intel® Manycore Testing Lab's 32-core Linux machine. Other tests were done using a Windows 7 x64 PC with an AMD Phenom 4-core processor and 4GB of RAM.

Test: Scalability by Number of Cores For this test, the algorithm was run with four threads per processor, and a consistent proportion of threads of each type (50% search, 25% insert, 25% delete).

The results are shown in Figure 4 and Figure 5. The standard deviation of the measurements is fairly high, but it remained consistent as more tests were performed. Data from the sequential algorithm has been omitted from the first graph so that the error bars are more clearly visible.

The results show that the algorithm scales well as the number of processors increases. The version without merge performs significantly better in this type of tree, as does the sequential version. Even though the performance of the sequential version, of course, does not increase with a higher number of processors, the performance of the concurrent algorithm stops increasing after about 24 processors, and is only barely within the range of the sequential algorithm's. The performance gain achieved by removing the merge operation is fairly significant

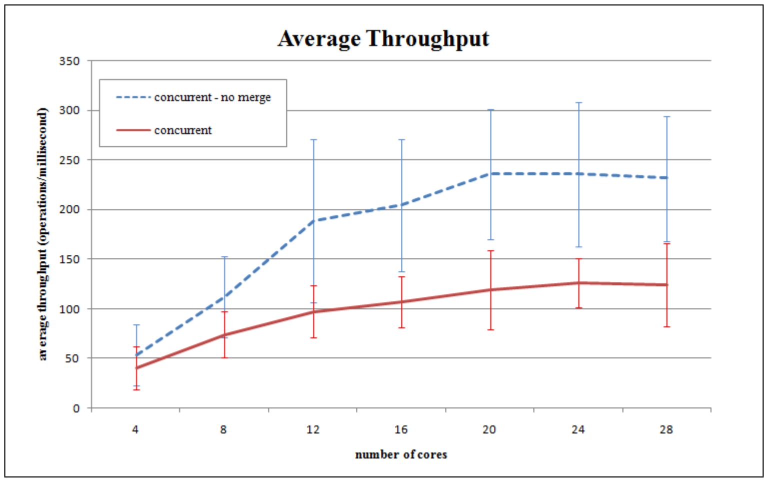


Fig. 4. Results of throughput scalability tests, sequential algorithm omitted. Error bars represent standard deviation.

here (though it is expected to decrease for larger, sparser trees), but its scaling behavior is the same: the performance stops increasing after about 20-24 cores.

Test: Overhead of Concurrency Figure 6 shows the results of tests on the Manycore computer using only one core. The number and proportion of threads are the same as in the section above. This gives an idea of how much overhead is required for the concurrent portions of the algorithms, compared to the sequential version where the concurrency is removed. Error bars show standard deviation.

Despite the high standard deviation, it is evident that concurrency requires significant overhead. This explains the improvement in throughput when using the sequential algorithm, though with enough cores the concurrent algorithms eventually achieve similar performance.

Test: Increasing Tree Sparsity The tests above show that Lanin and Shasha's merge operation creates noticeable overhead, decreasing its performance compared to Lehman and Yao's algorithm without merge. However, it is clear that the B-link tree without merge will be outperformed by Lanin and Shasha's tree in some situations. After a number of keys have been deleted, searches should be operating on sparser trees in the version of the algorithm without a merge operation, which should negatively impact their efficiency.

To verify this, tests were done in which the starting size of the tree was the same for each test, but the number of keys that the program attempted to delete was increased (since keys to insert and delete were generated randomly, not all deletions were successful). After the insertions and deletions were finished, searches were performed and timed.

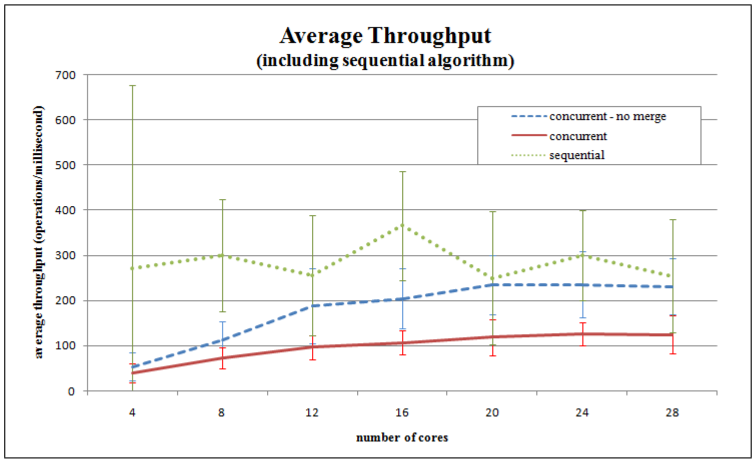


Fig. 5. Results of throughput scalability tests, including the sequential algorithm. Error bars represent standard deviation.

Results are shown in Figure 7. They indicate that searches done in a tree created by the algorithm with merge are, in fact, faster than searches done in the tree created without merge, once enough keys are deleted to make that tree sparse.

5 Java PathFinder Testing

To more thoroughly test the correctness of the code, the model checking program Java PathFinder (JPF) was used. It tests all possible thread interleavings to ensure that errors cannot occur.

5.1 Code Changes

So that JPF could more efficiently test the code, portions that existed only for performance testing were removed. The JPF version no longer records execution time or writes data to files. All CyclicBarriers were also removed; these were included to create a higher probability of errors due to thread interleaving, which is not helpful to JPF.

A while loop from the original algorithm was changed to a for loop that runs at most five times. The original loop normally runs only once, but could run forever. Exiting early only delays balancing of the tree, and improves JPF's performance. Another part of the original algorithm, the Critic, was removed entirely. It adds many more potential thread interleavings, but has little impact on the algorithm.

To reduce the JPF state space, Verify.beginAtomic() and Verify.endAtomic() are used for the code before and after the operations of interest. Furthermore,

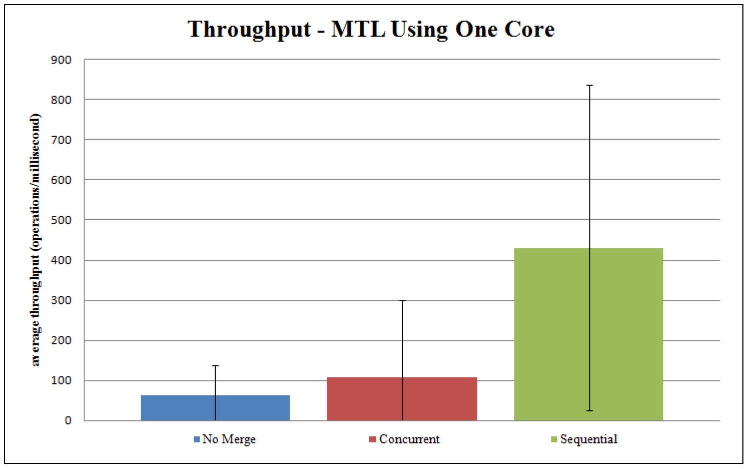


Fig. 6. Results of tests using only one core of the Intel Manycore machine. Error bars represent standard deviation.

the size of the tree nodes was set to two. This small node size triggers tree re-balancing after very few keys are changed, which allows JPF to test the re-balancing portions of the algorithm on trees of a manageable size.

Many possible errors already cause Java exceptions, and will be detected by JPF. For other types of error, such as incorrect return values, assert statements were added. The use of assert is different for each test method, and is described in more detail later.

5.2 Properties, Listeners and Search Strategies

The PreciseRaceDetector listener was used to find data races. NoDeadlockProperty and NoUncaughtExceptionsProperty were the properties used. It seems that even when a program always deadlocks and never terminates during a normal execution, JPF's test on that program can actually terminate and report no errors unless NoDeadlockProperty is specified.

Alternate search strategies used were DFSSearch and BFSHeuristicSearch. Since only one search strategy can be used per execution, only these two alternate search strategies were attempted, and only on select tests for which the default strategy ran out of memory before finding any errors.

5.3 Tests Performed

Insertion Threads Only This method was created to test the insertion method, including the split operation that re-balances the tree if too many keys occur in a single node. The method works as follows: A number of insertion threads is chosen. Each thread is given a unique number to insert. *Insert* returns a boolean indicating whether the insertion was successful. Assert is used on this

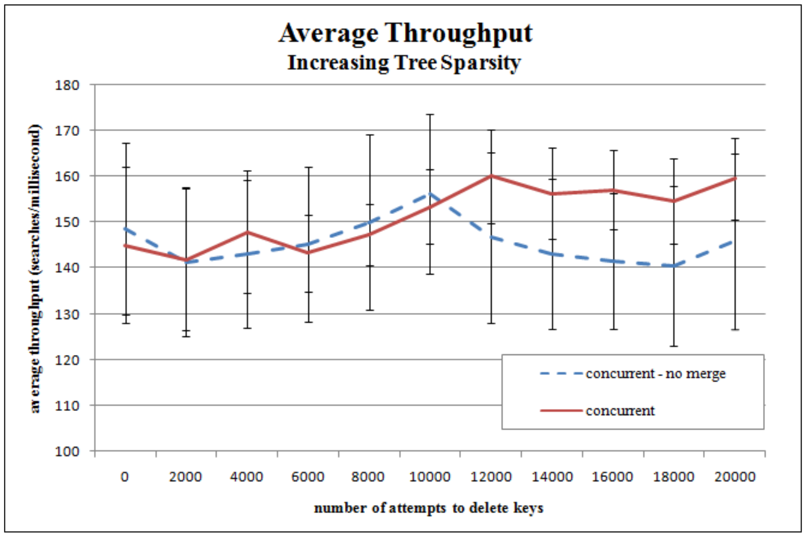


Fig. 7. Results of tests in which searches were performed after deleting an increasing number of keys from the tree. Error bars show standard deviation.

return value to check that it is always true, since all insertions should complete successfully.

JPF successfully tested this method using two and three insertion threads with no errors. With one thread, JPF ran out of memory.

Deletion Threads Only This method was written to test deletion from the tree, and the re-balancing via the merge operation introduced in [7]. It works as follows: A number of deletion threads is chosen. Each deletion thread is given a random, unique number to delete, which is added to the tree before testing begins. The delete operation returns a boolean indicating whether it was successful. If the item was in the tree and only one thread was attempting to delete it, as is the case in this test, this return value should always be true. Assert is used to check this.

Using this method, JPF quickly found errors in which the delete operation returned false when it should have returned true. Upon examination, whenever this occurred the tree did, in fact, have the key removed. This shows that the deletion completed successfully, but its return value was incorrect. This incorrect value originates in a very simple method used by the delete function, shown below. *keys* is a Vector<Integer> object, and a lock is already held on the node before the method is called.

```
public boolean remove_key(int value) {
     if (keys.contains(value)) {
          keys.removeElement(value);
          return true;
```

```
        } else
            return false;
        }
    }
```

The reason the method above could return false when an element is removed from *keys* is unclear. Debugging was attempted using the ExecTracker listener, but more expertise is required to interpret those results correctly.

All Thread Types This method allows any combination of insert, delete or search threads to be tested, and checks whether searches return the correct results. As mentioned earlier, search results are an important indicator of tree correctness. Many types of invalid tree state could result in a search being unable to find a key in the tree.

For this method, the numbers of insert, search and delete threads are chosen. Each search thread is randomly assigned an odd number to search for, which is added to the tree before the threads are started so that the search should always return true. Each remove thread is randomly assigned an even number to remove. This number is added to the tree before the threads begin, to ensure that the removal will actually take place. Each insert thread is randomly assigned an even number to insert. Assert statements are used to verify that all searches return true.

The results are summarized in Table 1. Tests using only insertion or deletion did not find any errors. It appears that these methods alone do not interfere with searches. However, using one insertion, one deletion, one search, and the BFSHeuristic search strategy, a serious error was quickly found: A search for a key in the tree did not find that key. Since insertions and deletions alone do not seem to cause this problem, the error must result from the combination of insert, delete and search together.

This error was checked for throughout performance testing and was never encountered. It was also not found by the default search strategy before JPF ran out of memory. The error is therefore very uncommon. A large variety of problems with the tree could be the cause of this type of error, so at the moment the cause has not been determined.

number of threads:			JPF results
insert	delete	search	
1	0	1	finished with no errors
0	1	1	finished with no errors
2	0	1	out of memory, no errors found
0	2	1	error found: incorrect return value from *delete*
1	1	1	error found with BFSHeuristic: *search* cannot find key in tree

Table 1. Results when JPF was run with various combinations of insert, delete and search threads.

6 Conclusion

The algorithm created by Lanin and Shasha in [7] was successfully implemented in Java.

Randomized performance tests show that the implementation's throughput scales fairly well as the number of processors increase. It has high overhead, however, and may only gain a performance advantage over a sequential B-tree when the number of processors available is very high (over 24 or so).

The randomized correctness testing did not find any errors. However, some errors were found using Java PathFinder. Their exact causes have yet to be discovered. Fortunately, they seem unlikely to occur with normal use of the code.

7 Acknowledgements

Thanks to the Intel® Manycore Testing Lab (www.intel.com/software/manycoretestinglab) for the use of their systems.

References

1. Lehman, P.L., Yao, S.B.: Efficient locking for concurrent operations on B-trees. ACM Transactions on Database Systems **6**(4) (December 1981) 650–670
2. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. 2nd edn. MIT Press, Cambridge, Massachusetts (2001)
3. Bayer, R., Schkolnick, M.: Concurrency of operations on B-trees. Acta Informatica **9**(1) (1977) 1–21
4. Samadi, B.: B-Trees in a system with multiple users. Information Processing Letters **5**(4) (1976) 107–112
5. Mond, Y., Raz, Y.: Concurrency control in B+-trees databases using preparatory operations. In: Proceedings of the 11th international conference on Very Large Data Bases - Volume 11, VLDB Endowment (1985) 331–334
6. Sagiv, Y.: Concurrent operations on B*-trees with overtaking. Journal of Computer and System Sciences **33**(2) (October 1986) 275–296
7. Lanin, V., Shasha, D.: A symmetric concurrent B-tree algorithm. In: Proceedings of 1986 ACM Fall joint computer conference, Los Alamitos, CA, USA, IEEE Computer Society Press (1986) 380–389
8. Johnson, T., Sasha, D.: The performance of current B-tree algorithms. ACM Transactions on Database Systems **18**(1) (March 1993) 51–101
9. Srinivasan, V., Carey, M.J.: Performance of B+ tree concurrency control algorithms. The VLDB Journal **2**(4) (October 1993) 361–406