
# Non-blocking k-ary Search Trees

**Trevor Brown and Joanna Helga**

Technical Report CSE-2011-04

June 2011

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

# Non-blocking $k$-ary Search Trees

Trevor Brown and Joanna Helga
DisCoVeri Group
Department of Computer Science and Engineering
York University

June 1, 2011

**Abstract**

This paper presents the first concurrent non-blocking $k$-ary search tree. Our data structure generalizes the recent non-blocking binary search tree of Ellen et al. to trees in which each internal node has $k$ children. Larger values of $k$ decrease the depth of the tree, but lead to higher contention among processes performing updates to the tree. Our Java implementation uses single-word compare-and-set operations to coordinate updates to the tree. We present experimental results from two machines. The first one is a 32-core Intel machine with 32 hardware threads available, and the second machine is a 16-core Sun machine with 128 hardware contexts. The experimental results show that our implementation achieves higher throughput than the Java class library's non-blocking skip list. Our implementation also outperforms the lock-based AVL tree of Bronson et al., which is the leading concurrent search tree. Our experimental results show that our algorithm scales extremely well.

## 1 Introduction

With the arrival of machines with many cores, there is a need for efficient, scalable concurrent implementations of often-used abstract data types (ADTs) such as the dictionary. Most existing concurrent implementations of the dictionary ADT are lock-based (see, e.g., [3, 8]). However, locks have some disadvantages (see, e.g., [6]). Other implementations make use of operations that are not directly supported by most multicore machines such as load-link/store-conditional [2] and multi-word compare-and-swap (CAS) [7]. The dictionary has also been implemented by means of software transactional memory (STM) (see, e.g., [9]). However, such an implementation is currently not efficient [3].

Most multicore machines support (single-word) CAS operations. Non-blocking implementations of dictionaries have been given based on skip list (SL) and binary search tree (BST) data structures. Sundell and Tsigas [11], Fomitchev and Ruppert [5], and Fraser [7] have implemented a skip list using CAS operations. A sketch of a binary search tree (BST) implementation using only CAS operations was given by Valois [12], but the first complete algorithm was presented by Ellen et al. [4]. All of these implementations are non-blocking. The non-blocking property is desirable because it ensures that, while a single operation may be delayed, the system as a whole will always make progress. The BST introduced by Ellen et al. is the first practical non-blocking tree data structure.

In this paper, we generalize their BST to a $k$-ary search tree ($k$-ST) in which each internal node contains $k-1$ keys and has $k$ children. Larger values of $k$ decrease the average depth of nodes, but increase the local work done at each internal node in routing searches and performing updates to the tree. By varying this $k$, we can balance tree depth and local work. This way, the structure can be tailored to perform well under an expected level of contention, or for an expected ratio of updates to searches.

Searches are extremely simple and fast. They completely ignore concurrent updates, and behave exactly as they would in the sequential case. The essential new idea behind the $k$-ST is the new scheme for insertions and deletions (in particular the generalizations from the BST's deletion to pruning deletion, and from the

BST's insertion to sprouting insertion). This algorithm provides evidence that the update coordination scheme used in the BST can be extended to implement more complex tree structures.

Since Java is among the most popular programming languages, there is a need for efficient concurrent implementations of ADTs in Java. We have implemented both the BST of Ellen et al. and our $k$-ST in Java. Since Java supports compare-and-set (CASet), and not CAS, we show that each CAS operation can be replaced by a snippet that uses CASet, so long as the algorithm does not suffer from the ABA problem. The ABA problem can occur when an algorithm reads a field twice and uses the equality of the two results to determine that the value has not changed in between reads (unless precautions are taken, other processes could have written another value, then re-written the original). The $k$-ST algorithm does not suffer from the ABA problem and, hence, this transformation is applicable.

We also compared our algorithm's performance against ConcurrentSkipListMap from the Java class library, and the lock-based AVL tree of Bronson et al. [3]. The AVL tree is the leading concurrent search tree. It has been compared in [3] with SL, a lock-based red-black tree, and a red-black tree implemented using STM. Since SL and AVL drastically outperform the other two implementations, we have not included them in our comparison. In our experiments, the BST and 4-ST ($k$-ST with $k = 4$) algorithms are top performers in both high and low contention cases. In high contention cases, the simplicity of the BST implementation and the smaller number of keys affected by each update afford it a higher degree of parallelism, pushing it into the lead position. In low contention cases, the reduction in tree depth propels the 4-ST to the top. In our experimental setup, we did not observe any significant benefit of using values of $k$ greater than four. Since the focus of the work of Ellen et al. [4] was on proving correctness, this paper provides the first performance analysis of their BST.

The BST and $k$-ST are both unbalanced trees. All performance tests in this paper use uniformly distributed random keys for updates. If keys are not random then, in certain cases, SL (a randomized algorithm) and AVL (a balanced tree) will outperform BST and $k$-ST. Based on our experiments, we observe that concurrent updates using uniformly distributed random keys result in $k$-STs of logarithmic height. A logical direction for continued research will be to perform balancing so that our algorithm's strong performance can be guaranteed in all cases. We believe that the techniques used in this paper can be extended to produce a balanced tree.

# 2 $k$-ary Search Trees

## 2.1 The Structure

We use a leaf-oriented, non-blocking $k$-ST to implement the dictionary ADT. A dictionary stores a set of keys from an ordered universe. It does not admit duplicate keys. Here, we define the operations on the ADT to be Find($key$), Insert($key$), and Delete($key$). The Find operation returns True if $key$ is in the dictionary, and False otherwise. An Insert($key$) operation returns False if $key$ was already present. Otherwise, it adds $key$ to the dictionary and returns True. A Delete($key$) returns False if $key$ was not present in the dictionary. Otherwise, it removes $key$ from the dictionary and returns True. Often the dictionary ADT allows a value to be associated with each key. It is a simple task, as we shall see, to modify the Insert operation to store this value in the leaves of the tree, alongside the corresponding key, and the Search operation to return this information.

The $k$-ST is leaf-oriented, meaning that at all times, the set of keys in the leaves of the tree is precisely the set of keys in the dictionary. Keys in internal nodes of the $k$-ST may or may not be present in the dictionary, and serve only to direct searches down the tree. In this work, we understand any $k$-ST to be leaf-oriented.

Each leaf in a BST has zero children and one key. Each internal node has exactly two children and one key (see Figure 1). In a $k$-ST, each leaf has zero children and at most $k - 1$ keys. Each internal node has exactly $k$ children and $k - 1$ keys (see Figure 2). It is permitted for a leaf of the $k$-ST to have zero keys. In this case, it is said to be an empty leaf. Inside each node, keys are maintained in increasing order.

The search tree property for $k$-STs is a natural generalization of the familiar BST property, and is shown

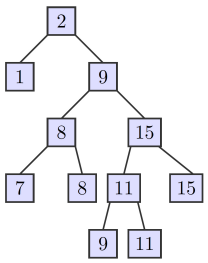



Figure 1: A BST ($k$-ary tree with $k = 2$) resulting from insertions: 1, 2, 7, 9, 8, 15, 11, and the deletion of 2.

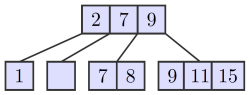



Figure 2: A $k$-ary tree ($k = 4$) resulting from insertions: 1, 2, 7, 9, 8, 15, 11, and the deletion of 2.

in Figure 3. For any internal node with keys $a_1, a_2, ..., a_{k-1}$, subtree 1 (leftmost) contains keys less than $a_1$, subtree $k$ (rightmost) contains keys greater than or equal to $a_{k-1}$, and subtree $i$ where $1 < i < k$ contains keys greater than or equal to $a_i$ and less than $a_{i+1}$.

## 2.2 Modifications to the Tree

We first describe a sequential implementation of the dictionary operations, and subsequently transform it into a concurrent and non-blocking implementation in the following sections. As a consequence of the leaf-oriented nature of the $k$-ST, the INSERT and DELETE procedures always operate on leaves. Inserting a key into the dictionary replaces a leaf by a "larger" leaf (with one more key), or by a small subtree if the leaf is full (has $k - 1$ keys). Deleting a key replaces a leaf by a smaller leaf (with one less key), or prunes the leaf and its parent out of the tree.

More precisely, the operation INSERT($key$) first searches for $key$. If it is found, the INSERT returns FALSE. Otherwise, it proceeds according to two cases (see Figure 4). Let $l$ be the leaf into which $key$ should be inserted. If $l$ is full (has $k - 1$ keys) then we cannot insert the key into the leaf, so INSERT replaces $l$ by a *newly created* subtree of $k + 1$ nodes. This subtree consists of an internal node $n$ whose keys are the $k - 1$ greatest out of the $k - 1$ keys in $l$ and the new key $key$. The children of $n$ are $k$ *new* nodes, each containing one of the $k$ aforementioned keys. We call this first type of insertion a sprouting insertion because it causes the tree to sprout new leaves. Otherwise, if $l$ is not full (has fewer than $k - 1$ keys), INSERT replaces $l$ by a *new* leaf that includes $key$ in addition to all of the keys that were in $l$. We call this second type of insertion a simple insertion.

The operation DELETE($key$) first searches for $key$. If it is not found, the DELETE returns FALSE. Otherwise, it proceeds according to two cases (see Figure 5). Let $l$ be the leaf from which $key$ should be deleted. If $l$ has one key and the parent of $l$ has exactly two non-empty children, then the entire leaf $l$ can be deleted

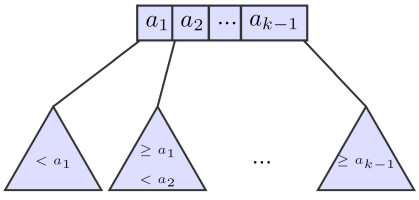



Figure 3: The search-tree property for a node of a $k$-ary tree.

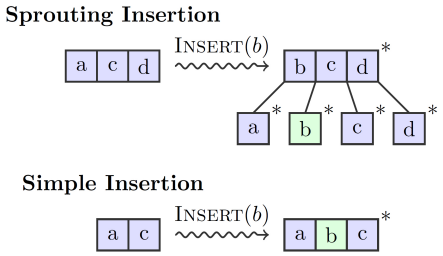



Figure 4: Sprouting insertion occurs when the leaf being inserted into has $k-1$ keys. It replaces the leaf by a small subtree. Simple insertion occurs when there are fewer than $k-1$ keys. It replaces the leaf by a new leaf with the inserted key. Asterisks indicate that a node is newly created and stored in freshly allocated memory.

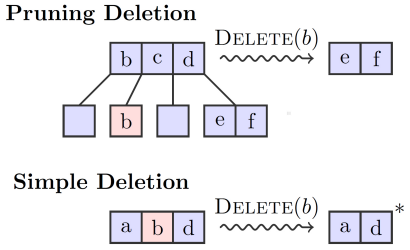



Figure 5: Pruning deletion occurs when the leaf being deleted from has one key, and its parent has exactly two non-empty children. This replaces the parent with its remaining non-empty child (the sibling of the leaf whose key is deleted). Simple deletion replaces the leaf by a new leaf with the deleted key removed. Asterisks indicate that a node is newly created and stored in freshly allocated memory.

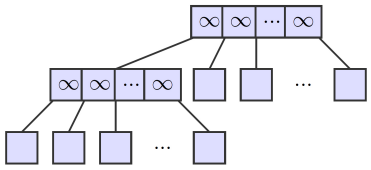



Figure 6: The initial state of the $k$-ary search tree.

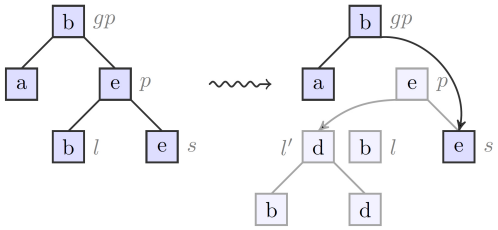



Figure 7: Example of the danger of uncoordinated concurrent updates. Faintly shaded nodes are no longer in the tree. If $gp$'s right child will soon be changed to $s$ by a DELETE($b$), and $p$'s left child is changed to $n$ by an INSERT($d$), then the new key $d$ will be lost.

and, because it has only one non-empty sibling $s$, the parent node is no longer useful (since its keys just direct searches). The DELETE procedure can simply replace the parent with $s$. We call this first type of deletion a pruning deletion because it prunes some "dead wood" out of the tree. Otherwise, $l$ has more than one key or the parent of $l$ has more than two non-empty children, so DELETE replaces $l$ by a *new* leaf with *key* removed. We call this second type of deletion a simple deletion. With this insertion and deletion scheme the parent always has at least two non-empty children.

Note that a pruning deletion changes a child pointer of the grandparent of $l$ to point to $l$'s only non-empty sibling. To avoid dealing with degenerate cases when there is no parent or grandparent of $l$, we initialize the tree with two dummy internal nodes and $2k-1$ empty leaves at the top (as shown in Figure 6). Each dummy node has $k-1$ keys valued $\infty$ (a special key, larger than any key in the dictionary). One of the dummy nodes becomes the root and takes $k-1$ empty leaves as its rightmost children, and the other takes the remaining $k$ empty leaves as children and becomes the leftmost child of *root*. All dictionary keys are then stored in the subtree rooted at the leftmost grandchild of *root*.

## 2.3 Coordination Between Updates

Without some form of coordination, interactions between concurrent updates may produce incorrect results. Suppose that a pruning deletion and a simple insertion are performed concurrently in the binary tree on the left in Figure 7. If the steps of the INSERT($d$) and DELETE($b$) are interleaved in a particular order, key $d$ may be inserted as a grandchild of $p$, and erroneously pruned out of the tree when the DELETE changes the right pointer of $gp$ to point to $s$ (see the tree on the right).

To avoid situations such as this, internal nodes are augmented to contain uniquely identifiable UpdateStep objects that indicate an operation has exclusive access to the child pointers of a node. The threat that exclusive access poses to progress is overcome with the helping mechanism we shall discuss later. This coordination scheme extends the work of Ellen et al. [4] which was, in turn, inspired by the work of Fomitchev and Ruppert [5]. These objects serve as something similar to locks, because all processes operate under the following agreement. Before an operation can modify a child pointer of an internal node $n$, it must successfully store a uniquely identifiable UpdateStep at $n$ (using a CAS operation), indicating that it intends to perform the modification. An operation cannot store an UpdateStep at node $n$ if another operation has already stored an UpdateStep at that node, until $n$ is cleared.

Different types of UpdateStep objects are utilized by our algorithm. The first, ReplaceFlag, is stored at

an internal node $p$ to indicate that an operation intends to replace a leaf (child of $p$) with another leaf, or a small subtree. It is used for both types of insertion, and for simple deletions, where a leaf is replaced by another leaf with one less key. The second type, the PruneFlag, is used for pruning deletion. For the rest of this section, let $l$ be a leaf into which a key is being inserted, or from which a key is being deleted, let $p$ be the parent of $l$, and let $gp$ be the grandparent of $l$. A PruneFlag at internal node $gp$ indicates that a DELETE operation intends to change a child pointer of $gp$ from $p$ to the only non-empty sibling of $l$, pruning $p$ and $l$ from the tree in the process.

More precisely, flagging is implemented by giving every internal node a *pending* field that can store an UpdateStep object. This field is modified exclusively by CAS operations that write references to newly created objects. The type of these objects can be the above mentioned ReplaceFlag or PruneFlag, Mark (to indicate the node is marked, as described below) or Clean (to indicate the node is neither flagged nor marked). Initially, a node is neither flagged nor marked and, hence, its pending field contains a Clean object.

We also employ marks to resolve the situation in Figure 7. Before an internal node is to disappear from the tree, it must first be *marked*. Marks also serve as a sort of lock. As with flags, no operation can change a child pointer of a node whose pending field contains a Mark object. However, there is a difference in that flags are temporary, and are removed when a modification is completed. Marks are permanent and signify that the child pointers of a node can never change again.

We return to Figure 7 and illustrate how flagging and marking coordinates the interactions between the simple insertion and the pruning deletion. After DELETE($b$) has successfully stored a PruneFlag at $gp$ (indicating that it intends to change a child pointer of $gp$), it must store a Mark at $p$ (indicating that it intends to prune this internal node). Once the mark has been successfully stored at $p$, we know that it is safe to prune $l$ and $p$ out of the tree. In particular, we know that no child pointer of $p$ will ever change.

Since flagging and marking is a two-step process, it is possible that DELETE($b$) stores a PruneFlag object at $gp$ and, before it can store a Mark object at $p$, INSERT($d$) writes a ReplaceFlag object at $p$. If DELETE($b$) attempts to mark $p$ before INSERT($d$) has completed its insert, the CAS step will fail, because the *pending* field is already occupied. If, however, DELETE($b$) tries to mark $p$ after the insertion has finished, it should not succeed–if the delete were completed, $gp$'s right child pointer would be changed to point to $s$, and the newly inserted key would be lost. We overcome this by forcing DELETE($b$) to read the *pending* field of $p$ before it writes the first PruneFlag into $gp$. This value will be used as the "expected" value for the CAS operation that attempts to mark $p$. When INSERT($d$) finishes, it stores a *new*, uniquely identifiable Clean value in the *pending* field of $p$. This Clean value will not match any value that DELETE($b$) could have read earlier, so the CAS that tries to store a Mark at $p$ will fail. Since the DELETE can no longer proceed, it must unflag $gp$ and retry the update from scratch. When we unflag $gp$ without having marked $p$ or performed the deletion, we call it *backtracking* a PruneFlag.

The details of the INSERT and DELETE operations, including flagging and marking steps, are as follows. A sprouting insertion (see Figure 4) proceeds by creating the new subtree, flagging $p$ with a ReplaceFlag object, changing the child pointer of $p$ (child pointers are always changed with CAS), and unflagging $p$. A simple insertion creates a new leaf, flags $p$ with a ReplaceFlag object, changes the child pointer of $p$, and unflags $p$. A pruning deletion (see Figure 5) proceeds by flagging $gp$ with a PruneFlag object, then marking $p$ or backtracking the PruneFlag on $gp$. If $p$ was successfully marked, then the child pointer of $gp$ is changed, accomplishing the deletion, and $gp$ is unflagged. Otherwise, if $p$ was not marked, then $p$'s *pending* field was already occupied, so the PruneFlag on $gp$ is backtracked and the DELETE must be retried. A simple deletion is more straight forward. It proceeds by creating a new leaf with one less key (the deleted key is removed), flagging $p$ with a ReplaceFlag object, changing the child pointer of $p$, and unflagging $p$.

## 2.4 Helping

If flags and marks are not augmented with some additional mechanism, they threaten progress in the case that an operation is unexpectedly delayed while holding exclusive access to a flagged or marked node. To overcome this hurdle, we follow the approach taken by Ellen et al. [4], which has some similarities to Barnes' cooperative technique [1]. Say that a process $P$ flags or marks a node hoping to complete some tree modification $C$. The flag or mark object is augmented to contain sufficient information so that any process

```
1   ▷ Type definitions:
2   type Node {
3       Key ∪ {∞}  a_1, ..., a_{k-1}          ▷ set only at creation
4   }
5   type Internal {                           ▷ subtype of Node
6       Node  c_1, ..., c_k
7       UpdateStep  pending
8   }
9   type Leaf {                               ▷ subtype of Node
10      int keyCount                          ▷ set only at creation
11  }
12  type UpdateStep { }
13  type ReplaceFlag {                        ▷ subtype of UpdateStep
14      Node  l, p, newChild
15  }
16  type PruneFlag {                          ▷ subtype of UpdateStep
17      Node  l, p, gp
18      UpdateStep  ppending
19  }
20  type Mark {                               ▷ subtype of UpdateStep
21      PruneFlag  pending
22  }
23  type Clean { }                            ▷ subtype of UpdateStep

    ▷ Initialization:
24  shared Internal root := new Internal node with k − 1 keys, all ∞, and k children, the rightmost k − 1
        of which are empty leaves. Its leftmost child has k − 1 ∞ keys, and k empty leaves as children.
        As with all Internal nodes, the pending fields of root and root.c_1 refer to new Clean objects.
```

Figure 8: Type definitions and initialization.

can read the flag or mark and complete $C$ on $P$'s behalf. This allows the entire system to make progress even if individual processes are stalled indefinitely.

Unfortunately, helping can mean duplication of effort. Several processes may come across the same flag and perform the work necessary to complete the operation, but only one can perform the final CAS and modify the tree. For this reason it is advantageous to limit helping as much as possible. In our implementation, searches ignore flags and marks and proceed down the tree without helping any operation. An insertion or deletion helps only those operations that interfere with its own completion. Thus, an INSERT will only help an operation if it has flagged or marked $p$, and a DELETE will help an operation if it has flagged or marked $p$ or $gp$. After an INSERT or DELETE helps another operation, it restarts, again performing a search from the top of the tree. An INSERT or DELETE operation is repeatedly attempted until it successfully modifies the tree or finds that it can return FALSE. As we will see in Section 4.3, the amount of helping is very small.

## 2.5 Pseudocode

The Java-like pseudocode for all operations is found in Figure 8 through Figure 11. We borrow the concept of a reference type from Java. A variable $x$ of a class type $C$ is a reference to an instance of class $C$. Such an $x$ behaves somewhat like a C pointer, but does not require explicit dereferencing. References can point to an object or take on the value NULL, and management of their memory is automatic: memory is garbage-collected once it is unreachable from any reference in scope. We use $a.b$ to refers to field $b$ of the

object whose reference is stored in $a$.

The SEARCH($key$) operation is straightforward. Beginning at the leftmost child of *root* (line 26) and continuing until it reaches a leaf (line 28), it compares *key* with each key stored at the current node and takes the appropriate child pointer (line 33). (It is easy to do this, because the keys of a node never change.) Note that SEARCH operations ignore all flags and marks and proceed down the tree unhindered by updates. The FIND($key$) operation returns TRUE if the SEARCH($key$) operation finds a leaf containing *key*; otherwise it returns FALSE. If the dictionary ADT is defined to allow values to be stored with keys, then SEARCH can be modified to read this value from $l$ and return it (the modification to insertion and deletion is discussed below).

We next turn to the INSERT($key$) operation. The invoking process $P$ locates the leaf $l$, its parent $p$ and stores the parent's *pending* field in *ppending* (line 46). If $l$ contained *key* when it was read then, since duplicate keys are not permitted in the dictionary, the key is not inserted and the operation returns FALSE (line 47). (If such an INSERT is intended to modify a value stored with the already existing key, the algorithm would require some small modifications. This can be done by adding *value* fields to the leaf nodes. To modify the value of an existing key, we replace the leaf that contains that key with a new leaf with the modified value.) Otherwise, $P$ checks whether the parent's *pending* field was of type Clean when it was read (line 48). If *p.pending* was occupied by a flag or mark, then the current operation $C$ is put on hold so that $P$ can help complete the other operation that previously flagged or marked $p$. After the other operation finishes, $P$ re-attempts $C$ from scratch. Otherwise, if *p.pending* was Clean, then $P$ tries to flag $p$. In this case, $P$ creates *newChild*, a new leaf or subtree depending on which case applies to $l$ (lines 51 to 55) and stores it, along with $l$ and $p$ in a new ReplaceFlag object *op* to facilitate helping (line 56). Then $P$ attempts to CAS the ReplaceFlag object into the *pending* field of $p$ (line 57). If the CAS succeeds, $P$ calls HELPREPLACE(*op*) to finish the insertion (line 59) and the operation returns TRUE (more on HELPREPLACE below). Otherwise, the CAS failed, so another process must have changed $p$'s *pending* field. The field may have changed to a ReplaceFlag object, a PruneFlag object, a Mark object, or a new Clean object (different from the one read at line 46). Process $P$ helps this other operation (if any) to complete by calling HELP passing the *pending* field that caused the CAS failure. After helping, $P$ retries its own operation. As above, if the dictionary ADT is defined to allow values to be stored with keys, then INSERT can be modified to accept this value as an argument, and store it in *newChild*.

A call to HELPREPLACE invokes CAS-CHILD to change the appropriate child pointer of $p$ from $l$ to *newChild* (line 101), and invokes CAS to unflag $p$ (line 102).

When process $P$ performs a DELETE($key$) operation, it first locates the leaf $l$, its parent $p$ and grandparent *gp*, and stores the parent's and grandparent's *pending* fields in *ppending* and *gppending* (line 71). If $l$ did not contain *key* when it was read, then the operation returns FALSE (line 72). (If the tree stores values, then such a DELETE would return NULL. Otherwise, if the key was found and DELETE successfully deleted it, it would return the value associated with the deleted key.) Otherwise, $P$ checks *gppending* and *ppending* to determine whether *gp* and $p$ were Clean when their *pending* fields were read (lines 73 and 75). If either was occupied by a flag or mark, then $P$ helps complete the other operation that previously flagged or marked *gp* or $p$. Next, $P$ re-attempts its own operation from scratch. Otherwise, it counts the number of non-empty children of $p$ to determine the deletion case to apply. We shall discuss why counting the children is not problematic when we discuss correctness. We consider the two types of deletion separately.

If the operation is a pruning deletion (line 79), then $P$ tries to flag *gp* with a PruneFlag. It stores $l$, $p$, *gp* and *ppending* in a new PruneFlag object *op* to facilitate helping (line 80). Then $P$ attempts to CAS the PruneFlag object into the *pending* field of *gp* (line 81). If the CAS succeeds, $P$ calls HELPPRUNE(*op*) to finish the deletion (line 83) and the operation returns TRUE (more on HELPPRUNE later). Otherwise, the CAS failed, so another process must have changed $p$'s *pending* field. The field may have changed to a ReplaceFlag object, a PruneFlag object, or a new Clean object (different from the one read at line 71). The process $P$ helps complete this other (potential) operation, by calling HELP, passing the result of the CAS (the *pending* field that caused the failure). After helping, $P$ retries its own operation from scratch.

If the operation is a simple deletion (line 88), then $P$ tries to flag *gp* with a ReplaceFlag. It creates *newChild*, a new copy of leaf $l$ with *key* removed, stores $l$, $p$ and *newChild* in a new ReplaceFlag object *op*

```
25 SEARCH(Key key) : ⟨Internal, Internal, Leaf, UpdateStep, UpdateStep⟩ {
26     Node gparent, parent := root, leaf := parent.c1
27     UpdateStep gppending, ppending := parent.pending

28     while type(leaf) = Internal {
           ▷ Remember details for parent and grandparent of leaf
29         gparent := parent
30         parent := leaf
31         gppending := ppending
32         ppending := parent.pending
33         leaf := the appropriate child of parent
           ▷ (according to the search tree property)
34     }
35     return ⟨gparent, parent, leaf, ppending, gppending⟩
36 }

37 FIND(Key key) : boolean {
38     if Leaf returned by SEARCH(key) contains key, then return TRUE
39     else return FALSE
40 }

41 INSERT(Key key) : boolean {
42     Node p, newChild
43     Leaf l
44     UpdateStep ppending

45     while TRUE {
46         ⟨−, p, l, ppending, −⟩ := SEARCH(key)
47         if l already contains key then return FALSE
48         if type(ppending) ≠ Clean then {
               ▷ Help the operation pending on p complete first
49             HELP(ppending)
50         } else {
51             if l contains k − 1 keys {
52                 newChild := new Internal node with sorted keys:
                       k − 1 largest in S = {key} ∪ {k − 1 keys of l},
                       and k new children, sorted by keys, each having
                       one key from S, and pending field Clean.
53             } else {
54                 newChild := new Leaf node with sorted keys:
                       {key} ∪ {keys of l}.
55             }
56             ReplaceFlag op := new ReplaceFlag(l, p, newChild)
57             UpdateStep result := CAS(p.pending, ppending, op)

58             if result = ppending then {
                   ▷ CAS succeeded–finish the insertion
59                 HELPREPLACE(op)
60                 return TRUE
61             } else {
                   ▷ CAS failed–help the operation pending on p instead
62                 HELP(result)
63     }   }   }   }
64
65
```



Figure 9: Pseudocode for SEARCH, FIND and INSERT.

```
66 DELETE(Key key) : boolean {
67     Node gp, p
68     UpdateStep gppending, ppending
69     Leaf l

70     while TRUE {
71         ⟨gp, p, l, ppending, gppending⟩ := SEARCH(key)
72         if l does not contain key, then return FALSE
73         if type(gppending) ≠ Clean then {
               ▷ Help the operation pending on gp complete first
74             HELP(gppending)
75         } else if type(ppending) ≠ Clean then {
               ▷ Help the operation pending on p complete first
76             HELP(ppending)
77         } else {
               ▷ Try to flag gp
78             int ccount := number of non-empty children of p
                   obtained by checking all children in sequence

79             if ccount = 2 and l has one key then
                   ▷ Swing appropriate child pointer of gp to the
                       only non-empty sibling of l.
80                 PruneFlag op := new PruneFlag(l, p, gp, ppending)
81                 UpdateStep result = CAS(gp.pending, gppending, op)

82                 if result = gppending then {
                       ▷ CAS successful–either delete or unflag
83                     if HELPPRUNE(op) then return TRUE;
84                 } else {
85                     ▷ CAS failed–help the operation pending on gp
86                     HELP(result)
87                 }
88             } else {
                   ▷ Replace the leaf l by one without key.
89                 Node newChild := new copy of l with key removed
90                 ReplaceFlag op := new ReplaceFlag(l, p, newChild)
91                 UpdateStep result := CAS(p.pending, ppending, op)

92                 if result = ppending then {
                       ▷ CAS succeeded–insert the replacement leaf
93                     HELPREPLACE(op)
94                     return TRUE
95                 } else {
                       ▷ CAS failed–help the operation pending on p
96                     HELP(result)
97     }   }   }   }   }
98
99
```

Figure 10: Pseudocode for DELETE.

```
100 HelpReplace(ReplaceFlag op) {   ▷ Precondition: op is not Null
      ▷ Replace one node with another according to instructions in op
101   CAS-Child(op.p, op.l, op.newChild)
102   CAS(op.p.pending, op, new Clean())
103 }
104
105 Help(UpdateStep op) {              ▷ Precondition: op is not Null
106   if type(op) = ReplaceFlag then HelpReplace(op)
107   else if type(op) = PruneFlag then HelpPrune(op)
108   else if type(op) = Mark then HelpMarked(op.pending)
109 }
110
111 CAS-Child(Node parent, Node oldChild, Node newChild) {
      ▷ This routine tries to change the appropriate child field of
        parent from oldChild to newChild.
      ▷ Precondition: no argument is Null and type(parent) =Internal
        and type(parent.pending) is ReplaceFlag or PruneFlag
112   int i := index such that parent.c_i = oldChild or 0 if no such index
113   if i > 0 then CAS(parent.c_i, oldChild, newChild)
114 }
115 HelpPrune(PruneFlag op) : boolean {
      ▷ Precondition: op is not Null
116   UpdateStep result := CAS(op.p.pending, op.ppending, new Mark(op))
117   if result = op.ppending or
        result is a Mark with result.pending = op then {
        ▷ Node successfully marked–complete the deletion
118     HelpMarked(op)
119     return True
120   } else {
        ▷ Mark CAS failed–help the operation pending on p
          and backtrack (erase) the flag on gp.
121     Help(result)
122     CAS(op.gp.pending, op, new Clean())
123     return False
124   }  }
125
126   HelpMarked(PruneFlag op) {
        ▷ Precondition: op is not Null
127     Node other := the only non-empty sibling of op.l,
          obtained by visiting all children of op.p.
        ▷ Splice op.p out of the tree, replacing it by other
128     CAS-Child(op.gp, op.p, other)
129     CAS(op.gp.pending, op, new Clean())
130   }
```

Figure 11: Pseudocode for other supporting functions.

to facilitate helping (line 90). Next, $P$ attempts to CAS *op* into *p.pending* (line 91) and, if it succeeds, it calls HelpReplace to finish the deletion (line 93). Otherwise, the CAS failed, so $P$ helps the other operation that changed *p.pending* by calling Help passing the result of the CAS. Again, after helping, $P$ retries its own operation from scratch. Note that this case is very much like the case of simple insertion.

The HelpPrune procedure, invoked by the Delete operation (and by Help), attempts the second (marking) step of a pruning deletion. Recall that *op*, created in the Delete routine, contains pointers to $l$, the leaf containing the key to be deleted, its parent $p$, and its grandparent *gp*. The HelpPrune procedure begins by attempting to mark the parent *op.p* (line 116). If the CAS successfully marks *op.p*, or another helping process already stored a mark for this operation, then the mark is considered to be successful. In this case, HelpMarked is called to finish the pruning deletion (line 118), and True is returned. Otherwise, if the CAS failed and the mark was not performed by a helping process, another operation is pending on this node. The other operation is helped to complete (line 121), the PruneFlag at the grandparent *op.gp* is backtracked (line 122), and False is returned by HelpPrune. The process that invoked the Delete procedure will ultimately retry the operation from scratch (it makes little sense to proceed from here, since *op.p* will change as a result of the other operation).

The HelpMarked procedure performs the final step of a pruning deletion, pruning out some dead wood by changing the appropriate child pointer of *op.gp* from *op.p* to point to the only non-empty sibling of *op.l*. This sibling of *op.l* is found on line 127. (It is a straightforward task to inspect the keys of *op.gp* in sequence to select this child pointer, since the keys of a node never change.) The CAS-Child routine is invoked to change the child pointer of *op.gp* (line 128), and CAS is invoked to unflag *op.gp* (line 129).

## 2.6 A Brief Correctness Argument

It can be demonstrated that our algorithm exhibits linearizability, a correctness condition defined by Herlihy and Wing [10]. Parallel executions are modeled as a sequence of operation invocations and responses. A parallel execution is said to be linearizable if each operation can be assigned a linearization point at some instant between its invocation and its response such that, if all operations complete instantaneously at their linearization points, each returning the same result that it returns in the parallel execution, the resulting sequential execution behaves according to the sequential specification of the data structure. An algorithm is linearizable if every possible execution is linearizable. This ensures that, while there may be many possible interleaving of concurrent operations, every possible interleaving is equivalent to a correct sequential execution.

We omit a full proof, noting instead that the proof of correctness by Ellen et al. for lock-free BSTs in [4] can be generalized to $k$-STs, and highlight the most interesting modifications to the proof.

Selecting the appropriate pointer in Search (line 33) is easy because the keys of a node never change. By the same reasoning, it is easy to check if a leaf has a key by inspecting the keys in sequence (lines 38, 51, 72).

On line 112, all indices can simply be checked in sequence. To see why, observe that in every invocation of CAS-Child *parent* is flagged. Further, as long as *parent* is flagged, exactly one of its children can be changed, exactly once, by the CAS on line 113. Thus, the only outcomes for an iteration over the children of *parent* are that a matching child pointer is found, or not. If it is found, no process has yet performed the CAS. Otherwise, another process has already performed the CAS and, since there is no ABA problem, no child pointer will match *oldChild*.

The children of $p$ can simply be checked in sequence on line 78. Observe that the *pending* field of *gp* is read in the Search on line 71 and stored in *gppending*, and this *gppending* field is used as the expected value for the CAS that attempts to flag *gp* on line 81. Any operation that modifies a child of *gp* between these two lines must first flag *gp* and, ultimately, unflag *gp* by writing a *new* Clean object, different from the one stored in *gppending*. Thus, if any child pointer changes while the non-empty children are being counted, the flagging step on line 81 will fail, and the Delete will be retried.

It is also simple to obtain *other* on line 127; since the children of a marked node never change, and a mark is never removed once it is stored, and *op.p* is marked when HelpMarked is called, the children of *op.p* can safely be checked in sequence.

**x.CAS(expect, value)**
1: result := read(x)
2: if ( result $\neq$ expect ) return result
3: if ( x.CASET(expect, value) ) return expect
4: return read(x)

Figure 12: A linearizable implementation of CAS using CASET.

# 3 Java Implementation

The first implementation hurdle to overcome was the lack of a compare-and-swap primitive in Java. Instead, a compare-and-set primitive is exposed by the `java.util.concurrent.atomic` package. A call to $x$.CAS(*expect*, *value*) atomically: reads $x$, compares its value to *expect*, updates $x$ in the event of a match, and returns the value initially read. A call to $x$.CASET(*expect*, *value*) is identical save for its return value. A CASET returns TRUE if the update was successful, and FALSE otherwise.

It is a straightforward transformation to take an algorithm $A$ that uses CAS and produce one that uses only CASET so long as $A$ does not suffer from the ABA problem. Given such an $A$, each CAS operation need only be replaced by the linearizable subroutine in Figure 12. To see that this transformation applies to our Java implementation observe that all CAS operations on *pending* fields and, in all updates except for pruning deletion, all CAS operations on child pointers, store references to newly allocated objects. For these cases, there is no ABA problem; at the moment when process $P$ invokes the *new* operator and acquires a reference to a memory address $R$, no other process can hold a reference to $R$, or else it could not have been de-allocated memory. In the case of pruning deletion, CASET operations are equivalent to CAS operations, since the return value of the CAS is immediately discarded. In our code, we further simplify this transformation by using the fact that every time $x$.CAS(*expect*, *value*) is invoked in our algorithm, the value *expect* has previously appeared in $x$. Given this condition, we can show that it suffices to perform a CASET and read $x$ afterwards.

Java also provides two varieties of CASET. The first wraps an object $x$ in an AtomicReference object $y$. All reads and writes on $x$ go through $y$. Our implementation uses the second variety, which was introduced to circumvent the performance penalty induced by this indirection. One AtomicReferenceFieldUpdater object is created to protect all instances of a class field. This means that, for a BST, there would be one object to provide CASET for *all* left child pointers, and one for *all* right child pointers. An invocation of CASET takes three arguments: the object that contains the field to be updated, the expected value, and the new value. We found this variety of CASET to be significantly faster in our experiments.

In the interest of flexibility, our implementation parameterizes the type of keys (this generality was preserved in the experimental setup). In order to avoid the overhead of loop counters and array indexing arithmetic, $k$ was made a constant. A utility was written to generate the code for a $k$-ST given a particular $k$. Many standard optimizations were also performed. The SEARCH operation was inlined in FIND, INSERT, and DELETE. Polymorphism was avoided, and methods and fields were made final where possible. For correctness, all child pointers and *pending* fields were declared as volatile (to prevent erroneous caching by individual threads).

The special key $\infty$ was defined to be NULL. This eliminated the need for a special expanded class of keys (and the inherent polymorphism and indirection), but ruled out the use of NULL as a key in the dictionary. Incidentally, this was also the decision made in the design of the Java class library's ConcurrentSkipListMap.

In the $k = 2$ case, our algorithm degenerates into the BST of Ellen et al. [4] (with slight modifications to the top of the tree). Since, in this case, all insertions are sprouting insertions and all deletions are pruning deletions, we optimized further by eliminating dead code that performed simple insertion and simple deletion. Additionally, the code that maintains the keys of a node in sorted order was eliminated (each node contains only one key).

# 4 Experiments

## 4.1 Setup

We conducted a set of experiments to test the performance of Ellen et al.'s lock-free BST and our lock-free $k$-ST. We compared our 4-ST and BST implementations with the ConcurrentSkipListMap (SL) from the Java class library and the concurrent lock-based AVL tree of Bronson et al. [3] by running the same performance measurements for each implementation. In each implementation, an insertion inserts the new key and value to the data structure only if the key is not already present in the data structure. An insertion returns FALSE if the key is found in the data structure.

An experiment is determined by its insert-delete ratio and the range of keys that can be inserted into the dictionary. We present results for six different insert-delete ratios and three different key ranges. The ratio numbers are presented in percentage form; that is, 2i-8d means 2% of the operations are insertions, 8% are deletions, and the remaining 90% are finds. The first insert-delete ratio is 0i-0d, representing the environment where no updates occur. The two next ratios are 2i-8d and 8i-2d. The former represents an environment where we have more deletions than insertions and the latter captures a situation where there are more insertions than deletions. Finally, the last three ratios are 5i-5d, 25i-25d, and 50i-50d. Each represents an environment where there are a similar number of insertions and deletions. These ratios also measure performance when we have a low, medium, and high number of updates.

In addition to varying the operation-ratios, we also set out to compare the behaviour of the algorithms when their structures were limited to certain sizes (they could contain only a certain number of keys). We used three different key ranges to set an upper bound on this size: $10^2$, $10^4$, and $10^6$. This allowed us to measure performance under various levels of contention. For example, when the key range was of size $10^2$, the data structure remains very small, so it is much more likely that threads would be accessing the same portion of the structure simultaneously. For each experiment, we computed results over nine different numbers of threads: 4, 16, 32, 48, 64, 80, 96, 112, and 128, running 17 trials for each case. The duration of each trial was three seconds. The average throughput of each algorithm was then computed as the average total number of operations completed per second.

Each thread used a pseudo-random number generator (RNG) to randomly choose its sequence of operations and keys according to the experiment's insert-delete ratio and key range. In each trial, the algorithms were given the same set of seed numbers, where each seed is used to initialize the RNG of a thread. The set of seed numbers given was different in each trial. Since all algorithms were given the same initial seeds, the sequence of operations and keys generated were the same for each. Initially, the data structure was empty for each trial, except for the 0i-0d ratio. In this particular case, we pre-filled the data structure by running the algorithm with the 50i-50d ratio until the data structure size stabilized. With the 50i-50d ratio, the data structure is expected to stabilize to half-full, which means that the data structure contains about half of the possible keys. We stopped pre-filling the data structure when its size was within 5% of half of the key range. We did this pre-filling step before each trial, so trials have different initial data structures.

Since the Java Virtual Machine (JVM) performs optimizations on code at runtime, we omitted the first few trials of each experiment so that the JVM could "warm up." Our experimental results showed that the throughput for repeated trials stabilized after the first two or three seconds; so we omitted the first two trials (which sum to six seconds) from each experiment. We also prevented Java's garbage collection mechanism from activating in the middle of a trial to avoid its haphazard impact on throughput measurement (its activation and performance are subject to great variation due to the randomness of our trials). To this end, we manually triggered garbage collection before each trial and pre-allocated a very large heap so that a trial could run without any risk of experiencing memory pressure.

The experimental suite was executed on two different machines. The first was a machine at Intel's Manycore Testing Lab. This machine had four Intel Xeon X7560 CPUs and 16GB RAM. Each CPU had eight 2.26GHz cores, and each core had two hardware contexts. However, since hyper-threading was disabled, the system was only able to execute 32 threads in parallel (out of the 64 theoretically possible). The system permitted us to run approximately 150 threads. For experiments on this machine, we fixed the number of threads of our experiments at four per core (128 threads when the number of cores is 32) and varied the

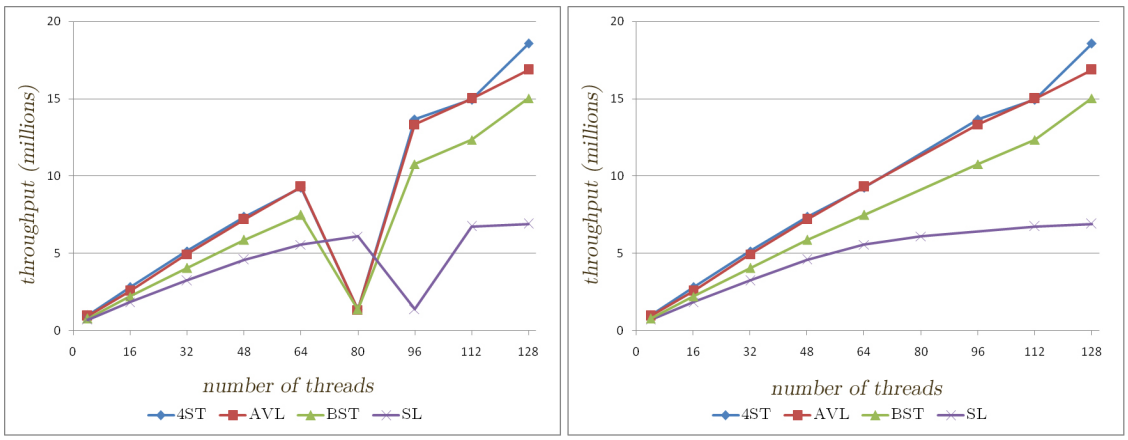



Figure 13: Result from the first run on the Intel machine, 25i-25d ratio and $10^6$ key range. Figure (a) shows the original data, Figure (b) shows the graph after removing the anomalous points.

number of active cores. This was done to test how well the algorithms scale with the number of cores. The operating system was RedHat Linux Enterprise Edition 5.4 kernel 2.6.18-164.el5. All experiments on this machine were run in server mode on the Sun Java 64-bit Virtual Machine version 1.6.0_20-b02 with the -d64 flag enabled and a 10GB initial (and maximum) heap.

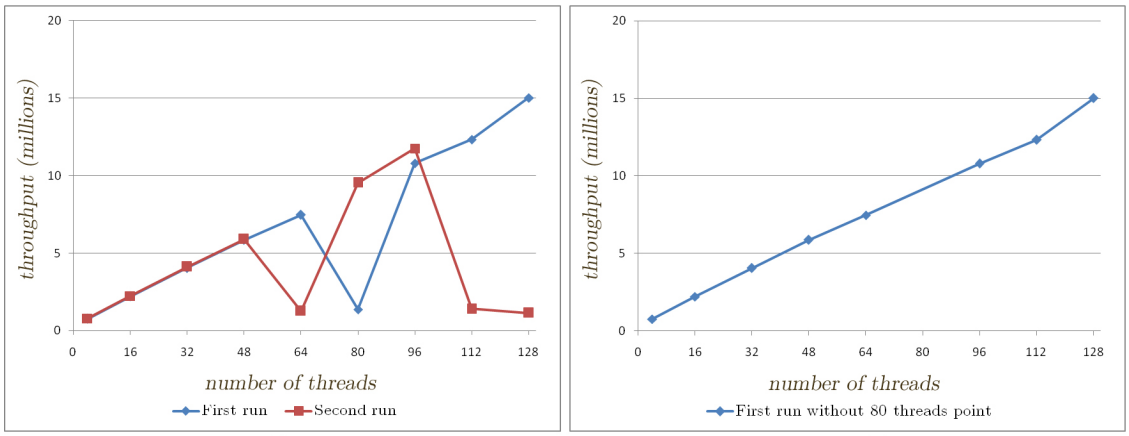
The other machine was a Sun machine at the University of Rochester. This machine had two Sun UltraSPARC-III CPUs and 32GB of RAM. Each CPU had eight 1.2GHz cores, and each core had 8 hardware contexts (meaning each CPU could execute 64 threads in parallel). The system did not allow us to change the number of active cores, so we only varied the number of threads from 4 to 128. All of these experiments were run on 16 cores. The operating system was Sun's Solaris 10. All experiments on this machine were run in server mode on the Sun Java 64-bit Virtual Machine version 1.6.0_21 with the -d64 flag enabled and a 15GB initial (and maximum) heap.

### Results from the Intel machine

Our experiment suite consists of six different ratios, three different key ranges, and nine different thread counts. It takes a considerable amount of time to run the suite, so we divided the suite into several small batches. From the first set of results of the Intel machine, we observed anomalies for several different thread counts. SL suffered throughput degradation on 96 threads. All other algorithms also suffered from the same problem, but for different thread counts, namely 128 threads when the ratio was 0i-0d and 80 threads for all other ratios. Moreover, the throughputs observed for those anomalous cases are very similar for all algorithms. Figure 13a shows the graph for the 25i-25d ratio and $10^6$ key range case.

We re-ran the whole experiment for a second time and got another set of data that also contains some anomalous points. However, in this set of data, the anomalies occurred for 64, 112, and 128 threads. We compared both data sets and found that, other than the anomalous points, they match nicely. Figure 14a shows the comparison of BST's data from the first run and second run for the 25i-25d ratio and $10^6$ key range case.

It is hard to imagine that the anomalous data points accurately reflect the true performance of the algorithms for several reasons. Firstly, the problem occurs for all trials of all algorithms that were run in the same batch, and all anomalous points have a very similar throughputs. Secondly, we believe that the anomalous points are not caused by our test harness, since we ran the same experiments on the Sun machine, but did not encounter any anomalies. Finally, since the non-anomalous points match nicely between the two runs, it shows that the non-anomalous results are reproducible. It may be that the anomalous points were




(a) (b)

Figure 14: (a) Comparison of BST's throughput taken from the first and the second run of experiment on the Intel machine, 25i-25d ratio and $10^6$ key range. (b) BST's throughput after removing the anomaly point (80 threads).

caused by some hardware problem. However, it is unclear to us what kind of hardware problem might cause this behaviour. We produced our graphs for the Intel machine by choosing the experimental data with the fewest number of anomalous points (the first run), and removing those points. Figure 13b and Figure 14b show the resulting graphs after removing the anomalous points from Figure 13a and Figure 14a.

## 4.2 Experimental Results

We present a full set of graphs for the six different insert-delete ratios in Figure 15 through Figure 20. Error bars are drawn on our graphs, showing one standard deviation. In most cases of the Sun machine's graphs, the bars are too small to see. However, there are several cases in the Intel machine's graphs in which the standard deviation is considerable.

The graphs in the left column of each page represent the results from the Intel machine, while the graphs in the right column represent the results from the Sun machine. For experiments using the key ranges of size $10^4$ and $10^6$, we present each graph for the Intel machine with the same y-axis scale as the corresponding graph for the Sun machine. However, for the key range of size $10^2$, we used different scales for the y-axis to make the measurements more readable. When the key range was $10^2$, all algorithms achieved about twice as much throughput on the Sun machine as on the Intel machine, except for the 0i-0d case, where they achieved about 25% more throughput on the Intel machine. For the larger key ranges, 4ST and BST generally had similar throughput on both machines, but AVL and SL did not. AVL performed better on the Intel machine, but SL performed better on the Sun machine.

Although we present the graphs side by side for easy comparison, recall that the Intel machine had 32 cores and was only able to run one hardware thread on each of them. On the other hand, the Sun machine had fewer cores (16), but each core could run up to eight hardware threads. Also, all experiments on the Sun machine were run with 16 active cores, but the Intel machine's experiments were run on varied numbers of active cores.

### $10^2$ key range

There were three different key ranges in our test suite. The $10^2$ key range is presented in the first row of each page of graphs, followed by $10^4$ and $10^6$ in the second and third row. The $10^2$ key range case represents environments with very high contention. There are at most $10^2$ keys in the dictionary, and as many as 128

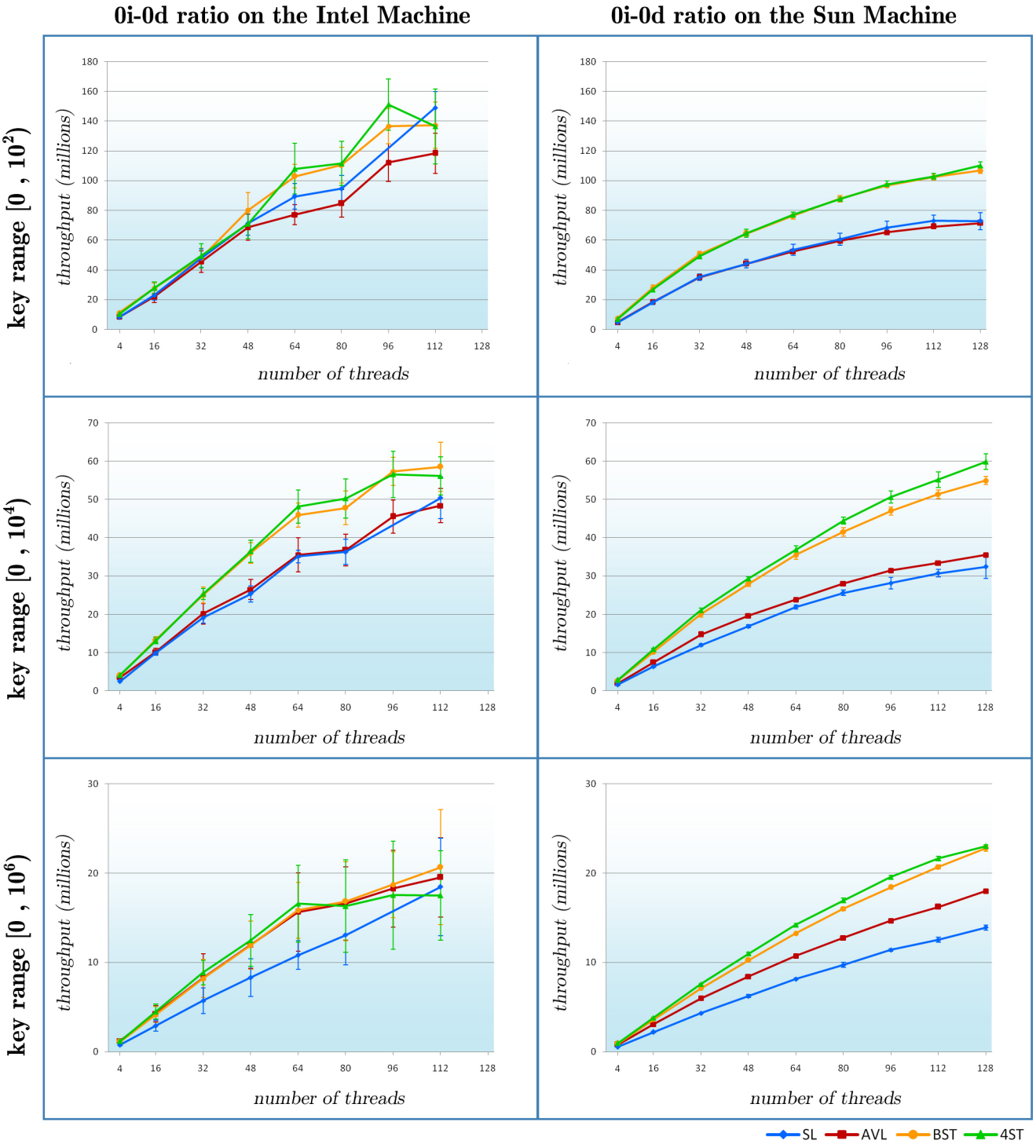



Figure 15: Experimental result for 0i-0d ratio

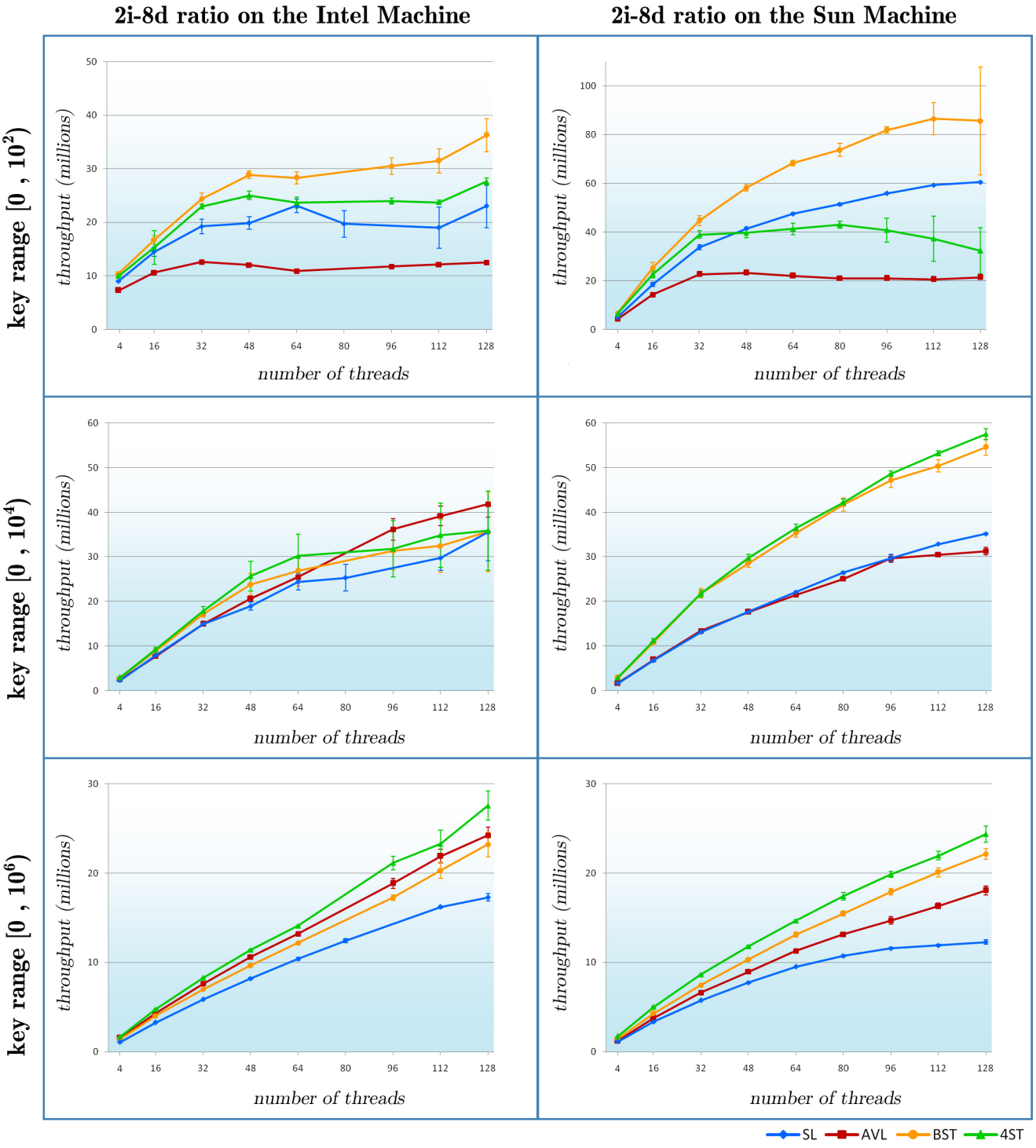



Figure 16: Experimental result for 2i-8d ratio

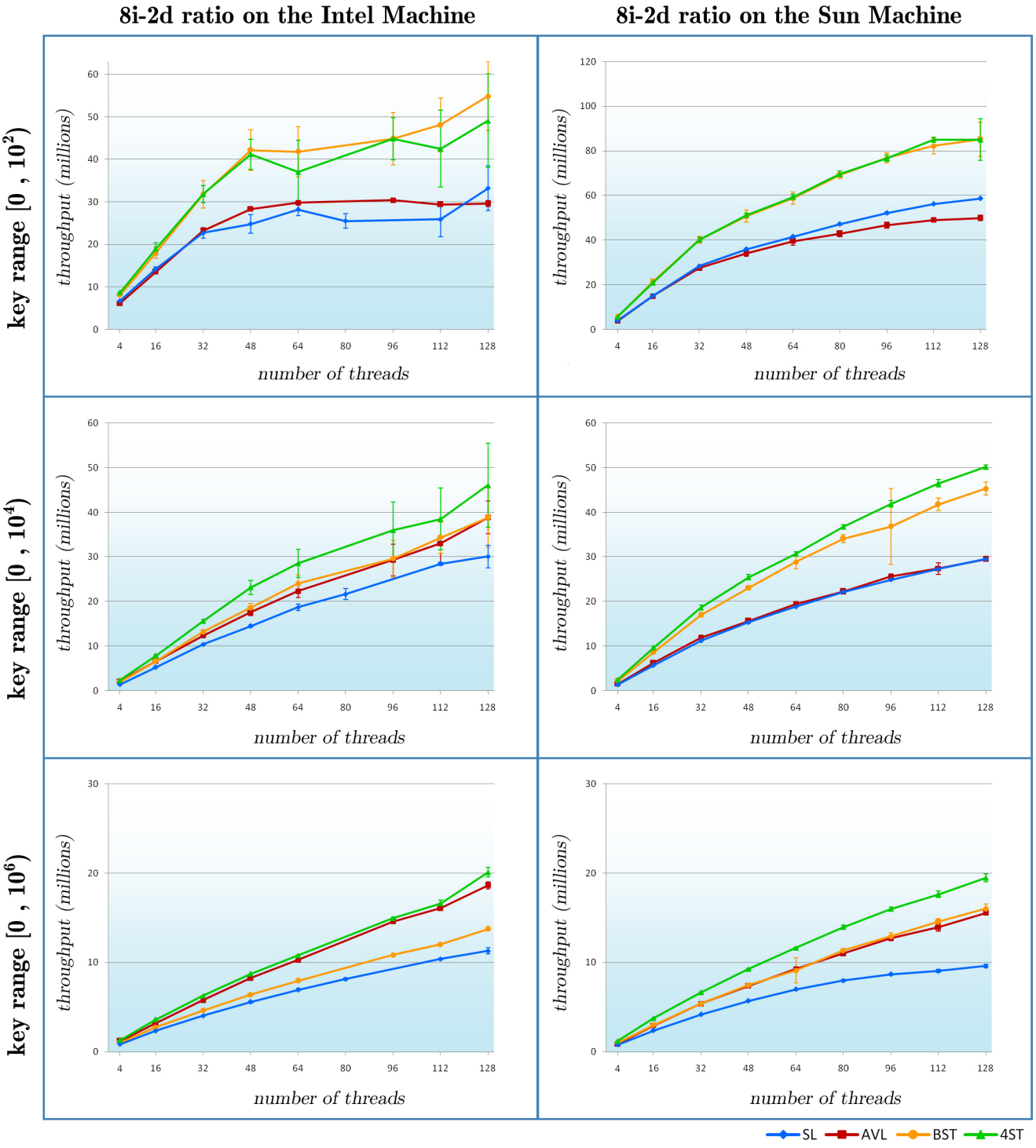



Figure 17: Experimental result for 8i-2d ratio

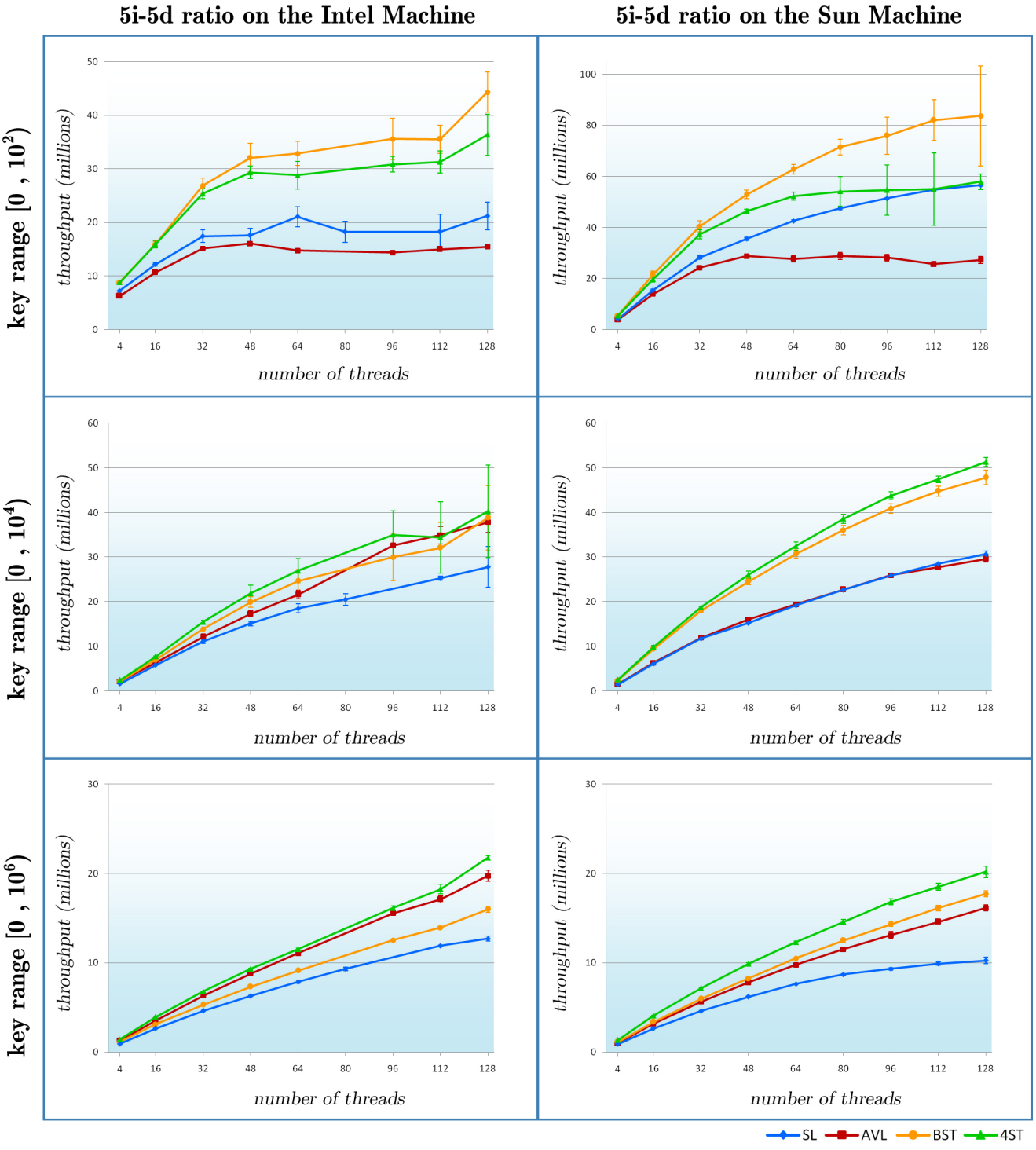



Figure 18: Experimental result for 5i-5d ratio

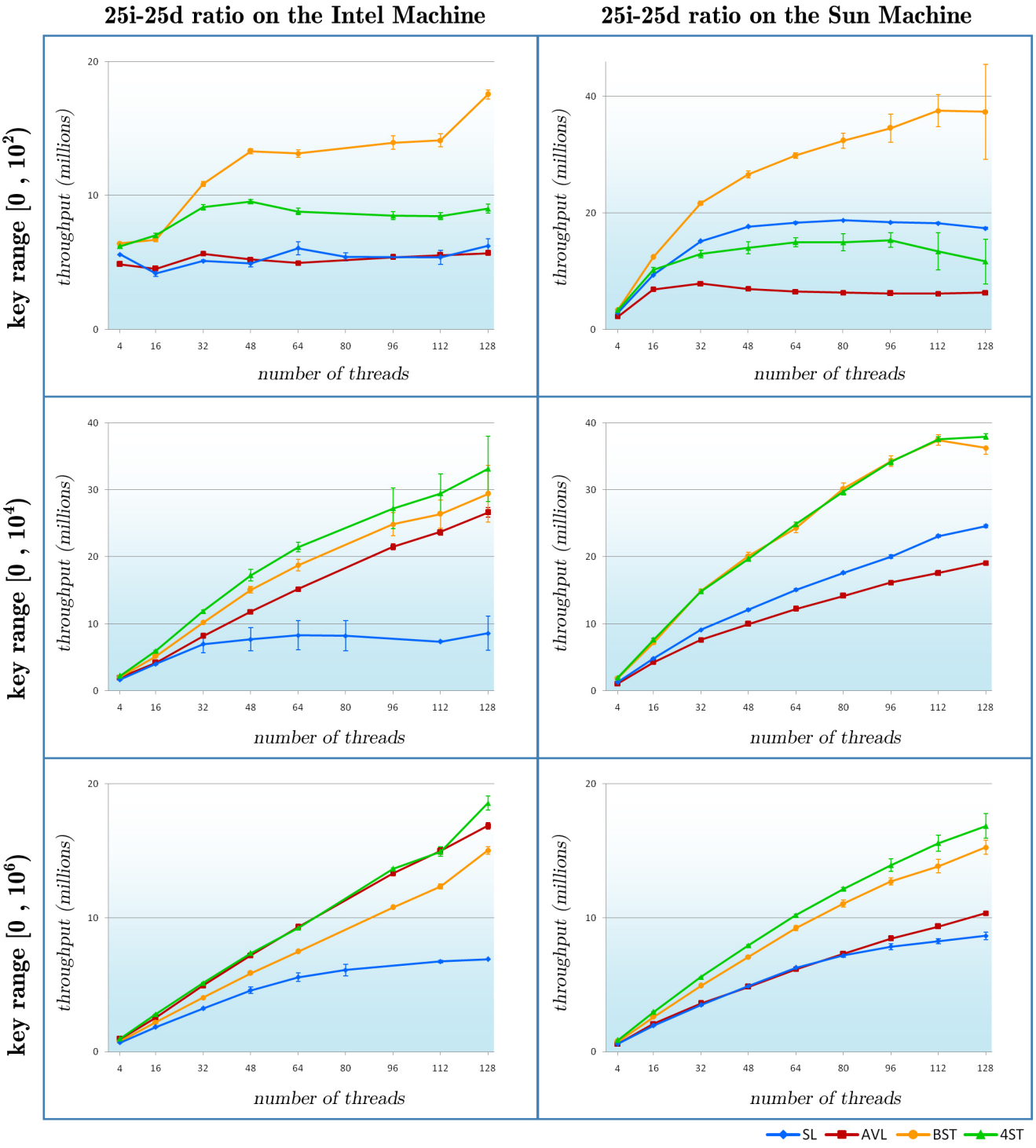



Figure 19: Experimental result for 25i-25d ratio

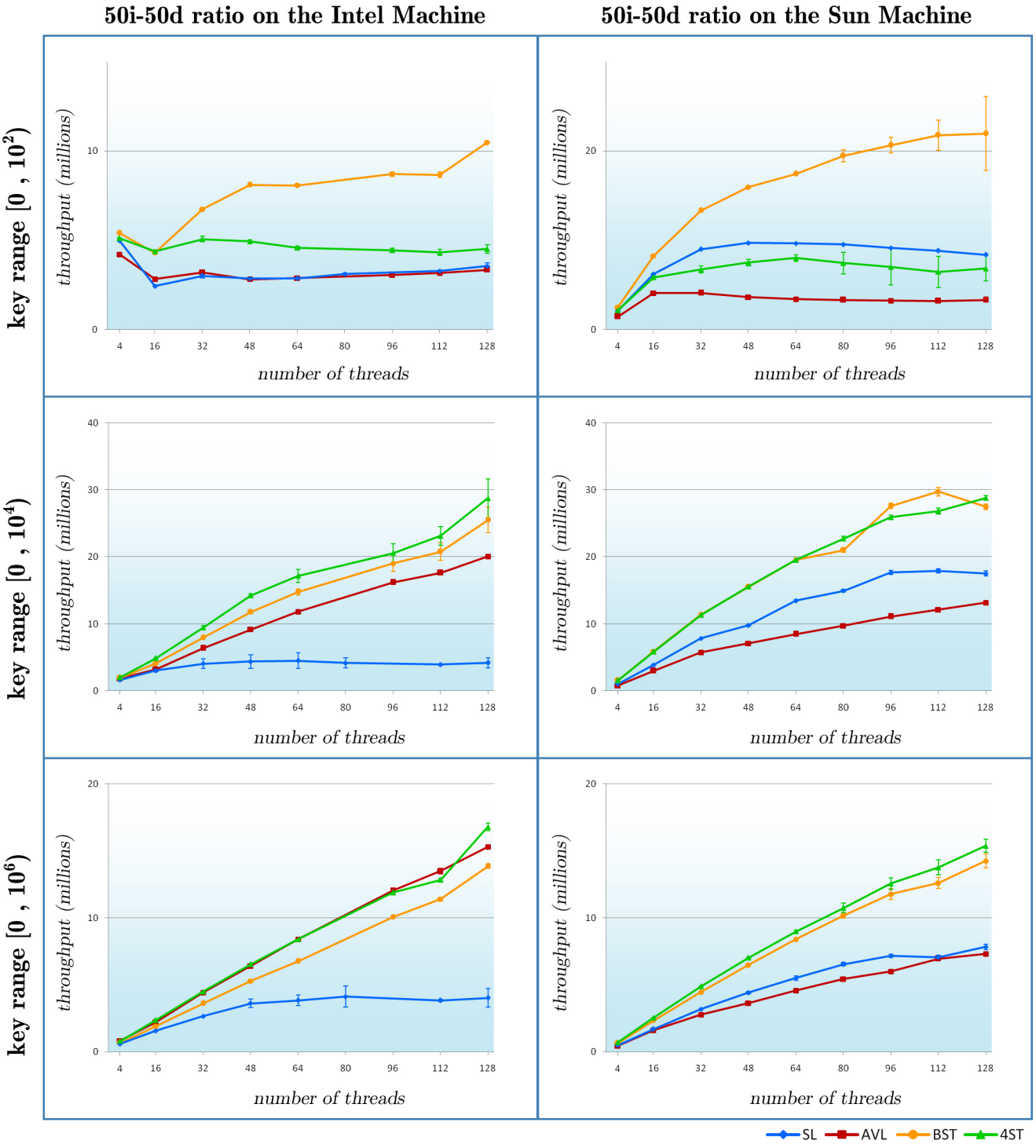



Figure 20: Experimental result for 50i-50d ratio

threads accessing the tree. In this case, BST was the top performer in all experiments. BST's low degree is advantageous, permitting many simultaneous updates to different parts of the tree.

In most cases, 4ST had similar or lower throughput compared to BST. It outperforms SL and AVL except for some cases on the Sun machine (2i-8d, 25i-25d and 50i-50d), in which SL outperforms 4ST.

We also observe that all algorithms flattened out after 48 threads on the Intel machine, except the 0i-0d case where we do not have any update operations. However, we do not see the same phenomenon on the Sun machine. BST scaled very well in all cases. We still see 4ST, AVL and SL flattening out in some cases on the Sun machine, but generally they scaled better on the Sun machine, compared to the Intel machine.

#### $10^4$ and $10^6$ key ranges

In the $10^4$ key range experiments, 4ST is generally the top performer, followed by BST. In some cases, BST got very similar throughput to 4ST, and in the other cases it has slightly lower throughput. As the tree gets bigger, the shallower tree of 4ST starts to gain an advantage over BST. 4ST and BST outperform both AVL and SL in all experiments, except for three cases on the Sun machine with low numbers of updates (2i-8d, 8i-2d, and 5i-5d). In these cases, 4ST, BST, and AVL perform similarly. Unlike the $10^2$ key range case, all algorithms scale nicely in the $10^4$ case. However, SL still flattens out quickly for the 25i-25d and 50i-50d ratios on the Intel machine.

When the key range is of size $10^6$, 4ST is also the top performer. The ordering between 4ST, BST, and SL is similar to the ordering in $10^4$ key range case. The only difference is that in some of the $10^4$ key range cases, BST has a very similar throughput to 4ST, but in this $10^6$ key range case, it is clear that 4ST outperformed BST. However, AVL's position relative to the other algorithms changed significantly. On the Intel machine, AVL outperforms BST and it is comparable to 4ST. Although it does not outperform BST on the Sun machine, AVL surpasses SL's throughput, except for the 50i-50d ratio.

As the tree gets bigger, the time required to execute each operation also increases. So we expect to get lower throughput when the key range increases. However, the nature of each algorithm also affects how fast their throughput decreases. 4ST and AVL's throughput decreases more slowly compared to BST and SL's.

#### Comparison between different ratios

We have chosen six different ratios to capture various situations. The 0i-0d ratio is the simplest case, where the data structure is never updated. This means that there is no write contention on the data structure. In this case, all algorithms scale nicely on both machines. The standard deviation from the Intel machine is quite high, so it is hard to tell which algorithm performs better. However, we can clearly see from the Sun machine's graph that 4ST and BST outperforms AVL and SL.

The next two ratios (2i-8d and 8i-2d) capture the environment with more insertions and more deletions respectively. Both ratios have the same total percentage of updates. The 2i-8d case has fewer insertions, so we expect a smaller data structure compared to the 8i-2d case. For this reason, it makes sense that the throughput in the 2i-8d case is slightly higher than the 8i-2d one. In the $10^2$ case, BST and SL had similar performance in both machines, but this is not the case for 4ST and AVL. Their throughputs are considerably lower for the 2i-8d ratio compared to the 8i-2d ratio. Recall that we ran the experiment on the Intel machine twice. This phenomenon occurred in both of the Intel machine's runs, as well as on the Sun machine's run, so it is likely due to the algorithm itself. Their throughputs for the 5i-5d ratio are nicely sandwiched between 2i-8d's and 8i-2d's throughput.

The last three ratios (5i-5d, 25i-25d, and 50i-50d) present results when we have low, medium, and high numbers of updates to the data structure. Since the update operations are more complicated than the find operation, it takes them longer to finish. As expected, as the percentage of updates increases, the observed throughput decreases. For example, BST's throughput for the 50i-50d is about 50% of its throughput for the 25i-25d, and about 25% of its throughput for the 5i-5d ratio. We observed that the throughput of 4ST and BST decreases similarly on both machines. However, this is not the case for AVL and SL. AVL's throughput decreases faster on the Sun machine, while SL's throughput decreases faster on the Intel machine. This phenomenon causes the two machines to have a different ordering of SL and AVL when the ratio is 50i-50d.

## 4.3 Additional Experiments

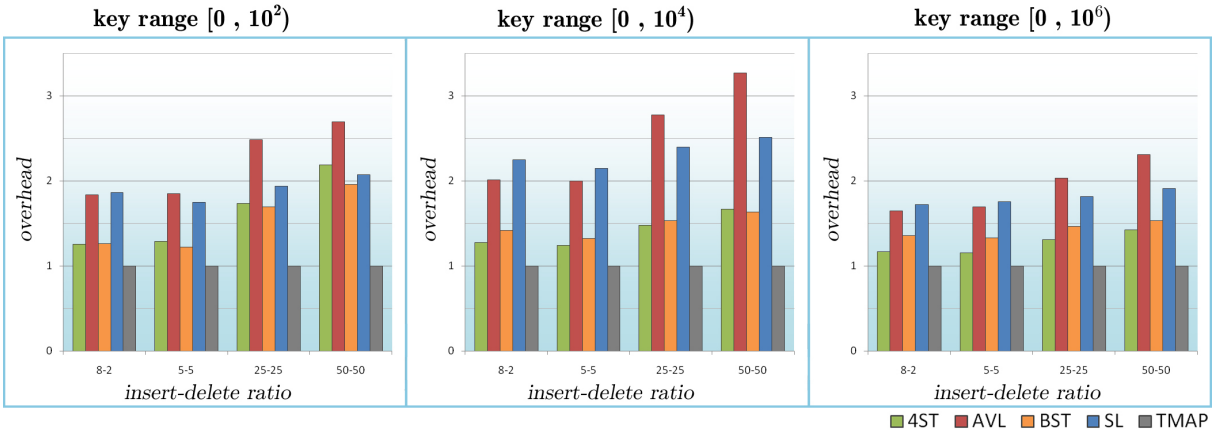



Figure 21: A comparison of the sequential performance of SL, AVL, BST, and 4ST relative to the Java class library's TreeMap (which is scaled to 1.0).

### Comparison with sequential TreeMap

In addition to the main experiments, we also performed several smaller measurements. Firstly, we measured the sequential overhead of all algorithms in comparison with the sequential TreeMap (TMAP) from the Java class library. Using the experimental setup described in Section 4.1, we measured the performance of TMAP and the four other data structures using only one thread. The throughput of TMAP was normalized to 1.0, and we computed the overhead of the other algorithms as $\frac{throughput\ of\ TMAP}{throughput\ of\ algorithm}$. This experiment was run on the Sun machine. Figure 21 shows the overhead associated with each data structure in the 2i-8d, 5i-5d, 25i-25d, and 50i-50d ratios, with $10^2$, $10^4$, and $10^6$ key range cases, respectively. Both BST and 4ST have less overhead than SL and AVL, in all cases, except on the 50i-50d ratio and $10^2$ key range case, in which 4ST has higher overhead than SL. In the case of the $10^2$ key range, BST has slightly lower overhead than 4ST. As the tree gets larger, 4ST's overhead decreases and it outperforms BST in the $10^4$ and $10^6$ key range cases.

### Throughput of 4ST over various kinds of insert-delete ratios

Figure 22 shows average throughput of 4ST over various insert-delete ratios. All experiments were done on the Sun machine using the $10^6$ key range and 64 threads. We varied the ratio of insertions from 0% to 100% with 2% intervals, then varied the ratio of deletions in the same manner, while keeping the total ratio not exceeding 100%. If the total insert and delete ratio do not sum up to 100% then the remainder are find operations. As we see in the graph, the throughput of 4ST increases as the number of updates decreases.

### Tree size vs $k$ value

In the main experimental results, we can see that 4ST gains an advantage over BST as the dictionary size increases. However, large $k$ values result in more overhead as nodes are created, searched and modified. Figure 23 compares the throughput of $k$-ary search trees for different $k$ values when the ratios are 2i-8d and 50i-50d, the key range is of size $10^6$, and the number of threads is 64. This experiment was done on the Sun machine. In this particular setting, the top performer is 4ST on the 2i-8d ratio, and 3ST on the 50i-50d ratio.

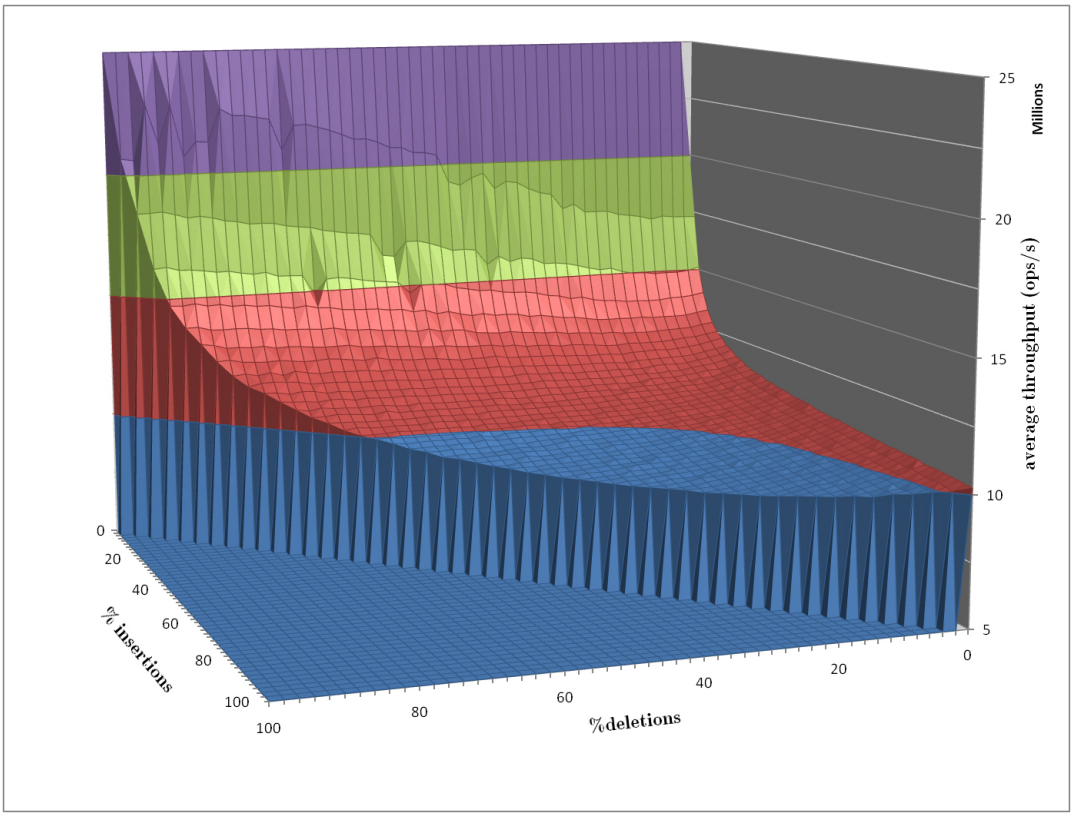



Figure 22: Throughput of 4ST over various kind of insert-delete ratios. The x- and z-axis shows the probability of deletions and insertions respectively, while the y-axis shows 4ST's throughput in millions. The experiment was done on the Sun machine, with $10^6$ key range and 64 threads.

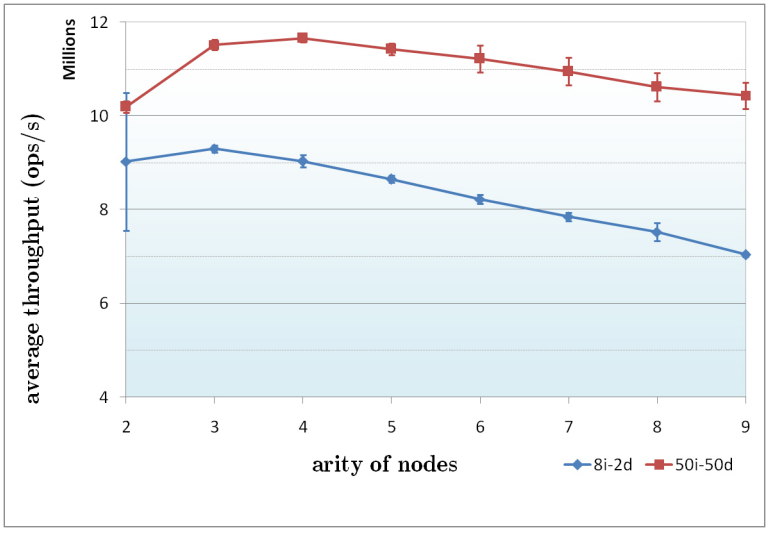



Figure 23: Comparison of the throughput of $k$-ary search trees with ratios 2i-8d and 50i-50i, and key range $10^6$, as $k$ varies between 2 and 9.

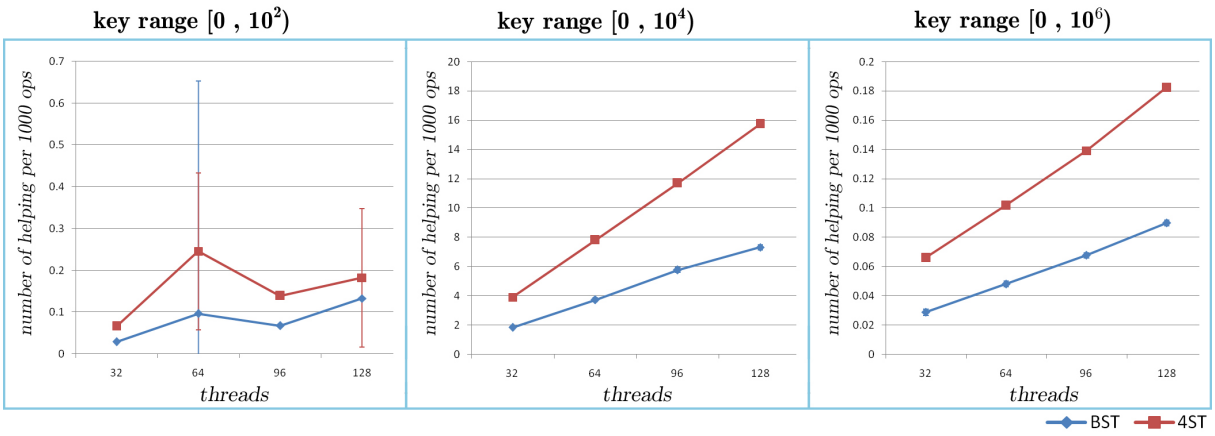



Figure 24: Average number of helping per operation when the ratio is 50i-50d (per 1000 operations).

### The number of helping

We are also interested to compute the average amount of helping done by each operation. We ran two experiments on the Sun machine with insert-delete ratios: 5i-5d and 50i-50d, and varied the number of threads to be: 32, 64, 96, and 128. The number of times an operation helped another operation observed was very small. Figure 24 shows the average amount of helping per 1000 operations in the 50i-50d ratio when key range is of size $10^2$, $10^4$, and $10^6$. We do not include the graphs for the 5i-5d ratio because all of them have a very similar shape to the 50i-50d ratio graphs, only they have a considerably smaller helping amount.

## 5 Conclusion and Future Work

We have seen that for both 4ST and BST, the performance on the Intel machine was similar to the Sun machine. AVL performs better on the Intel machine, and SL performs better on the Sun machine.

BST gains the most advantage in a high contention setting. Its simplicity pushes its performance beyond all other algorithms. As the tree gets larger (and the contention decreases), 4ST outperforms BST and becomes the top performer. Similar to 4ST, AVL also performs better when the data structure size increases. On the Intel machine, AVL is comparable to 4ST when the key range is $10^6$. The same phenomenon occurs on the Sun machine as well, although the increase of AVL's performance is not as high as it is on the Intel machine. In case for SL, it performs better on smaller data structures.

In our experiments, we manually triggered garbage collection before each trial to avoid its impact to the throughput measurement. But in practice, memory management is an important aspect of an algorithm. In our future work, we would like to test the impact of garbage collection on the performance of various algorithms.

AVL is a balanced tree, so it does some extra work for maintaining this property. However, since our experiment inserts random keys to the dictionary, 4ST and BST also are nearly balanced. So in this setting, the balancing work of AVL does not pay off. In a situation where the keys inserted are not random, AVL might have a significant advantage over 4ST and BST. Since our BST and 4ST outperform AVL and SL by a fair margin, we believe that it may be possible to add balancing and stay competitive, while offering a non-blocking progress guarantee.

# 6 Acknowledgment

We thank the team at the Intel's Manycore Testing Lab for providing us access to their machine, as well as helpful technical support during our experiments. We also thank Michael L. Scott for letting us do experiments on the multi-core machine at the University of Rochester. Financial support for this research was provided by the Natural Science and Engineering Research Council of Canada.

# References

[1] Greg Barnes. A method for implementing lock-free data structures. In *Proc. 5th ACM Symposium on Parallel Algorithms and Architectures*, pages 261–270, 1993.

[2] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Bradley C. Kuszmaul. Concurrent cache-oblivious B-trees. In *Proc. 17th ACM Symposium on Parallel Algorithms and Architectures*, pages 228–237, 2005.

[3] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *Proc. 15th ACM Symposium on Principles and Practice of Parallel Programming*, pages 257–268, 2010.

[4] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *Proc. 29th ACM Symposium on Principles of Distributed Computing*, pages 131–140, 2010. Full version appeared as Technical Report CSE-2010-04, York University.

[5] Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *Proc. 23rd ACM Symposium on Principles of Distributed Computing*, pages 50–59, 2004.

[6] Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Transactions on Computer Systems*, 25(2):5, 2007.

[7] Keir A. Fraser. *Practical lock-freedom.* PhD thesis, University of Cambridge, 2003.

[8] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *Proc. 19th IEEE Symposium on Foundations of Computer Science*, pages 8–21, 1978.

[9] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proc. 22nd ACM Symposium on Principles of Distributed Computing*, pages 92–101, 2003.

[10] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[11] Håkan Sundell and Philippas Tsigas. Scalable and lock-free concurrent dictionaries. In *Proc. 19th ACM Symposium on Applied Computing*, pages 1438–1445, 2004.

[12] John D. Valois. Lock-free linked lists using compare-and-swap. In *Proc. 14th ACM Symposium on Principles of Distributed Computing*, pages 214–222, 1995.