Implementation and Evaluation of Concurrent Binary Search Trees on Many-core Architectures

Semester Project - Fall 2013

Oana Balmau and Igor Zablotchi

Ecole Polytechnique Fédérale de Lausanne (EPFL)
name.surname@epfl.ch

Abstract. This report presents a study of concurrent binary search trees. In contrast to other concurrent data structures, the parallelization of trees is more difficult, because multiple fields need to be updated atomically when performing insertion or removal operations. We discuss the most interesting issues faced during the implementation in C of two concurrent binary search tree algorithms: a lock-based relaxed-balance AVL tree proposed by Bronson et al. and a lock-free internal tree proposed by Howley et al. We proceed by comparing the performance of the two with a third lock-free external tree proposed by Ellen et al. on three different multi-core architectures, with a focus on scalability.

There are contexts in which each one of the trees is the most suitable choice. Its rebalancing mechanism, along with its internal tree structure make the Bronson tree ideal for read-dominated scenarios on sizeable lists. The simplicity of the Ellen tree gives it an advantage in write-dominated situations on small-scale lists and on architectures with simpler cores. For small and medium-sized lists, the Howley tree is an ideal candidate both in read and write intensive cases; we believe that enhancing it with a rebalancing procedure would make the Howley tree a suited choice for large lists as well.

1 Introduction

According to Moore's law, the number of transistors that can be fit in the same space on an integrated circuit doubles approximately every two years. The capabilities of electronic devices are strongly linked to Moore's law. Until recently (the 2000s), computational progress was achieved by the idea that more transistors translated to faster processors. However, this growth dimension eventually reached a plateau, because of the excessive overheating caused by clock speed augmentation. Nowadays, focus has shifted from speed to parallelism: instead of a single fast processor, modern architectures come with increasingly large clusters of processors, working together through shared memory. This shift from single to multi-processor architectures determined a foreseeable paradigm change in programming. Basic algorithms, optimized to be used by a single thread, needed to be adapted in order to efficiently exploit the parallelism of new architectures.

One of the main challenges of multi-threaded programming arises from the fact that synchronization between threads has to be achieved in a system that is asynchronous by construction. Threads can be delayed or interrupted in an unpredictable manner and the orders of magnitude of these delays can greatly vary. For tasks where collaboration is needed, threads typically synchronize through the use of data structures, stored in shared memory. Therefore, the implementation and design of these data structures is of paramount importance for the efficiency of concurrent algorithms.

Up to now, concurrent data structure algorithm design has been under the influence of two main paradigms: **lock-based** (or blocking) and **lock-free** (or non-blocking).

In the lock-based approach, critical sections are protected by locks. A critical section is a piece of code that accesses a shared resource, which must not be concurrently accessed by more than one thread. In order to enter a critical section, a thread first has to obtain its corresponding lock. As a consequence, a thread that is delayed (or that fails) inside a critical section could potentially stop all the other threads in the system from making progress. These delays can be of different natures, such as cache misses, page faults, or OS preemptions.

Ideally, every function call by any thread should finish in a finite number of steps. This property of concurrent algorithms is called **wait-freedom**. Wait-free algorithms exist, but are currently rather inefficient in practice. However, a different breed of algorithms is emerging. They belong to the second design paradigm mentioned above, the lock-free paradigm. An algorithm is lock-free (or non-blocking) if the delay of one or more threads does not delay all of the other threads, i.e. the system is achieving overall progress. This property is not as strong as wait-freedom, because it only guarantees the existence of at least one thread that achieves progress. Lock-freedom does not exclude the possibility that threads starve (are delayed indefinitely). The design of lock-free algorithms is often based on a **helping mechanism**: threads complete their own operations, if possible, or help the completion of any obstructing operations (initiated by other threads).

Non-blocking algorithms avoid lock-related issues, such as deadlocks, priority inversions (a thread with low priority preventing a thread with high priority from entering a critical section), contention problems, or convoying (a thread that is delayed or fails inside a critical section blocks a large number of other threads). However, because non-blocking algorithms increase concurrency (which in itself is a desirable property), thread interaction in the lock-free scenarios is more complex to analyze.

Concurrent binary search trees were the focus of this project for two reasons: their implementation is both a **relevant** and a **challenging** problem.

Firstly, the binary search tree is a fundamental data structure, applicable to various types of software projects. This is because trees can be used to implement an ordered set of keys or an ordered map from keys to values, two very common concepts in programming. These data structures can be implemented via linked lists or skip lists as well, but binary search trees often outperform these structures [1,3,7]. Therefore, the impact on performance of the choice between a lock-free or lock-based binary search tree is entirely relevant, not only as an academic curiosity, but also for developers of real-world applications and systems.

Secondly, trees are difficult to parallelize efficiently, both in the lock-free and lock-based paradigms. The problem is explained in more detail in Section 2, but the gist of it is that multiple fields — sometimes not part of the same tree node — need to be updated atomically while minimizing hindrance towards concurrent non-conflicting operations, which is not trivial to achieve.

The purpose of this project was to evaluate the impact that different design characteristics of BSTs have on performance, from a scalability standpoint. Our comparison is conducted over three different BSTs described in Section 2. We implemented one lock-free and one lock-based BST in C (proposed by Howley et al. [7] and by Bronson et al. [1] respectively) and integrated them in the SSYNC cross-platform synchronization suite [2]. We provide a detailed discussion concerning the more delicate problems encountered during the implementation phase in Section 3. The two algorithms were compared with a third concurrent BST, proposed by Ellen et al. [3], which was already in the SSYNC library. Scalability and correctness tests were run on three different multi-core architectures: a 48-core AMD Opteron, a 36-core Tilera TILE-Gx36 and an 8-core Sun Niagara 2. The outcome of this evaluation was that the choice of data structure (lock-based versus lock-free, internal tree versus external tree etc.) is tightly coupled with the access patterns imposed by the nature of the application and with the underlying architecture.

In general, lock-free implementations have a more effective mechanism of dealing with contention, so they tend to be better on small and medium sized lists. Furthermore, as the list size increases, we can observe the benefits of rebalancing mechanisms, as well as the structure advantages of internal trees over external trees: since the former do

not always need to go all the way to the leaves in order to access information, they are faster. However, maintaining the structure of internal trees involves more complex operations than for external trees. Therefore, depending on the list size, in write-intensive contexts, external trees may be the wiser choice, due to the simplicity of their add/remove operations. The low sequential complexity of these operations also make lock-free external trees better suited for architectures with simpler cores and many hardware threads.

2 Background

We begin by briefly describing binary search trees (BSTs) in the context of this project. A binary search tree is a type of binary tree which associates a key (and optionally, a value) to each node v such that: (1) all the keys associated to nodes in the left subtree of v are strictly less than the key of v and (2) all the keys associated to the nodes in the right subtree of v are greater than or equal to the key of v. Moreover, duplicate keys are not allowed in the data structure.

Binary search trees can be classified as internal or external BSTs. In an **internal** tree, all nodes contain a key-value association whereas in an **external** tree, only leaf nodes contain such an association. Non-leaf (internal) nodes are only used to direct searches towards the correct leaf and are thus aptly named routing nodes.

An **AVL tree** is a type of BST which satisfies the following invariant: for every node, the height difference between its right and left subtrees may differ by at most one. If an update operation causes this condition to no longer be satisfied, one or more rotations are triggered to re-establish the AVL invariant. Parallelizing this variant of an AVL tree poses a problem because any update operation must not only guarantee its own atomicity but also the atomicity of the rotation which may involve several nodes. This led to the introduction of relaxed balance trees, in which mutating operations and rebalancing operations are separated from each other: updates are allowed to violate the balancing condition which is subsequently restored by local applications of rebalancing operations.

Designing an efficient concurrent BST is a challenge, both in the lock-free and lock-based approaches. In the lock free case, one major obstacle is the inability to update multiple fields atomically. Linear data structures such as linked lists and skip lists have overcome this problem by a two stage mark-and-delete removal process [5]: in a first pass, a flag is atomically updated to mark the logical deletion of a node, and a subsequent pass will physically remove the node. However, in a BST nodes have two child pointers; thus a child pointer may be marked for deletion while the other child is updated, leading to the latter update not being visible. A work-around to this problem has been proposed in [4], using multi-word CAS, but this operation is not supported by most architectures, which makes this solution impractical.

Another problem that affects both lock-free and the lock-based internal BSTs is posed by remove operations. When a node having two children is to be deleted, a replacement needs to be found for that node: the node containing the next largest or the next smallest key. This replacement, however, might be several nodes away from the soon-to-be-removed node. This raises three challenges: (1) the changes to the two nodes need to appear atomic, (2) no other operation should update the tree between the replacement node and the removed node and (3) in the lock-free case, any read that has been invalidated by the removal needs to be detected and restarted. External trees avoid this problem, of course, at the cost of wasting memory on routing nodes: an external tree holding n keys will need $n-1$ routing nodes in addition to the n leaves.

We now provide an overview of the three BST algorithms that we evaluated:

- The tree proposed by Ellen et al. (*Ellen tree*) is an external lock-free BST. An insertion replaces a leaf by a subtree of three nodes (a new routing node, a new leaf holding the key to be inserted and the old leaf), and a deletion removes a leaf and its parent by making the leaf's sibling a child of its former grandparent.

When modifying a node, it is flagged with a pointer to a structure containing all the information necessary to complete the update, in case the initiating thread fails or is delayed. The child pointers of a node thus marked cannot be modifed until the pending operation completes and the node is unflagged. A process helps complete the operation of another process if and only if the other operation is preventing its progress, to avoid doing unnecessary work.

- The tree proposed by Howley et al. (*Howley tree*) is an internal lock-free BST. It uses the same cooperative update mechanism as the Ellen tree (nodes are marked with a pointer to a structure containing all necessary information needed to complete the ongoing operation on that node; marked nodes also prevent all competing operations from updating them until completion), but adapts it to the internal tree structure. When inserting a key, the node gaining a new child is marked; the physical update can then be performed by any thread using the supplied information, by CASing the pointer to the new child into its future parent. Deleting a key with less than two children is done by marking it; any process that later encounters the marked node will excise it from the tree. The major contribution is the algorithm for deleting a node with two children. Say a node A needs to be removed; a search is initiated to locate the node B containing the next largest key (B cannot have more than one child in this case). A's key is replaced with B's key and B is removed from the tree. To do this, the data required to make the change is stored in a structure and a pointer to this structure is placed at B. Then an attempt is made to place the same reference at A. If this succeeds, A's key is modified and B is marked for deletion as a node with less than two children. If it fails, B is unmarked and the operation is restarted. When a replace succeeds, A's new key is larger than the old one, meaning that its right subtree has a reduced range of possible keys. Therefore, failed ongoing searches in that subtree are restarted.
- The tree proposed by Bronson et al. (*Bronson tree*) is a lock-based relaxed-balance AVL BST. Per-node locks are used to manage mutating operations. During a removal of a of a node with two children, the node's key is removed from the tree but the node remains in the tree and acts as routing node. Maintenance operations such as rebalancing and the removal of routing nodes are disconnected from update operations. An Optimistic Concurrency Control mechanism is used to allow readers to be invisible: per-node version numbers are used to signal when maintenance operations are on-going; readers optimistically assume that no mutation will occur during their critical sections, and restart if that assumption fails (the version number has changed during their critical section).

3 Implementation

For this project, we explored both design paradigms — lock-based and lock-free — by implementing the two binary search tree algorithms, described in Section 2. During the implementation phase, we have been concerned with two dimensions: correctness and scalability of the algorithms. In this section we first discuss what structure changes needed to be made to the algorithms to integrate them in the SSYNC library; secondly, we present the approach we used to test the BSTs for correctness; then, in the following two subsections we highlight the most interesting correctness and scalability related problems that occurred during this project.

3.1 Language-related adaptations

A first step towards integrating the two BSTs into SSYNC was to rewrite the algorithms in C. The Bronson tree implementation was adapted from the paper code, along with the Java code in a repository mentioned by the authors and the C++ implementation in [8]. The Howley tree implementation was based entirely on the C++ code provided in the paper. The basic versions of the algorithms were considered, without any of the supplementary

optimizations proposed in the papers (fast clone support for Bronson and fast traversal for Howley). Even though our implementation kept close to the pseudo-code in the papers describing the BSTs, there were still some adaptations that needed to be done when passing from an object-oriented programming style to a procedural programming style:

- Replacing classes with appropriate C `structs`. Classes with subclasses were translated to `unions`.
- For the Bronson tree, translating Java `synchronized` blocks to lock-based critical sections. Special care had to be taken when returning from inside a synchronized block: first compute the value to be returned, then yield the lock and finally return the computed value.
- Also for the Bronson tree, in a direct translation from Java, during some critical sections it was possible that the lock would not be accessible through the same reference it was locked from, because its referencing node had changed. Therefore, whenever a thread needed to acquire a node's lock, references to the lock were stored directly and not accessed via the node's own reference. This is different from Java, in which `synchronized` blocks automatically track the synchronization object.
- Performing memory allocations using a library-specific allocator, `ssalloc`, and not the standard C memory allocator. This was done in order to avoid contention problems caused by threads concurrently requesting memory from the OS. Every thread was initially allocated a private pool of memory large enough for all its objects and memory was only released back to the OS at the end of an execution. A notable detail concerning the SSYNC allocator is that a new allocator had to be defined for each type of object (for example, if we need both node and operation structures, two different allocators have to be used).

3.2 Testing for correctness

Correctness in a multi-threaded context is not as straightforward to check as for single threads. Therefore, algorithm testing was performed in a step-by-step approach. First, we checked if the yielded BSTs would have the correct structure when running the algorithm on a single thread. Then, we gradually increased the concurrency, mapping new threads to the cores (in general, one thread per core). When we were satisfied with the results of these preliminary tests, we passed to the next stage: this test would take the number of threads, the maximum number of elements in the data structure and the access pattern (searches, insertions, deletions percentages) as parameters. Initially, the data structure was half-filled by all the threads. Then, the algorithm would run for one second, when the threads would execute the given access pattern. In the end, the actual size of the trees was compared to their expected size (initial number of elements in the tree plus the sum of successful insertions by all threads minus the sum of successful deletions by all threads). Even if this test was not exhaustive, it was a simple method to spot problems in scenarios with large lists and many cores. Finally, the same tests were conducted in high-stress situations, such as many threads performing many updates on small-sized lists.

3.3 Correctness issues

One of the more interesting issues that we encountered while implementing the Bronson BST was dealing with "heisenbugs". These are bugs that seem to disappear when one attempts to study them (using debugging tools, for example). In our case, the algorithm was behaving correctly when the compiler optimization options were turned off (set to `O0`), but was blocking when the program was run with all compiler optimizations (`O3`). The explanation for this behavior was that when the compiler optimizations were turned on, the order of certain instructions was changed at compile time. While this does not affect a single-thread execution, it can lead to deadlocks in a multi-threaded context. More precisely, the compiler must not change the order in which locks are obtained and released.

This issue has been fixed by adding the `volatile` qualifier to the lock field of the node structure. In C, using the `volatile` qualifier indicates to the compiler that the variable might change state even if no instructions appear to alter it, thus disabling compiler-related optimizations for that particular variable. Along the same lines, apart from deadlocks, compiler reorderings could also lead to segmentation faults (for example a thread tries to access a variable that became `NULL` in the meantime, because of reorderings). As mentioned above, the solution was to make all fields that could be altered by other threads (such as node height, parent node, children nodes etc.) volatile.

The Howley BST implementation also revealed some interesting aspects when testing correctness. In addition to the reorderings made by the compiler, the cross-platform nature of our framework introduced the problem of different memory consistency guarantees for different machines. This was especially challenging when porting our lock-free algorithm to the Tilera (described in Section 4.1). The problem lied in the fact that when a processor needs to write values to main memory, since this is a costly operation, the values are not directly written, but marked as "dirty" and placed in a local buffer. At a later point in time, all the values stored in the buffer are written back to main memory. In our case, on the Tilera, we needed to force some of these writes to take place right away, to ensure the correctness of the algorithm.

This was achieved through instructions called memory barriers (or memory fences). A memory barrier triggers the writing to memory of the information stored in all write buffers at that point in time, so that the processor invoking the memory barrier can see all changes performed by other processors. Since memory barriers can be expensive operations (hundreds of cycles), we tried to use as few as possible in our Howley tree implementation: only two barriers were used, one before the CAS on the operation structure responsible for adding a node and one before the CAS on the operation structure responsible for removing a node. Note that the memory barriers were used immediately before the linearization points of the add and remove operations and not at the point in time when nodes were physically added to or removed from the tree (as these may not always coincide).

Related to the previous observation, these linearization points also need to be taken into account when performing auxiliary testing related operations, such as computing the size of the data structure. It does not suffice to check if the node exists physically in the structure, but one also needs to take into account the marked operation pointers. This misinterpretation of the linearization point was the reason why during testing we sometimes obtained a greater expected size than the actual size of the tree: some "ghost" nodes were still physically present in the tree (sometimes even duplicate) even after they had been logically deleted by being marked for removal (the linearization point of the remove operation is on the CAS that marks a node).

3.4 Scalability issues

Once the data structures behaved correctly, the next step was to make sure that they were efficient as well. Our goal was to improve the scalability of the two BSTs. As a sanity check, we looked at the cases where only searches were performed on the data structures. Because in read-only mode no resources are blocked during the execution, the algorithms were expected to scale linearly (n threads perform roughly cn times more operations than one thread for a constant $0 < c \leq 1$). Initially, this was not the case. For both trees, scalability was being hindered by false sharing. False sharing is a phenomenon that occurs when processors modify pieces of data that are stored on the same cache line, but are logically distinct. A modification of one of the data items invalidates the cache line, thus invalidating the traffic of other processors that may access other unchanged objects stored on the same cache line. In our case, false sharing was the cause of poor performance. However, having several items stored on the same cache line could be desirable in some contexts, since it allows exploiting data locality. Therefore, a trade-off arises between the likelihood of false sharing and good locality.

In general, false sharing can be avoided by aligning and padding objects that need to be accessed independently, so that they are stored on separate cache lines. In our implementation, the Howley tree had the nodes and the

operation structures aligned to the cache line size (64 bytes). Moreover, because the root is the most frequently accessed node, it was also padded. In the blocking BST implementation, all of the nodes were aligned and padded, in order to avoid false sharing on the locks (which were included as a field in the node structure). The padding was computed at compile time, as a function of the size of the lock.

Another important detail that needs to be taken into account for achieving good scalability is making sure that the tests do not impede the algorithms' performance (i.e. there is no false sharing occurring in the tests). In our case, all thread-related statistics were stored in an aligned and padded data structure. Moreover, the implementation of library-specific utility functions was checked to make sure they did not induce false sharing.

4 Evaluation

In this section we show a comparison of the three BSTs: the Howley tree, the Ellen tree and the Bronson tree. First, we describe the experimental setup, followed by a discussion of our most relevant results.

4.1 Experimental setup

The three BST algorithms were run on three different multi-core architectures: a 4-socket, 48-core AMD Opteron (*Opteron*), a 36-core Tilera TILE-Gx36 (*Tilera*) and an 8-core Sun Niagara 2 with 64 hardware threads (*Niagara*). For more detailed information about these platforms, please see [2].

Apart from the different architectures, we have varied two other dimensions: the key range (which also indicates the maximum size of the BSTs) and the access pattern (the percentage of insertions, removals and searches). The key ranges used were 2^{11} (roughly 2K), 2^{14} (roughly 20K), 2^{18} (roughly 200K) and 2^{21} (roughly 2M). Moreover, the chosen access patterns were varied from exclusively reads (100% contains), to exclusively writes (50% inserts and 50% removes), with a more realistic workload in between (90% contains, 5% inserts, 5% removes).

For each experiment run, the data structure was first filled up to roughly half of its capacity, concurrently and in equal measure by all threads. Then, the threads were started simultaneously (using a barrier) and were allowed to run for one second. The frequency of the operation types (searches, inserts and removes) for each thread followed one of the above-mentioned access patterns. The keys to be added, removed, or searched for, as well as the operation type, were generated at random at each step. Finally, the throughput of each thread was measured as the number of successful updates (inserts and removes) plus the number of searches completed before the thread was stopped.

Moreover, the Bronson tree has been tested with four different types of locks: the Pthread Mutex lock, the Pthread Spin lock and SSYNC implementations of two other varieties of spin locks, a Ticket and a Test-and-set lock. The main difference between mutex locks and spin locks is that in mutex locks, if a thread cannot enter a critical section, it sleeps until it can acquire the lock, while in the spin lock case, threads perform some type of busy waiting. A trade-off arises between the time it takes to send a thread to sleep and to wake it up when using mutexes and the suboptimal use of CPU cycles caused by spin locks.

4.2 Results

In this subsection we compare the throughput performance of the Howley, Bronson and Ellen trees with a focus on scalability. First we compare their performance on the three architectures (Opteron, Tilera and Niagara) and then we provide a more in-depth evaluation on a single representative architecture.

Figure 1 shows the overall throughput of the three data structures on the y axes and the number of threads on the x axes. The upper row shows the behaviour of the algorithms in the 10% update case, and the lower row does the same for the 100% update case. The key range was fixed at 200K. Note that the scales of both the x and the y axes vary with the architecture: the three machines have a different maximum supported number of concurrently executing threads, thus the throughput will also naturally not be in the same range.

When looking at the first two columns, corresponding to the Opteron and the Tilera, we see that the relative performance of the algorithms is very similar across these two architectures. The absolute throughput is considerably better on the Opteron, however, even when taking into account the higher number of cores (for instance, with 36 threads, the throughput of the Howley tree is roughly 5 times higher on the Opteron than on the Tilera in the 10% update case and roughly 4 times higher with 100% updates). This is explained by the overall better technical specifications of the Opteron.

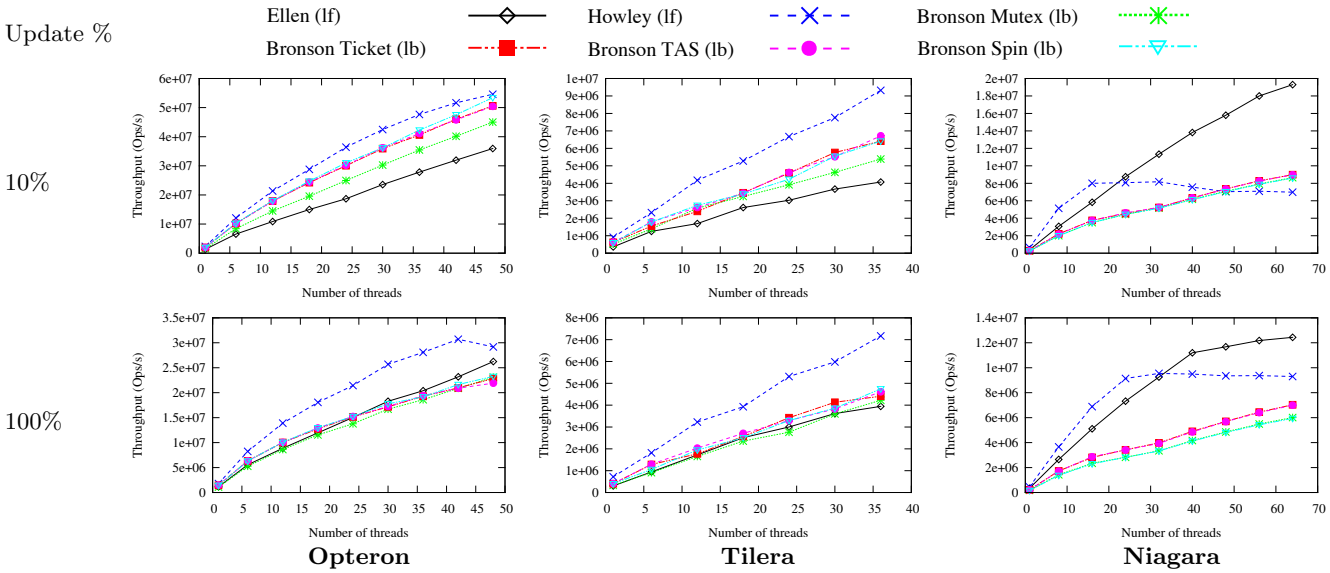


Fig. 1: Architecture comparison. The x axis shows the number of threads and the y axis shows the throughput in operations per second. For this comparison, the key range was fixed at 200K.

On the Niagara, we can observe a significantly different behaviour of the lock-free data structures, especially the Howley tree. The latter scales quasi-linearly up to a number of threads, then reaches a plateau. The poor performance of the Howley tree on this architecture is due to its more complex find function (more instructions per call) being run on a "simpler" processor. More precisely, in the Niagara, each core is less complex than those of current high-end processors, allowing 8 cores to fit on the same die [9]. For instance, each core has only 2 integer ALUs (each one shared among 4 hardware threads), thus creating a bottleneck for the several `if` statements in each call to `find`. To confirm this explanation, we simplified the Howley tree find function by removing the helping mechanisms and extra checks. This simplified version was compared to the original Howley tree and the Ellen tree in the 0% update case. Figure 2 confirms that the extra complexity of the find function is the source of the bottleneck for the Niagara architecture.

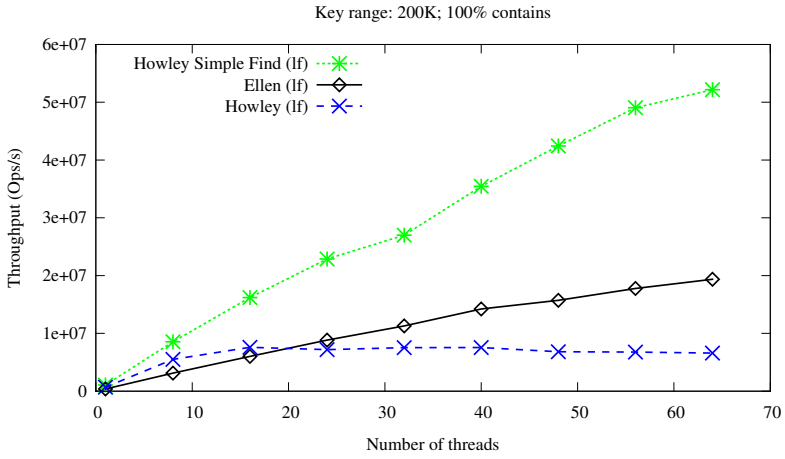


Fig. 2: Comparison of Howley tree with a simplified find function with the regular Howley tree and the Ellen tree on the Niagara.

In the second part of our discussion, let us fix the architecture to the Opteron machine and have a more in-depth look at the three algorithms. Figure 3 shows the performance of the three BSTs, for different key ranges (varied along the rows) and different access patterns (varied along the columns). For this comparison we have used all key ranges and access patterns described in Section 4.1.

At a first glance, it is apparent that the Howley tree outperforms the other two trees in almost all situations (all but 2K-100% contains and 2K-100% updates). However, this comparison is worth further nuancing.

It can be observed that the Ellen tree (as well as the Howley tree) exhibit low contention on small lists (2K key range), due to their lock-free design, giving them an advantage over the Bronson tree. Nevertheless, this effect evens out for larger lists (such as the 2M key range), where there is less chance for threads to access the same memory location simultaneously.

Due to the low complexity of its search function, the Ellen tree performs better on smaller lists than on larger lists. This is because, even if all the information in the Ellen tree is stored at the leaf level, with small lists the height of the tree is manageable, so the simplicity of the search partly balances out the suboptimal number of nodes to be traversed. Furthermore, the external tree structure allows for a simpler remove function. This is reflected in the increase in scalability (relative to the other two trees), as the percentage of updates grows. However, for read-dominated access patterns, the Bronson and Howley trees outperform the Ellen tree in every case. This is because neither of them needs to traverse the full height of the tree with each operation. The effect is aggravated by read-intensive work-loads on large lists, because the height of the data structure is larger (illustrated in the 2M-100% contains case).

Even if the Bronson tree and the Howley tree belong to different design paradigms (one is lock-based and the other one is lock-free), their internal tree structure creates similarities in their behaviours: their scalability profile is consistent for each access pattern, across all key ranges except the smallest one. The Bronson tree has a poor performance on the smallest list size because of high contention. However, as the list size increases, so does its throughput. Also, the Bronson tree behaves better in read-intensive workloads than in write-intensive workloads, due to its optimistic concurrency control mechanism, which makes readers invisible. Consequently, in the 2M list case, the Bronson tree is as good as the Howley tree for the non-update-dominated scenarios. There is a tendency that

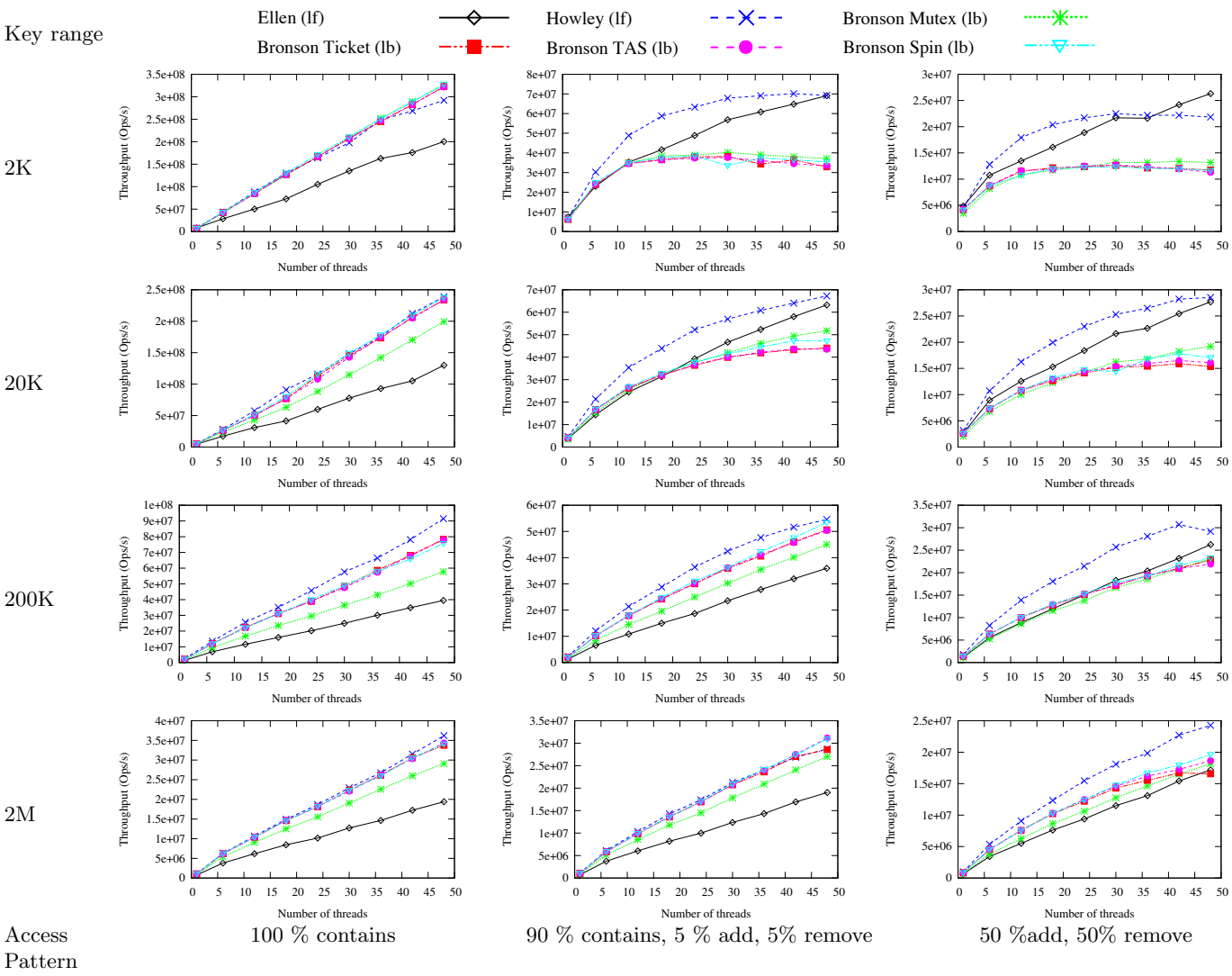
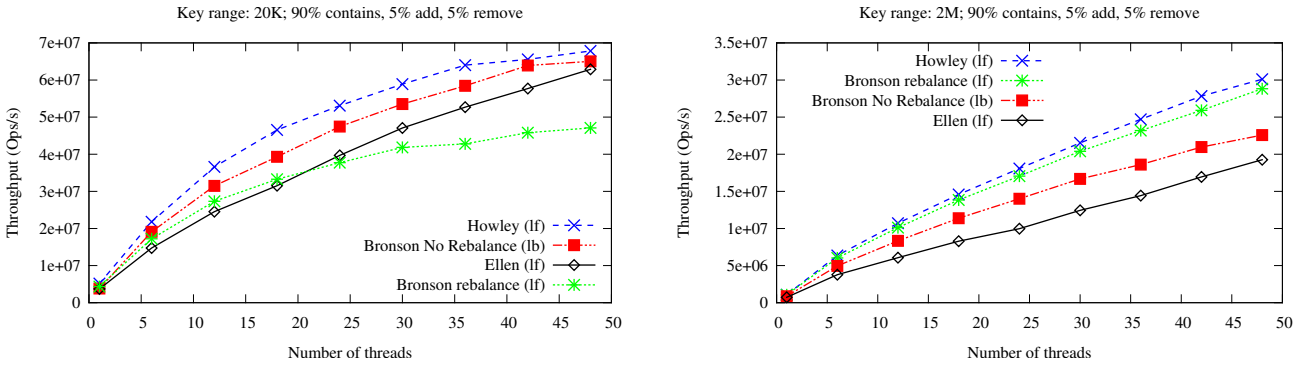


Fig. 3: Algorithm comparison on the Opteron machine. The x axis shows the number of threads and the y axis shows the throughput in operations per second.

suggests that for even larger lists the Bronson tree would outperform Howley, due to its rebalancing mechanism. A more conclusive test with larger lists was unfortunately not possible, because of memory constraints. The hypothesis that the Bronson tree would eventually overcome the Howley tree is however supported by Figure 4, which illustrates the growing importance of rebalancing as the key range increases. Even if for small lists nothing is gained from rebalancing, for larger lists, the original Bronson tree (with rebalancing) outperforms its non-rebalancing variant. We believe that such a mechanism would also benefit the Howley tree, as list sizes increase.

Another dimension which is interesting to explore when analysing the Bronson tree is the effect of different types of locks. As described in Section 4.1, four lock types were evaluated. Figures 1 and 3 show that there is no clear

winner: every lock is sometimes better than the others. In addition, the throughput gap is often negligible between the best two locks.



(a) Key Range: 20K

(b) Key Range: 2M

Fig. 4: Comparison of the effect of rebalancing in the Bronson tree, w.r.t. the list size

5 Conclusion

The over-arching goal of this semester project has been to reach a better understanding of the nature of concurrency. Another, more down-to-earth goal was to identify what design strategy is best suited for a variety of contexts in parallel programming. To this end, we have implemented two binary search trees based on different designs — a lock-free internal BST and a lock-based relaxed-balance AVL tree — and compared them to a third lock-free external BST. The comparison was carried out with various workload classes and tree sizes, on three multi-core architecture models. The results showed that there is no clear winner but identified an ideal use case for each of the three structures.

6 Acknowledgements

We would like to give special thanks to our two supervisors, Tudor David and Vasileios Trigonakis, who guided us throughout this semester. Their advice was always very helpful, constructive and prompt, allowing us to successfully carry out our project while understanding as much as possible on our own.

References

1. Bronson, N.G., Casper, J., Chafi, H., Olukotun, K.: A practical concurrent binary search tree. In: ACM Sigplan Notices. vol. 45, pp. 257–268. ACM (2010)
2. David, T., Guerraoui, R., Trigonakis, V.: Everything you always wanted to know about synchronization but were afraid to ask. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. pp. 33–48. ACM (2013)
3. Ellen, F., Fatourou, P., Ruppert, E., van Breugel, F.: Non-blocking binary search trees. In: Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing. pp. 131–140. ACM (2010)
4. Fraser, K.: Practical lock-freedom. Ph.D. thesis, PhD thesis, Cambridge University Computer Laboratory, 2003. Also available as Technical Report UCAM-CL-TR-579 (2004)
5. Harris, T.L.: A pragmatic implementation of non-blocking linked-lists. In: Proceedings of the 15th International Conference on Distributed Computing. pp. 300–314. Springer-Verlag (2001)
6. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming, Revised Reprint. Elsevier (2012)
7. Howley, S.V., Jones, J.: A non-blocking internal binary search tree. In: Proceedinbgs of the 24th ACM symposium on Parallelism in algorithms and architectures. pp. 161–171. ACM (2012)
8. Wicht, B.: Binary trees implementations comparison for multicore programming. Tech. rep., HES-SO (2012)
9. Zhang, J.: Chip multi-threading and sun's niagara-series (2009), `http://www.cs.wm.edu/~kemper/cs654/slides/niagara.pdf`