**Trie:**

The trie benchmark comes to show the benefit gained by using trie for different node sizes. There's one trie per node, and the trie size has the same size as the node size. Our first observation was that trie should be used when the application has a low modification ( update/remove) percentage. The scenario we explored first was the following : 10 seconds run, using 8 threads, maximum of 10 levels, key range between 0 to 100000, range query size of 1000 keys. Operation distribution was 60% lookup, 30% range query, 10% modification (update/remove distributed equally ). The different trie sizes tested were : 32,64,128,256,512,1024,2048. Before starting measuring results for the 10 seconds run, the data structure was initialized with random values. This was done in order to make sure that the results won't be affected by performance of the data structure when it's small ( causing a lot of splits).

We got the following results:

|  |  |
| --- | --- |
|  |  |

We noticed that when modification is 10% of the operations, trie is not useful and not using a trie would result in a better throughput. The reason is the creation time of a new trie. A new trie is created every time a new node is created, which is every time there's a modification. When a split occurs two nodes are created, resulting in two trie creations for a split. This will explain the big throughput difference for the java experiment, when using nodes that contain a small amount of elements. When node size is small, split happens more frequently and more nodes are created. Therefore, a lot of tries are created which result in a lot of time spent creating the tries. The main benefit of using tries is O(log n) search time for a key. When the node size is small, this benefit is neglectable since traversing a small array takes a short period of time anyway. When looking at a large node size ( 2048 elements ) we see that the results between using a trie a not using one get closer. This happens from the same reasons there's a big difference for small node sizes. When a node is big it's less likely to be split, therefore less splits occur in a run ( for the same amount of time ) . The benefit of a quick search time in tries is much more relevant in this case. Since the array in each node is large, the O(log n) search time is much quicker the traversing the whole array.

Given the result above, we wanted to see where does the trie actually does help. We kept the same experiment parameters, but reduced the modification percentage. For the c implementation the Operation distribution was 60% lookup, 39% range query, 1% modification (update/remove distributed equally ). For the Java implementation the Operation distribution was 60% lookup, 40% range query, 0% modification ( not using trie still beat the throughput of using trie at 1% modification).

|  |  |
| --- | --- |
|  |  |

Looking at the c Implementation we see that using a trie has a better throughput than not using a trie for a trie size of 128 and up. When a modification rarely happens we can focus on the search time and its efficiency using a trie. For larger node sizes, using the trie improves throughput significantly . Almost all operations require a key lookup in the node's array. Using a trie in these scenarios is much more beneficial, thanks to the quick search time using a trie. We see that throughput decreases for bigger node sizes than 128, when trie is not used. It happens due to a long search time for a key in a large array. On the other hand, when using a trie, we see that the throughput increases when the node size gets bigger ( except for 2048 node size, where the trie creation time is so long it affects the total performance ). Bigger nodes mean less splits and less tries to create, meaning less time spent on creating new tries.

Looking at the Java Implementation we see that using a trie has a better throughput than not using a trie for a trie size of 512 and up. Quick search as opposed to a whole array traversing wins for large node sizes in the java implementation as well. The java implementation using trie started being better with node size of 512 elemetns and modification percentage of 0 ( as opposed to node size of 128 and modification percentage of 1 for the c implementation ). We assume that the reason is a long trie creation time and the java memory model.

**Java STM**

This benchmark shows the difference between the performance of the given algorithm for Java STM and the our improved solution. The improved solution was making the transactions more fine grained. The remove/update operations got an array of key-value pairs to make the operation on. The improved solution had the transaction scope set per operation on a key value pair, and not over all the key value pairs given as parameters to the function. This way if a transaction fails on the 4th key value pair of the operation, the last 3 don't have to reapplied. When less successful operations are reapplied, throughput increases due to not applying redundant operations. We tested a few scenarios to see how does the change impacts in various situations. We test the scenarios using our default running parameters for different number of threads ( 1, 2, 4, 8 ). The scenarios are :

1. **Simple Run** - every update/remove operation gets an array of 4 key value pairs for 4 different lists. The lists are the same and in the same order for each operation. We excepted to see here many collisions causing many transactions aborts since all threads run on the same lists and in the same order.
2. **1/2 Backwards Run** - every update/remove operation gets an array of 4 key value pairs for 4 different lists. The lists are the same, half the operations are done in one order and half are done in a backwards order. Meaning, half the runs had the order "List1,List2,List3,List4", and half of them had the order "List4,List3,List2,List1". This way we excepted to have less collisions in the lists and thus less transaction aborts.
3. **Random Run** - every update/remove operation gets an array of 4 key value pairs for 4 different lists. The lists are randomly selected from a pool of 7 different lists, and in are ordered randomly . This is a more realistic approach where different lists are accessed by different application. Because of the randomization of the operations we excepted to see less collisions than the other 2 scenarios.

We got the following results ( Throughput states the number of successful operations per second):

First we notice that running on 1 thread has no difference in throughput between the runs, which is obvious since transaction wouldn't fail when only one thread accesses all the lists. We see that for 2 or 4 threads the throughput of "STM Improved" is much better than the throughput of "STM". The throughput is better since no operations are reapplied when not needed. We do see that for a large number of threads ( 8 ), the throughput of "STM Improved" is still better, but it's pretty close to the throughput of "STM". We suspect that it happened because the following scenario for "Improved STM" happened quite a lot : Say a few threads start an operation at the same time, all of them must apply the operation on list1,list2,list3 and list4 in that order. Therefore the threads will have their transaction fail on every list, since they all try to apply the operation on list1 and then list2 and so on. All of the threads will try to reapply the operation on the same list over and over again. On the other hand, for "STM", accesses to lists will be distributed to different lists after some time. Since transaction failure means starting over all of the operation in the function, some threads may "lag" behind because they keep getting transaction abort. Say a few threads are all trying to modify list3, upon failure a thread will go back and try to apply the operation on list1 again ( and not on list3). This may cause a scenario where some threads are accessing list4, some access list1 and the parallelism on same list access reduces, since threads access different lists at the same time.

Here we see that the throughput is generally better then the simple run. Since not all threads run the operations on the same order, less collisions occur. The optimal result is received for 2 threads for "STM Improved". In this case we except not to have a lot of collisions because a lot of the parallel runs will access different lists (e.g one start applying to operation on list1,list2,list3,list4 and the other on list4,list3,list2,list1). With more threads, throughput decreases Large contention on lists can be received from both ends. We see who throughput drops for "STM" when running with 8 threads. we suspect that the following scenario might have happened a lot: Thread 'A' applies the 4th change on list4, Thread 'B' applies the 4th change on list4 and Thread 'C' applies the 1st change on list4. In this case on "STM" both threads 'A','B' have to reapply 3 changes each, whereas in the simple run they might not get to this situation since the contention happened earlier on list1 and list2. Causing less redundant reapplications.

Here we see that "STM Improved" is much better for 2 threads and up, especially for 8 threads. Since the lists that the operation runs on are chosen randomly, less contention occurs while accessing each lists. STM's throughput decreases as number of threads increases. Since we have a pool of 7 lists and 4 lists are given as parameters to a modify function, there's still a good chance of contention per list and thus the transaction will abort. Since lists are randomly chosen and order is random as well, the collision have the same chance of occurring when applying the operation on the 4th list as in applying the operation on the 1st list. As stated before for "STM", collision on the 4th list means redundant reapplication of operations on the last 3 lists. In the "Simple Run" we excepted more collisions on the 1st or 2nd list, since all the threads have the same list order, and were dispatched at the same time. So here, there were a lot more reapplications of operations for lists then for the "Simple Run". It caused less and less actual needed operations to be applied, therefore low throughput.

Also, throughput for 2 threads is still the best for both implementations. Showing that the usage of a large number of threads won't help to get a better throughput, when using STM.