Workshop - Leaplist

Shay Peled, Tel Aviv University (shaypele@gmail.com)

David Meriin, Tel Aviv University (meriind@yahoo.com)

Yossef Yakobi, Tel Aviv University (yossi_ya@yahoo.com)

Table of Contents

Introduction

This code was developed as part of "Workshop on Multicore Algorithms" at Tel-Aviv University, under the guidance of Prof. Nir Shavit and Moshe Sulamy.

Our source code can be found here - Source Code

In our work we relied upon the data structure conceived by Nir Shavit, Hillel Avni and Adi Suissa. This data structure, "Leaplist", was based upon the known "Skiplist" data structure where the main difference being that each node in the list can hold several key-value pairs instead of just one. We build several implementations of the structure each with a different approach to allow multi-threaded use. Each implementation was written in Java and in C. We ran several benchmarks to test the advantages of each module under different conditions.

Leaplist Structure

We now offer a short overview of the Leaplist data structure. The data structure itself actually holds several individual lists. Note that the updating operations run on several lists and the query operations operate on an individual list. For further reading refer to http://mcg.cs.tau.ac.il/papers/transact2013-leaplist.pdf and the code itself.

Operations

The data structure supports the following operations:

LookUp (l, k) – Receives a list and key. If the key exists in the given list it will return the value attached to that key. Otherwise it will return null.

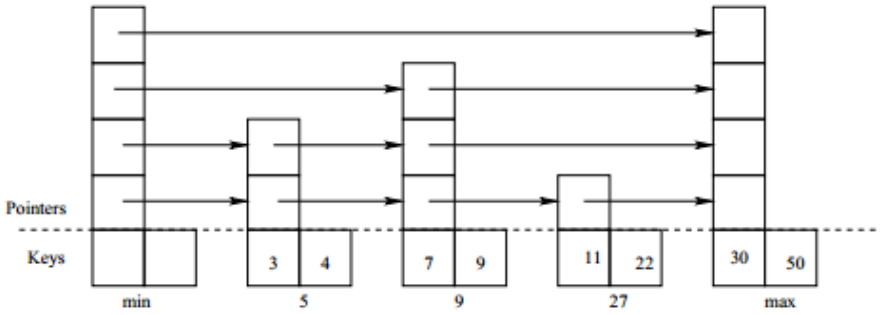
Range-Quarry (l, k_from, k_to) – Receives a list, a low key and a high key and returns an array of values matching the keys in the range.

Update (ll, k, v, s) – Receives arrays of Leaplists (ll), keys(k), values(v) and the size (s) of the arrays. For each index it updates the key and value for those lists. For example for index=1 it updates key k[1] with the value v[1] in list ll[1]. If the key doesn't exists the key value pair is added to the list.

Remove (ll, k, s) – Receives arrays of Leaplists (ll) and keys (k) and the size (s) of the arrays. As in the update operation it removes the key-value pair for k[i] in list ll[i].

We will further elaborate on the implementation of these functions. We will also describe some of the more crucial helping methods for better understanding of this data structure.

General Structure



A database of lists holds several lists. Upon initialization each list will contain two nodes. One sentinel node whose range is bounded from above by -∞, and a second node with no keys and a high value of ∞. The level of the sentinel node will be the maximum level (predefined) and all its pointers will point to the second node. Two consecutive nodes define the range encompassed by the second node. The second node's range will be from the first node's high value up to its own high value. When looking up a certain key the search predecessors function will be used (explained later on) to find the specific node whose range match's the key. Then the Trie is checked to see if the key exists and if so at what index of the key-value pair. The key-value array (and the whole node for that matter) is immutable and never changes after its creation. So whenever an update is performed a new node is created and replaces the old one. Same is done when removing a key-value pair. If the node is full (i.e. the number of pairs reaches a predefined number) it will split into two consecutive nodes. If consecutive nodes are sparse (the number in both nodes is less than a predefined number) they will be merged into a single node.

Node Structure

Leap-List Node data-structure

```
1 define Node: struct {
2     live: boolean,
3     high: unsigned long,
4     count: unsigned long,
5     level: byte,
6     next: array of *node,
7     values: {key-value} sorted_pairs,
8     trie: {key-index in node} trie
9 };
```

live – True when the node is actively part of the list, false otherwise.

high – Denotes the highest key in the node.

count – The number of key-value pairs in the node. The Maximum is predefined.

level – Denotes the maximum size of the pointers array next.

next – Array of pointers to the next nodes in the list.

values – An array holding the key value pairs.

trie – A trie data structure holding all the keys and is used to determine the index of a certain key in the key-value array.

Modules

We designed 8 different implementations of the data structure, 4 in "Java" and 4 in "C". Each presents a different approach to handling multi-threaded use of the Leaplist data structure.

We designed 4 implementations in java. Global lock, Grained lock, STM and Improved STM. The STM is a close translation from the implementation we got in C. This used transactional memory and 'Improved STM' module is an improvement we performed on that implementation.

Global lock

In this module one lock was used to keep the threads from accessing the data structure in the same time. A lock had to be taken by a thread at the beginning of each operation and released at the end of the operation. This is done regardless of the nodes it was approaching or the type of action it was taking. This way in fact only one thread can be working on each data structure at the same time. This implementation is slow compared to other implementations and utilizes very little of what multi-threaded work has to offer.

This implementation was the easiest to implement. This implementation is fairly close to the serial solution. One lock blocked all other threads, and only 2 lines of code were added to each main operation (lock at the beginning and release in the end).

Grained lock

In this module a lock was assigned to each node. This fine grained implementation Is based on the "lazy skip list" algorithm. This way only the nodes that are being changed and their predecessors are actually locked and so several threads can work on the same data structure as long as they are working on different areas of it.

This implementation was the hardest to implement, since a big amount of locks had to be managed and thread logic had a lot more pitfalls than the other implementations.

Transactional memory

In this module we used a set of atomic blocks instead of locks. For the update and remove operation an additional part was added to the function's logic. This is part is in charge of checking if any of the nodes being handled are currently being changed by another thread (using the live mark of each node). If one of the nodes is not marked live the function will fail and the process will restart all over again. Additionally, the software transaction (Deuce STM for java, gcc's software transaction for c) also made sure that there are no 2 threads that try and change the same cache block, if so the transaction will fail and the operation will start over.

For the C implementation we were asked to use hardware transactions. Since the transaction scope was too large to be handled with hardware transactions. Therefore we

used software transactions in c, and hardware transactions were used only for the improved solution.

Transactional memory improvement

As stated before update/remove operations get an array of lists to do the operation on. In the given solution that was handed to us, in case there's a transaction failure caused when trying to do the update/remove operation on any key-value pair, the entire process will begin all over again. For example if we try to update 7 key-value pairs, and the transaction failed on the 4^{th} update, the algorithm will revert all the changes made and try to update all the 7 key-value pairs again. Meaning, successful updates are updated once again for no reason.
Our improvement is that each key-value pair is handled in a different transaction, and once a key-value pair update was successful, it won't be issued again. This way no redundant updates are made.

Benchmarks

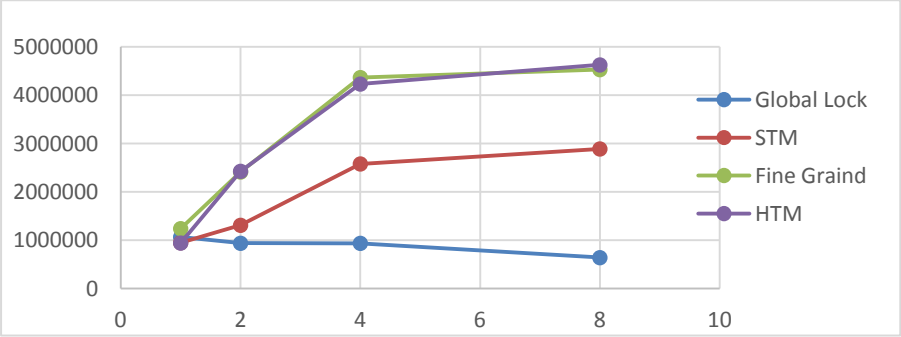
Throughout our benchmarks our general scenario was the following (unless states otherwise):

- 10 seconds run
- initialized Leap-List with 100,000 random values
- Node maximum size is 300
- maximum level of 10 per node.
- 3-5 runs per each experiment.
- Key range : 0 - 100000
- Range of 'Range Query ' operations - 1000.

Different mixtures of operations:

This benchmark comes to show the throughput of each implementation of the Leap-List in different mixtures of update, remove, lookup and range query in C and in Java. We show the average of the five results and for different number of threads 1, 2, 4 and 8. (based on the general scenario run parameters)

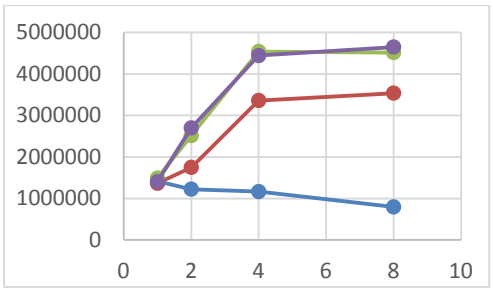
in C :
The result of 60% lookup, 30% range query, 9% update and 1% remove :



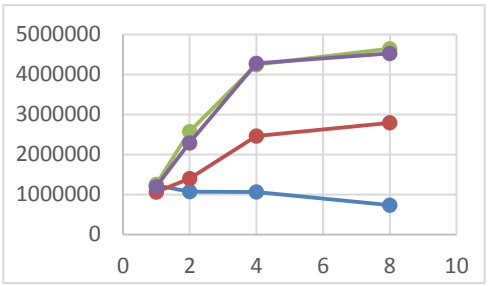
From the results we can see that the algorithm of HTM (Hardware Transactional Memory) and Fine grained both have the best throughput almost 500% percent better when we compare them to Global lock and about 150% better when compare them to STM algorithm. For different mixture for 90% search and 10% modified we get the same results and conclusions that the two algorithm of HTM and Fine grained have the best throughput and

dominate the other two algorithm.

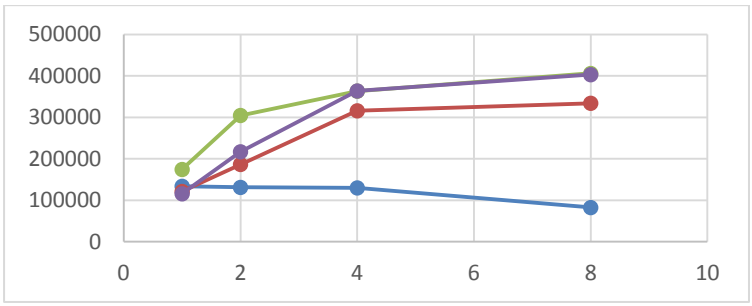
90% lookup, 10% modified :



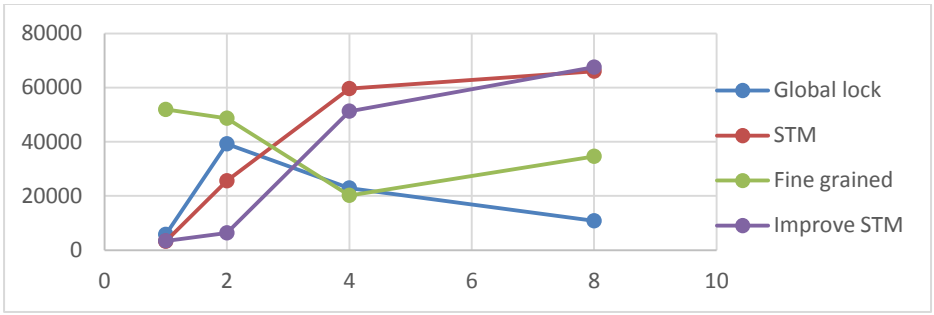
90% range query, 10% modified :



In 100% modified of 50% update and 50% remove we get the results:



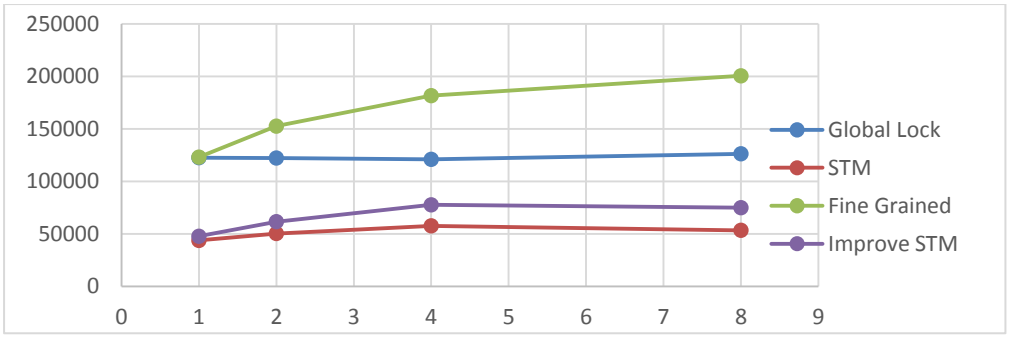
From those results we see that also in 100% modified the algorithms of fine grained and HTM dominates the algorithm of global lock, they have about 400% better throughput then the global lock. In 100% modified we see that fine grained and HTM still have better throughput then the STM but now the STM throughput is closer to their throughput.
The next graph show the standard deviation of the first mixture (the others look the same):



we can see that we have a bigger stdev in the transactional memory algorithms the from the lock algorithms and the reason for this is that in run that don't have many collisions the HTM and STM don't use lock and give excellent performance but in runs that we have many collisions the part of the transactional memory in the code can run many times until it succeed to run atomically and neither of the threads make a progress. Because of the two possibilities we could have bigger difference in the throughput of two runs.

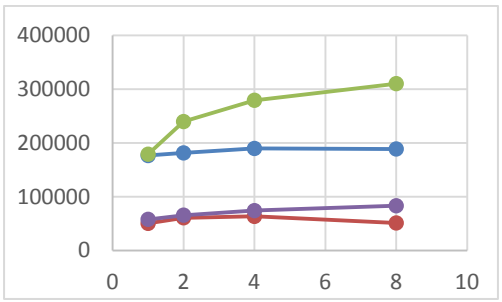
In Java :

The result of 60% lookup, 30% range query, 9% update and 1% remove :

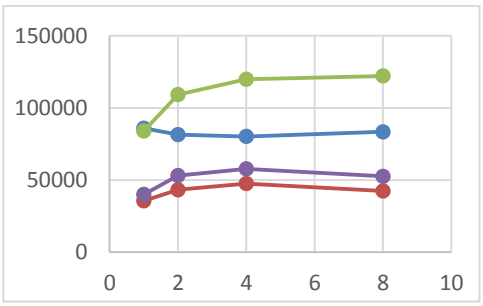


From the results we can see that in java the picture is entirely different from C.
the algorithms of transactional memory give us very poor throughput compare to fine grained and even compare to the simple global lock.
The reason for this is that the Deuce STM is not powerful enough and have very high parallel overhead and we can see that from the results for one thread only when there is no collisions that the performance of the Deuce STM algorithms is much worse than the global lock and fine grained lock.
Also when it have more than 1% modified the two algorithms suffers from many collisions that causes more cache misses and the performance is decreasing.
Fine grained still has the best throughput, about 150% better then global lock and about 300% better than the our improve version of the STM code we get and about 400% better than the original STM.
For different mixture for 90% search and 10% modified we get the same results and conclusions that Fine grained has the best throughput and the algorithms of software transactional memory have the worst throughput.

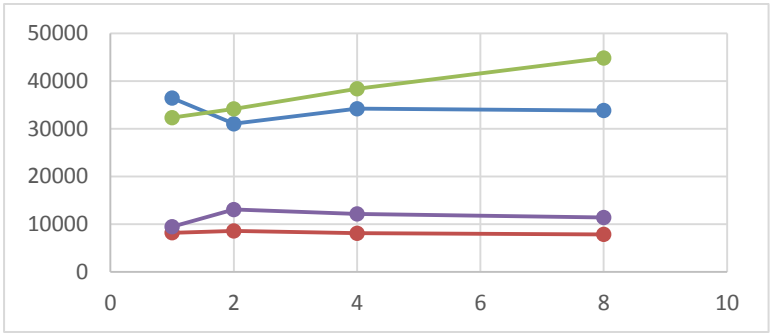
90% lookup, 10% modified :



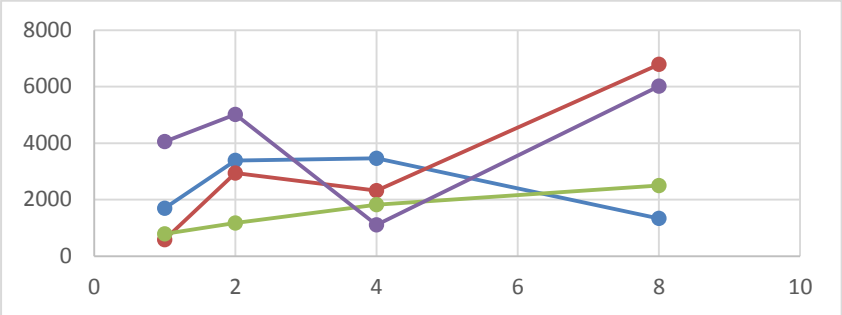
90% range query, 10% modified :



In 100% modified of 50% update and 50% remove we get the results:



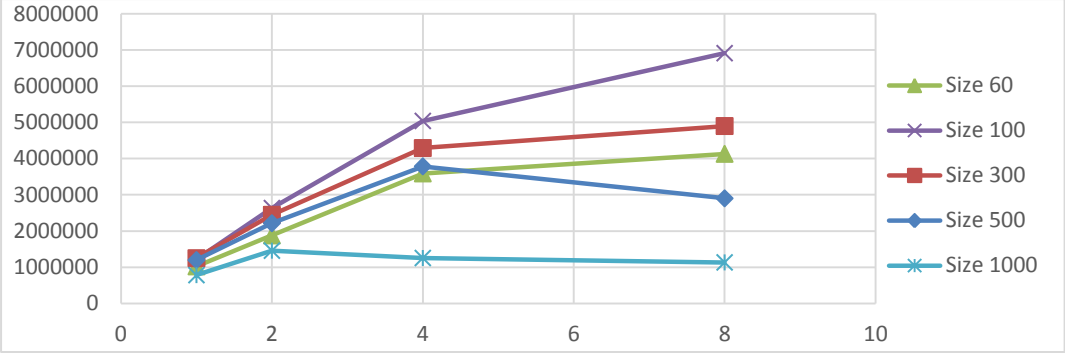
In 100% modified we still see that fine grained has the best throughput and it about 400% better than STM and our improve version of it.
We can see that still our version has better throughput then the original STM even in 100% modified.
The next graph show the standard deviation of the first mixture (the others look the same):



We can see that also in java even when the throughput of the STM and our improve STM is the lowest they have bigger standard deviation then the lock algorithms for the same reasons we mentioned above for C.
In both C and Java one can see that the search dominated workload has a much higher throughput than the modified-only workload. This is because a higher modifications rate incurs a high overhead of update and remove operations that cause much more conflicts and retries in transactional memory algorithms and contention on the locks in fine grained algorithms. In C the global lock not affected by the mixture because we lock the entire data structure in each operation. But in Java we can see difference between the executions with difference mixture of operations because of the complexity of the function update and remove that creates new nodes and their trie what takes big part of the execution time.
We can see also a difference in java between 90% lookup that we get much bigger throughput than 90% range query and that because of the traverse on all of the nodes we get from search predecessor and add the appropriate values to the set we return actions which takes more time.
We can see from both of the results for C and for Java that as we expected all the algorithms and in all mixture of the operation the throughput we get in C is much bigger than the throughput we get in Java.

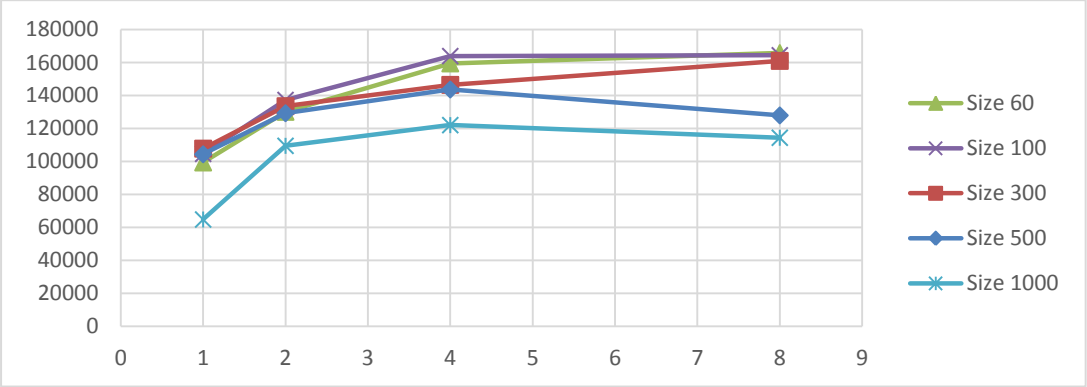
Node Size benchmark

This benchmark comes to show the effect of different node sizes on the throughput of Fine Grained lock algorithm of the Leap-List in C and in Java. All the experiments were held with 9% update, 1% remove, 45% lookup and 45% range query (based on the general scenario run parameters). We show the average of the five results and for different number of threads 1, 2, 4 and 8
In C:



We can see that we get the best throughput when the node size is 100. When the node is in size 60 we expect that there will be less conflict but there is much more split in updates and it take longer time to find the key because we need to go over more nodes and we have much more cache misses on the way, And we are getting an about 170% better throughput in node size 100 than 60 because of this reasons.
We can see from the results that the more we increase the node size to 300, 500 and 1000 we get less throughput node size 100 has an about 700%best throughput than size 1000, about 230% better than size 500 and about 140% better than size 300. The reason that it happens is because of fact that the bigger the node is it holds more keys and we have a much more conflict in the nodes between threads that causes contention on the nodes locks and damage the performance of the algorithm.
From the results we can also see that for smaller node sizes 60,100,300 we get more parallelism and the throughput increase when we have more threads. But in bigger node sizes because of the contention on the locks we get less throughput for bigger number of threads, in size 500 we get bigger throughput in 4 threads than with 8 threads and in node size of 1000 we get the best throughput with only 2 threads what means that the contention on the locks damage the performance more than the exploit of the parallelism of the data structure.

In Java:



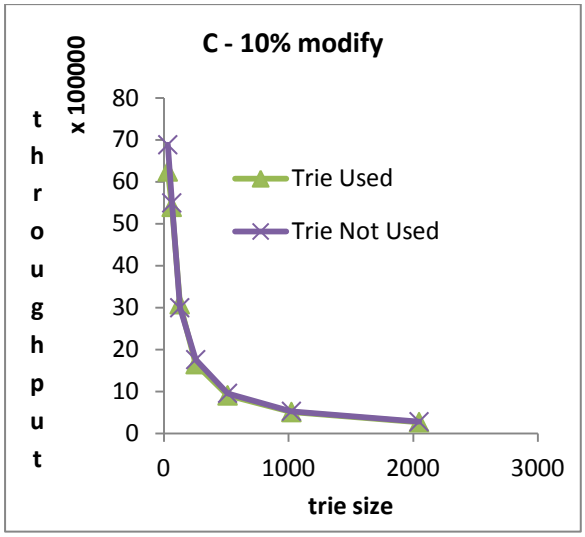
In Java we get different results from C here we can see that is better use smaller node sizes, for 8 threads we get the best results for sizes 60,100 and 300, for 4 threads we get best results for sizes 60 and 100. We can see difference in the throughput even in 1 and 2 threads

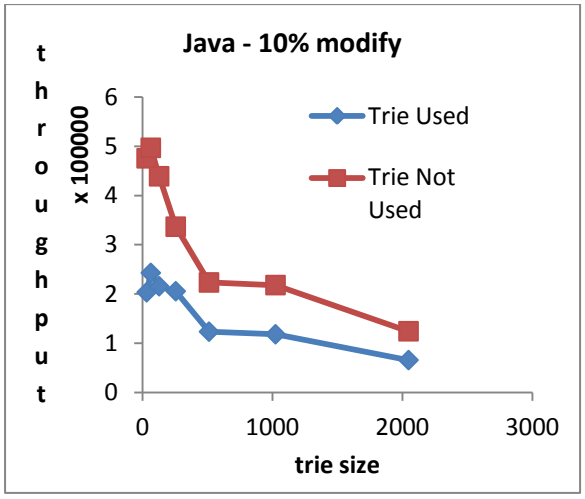
for size 1000 and all the other sizes.
The main reason for this is that in Java the time of the node creation and his trie creation is much bigger than in C and when the when the size of the node gets bigger it is more crucial and we get poor performance.
We can see the same behavior as in C for the sizes in relation to the number of threads, for smaller size the performance is improving when the number increase and we are getting the biggest throughput for 8 threads, and for bigger sizes as 500 an 1000 the performance is decreasing and we get better throughput in 4 threads than in 8.

Trie Size

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. Additionally to the general case, the following run parameters were used : 8 threads run, 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).

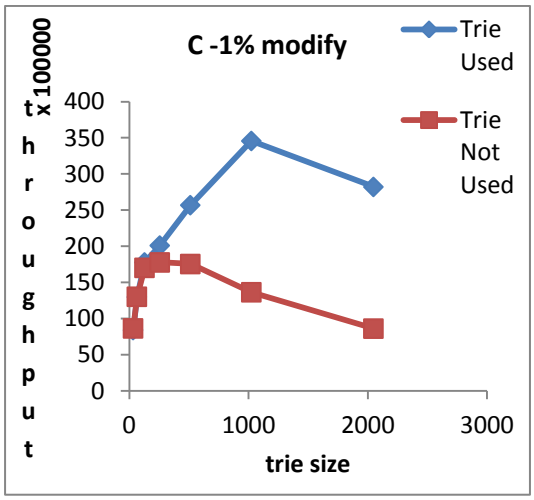
We got the following results (Throughput is in ops/sec units):

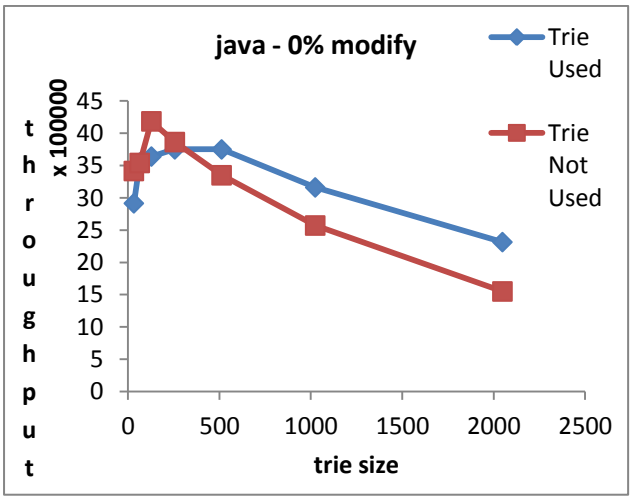




We noticed that when modification is 10% of the operations, trie is not useful and not using a trie would result in a better throughput. The reason is the creation time of a new trie. A new trie is created every time a new node is created, which is every time there's a modification. When a split occurs 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.

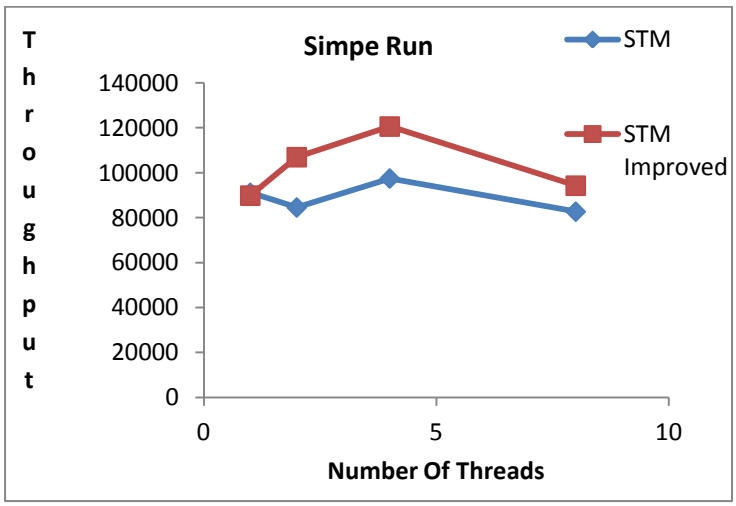
Looking at the Java Implementation we see that using a trie has a better throughput than not using a trie for a trie size of 512 and up. Quick search as opposed to a whole array traversing wins for large node sizes in the java implementation as well. The java implementation using trie started being better 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 :

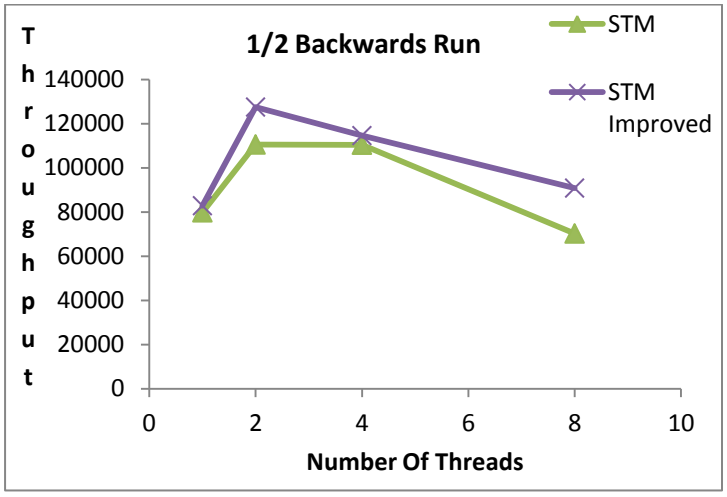
1. **Simple Run** - every update/remove operation gets an array of 4 key value pairs for 4 different lists. The lists are the same and in the same order for each operation. We excepted to see here many collisions causing many transactions aborts since all threads run on the same lists and in the same 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):

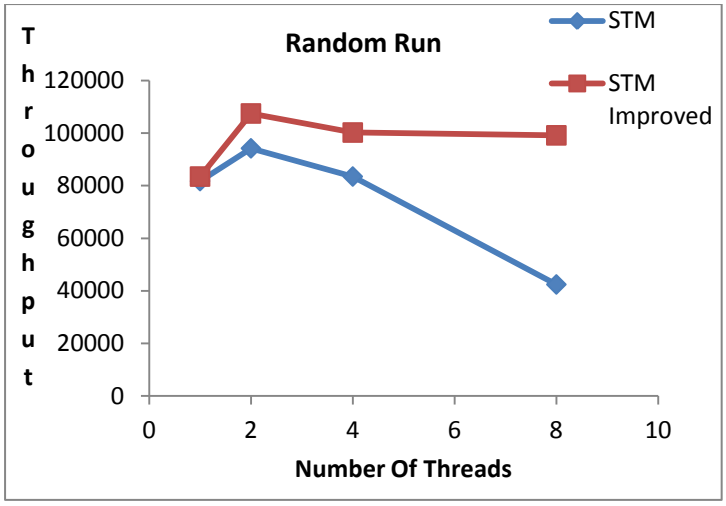


First we notice that running on 1 thread has no difference in throughput between the runs, which is obvious since transaction wouldn't fail when only one thread accesses all the lists. We see that for 2 or 4 threads the throughput of "STM Improved" is much better than the

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 that 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.

Key-Range Benchmark

In this benchmark we try to examine the different behavior of the different modules when the max-Key input is changed. The key range (the range which all the keys fall within) is set to be between 0 and the user defined max-key value. As was explained earlier each node of the Leaplist can hold a predefined number of key-value pairs. All the keys in each particular node fall in the range between its predecessor's high value and its own high value. If for instance the limit for a node's size is 100 and there are 50 keys then all the keys will be inside that one node. For this reason the maximum size of the nodes have a great impact on the performance of the data structure under the varying max-key input.

There are two main factors that vary greatly when changing the max-key. The first is obviously the collision factor. When the max key range is close to the node size it means that it doesn't matter how many key-value pairs will be inserted updated or looked for during the 10 second run time, the number of nodes will remain low. For instance if the node size is 10 and the max-key is 99 than no matters what operations the programs performs there will only be maximum of 10 nodes per list. When several threads are operating in parallel this will entail a great deal of collisions between the threads. Therefore, it is expected that when increasing the key range the throughput will increase, since the likelihood of a collision is greatly reduced.

Here we need to take into consideration the second factor affected by changing the max-key value. This will be the number of splits and merges, and the number of action each 'searchPredecessors' call makes. When the key range is big and there are many nodes the 'searchPredecessors' function can traverse through until it finally reaches the node holding the desired key. Each time a key-value pair is updated (inserted, updated or delated) its containing node will be replaced in the list with a new node. If the key range is much greater than the node size this will mean many more split and merge operations will be done. The searchPredecessor function is called in all operations so we would expect its affect to be very noticeable.

Test Conditions

Node size (java): 10 , Node size (C): 100

Range-Quarry size: 100

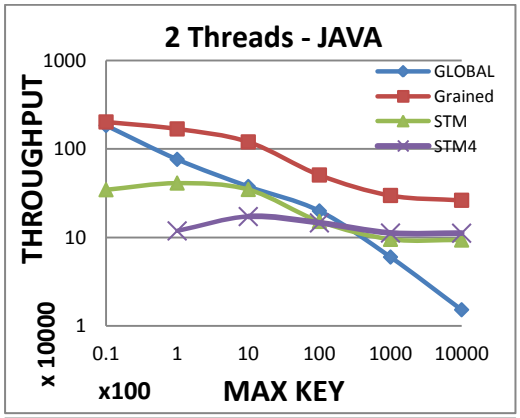
Number of lists: 4

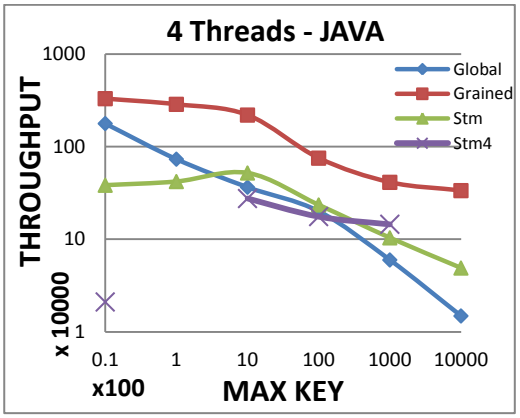
Seconds: 10

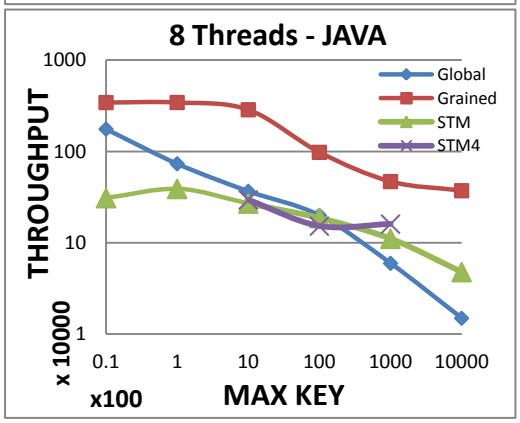
Operation proportion: 60% lookup, 30% Range query, 9% update, 1% delete.

Number of threads: The tests were performed for 2, 4 and 8 threads.

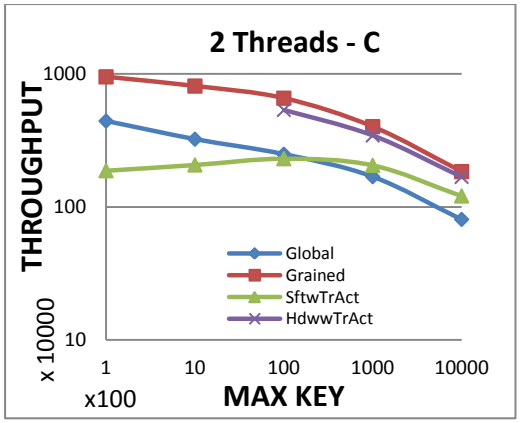
Java Results:

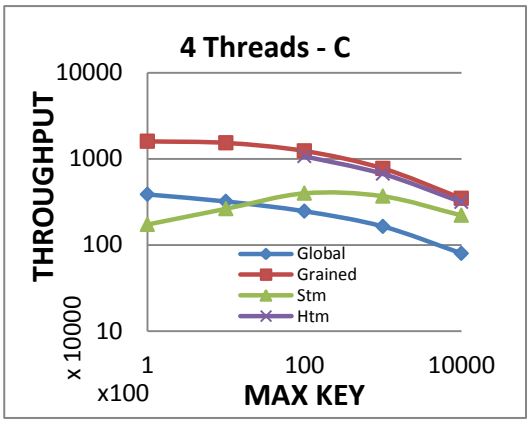


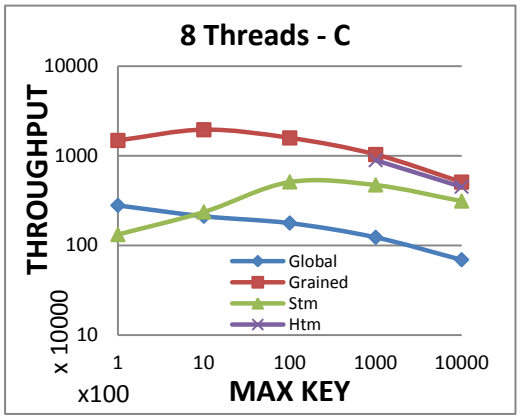




C results:







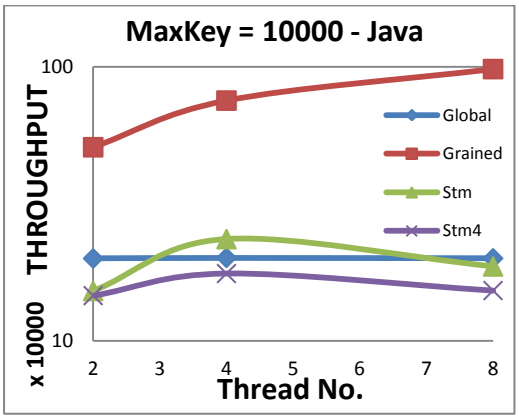
From all the results, both in C and in Java, we can notice that as the max-key value increases a shifting balance between the effect of the decreasing collisions and the increasing weight of each searchPredecessor action is noticed. When the max-key value is low a collision is almost guaranteed. In this case there isn't much difference between the global and grained modules and this is clearly noticeable in Java. As we increased the max-key value we can see that the global module's throughput is linearly declining. This is expected since the scale is logarithmic and the searchPredecessors runtime is log(n). For that modules the effects of fewer collision is null since the each action requires a lock. We can see that the Grained and the transactional modules also decline in performance (for the most part) but on a much lower rate. This is due to the advantage of the fine grained locking that counteracts the negative affect of the longer search time.

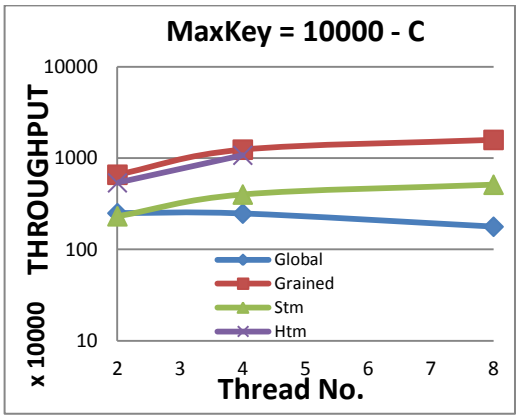
If we look at the graphs we can see that the transactional memory module's throughput decreases at a lower rate than the fine grained locking module. At some cases the throughput actually increases for some of the test. From this we can reach the conclusion that these modules are more granular than the fine grained lock, yet they are still less affective due to their high overhead. This is more evident in the C modules where the transactional memory modules show a clear increase in their throughput.

If we compare the C and Java modules we can easily say that the C modules are more affective and that the transactional modules in the C language are more much more granular than the fine grained locking modules. If their overhead wasn't higher they would be more effective.

One other issue that came up during the benchmarks was the fact that the hardware transactional module in c and 'Improved STM' module in Java were unstable at a lower max-key value. This goes to show that when many collisions are expected they are not recommended for stable use.

Thread no. comparison for max-key = 10000 and node size of 100.





The last two graphs show the throughput results for 2, 4 and 8 threads when the max-key value is 10000. For both C and Java (although more obvious in Java) we can see that increasing the number of threads by 2 and 4 doesn't increase the throughput by the same ratio. This is due to that fact that when two threads "collide" one of them will be ineffective. As expected the global lock module show almost the same throughput for all thread number. This is because only one thread actually works at any given time.

The Grained Lock and transactional modules show a steady improvement yet not at the same ratio as the number of threads. This is because of collisions that occur between the threads and due to the parallel overhead.

From the parallel overhead benchmark we noticed that the transactional memory modules had a very high overhead in the Java modules and a low overhead in the C modules. Sure enough we can see that in the Java results, as we increase the number of threads to 4 the throughput does increase a little but as we raise it to 8 threads it falls back down. In the C modules the throughput for both the software and the hardware modules continue to rise as we increase the number of threads. We believe that is due to the high overhead for these modules in Java compared to C.

Parallel Overhead Benchmark

In this benchmark we aim to check the overhead the different modules have compared to the "clean", non-parallel data structure. When a certain database is worked on by many threads there are certain actions that need to be done to ensure its proper use. This action include lock "taking" and "releasing", certain action retrying loops and atomic code segments that if unsuccessful could be repeated (in the transactional memory modules).

This means that each "parallel" module has a certain overhead that slows it down compared to a "clean" data base. If this overhead is too great it could mean that the advantages of several threads working on it in parallel will be very low to non-existent. This data could also help to better understand the results received in the other benchmarks.

To test the overhead we built a "clean" and tested its throughput. All the modules were tested under the same conditions with only one thread running. The resulting throughput of each module was divided by the through put of the clean module so the higher the score, the lower the overhead.

Test Conditions:

Node size (java): 300 , Node size (C): 300

Range-Quarry size: 1000

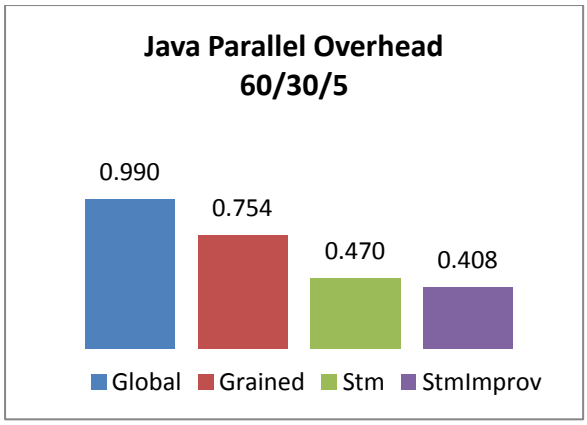
Number of lists: 4

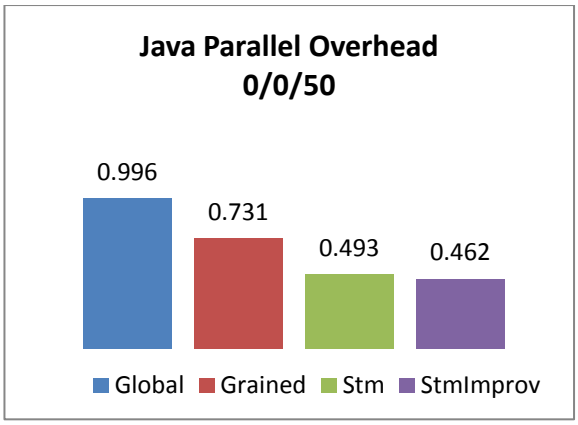
Seconds: 10

Max-Key: 10000

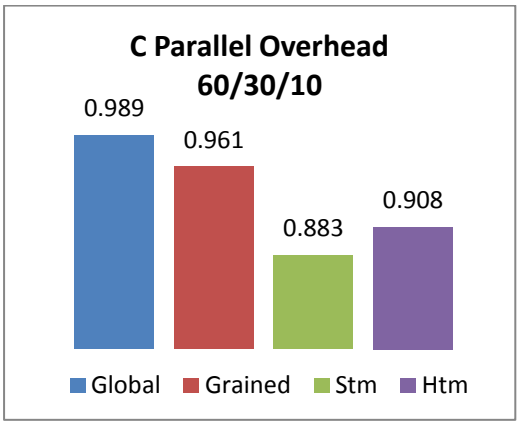
Number of threads: 1.

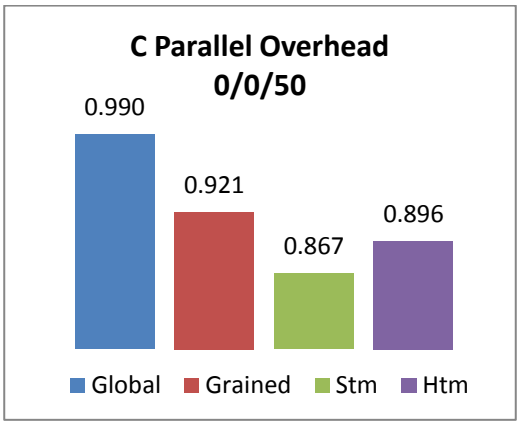
Java Results:





C results:





We can see that the results didn't hold many surprises. In both languages the Global Lock had the lowest overhead. This is because there aren't many locking actions. There is only one action at the beginning of each method and one (release) at the end. The Global Lock is followed by the Grained Lock module which has a few more locking actions per module. The overhead for both these modules is very low in "C" with an overhead of less than 10%. In Java the overhead for the Grained Lock is slightly higher.

The Transactional modules show a greater overhead although in "C" this isn't significant. In contrast the overhead for these modules in Java is much higher and exceeds 50%. This might greatly impact the performance of these modules and deem them unworthy for use in real implementations. We expect to see the impact of these results in the other benchmarks we performed. One of the reasons for the great overhead for the transactions implementations might be that when our code is instrumented using the library, a lot of extra code is added to handle the actual transactions. This code has a lot of logic and clearly impacts the run time of the program.

Conclusion

After researching the data structure with different parameters , we recommend to use the following setting:

- The fine grained implementation provides the best throughput.
- Use trie only in cases where the data structure is used for lookup or range query only (or a very small percentage of modification) .
- Maximum node size of 100 elements is recommended.
- Keep in mind that a big key range will affect performance.