**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. Each algorithm run on an initialized Leap-List with 100,000 random values, the node size was 300, max level of 10, each node uses trie to find the index of the key in the data. Each experiment execution is set to 10 seconds, and is repeated five times. We show the average of the five results and for different number of threads 1, 2, 4 and 8. The key range was 0 to 100000, range query size of 1000 keys.   
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 DeuceSTM is not powerful enough and when it have more than 1% modified the algorithm suffers from many collisions that causes many of cache misses that cause the performance to be very poor.  
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 was with 9% update, 1% remove, 45% lookup and 45% range query. All the experiments was on an initialized Leap-List with 100,000 random values, max level of 10, each node uses trie to find the index of the key in the data. Each experiment execution is set to 10 seconds, and repeated five times. We show the average of the five results and for different number of threads 1, 2, 4 and 8. The key range was 0 to 100000, range query size of 1000 keys.   
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.