ECE454 - HW4

Sina Zargaran - 998363621
Allan Johns - 998256603

Q1 - Why is it important to #ifdef out methods and data structures that aren't used for different versions of randtrack?

Using #ifdef allows us to keep one file while having multiple versions of the implementation. And it's important not to mix the implementations since some clash with each other. Also, we would need to have different .h files in the case of list level lock if we don't use #ifdef

Q2 - How difficult was using TM compared to implementing global lock above?

Using TM was simpler compared to single global lock since we only had to add 1 line to make synchronization work. But in global lock, we had to define and initialize a mutex variable, and add two lines to lock and unlock before and after the critical region.

Q3 - Can you implement this without modifying the hash class, or without knowing its internal implementation?

I believe it is not possible to implement list level locking without knowing the internal implementation of the hash class.

However, I implemented the list level lock by not modifying the hash class, but I had to bring the "HASH_INDEX" method in order to use the same formula as the hash table to reference the lock array.

Q4 - Can you properly implement this solely by modifying the hash class methods lookup and insert? Explain.

No. Modifying lookup and insert methods only will not allow us to properly implement list lock since the counter increment is outside of the scope of these methods and that instruction still needs to be atomic.

Q5 - Can you implement this by adding to the hash class a new function lookup_and_insert_if_absent? Explain.

Alternatively, this option could work considering the lock mechanism is added to the hash class as well. We can lock at the beginning of the function and unlock at the end, and basically entail the whole critical region within this function.

Q6 - Can you implement it by adding new methods to the hash class lock_list and unlock_list? Explain.

Yes, ultimately we can create a list of locks the same size as our hash table which is indexed and locked based on the hash indexing formula and unlocked at the end of the critical region. This way, each list in the hash array of the size (2^14) has a dedicated lock which we can utilize by having methods inside the hash class.

Q7 - How difficult was using TM compared to implementing list locking above?

TM in my opinion was a lot simpler since we only needed to add one instruction and wrap it around the critical region.

Q8 - What are the pros and cons of this approach?

Pros:
This approach is a lot faster and more efficient compared to the other methods since each thread does not need to be stalled waiting for other threads to lock and unlock. Also, there is no risk of two or more threads accessing shared data.

Cons:
This approach takes up more space since each thread has its own private copy of the hash table, and there might be some extra time wasted on merging the tables depending on the size of the tables.

Table 1: Results of averaging 5 runs of each with samples skipped set to 50

Implementation	1 thread (elapsed, s)	2 threads (elapsed, s)	4 threads (elapsed, s)
Original	17.72	N/A	N/A
Global Lock	19.44	14.17	23.06
Transactional Mem	21.11	20.75	13.17
List Lock	19.78	10.65	6.90
Element Lock	19.37	10.11	6.17
Reduction	17.70	8.97	4.50

Table 2: Results of averaging 5 runs of each with samples skipped set to 100

Implementation	1 thread (elapsed, s)	2 threads (elapsed, s)	4 threads (elapsed, s)
Original	35.05	N/A	N/A
Global Lock	36.70	22.34	19.35
Transactional Mem	38.40	29.50	16.31
List Lock	37.12	19.33	10.95
Element Lock	36.83	18.80	10.43
Reduction	35.10	17.64	8.83

Q9 For samples-to-skip set to 50, what is the overhead for each parallelization approach?

Table 3: Overhead of multithreading for each method

Implementation	one thread/original
Global Lock	1.097
Transactional Mem	1.191
List Lock	1.116
Element Lock	1.094
Reduction	0.999

Q10 How does each approach perform as the number of threads increases?

For the simpler approaches performance increase was not linear with the number of threads added, and in some cases actually ended up taking longer with more threads. For global lock, 4 threads was actually worse than 2 threads. This may have happened because of all the overhead of switching threads plus the fact that no threads can actually advance until any earlier threads are finished with the locked section. Since the locked section is so large, a lot of threads are doing a lot of waiting.

For each increasingly fine grained approach, we are getting closer to linear performance gains. This is expected, since each method allows for more uninterrupted work to be done by each thread with less waiting for other threads.

Q11 With 100 samples,how does this change impact the results compared with when set to 50?

With more samples to loop through, there is less perceived system overhead proportionately to the workload. More time can be spent doing work by each thread and less time spent checking conditions and switching between threads. For this reason, the performance gains are more apparent with 100 samples than 50 for the high overhead methods such as global lock and transactional memory.

Q12 Which approach should OptsRus ship?

OptsRus should ship the reduction method, as it has linear gains with number of threads used. It is the fastest implementation with any number of samples skipped and any number of threads used. It is objectively the best method. It even has very low overhead so barely affects single cored machines. It would be the fastest on any computer with any amount of cores.