ECE454 Computer Systems Programming
Homework 4: Pthreads and Synchronization

Michael Law - 997376343
Jonathan Ng - 997836141

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

It is important because those methods and datastructures affect the amount of memory used, program size, and the timing of the program's execution.

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

It was the same level of difficulty, the lock and unlock of the global lock was replaced with the opening and closing brackets of __transaction_atomic indicating the critical section of the code.

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

No, because the only public function to the hash class that returns a value is the lookup(), which returns an Ele. WIthout access to the list, we will not be able to implement list locking. The modification can be either, asking the hash class for some information such as the index to the list or the list itself and locking it, or telling the hash class to lock the list by passing in the key.

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

No, because we have not addressed the race case condition where two or more threads are incrementing the count at the same time.

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

No, this would replace the existing randtrack code that calls lookup() and then insert() if lookup() returns a Null value. However, this still contains the race condition for the count increment as in Q4.

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

Implement the simplest solution above that works (or a better one if you can think of one).

Yes, as mentioned in Q3, the lock_list and unlock_list would have a call signature of Keytype the_key and the hash class would handle locking and unlocking by taking this key, mapping it to the index of that list entry in the hash table, and then grabbing/releasing the mutex from the array of mutexes at that index.
This would then be called with within lookup() and insert(), and also from randtrack for the count increment.

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

It was much easier using TM compared to implementing list locking as we did not have to modify the hash class and it there is no overhead and initialization of mutex locks. TM was only two lines of code wrapping the critical section of the code.

Q8. What are the pros and cons of this approach? (Reduction Version)

<threads>	<samples to skip>	<runtime of randtrack_reduction (in seconds)>
1	50	17.76
2	50	8.98
4	50	4.52
1	100	35.14
2	100	17.63
4	100	8.81

As we can see, the pros of this approach is that there is a significant increase in performance as we increase the number of threads. The runtime is consistent as it doubles when we double the number of samples to skip. In fact, these runtimes are the lowest among all of the different experiments.

The cons of the reduction approach is that we are using a lot more memory, each thread has its own copy of the hash table, each containing 2^14 entries, and each entry containing a list of nodes. We are also increasing the amount of work the main thread has to do. Once all threads return from execution, the main function has to process each individual hash table, and every possible key, to combine the hash tables into the main hash table.

Although, impressive performance, we don't believe this to be a scalable approach due to the private copies of the hash table.

4.2 Experiments to Run

Each timing measurement is done 5 times and averaged, and displayed below.

<program num_threads samples_to_skip> <elapsed time (in seconds)>

randtrack 1 50	17.744
randtrank_global_lock 1 50	19.228
randtrank_global_lock 2 50	14.194
randtrank_global_lock 4 50	22.010
randtrack_tm 1 50	21.100
randtrack_tm 2 50	21.298
randtrack_tm 4 50	13.428
randtrack_list_lock 1 50	20.906
randtrack_list_lock 2 50	11.222
randtrack_list_lock 4 50	7.220
randtrack_element_lock 1 50	19.358
randtrack_element_lock 2 50	10.112
randtrack_element_lock 4 50	6.170

Q9. For samples to skip set to 50, what is the overhead for each parallelization approach? Report this as the runtime of the parallel version with one thread divided by the runtime of the single-threaded version.

The overhead of each parallelization approach is displayed below:
runtime of single-threaded version = 17.744
overhead for each = runtime with one thread / runtime of single-threaded version

	runtime with 1 thread (in seconds)	overhead
randtrank_global_lock	19.228	1.084
randtrank_tm	21.1	1.189
randtrank_list_lock	20.906	1.178
randtrank_element_lock	19.358	1.091

Q10. How does each approach perform as the number of threads increases? If performance gets worse for a certain case, explain why that may have happened.

randtrank_global_lock
The runtime the global lock approach decreased when using two threads instead of one, however once we started using four threads, the overhead of running four threads at a time with the lock mechanism was significant enough to have a performance drop worse than running global lock with one thread

randtrack_tm
The transactional memory approach did not see any performance gain from one to two threads, but saw a

significant performance gain from two to four threads.

randtrack_list_lock
The list lock approach shows a constant increase in performance overall as the number of threads increased

randtrack_element_lock
The element lock approach shows a similar trend of performance gain as the list lock approach. Our element lock approach uses one list lock instead of three, and one element lock for the count increment. The decrease in the number of times it is being locked and unlocked accounts for the speedup for this approach over the list lock approach.

Q11. Repeat the data collection above with samples to skip set to 100 and give the table. How does this change impact the results compared with when set to 50? Why?

<program> <num_threads> <samples_to_skip>	<elapsed time (in seconds)>
randtrack 1 100	35.076
randtrank_global_lock 1 100	36.67
randtrank_global_lock 2 100	22.302
randtrank_global_lock 4 100	20.61
randtrack_tm 1 100	38.106
randtrack_tm 2 100	29.844
randtrack_tm 4 100	16.79
randtrack_list_lock 1 100	38.024
randtrack_list_lock 2 100	19.842
randtrack_list_lock 4 100	11.252
randtrack_element_lock 1 100	36.414
randtrack_element_lock 2 100	18.67
randtrack_element_lock 4 100	10.414

The runtime of the approaches with single thread has almost doubled compared with the runtime set to 50. The runtime comparison for two threads is less than double, and the runtime comparison for four threads is about 1.5 times. This shows an increasing decrease in runtime as the number of threads increase.

Q12. Which approach should OptsRus ship? Keep in mind that some customers might be using multicores with more than 4 cores, while others might have only one or two cores.

Since the element_lock approach has the smallest runtime for every case, one thread, two thread, and four

threads. Same comparison when using samples to skip set to 100 instead of 50. Also having the second smallest overhead, with the global lock approach having the smallest. Reduction version is not scalable as discussed. We can conclude to say that OptsRus should use the element locking approach.
