HW 4

Zaara Syeda - 998199765
Ghulam Umar - 998513988

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

Code between ifdef statements is omitted by the compiler on the pre-processor step. Leaving extra methods and data structures in the code will lead to the compiled object file being bloated with useless methods. Also the address space will be needlessly occupied by things that are never used. All of this will mean the code will run slower.

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

Using TM was extremely easy compared to global locks as it required the addition of one line of code.

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

This would possible without modifying the hash class but not without knowing the internal working of the hash class. If we had access to the hashing function it would be possible to create an array of locks in the file calling the hash table functions. This array of locks could be indexed using the hashing function, locking and unlocking only certain locks as required.

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

No, It is not possible as:

1) The element returned is modified outside the hash class; s->count++, this needs to be protected from concurrent writes.
2) The entire function body of lookup and insert needs to be atomic. Therefore, we cannot put locks inside the function since the functions return a value. If we had locked inside, we would have to unlock before the return statement however, this way the entire function is not atomic anymore.

Q5. Can you implement this by adding to the hash class a new function lookup and insert if absent? Explain.

Yes this can be done by doing a lookup to see if the element exists in the hash and then inserting in the same function. This would require that the file calling the lookup and insert will also need to be modified as the new lookup and insert function would need to be surrounded by locks.

Q6. Can you implement it by adding new methods to the hash class lock list and unlock list? Explain.

Yes, this is possible. This implementation addresses the problems mentioned in question 4 by releasing and acquiring locks once all reads and writes on a particular list have been completed by a particular thread.

Q7. How difficult was using TM compared to implementing list locking above?
TM was alot easier than implementing the above locking scheme.

Q8. What are the pros and cons of this approach?
Pros:

1) Requires no synchronization
2) Faster than all other methods of parallelization

Cons:

1) Difficult to implement
2) Requires greater overhead to create and initialize n number of hash tables
3) Poor memory usage
4) Solution does not scale up with the number of threads, without significant effort

sample size 50:

#threads	**global lock**	**tm**	**list level**	**element level**	**reduction**
1	19.428	21.007	19.887	19.566	17.605
2	14.148	21.196	10.722	10.200	8.972
4	24.087	13.381	6.937	6.194	4.516

Original rand_track single thread: 17.760

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.

	global lock	**tm**	**list level**	**element level**	**reduction**
Overhead	1.11	1.18	1.12	1.10	0.991

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.
The general trend is that the performance improves as the number of threads increases. However this is not the same for global_lock. Global locks essentially reduce the parallelization for the program to 1. As the number of threads increase we have more threads

waiting on locks and also have more context switches, all of which incur an overhead on performance, we therefore see a marked increase in run times.

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?

#threads	global lock	tm	list level	element level	reduction
1	36.738	37.822	37.224	36.989	34.786
2	22.596	30.583	19.392	18.897	17.615
4	22.488	16.608	11.198	10.504	8.839

The timings for sample_to_skip set to 100 decreases performance as compared to 50. This is the case because the loop that skips samples runs for a longer period of time not allowing the threads to progresses in execution till 100 samples are skipped.

Q12) Which approach should OptRus 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.

If performance is the most important factor for OptRus, they should ship the reduction version. This version performs very well because no synchronization is required. Since each thread has its own copy of the data, there are no race conditions. However, this approach requires a lot more memory since each thread needs its own hash table.

If memory usage is also important, OptRus should ship element level lock. This option does not use as much memory as reduction and still has good performance. This is because we are using more fine grain locks rather than a coarse grain lock.