CSP Lab 4 Questions

Ryan Santhirarajan 998461338
Ashkan Parcham Kashani 998446105

Q1. Why is it important to #ifdef out methods and data structures that aren't used for different versions of randtrack?
It prevents any conflicting declarations from occurring. The unused data structures should be optimized out by the compiler if it is truly unused.

Q2. How difficult was using TM compared to implementing global lock above?
Implementing TM was slightly easier than implementing global lock, as the initialization and use of the mutex locks was required.

Q3. Can you implement this without modifying the hash class, or without knowing its internal implementation?
It is difficult to do this without knowing the internal implementation as it is neccessary to know how to convert keys to hash table locations. If this is known then the list lock can be implemented.

Q4. Can you properly implement this solely by modifying the hash class methods lookup and insert? Explain.
The critical section of the program is the lookup of a sample and then in the event of missing sample it is inserted. These two actions need to occur atomically. Thus it will not be sufficient as an implementation.

Q5. Can you implement this by adding to the hash class a new function lookup and insert if absent? Explain.
Yes, by having an lookup and insert if absent, the critical section can be made atomic. The situation described above will occur without any interleaving.

Q6. Can you implement it by adding new methods to the hash class lock list and unlock list? Explain.
By Implementing list lock and unlock, these locks will take the place of the simple mutex locks, essentially the list locking functions will act as mutex locks with a knowledge of the underlying hash implementation.

Q7. How difficult was using TM compared to implementing list locking above?
Using TM was extremely easy compared to the list locking implementation.

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

The list locking implementation allows the finer granularity, allowing for better optimization. However this implementation is more complex, requiring a knowledge of the underlying hash implementation. Furthermore there is more overhead with the thread and lock creation.

Measurement results for the different parallelization techniques with a sample skip value of 50

Program	**Number of Threads**	**Time Elapsed**
randtrack	1	0:17.74
randtrack_global_lock	1	0:19.18
randtrack_global_lock	2	0:14.02
randtrack_global_lock	4	0:21.57
randtrack_tm	1	0:21.04
randtrack_tm	2	0:20.93
randtrack_tm	4	0:15.59
randtrack_list_lock	1	0:20.00
randtrack_list_lock	2	0:10.42
randtrack_list_lock	4	0:07.53
randtrack_element_lock	1	0:19.99
randtrack_element_lock	2	0:10.29
randtrack_element_lock	4	0:07.21
randtrack_reduction	1	
randtrack_reduction	2	
randtrack_reduction	4	

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.

Program	**Overhead**
randtrack	**0:17.74/0:17.74 = 1**
rantrack_global_lock	**0:19.18/0:17.74 = 1.08**
randtrack_tm	**0:21.04/0:17.74 = 1.19**
randtrack_list_lock	**0:20.00/0:17.74 = 1.13**
randtrack_element_lock	**0:19.99/0:17.74 = 1.13**
randtrack_reduction	**0:17.74**

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.
As the number of threads approaches the performance increases for each of the parallelization techniques. The exception to this is the global lock, the performance for this decreases when the number of threads is 4. This is probably due to the combination of thread overhead and lack of lock granularity.

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?

Measurement results for the different parallelization techniques with a sample skip value of 100

Program	Number of Threads	Time Elapsed
randtrack	1	0:35.17
randtrack_global_lock	1	0:36.54
randtrack_global_lock	2	0:23.14
randtrack_global_lock	4	0:19.66
randtrack_tm	1	0:39.12
randtrack_tm	2	0:31.92
randtrack_tm	4	0:22.40
randtrack_list_lock	1	0:37.42
randtrack_list_lock	2	0:21.89
randtrack_list_lock	4	0:11.40
randtrack_element_lock	1	0:36.80
randtrack_element_lock	2	0:18.83
randtrack_element_lock	4	0:11.26
randtrack_reduction	1	
randtrack_reduction	2	
randtrack_reduction	4	

The performance is worse because when sample to skip is increased from 50 to 100, the number of times the random generation function is called is doubled, as in order to skip a certain number of samples, the randr function is called on the previous iteration value.

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.
OptsRus should ship the element level lock as it has the least overhead among the parallelization techniques, other than global lock. Furthermore, it along with the list level lock have the best performance. However it seems that as the number of samples to skip increases the performance for the element lock implementation improves faster than the list lock implementation.