**Daniel Chiang**
**998825534**
**d.chiang@mail.utoronto.ca**

**Kunsu Chen**
**999069029**
**kunsu.chen@mail.utoronto.ca**

**Q1. Why is it important to #ifdef out methods and data structures that aren't used for different versions of randtrack? More compilation details for this assignment:**
If we do not #ifdef out the unused data structures and methods out, it could affect the performance and there could be conflicts within the code. The unused structures will still be created by the program and decrease the program run time.

## TM

**Q2. How difficult was using TM compared to implementing global lock above?**
Implementing synchronization using TM is easier than making global lock. Since all that's needed is to wrap the critical section within the __transaction_atromic tag and enabling TM support inside the GNU compiler.

## List Level

**Q3. Can you implement this without modifying the hash class, or without knowing its internal implementation?**
Yes we could do this without changing the hash class. We will initialize a lock for each possible list since the size is given to us. All we needed to do is the make the operations lookup, insert and count++ atomic. Adding a lock around that section in main class would be sufficient.

**Q4. Can you properly implement this solely by modifying the hash class methods lookup and insert? Explain.**
No, we can't. Lookup and insert has to be atomic. If we can only modify the contents of the function lookup and insert, we cannot ensure that no other threads are inserting between the two functions. Unless we copy lookup code into insert and put a lock around the function, but that defeats the purpose of having lookup and insert separately.

**Q5. Can you implement this by adding to the hash class a new function lookup_and_insert_if_absent? Explain.**
Yes we can. After adding lookup_and _insert_if_absent, we can make sure that lookup and insert is done atomically; furthermore, we can increment the count within the function as well.

**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, we can. In order for the lock_list and unlock_list functions to work inside hash class, we need to make the functions public so the main class can call those functions before lookup and after count++. In this case, the initialization of the locks and the lock and unlock will be inside hash class.

**Q7. How difficult was using TM compared to implementing list locking above?**
TM is much easier than implementing list-level locking. For TM, we need to wrap the critical section within the __transaction_atromic tag and enabling TM support inside the GNU compiler. For list level locking, we need to create a mutex lock for each list which takes more time to implement.

**Reduction**
**Q8. What are the pros and cons of this approach?**
The reduction approach eliminates the need for locks so the threads won't be blocked since each thread has its own hash table. The cons include utilizing more memory due to initializing more than one hash table and spending time to merge the tables at the end.

## Measurements

| **samples_to_skip = 50** | Elapsed Time 1 Thread | Elapsed Time 2 Thread | Elapsed Time 4 Thread |
|---|---|---|---|
| randtrack | 10.336 | 10.354 | 10.364 |
| randtrack_global_lock | 10.53 | 5.748 | 4.91 |
| randtrack_tm | 11.254 | 9.518 | 5.964 |
| randtrack_list_lock | 10.70 | 5.552 | 3.078 |
| randtrack_element_lock | 10.61 | 5.52 | 2.924 |
| randtrack_reduction | 10.366 | 5.694 | 4.706 |

**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.**

| **samples_to_skip = 50** | Elapsed Time(s) 1 Thread | Overhead % (Base 10.336) |
|---|---|---|
| randtrack | 10.336 | 100% |
| randtrack_global_lock | 10.53 | 101.88% |

| | | |
|---|---|---|
| randtrack_tm | 11.254 | 108.88% |
| randtrack_list_lock | 10.70 | 103.52% |
| randtrack_element_lock | 10.61 | 102.35% |
| randtrack_reduction | 10.366 | 100.29% |

**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.**
Based on our observation, the performance increases for all the parallelization approaches as the number of the threads increases. This is expected since the newly spawned threads will help reduce the workload and thus reduce the elapsed time.

**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?**

| **samples_to_skip = 100** | Elapsed Time 1 Thread | Elapsed Time 2 Thread | Elapsed Time 4 Thread |
|---|---|---|---|
| randtrack | 20.64 | 20.628 | 20.622 |
| randtrack_global_lock | 20.8 | 11.418 | 7.226 |
| randtrack_tm | 21.606 | 14.686 | 8.296 |
| randtrack_list_lock | 21.076 | 10.726 | 5.748 |
| randtrack_element_lock | 20.932 | 10.644 | 5.6 |
| randtrack_reduction | 20.628 | 10.878 | 5.968 |

This increases the run time for all cases. It's because when samples_to_skip increases from 50 to 100, the innermost for loop calling rand_r() is called 100 times instead of 50. The loop has doubled its iterations and the for loop is ran for every single sample that we wished to collect. Since the for loop has doubled the iterations, we can see the elapsed time is also doubled in single thread mode.

**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.**

The answer is based on the assumption that samples_to_collect is 10000000, RAND_NUM_UPPER_BOUND is 100000, and NUM_SEED_STREAMS is 4.

OptsRus should use list lock in their design to adopt different running environment while maintaining good performance. Although list lock has the second most overhead other than tm in single thread mode, it has the second best performance in 2 thread and 4 thread modes compared to other approaches. The difference in performance gain is even greater when samples_to_skip is higher. Although element lock is the fastest in 2 thread and 4 thread modes, it requires more memory than list lock since every element has its own lock. Moreover, the performance of list lock is not far behind element lock. Having more cores would generally make performance faster but it won’t contribute to one approach more significantly than others. Therefore, list lock is the optimal approach since it has performance close to element lock in all 3 modes while using a lot less memory.