ECE 454 Lab 4 Report

Taylan Gocmen 1000379949
Gligor Djogo 1000884206

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

Answer: Placing #ifdef around unused code will allow the preprocessor to ignore that code, as if it was commented out, by not passing it to the compiler. It is important to remove this code so that upon compilation the total size of the file is minimized and program counter locality can be exploited by not fetching the wrong instructions. Also, the compiler will focus on optimizing the existing parallelization control structures (locks, conditions) and the other unused structures will not interfere with the compilers interpretation of the current code. For all of the reasons above combined we will get a more accurate benchmarking result doing this.

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

Answer: It was slightly easier to implement TM, compared to the Global Lock. Global Lock was done with a few added lines defining the critical section using a global mutex lock, while transactional memory simply required one line and two parenthesis around the critical section and all the locking was done behind the scenes.

Q3) Can you implement this (List-Level Locks) without modifying the hash class, or without knowing its internal implementation?

Answer: No. In order to implement finely grained locks, namely List-level Locks, we need to know what is the index used for hash entries and how it is used, whether it only reads/writes the sections only associated with the index and not the others or not. In this case the index for the hash was a key generated with a set seed stream and it only modifies one list and not the others, therefore we can go ahead and implement a lock system based on this information. If we didn’t have this information we could have implemented an incorrect parallelization, in which threads would overwrite each other’s changes, causing inconsistencies, known as the race condition.

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

Answer: No. Even though this protects against threads overwriting each other's changes, causing
inconsistencies in the hash lists, but it would not protect against the inconsistencies individual elements in the hash lists, ie. sample count race condition. A thread still would be able to access and increment the the sample count from outside the critical section, which needs to be in
the critical section.

Q5) Can you implement this (List-Level Locks) by adding to the hash class a new function lookup_and_insert_if_absent? Explain.

Answer: Yes. If we increment the count as soon as we insert before leaving the aforementioned lookup_and_insert_if_absent function we will have an atomic insert function, essentially having a
finely grained TM implementation. This will also protect against the sample count race condition
described in the Q4 answer.

Q6) Can you implement it (List-Level Locks) 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).

Answer: Yes. We can use lock_list function we wrote in the hash_lock class. This will lock the list
with the given index, key and only one thread will insert a sample and/or increment the sample count.

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

Answer: TM was a lot more easier to implement than List-level Locks. List-level Locks required us

creating a new class hash_lock, based on hash whereas the TM was only 2 lines of code around the
critical section.

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

Answer:
Pros: It removes all the inconsistencies, solves all the race conditions for the parallel threads without the use of locks therefore is quite fast.
Cons: It introduces the memory issue. With large hash structures we can easily run out of memory.
The speedup from the threading part needs to be balanced with the overhead from the reduction
part.

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.

Answer: Got these run time averages from the tests for Q10. All these have an overhead > 1, still Reduction has the smallest overhead since it doesn't use locks.

Parallelization overheads when samples_to_skip is set to 50 while time(randtrack) = 10.364			
	Run time	/ Divide by	= Overhead
randtrack_global_lock	10.556	10.364	1.01853
randtrack_tm	11.258	10.364	1.08626
randtrack_list_lock	10.680	10.364	1.03049
randtrack_element_lock	11.144	10.364	1.07526
randtrack_reduction	10.374	10.364	1.00096

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.

Answer: For each parallelization implementation the performance consistently increases as the number
of threads increases. This means that the parallelization overhead is less than the speedup in the
computation section divided among parallel thread.

The run time averages over 5 runs when samples_to_skip is set to 50				
num_threads	1	2	4	
randtrack	10.364	N/A	N/A	
randtrack_global_lock	10.556	6.480	5.622	
randtrack_tm	11.258	9.508	5.488	
randtrack_list_lock	10.680	5.580	3.046	
randtrack_element_lock	11.144	5.838	3.166	
randtrack_reduction	10.374	5.302	2.776	

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?

Answer: For all implementations, with higher samples_to_skip, though the net run time increases, the
speedup increases as the number of parallel threads increases and the overhead rate decreases and
for Reduction it is < 1, this is due to the portion of the parallelized section increasing in net
CPU time and the effect of parallelization reducing the overall run time.

The run time averages over 5 runs when samples_to_skip is set to 100				
num_threads	1	2	4	
randtrack	20.670	N/A	N/A	

randtrack_global_lock	20.862	11.532	7.798
randtrack_tm	21.584	14.846	8.382
randtrack_list_lock	20.946	10.738	5.864
randtrack_element_lock	21.548	11.062	6.014
randtrack_reduction	20.548	10.448	5.430

Parallelization overheads when samples_to_skip is set to 100 while time(randtrack) = 20.670			
	Run time	/ Divide by	= Overhead
randtrack_global_lock	20.862	20.670	1.00929
randtrack_tm	21.584	20.670	1.04422
randtrack_list_lock	20.946	20.670	1.01335
randtrack_element_lock	21.548	20.670	1.04248
randtrack_reduction	20.548	20.670	0.99410

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.

Answer: For the users that are going to use 1 - 4 cores Reduction is the best implementation, whereas the users that wish to use our implementation with more cores would be best served with a
List-level Lock or Element-level Lock. Considering these, we think List-level lock would be a good mid-way between the two types of users and would give a better market position to OptsRus.