ECE454, Fall 2015
Homework4: Pthreads and Synchronization

Team Herbert!!!
Joo Young Kang(995903330)
He Zhang(1000347546)

Q1

It is important to #ifdef out methods and data structures that aren't used for different versions, because they will consume memory space, affect execution time by reducing overhead of initialization. When they are ifdef'd out, compilers will not look at such methods or data structures that are not being used in current version.

Q2

Using Transactional Memory was not difficult at all as we only needed to put critical sections within __transaction_atomic { *critical section code*…. }. In such cases where we need to implement more than 2 locks, TM method would have been much easier.

Q3

No, since we need to create a lock for each list or hash value inside the hash table. But the list is only known by the hash table so we need to get its hash value.

Q4

No, since we need to keep the s->count ++ in critical section too. And the two methods lookup and insert should be atomic.

Q5

No, this make the lookup and insert atomic. But then the "s->count ++" is not atomic with these two functions.

Q6

Yes,
we can put lookup, insert, and increment of counter inside the lock_list, and then they can be atomic.

Q7

TM implementation was much easier than list-lock implementation which only needed to wrap critical section, while list lock implementation requires analyzing internal implementation of hash table and then create different locks for each list.

Q8

Reduction version performance is significantly faster than any other methods we implemented above. However, this method uses lots of memory since private hash tables are required for each threads.

Measurements

Parallelization Methods	**Performance** (usr/bin/time "elapsed time" in seconds)	
	Samples to skip set to **50**	Samples to skip set to **100 (used for Q11)**
randtrack_**global_lock 1**	10.98	21.31
randtrack_**global_lock 2**	6.31	11.19
randtrack_**global_lock 4**	6.45	8.09
randtrack_**tm 1**	11.33	21.41
randtrack_**tm 2**	9.74	14.70
randtrack_**tm 4**	5.59	8.20
randtrack_**list_lock 1**	11.39	21.45
randtrack_**list_lock 2**	5.90	10.96
randtrack_**list_lock 4**	3.38	5.98
randtrack_**element_lock 1**	11.44	21.51
randtrack_**element_lock 2**	5.80	11.06
randtrack_**element_lock 4**	3.23	5.92
randtrack_**reduction 1**	10.89	21.04
randtrack_**reduction 2**	5.67	10.64
randtrack_**reduction 4**	2.90	5.55

Q9

For samples_to_skip set to 50, overhead for each parallelization methods are listed below.

Overhead for Each Parallelization Approach Base: Randtrack (10.39 Elapsed)	
Global_Lock (1 thread = 10.98)	1.057
TM_Lock (1 thread = 11.33)	1.09
List_Lock (1 thread = 11.39)	1.096
Element_Lock (1 thread = 11.44)	1.101
Reduction Version (1 thread = 10.89)	1.048

Q10

With exception of "Global Lock method", 4 parallelization method scales very well to the number of threads. In the global lack case, there wasn't much difference in performance between 2threads and 4threads, perhaps the latter was worse (06.31 vs 06.45). The reason is that one thread can process regardless of number of threads. 2 threads have reduced the time, however after 2 threads and up performance will be worse due to more context switches and overhead.
Rest of parallelization methods scaled well with relative to the number of threads, fastest performer being the randtrack_reduction.

Q11

Measurements are shown in above table (50 vs 100). In 1 thread case, all methods' execution time approximately doubles. In 2 or 4 thread cases, execution times for "samples_to_skip set to 100" cases cost less than the double of "skip 50". This shows that more threads can scale better when there is more time to deal with the data. This double execution time is due to the time waste on the for loop to skip samples. The global lock's 4 threads scales much better than the one when skip 50 samples because there is more cpu time spent in the skipping for loop before they enter the critical section, so that the cpus are not wasted waiting for other locks but utilized to skip the samples. The list_lock and element_ lock, reduction, tm did well as 50 "samples_to_skip" cases, because they have already utilized the cpu very well.

Q12

Considering that clients have multiple hardware configurations (1, 2, 4, or more cores), we should choose the well-balanced performer in every cases. Therefore, we chose list_lock to be the best overall performer among others in following performance tests (1,2,4 threads and 50 vs 100 samples_to_skip tests). The reasons include the list_lock utilized the multiple cpus very well. And it will not introduce too much overhead as in element_lock so in the case of 1 cpu, it can still perform well. In addition, it won't use that much of memory space as the reduction method since as the sample space get larger, the memory space will be a big problem.