ECE454 Lab 4

Nayeem Husain Zen and Zaid Al-Khishman

Q1) Because we don't want those to be compiled into code that takes up unused memory (including cache) and cause performance hits. This is especially true for an instruction cache that is not fully associative (which is almost always the case). It also increases readability if you have different versions of the program.

Q2) Less difficult. No need to learn or deal with various APIs for dealing with pthread mutexes.

Q3) No. We do not have direct access to the lists from randtrack.cc and consequently we have no way of syncrhonzing access to them.

Q4)Yes, we implement a hash table that associates every list with a lock, lookup->insert sequences must acquire the corresponding lock from the hash table first before entering the critical section

Q5)Yes, from a general overview, this is what the code will be doing i.e any synched code will always end up doing lock -> lookup -> insert -> unlock or lock->lookup->unlock, which reduces to lock->lookup_and_insert_if_absent->unlock. excluding reduction. This is what we implemented.

Q6)Yes, this is a matter of how a lock that corresponds to a specific list is acquired/released. Instead of creating separate data structures we added a lock object to the list class. This way every list ships with its own unique lock straight out of the box.

Q7) Less difficult. Requires little to no knowledge of how the hash class is implemented, and no need to learn/deal with various APIs for dealing with pthread mutexes.

Q8) Pros:

- No synchronization when threads are running without any shared memory
- Threads don't have to stall to acquire locks

Cons:

- Uses a lot more memory (hash table copies)
- Reduction is always a part of the critical path
- Reduction is sequential (although could be parallelized further in a map-reduce style implementation where all the same numbers in each of the hash tables would end up in the same worker thread by means of a hash function, which could proceed to combine the counts without any synchronization)

	1 Thread	2 Threads	4 Threads
original	0:10.36	N/A	N/A
global_lock	0:10.71	0:05.98	0:05.59
TM	0:11.32	0:09.51	0:05.48
list_locks	0:10.79	0:05.58	0:03.16
element_level	0:10.69	0:05.55	0:03.15
reduction	0:10.37	0:05.50	0:02.94

Q9)
Overheads
global_lock: 1.03378
TM: 1.0926
list_locks: 1.0415
element_level: 1.03185
reduction: 1.000965

Q10) As the number of threads increase, the elapsed time for the entire program execution generally decreases. Significant improvements are observed when the randtrack program is ran with two threads compared to only one. Furthermore, running the executable with 4 threads reduced the elapsed time by 55-60% for most synchronization techniques compared to two-thread execution. There were no cases where a synchronization technique became worse as the number of threads increased.

Q11)

	1 Thread	2 Threads	4 Threads
original	0:20.71	N/A	N/A
global_lock	0:20.73	0:11.38	0:07.33
TM	0:21.42	0:14.66	0:08.58
list_locks	0:20.87	0:10.54	0:05.91
element_level	0:21.05	0:10.79	0:05.85
reduction	0:20.78	0:10.82	0:05.64

Q12) Should choose reduction version. Smallest elapsed time compared to all the other approaches for any configuration of threads, and little to no overhead compared to the original single threaded version. So it would perform best even for customers with 1-2 cores or ones with 4 cores.