Assignment 3

April 18th, 2021

ECSE 420
Parallel Computing
Winter 2021
Group 26

Rintaro Nomura - 260781007
Edward Latulipe-Kang - 260746475

Question 1

1.1)
The value *L'* represents the total number of elements that can be contained within the array in a processor cache without the occurrence of a cache miss. If all elements are less than this value, then *T0* represents the time required to access each element in the array as they will be found in the cache. Threads can spin within their respective local location without the need for access to main memory.

1.2)
Should *L'* be sufficiently large, then *T1* represents the time required to fetch data from memory as the cache did not contain the required data. *T1* is held constant, because if an element of the array needs to be retrieved from memory, then each retrieval will take some constant time. Should all elements require access to main memory, then the mean access time will be *T1*. This establishes an upper-bound on the access time.

1.3)
Part 1: This portion of the graph refers to when each element of the array is contained within the cache. More specifically, each element access is effectively a cache hit and the locks spin on their respective bit.

Part 2: This portion of the graph refers to when each element of the lock is contained within the cache, however there is some degree of spillage. That is to say, some elements require memory access. Should a lock spin on a bit requiring memory access, we would be in part 2. This leads to an increase in access time. As the number of elements requiring such an access grows, so too does the overall access time.

Part 3: This portion of the graph refers to when each element of the array requires access to memory. In such a situation, the upper-bound of the access time is reached, and we cannot go beyond that, as that is the slowest possible pace. Every bit on which locks spin requires access to main memory.

1.4)
Using padding ultimately requires a bigger array such that threads can spin unbothered. To that end, more memory is required as a whole for our lock to operate as intended. In this way, Anderson Queue lock can become a big space hog, as we are essentially using one cache line per thread. This is not ideal and would degrade performance where memory is a concern. If a small number of threads require locks, then this method is wasteful.

Question 2

2.1)
The contains() method is a repurposed version of the removes() method. Given that removal of an item from the set requires that said item be first identified in the set, it makes sense to repurpose most of the logic. However, please see FineGrainList.java for the full implementation.

2.2)
One way to test our contains() method is to ensure that some invariant holds true no matter what. We must remember that the invariant, or the truth, is preserved and enforced by all methods and verifies that our implementation behaves correctly when used from multiple threads. Fortunately, this is already established by the algorithms obtained from the "*The Art of Multiprocessor Programming*" textbook. In the event that a contains() is called upon some node or data, we can then confirm the proper functionality of our method. To do so, we simply check to see if each individual thread can run a contains() method to detect an item of the user's choosing. If all threads return the expected outputs, then it proves that our implementation

can handle multithreaded applications, and, therefore, illustrates the overall correctness of our implementation.

Question 3

3.1)

See BoundedLockQueue.java

3.2)

See BoundedLockFreeQueue.java

Since we are using the array instead of a variable, it is challenging to let the array element behave like synchronized. Therefore the difficulty is to determine if a certain index of the array could be read or written (i.e. not "locked"). We may tweak this by making head and tail an atomic variable to keep track of how many elements are under "lock" condition.

Question 4

4.1, 4.2)

See MatrixMultiplier.java

4.3)

For the 2000 threads, the test showed that for the parallel matrix multiplication it took 44 ms, whereas for the sequential matrix multiplication it took 23 ms. In terms of speedup, that is 23 / 44 = 0.52 times speedup (parallel is slower).

Instead, for the 2 threads, the test showed that the parallel matrix multiplication took 21.95 ms. The sequential matrix multiplication took 24.39 m. In terms of speedup, that is 24.39 / 21.95 = 1.11 times speedup (parallel is faster this time).

4.4)

The critical-path length of the implementation would be $\Theta(n)$, because we are splitting the matrix into Threads-amount of pieces, and then spawning the 2 loops that are in parallel, and then there are n executions carried out in the innermost in the loop.

The work would be the same as sequential, as we covered in the lecture, thus $\Theta(n^3)$.

As a result, the parallelism would be

$$T_1(n) / T_\infty(n) = \Theta(n^3) / \Theta(n) = \Theta(n^2)$$