Assignment 3

Stuart Mashaal
260639962

Oliver Tse Sakkwun
260604362

Due: December 4, 2018

Question 1

1.1

The cache-line size is 4 words. Therefore, the total cache size is the $4 \cdot k$ words, where k is the number of cache-lines in the cache.

When all L elements of the array fit in the cache, the average time to access any element of the array will remain constant and small. Therefore, let L' represent the maximum size of the array such that all elements fit in the cache. Note that L' is therefore dependent on that total cache size *but also* on the size of each element of the array. In this example, the size of an element is given as 1 word. Therefore we have

$$L' = \frac{4 \cdot k}{elementsize} = \frac{4 \cdot k}{1} = 4 \cdot k$$

However, we cannot simply say that the space alloted to each element is one word, because the number of words of spacing s between the elements is also a variable. Therefore, the formula becomes

$$L' = \left\lceil \frac{4 \cdot k}{s+1} \right\rceil$$

where $s + 1$ is the amount of space 'allotted' to each element of the array.

We are told in the question that the maximum value that s takes on is $\frac{L}{2}$, which gives us the final formula

$$L' = \left\lceil \frac{4 \cdot k}{\lfloor \frac{L}{2} \rfloor + 1} \right\rceil$$

1.2

In the previous question we discussed how the average time of array access is lowest when the entire array fits in the cache. t_1 is therefore the average time to access a location when there is a cache miss, that is, to access a location from main memory.

It is, of course, only possible to have cache misses when the size of the array, L, is greater than L'. In other words, when the entire array does not fit in the cache.

1.3

The first part of the graph concerns the case then $L < L'$ – when the entire array fits in the cache. We see that the spacing is irrelevant so long as this is true. When $L < L'$, the average access time of the array remains constant at the cache access time.

The second part of the graph concerns the case where $L > L'$, but not so large that every single access is a cache miss. In this case, the average array access time is somewhere in between t_1 (cache access time) and t_2 (memory access time). We see that as s increases, the average access time incrases. This makes sense because as s increases, the number of elements that fit in the cache decreases and thus the probability that an array access is a cache miss increases.

The third part of the graph conerns the case where L is so much larger than L' that every single array access is a cache miss. We see that this occurs when s is large enough that $2 \cdot (s + 1)$ is greater than $4 \cdot k$. In other words, when only a single array element can fit in the cache.

1.4

In the textbook, it was posited that the performance of the Anderson Lock improves when each boolean flag is in its own cache line. It was claimed that the performance improves because *false sharing* is avoided, meaning that when one of the booleans is changed, no other booleans are cache-invalidated for other threads accessing the lock.

However, we have now seen that whatever benefits come of avoiding false sharing, they are probably void in comparison to the massive cost of constant cache misses. Therefore, if the padding between each boolean flag in the Anderson Lock is large enough (and its capacity too), then its performance degrades due to cache misses in accessing the boolean flag array.

There is likely some sweetspot in the size of s, which is dependent on the capacity of the Anderson Lock instance, such that the majority of calls to `lock()` do not cache miss but *do* require accessing a unique cache line.

Question 2

2.1

Please see the code in

`code/src/main/java/ca/mcgill/ecse420/a3/FineGrainedSetMembership/FineGrainedSet.java`

2.2

Please see the test code in

`code/src/test/java/ca/mcgill/ecse420/a3/FineGrainedSetMembership/FineGrainedSetTest.java`

The `contains(T item)` implementation is correct because it both has no concurrency bugs and has no logical bugs.

The only logical bugs that the `contains(T item)` method might have are that it

1. returns `true` when the `item` **is not in** the set, or that it
2. returns `false` when the `item` **is in** the set

The `contains(T item)` method scans the set's internal linked-list until it reaches a node `curr` whose

1. `key` (the `curr.item.hashCode()`) is not less than the `key` being searched for, or
2. `curr.item.equals(item)`

The `contains(T item)` method then returns the value of `curr.item.equals(item)`. This return value can only be `true` when `curr.key == key && curr.item.equals(item)` and is only `false` otherwise. Therefore, the `contains(T item)` method only returns `true` when `curr`'s `item` **is** the `item` being searched for, and can only return `false` when `curr`'s `item` **isn't** the `item` being searched for. So `contains(T item)` is logically correct.

The `contains(T item)` method is also free of concurrency bugs.

It uses hand-over-hand locking in the same order as the `add(T item)` and `remove(T item)` methods to prevent any deadlock or data corruption. Deadlock is impossible because locks are acquired according to a global order, preventing circular wait. Corruption is impossible (that is, conflicting calls to `add(T item)` or `remove(T item)` during the call to `contains(T item)`) because to acquire the lock on `curr`, all methods must first acquire the lock on `pred`, and to release the lock on `curr`, all methods must first release the lock on `pred`. It is therefore not possible for the `contains(T item)` method to be checking the contents of `curr` while a call to `add(T item)` or `remove(T item)` is simultaneously manipulating either `pred` or `curr`.

Question 3

3.1

See the code in `BoundedQueue.java`

3.2

See the code in `LockFreeBoundedQueue.java`

The difficulty is in defining the criteria for permitting read and writes to the array positions. We had to add two more atomic variables: the first representing how many elements can still be inserted; the second represents the valid indices for reading.

Question 4

This question asks us to give an implementation of matrix vector multiplication that is

- highly parallel, having a critical path $\Theta(log(n))$
- efficient, achieving speedup over the naive sequential implementation for $N = 2000$ (where the matrix is $N \times N$ and the vector is $N \times 1$)

We have determined these two requirements to be contradictory and mutually exclusive. The implementation requested is not practically possible.

Meeting the first requirement requires a divide-and-conquer solution like the one outlined in section 4.4. At each 'divide' step, the running thread must create two threads to wait for. For $N = 2000$, this can involve spawning over 8000 such inter-dependent and blocking threads! The number of processors and scheduling efficiency needed to overcome the massive overhead associated with spawning and managing these threads are simply not available on the hardware provided – and might not be available on anything other than research-grade supercomputer.

Nevertheless, we gave our best attempt. We emulated the approach taken by *Herilihy and Shavit* in Chapter 16.1 of *The Art of Multiprocessor Programming*, wherein they give highly parallel (critical path $\Theta(log(n))$) implementations of matrix-matrix addition and multiplication. Despite using only the best inspiration for our algorithm design, our implementation hung indefinitely for $N > 250$ and even crashed our computer for $N > 1000$! For this reason, the benchmarks discussed in section 4.3 use $N = 200$ instead of $N = 2000$.

While the performance of the algorithm is terrible, it is the only one that meets requirements of question 4. So, the answers in sections 4.3 and 4.4 concern this *highly parallel* algorithm that, while not so in practice, is theoretically efficient given thousand of processors and extremely efficient and scalable thread scheduling.

Even though our divide-and-conquer algorithm is technically the right answer, its performance was so terrible that we felt it was necessary to implement a much more practical (though much less parallel) algorithm for comparison. This other algorithm, which can be found in `ParallelMultiplier_Practical.java`, actually achieves speedup over the sequential algorithm for $N > 8000$. In fact, for $N = 25000$ the practical parallel algorithm was twice as fast as the sequential one.

4.1

See the code in `SequentialMultiplier.java`

4.2

See the code in `ParallelMultiplier.java`

4.3

Because of the massive overhead of spawning and scheduling all the highly-interdependent threads, the parallel algorithm actually ran slower given the non-infinite number of processors available. Using the math from section 4.4, we can determine that approximately $\lceil 4 \cdot log_2(2000) \rceil = 44$ threads on 8 virtual processors were used to run the code. However, 44 represents the minimum number of threads necessary, and due to the randomness of thread scheduling it is possible and even likely that a large multiple of this number of threads were actually running.

On a matrix M and vector A of size 200×200 and 200×1 respecitvely, the parallel algorithmn took 800ms whereas the sequential one took only 1ms.

However, increasing the number of processors would, theoretically, linearly speed up the code until the critical path becomes the bottleneck.

If we benchmark the *practical* parallel algorithm against the sequential one, then at $N = 2000$ the sequential algorithm takes 5ms while the practical parallel algorithm takes 14ms. At $N = 25000$, the sequential one takes 201ms while the practical parallel one takes 104ms.

4.4

First, the implementation breaks up the matrix-by-vector multiplication into its various vector-vector dot products. It does this by breaking the matrix into two halves, then halving again and again until it reaches a half that consists of only one row. At this point, it initiates a dot product on that row.

Note that the number of *work nodes* of this step is therefore $\Theta(2N)$, the number of nodes in a fully-balanced binary tree containing N leaves.

Each "leaf" of this binary tree is a dot product to be computed. The dot product is broken up into the sum of two *sub-dot-products*, halving all the way down as before. At the bottom, there is a single two-integer multiplication to do. Once again, this is $\Theta(2N)$ work nodes.

Because $2N \cdot 2N \in \Theta(N^2)$, the implementation achieves work $\Theta(N^2)$.

Also note that the longest possible path through this graph is all the way to bottom of one tree, then another, then back up both of them. That's $\Theta(4 \cdot log_2(N)) \in \Theta(log_2(N))$, so the implementation meets both the *work* and *critical path* requirements.

Since the critical path is $\Theta(log_2(N))$ and the *work* is $\Theta(N^2)$, the parallelism is therefore $\Theta(\frac{N^2}{log_2(N)})$, which is notably much closer to $\Theta(N^2)$ than to $\Theta(N)$. In other words, the parallelism is nearly proportional to N^2 which is also what the work is proportional to, so with ∞ processors this highly-parallel algorithm would run much faster.