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

See in code

2.2

The contains method scans the list to find the pair of nodes (pred,curr) reachable from head such that pred.next == curr,pred.key ¡ key and curr.key ¿= key.The traversal uses hand-over-hand locking.

- Item is not in the list
  When curr.key == key is false that is curr.key ¿ key. From the sortedness invariant of the list, pred.next == curr, and pred.key ¡ key we conclude that item cannot be in the set.
- Item is in the list
  When curr.key == key is true, then from the uniqueness of keys that curr.item = item. Hence, item is in the set.

Question 3

3.1

Working on that

3.2

The difficulty is to define the criteria where the indexes are valid to be read or written. 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

4.3

Given the massive overhead of all the state needed for all the **tasks** that were run on the threads, the parallel algorithm actually ran slower given the non-infinite number of processors available ($\lceil 5 \cdot log_2(2000) \rceil = 55$ threads on 8 virtual processors to be exact).

Specifically, the parallel algorithm ran in 10ms while the sequential ran in 2ms. Not very good speedup, to say the least.

However, the theoretical speedup is linear so long as the critical path is not the bottleneck.

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 proportional to N^2 which is also what the work is proportional to, so with ∞ processors this highly-parallel algorithm would run much faster.