Q1

1. L' is the cache size. Cache is the number of words that can be stored in the cache given stride s.
   t0 is the cache access latency to fetch a cache hit. If L < L′, then the stride size will either be 0 or a size that fits within the case. So, the time to fetch a hit will be constant since they are all cache hits. Therefore, all the time delay for access comes from reading the case.
2. t1 indicates time to fetch for a cache miss. Given that L > L', the words will be separated so that we can't read only from cache. The maximum read time is reached at t1. The stride is larger than the actual cache size. So, it is a constant line. t1 is memory access latency only because we are not reading from cache.
3. When L < L′, all of the data of the array will fit into the cache. So, the time is constant and is equal to the time to access the cache, as stated in question 1.1.
   When L > L', it is divided into two parts.
   In part 2, the stride is continuously increasing and the less elements we can have in our cache. The increasing cache misses are causing the average access time to increase.
   At the end of part 2, stride takes up all of the cache. Every single memory is a cache miss and then go back to memory, which is why average memory is also a constant.
   Part 2means some of values are in cache and some are in main memory while in part 3 all values are in main memory.

4. Padding will degrade the overall performance of the lock because it will make each word larger. That means less amount of works could be stored in the cache, thus causing more cache misses. So, it will increase the memory access time.

Q2.

1. See FineGrainedLock.java
2. The FineGrainedLock list is sorted in ascending hashcode ("key" in Node()) order. We test what will happen when several threads tried to check for the existence of a single node. We know the locks are working because no lock is locked before unlocked. We can also see that all the threads find the target node.

```
Thread with id = 12 Searching for value = 2.
the node = 0 is locked
Thread with id = 13 Searching for value = 2.
Thread with id = 10 Searching for value = 2.
Thread with id = 11 Searching for value = 2.
the node = 1 is locked
the node = 0 is unlocked
the node = 2 is locked
the node = 1 is unlocked
the node = 0 is locked
the node = 2 is unlocked
the node = 1 is locked
the node = 0 is unlocked
Thread with id = 12 found value
the node = 2 is locked
the node = 1 is unlocked
the node = 2 is unlocked
the node = 0 is locked
the node = 1 is locked
the node = 0 is unlocked
Thread with id = 13 found value
the node = 0 is locked
the node = 2 is locked
the node = 1 is unlocked
the node = 2 is unlocked
the node = 1 is locked
the node = 0 is unlocked
the node = 2 is locked
the node = 1 is unlocked
the node = 2 is unlocked
Thread with id = 10 found value
Thread with id = 11 found value
```

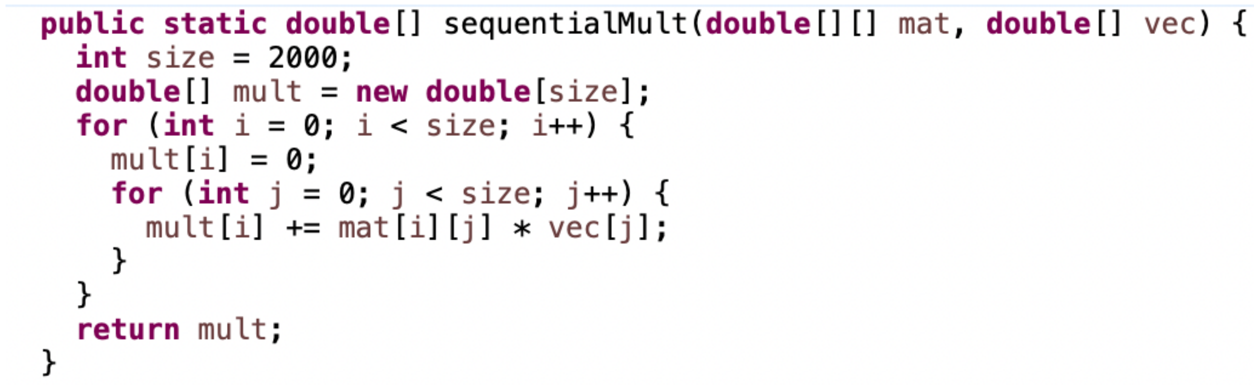
Q3.1. A bounded lock-based queue using array was implemented in this part. Parallelism was implemented using reentrant locks. The head and tail of the queue is kept track of using two integers: head and tail. During enqueue, tail lock is locked and await() is called on the notFull condition and then the list is checked to see if it is full. An item is stored in the array and tail lock is unlocked. The new item is placed as follows in a circular array:

```
items[(tail) % items.length] = item;
```

During dequeue, head locked is locked and then the list is checked to see if it is empty. await() is called if it is empty. If the list is not empty, then return the item that is at the head index of the list. Then finally the lead lock is unlocked.

Q3.2. A lock free implementation was attempted in this part. Enqueue is done by doing a get and increment on the index corresponding to the tail. A linear array with a known size has to be used in this implementation. The atomic integer “size” was used to take care of this. So when it comes to enqueueing, the value of size is used to find out if there is space in the array.

Q4.1.



```
public static double[] sequentialMult(double[][] mat, double[] vec) {
  int size = 2000;
  double[] mult = new double[size];
  for (int i = 0; i < size; i++) {
    mult[i] = 0;
    for (int j = 0; j < size; j++) {
      mult[i] += mat[i][j] * vec[j];
    }
  }
  return mult;
}
```

Q4.3. The parallel multithreaded algorithm takes more time to multiply a matrix of size 2000x2000 with a vector of size 2000 with 16 threads. This might be due to the fact that it is inefficient to create short lived threads as the creation, scheduling and termination of the threads introduce overhead.

Q4.4. The parallel implementation was inspired from chapter 16 in the textbook: The art of multiprocessor programming. The critical path of the parallel implementation is:

$$\begin{aligned} A\infty(n) &= A\infty(n/2) + \Theta(1) \\ &= \Theta(\log n) \end{aligned}$$

The work to add two n x n matrices on one processor is given by the recurrence:

$$\begin{aligned} A1(n) &= 4A1(n/2) + \Theta(1) \\ &= \Theta(n2). \end{aligned}$$

The work of the implementation is:

$$\begin{aligned} M1(n) &= 8M1(n/2) + 4A1(n) \\ M1(n) &= 8M1(n/2) + \Theta(n2) \\ &= \Theta(n3). \end{aligned}$$

The length of the crtical path is:

$$\begin{aligned} M\infty(n) &= M\infty(n/2) + A\infty(n) \\ &= M\infty(n/2) + \Theta(\log n) \\ &= \Theta(\log 2n) \end{aligned}$$

And thus the parallelism is:

$$M1(n)/M\infty(n) = \Theta(n3/ \log 2\ n),$$