**Explanations and Report**

I couldn’t fit in the whole explanation and the thought process in the comments, therefore i am going to provide a little guide into my code, explain my thinking and why i think it improved the performance.

**Task 1.**

The code that we were given was causing a deadlock problem, because of the logic, which was pretty obvious if we ran the program for 2 philosophers. At some point, both of them picked up the available fork at the same time and since neither of them would have let go of it until the other fork was free, they would sit like that forever, in other words, it would cause deadlock.

I have browsed Internet to find the solution to this particular problem and after reading through them, I changed the code below accordingly to this particular solution:

One possible way to fix the deadlock in the code is to change the order in which the philosophers pick up their forks. Instead of always picking up the left fork first and then the right fork,the philosophers can alternate between picking up the left fork first and the right fork first. This will prevent the situation where all philosophers pick up their left fork first and are then waiting for their right fork, leading to a deadlock.

**Task 2.**

When it comes to the Sieve of Eratosthenes, I have followed the instructions.

First of all, I started off by creating a boolean array called prime filled with “true” values and then finding the seed, with help of the function i created called “isPrime”, which returns either true or false. Thus, every number until the square root of the number that is not prime would be counted as false in the prime array.

Then I used the seed to my advantage. Created 4 threads and divided the left space, where we had to find prime numbers into 4 little chunks, to each thread gave the function called ,,notPrime”, start point of the chunk, end point of the chunk, original value (the number until which we wanted to find prime numbers) and the array prime. In the function, I loop over the seeds by creating the loop until the square root of original value and if the number is prime, in other words, if the number is in seed, then I find the new\_start, which is after the start nearest number that can be divided by this seed. From new\_start I start a new loop till the end of the chunk increasing by the amount of seed, which will make sure that all of the i numbers in the array will be divisible by the given seed, therefore I make every value at that particular index in the prime array false.

**Task 3.**

In task 3, I had to implement two kinds of locking: fine-grained and coarse-grained. In the file sorted\_list.hpp i have implemented fine-grained locking and in the file sorted\_list2.hpp i have implemented coarse-grained locking. I’ll provide my reasoning in both implementations in the text below:

1. With fine-grained locking my logic was to create separate locks for each node, which would enable us to only lock specific nodes that we need and not the whole list, while working on different parts of it.  
     
   In the insert function, I decided that it was important to take care of the two nodes, between which a new value had to be inserted, those were: pred and succ. I did that by locking the succ value (unless the array was empty, therefore succ would point to nullptr and we wouldn’t be able to lock it) before entering the while loop.   
   Upon entering the while loop, if pred is not a null pointer, we would unlock pred. Now, pred will not be a null pointer after leaving the loop for the first time, which by then pred will be pointing to the node that succ was pointing before the while loop and succ will be pointing to the next node of it.   
   That’s why we have to lock that new node, which succ is pointing to, but not the one pred is pointing to, since it was already locked, when succ was pointing at it and we never unlocked it.   
   This way after leaving the while loop, both of the nodes that we are concerned about at this particular moment and potentially where we will be inserting our new node(if the condition in the while loop is not true anymore), will be locked and other nodes will be free, unlocked.

After inserting the node successfully, we will be freeing the both nodes that were locked.

The same logic works behind the insertions of lock on the nodes in the remove function.  
  
As for the count function, I have decided that whenever we count, the whole list shouldn’t be locked.

1. The explanation for the coarse-grained locking is quite simple. I inserted the lock guard and locked the whole list whenever we entered any function. This way only one function would be working on the whole list at the same time and the list wouldn’t be under threat (I mean inserting a new node between the nodes, while one of them is being removed or some mistake of that sort).