## Question 1:

The general problem with using ‘for’ loops to initialize vectors is the comparison each rotation encapsulates, which accumulates to a long period, as shown in this question. The usage of STL functions modifies the contents of the vectors directly and does not require the additional comparison time.

STL iota is used to initialize the vector with increasing numbers. STL fill is used to enter the output vectors into the vecOfVecs.

For the next two tasks, STL accumulate is used, once to count the elements’ total using a nested accumulate lambda function to sum the nested vectors, and a second time to “flatten” the vecOfVecs into a one-dimensional array, using a lambda function that concatenates the nested arrays to each other.

## Question 2:

Adding threads to parallelize tasks is easy. The main problem is the global variable ‘sum’ that is modified by multiple threads, which can lead to mistakes in calculation - when multiple threads try to modify the variable at once, the result becomes corrupted.

The solution is to use a mutex to lock the sensitive code in which the common variable is modified and unlock it after the modification is over. That is the purpose of the mutex ‘sumLock’ in the code.

## Question 3:

The usage of a “classic” pointer is problematic when dealing with multiple detached threads – the threads can’t share the information about when the pointer can be deleted (when all threads finished their tasks), and without proper deletion, the memory leaks. Therefore, the pointer ‘payload’ in the original code creates a memory leak.

The way to fix this issue is by using smart pointers, and more specifically shared\_ptr. This pointer keeps a count of the number of instances it has, so when a copy of it is sent to a thread, it increases the count by one. When all instances are deleted and the count depletes to 0, or more specifically when all threads are terminated, the shared\_ptr deletes itself. That makes it the perfect candidate for this question.