# Assignment 2 2INCO

Robert Jong-A-Lock 0726356
Tom Buskens 1378120
October 9, 2018

## 1. Overview

There were 2 main challenges while writing this program:

- Reduce chance of blocking mutex lock
- Limit amount of threads

## 2. Reduce chance of locks waiting

The exercise dictated that a buffer composed of an array of type uint128_t a type (which is 128 bits) is toggled by multiple threads. This buffer must be protected to prevent race conditions. This protection is implemented by using a mutex lock around the computations which should be atomic. This made sure that threads will have to wait on each other to access the buffer. To limit the amount of times the threads must wait, a few optimizations were implemented.

- Every element in the buffer has its own mutex lock. This splits the entire buffer in many smaller parts that can be accessed concurrently.
- All the masks required for the current element in the array are generated first. These are then combined and utilized at once. This approach reduces the amount of time the thread needs to lock an element in the buffer. These masks are used inside the critical section to toggle all targeted bits with one bitwise XOR operation.
- When a mask only contains zeros, it is not necessary to modify the buffer element at that location since no bits need to be toggled (which is what a zero mask means). Therefore, buffer indexes with these masks are just skipped.
- Lastly, every thread starts applying its mask at a different start position in the buffer. So, each thread loops through all elements but starts at a different index in the buffer. The indices where threads start are evenly distributed across the buffer.

## 3. Limit amount threads

Another aspect of the exercise was to limit the amount of threads that run at the same time. Additionally, threads should be restarted as soon as possible after they are finished. This was accomplished by keeping track of the state of each (worker) thread.

The application utilizes a semaphore which is used to block the main thread. Initially all worker threads are in the *uninitialized* state. During this phase the semaphore will not block and worker threads are started until the maximum amount of threads is reached. At this point the main thread will be blocked by the semaphore. When a thread is started it updates its state from *uninitialized* to *busy* Indicating that they are processing the buffer.

At some point one of the threads will finish, put its state to *finished* and increase the semaphore. This will unblock the main thread which then starts looking through all the thread states to find the one that signaled that it’s done. When found the main thread it will join with the thread and start a new one. After this the main thread will block again.

This concept keeps working when multiple threads finish at the same time because the semaphore will unlock for each thread that finishes.

A mutex cannot not be used to accomplish this behavior because it’s possible that multiple threads finish at the same time which will result in unlocking a mutex multiple times. Unlocking a mutex more than once is not defined and therefore should not be used to signal state changes such as this.