# Homework 3

**due date: 28.6 23:59**

**General Guidelines:**

- Submit a zip file with all source code + relevant files (e.g., MSVS soln. files) to build your programs.
- Your code should be compiled and run in release/x64 (in MSVS) or "g++ -O3" in WSL/Ubuntu
- Use only native C++11 (no STL)
- In case of g++, include documentation of the g++ command-line options (e.g., enabling extended instruction set) needed for building your programs.
- Don't use any external libraries except possibly TBB.
- Read the below instructions carefully.

**Exercise 1: Parallel Nbody simulation – 70 points**

The Nbody simulation simulates the movement of particles in a 3D space due to gravity forces between every two particles.

You are given a serial Implementation of the Nbody simulation in the attached Nbody.cpp. Your goal is to add an optimized parallel code into this source code.

1. The minimum required is both multi-threading and vectorization.
2. You are not allowed to change the serial code implementation.
3. For the parallel version you should add two functions and integrate them inside *run_simulation*. Look for "TODO" comments inside the code.
4. The code includes validation. The parallel code should behave (results) exactly as the serial code.
5. Submit the modified Nbody.cpp with your addition of the parallel version + additional source code files, if any.
6. Vectorization can be applied either via intrinsic ops or auto-vectorization (possibly using pragmas).
7. If you use intrinsic ops,
    a. With g++, use the *-mavx* command-line param (e.g., -mavx2 for AVX2). See https://gcc.gnu.org/onlinedocs/gcc/x86-Built-in-Functions.html
    b. With MSVS, enable "Enable Enhance Instruction Set" in the solution properties→C/C++→code generation.
8. References to intrinsic ops include:
    a. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html (press the right checkbox for vectorization ISA of your machine e.g., AVX2.)
    b. The example with intrinsic ops in lecture presentation "multicore processors -part I"
9. Feel free to change the total number of particles, if needed.
10. You are NOT forced to use anything from the serial implementation in your parallel version. Your parallel version can be completely new.

**Exercise 2: Hierarchical locks – 30 points**

Implement a hierarchical lock class which complies with the attached Hlock.hpp.

The purpose of a hierarchical lock is to enforce ordering of locks and unlock ops so that deadlocks are prevented.

A hierarchical lock is a Mutex. The main difference with std::mutex is that:

- 
- A hierarchical lock has a unique level
- Its lock op should fail if <u>the same thread</u> already locked another hierarchical lock with higher (or equal) level. In this case, the lock op should throw a runtime error.
- Unlock ops <u>by the same thread</u> should be applied in the exact reversed order of their corresponding lock ops (otherwise, some future lock ops may throw the runtime errors). <u>You can assume this usage pattern. No need to enforce it in your implementation.</u>

Given:

```
HierarchicalMutex mutexA(1000);

HierarchicalMutex mutexB(10000);
```

The following is a <u>valid</u> use case (assume 1 thread):

```
mutexA.lock();

mutexB.lock();

mutexB.unlock();

mutexA.unlock();
```

The following is <u>an invalid</u> use case (assume 1 thread):

```
  Thread 1                      Thread2

mutexA.lock();

                        mutexB.lock();

mutexB.lock();

                        mutexA.lock(); // should throw a runtime error
```

1. You are not allowed to change the current public part of the Hlock class in Hlock.hpp. You can freely add a private part.
2. You can use std:: mutex for the basic lock/unlock of an hierarchical lock.
3. Submit an Hlock.cpp file with the class implementation.
4. Submit a separate main.cpp file which demonstrates the usage of the hierarchical lock.

Good luck !