Homework 3

Due date: Aug 21, 23:59

General Guidelines:

- This homework assignment includes also several .h and .cpp files. Read ahead.
- Make sure you use the exact file names, function names and class names mentioned below. Otherwise, you will face compilation errors during automatic testing.
- Don't use any external libraries (not part of C++ standard), except TBB, for this homework.
- The homework assignment is 100% auto graded. You can resubmit your solution to gradescope multiple times before the due date of this homework assignment.
- In total, you should upload (together) two files to gradescope:
  - nbody_impl.h.h (exercise #1)
  - hlock_impl.h (exercise #2)
- The automatic tests for this homework can run up to **few minutes**, per submission.

Exercise 1: Parallel Nbody simulation – 70 points

The Nbody discrete simulation simulates the movement of particles in a multi-dimensional space due to gravity forces between every two particles (https://en.wikipedia.org/wiki/N-body_problem). It can naturally be parallelized over sequential implementations.

You are given a fully sequential implementation of the Nbody simulation in 3D space. Your goal is to develop the most efficient parallel implementation for this Nbody simulation.

1. You are given *nbody.h* which includes three functions with the suffix of *"_serial". These* are the building blocks for the sequential Nbody simulation. Don't change the sequential code!
2. This *nbody.h* also includes the signatures of three equivalent functions (with the "_parallel" suffix) that you need to implement for the parallel version.
3. You should create *nbody_impl.h* (make sure to "*#include "nbody.h")* with the implementation of these missing *"_parallel"* functions.
4. You can add data structures and help functions to *nbody_impl.h, as needed.*
5. You are also given *main_nbody.cpp*. It is the reference code we use in Gradescope to measure the speedup of your parallel implementation. It can help you with the development, debugging and measuring the speedup of the parallel code
6. Don't try to upload *main_nbody.cpp* to gradescope. **The only source file you should eventually upload to gradescope is *"nbody_impl.h".***

The following are some additional guidelines for development:

1. Overall, learn all existing code carefully before you start to develop the parallel version.
2. Your parallel implementation should be based on both multi-threading and vectorization (you can assume AVX2 support in gradescope).
3. To ensure the correctness of your parallel code, you should initialize the particles in your parallel implementation with the exact values used in the reference sequential code. Otherwise, the correctness tests will fail.

4. If you plan to directly use AVX2 to vectorize your code, make sure to have *"include <immintrin.h>"* inside your code and use the additional "*-mavx2"* command-line flag. (examples of using AVX2 included in the course material on vectorization).
5. All your code should be compiled with "-O3" g++ flag (maximum optimizations).
6. Due to dynamic resource allocation in gradscope, your parallel version should be independent of a specific number of threads. Hence, you should use only the TBB library for multi-threading (No C++ threads)
7. You can assume Intel cores with L1 cache size=32KB and cache line size=64B.
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. An example of vectorized code with intrinsic op in the lecture presentation titled *"multicore processors -part I"*
9. You are NOT forced to reuse code from the serial implementation for your parallel version. Your parallel version (the functions with "__parallel" suffix) can be developed completely from scratch.
10. See additional guidelines given in lecture #9 regarding the submission and grading of this exercise.

Exercise 2: Hierarchical locks – 30 points

Implement a hierarchical lock. A hierarchical lock enforces ordering of locks and unlock ops to prevent deadlocks. A hierarchical lock is basically a mutex with the following differences:

- Every hierarchical lock has a unique level (aka id)
- The lock op should fail if the same thread already locked another hierarchical lock with higher (or equal) level.
- Upon a failure to acquire a lock, the *lock* op should throw an exception object of the HierarchicalMutexException class (defined in hlock.h). Use the C++ "throw" mechanism.
- Unlock ops by the same thread should be applied in the exact reversed order of their corresponding lock ops (otherwise, some future lock ops may throw the runtime errors). You can assume this usage pattern. No need to enforce it in your implementation.

You should develop a hierarchal lock class *"HierarchicalMutex_impl"* which is a subclass of the *HierarchicalMutex* base class (defined in *hlock.h)*.

1. You can use *std:: mutex* for the basic lock/unlock of an hierarchical lock
2. Submit *hlock_impl.h* file with your implementation of *HierarchicalMutex_impl* class.

Few examples:

```
HierarchicalMutex_impl mutexA(1000);

HierarchicalMutex_impl mutexB(10000);


The following is a valid use case (assume 1 thread):

mutexA.lock();
```

```
mutexB.lock();
mutexB.unlock();
mutexA.unlock();
```

The following is an invalid use case (assume 2 thread):

```
  Thread 1                        Thread2
mutexA.lock();
                           mutexB.lock();
mutexB.lock();
                           mutexA.lock(); // should throw a runtime error
```

Good luck!