Lab 4

Lab IV

Today we will get familiar with some concepts concerning concurent processing. We will try to solve the problems as we proceed with the material.

We will start with some simple examples of starting and managing threads using standard C++ tolls. We will try to show some common problems and solutions whan dealing with concurent processing.

Threads

To compile programs using `std::thread` use `g++ thread1.cpp -pthread`

Starting and joining

The easiest way to manage threads is to add to your source `#include <thread>` and make do wiht `std::thread` objects. New threads asre started by creating new `std::thread` instances, and passing a collable (one that has () and can be called) object to be executed in the new thread. Once the thread is started it executes, until it finishes. The first problem to look at is not to leave alive threads for control to go out of scope (i.e. to go out of the {} region), since this will cause running threads to trtminate and is in general an undefined behavior. To prevent this, threads should be joined with `.join()` (C++20 introduces a `jthread` that supports auto-joining).

See thread1.cpp, compile and run it, examine the commented out synchronization section.

Detaching

Threads started within a scope live only within this scope, and will be terminated when it finishes. This might be a problem for sytuations one wishes to use some form of thread caller functions. To prevent the thread execution from being abruptly terminated we can use `detach`, which detaches execution from the thread object. Note: Should be used with care.

See thread2.cpp

Race condition

Examine thread3.cpp, compile and run. What is the result of the program? Is it the same every time? Is this a problem? At least the program works, right?

Experiment with the `atomic` template. Is it any better now?
Modify incrementation of the `thread_counter`, is there any difference in using `thread_counter = thread_counter + 1` or `thread_counter += 1`?

Mutex and the critical section

Threads working concurrently use the same memory space, and as we have seen the consequent race condition is a problem. The region of the code our threads might interfere is the **citical section**, one that needs to be appropriatly protected. We will use the `mutex` (Mutually Exclusive Lock) mechanism. See thred4.cpp for an example. Compile and run the code. Than, modify thread3.cpp to use `mutex` instead of `atomic`.

TBB

Thread Building Blocks is a library designed to make parallelization of the already existing, or new code (relativly easy). The objective of TBB is to provide a template library, very much like STL for managing creation of and working with threads.

Serial implementation

Examine serial.cpp, compile and run it. It is a very simple program that puts data in to an array and terminates. Nothing fancy.
Extend the code, so an average value is evaluated at the end.

prallel_for

The for loop used to initialize the vector content is a great example of an embarrassingly parallel problem. There should be no data race, and no problem with concurent manipulation, and it should parallelize very eaisly. For this task, TBB ofers a simple `parallel_for`, which with little modification to the code manages starting, stoping and executing working threads.

```
tbb::parallel_for(range, kernel);
```

For `range` we will use `tbb::blocked_range<T>(T n0, T n1)`, which is a template class describing a one-dimensional iteration space from n_0 to $n_1 - 1$. For our problem it is the same as the the range of the for `(0,y.size())`.
`kernel` is a collable object that takes `tbb::blocked_range<T> r` as an argument and processes the chunk of the problem defined by `r`. In our problem we will use lambda expression `[&](tbb::blocked_range<int> r){}`.

Note: Our problem is very simple, but a rule of thumb is to put frequently accessed values into local variables of the `kernel`. This should help compiler to optimize the loop better. It seems local variables are easier for the compiler to track.

Examine the execution times and prepere a speedup plot using `tbb_parallel_for.cpp`. Than move the definition of `double x` out of the body of the lambda expresion (Note, this causes the race condition!) and examine resulting speedup, is there any difference?

False Sharing

is a performance degrading event that results from threads sharing resorurces that *lie to close to each other.*

> When a system participant attempts to periodically access data that is not being altered by another party, but that data shares a cache block with data that is being altered, the caching protocol may force the first participant to reload the whole cache block despite a lack of logical necessity.

Examine, compile and run `tbb_false_sharing.cpp` example.

parallel_reduce

The loop initializing the data was easy to parallelize with `parallel_for`. How about summation over all elements? In this operation elements of an array are *reduced* into a single result - the sum. TBB offers a `parallel_reduce` function template to perform reduction operations over the range. The simplest syntax is `parallel_reduce(range, identity_value, func, reduction)`, where: * `range` defines a a range, to which sub-ranges `func` will be applied. * `identity_value` is identity for the operation performed by `func` (0 for summation and 1 for multiplication). * `func` is a collable object. Could be a lambda expression. * `reduction` defines how sub-range reductions are joined to produce the final result. This can also be defined as a lambda expression, or we could use standard function objects.

More complex problem

Solve linear advection problem $\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0$, with periodic boundary conditions and an appropariate initial condition. Use `advection_serial.cpp` as a starting point, here first order forward finite difference and an explicit time integration is used. Examine possible speedup due to parallelization. Then consider implementation of the implicit method.