Par/Seq Policy Usage

Let's compare different situations to see where parallel execution works better.

WE'LL COVER THE FOLLOWING

Why do you need the sequential policy?

Most of the time you'll be probably interested in using parallel policy or parallel unsequenced one. But for debugging it might be easier to use `std::execution::seq`. The parameter is also quite convenient as you might easily switch between the execution model using a template parameter. For some algorithms, the sequential policy might also give better performance than the C++14 counterpart.

> Read more in the Benchmark section.

Limitations and Unsafe Instructions

The whole point of execution policies is to parallelize standard algorithms in a declarative way effortlessly. Nevertheless, there are some limitations you need to be aware of.

For example with `std::par` if you want to modify a shared resource you need to use some synchronization mechanism to prevent data races and deadlocks *[^readaccess]*:

> *[^readaccess]:* When you only want to read a shared resource, then there's no need to synchronise.

```
std::vector<int> vec(1000);
std::iota(vec.begin(), vec.end(), 0);
std::vector<int> output;
std::mutex m;
std::for_each(std::execution::par, vec.begin(), vec.end(),
[&output, &m, &x](int& elem) {
    if (elem % 2 == 0) {
        std::lock_guard guard(m);
        output.push_back(elem);
    }
});
```

Avoid locking inside a parallel operation

The above code filters out the input vector and then puts the elements in the output container.

If you forget about using a mutex (or another form of synchronization), then `push_back` might cause **data races** - as multiple threads might try to add a new element to the vector at the same time.

The above example will also demonstrate weak performance, as using too many synchronization points kills the parallel execution.

> When using `par` execution policy try to access the shared resources as little as possible.

With `par_unseq` function invocations might be interleaved, so it's forbidden to use unsafe vectorized code. For example, using mutexes or memory allocation might lead to data races and deadlocks.

```
std::vector<int> vec = GenerateData();
std::mutex m;
int x = 0;
std::for_each(std::execution::par_unseq, vec.begin(), vec.end(),
              [&m, &x](int& elem) {
                std::lock_guard guard(m);
                elem = x;
                x++; // increment a shared value
              });
```

Unsafe Instructions in Par Unseq Execution

Since the instructions might be interleaved on one thread, you may end up with the following sequence of actions:

```
std::lock_guard guard(m) // for i-th element
std::lock_guard guard(m) // for i+1-th element
...
```

As you can see, two locks (in the same mutex) will happen on a single thread causing a deadlock!

> Don't use synchronisation and memory allocation when executing with `par_unseq` policy.

In the next lesson, we will take a look at situations in which exceptions arise. Keep on reading to find out more!