Execution Policies

This lessons discusses the three types of execution policies as well as the differences between them.

WE'LL COVER THE FOLLOWING

Execution Policy Parameter

The execution policy parameter will tell the algorithm how it should be executed. We have the following options:

Policy Name	Description
`sequenced_policy`	It is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and requires that a parallel algorithm's execution not be parallelised.
`parallel_policy`	It is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and indicate that a parallel algorithm's execution may be parallelised.
`parallel_unsequenced_policy`	It is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and indicate that a parallel algorithm's

	execution may be parallelised and vectorised.

We have also three global objects corresponding to each execution policy type:

- `std::execution::seq`
- `std::execution::par`
- `std::execution::par_unseq`

> Please note that execution policies are unique types, with their corresponding global objects. They are not enumerations, nor do they share the same base type.

Execution policy declarations and global objects are located in the `<execution>` header.

Understanding Execution Policies

To understand the difference between execution policies, let's try to build a model of how an algorithm might work.

Consider a simple vector of `float` values. In the below example, each element of the vector is multiplied by 2 and then the result is stored into an output container:

```
std::vector<float> vecX = {...}; // generate
std::vector<float> vecY(vecX.size());
std::transform(
  std::execution::seq,
  begin(vecX), end(vecX),            // input range
  begin(vecY),                       // output
  [](float x) { return x * 2.0f; }); // operation
```

Here's a pseudo code for sequential execution of the above algorithm:

```
operation
{
    load vecX[i] into RegisterX
    multiply RegisterX by 2.0f
    store RegisterX into vecY[i]
}
```

In the sequential execution, we'll access one element (from `vecX`), perform an operation and then store the result into the output (`vecY`). Execution for all elements happens on a single thread (on the calling thread).

With the `par` policy the whole `operation` for the i-th element will be executed on one thread. But there may be many threads that process different elements of the container. For example, if you have 8 free threads in the system, 8 elements of the container might be computed at the same time. The element access order is unspecified.

The Standard Library implementations might usually leverage some thread-pool to execute a parallel algorithm. The pool holds some worker threads (generally as many as system cores count), and then a scheduler will divide the input into chunks and assign them into the worker threads. In theory, on a CPU, you could also create as many threads as elements in your container, but due to context switching that wouldn't give you good performance. On the other hand, implementations that use GPUs might provide hundreds of smaller "cores" so in that scenario the scheduler might work entirely differently.

The third execution policy `par_unseq` is an extension of the parallel execution policy. The operation for the i-th element will be performed on a separate thread, but also instructions of that operation might be interleaved and vectorized.

For example:

```
operation
{
    load vecX[i...i+3] into RegisterXYZW
    multiply RegisterXYZW by 2 // 4 elements at once!
    store RegisterXYZW into vecY[i...i+3]
}
```

The pseudocode above uses `RegisterXYZW` to represent a wide register that could store 4 elements of the container. For example, in SSE (Streaming SIMD Extensions) you have 128-bit registers that can handle 4 32-bit values, like 4 integers or 4 floating point numbers (or can store 8 16-bit values). However, such vectorization might be extended to even larger registers like with AVX

where you have 256 or even 512-bit registers. It's up to the implementation of the Standard Library to choose the best vectorization scheme.

In this case, each instruction of the operation is "duplicated" and interleaved with others. That way the compiler can generate instructions that will handle several elements of the container at the same time.

In theory, if you have 8 free system threads, with 128-bit SIMD registers, and we process float values (32-bit values) - then, we can compute 8*4 = 32 values at once!

Let's dig deeper into the details of parallel and sequential policies in the next lesson!