Summary

This section provides a summary of all the major concepts discussed throughout the section.

After reading the chapter, you should be equipped with the core knowledge about parallel algorithms. We discussed the execution policies, how they might be executed on hardware, what are the new algorithms.

At the moment, parallel algorithms show good potential. With only one extra parameter, you can easily parallelise your code. Previously that would require to use some third-party library or write a custom version of some thread pooling system.

For sure, we need to wait for more available implementations and experience. Currently, only Visual Studio and GCC 9.1 let you use parallel algorithms, and we're waiting for Clang's library to catch up. Executing code on GPU looks especially interesting.

It's also worth quoting the TS specification P0024:

> The parallel algorithms and execution policies of the Parallelism TS are only a starting point. Already we anticipate opportunities for extending the Parallelism TS's functionality to increase programmer flexibility and expressivity. A fully-realised executors feature will yield new, flexible ways of creating execution, including the execution of parallel algorithms.

Things to remember

- Parallel STL gives you a set of 69 algorithms that have overloads for the

- Parallel STL gives you a set of 69 algorithms that have overloads for the execution policy parameter.

- Execution policy describes how the algorithm might be executed.
- There are three execution policies in C++17 (`<execution>` header)
    - `std::execution::seq` - sequential
    - `std::execution::par` - parallel
    - `std::execution::par_unseq` - parallel and vectorised
- In parallel execution policy functors that are passed to algorithms cannot cause deadlocks and data races
- In parallel unsequenced policy functors cannot call vectorised unsafe instructions like memory allocations or any synchronisation mechanisms
- To handle new execution patterns there are also new algorithms: like `std::reduce`, `exclusive_scan` - They work out of order so the operations must be associative to generate deterministic results
- There are "fused" algorithms: `transform_reduce`, `transform_exclusive_scan`, `transform_inclusive_scan` that combine two algorithms together.
- Assuming there are no synchronisation points in the parallel execution, the parallel algorithms should be faster than the sequential version. Still, they perform more work - especially the setup and divisions into tasks.
- Implementations might usually use some thread pools to implement a parallel algorithm on CPU.

Now that all your concepts have been refreshed it's time for a short quiz. Click on the next lesson to give it a try!