C++17's Parallelism

This chapter talks about CPU and GPU computing. It also gives a basic overview of the execution policy parameter introduced in C++ 17

WE'LL COVER THE FOLLOWING

Not Only Threads

As mentioned earlier, using threads is not the only way of leveraging the power of your machine.

If your system has 8 cores in the CPU then you can use 8 threads and assuming you can split your work into separate chunks then you can theoretically process your tasks 8x faster than on a single thread.

But there's a chance to speed up things even more!

So where's the rest of the power coming from?

Vector Instructions from CPU & GPU computing

The first element - vector instructions - allows you to compute several components of an array in a single instruction.

> It's also called *SIMD - Single Instruction Multiple Data*.

Most of the CPUs have 128-bit wide registers, and recent chips contain registers even 256 or 512 bits wide (AVX 256, AVX 512).

For example, using AVX-512 instructions, you can operate on 16 integer values (32-bit) at the same time!

The second element is the GPU. It might contain hundreds of smaller cores.

There are third-party APIs that allow you to access GPU/vectorization: for example, we have CUDA, OpenCL, OpenGL, Intel TBB, OpenMP and many more. There's even a chance that your compiler will try to auto-vectorize some of the code. Still, we'd like to have such support directly from the Standard Library. That way the same code can be used on many platforms.

C++17 moves us a bit into that direction and allows us to use more computing power: it unlocks the auto-vectorization/auto-parallelization feature for algorithms in the Standard Library.

Overview

The new feature of C++17 looks surprisingly simple from a user point of view. You have a new parameter that can be passed to most of the standard algorithms: this new parameter is called **execution policy**.

This has been introduced in the `<execution>` library.

> Unfortunately, library support for GCC has not been released yet (2019). All the code shown in this section will work perfectly on MSVC.

Here's a template for passing the new execution policy parameter:

```
algorithm_name(policy, /* normal args... */);
```

We'll go into the details later, but the general idea is that you call an algorithm and then you specify **how** it can be executed. Can it be parallel or just serial?.

For example:

```
std::vector<int> v = genLargeVector();
// sort a vector using a parallel policy
std::sort(std::execution::par, v.begin(), v.end());
```

```
std::sort(std::execution::par, v.begin(), v.end());
```

The above example will sort a vector in parallel - as specified by the first argument `std::execution::par`. The whole machinery is hidden from a user perspective. It's up to the STL implementation to choose the best approach to run tasks in parallel. Usually, they might leverage thread pools.

Hint - *the execution policy parameter* - is necessary because the compiler cannot deduce everything from the code. You, as the author of the code, only know if there are any side effects, possible race conditions, deadlocks, or if there's no sense in running in parallel (such as if you have a small collection of items).

> C++17's Parallelism comes from the Technical Specification that was published officially in 2015. The whole project of bringing parallel algorithms into C++ took more than five years - it started in 2012 and was merged into the standard in 2017. See the paper: P0024 - The Parallelism TS Should be Standardised

---

The next lesson will explain the execution policies in detail. Read on to find out more!