Performance of Parallel Algorithms

This lesson explains in detail the performance of parallel algorithms and the factors that affect it.

Factors affecting execution

Parallel algorithms are a robust abstraction layer. Although they're relatively easy to use, assessing their performance is less straightforward.

The first point to note is that a parallel algorithm will generally do more work than the sequential version. That's because the algorithm needs to set up and arrange the threading subsystem to run the tasks.

For example, if you invoke `std::transform` on a vector of 100k elements, then the STL implementation needs to divide your vector into chunks and then schedule each chunk to be executed appropriately. If necessary, the implementation might even copy the elements before the execution. If you have a system with 10 free threads, the 100k element vector might be divided into chunks of 10k, and then each chunk is transformed by one thread. Due to the setup cost and other limitations of parallel code, the whole execution time won't be 10x faster than the serial version.

The **second** factor that plays an important role is synchronisation. If your operations need to synchronise on some shared objects, then the parallel execution performance decreases. A parallel algorithm performs best when you execute separate tasks.

The **third** element that has a big effect on the execution is memory throughput. If we look at a common desktop CPU, we can see that all cores

share the same memory bus. That's why if your instructions wait for the data to be fetched from memory, then the performance improvement over the sequential won't be visible, as all cores will synchronise on the memory access. Algorithms like `std::copy`, `std::reverse` might be even slower than their serial versions - at least on common PC hardware[^copylimitations]. It's best when your parallel tasks use CPU cycles for computation rather than waiting for memory access.

[^copylimitations]: See section "Current Limitations of the MSVC Implementation of Parallel Algorithms" in Using C++17 Parallel Algorithms for Better Performance | Visual C++ Team Blog where parallel `std::reverse` appeared to be 1.6 times slower.

The **fourth** important thing is that the algorithms are very implementation-dependent. They might use different techniques to achieve parallelism. Not to mention the device that an algorithm might be executed on - on CPU or GPU.

Right now (as of July 2019) there are two implementations available in a popular compiler - starting with Visual Studio 2017 and GCC 9.1.

The Visual Studio Implementation is based on thread pools from Windows and only supports execution on the CPU and skips the vectorisation execution policy [^vsparunseq].

> [^vsparunseq]: The VS implementation will usually process elements in blocks, so there's a chance that auto-vectorisation might still optimise the code and allow vector instruction for better performance.

In GCC 9.1, we got parallel algorithms that are based on a popular Intel implementation - PSTL. Intel offered the implementation, update the licence so that the code could be reused in the Standard Library. Internally it requires OpenMP 4.0 support and Intel TBB to be linked with the application.

Another thing is the GPU support. With hundreds of smaller computing cores, the algorithms might perform faster than their CPU version. It's important to remember that before executing something on the GPU, you have to usually copy the data to memory visible to the GPU (unless it's a shared memory like in integrated Graphics Cards). And sometimes the cost of the data transfer

might reduce the total performance.

> If you decide to use parallel algorithms, it's best to measure the performance against the sequential version. Sometimes, especially for a smaller number of elements, the performance might be even slower.

Examples

So far you've seen basic code samples with parallel algorithms. This section will introduce a few more examples with more complex scenarios.

- You'll see a few benchmarks and see the performance gains over the sequential version.
- We'll discuss an example of how to process several containers in the same parallel algorithm.
- There will also be a sample of how to implement a parallel version of counting elements.

Please note that all of the measurements for benchmarks were done only using Visual Studio - as it's the only available implementation of parallel algorithms in a popular compiler. The benchmarks will be updated when implementations, from GCC and Clang, will be ready.

Let's take a look at benchmark measurements in more detail in the next lesson.