Benchmark I

This section will give a comparison of sequential vs parallel computing using Run and Measure.

Finally, we can see the performance of the new algorithms.

Let's have a look at an example where we execute separate tasks on each element - using `std::transform`. In that example, the speed up vs the sequential version should be more visible.

> When comparing the parallel and serial execution time for the transform function using Benchmark I, the results *sometimes* indicate that parallel execution takes more time than serial execution. The examples in this course use a machine with single hardware Hyper-thread with 3.75 GB of memory. You can disregard this unexpected result; on a traditional CPU, that usually has several hardware threads, parallel execution times would be lower than serial ones.

input.cpp

simpleperf.h

```
#include <algorithm>
#include <execution>
#include <cmath>
#include <iostream>
#include <numeric>
#include "simpleperf.h"

int main(int argc, const char* argv[])
{
    const size_t vecSize = argc > 1 ? atoi(argv[1]) : 6000000;
    std::cout << vecSize << '\n';
    std::vector<double> vec(vecSize, 0.5);
    std::vector out(vec);

    RunAndMeasure("std::transform seq", [&vec, &out] {
        std::transform(std::execution::seq, vec.begin(), vec.end(), out.begin(),
        [](double v) {
            return std::sin(v)*std::cos(v);
```

```
        }
                        );
        return out.size();
    });

    RunAndMeasure("std::transform par", [&vec, &out] {
        std::transform(std::execution::par, vec.begin(), vec.end(), out.begin(),
        [](double v) {
            return std::sin(v)*std::cos(v);
        }
                        );
        return out.size();
    });

    return 0;
}
```

The code calculates `sin*cos` [^sincos] and stores the result in the output vector. Those trigonometry functions will keep CPU busy with arithmetic instructions, rather than just fetching an element from memory.

> *[^sincos]:* You can also use more optimal computation of `sin*cos` as `sin(2x) = 2 sin(x) cos(x)`.

The application was executed on two machines and three modes:

- i7 4720H VS - means Win 10 64bit, i7 4720H, 2.60 GHz base frequency, 4 Cores/8 Threads, MSVC 2017 15.8, Release mode, x86.
- i7 8700 VS- means Win 10 64bit, i7 8700, 3,2 GHz base frequency, 6 Cores/12 Threads, MSVC 2017 15.8, Release Mode, x86.
- i7 8700 GCC - means NixOS 19.03 64bit, i7 8700, 3,2 GHz base frequency, 6 Cores/12 Threads, GCC 9.1, Intel TBB

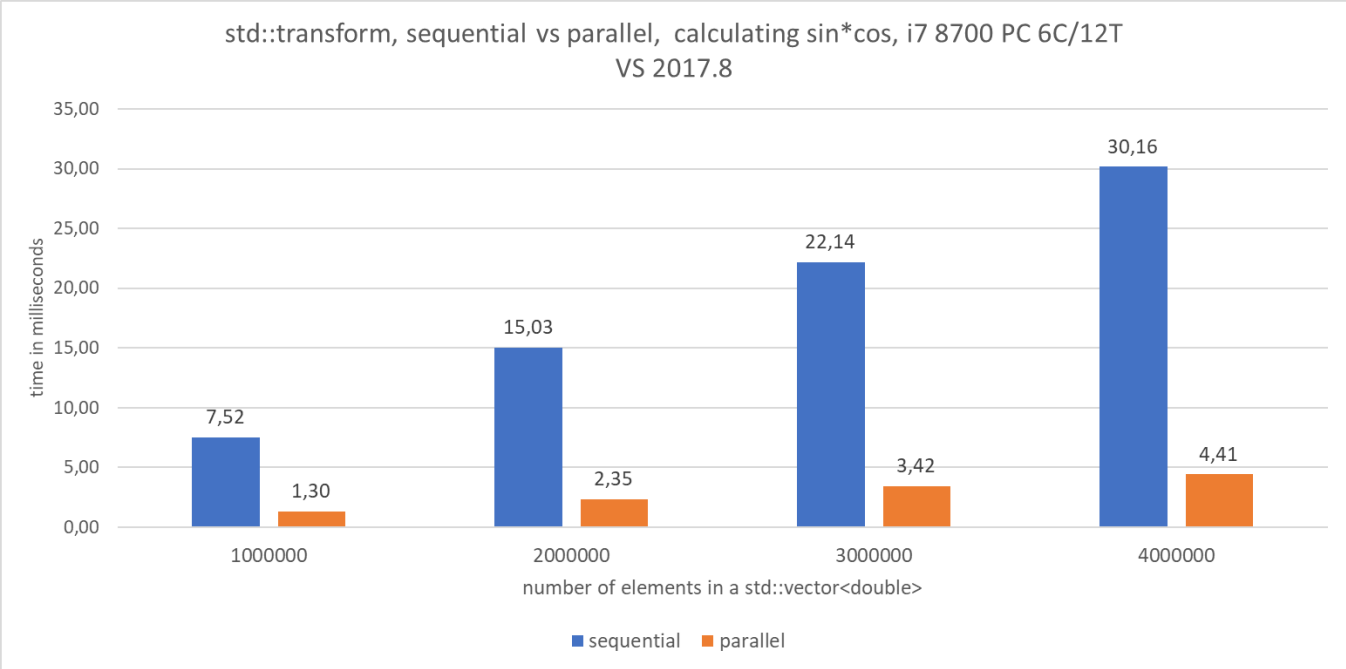
`RunAndMeasure` is a helper function that runs a function and then prints the timings. The result is used later so that the compiler doesn't optimise the variable away:

```
template <typename TFunc> void RunAndMeasure(const char* title, TFunc func) {
    const auto start = std::chrono::steady_clock::now();
    ret = func();
    const auto end = std::chrono::steady_clock::now();
    std::cout << title
```

```
        << ": " << std::chrono::duration<double, std::milli>(end - start).count()
        << " ms " << ret << '\n';
}
```

Here are the results (time in milliseconds):

algorithm	vector size	i7 4720H VS	i7 8700 VS	i7 8700 GCC
`std::transform, seq`	1000000	10.9347	7.51991	19.8189
`std::transform, par`	1000000	2.67921	1.30245	3.14286
`std::transform, seq`	2000000	21.8466	15.028	37.3226
`std::transform, par`	2000000	5.29644	2.34634	6.22417
`std::transform, seq`	3000000	32.7403	22.1449	55.8141
`std::transform, par`	3000000	7.79366	3.42295	9.34034
`std::transform, seq`	4000000	44.2565	30.1643	74.2437
`std::transform, par`	4000000	11.7558	4.40974	12.4206



The example above might be the perfect case for a parallelisation: we have an operation that requires a decent amount of instructions (trigonometry functions), and then all the tasks are separate. In this case, on a machine with 6 cores and 12 threads, the performance is almost 7X faster! On a computer with 4 cores and 8 threads, the performance is 4.2X faster.

GCC results are surprisingly slower than the Visual Studio version[^gccparres]. Still, we can also notice that with the parallel execution we get even 8x improvement on 6 cores/12 threads over the sequential execution.

> *[^gccparres]:* A similar machine was used, but the results were 2x slower. The full investigation is outside the scope of the book.

It's also worth to notice that when the transformation instructions are simple like `return v*2.0` then the performance speed-up might be not seen. This is because all the code will be just waiting on the global memory, and it might perform the same as the sequential version.

Next lesson will give us another example of benchmark value computation. Read on to find more!