Benchmark II

This section gives another example of benchmark measurement by computing the sum of all elements in a vector.

WE'LL COVER THE FOLLOWING

- Sum of All Elements in a Vector
  - Another reason to use sequential policy?

Sum of All Elements in a Vector

Below there's a benchmark of computing the sum of all elements in a vector:

> When comparing the parallel and serial execution time for the reduce function using Benchmark II, 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

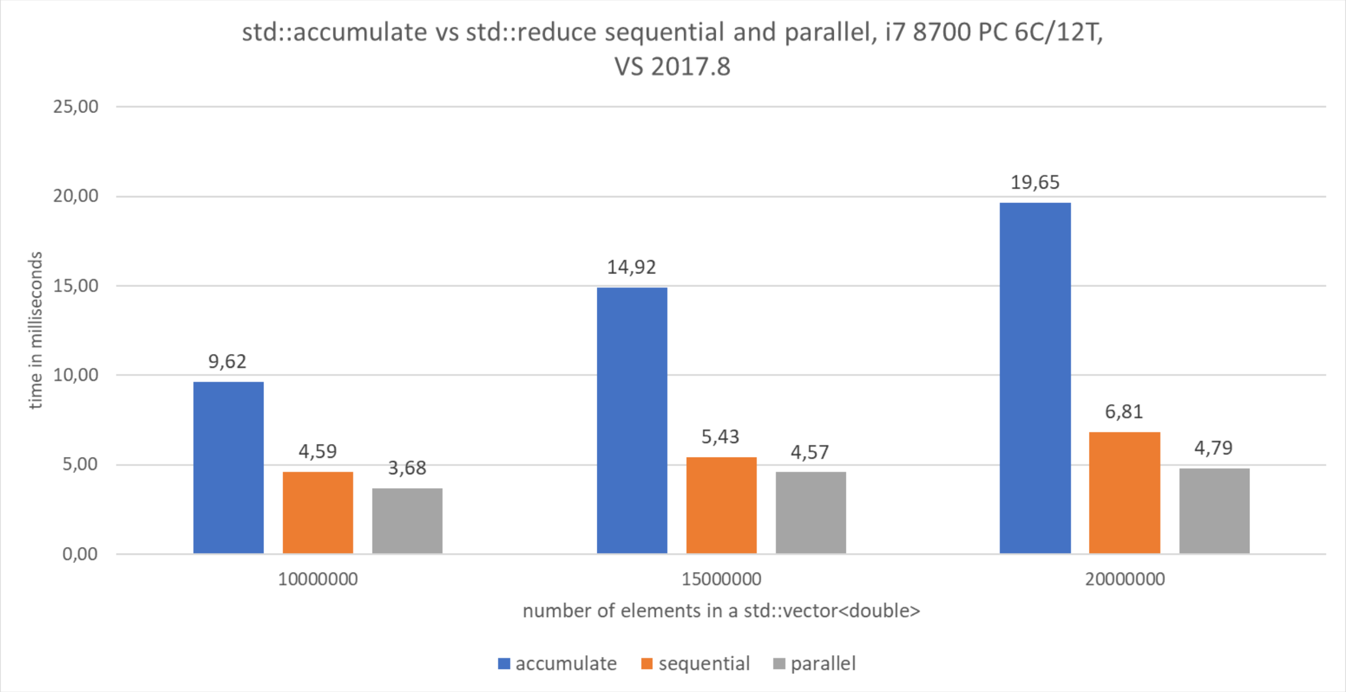
simpleperf.h

```
#include <algorithm>
#include <execution>
#include <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);

    RunAndMeasure("std::accumulate", [&vec] {
        return std::accumulate(vec.begin(), vec.end(), 0.0);
    });


    RunAndMeasure("std::reduce, seq", [&vec] {
            return std::reduce(std::execution::seq,
                vec.begin(), vec.end(), 0.0);
        }
    );

    RunAndMeasure("std::reduce, par", [&vec] {
            return std::reduce(std::execution::par,
                vec.begin(), vec.end(), 0.0);
        }
    );

    return 0;
}
```

Here are the results:

algorithm	size	i7 4720H VS	i7 8700 VS	i7 8700 GCC
`std::accumulate`	10000000	10.5814	9.62405	9.65569
`std::reduce seq`	10000000	6.9556	4.58746	9.20017
`std::reduce par`	10000000	4.88708	3.67831	2.45625
`std::accumulate`	15000000	17.8769	14.9163	14.2885
`std::reduce seq`	15000000	11.5103	5.42508	13.7725
`std::reduce par`	15000000	9.99877	4.5679	3.79334

`std::accumulate`	20000000	21.8888	19.6507	18.8786
`std::reduce seq`	20000000	16.2142	6.80581	18.4035
`std::reduce par`	20000000	10.8826	4.79214	5.141



During this execution, the `par` version was 2x…4x faster than the standard `std::accumulate`!

When looking at `par` and `accumulate`, this time, GCC results are almost the same as Visual Studio. It’s also clear that the GCC version switches to regular `std::accumulate` when you use sequential mode for `std::reduce`.

> **Another reason to use sequential policy?**
>
> In Visual Studio the sequential version of `std::reduce` was also faster than `std::accumulate`. This might happen because in `std::reduce` the order of operations is not determined, while `std::accumulate` is a left fold. The compiler has more options to optimise the code.

Let’s look at how to process several container simultaneously.