Parallel STL algorithms

Summary

- The STL algorithms, since C++17, provide an option to pass an `execution_policy`, defined in `<execution>` header, as an argument, which **allow** some changes in how the algorithms are executed (they do not guarantee, for instance, parallel execution):
  - `sequenced_policy` (`std::execution::seq`): execution may not be parallelized
  - `parallel_policy` (`std::execution::par`): execution may be parallelized
  - `parallel_unsequenced_policy` (`std::execution::par_unseq`): execution may be parallelized, element access functions may be invoked in any order, the execution may be vectorized
  - `unsequenced_policy` (`std::execution::unseq`, since C++20): execution may be vectorized, that is a single thread may use instructions operating on multiple elements
- The actual library implementations do not necessarily parallelize the execution:
  - GCC 10.2 with its default library implements this feature, but did not create any threads when tested.
    - Note: oneTBB may be used to change this behaviour, when used multiple threads are spawned for parallel execution policies.
  - Apple Clang 12 does not implement this feature (the `execution_policy` may not be passed to any algorithm).
  - MSVC (Visual Studio 2019) implements this feature and does indeed use multiple threads for parallel policies.

Usage

The full source code of the benchmark performed is attached in this repository as `bench.cpp`.

An example transformation of a container:

```
std::transform(std::begin(input), std::end(input),
               std::begin(output),
               [](const auto& element) { return element * 2; });
```

may be (usually) changed to a variant allowing parallel execution:

```
std::transform(std::execution::par,
               std::begin(input), std::end(input),
               std::begin(output),
               [](const auto& element) { return element * 2; });
```

Although most of the time it will *just work*, the iterators need to provide multipass guarantee. This prevents some undeterministic behaviour, as iterators such as `std::back_inserter` or `std::ostream_iterator` may not be used with this overload of algorithm. The output iterator should point to a structure which already has capacity for all the output data (an empty vector is not going to work).

Benchmark

A simple application executing `std::transform` using different execution policies has been created. It uses Celero benchmarking library. The results are expressed in iterations per second and in relation to a baseline of sequential execution policy.

Results (x86-64)

CPU: Intel i7-8650U (4 cores × 2 threads), OS: Windows 10, compiler: MSVC 2019, `/O2` option

Data size	x86 `seq`	x86 `par`	x86 `unseq`
32	354k (100%)	74.0k (**21%**)	234k (**66%**)
128	66.3k (100%)	33.2k (**50%**)	62.9k (**95%**)
512	18.7k (100%)	16.7k (**89%**)	15.3k (**82%**)
2048	4.70k (100%)	8.12k (**173%**)	4.75k (**101%**)
8192	1.10k (100%)	2.92k (**265%**)	1.12k (**102%**)
32768	277 (100%)	835 (**301%**)	276 (**100%**)
131072	65.5 (100%)	222 (**339%**)	68.3 (**104%**)

Results (ARM64)

CPU: Apple M1 (4 low power + 4 high performance cores), OS: macOS 11.2, compiler: GCC 10.2.1, `-O3` option

Data size	arm64 `seq`	arm64 `par`	arm64 `unseq`
32	1.81M (100%)	242k (**13%**)	1.82M (**101%**)
128	436k (100%)	86.9k (**20%**)	443k (**102%**)
512	96.5k (100%)	45.1k (**47%**)	100k (**104%**)
2048	17.7k (100%)	32.9k (**186%**)	20.4k (**115%**)
8192	4.20k (100%)	14.7k (**350%**)	4.08k (**97%**)
32768	993 (100%)	4.53k (**456%**)	985 (**99%**)
131072	242 (100%)	1.15k (**475%**)	243 (**100%**)

Results (ARM64, no compiler optimization)

CPU: Apple M1 (4 low power + 4 high performance cores), OS: macOS 11.2, compiler: GCC 10.2.1, `-O0` option

Data size	arm64 `seq`	arm64 `par`	arm64 `unseq`
32	628k (100%, 53% of `-O3`)	113k (**18%**, 47% of `-O3`)	1.14M (**182%**, 63% of `-O3`)
128	227k (100%, 52% of `-O3`)	61.4k (**27%**, 71% of `-O3`)	275k (**121%**, 62% of `-O3`)
512	59.5k (100%, 62% of `-O3`)	32.9k (**55%**, 73% of `-O3`)	64.5k (**108%**, 65% of `-O3`)
2048	14.9k (100%, 84% of `-O3`)	20.7k (**139%**, 63% of `-O3`)	16.1k (**108%**, 79% of `-O3`)
8192	3.72k (100%, 89% of `-O3`)	10.5k (**282%**, 72% of `-O3`)	3.98k (**107%**, 98% of `-O3`)
32768	923 (100%, 93% of `-O3`)	3.79k (**411%**, 84% of `-O3`)	1.01k (**109%**, 103% of `-O3`)
131072	231 (100%, 95% of `-O3`)	1.08k (**468%**, 94% of `-O3`)	252 (**109%**, 104% of `-O3`)

Observations

- The parallel execution becomes faster than sequential only with a data set which is big enough. Take note that the creation of threads has itself has a cost which may overcome the advantages of parallel execution.
- One can observe that without compiler optimizations enabled, execution with `seq` policy is much slower and the application benefits from `unseq` setting. Note how with the `unseq` policy the execution times with optimizations disabled can, with some workloads, slightly exceed those achieved with optimized code.
- This is, of course, a very simple benchmark and the possibilities of execution policies investigation are much broader. A source code with more algorithms and a simple benchmarking framework is attached.

Usage notes for `oneTBB`

Note: this was tested with GCC 10.2.1 on both openSUSE Tumbleweed (x86-64) and macOS 11.2 (ARM64).

- TBB development libraries are required, `zypper in tbb-devel`, `brew install tbb` or similar.
- The `tbb` library has to be used (`-ltbb` compiler option).
- For OSX GCC, the `tbb` include directory had to be specified manually (omitting this produced no errors, but the multithreaded implementation was not used), alhough this could be some configuration issue.