# STL Parallel

*C++17* added support for parallel algorithms to the standard library, to help programs take advantage of parallel execution for improved performance. MSVC first added experimental support for some algorithms in *15.5,* and the experimental tag was removed in *15.7*.

## How to use *Parallel Algorithms*

To use the parallel algorithms library, you can follow these steps:

- Find an algorithm call you wish to optimize with parallelism in your program. Good candidates are algorithms which do more than O(n) work like sort, and show up as taking reasonable amounts of time when profiling your application.
- Verify that code you supply to the algorithm is safe to parallelize.
- Choose a parallel execution policy. (Execution policies are described below.)
- If you aren't already, *#include <execution>* to make the parallel execution policies available.
- Add one of the execution policies as the first parameter to the algorithm call to parallelize.
- Benchmark the result to ensure the parallel version is an improvement. Parallelizing is not always faster, particularly for non-random-access iterators, or when the input size is small, or when the additional parallelism creates contention on external resources like a disk.

For the sake of example, here's a program we want to make faster. It times how long it takes to sort a million doubles.

```
#include <stddef.h>
#include <stdio.h>
#include <algorithm>
#include <chrono>
#include <random>
#include <ratio>
#include <vector>

using std::chrono::duration;
using std::chrono::duration_cast;
using std::chrono::high_resolution_clock;
using std::milli;
using std::random_device;
using std::sort;
using std::vector;

const size_t testSize = 1'000'000;
const int iterationCount = 5;

void print_results(const char *const tag, const vector<double>& sorted,
                high_resolution_clock::time_point startTime,
                high_resolution_clock::time_point endTime) {
  printf("%s: Lowest: %g Highest: %g Time: %fms\n", tag, sorted.front(),
         sorted.back(),
         duration_cast<duration<double, milli>>(endTime - startTime).count());
}

int main() {
  random_device rd;

  // generate some random doubles:
  printf("Testing with %zu doubles...\n", testSize);
  vector<double> doubles(testSize);
  for (auto& d : doubles) {
    d = static_cast<double>(rd());
  }

  // time how long it takes to sort them:
  for (int i = 0; i < iterationCount; ++i)
  {
    vector<double> sorted(doubles);
    const auto startTime = high_resolution_clock::now();
    sort(sorted.begin(), sorted.end());
    const auto endTime = high_resolution_clock::now();
    print_results("Serial", sorted, startTime, endTime);
  }
}
```

Note that the *Visual C++* implementation implements the parallel and parallel unsequenced policies the same way, so you should not expect better performance for using par_unseq on our implementation, but implementations may exist that can use that additional freedom someday. In the doubles sort example above, we can now add *#include <execution>* to the top of our program. Since we're using the parallel unsequenced policy, we add *std::execution::par_unseq* to the algorithm call site. (If we were using the parallel policy, we would use *std::execution::par instead*). With this change the for loop in main becomes the following:

```
for (int i = 0; i < iterationCount; ++i)
{
  vector<double> sorted(doubles);
  const auto startTime = high_resolution_clock::now();
  // same sort call as above, but with par_unseq:
  sort(std::execution::par_unseq, sorted.begin(), sorted.end());
  const auto endTime = high_resolution_clock::now();
  // in our output, note that these are the parallel results:
  print_results("Parallel", sorted, startTime, endTime);
}
```

The algorithms we parallelize in *Visual Studio 2017 15.8* are:

- adjacent_difference
- adjacent_find
- all_of
- any_of
- count
- count_if
- equal
- exclusive_scan
- find
- find_end
- find_first_of
- find_if
- for_each
- for_each_n
- inclusive_scan
- mismatch
- none_of
- partition
- reduce
- remove
- remove_if
- search
- search_n
- sort
- stable_sort
- transform
- transform_exclusive_scan
- transform_inclusive_scan
- transform_reduce

**C++17 offers the execution policy parameter that is available for most of the algorithms:**

- *sequenced_policy* - is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and require that a parallel algorithm's execution may not be parallelized. The corresponding global object is *std::execution::seq*
- *parallel_policy* - is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and indicate that a parallel algorithm's execution may be parallelized. The corresponding global object is *std::execution::par*
- *parallel_unsequenced_policy* - is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and indicate that a parallel algorithm's execution may be parallelized and vectorized. The corresponding global object is *std::execution::par_unseq*

In short:

- use *std::execution::seq* to execute your algorithm sequential
- use *std::execution::par* to execute your algorithm in parallel (usually using some Thread Pool implementation)
- use *std::execution::par_unseq* to execute your algorithm in parallel with also ability to use vector instructions (like SSE, AVX)

As a quick example you can invoke *std::sort* in a parallel way:

```
std::sort(std::execution::par, myVec.begin(), myVec.end());
    // ^^^^^^^^^^^^^^^^^^^
    // execution policy
```

**References**

1. https://devblogs.microsoft.com/cppblog/using-c17-parallel-algorithms-for-better-performance/
2. https://www.bfilipek.com/2018/11/parallel-alg-perf.html
3. https://dev.to/fenbf/examples-of-parallel-algorithms-from-c17-3jej