

Examples of Parallel Algorithms From C++17



Bartłomiej Filipek   Aug 27 '18 *Updated on Jul 24, 2019* • 9 min read

#programming #cpp #cpp17 #parallelism

MSVC (VS 2017 15.8, end of August 2018) is as far as I know the only major compiler/STL implementation that has parallel algorithms. Not everything is done, but you can use a lot of algorithms and apply `std::execution::par` on them!

Have a look at few examples I managed to run.

Introduction

Parallel algorithms look surprisingly simple from a user point of view. You have a new parameter - called **execution policy** - that you can pass to most

of the `std` algorithms :

```
std::algorithm_name(policy, /* normal args... */);
```

The general idea is that you call an algorithm and then you specify **how** it can be executed. Can it be parallel, maybe vectorized, or just serial.

We, as authors of the code, only know if there are any side effects, possible race conditions, deadlocks, or if there's no sense in running it parallel (like if you have a small collection of items).

Execution Policies

The execution policy parameter will tell the algorithm how it should be executed. We have the following options:

- **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`

New algorithms

A lot of existing algorithms were updated and overloaded with the execution policy: See the full list here:

[Extensions for parallelism - cppreference.com](https://en.cppreference.com/parallel)

And we got a few new algorithms:

- **for_each** - similar to `std::for_each` except returns `void`.
- **for_each_n** - applies a function object to the first `n` elements of a sequence.
- **reduce** - similar to `std::accumulate`, except out of order execution.

- `exclusive_scan` - similar to `std::partial_sum`, excludes the i-th input element from the i-th sum.
- `inclusive_scan` - similar to `std::partial_sum`, includes the i-th input element in the i-th sum
- `transform_reduce` - applies a functor, then reduces out of order
- `transform_exclusive_scan` - applies a functor, then calculates exclusive scan
- `transform_inclusive_scan` - applies a functor, then calculates inclusive scan

One of the most powerful algorithms is `reduce` (and its form of `transform_reduce`). Briefly, the new algorithm provides a parallel version of `std::accumulate`.

`Accumulate` returns the sum of all the elements in a range (or a result of a binary operation that can be different than just a sum).

```
std::vector<int> v{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
  
int sum = std::accumulate(v.begin(), v.end(), /*init*/0);
```

The algorithm is sequential only; a parallel version will try to compute the final sum using a tree approach (sum sub-ranges, then merge the results, divide and conquer). Such method can invoke the binary operation/sum in a *nondeterministic** order. Thus if `binary_op` is not associative or not commutative, the behaviour is also non-deterministic.

For example, you'll get the same results for `accumulate` and `reduce` for a vector of integers (when doing a sum), but you might get a slight difference for a vector of floats or doubles. That's because floating point operations are not associative.

`transform_reduce` will additionally invoke an operation on the input sequence and then perform reduction over the generated results.

MSVC Implementation

In the article: [Announcing: MSVC Conforms to the C++ Standard | Visual C++ Team Blog](#)

See the section **New Features: Parallel Algorithms:**

The following algorithms are parallelized.

- 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, reduce, remove, remove_if, search, search_n, sort, stable_sort, transform, transform_exclusive_scan, transform_inclusive_scan, transform_reduce

And we might expect more:

No apparent parallelism performance improvement on target hardware; all algorithms which merely copy or permute elements with no branches are typically memory bandwidth limited.

- copy, copy_backward, copy_n, fill, fill_n, move, move_backward, remove, remove_if, replace, replace_if, reverse, reverse_copy, rotate, rotate_copy, swap_ranges

Not yet evaluated; parallelism may be implemented in a future release and is suspected to be beneficial.

- copy_if, includes, inplace_merge, is_heap, is_heap_until, is_partitioned, is_sorted, is_sorted_until, lexicographical_compare, max_element, merge, min_element,

minmax_element, nth_element, partition_copy, remove_copy,
remove_copy_if, replace_copy, replace_copy_if, set_difference,
set_intersection, set_symmetric_difference, set_union,
stable_partition, unique, unique_copy

Anyway, a lot of new algorithms are done, so we can play with reduce ,
sorting, counting, finding and more.

Examples

All code can be found in my repo:

<https://github.com/fenbf/ParSTLTests>

I have three examples:

- a benchmark with a few algorithms
- computing the size of the directory
- counting words in a string

A Basic Example

A simple benchmark:

```
std::vector<double> v(6000000, 0.5);

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

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

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

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

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

RunAndMeasure("std::find, seq", [&v] {
    auto res = std::find(std::execution::seq, std::begin(v), std::end(v), 0.6);
    return res == std::end(v) ? 0.0 : 1.0;
});
```



```
RunAndMeasure("std::find, par", [&v] {
    auto res = std::find(std::execution::par, std::begin(v), std::end(v), 0.6);
    return res == std::end(v) ? 0.0 : 1.0;
});
```

`RunAndMeasure` is a helper function that runs a function and then prints the timings. Also, we need to make sure the result is not optimized away.

```
template <typename TFunc> void RunAndMeasure(const char* title, TFunc func)
{
    const auto start = std::chrono::steady_clock::now();
    auto ret = func();
    const auto end = std::chrono::steady_clock::now();
    std::cout << title << ": " <<
        std::chrono::duration <double, std::milli>(end - start).count()
        << " ms, res " << ret << "\n";
}
```

On My machine (Win 10, i7 4720H, 4Cores/8Threads) I get the following results (in Release mode, x86)

```
std::warm up: 4.35417 ms, res 3e+06  
std::accumulate: 6.14874 ms, res 3e+06  
std::reduce, seq: 4.07034 ms, res 3e+06  
std::reduce, par: 3.22714 ms, res 3e+06  
std::reduce, par_unseq: 3.0495 ms, res 3e+06  
std::find, seq: 5.13658 ms, res 0  
std::find, par: 3.20385 ms, res 0
```

As you can see there's some speed up!

Computing File Sizes

The below example is based on a code sample from [C++17 - The Complete... by Nicolai Josutti](#).

Parallel algorithms - `std::reduce` is used to compute sizes of the files in a directory (using recursive scan). It's a nice example of two C++17 features: parallelism and `std::filesystem`.

Here are the interesting parts of the code:

```
// Get all the available paths, recursively:
std::vector<std::filesystem::path> paths;
try {
    std::filesystem::recursive_directory_iterator dirpos{ root };
    std::copy(begin(dirpos), end(dirpos),
              std::back_inserter(paths));
}
catch (const std::exception& e) {
    std::cerr << "EXCEPTION: " << e.what() << std::endl;
    return EXIT_FAILURE;
}
```

Fetching all the paths is handled by so concise code!

For now `std::copy` cannot be used in a parallel way.

And the final computations:

```
template <typename Policy>
uintmax_t ComputeTotalFileSize(const std::vector<std::filesystem::path>& paths,
                               Policy policy)
{
    return std::transform_reduce(
        policy,
```

```

        paths.cbegin(), paths.cend(),           // range
        std::uintmax_t{ 0 },                   // initial value
        std::plus<>(),                          // accumulate ...
        [] (const std::filesystem::path& p) {   // file size if regular file
            return is_regular_file(p) ? file_size(p)
                : std::uintmax_t{ 0 };
        });
    }

```

The main invocation:

```

start = std::chrono::steady_clock::now();
uintmax_t FinalSize = 0;
if (executionPolicyMode)
    FinalSize = ComputeTotalFileSize(paths, std::execution::par);
else
    FinalSize = ComputeTotalFileSize(paths, std::execution::seq);

PrintTiming("computing the sizes", start);

std::cout << "size of all " << paths.size()
            << " regular files: " << FinalSize/1024 << " kbytes\n";

```

The "problem" I found is that the `par` and `seq` policies are not of the same type. That's why I moved the code into a template function and then I could control it via the boolean flag.

Some results (running on the intermediate directory from the builds, 108 files, ~20MB total):

```
// parallel:
PS D:\github\ParSTLTests\Release> .\FileSizes.exe ..\IntDir\ 1
Using PAR Policy
gathering all the paths: 0.74767 ms
number of files: 108
computing the sizes: 0.655692 ms
size of all 108 regular files: 20543 kbytes

// sequential:
PS D:\github\ParSTLTests\Release> .\FileSizes.exe ..\IntDir\ 0
Using SEQ Policy
gathering all the paths: 0.697142 ms
number of files: 108
computing the sizes: 1.0994 ms
size of all 108 regular files: 20543 kbytes
```

For this test, I got 1.0994 ms vs 0.655692 ms - in favour of the PAR version.

Counting Words in a String

The below example comes from Bryce talk about parallel algorithms.

<https://www.youtube.com/watch?v=Vck6kzWjY88&t=916s>

He showed an interesting way of computing the word count:

- In the **first phase** we transform text into 1 and 0. We want to have 1 in the place where a word starts and 0 in all other places.
 - If we have a string "One Two Three" then we want to generate an array 1000100010000.
- Then we can reduce the computed array of 1 and 0 - the generated sum is the number of words in a string.

This looks like a "natural" example where `transform_reduce` might be used:

```
bool is_word_beginning(char left, char right)
{
    return std::isspace(left) && !std::isspace(right);
}
```

```
template <typename Policy>
std::size_t word_count(std::string_view s, Policy policy)
{
    if (s.empty())
        return 0;

    std::size_t wc = (!std::isspace(s.front()) ? 1 : 0);
    wc += std::transform_reduce(policy,
        s.begin(),
        s.end() - 1,
        s.begin() + 1,
        std::size_t(0),
        std::plus<std::size_t>(),
        is_word_beginning);

    return wc;
}
```

Here's a benchmark code:

```
const int COUNT = argc > 1 ? atoi(argv[1]) : 1'000'000;
std::string str(COUNT, 'a');

for (int i = 0; i < COUNT; ++i)
{
```

```
        if (i % 5 == 0 || i % 17 == 0)
            str[i] = ' '; // add a space
    }

    std::cout << "string length: " << COUNT << ", first 60 letters: \n";
    std::cout << str.substr(0, 60) << std::endl;

    RunAndMeasure("word_count seq", [&str] {
        return word_count(str, std::execution::seq);
    });

    RunAndMeasure("word_count par", [&str] {
        return word_count(str, std::execution::par);
    });

    RunAndMeasure("word_count par_unseq", [&str] {
        return word_count(str, std::execution::par_unseq);
    });
```

And some results:

```
PS D:\github\ParSTLTests\Release> .\WordCount.exe
string length: 1000000, first 60 letters:
    aaaa aaaa aaaa a aa aaaa aaaa aaa  aaaa aaaa aaaa  aaa aaaa
word_count seq: 3.44228 ms, res 223529
word_count par: 1.46652 ms, res 223529
```



```
word_count par_unseq: 1.26599 ms, res 223529
```

```
PS D:\github\ParSTLTests\Release> .\WordCount.exe 20000000
```

```
string length: 20000000, first 60 letters:
```

```
aaaa aaaa aaaa a aa aaaa aaaa aaa aaaa aaaa aaaa aaa aaaa
```

```
word_count seq: 69.1271 ms, res 4470588
```

```
word_count par: 23.342 ms, res 4470588
```

```
word_count par_unseq: 23.0487 ms, res 4470588
```

```
PS D:\github\ParSTLTests\Release> .\WordCount.exe 50000000
```

```
string length: 50000000, first 60 letters:
```

```
aaaa aaaa aaaa a aa aaaa aaaa aaa aaaa aaaa aaaa aaa aaaa
```

```
word_count seq: 170.858 ms, res 11176471
```

```
word_count par: 59.7102 ms, res 11176471
```

```
word_count par_unseq: 62.2734 ms, res 11176471
```

The parallel version is sometimes almost 3x faster! And there are even differences for `par_unseq`.

Summary

I hope you see some potential in the parallel versions of the algorithms. Probably it's not the last word from the MSVC implementation, so maybe we can expect more algorithms and perf boost in the future.

Here's the link to the proposal of Parallel Algorithms: [P0024R2](#)

It would be great if other STL implementations catch up:

- [LLVM libc++ C++1Z Status](#) - so far all of the items for parallelism are not done yet.
- [GNU libstdc++ C++17 status](#) - not implemented yet

And there are also other implementations, from third party vendors:

- Codeplay: [SyclParallelSTL](#)
- [HPX](#)
- [Parallel STL](#)
- [Intel](#)

It might be interesting to see if MSVC implementation is faster or slower compared to the third party implementations.

Call to action

If you work with Visual Studio, you can copy the examples from the article (or go to my [GitHub](#) and download the solution) and report the results that

you got. I wonder what's the average speed up that we currently have with the MSVC implementation.




More from the Author

Bartek recently published a book - "[C++17 In Detail](#)" - rather than reading the papers and C++ specification drafts, you can use this book to learn the new Standard in an efficient and practical way.



Bartlomiej Filipek + FOLLOW

Software developer with a blog about C++ and stories about native programming. Author of C++17 In Detail - <https://leanpub.com/cpp17indetail>

@fenbf  fenbf  fenbf  www.bfilipek.com

Add to the discussion



PREVIEW

SUBMIT

[code of conduct](#) - [report abuse](#)

Classic DEV Post from Mar 24

whats the best source of website templates?



FultonB

I'm looking for some website templates for a basic product page, whats the best source of website tem...



67



15

Another Post You Might Like

Node.js Under The Hood #1 - Getting to know our tools



Lucas Santos

I was recently called to speak at a huge Brazilian conference called The Conf. The whole point of...



963



23

Another Post You Might Like

Getting Cozy With C++



Ben Lovy

Swallowing The Pill Some Background C++ is my self-learning white whale. I've...



208



25

Creating a JavaScript Function to Calculate Whether It's a Leap Year

Nick Scialli (he/him) - Apr 24

Daily HackerRank Challenge - Day 13

Wing-Kam - Apr 25

Async in Dart (1) รู้จัก Isolates และ Event Loop กับการทำงานไม่ประสานเวลา

Ta - Apr 25

Want To Write The Best Code You Possibly Can? Read this

iamdi - Apr 24

[Home](#) [About](#) [Privacy Policy](#) [Terms of Use](#) [Contact](#) [Code of Conduct](#)

DEV Community copyright 2016 - 2020 