Bartek's coding blog

C++ and native programming stories.

12 November 2018

The Amazing Performance of C++17 Parallel Algorithms, is it Possible?

35
Shares

The Amazing Performance of
Parallel Algorithms
from C++17
std::transform(std::execution::par, ...);
Is It Possible?

th the addition of Parallel Algorithms in C++17, you can now easily update your "computing" code to benefit from parallel execution. In the article, I'd like to examine one STL algorithm which naturally exposes the idea of independent computing. If your machine has 10-core CPU, can you always expect to get 10x speed up? Maybe more? Maybe less? Let's play with this topic.

Table of Contents

Update 13th Nov: I've applied the comments from r/cpp discussions, used proper ranges for trigonometry/sqrt computations, and some minor changes. The benchmarks were executed another time.

Intro to Parallel Algorithms

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`

35
Shares

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
        // ^^^^^^^^^^^^^^^^^^^
        // execution policy
```

Please notice that it's so easy just to add parallel execution parameter to an algorithm! But can you always experience a huge performance boost? Is it always faster? Or maybe there are cases where it might slow things down?

Parallel `std::transform`

In this post I'd like to have a look at `std::transform` algorithm that potentially might be one of the building blocks of other parallel techniques (along with `std::transform_reduce`, `for_each`, `scan`, `sort`…).

Our test code will revolve around the following pattern.

35
Shares

```
d::transform(execution_policy, // par, s
             inVec.begin(), inVec.end(),
             outVec.begin(),
             ElementOperation);
```

uming the `ElementOperation` function doesn't use / method of synchronisation, then the code might /e a good potential to be executed in parallel or even vectorised. Each computation for an element is independent, the order is not important, so the implementation might spawn multiple threads (possibly on a thread pool) to process elements independently.

I'd like to experiment with the following cases.

- size of the vector - big or small
- simple transformations that spend time mostly on memory access
- more arithmetic (ALU) operations
- ALU in a more realistic scenario

As you can see, I'd like not only to test the number of elements that is "good" to use a parallel algorithm, but also ALU operations that keep CPU busy.

Other algorithms like sorting, accumulate (in the form of `std::reduce`) also offers parallel execution, but they require more work (and usually merging steps) to compute the results. So they might be candidates for another article.

Note on Benchmarks

I'm using Visual Studio 2017, 15.8 for my tests - as it's the only implementation in a popular compiler/STL implementation at the moment (Nov 2018) (GCC on the way!). What's more, I focused only on `execution::par` as `execution::par_unseq` is not available in MSVC (works the same way as `execution::par`).

I have two machines:

- i7 8700 - PC, Windows 10, i7 8700 - clocked at 3.2 GHz, 6 cores/12 threads (Hyperthreading)
- i7 4720 - Notebook, Windows 10, i7 4720, clocked at 2.6GHz, 4 cores/8 threads (Hyperthreading)

the code is compiled in x64, Release more, auto vectorisation is enabled by default, and I've enabled

35
Shares

anced instruction set (SSE2), as well as OpenMP (2.0)

e code is located on my github: hub/fenbf/ParSTLTests/TransformTests/TransformT s.cpp

OpenMP (2.0) I'm only using parallel for loops:

```
ragma omp parallel for
for (int i = 0; ...)
```

I'm running the code section 5 times, and I look at the min numbers.

Warning: The results are shown only to present some rough observations, and please run it on your system/configuration before using it in production. Your requirements and environment might be different than mine.

You can read more about MSVC implementation in this post:
Using C++17 Parallel Algorithms for Better Performance | Visual C++ Team Blog

And here's a recent Billy O'Neil's talk at CppCon 2018 (Billy implemented Parallel STL in MSVC):
https://www.youtube.com/watch?v=nOpwhTbulmk

OK, let's start with some basic examples!

Simple Transformation

Consider a case where you apply a really simple operation on the input vector. It might be a copy or a multiplication of elements.

For example:

```
std::transform(std::execution::par,
               vec.begin(), vec.end(), out
               [](double v) { return v * 2
);
```

My machine has 6 or 4 cores… can I expect to get 4…6x perf of a sequential execution?

Here are the results (time in milliseconds):

35
Shares

Operation	Vector Size	i7 4720 (4 Cores)	i7 8700 (6 Cores)
execution::seq	10k	0.002763	0.001924
xecution::par	10k	0.009869	0.008983
openmp arallel for	10k	0.003158	0.002246
xecution::seq	100k	0.051318	0.028872
execution::par	100k	0.043028	0.025664
openmp parallel for	100k	0.022501	0.009624
execution::seq	1000k	1.69508	0.52419
execution::par	1000k	1.65561	0.359619
openmp parallel for	1000k	1.50678	0.344863

As you see on the faster machine, you need like 1 million elements to start seeing some performance gains. On the other hand on my notebook, all parallel implementations were slower.

All in all, as might guess there's a weak chance we'll any considerable speed-up using such transformations, even when we increase the number of elements.

Why is that?

Since the operations are elementary, CPU cores can invoke it almost immediately, using only a few cycles. However, CPU cores spend more time waiting for the main memory. So, in that case, they all be mostly waiting, not computing.

> *Reading or writing to a variable in memory takes only 2-3 clock cycles if it is cached, but several hundred clock cycles if it is not cached https://www.agner.org/optimize/optimizing_cpp.pdf*

We can give a rough observation that if your algorithm is memory bound, then you cannot expect to have a better performance with the parallel execution.

More Computations

35
Shares

ce memory throughput is essential and might slow ngs down… let's increase the number of computations ıt affect each element.

ɘ idea is that it's better to use CPU cycles rather than ɜnd time waiting for memory.

· a start, I'll use trigonometry functions, for example, _rt(sin*cos) (those are arbitrary computations, not optimal form, just to keep CPU busy).

We're using `sqrt`, `sin` and `cos` which might take up ~20 per sqrt, ~100 per a trigonometry function. That amount of computation might cover the latency on the memory access.

More about instruction latencies in this great Perf Guide from Agner Fog

Here's the benchmark code:

```
std::transform(std::execution::par, vec.be
            [](double v) {
                return std::sqrt(std::sin(
            }
);
```

How about now? Can we get some better perf than our previous attempt?

Here are the results (time in milliseconds):

Operation	Vector Size	i7 4720 (4 Cores)	i7 8700 (6 Cores)

Operation	Vector Size	i7 4720 (4 Cores)	i7 8700 (6 Cores)
`execution::seq`	10k	0.105005	0.070577
`execution::par`	10k	0.055661	0.03176
`openmp parallel for`	10k	0.096321	0.024702
`execution::seq`	100k	1.08755	0.707048
`execution::par`	100k	0.259354	0.17195
`openmp arallel for`	100k	0.898465	0.189915
`xecution::seq`	1000k	10.5159	7.16254
`xecution::par`	1000k	2.44472	1.10099
`openmp arallel for`	1000k	4.78681	1.89017

Now, we’re finally seeing some nice numbers :)

For 1000 elements (not shown here), the timings for parallel and sequential were similar, so above 1000 elements, we can see some improvements for the parallel version.

For 100k elements, the faster machine performs almost 9x faster than the sequential version (similarly for OpenMP version).

For the largest set of a million elements - it’s 5x or 8x faster.

For such computations, I could achieve the speed-up that is “linear” to my CPU core count. Which is probably what we should expect.

Fresnel and 3D Vectors

In the section above I’ve used some “imaginary” computations, but how about some real code?

Let’s compute Fresnel equations that describe reflection and refraction of light at uniform planar interfaces. It’s a popular technique for generating realistic lightning in 3D games.

$$F_{R\parallel} = \left(\frac{\eta_2 \cos\theta_1 - \eta_1 \cos\theta_2}{\eta_2 \cos\theta_1 + \eta_1 \cos\theta_2}\right)^2,$$

$$F_{R\perp} = \left(\frac{\eta_1 \cos\theta_2 - \eta_2 \cos\theta_1}{\eta_1 \cos\theta_2 + \eta_2 \cos\theta_1}\right)^2.$$

$$F_R = \frac{1}{2}(F_{R\parallel} + F_{R\perp}).$$

35
Shares

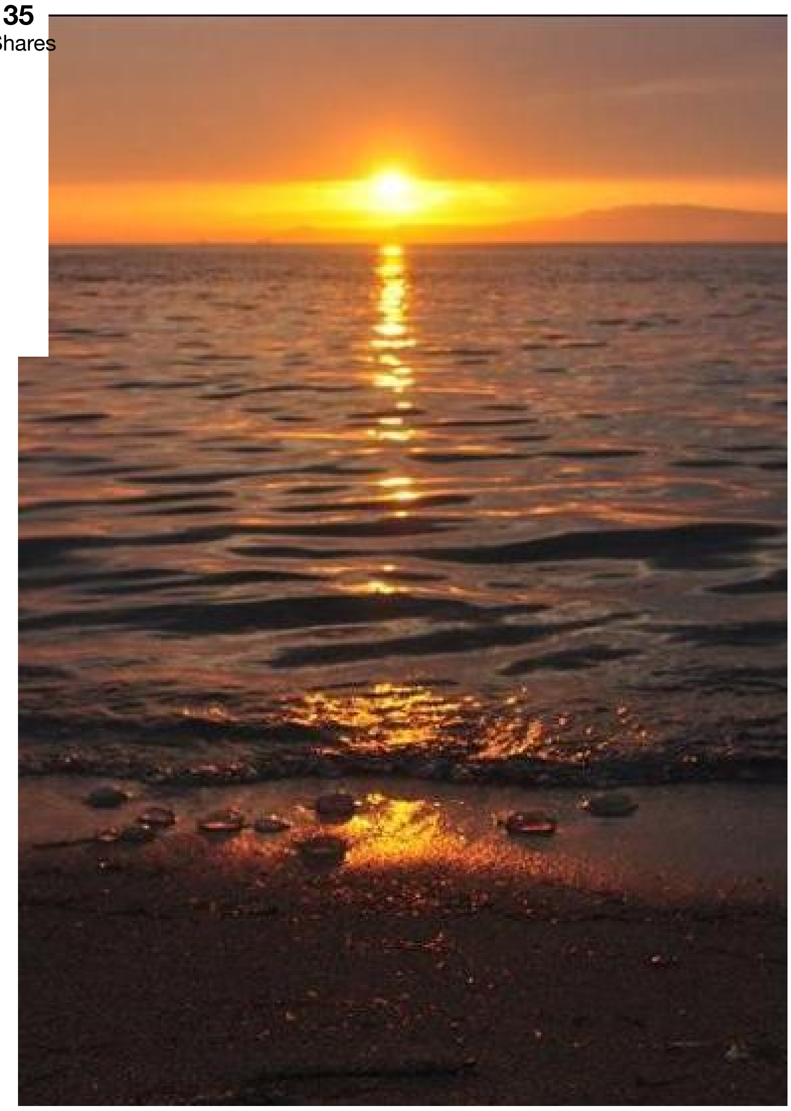


Photo from Wikimedia

As a good reference I've found this great description and the implementation:
Introduction to Shading (Reflection, Refraction and Fresnel) @scratchapixel.com

About Using GLM library

Rather than creating my own implementation, I've used the `glm` library. I've used it a lot in my OpenGL projects.

The library is available easily through Conan Package Manager, so I'll be using that as well:

The link to the package: https://bintray.com/bincrafters/public-conan/glm%3Ag-truc

Conan file:

```
[requires]
glm/0.9.9.1@g-truc/stable

35 ;enerators]
Shares .sual_studio
```

ı the command line to install the library (it will ıerate props file that I can use with my Visual Studio ›ject)

```
›nan install . -s build_type=Release -if I
```

The library is header only, so it's also easy to download it manually if you prefer.

The actual code & benchmark

I've adapted the code for `glm` from scratchapixel.com:

```
// implementation adapted from https://www
float fresnel(const glm::vec4 &I, const glı
{
    float cosi = std::clamp(glm::dot(I, N)
    float etai = 1, etat = ior;
    if (cosi > 0) { std::swap(etai, etat);

    // Compute sini using Snell's law
    float sint = etai / etat * sqrtf(std::ı
    // Total internal reflection
    if (sint >= 1)
        return 1.0f;

    float cost = sqrtf(std::max(0.f, 1 - s.
    cosi = fabsf(cosi);
    float Rs = ((etat * cosi) - (etai * co:
               ((etat * cosi) + (etai * co:
    float Rp = ((etai * cosi) - (etat * co:
               ((etai * cosi) + (etat * co:
```

```
    return (Rs * Rs + Rp * Rp) / 2.0f;
}
```

The code uses a few maths instructions, dot product, multiplications, divisions, so that should keep CPU busy as well. Rather than a vector of doubles we're also using 4-element vectors, so the memory used has also increased.

The benchmark:

```
std::transform(std::execution::par,
               vec.begin(), vec.end(), vec
               vecFresnelTerms.begin(),
               [](const glm::vec4& v, cons
                  return fresnel(v, n, 1.
               }
;
```

35
Shares

re are the results (time in milliseconds):

Operation	Vector Size	i7 4720 (4 Cores)	i7 8700 (6 Cores)
xecution::seq	1k	0.032764	0.016361
execution::par	1k	0.031186	0.028551
openmp parallel for	1k	0.005526	0.007699
execution::seq	10k	0.246722	0.169383
execution::par	10k	0.090794	0.067048
openmp parallel for	10k	0.049739	0.029835
execution::seq	100k	2.49722	1.69768
execution::par	100k	0.530157	0.283268
openmp parallel for	100k	0.495024	0.291609
execution::seq	1000k	25.0828	16.9457
execution::par	1000k	5.15235	2.33768

Operation	Vector Size	i7 4720 (4 Cores)	i7 8700 (6 Cores)
`openmp parallel for`	1000k	5.11801	2.95908

With the "real" computations we can see that parallel algorithms offer good performance. On my two Windows machines, for such operations, I could get speed-up that is almost linear to the number of cores.

For all tests I also showed you result from OpenMP and both implementations: MSVC and OpenMP seem to perform similarly.

Summary

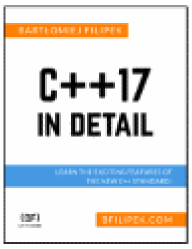
35
Shares

the article, I've shown three cases where you can rt using parallel execution and parallel algorithms. ile replacing all standard algorithms with just their `d::execution::par` version might be tempting, it's : always a good way to do that! Each operation that ı use inside an algorithm might perform differently ı be more CPU or Memory bound, and that's why you nave to consider each change separately.

Things to remember

- parallel execution will, in general, do more work than the sequential version, it's because the library has to prepare the parallel execution
- it's not only the count of elements that is important but also the number of instructions that keeps CPU busy
- it's best to have tasks that don't depend on each other nor other shared resources
- parallel algorithms offers a straightforward way to spawn work into separate threads
- if your operations are memory bound that you cannot expect much performance increase, or in some cases, the algorithm might be slower
- to get decent performance increase always measure the timings for each problem, as in some cases the results might be completely different

Special Thanks to JFT for help with the article!



See my Book!

Recently Published

C++ Tricks: IIFE for Complex Variable Initialization

Popular Posts:

C++ Ecosystem: Compilers, IDEs, Tools, Testing ar
C++ Tricks: IIFE for Complex Variable Initializatio
Using C++17 std::optional
Everything You Need to Know About std::variant f
How to Iterate Through Directories in C++
Lambdas: From C++11 to C++20, Part 1
17 Smaller but Handy C++17 Features
How To Use Vocabulary Types from C++17, Preser
The Pimpl Pattern - what you should know
Custom Deleters for C++ Smart Pointers

Custom Search

Cpp Cpp17 links experiments STL performance Cpp11 books tips Visual Studio OpenGL libraries Cpp14 Cpp20 SIMD refactoring templates algorithms filesystem code style optimization concurrency lambda

For more references you can also have a look at my other resources about parallel algorithms:

- Fresh chapter in my C++17 In Detail Book about Parallel Algorithms.
- Parallel STL And Filesystem: Files Word Count Example
- Examples of Parallel Algorithms From C++17

Have a look at another article related to Parallel Algorithms: How to Boost Performance with Intel Parallel STL and C++17 Parallel Algorithms

Your Turn

What's the answer to my question from the title? Can 35 Shares get the amazing performance from parallel orithms?

ve you played with the parallel execution? Did it bring : expected speed up?

the article I've only touched "simple" parallel orithms - `std::transform`. Things get even more nplicated when we talk about `std::reduce`.

7 Comments Bartek's coding blog

Recommend Tweet Share

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Benjamin Navarro • a year ago

Thanks for the nice article! Any idea why OpenMP
better than the std parallel algorithms?

4 ^ | v • Reply • Share ›

Bartlomiej Filipek Mod → Benjamin Navarro • a ye

hello, thanks for the comment! Not sure ab
implementation of OpenMP, so I cannot giv
answer here. I'll try to investigate..

^ | v • Reply • Share ›

35
Shares

Stefan Atev → Bartlomiej Filipek • a year a

OpenMP is optimized for finer-grain
has less launch overhead than MSV
algorithms, so it will be competitive
Also, the way it's used here probabl
some OpenMP code - schedule(sta
chunk_size) should be used to optin
For example, on memory-bound stu
chunk size large enough that you dc
(so try to have each thread process
- 4KB worth of data); you definitely
adjacent threads reading the same
worse, writing to it... if you have loa
want a dynamic schedule (that has
etc.

Adding schedule(static, 512) to the
may show more difference (wild cor
to test on my machine...)

Anyhow - keep the articles coming!

1 ^ | v • Reply • Share ›

Bartlomiej Filipek Mod → Stefar
• a year ago

Thanks for the explanation! I
openMP so that I have some
I could see how the new thin
something that's available fo

Billy O'Neil also wrote e nice

differences between openMP
r/cpp:

https://www.reddit.com/r/cp

Reply • Share ›

Dawid Drozd • a year ago

Did you consider: https://github.com/cpp-task... ?

Reply • Share ›

Bartlomiej Filipek Mod Dawid Drozd • a year ag

no, but it looks interesting!

Newer Post Home Older Post

35
Shares