Evaluating C++17 parallel STL on CPUs and GPUs

Tutored Research Project I. Report

Domonkos Péter Ábrahám

Neptun ID: BZMKMC

Supervisor: István Zoltán Reguly, Ph.D.

Pázmány Péter Catholic University

Faculty of Information Technology and Bionics

Budapest

December 17, 2020

Table of contents

Objectives

Parallelism is a really good technik to accelerate program execution. Historically, if you wanted to apply parallelism in your c++ code, you had to use several language extensions or you had to add external libraries. So you could not use just the c++ standard library. But the good news is, that from c++17 the standard became extended with parallel support for the STL algorithms [1]. These algorithms can use the several execution policies (implemented in the standard execution header) as a parameter, and so they can be parallelized.

But the question is if it is so good as promised? Is it as fast as the state of the art? How does it scale up with the data size?

To get an answer for this question my aim was to evaluate the performance of the parallel STL both on processors (CPUs) and on video cards (GPUs).

Parallel STL

The c++ standard library is always changing. It is always trying to be more easy-to-use and adapting to the new challenges. Nowadays one of the big challenges is to make the code execution parallel. Until 2017 the c++ STL algorithms have had no support for parallel execution. Not surprising, that it was forced to solve this problem and so in c++17 standard library release it became solved.[1]

The solution is that you can invoke the stl algorithms with an execution policy. The policy will be a parameter of the algorithm function.[2] For instance, in case of `std::copy` and `std::execution::par` it works like this:

```
std::copy(std::execution::par, a.begin(), a.end(), b.begin())
```

where *a* and *b* are STL containers.

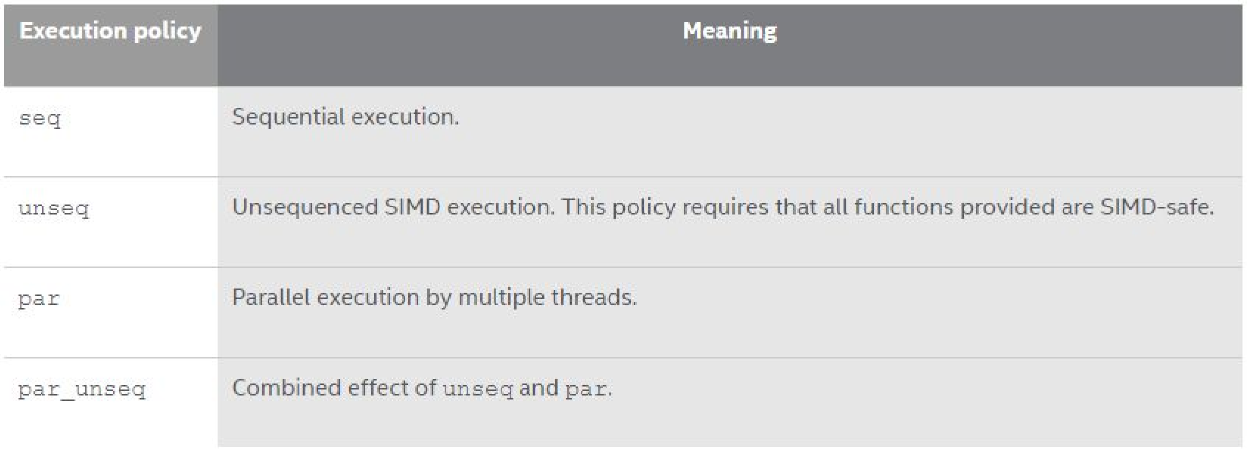
The execution policies

The execution policies - as we can deduce from the name - determine with which method should be the particular algorithm executed.

There are four execution policy types [3]:

- sequenced_policy
- unsequenced_policy
- parallel_policy
- parallel_unsequenced_policy

More commonly we use the instances of these, namely:



Execution policy	Meaning
`seq`	Sequential execution.
`unseq`	Unsequenced SIMD execution. This policy requires that all functions provided are SIMD-safe.
`par`	Parallel execution by multiple threads.
`par_unseq`	Combined effect of `unseq` and `par`.

Execution policies [2]

My research

Compilation and running

For compilation on CPU we used the Intel® C++ Compiler 19.1 for Linux [4]. This is a compiler, distributed by Intel. “This compiler produces optimized code that can run significantly faster by taking advantage of the ever-increasing core count and vector register width in Intel® Xeon® processors and compatible processors.” [2] So it was the best choice to be able to measure state of the art running times.

For compilation on GPU we used nvc++ with -stdpar flag. [5]

As a hardware we used the High Performance Cluster of my university (PPCU FITB) . It is a server with

- *Intel® Xeon® Processor E5-2697 v3* CPU

    - https://ark.intel.com/content/www/us/en/ark/products/81059/intel-xeon-processor-e5-2697-v3-35m-cache-2-60-ghz.html
- *NVIDIA Tesla V100 PCIe* GPU inside.
    - http://images.nvidia.com/content/technologies/volta/pdf/tesla-volta-v100-datasheet-letter-fnl-web.pdf

Time measuring

For the research our aim was to evaluate the features and performance of the parallel STL. It is not too concrete yet. So we had to make our experiment more specific. So we decided to measure time to get to know how fast the algorithm is. For this we used the std::chrono library [6].

It was a really good choice, because with this tool we can measure time really precisely. Even elapsed nanoseconds can be counted by this. It was needed, because we wanted to measure in the really small ranges of array size also and in this region of course the running time is really short.

Range of the measurements

In the measurement it was important to cover a large range of arraysize, in order to have a wide picture about the performance of the parallel STL library. We decided to measure on the [1, 2^{30}] range of array sizes. We used it as a logarithmic range. So we measured on the powers on two.

Calculating the bandwidth

Only the raw measured times are not really interpretable yet. We calculated the bandwidth for every measured time. (transferred data / elapsed time)

We used arrays with integers. One integer occupies four bytes in the memory. We used this information for the bandwidth calculation.

Implementation

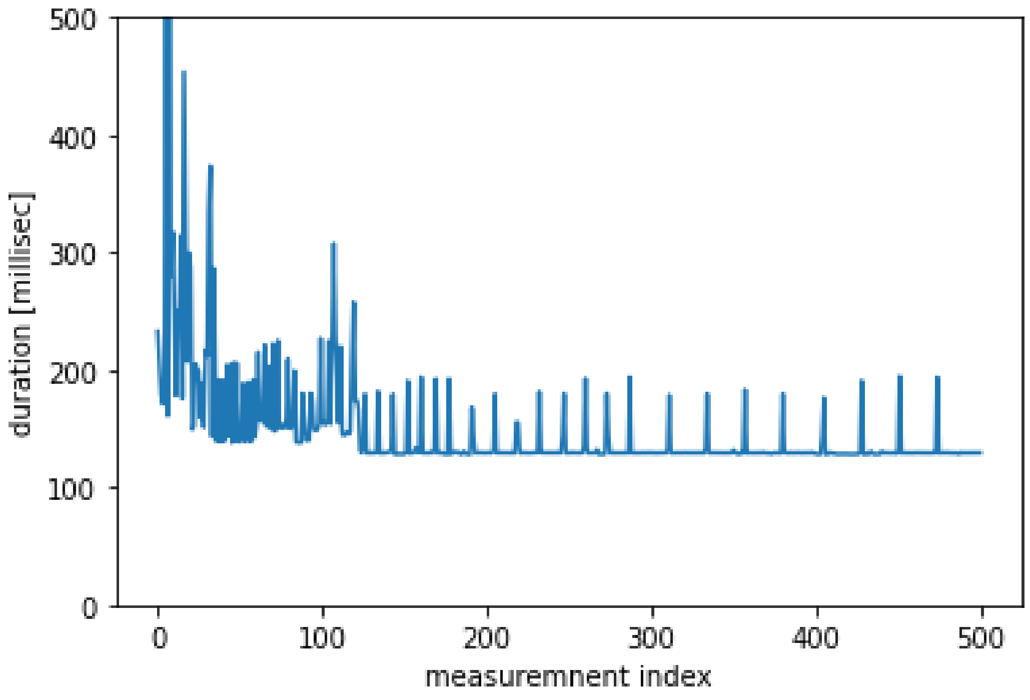
During the implementation we bumped into several difficulties. I will mention now the most important ones.

To be sure that my data is reliable what I measure, I made a measurement where I made time-measuring on repeated measurements on one gigabyte-size array with sequential policy.

It was a good idea, because several issues turned out.

Two CPU sockets

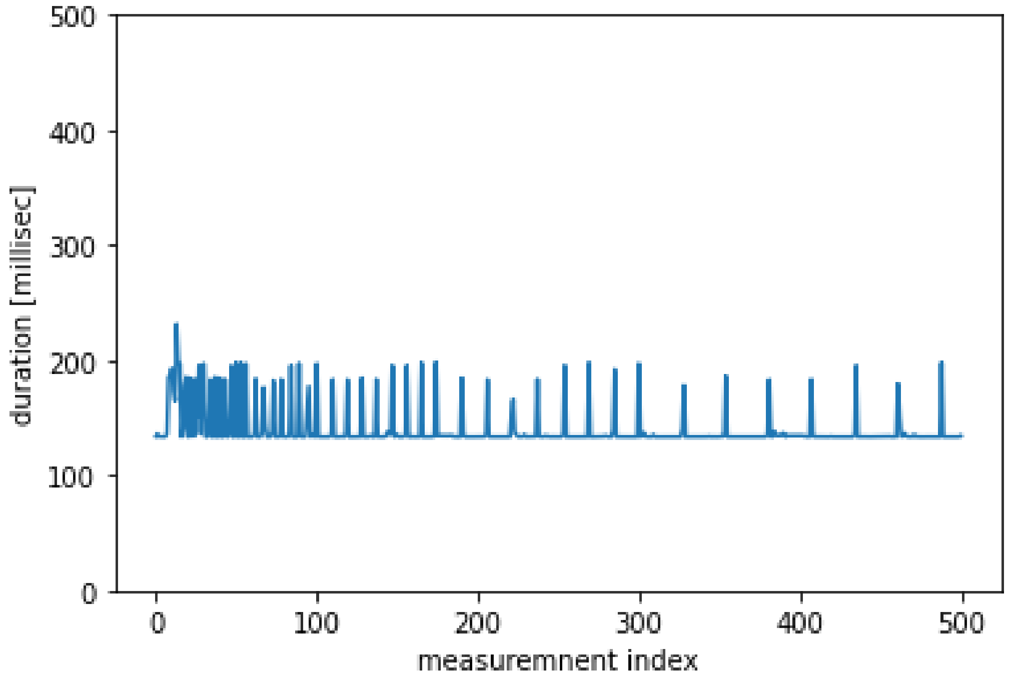
The first problem was that as you can also notice on the following plot, the runs were faster and faster slightly. It is in the real world not a problem but by my research it would bring uncertainty to the measurement.



Repeated measurements with large noise (it is because of multiple processor cores communication

We guessed that it is because there are two cpu sockets on the machine and it causes this noise. Because the cores are communicating with each other and it causes a large overhead.

Our guess was right. After we switched off one of the two sockets, the phenomenon disappeared. You can see it on the next plot.



Repeated measurements with fewer noise (here just one CPU core is working)

Noisy measurement in the beginning

Another problem was, as you can see on the previous plot, during the repeated measurements the first measurements were really noisy.

We could not find out why it is, so we decided to get samples between the 300th and the 400th measurement and average them because there is much less noise. This way we have got much reliable data.

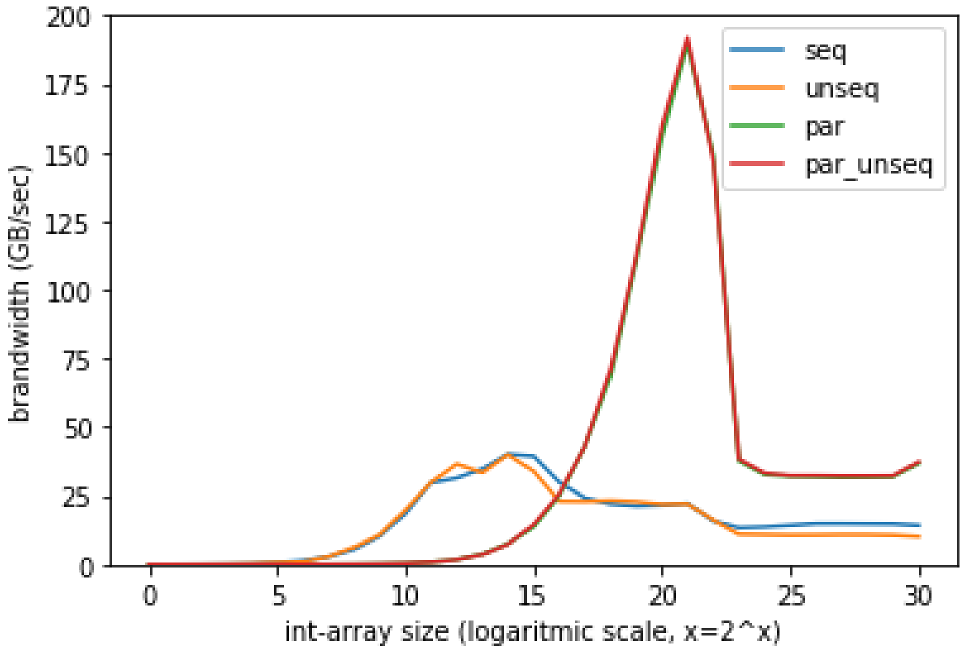
Runnings for each execution policies

Running on CPU

We run the evaluation application with each execution policy. Firstly on CPU

Evaluation of std::copy

Firstly we evaluated the std::copy algorithm. The result is visible on the following plot.



Std::copy (the green par execution is not really visible, because it is nearly the same as the par_unsec)

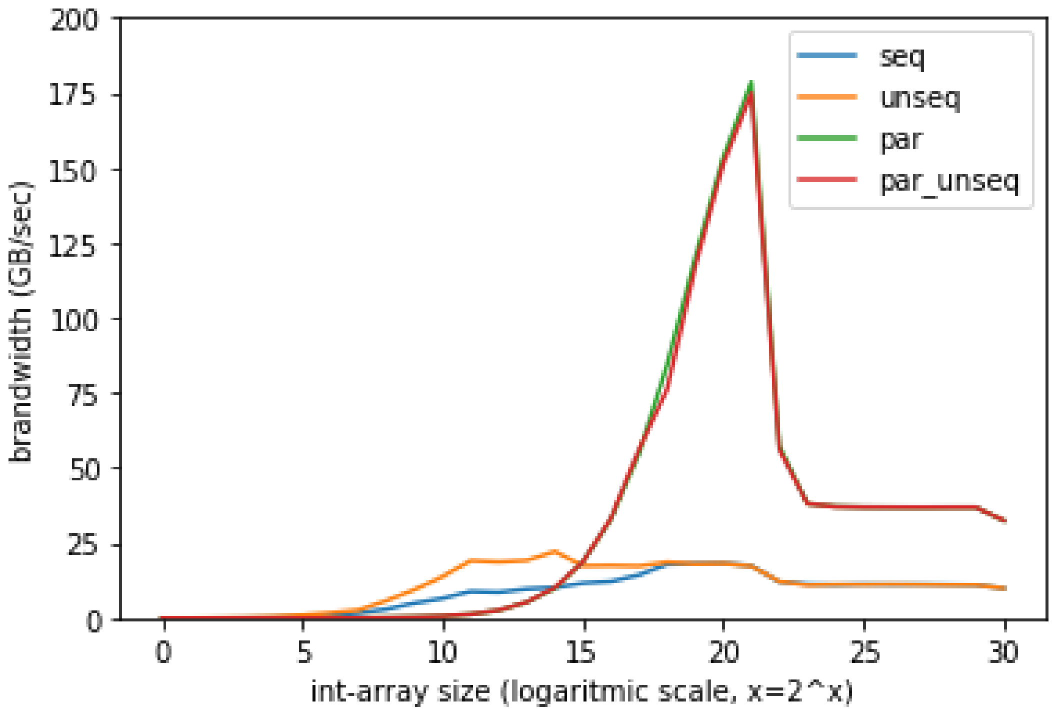
We can see that the parallel execution (par, par_unsec) of the copy is not really fast with small array sizes , because there is a huge overhead.

In contrast over 2^{16} size it is much faster than the sequential execution. In case of 2^{20} size the parallel execution is around 7 times faster than the sequential.

Over 2^{20} arraysize the performance falls back drastically. The reason for that is we run out here from the capacity of the cache memory.

Evaluation of std::transform

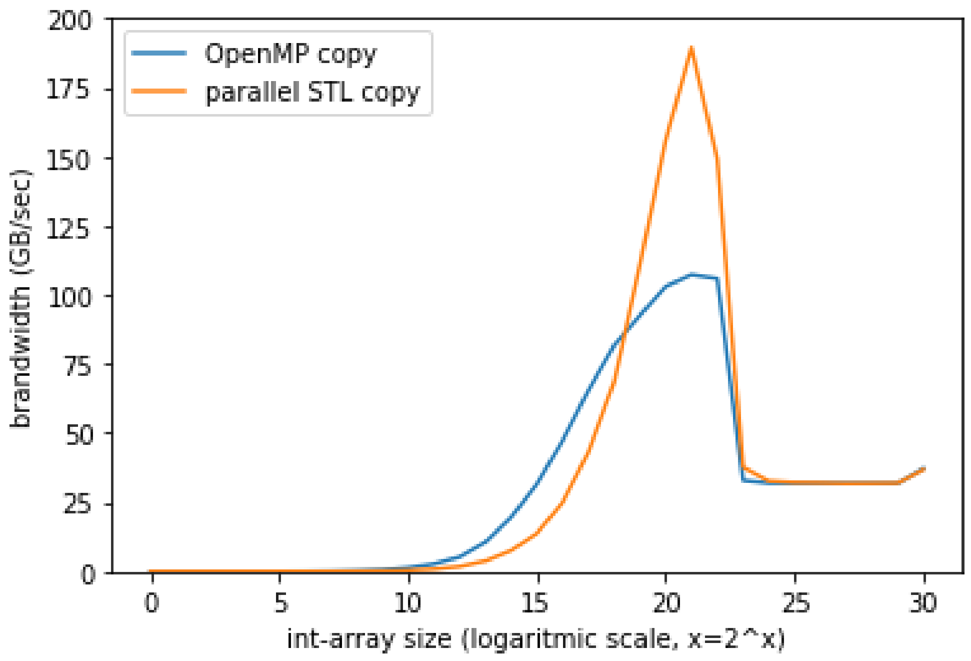
Secondly we evaluated the std::transform function. The phenomenon is really similar to the copy.



std::transform

Compare with OpenMP

We also ran with OpenMP. The reason was that OpenMP is the most commonly used parallelism extension for c++. So we were really curious about the result.

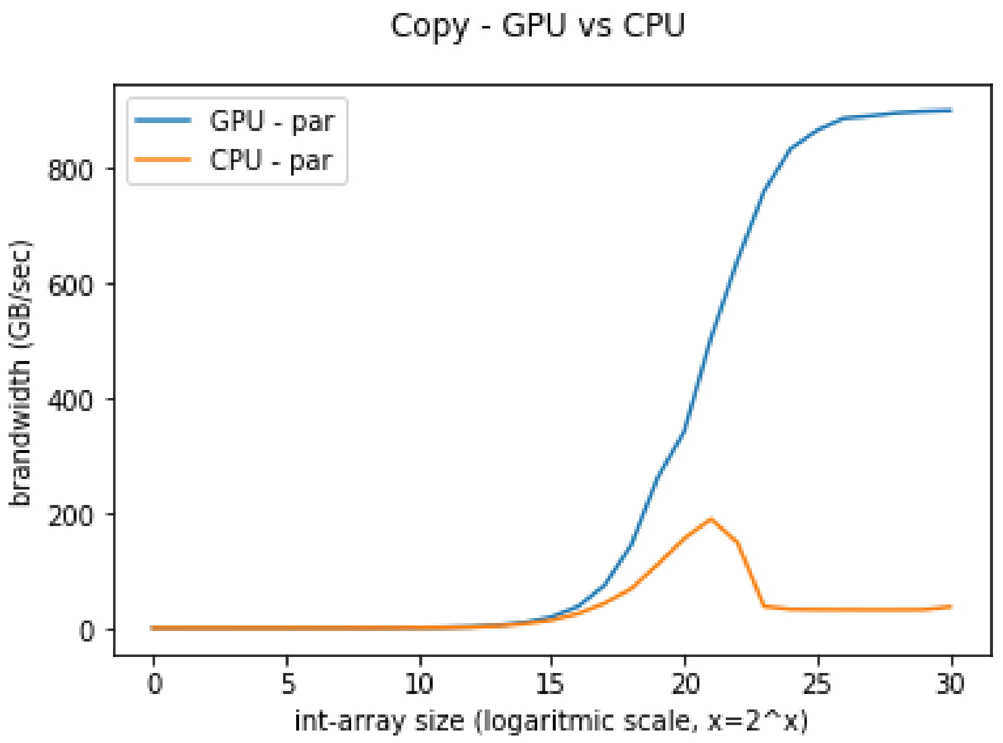


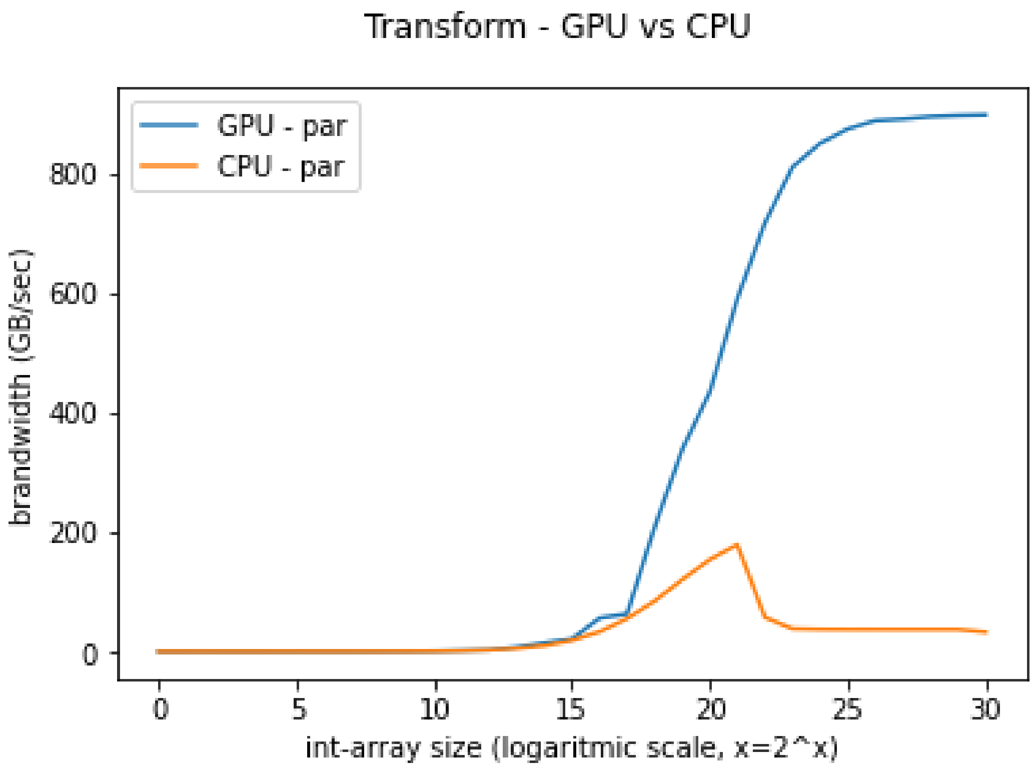
Std::copy with par policy vs OpenMP

The result as you can see on the plot above was really interesting. It turned out that from 2^{10} int-array size to 2^{18} the OpenMP is more efficient.
In contrast form 2^{18} int-array size (it means from 2^{20} byte array size) the Parallel STL is faster than OpenMP. (or around the same after we run out from the cache)

Running on GPU

Secondly we run the application on GPU. It became much faster as you can see on the following plots. It is because GPU-s are designed for these tasks, they can execute a single instruction on multiple data really efficiently via parallelism.


Copy - GPU vs CPU
GPU - par
CPU - par
brandwidth (GB/sec)
800
600
400
200
0
0
5
10
15
20
25
30
int-array size (logaritmic scale, x=2^x)


Transform - GPU vs CPU
GPU - par
CPU - par
brandwidth (GB/sec)
800
600
400
200
0
0
5
10
15
20
25
30
int-array size (logaritmic scale, x=2^x)

Summary

Our aim was to make a detailed evaluation about the parallel STL on CPU and on GPU, which we have successfully done. We were able to implement the copy and the transform algorithms with the parallel standard library. We evaluated it on different array sizes.

We were also successful in overcoming several implementation obstacles to have indeed valid evaluation.

We saw that over 2^{20} byte array size the parallel STL is more efficient than the OpenMP. It is a great result because OpenMP is the most used nowadays. We can say, it is the state of the art.

So we can say as a conclusion that parallel STL is a really good library, we can recommend it to use in your c++ application if you do algorithms with large containers.

Future work

A future work can be an application of Parallel STL. A concrete example is that there is a simulation application on the Faculty which has a really high cost part. Maybe a parallel STL sort can solve this part more efficiently.

Acknowledgement

I am thankful for my supervisor for the lot of help, without which this research would not have been possible. In addition I thank my Faculty PPCU FITB for making it possible to use the GPU server for my work.

References

[1] “Accelerating Standard C++ with GPUs Using stdpar,” *NVIDIA Developer Blog*, Aug. 04, 2020. https://developer.nvidia.com/blog/accelerating-standard-c-with-gpus-using-stdpar/ (accessed Dec. 10, 2020).

[2] “Get Started with the Intel® C++ Compiler 19.1 for Linux*,” *Intel*.

https://www.intel.com/content/www/us/en/develop/documentation/get-started-with-cpp-compiler-for-linux/top.html (accessed Dec. 14, 2020).
[3] “std::execution::sequenced_policy, std::execution::parallel_policy, std::execution::parallel_unsequenced_policy, std::execution::unsequenced_policy - cppreference.com.” https://en.cppreference.com/w/cpp/algorithm/execution_policy_tag_t (accessed Dec. 14, 2020).
[4] “Intel® C++ Compiler 19.1 Developer Guide and Reference,” *Intel*. https://www.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top.html (accessed Dec. 14, 2020).
[5] “NVIDIA HPC SDK Version 20.11 Documentation.” https://docs.nvidia.com/hpc-sdk/index.html (accessed Dec. 14, 2020).
[6] “Date and time utilities - cppreference.com.” https://en.cppreference.com/w/cpp/chrono (accessed Dec. 16, 2020).