13th International Conference on Applied Informatics
Eszterházy Károly Catholic University
Eger, Hungary, February 19–21, 2026

Extensions of the C++ Parallel STL library

Zoltán Porkoláb

ELTE Eötvös Loránd University
Faculty of Informatics
Dept. of Programming Languages and Compilers
gsd@inf.elte.hu

1. Introduction

The C++ programming language is still very popular amongst developers who are responsible to implement high performance systems, many times utilize modern multicore hardware writing highly parallel applications. However, implementing platform independent multithreaded programs are far from trivial. C++17 introduced Parallel STL to simplify writing parallel code for developers already comfortable with the long-term used Standard Template Library. However, while Parallel STL can be powerful, it also presents risks: as many experts have noted, issues like non-commutative or non-associative algorithms can lead to unexpected results, and the standard offers plenty of other ways to “shoot yourself in the foot”. In this paper we discuss some of these problems, including the implementation of a generic filter-reduce algorithm for solving niche parallel tasks, and lessons learned along the way.

2. The Parallel STL library

Since the early years of the new millennia, the importance of parallel programming has been well-known for programmers. As the Moore’s Law does not apply anymore for the speed of the processors, software performance could be increased mostly via enabling parallel execution [7]. However, writing efficient, scalable, and correct parallel programs is inherently challenging. While most programming languages provide low-level multi-threading elements: thread-management classes, mutex objects, locks, conditional variables, etc., complex programs should avoid various traps and pitfalls like possible race conditions, deadlocks, or resource starvation. Here these low-level language constructions are not really helpful.

In the same time writing multi-threaded programs usually does not bring efficiency and scalability automatically. Often the programmer not only has to understand effective parallel constructs but also should be aware of platform-specific features, sometimes even down to the hardware level.

Since the C++17 version of the Standard [2, 8], the standard library provides the *Parallel Stl*, a parallel version of the Standard Template Library. The Standard Template Library (STL) is part of the C++ language since the first standard, known and widely used by the developer community [5]. In C++17 most of the STL algorithms received new overloads that take a first *execution policy* parameter, which specifies the algorithm's – possibly parallel – execution. [4] The C++ community highly appreciated the Parallel STL library as it promised a safe and effective way to implement business logic on a higher abstraction level; while it still promised efficient platform-specific concurrent implementation.

3. Problem statement

Since the introduction of the Parallel STL, practice identified several issues, mostly related to non-commutative and non-associative algorithms, that can cause serious errors for inattentive or inexperienced developers [1, 3, 6]. While these problems are widely discussed in the C++ community, the na ive use of Parallel STL may lead to surprising results in other, less known areas as well. In this work we will discuss some of these problems, namely algorithms, which requires returning the results in some well-defined order.

Many old-style STL algorithms, like `find` and `find_if` returns the first element that matches the search criteria inside a container. However, when we apply the corresponding Parallel STL algorithm to the very same data, the outcome may differ, returning a different result or not to find any result at all. The root cause of this phenomenon is the na ive implementation of the parallel version of the algorithm, which usually just applies the same criterion on multiple slices of the container and then tries to fold the results. However, the parallel execution on the slices most cases does not guarantee a stable behavior; the results on the slices return in undefined order.

4. A filter-reduce algorithm

We designed a generic filter-reduce algorithm to solve the problem described in the previous section. The algorithm executes a parallel *filter* step to search elements with a given criteria in a container then executes a sequential *reduce* step to identify the element(s) we are looking for. The filter criteria is local, i.e., it does not depend on other elements in the container, while the reduce step keeps the order of the elements. The algorithm is specifically effective in cases where the result set of the filter step is significantly smaller than the original data set, therefore the sequential reduce step is executed on relatively few elements.

We implemented the algorithm as a working prototype using standard C++17 features and on the bases of the Parallel STL itself. That way we can utilize the effective portable implementation of the Parallel STL while providing a flexible way to parameterize the algorithm using C++ functors.

Measurement data shows that the prototype produces the very same results as the classical sequential STL algorithms, but in can overperform the sequential solutions.

References

[1] B. Barth, R. Szalay, Z. Porkoláb: *Towards Safer Parallel STL Usage*, in: 2022 IEEE 16th International Scientific Conference on Informatics (Informatics), 2022, pp. 39–44, doi: 10.1109/Informatics57926.2022.10083416.

[2] *ISO/IEC 14882:2020 – Programming languages C++*, visited 2022-05-20, url: https://www.iso.org/standard/79358.html (visited on 05/20/2022).

[3] N. M. Josuttis: *C++17: The Biggest Traps – [C++ on Sea 2019]*, visited 2022-05-07, url: http://youtube.com/watch?v=mAZyaAo3M70&t=3975.

[4] D. Kühl: *CppCon 2017: C++17 Parallel Algorithms*, visited 2022-10-14, url: http://youtube.com/watch?v=Ve8cHE9LNfk&t=1784.

[5] D. R. Musser, A. A. Stepanov: *Algorithm-oriented generic libraries*, Software: Practice and Experience 24.7 (1994), pp. 623–642, url: http://stepanovpapers.com/musser94algorithmoriented.pdf.

[6] N. Pataki: *Safe iterator framework for the C++ Standard Template Library*, Acta Electrotechnica et Informatica 12.1 (2012), p. 17, url: http://aei.tuke.sk/papers/2012/1/03_Pataki.pdf.

[7] H. Sutter et al.: *The free lunch is over: A fundamental turn toward concurrency in software*, Dr. Dobb's journal 30.3 (2005), visited 2022-10-14, pp. 202–210, url: http://gotw.ca/publications/concurrency-ddj.htm.

[8] *Working Draft – Standard for Programming language C++*, visited 2022-05-20, url: http://eel.is/c++draft (visited on 05/20/2022).