Annales Mathematicae et Informaticae
63 (2026) pp. 122–131
DOI: 10.33039/ami.2026.06.007
URL: https://ami.uni-eszterhazy.hu

Extensions of the C++ Parallel STL library

Zoltán Porkoláb, Gábor Tóthvári

ELTE Eötvös Loránd University
Faculty of Informatics
Department of Programming Languages and Compilers
Budapest, Hungary
gsd@inf.elte.hu
eyjqmy@inf.elte.hu

Abstract. 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 traps, including the set of problems, where the answer must preserve the original order of the elements.We propose a generic *filter-reduce* algorithm for solving such parallel tasks. The algorithm is implemented on top of the Parallel STL itself, therefore it utilizes all of its benefits. Tests prove that filter-reduce is scalable and performs well on the specific set of problems.

Keywords: C++, ParSTL, parallel algorithms

2020 *Mathematics Subject Classification:* Primary 68W01, 68W10

1. Introduction

Since the early years of the new millennia, the importance of parallel programming has been well-known for programmers. Earlier hardware development provided faster and faster chips and programmers were able to utilize this speed-up with single threaded programs. However, by the early years of the new millennia hardware development reached strong barriers and the generic processor speed-up

Submitted: March 31, 2026
Accepted: June 30, 2026
Published online: July 17, 2026

has stopped. 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 [11].

Unfortunately, writing efficient, scalable, and correct parallel programs is inherently challenging. Most programming languages provide low-level multi-threading elements: thread-management classes, mutex objects, locks, conditional variables, etc. and it is the programmers' task to build efficient higher level algorithms from these components. Meanwhile, these complex programs should avoid various traps and pitfalls like possible race conditions, deadlocks, or resource starvation.

At 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. What is the cost of starting a new thread on the actual platform? How much resources (processors, memory) are available without overcommitting? How expensive is the context switch between threads? These questions become even more critical when implementing a portable software running on various platforms.

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. Since the C++11 standard, the C++ programming language provides the necessary building blocks for writing portable parallel programs: the thread class, various mutexes, lock guards, conditional variables and even atomic operations. However, implementing *effective* platform independent multithreaded programs are far from trivial in C++11.

Since the C++17 version of the Standard [3, 14], 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 [6]. 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 [5]. The C++ community highly appreciated the Parallel STL library as it promised a safe and effective way to implement business logic on a (relatively) higher abstraction level; while it still promised efficient platform-specific concurrent implementation. As the original STL library is well known for everybody coding in C++, Parallel STL provides an easy entry level to the parallel world for the average programmer.

Unfortunately, this promise can be misleading in some cases. Some algorithms require special conditions that are less familiar to programmers. Others work deceptively and the results differ from a sequentially executed version of the same algorithms. In this article, we will discuss the latter cases of tasks, specifically where we want to obtain the result in sequential order after a filtering that can be performed in parallel. We implemented a new *filter-reduce* algorithm, where the filter step can be effectively executed in a parallel way, but the reduction step requires keeping the order of the resulting elements. We implemented a working prototype of the algorithm in two variants built entirely on top of the Parallel STL,

therefore preserving its portability and efficiency. The two variants using either a mutex or a lock-free solution for the only critical part of the algorithm. We tested the prototype on several test cases to prove its scalability.

This paper is organized as follows. In Section 2 we overview the basics of the Parallel STL library. We discuss the known issues regarding the library in Section 3. Our filter-reduce algorithm and their implementation details are described in Section 4. Then in Section 5 we evaluate our measurement results. The paper concludes in Section 6.

2. The Parallel STL library

The *Standard Template Library (STL)* is a central element of the C++ programming language since the original C++98 standard [3, 14]. The STL provides various data structures – sequential, associative and (since C++11) unsorted containers and fundamental algorithms for searching, copying, and modifying data. The connection between the containers and the algorithms is provided by *iterators* – an abstraction of the pointer type [6]. The essence of the STL is to generalise as much as possible without a performance penalty [9] using parametric polymorphism – *templates* in C++ [12]. The algorithms often accept *functor* parameters as customisation points: specific classes that provide public `operator()` methods and thus behave as if they were functions. Such functors can have a state and are the higher level generalisation of the basic operations and functions. Since C++11, one can also use *lambda functions* instead of named functor classes.

Writing parallel algorithms for the STL is not trivial. One should decide how to split the containers into partitions for parallel execution, how many threads can be run efficiently on a specific target platform, and whether the cost overhead of parallelisation is less than the single-threaded execution. Such decisions are well-observable in Listing 1, an example from Bjarne Stroustrup [10] for summing the elements of a `std::vector`. Most of these questions depend on the size of the container, the task we execute, and on specifics of the target platform.

Parallel STL became part of the C++ Standard in 2017 to simplify the execution of parallel algorithms. The new library introduced overloaded versions for most of the original STL algorithms by extending them with a new first parameter, the *execution policy* which specifies how the algorithm is supposed to be executed. Possible values of the execution policy specify sequential or parallel execution with further specifying possible SIMD execution. The parameter's value describes the *requested* execution policy, but this is just a suggestion: the implementation may decide a different strategy, e.g., when the cost of starting new threads would be greater than simple sequential execution. The Standard on execution policy is *open*: additional execution policies may be provided by a standard library implementation, e.g., for the benefit of a specific platform.

The main advantage of the Parallel STL is to reduce the entry level to start using parallel algorithm. Programmers can utilize their knowledge of the classical STL while they do not need to know platform specific information. As an example,

observe Listing 2 as the Parallel STL based solution of summarization of the vector elements we have seen on Listing 1 implemented without the Parallel STL.

Listing 1. Manual parallelisation of STL based on Bjarne Stroustrup.

```
// Simple accumulator function object.
template <class T, class V>
struct Acc {
  T* b;
  T* e;
  V val;
  Acc(T* bb, T* ee, const V& vv) : b{bb}, e{ee}, val{vv} { }
  V operator()() { return std::accumulate(b, e, val); }
};
double sum(const std::vector<double>& v) {
  if (v.size() < 10000) // execute non-parallel for small vectors
    return std::accumulate(v.begin(), v.end(), 0.0);

  // spawn to 4 tasks if v is large enough.
  auto f0{async(Acc{&v[0], &v[v.size() / 4], 0.0})};
  auto f1{async(Acc{&v[v.size() / 4], &v[v.size() / 2], 0.0})};
  auto f2{async(Acc{&v[v.size() / 2], &v[v.size() * 3/4], 0.0})};
  auto f3{async(Acc{&v[v.size() * 3/4], &v[v.size()], 0.0})};

  return f0.get() + f1.get() + f2.get() + f3.get();
}
```

Listing 2. Summarizing the elements of a vector using Parallel STL.

```
// Using the Parallel STL
double sum(const vector<double>& v) {
  double res = std::reduce(std::execution::par, v.begin(), v.end(), 0.0);
  return res;
}
```

Parallel STL algorithms are usually implemented on the top of a lower level library.In most UNIX systems implementation uses Intel's *Threading Building Blocks* (TBB) library [2, 8]. In the core of the implementation we usually find a thread pool which works on the container divided into small slices. Contrary to the example on Listing 1 there are much more slices than working threads, so each thread in the pool will work on many slices. This helps to balance the workload between threads, as fast-finishing threads may process more slices. In the same time there is no guarantee that the earlier completed slice will contain the earlier elements in the container.

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, 4, 7]. While these problems are widely discussed in the C++ community, the naïve 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 naïve 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. Moreover, if the criteria of the search is non-local, e.g., we are looking the Nth element fulfilling the predicate, that "count" will be local for the slice a thread in working on.

Listing 3. Naïve use of parallel find_if to find the 3rd element greater than 50.

```
#include <vector>
#include <iostream>
#include <algorithm>
#include <execution>
int main() {
 std::vector<int> v(100'000, 2); // fill a large vector with small elements
 // set a few random elements greater than 50
 v[5001] = 123;
 v[11001] = 999;
 v[22222] = 1233; // <--- 3rd
 v[33333] = 546464;
 v[55555] = 555;
 v[77777] = 7657;
 v[77778] = 7658;
 v[77779] = 7659;
 v[88888] = 312231;

 const auto res = std::find_if( std::execution::par, v.begin(), v.end(),
           [counter = 0](int i) mutable { if ( i>50 ) return ++counter == 3;
                                          else return false; });
 if ( res != v.end() )  // we found it
   std::cout << *res << '\n';
 return 0;
}

```

Consider the example on Listing 3. We initialized a large vector with small elements (almost all are smaller than 50), except a few random elements which value is greater than 50. We are looking for the 3rd element which has a value greater than 50. We use the well-known `find_if` algorithm from STL, which takes the first parameter as the execution policy, then two parameters as the specification

of the search interval, and the last parameter as a predicate functor to specify the element we are searching for. If we run this algorithm sequentially we correctly find the 3rd element greater than 50 at index 22222 (line 10). However, when we run the algorithm with parameter `std::execution::par`, it can happen that we find the result e.g., at index 77779 (at line 15) or we do not find the third element greater than 50 at all.

The reason is that when running in parallel mode, the algorithm applies the functor *for all slices independently*. Even if there are many elements greater than 50, it can happen that neither of the slices contain 3 elements fulfill the search criteria. Moreover, if we are "lucky" to find 3 elements greater than 50 in a certain slice we do not know whether we have similar elements at any of the previous indexes.

One can suggest to use a shared state to count the number of elements found. A possible implementation could be a shared pointer from the functors pointing to a single atomic integer variable. Apart from the fact that it would completely destroy the efficiency of parallel execution, another problem is that we do not know the order in which each slice is processed. It may happen that a slice containing higher indexes is finished to process before the slice(s) of lower indexes. Therefore our shared counter would be totally unusable.

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.

A possible usage scenario can be seen of Listing 4. Similar to the basic idea of the `transform_reduce` algorithm [1, 4], we also split the search to be performed into a locally and parallel executed filter step and a sequential reduce step performed on the filter result.

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.

The filter step is executed by a Parallel STL `for_each` algorithm, which detects when a new slice starts to be processed and creates a new vector of iterators for each slices to collect the filter results. As these vectors are created by slice, no data race or even false sharing occurs during the filter step between the execution threads. However, as we do not know in which order the slices are processed we need to collect these result vectors into one single container – called *Collector* – preserving

the relative ordering of the results. That is the only step when we have to be concerned with possible data race. We came up with two possible implementations: one uses ordinary mutex-based locking and the other is implemented with a lock-free data structure. The mutexed version uses `std::mutex` for synchronization, while the lock-free one is based on an implementation by Williams [13]. It is a linked list stack, using the `compare_exchange_weak` operation from the C++ standard library. The difference when compared with mutexes is when a thread has to wait before entering the critical section it is not put to sleep, but rather it can immediately retry the operation.

After the parallel filter step has finished the reduce step is running in a sequential way. This reduce step walks thru the filter results stored in the Collector and returns whatever return type is specified in the reduce parameter.

Listing 4. Usage example for the filter-reduce algorithm.

```
#include <vector>
#include <iostream>
#include <algorithm>
#include <execution>
int main() {
 std::vector<int> v(100'000, 2); // fill a large vector with small elements
 // set a few random elements greater than 50
 v[5001] = 123;
 v[11001] = 999;
 v[22222] = 1233; // <--- 3rd
 v[33333] = 546464;
 v[55555] = 555;
 v[77777] = 7657;
 v[77778] = 7658;
 v[77779] = 7659;
 v[88888] = 312231;

 const auto res = filter_reduce(std::execution::par, v.begin(), v.end(),
                  [](int i) { return  i > 50; },  // filter
                  [cnt = 0] (int) mutable { return 3 == ++cnt; }); // reduce
 if ( res != v.end() )  // we found it
   std::cout << *res << '\n';
 return 0;
}
```

5. Evaluation

We have benchmarked the speed of the implementations and compiled the results into Table 1 and Table 2. As testing the correctness of the algorithm, the prototype always has produced the same results as the classical sequential STL algorithms. As expected, parallel execution has greatly reduced the running times. For small amounts of data, the lock-free implementation enjoyed a noticeable lead over the

mutexed version. As the size of the dataset increased, however, the difference was much smaller.

The benchmarks tested two kinds of workloads, depending on the cost of the filter (search condition) function. There was an "easy" workload, with the filter condition being the greater than operator (see Table 1). The operator has a minimal cost of execution, therefore the parallel filter step required less effort. Furthermore, a hard workload was also tested where the condition was a prime number testing function (see Table 2). Here, the parallel filter step required much more effort.

When we present the results on a logarithmic scale on Figure 1, it is observable that the algorithm scales well when the input size increases.

Table 1. Timing results of easy workload benchmark in milliseconds.

Amount of numbers	10^5	10^6	10^7	10^8
Maximum number greater than `RAND_MAX`				
Singlethreaded	1.11	7.23	78.65	736.06
Mutexed	1.69	2.69	16.63	146.69
Lockfree	0.38	2.13	15.41	143.95
Nth number greater than `RAND_MAX`				
Singlethreaded	0.11	0.80	6.75	75.35
Mutexed	1.02	2.34	2.04	10.85
Lockfree	0.23	0.33	0.94	8.17

Table 2. Timing results of hard workload benchmark in milliseconds.

Amount of numbers	10^5	10^6	10^7	10^8
Maximum prime number search				
Singlethreaded	128.43	1246.65	12 360.50	124706.00
Mutexed	24.94	225.53	2237.01	22391.90
Lockfree	23.70	225.46	2231.61	22375.30
Nth prime number search				
Singlethreaded	251.10	2474.89	24 640.80	255497.00
Mutexed	45.95	446.85	4441.81	50154.80
Lockfree	44.91	446.55	4445.92	61809.60

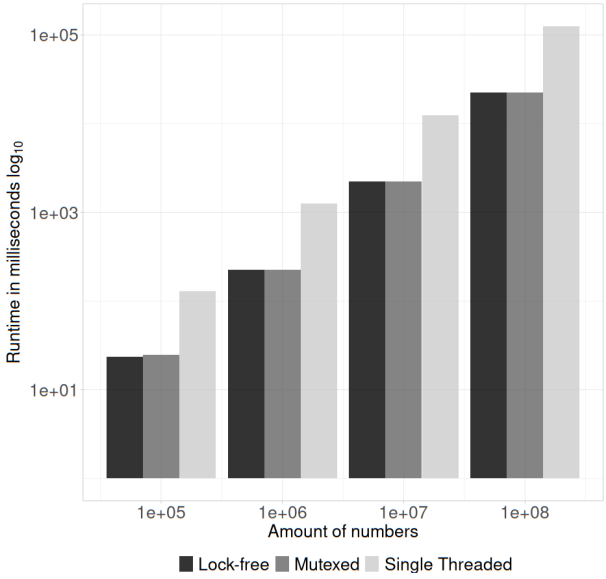

Figure 1. Benchmark results of the maximum prime number search.

6. Conclusion

Writing effective and portable parallel code is a challenging task which requires deep understanding of the language elements and the platforms. Since C++17, the C++ programmers can use the Parallel STL, an easy-to-learn extension of the customary Standard Template Library. However, naïve use of some Parallel STL algorithms may produce puzzling results. There is also lack of algorithms which obtain the result in sequential order after a filtering executed in parallel way. To fill this gap, we introduced a filter_reduce algorithm, which applies a filter criteria executed in parallel way on the elements, then a reduce step provides a well-order result sequentially.

We implemented a working prototype with two locking mechanism: one using a classical mutex and the other implemented as a lock-free algorithm. We tested the prototype on various configurations and proved that the algorithm is useful, especially when the filtering criteria is true on a significantly smaller number of elements.

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] Intel Corporation: *Intel Threading Building Blocks*, visited 2022-05-07, URL: http://intel.com/content/www/us/en/developer/tools/oneapi/onetbb.html.

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

[4] 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.

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

[6] 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.

[7] 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.

[8] J. Reinders: *Intel Threading Building Blocks - outfitting C++ for multi-core processor parallelism*, Jan. 2007, ISBN: 978-0-596-51480-8, URL: http://oreilly.com/library/view/intel-threading-building/9780596514808.

[9] A. Stepanov, D. E. Rose: *From Mathematics to Generic Programming*, 1st ed., New York: Springer, 2009, ISBN: 978-0387952902.

[10] B. Stroustrup: *C++11 - the new ISO C++ standard*, visited 2022-05-07, URL: http://stroustrup.com/C++11FAQ.html (visited on 05/07/2022).

[11] 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.

[12] D. van de Voorde, N. M. Josuttis, D. Gregor: *C++ Templates: The Complete Guide (2nd Edition)*, 2nd, Addison-Wesley Professional, Sept. 2017, ISBN: 978-0-321-71412-1, URL: http://tmplbook.com.

[13] A. Williams: *C++ Concurrency in Action*, 1st ed., Manning Publications, Feb. 2012, ISBN: 9781933988771.

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