Towards Safer PARALLEL STL Usage

Benjámin Barth
Faculty of Informatics
Eötvös Loránd University
Budapest, Hungary

Richárd Szalay [0000−0001−5684−5158]
Department of Programming Languages and Compilers
Eötvös Loránd University
Budapest, Hungary
szalayrichard@inf.elte.hu

Zoltán Porkoláb [0000−0001−6819−0224]
Department of Programming Languages and Compilers
Eötvös Loránd University
Budapest, Hungary
gsd@inf.elte.hu

***Abstract*—Effective and safe parallel programming is among the biggest challenges of today's software technology. The C++17 standard introduced PARALLEL STL: a set of overloaded functions taking an additional "execution policy" parameter in the Algorithms chapter of the Standard library. During the years since its introduction, a few shortages of *Parallel STL* have been revealed. While the Standard defines the semantics of the individual algorithms, adherence to their abstract requirements – e.g., absolutely no data races or deadlocks during the evaluation of a predicate or other customisation point – is up to the developer. Experience shows that programmers frequently make mistakes and write erroneous code, which is hard to debug. In this paper, we investigate some of the critical issues of the *Parallel STL* library and suggest improvements to increase its safety. While a fully automatic detection of erroneous constructs is computationally infeasible to do, we introduce a framework with which the user will be able to indicate – axiomatically, based on absolute trust – that an operation has "safe" properties, e.g., commutativity of certain functors. We implemented a prototype of the proposed framework to demonstrate its usability and effectiveness.**

***Index Terms*—C++ programming language, Standard Template Library, concurrent programming, parallel programming, type safety**

I. INTRODUCTION

Since the early years of the new millennia, the importance of parallel programming has been well-known for programmers. The change of the fundamental hardware features, as Moore's Law does not apply anymore for the speed of the processors, started a new era of software development where better software performance could be achieved primarily via enabling parallel execution [1].

However, writing efficient, scalable, and correct parallel programs is inherently challenging. Most programming languages provide only low-level multi-threading elements: thread-management classes, mutex objects, locks, conditional variables, etc. The programmer has to construct higher-level algorithms to handle critical sessions, mutual exclusions, and should avoid various traps and pitfalls like possible race conditions, deadlocks, or resource starvation. Meanwhile, efficiency and scalability do not come for free either. 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.

The *Standard Template Library (STL)* is one of the most fundamental elements in the C++ standard library. The STL provides data structures as containers and algorithms that work on them. Using the STL, programmers can solve the most common daily tasks at a relatively high abstraction level and can use the elements and vocabulary of the STL to build more complex software components. We can rarely find C++ programs that do not employ the STL – or a library built with STL techniques – and all C++ programmers should be familiar with both its syntax and semantics.

Since the C++17 iteration of the Standard, the C++ standard library provides a version of the STL that enables *parallel programming*, called PARALLEL STL. Most of the algorithms received new overloads that take a first *"execution policy"* parameter, which specifies the algorithm's – possibly parallel – execution. 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.

However, the practice shows that in case of non-trivial problems, Parallel STL programmers tend to write suboptimal and not infrequently even erroneous code. The reason is that Parallel STL offers only minimal safety guarantees while expecting the programmers to ensure a number of hidden rules – rules which are written in the language standard, but neither the compiler nor other tools check them. In some cases, non-expert programmers are not even aware of these restrictions.

In this paper, we overview some of the known issues of Parallel STL. We discuss some theoretically possible solutions and suggest a few small practical steps towards the safer usage of Parallel STL. Our main contribution is a library-level framework which warns the programmer about the incorrect use of the library features. While our solution is not "hard" in the sense that it requires the co-operation of the programmer, we think it is helpful to raise the programmer's attention to some typical possible problems and thus avoid some accidental misuses of the library.

The rest of the paper is organised as follows. In section II, we overview the C++17 Parallel Standard Template Library. Section III discusses the problems the C++ community experienced with the library. Possible theoretical solutions with their advantages and disadvantages are evaluated in section IV. Our solution is introduced in section V. Future plans are discussed in section VI. The paper concludes in section VII.

II. The Parallel STL library

In this section, we give an informal introduction to the STL and its Parallel STL library. For a more formal approach, the Reader is encouraged to refer to the C++ Standard [2], [3] and related literature, such as the *Template Guide* book [4].

The *Standard Template Library (STL)* is a central element of the C++ programming language. It has been part of the language Standard since the original C++98. The STL provides various data structures (sequential, associative and unsorted containers) and fundamental algorithms for searching, copying, and modifying data. The connection between the containers and the algorithms is provided by *iterators* – a generalisation of the pointer type [5].

The essence of the STL is to generalise as much as possible without a performance penalty. For this reason, both the containers and the algorithms utilise *parametric polymorphism*, i.e., they are class and function *templates* [6]. Moreover, to further generalise, 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 use *lambda functions* instead of named functor classes.

```
double sum(std::vector<double>& v) {
  return std::accumulate(v.begin(), v.end(), 0.0);
}
```

Listing 1: A sample STL algorithm call.

The novel feature of the STL is its symmetric approach: one can implement a container with a constant effort regardless of the number of the algorithms, and a new algorithm with also a constant effort that can work on different – already implemented or future – containers. Therefore, the STL is a good answer for the *expression problem* [7]. This is possible because the algorithms need not be aware of the internal structure of the containers: they are using purely the interface provided by the iterators. Iterators – implemented usually as member classes for the containers – are responsible for accessing the elements of their containers.

Iterators are classified into different categories based on their strategy to access the elements. Some categories are *refinements* of others, i.e., they provide the same properties with some extensions. *Input* and *output* iterators are for reading and writing elements of containers, sometimes corresponding to various stream objects. *Forward* iterators can walk through containers one-by-one and access elements for reading or writing by dereferencing, possibly multiple times. *Bidirectional* iterators are further refinements of forward iterators, adding the capability of stepping in the backward directions, too. An example of a standard C++ container with a bidirectional iterator is `std::list`. *Random access* iterators can access the container elements directly without walking through the elements one-by-one. A typical container with random access iterators is `std::deque`. Since C++20, a further refinement defines the *contiguous* iterator category, which guarantees that the elements in the container are stored contiguously in the memory. Such a container is `std::vector`.

Writing parallel algorithms for the STL was never an easy task. One should decide how to split the containers into (equal) 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 2, an example from Bjarne Stroustrup [8] 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.

```
// 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);
  }
};

// Spawn many tasks if v is large enough.
double sum(std::vector<double>& v) {
  if (v.size() < 10000)
    return std::accumulate(v.begin(), v.end(), 0.0);

  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{asnyc(
    Acc{&v[v.size() * 3/4], &v[v.size()], 0.0})};

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

Listing 2: Manual parallelisation of STL.

Parallel STL became part of the C++ Standard in 2017 to simplify the execution of parallel algorithms. To minimise the learning curve, the new library introduced overloaded versions for most of the original STL algorithms by extending them with a new first parameter, the *"execution policy"*. The possible values of this parameter are entities in the `std::execution` namespace: `seq`, `par`, `par_unseq`, and – since C++20 – `unseq`. 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 is open here: additional execution policies may be provided by a standard library imple-

mentation, e.g., for the benefit of a specific platform. Possible future additions may include `std::execution::cuda` and `std::execution::opencl`. The container and the iterator libraries were not affected by the introduction of Parallel STL.

```
std::vector<double> xv{1000, 1.0}, yv{1000, 1.0};

double result = std::transform_reduce(
  std::execution::par,
  xv.begin(), xv.end(), yv.begin(), 0.0);
```

Listing 3: A sample PARALLEL STL algorithm call.

The advantage of such a library design is obvious: the programmer can reuse their previous knowledge of the STL and (theoretically) need not be bothered with the platform-specific details and the manual control of parallelisation, one can just specify the task, and the implementation finds the best way of the execution. That implementation is usually platform-dependent. On UNIX-like systems, this often uses the *Intel Threading Building Blocks (TBB)* library [9]. On Windows, the WINAPI-specific implementation is preferred.

Most algorithms are based on a straightforward method: the interval where the algorithm will execute is split into more-or-less equal subintervals, which are then associated with separate threads. The results are created by left folding the operation. All current implementations use work-stealing methods to speed execution up even further.

III. KNOWN ISSUES WITH PARALLEL STL

When using the *parallel* execution policy, it is the programmer's responsibility to avoid data races and deadlocks. Theoretically, programmers can implement their customised actions in a way that they provide the necessary synchronisations. However, this is far from trivial due to the lack of information on how the implementation will handle threads. In certain cases – related to vectorisations – programmers should also avoid memory allocation and deallocation, the acquisition of mutexes, etc. Parallel STL does not check and validate any of these precondition properties. Moreover, many Parallel STL algorithms cause *undefined behaviour* if we apply them with operations that lack certain semantic properties, such as *commutativity* or *associativity*.

The following example [10] demonstrates such an issue. In listing 4, we see the classic STL solution to add all the square values of an interval.

```
1 auto sqrsum = [](auto s, auto val) {
2   return s + val * val;
3 };
4 auto sum = std::accumulate(v.begin(), v.end(),
5                            0ULL, sqrsum);
```

Listing 4: Classic sum of squares in STL.

Line 2 defines the `sqrsum` functor – in the form of a lambda function – which implements the $s \mathrel{+}= v^2$ expression. This overload of the `std::accumulate` algorithm takes the range by the first two parameters `v.begin()` and `v.end()`; the initial value of the sum `0ULL`,[1] and the `sqrsum` functor. The algorithm then iterates on the $[$`v.begin()`, `v.end()`$)$ semi-closed interval left to right and calls `sqrsum` with the actual element of the range.

We have seen the manual parallelisation of algorithms in listing 2. The natural way with Parallel STL would be using the `std::reduce` algorithm with the same parameters we passed for `std::accumulate`. This is shown in listing 5.

```
auto sqrsum = [](auto s, auto val) {
  return s + val * val;
};
auto sum = std::reduce(std::execution::par,
  v.begin(), v.end(), 0ULL, sqrsum);
```

Listing 5: Naïve, and wrong, sum of squares with PARALLEL STL.

Let us recognise that there is no parallel version of `std::accumulate`. The `reduce` algorithm behaves like `accumulate`, except that the elements of the range may be grouped and rearranged in an arbitrary order. That is necessary to apply meaningful parallelism.

The "solution" in listing 5 may work for small ranges until the implementation enables real concurrent execution. At that moment, above a certain size threshold, we will start experiencing garbage results. The reason is the hidden contract which requires the used functor (`sqrsum`) to be both *commutative* and *associative*. Otherwise, the result of `std::reduce` is *non-deterministic* [11]. It is trivial to see that $f(s, v) = s + v^2$ breaks this rule.

```
auto sum = std::transform_reduce(
    std::execution::par, v.begin(), v.end(), 0ULL,
    /* Reduce    =*/ std::plus<>(),
    /* Transform =*/ [](auto v) { return v * v; });
```

Listing 6: Correct sum of squares with PARALLEL STL

The correct solution is shown in listing 6. Here, we use the function `std::transform_reduce` [12], which implements a *map-reduce* algorithm [13]. This algorithm takes two parameters instead of the single functor (like `reduce` did): the *reducer* – which implements the addition, here $+$ represented by `std::plus` from the standard library – and the *transformer* – which provides the squares of the elements, implemented by a trivial lambda function. The *reducer* parameter must be commutative and associative, which is fulfilled.

IV. POSSIBLE SOLUTIONS

As we can see in the examples, the "natural" way programmers familiar with the STL would take rewriting their code to

[1] `0` as an `unsigned long long` literal. This ensures that the largest representable integer numbers are used in the calculation.

a parallel version easily leads to fatal mistakes. Our goal is to warn them about the hidden preconditions, especially in the case of functor parameters of parallel algorithms.

There are three fundamentally different approaches to avoiding such errors in practice. The dynamic approach cannot avoid creating bad code but helps detect it in the testing phase. Such approaches were used to implement, e.g., *checked iterators* [14] and are also profoundly studied by others [15]. However, the dynamic approach is not the safest solution in the case of concurrent programs. As the actual execution may depend on various platform-specific and environmental factors, we cannot guarantee that the problematic code will be detected in the majority of the situations.

Programming language compilers make extensive checking of various syntactic and semantic issues. Today, we expect the compiler to warn us about passing wrong parameters to functions, conversions that may lose information, or even unused variables. However, checking semantic issues for library functions usually does not belong to this category. One of the rare examples is the parameters of the infamous `printf` and `scanf` C functions, where a mismatch between the format string and the actual parameter type happens too often and may lead to information leakage or buffer overflows. To enforce a compiler diagnostic, we have to map the semantic issue – using a non-commutative or non-associative functor as a parameter – to a type system error.

A third approach is to implement library-specific static analysers. Static analysis is a method which tries to detect code smells and other possible errors without running the actual program [16], [17]. Today various static analyser tools are available to check industry-size C++ projects [18]–[23]. Static analyser tools are not very strong at analysing parallel programs, but here we are in a bit easier situation. We do not need to check the correctness of the concurrent execution; it is enough to detect the incorrect parameters, i.e., to decide whether the functor passed was non-commutative or non-associative.

V. Our solution

We have chosen to generate compiler diagnostics in case of the misuse of Parallel STL. Our solution is based on introducing *tag types* and modifying the critical STL algorithms to require these types as parameters.

The use of tag types is not unknown in the C++ programming language. These are objects of types that may not have state (data members) but carry information by virtue of being instances of dedicated types themselves. Such types are the various `iterator_category` types – like `std::forward_iterator_tag` – or the `std::nothrow` of the allocating **`new`** expression, and the execution policies, too.

We created a wrapper library in the `pstl` namespace around the parallel algorithms of the standard C++ library declared in the `<algorithm>` and `<numeric>` headers. The main role of the wrapper library is to apply compile-time checking for the required criteria and then just forward the call to the original standard functions. In our wrapper library, the functions have the same signature as the standard functions, except that all parameters are declared as *forwarding (universal) references* [24]. As the wrapper functions could be *inlined* and declared as **`constexpr`**, the solution does not introduce any run-time overheads [25].

If a function is customised with a user-defined functor – e.g., a predicate – it must be wrapped by a wrapper class from our `pstl` library. Such wrapper classes are also optimised out during compilation. The wrapper classes are marked with the tag types, labelling the functors with the *commutative*, and *associative* properties, that the code does *not modify*, or does *not invalidate* the parameter. For convenient use, all tag types have a **`constexpr`** instance, similarly to the execution policies, which are the following: `pstl::associative`, `pstl::commutative`, `pstl::no_mod`, and `pstl::no_invalidate`.

An additional safety measure is that all of these functors should be **`const`**, excluding the possibility of some side effects, like modifying the elements, which – under concurrent execution – may lead to undefined behaviour. **`Const`**-ness is also checked by the compiler.

```
1  auto w1 = pstl::wrapper(std::plus<int>{},
2      pstl::associative, pstl::commutative);
3
4  int foo(int a, int b) { return a + b; }
5  auto w2 = pstl::wrapper(foo,
6      pstl::associative, pstl::commutative);
7
8  auto w3 = pstl::wrapper(
9      [](int a, int b) {return a + b; },
10     pstl::associative, pstl::commutative);
```

Listing 7: Examples for wrapper creation.

In listing 7, we see three examples of the creation of `pstl` wrappers. In line 1, `w1` is a wrapper around the standard library `plus` functor, labelled as *associative* and *commutative*. In line 5, `w2` is a wrapper around the function `foo` defined in line 4. We also label that as *associative* and *commutative*. Finally, in line 8, we label the `w3` wrapper the same way, encapsulating a lambda function returning the sum of its parameters. The effect of these three functors, and their wrappers, are the same.

```
1  auto sum = pstl::transform_reduce(
2    std::execution::par, v.begin(), v.end(), 0ULL,
3    /* Reduce    =*/ pstl::wrapper(
4          std::plus<int>{},
5          pstl::associative, pstl::commutative),
6    /* Transform =*/ pstl::wrapper(
7          [](auto v) { return v * v; },
8          pstl::associative, pstl::commutative)
9    );
```

Listing 8: Correct sum of squares, type checked.

In listing 8, we see the same example as in listing 6 for the correct sum of squares in a range, imple-

mented using our library. We can recognise the wrapper `pstl::transform_reduce` function from our library instead of the standard `std::transform_reduce`; and the last two parameters, which are wrapped and labelled with `pstl::associative` and `pstl::commutative`. Otherwise, the values and the interface are the same.

The call of the `pstl::transform_reduce` function is checked for whether the last two parameters had been labelled as associative and commutative. The compile-time checking is done via *concept-checking* methods. If the compilation passes, the generated code will just call `std::transform_reduce`, passing the first four parameters. For the last two parameters, `std::plus<int>{}` and the lambda will be extracted from the wrapper and passed as the fifth and sixth parameters.

If the compilation fails, however, the compiler emits a concise error message, as depicted in listing 9. The error message notifies the developer about the missing tags by emphasising that the passed functors *do not derive from* the *associative* tag. Unfortunately, as these messages are not controllable by library code, developers will have to consult the documentation as to how the error can be fixed.

```
auto sum = pstl::transform_reduce(
  std::execution::par, v.begin(), v.end(), 0ULL,
  // Note no pstl::wrapper applied to functors:
  std::plus<int>{}, std::multiplies<int>{});
```

```
error: no matching function for call to
    'transform_reduce(...)'
note: candidate 'T pstl::numeric::transform_reduce(
    ExecutionPolicy&&, ForwardIt1&&, ForwardIt1&&,
    T&&, ReduceOp&&, TransformOp&&)'
note: constraints not satisfied:
    required for the satisfaction of 'derived_from<
        ReduceOp, pstl::associative_tag>'
        [with ReduceOp = std::plus<int>]
    required for the satisfaction of 'derived_from<
        TransformOp, pstl::associative_tag>'
        [with TransformOp = std::multiplies<int>]
```

Listing 9: Incorrect invocation due to lacking wrappers and the associated compiler diagnostics.

The wrapper classes were designed using the *variadic mix-in* pattern, i.e., as class templates inheriting from their template parameters [26]. The first parameter is always the operation we wrap; the following parameters are an arbitrary number of tag types to label the operations. As the tag types are classes with an empty body, *empty base optimisation* [27] can be performed on them. Thus, the total size of the wrapper object will be the same as the wrapped operation object; the labelling tag types are optimised out. To construct the mix-in object, we use C++17 *Class Template Argument Deduction (CTAD)* with an explicit deduction guide. We see a similar construct in listing 10.

We must emphasise that no check is done whether the wrapped method is indeed associative and/or commutative. This is beyond the current possibilities of static analysis. However, the programmers **must** set these parameters, which serve as a warning for them to consider the situation and decreases the possibility of accidental misuse.

```
struct A {};
struct B {};

template <class Op, class... Ts>
struct Wrapper : public Ts ... // Mixin C++11
{
  Op op;
  Wrapper(Op o, Ts...) : op(o) {}
};

template <class Op, class... Ts> // CTAD C++17
Wrapper(Op, Ts...) -> Wrapper<Op, Ts...>;

int neg(int a) { return -a; }
int main()
{
  auto w = wrapper(foo, A{}, B{});
  int a = w.op(5);
  (void)a;  // a == -5
  static_assert(sizeof(w) == sizeof(w.op));
}
```

Listing 10: Variadic mix-in and empty base optimization.

VI. Future plans

Naturally, the actual checking of the functors passed as parameters to the parallel algorithms would be a stronger solution than the current one which just forces the programmer to set up some wrapper parameters. Due to the complexity of the problem and the required resources, we do not think an implementation as a compiler extension would be feasible for this. We rather see the possibility of using an external static analysis tool. Also, a static analyser tool can be introduced in a very conservative way: emit warnings only in cases where the violation of rules is very likely. Still, static analyser tools are usually based on heuristics, and *false positive* reports are inevitable.

However, we have a few possible starting points. We can trivially accept all commutative and associative standard library operations like `std::plus` as type-safe. We can also accept those user-defined functors and lambdæ which have no state and are using only standard library or user-defined operations already accepted as associative/commutative.

Not only the reduce-like algorithms and their required associative and commutative parameters are problematic in Parallel STL. The innocent-looking `std::find_if` algorithm may also lead to bogus behaviour [28]. The classic `std::find` algorithm from the STL looks for the first occurrence of a particular value. Its parallelisation is trivial: we split the interface into smaller ranges, execute the search in all of them, and then *left-fold* for the result. However, `find_if` is different: it looks for the first value which satisfies a specific *predicate* function. The problem arises when the predicate functor has a state. That state will be shared between threads, and its updates may cause race conditions. If we use locks, the overhead likely destroys the advantages of parallelisation.

Our idea is to develop an interface which separates the predicate into a thread-local and an inter-thread part. We can safely use the thread-local part in a concurrent way, and when

the threads are ready with their tasks, the final result can be computed by the inter-thread part of the predicate. Our implementation for this feature is under testing.

VII. Conclusion

Writing safe and efficient parallel programs is a challenging task. The C++ programming language is not an exception. One should decide how to split the data up for parallel execution, how many threads can be run efficiently on a specific target platform, and how the overall overhead of parallelisation – e.g., the starting of threads – compares to the gain of concurrent execution. Such decisions may depend on the data size, on the task we execute and on possible hidden platform specific information.

The *Standard Template Library (STL)* is one of the most important parts of the C++ Standard library. As an exemplar of the generic programming paradigm, the STL provides us (templated) containers (data structures), generic algorithms, and iterators to connect these domains. Since its release, the STL has proved to be flexible, extensible, and easy-to-use. It became part of the everyday toolset of all C++ programmers.

Parallel STL became part of the C++ standard library since C++17. It simplifies the parallel execution of STL-like functions. The new library was intentionally designed to have a similar interface to the original STL to ease its use. However, the use of Parallel STL revealed a few traps and pitfalls.

In this paper, we investigated some of the critical issues of Parallel STL, especially the requirements for using associative and commutative functors for certain algorithms. The violation of these contracts is currently not checked by the compiler and may lead to unspecified results. We suggested a new library framework with wrapper classes, which can label customisation points with the properties required by certain algorithms. The lack of these labels results in compile time errors; thus, the programmer is encouraged to pre-emptively handle the situation. While our solution is not "hard" in that no confirmation or analysis is done as to whether the wrapped method fulfils the properties announced by the labels, it helps to avoid issues coming from the programmer's improper knowledge or ignorance.

We implemented a prototype of the suggested framework to demonstrate its usability and effectiveness.

References

[1] H. Sutter *et al.*, "The free lunch is over: A fundamental turn toward concurrency in software," *Dr. Dobb's journal*, vol. 30, no. 3, pp. 202–210, 2005, visited 2022-10-14. [Online]. Available: http://gotw.ca/publications/concurrency-ddj.htm

[2] "ISO/IEC 14882:2020 – Programming languages C++," visited 2022-05-20. [Online]. Available: https://www.iso.org/standard/79358.html

[3] "Working draft – Standard for Programming language C++," visited 2022-05-20. [Online]. Available: http://eel.is/c++draft

[4] D. van de Voorde, N. M. Josuttis, and D. Gregor, *C++ Templates: The Complete Guide (2nd Edition)*, 2nd ed. Addison-Wesley Professional, 09 2017. [Online]. Available: http://tmplbook.com

[5] D. R. Musser and A. A. Stepanov, "Algorithm-oriented generic libraries," *Software: Practice and Experience*, vol. 24, no. 7, pp. 623–642, 1994. [Online]. Available: http://stepanovpapers.com/musser94algorithmoriented.pdf

[6] J. Coplien, "Multiparadigm design and implementation in C++," in *TOOLS Europe 1999: 29th International Conference on Technology of Object-Oriented Languages and Systems, 7-10 June 1999, Nancy, France*. IEEE Computer Society, 1999, p. 408. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/TOOLS.1999.10007

[7] S. Krishnamurthi, M. Felleisen, and D. P. Friedman, "Synthesizing object-oriented and functional design to promote re-use," in *ECOOP'98 - Object-Oriented Programming, 12th European Conference, Brussels, Belgium, July 20-24, 1998, Proceedings*, ser. Lecture Notes in Computer Science, E. Jul, Ed., vol. 1445. Springer, 1998, pp. 91–113. [Online]. Available: http://doi.org/10.1007/BFb0054088

[8] B. Stroustrup, "C++11 - the new ISO C++ standard," visited 2022-05-07. [Online]. Available: http://stroustrup.com/C++11FAQ.html

[9] J. Reinders, *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. "O'Reilly Media, Inc.", 2007.

[10] N. M. Josuttis, "C++17: The biggest traps – [C++ on Sea 2019]," visited 2022-05-07. [Online]. Available: http://youtube.com/watch?v=mAZyaAo3M70&t=3975

[11] "CppReference: `reduce`," visited 2022-05-07. [Online]. Available: http://en.cppreference.com/w/cpp/algorithm/reduce

[12] "CppReference: `transform_reduce`," visited 2022-05-07. [Online]. Available: http://en.cppreference.com/w/cpp/algorithm/transform_reduce

[13] S. Khezr and N. J. Navimipour, "MapReduce and its applications, challenges, and architecture: a comprehensive review and directions for future research," *Journal of Grid Computing*, vol. 15, pp. 295–321, 2017. [Online]. Available: http://link.springer.com/article/10.1007/s10723-017-9408-0

[14] Microsoft Corporation, "Checked Iterators," visited 2022-05-07. [Online]. Available: http://learn.microsoft.com/en-us/cpp/standard-library/checked-iterators

[15] N. Pataki, "Safe iterator framework for the C++ Standard Template Library," *Acta Electrotechnica et Informatica*, vol. 12, no. 1, p. 17, 2012. [Online]. Available: http://aei.tuke.sk/papers/2012/1/03_Pataki.pdf

[16] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: Using static analysis to find bugs in the real world," *Commun. ACM*, vol. 53, no. 2, pp. 66–75, Feb. 2010. [Online]. Available: http://doi.acm.org/10.1145/1646353.1646374

[17] S. Hallem, B. Chelf, Y. Xie, and D. Engler, "A system and language for building system-specific, static analyses," in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, ser. PLDI '02. New York, NY, USA: ACM, 2002, pp. 69–82. [Online]. Available: http://doi.acm.org/10.1145/512529.512539

[18] D. Marjamäki, "CppCheck: a tool for static C/C++ code analysis," 2013, visited 2022-10-14. [Online]. Available: http://cppcheck.sourceforge.net

[19] LLVM Foundation, "Clang Static Analyzer: Checker Developer Manual," 2019, visited 2019-02-28. [Online]. Available: http://clang-analyzer.llvm.org/checker_dev_manual.html

[20] A. Zaks and J. Rose, "Building a checker in 24 hours," 2012, visited 2019-02-22. [Online]. Available: http://youtube.com/watch?v=kdxlsP5QVPw

[21] Ericsson AB., "CodeChecker," 2019, visited 2019-02-28. [Online]. Available: http://github.com/Ericsson/CodeChecker

[22] Synopsys, "Coverity," 2019, visited 2019-02-28. [Online]. Available: http://scan.coverity.com

[23] Roguewave, "Klocwork," 2019, visited 2019-02-28. [Online]. Available: http://roguewave.com/products-services/klocwork

[24] S. Meyers, *Effective Modern C++: 42 specific ways to improve your use of C++11 and C++14*. O'Reilly, 2014.

[25] B. Stroustrup, "Evolving a language in and for the real world: C++ 1991–2006," *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, 2007. [Online]. Available: http://dl.acm.org/doi/10.1145/1238844.1238848

[26] D. P. Gregor and J. Järvi, "Variadic templates for C++0x," *J. Object Technol.*, vol. 7, pp. 31–51, 2008. [Online]. Available: http://jot.fm/issues/issue_2008_02/article2.pdf

[27] T. Ramananandro, "Machine-checked object layout for C++ multiple inheritance with empty-base optimization," Tech. Rep., 2010, visited 2022-10-14. [Online]. Available: http://gallium.inria.fr/~tramanan/cpp/object-layout/techrep.pdf

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