# Multicore C++ Standard Template Library with C++0x

Zalán Szűgyi, Márk Török, Norbert Pataki and Tamás Kozsik

*Department of Programming Languages and Compilers, Eötvös Loránd University*
*Pázmány Péter sétány 1/C H-1117 Budapest, Hungary*

**Abstract.** Nowadays, one of the most important challenges in the programming is the efficient usage of multicore processors. Many new programming languages and libraries support multicore programming. C++0x, the proposal of the next standard of C++ also supports multithreading at low level.

In this paper we argue for some extensions of C++ Standard Template Library based on the features of C++0x. These extensions enhance the standard library to be more powerful in the multicore realm. In this paper we deal with the functors and lambda expressions that are a major extension in the language. We present a technique to write effective pipelines. Speculative functors aim at the effective evaluation of composite functors. Algorithms are overloaded on the associativity of lambda expressions as well.

**Keywords:** C++, C++0x, STL, multicore
**PACS:** 68N19

## INTRODUCTION

The new standard proposal of C++ programming language supports parallel computation. Strictly speaking, it supports only low level constructs. Although several libraries are available that provide high level parallelization tools for C++, there are numerous occasions when the usage of these libraries is not beneficial [7]. These libraries are complex and robust, their structure can be different from the other ones and they have own coding style. Thus if the programmer wants to use one of these libraries first he need to spend a lot of time to get familiar with it to be able to use it properly.

The *C++ Standard Template Library* (STL) was developed by *generic programming* approach [2]. In this way containers are defined as class templates and many algorithms can be implemented as function templates. Furthermore, algorithms are implemented in a container-independent way, so one can use them with different containers [6]. C++ STL is widely-used inasmuch as it is a very handy, standard C++ library that contains beneficial containers (like list, vector, map, etc.), a large number of algorithms (like sort, find, count, etc.) among other utilities.

The STL was designed to be extensible. We can add new containers that can work together with the existing algorithms. On the other hand, we can extend the set of algorithms with a new one that can work together with the existing containers. Iterators bridge the gap between containers and algorithms. The expression problem [9] is solved with this approach. STL also includes adaptor types which transform standard elements of the library for a different functionality.

*Functor objects* make STL more flexible as they enable the execution of user-defined code parts inside the library [5]. Basically, functors are usually simple classes with an `operator()`. Inside the library `operator()`s are called to execute user-defined code snippets. This can called a function via pointer to functions or an actual `operator()` in a class. Functors are widely used in the STL inasmuch as they can be inlined by the compilers and they cause no runtime overhead in contrast to function pointers. Moreover, in case of functors extra parameters can be passed to the code snippets via constructor calls.

Functors can be used in various roles: they can define predicates when searching or counting elements, they can define comparison for sorting elements and properly searching, they can define operations to be executed on elements.

C++0x, the proposal of the next standard of C++, includes a new feature called *lambda functions* or *lambda expressions* [4]. Lambda expressions are able to express the functionality of a function call operator without writing explicit functor types. The call of algorithm and the inner logic does not being separated with this technique. Lambda expressions can be considered as locally defined functors. The experimental compilers generate functor types from lambda expressions. Our solution also supports lambda expressions.

It is frequently advised that prefer the standard library to other ones. C++ programmers are familiar with the STL. Unfortunately, whereas the STL is pre-eminent in a sequential realm, it is not aware of multicore environment [8].

In this paper we present our results to provide high level parallelization by extending the STL. In our research we made an effort to highly reuse the existing utilities of the STL. Thus the programmers, who are familiar with STL can easily adopt our extensions.

The rest of this paper is organized as follows. In section *High Level Parallel Tools* different functor-related techniques are implemented to make STL a more advanced, considerable multicore library. One of technique presents how a pipeline can be effectively implemented with functors. Another one evaluates composite predicates in a multithreaded way. A technique is also presented that is able to choose a faster evaluation if we use an associative computation on huge amount of data. Finally, this paper concludes.

## HIGH LEVEL PARALLEL TOOLS

### Pipeline

The algorithm `for_each` of STL applies a functor to each element in the given range. If the functor is defined in a special way, it is able to represent the stages of a pipeline, while the algorithm `for_each` itself feeds it with data.

We extended the STL with a new functor adaptor to help the programmer to create this kind of special functor. The name of this new functor adaptor is `parallel_compose`, which adapts two functors to a functor composition, and processes them in the following way: Let suppose the `parallel_compose` adapts the functors `g` and `f`, and the input parameter is `x`. The result of the computation is $f(g(x))$. However, after $g(x)$ is processed, the result value is passed to the functor `f` in a new thread. Hereby while the functor `f` computes its result, the functor `g` can start to process the next input data.

One of the input functor of `parallel_compose` can be another `parallel_compose` thus it is possible to create arbitrary long functor composition.

This way the algorithm `for_each` acts as a pipeline. The simple functors in a functor composition act as a stages of the pipeline, while the algorithm `for_each` feeds it with data defined by the input range.

The example below presents the way, how to apply our pipeline solution in a classical image processing task [3]. The image processing contains three steps: transformation, rasterization and pixel processing. These steps will be the stages of the pipeline, and the input data are triangles. The hereinafter example demonstrates our approach, but it highly simplifies the problem. In a real life the image processing is a more complex process, and the stages can be split into more sub stages to improve the performance.

```
struct transformation {
   triangle operator()(triangle value) { /*...*/}
};
struct rasterization {
   triangle operator()(triangle value) { /*...*/}
};
struct pixel_processing {
   triangle operator()(triangle value) { /*...*/}
};
for_each(input_iterator_begin, input_iterator_end,
         pcompose(transformation(),
                  pcompose(rasterization(),pixel_processing()));
```

The `input_iterator_begin` and `input_iterator_end` are two iterators defining the input range of triangles. The stages are functors thus they have to overload the `operator()` which do the computation process. The `pcompose` is a helper function that creates `parallel_compose` functor object by its arguments and returns it. Helper functions simplify the creation of functors, thus they are very common in the STL because the C++ compiler can deduce the template arguments by the type of actual parameters.

### Speculative Functors

There are several algorithms in the STL that takes a predicate functor as argument to decide whether an element is must be processed. The predicate is a unary functor (its `operator()` has exactly one argument) that returns a boolean value. If the predicate returns true for a given argument, the algorithm deals with that. The names of these algorithms have an `_if` postfix, such as: `find_if`, `count_if`, `remove_if`, `replace_if`, etc.

In many cases the predicates are very complex. As the predicate is a logical condition it is usually constructed by functors composed by `logical_and` or `logical_or`.

If the subexpressions composed by `logical_and` are complex it is worth to evaluate them parallelly. The more complex the subexpression is, the more speed-up we can achieve. We introduce a new functor adaptor called `speculative_logical_and` which can evaluate the subexpressions in different threads. If the first thread computes the result of the first subexpression we check it. If the result is `false` we kill the second thread, or drop its result if it is terminated already. Otherwise we wait for the result of second thread and evaluates if both are true.

Technically the `speculative_logical_and` is a unary functor that composes the predicates `f` and `g` in the following way: $f(x)\&\&g(x)$ where `x` is the input parameter.

The `speculative_logical_or` functor similar to `speculative_logical_or`. The only difference is it kills the second thread if the result of the first condition is `true`.

The example above shows the usage of our solution. There is a range of log entries which contains several fields, such as: timestamp, priority, user name, log message. We would like to find those ones which are created on 20.03.2011 and the message fits to a given regular expression.

```
struct is_proper_date {
    bool operator()(log_entry le) { /*compute if le is created on 20.03.2011*/ }
};
struct has_proper_message {
    bool operator()(log_entry le) { /*compute if message fits to a regex*/ }
};
std::find_if(input_iterator_begin, input_iterator_end,
              speculative_and(is_proper_date(), has_proper_message()));
```

The `input_iterator_begin` and `input_iterator_end` are two iterators defining the input range of log entries. The `speculative_and` is a helper function to create `speculative_logical_and` functor. This helper function behaves similarly to the helper function `pcompose` described in a previous section.

This solution is efficient to use when the subexpressions are complex.

## Associative Functors

In this section we present an approach to compute an associative operation on huge amount of data effectively. We develop the STL's `accumulate` algorithm to be as effective as possible.

By default the algorithm `accumulate` computes the sum of the elements of a given range. However, we can customize the algorithm defining an own operation instead of addition. The operation is defined by a binary functor (it has two arguments) and it is an argument of accumulate. If the operation is associative we can apply the optimized version of accumulate algorithm.

A technique is presented to overload algorithms on the associativity of their functor in [8]. This technique includes a trait type called *functor traits*. This type is similar to STL's iterator traits, functor traits consists of some typedefs. It is possible to overload algorithms on the associativity of the functor based on these typedefs.

Our main goal was to support the new standard proposal, where lambda expressions can replace functors. However, it is not possible to define functor traits in lambda expression. We need to denote that operation is associative in a different way.

In our solution an extra argument of lambda expression, which has a special type, called `associative`, denotes that the operation is associative. Our implementation of algorithm `accumulate` is able to detect whether a given lambda expression has that extra argument or not. If the lambda expression is associative the optimized algorithm [8] is chosen otherwise we take the original one.

The example below shows the way we determine if the lambda expression has the extra argument.

```
template<typename InputIterator, typename T, typename BinFunctor>
T accumulate(InputIterator first, InputIterator last, T init, BinFunctor bf) {
    typedef T (BinFunctor::*funtype) (T, T, associative) const;
    if(std::is_same<decltype(&BinFunctor::operator()), funtype>::value)
```

```
        return optimized_accumulate(first, last, init, bf);
    else
        return std::accumulate(first, last, init, bf);
}
```

The BinFunctor template type refers to the lambda expression, while T refers to the elements of the input range. The static field value of template type is_same is true if the its two template arguments are same. We instantiate it with a type of the member function pointer of the operator() which has that extra argument and the type of the member function pointer of operator() of the current functor. If the lambda expression has the extra argument, the two types are the same. The value is computed at compile time. As C++ template metaprograms run during compilation [1], the if statement in the example can be replaced by a template metaprogram to make our solution more effective. That way the selection of the proper algorithm is done at compile-time. This solution is backward compatible to the original functor usage.

The example below shows the usage of our solution to calculate the product of the input range of integers.

```
accumulate(input_iterator_begin, input_iterator_end, 1,
           [](int a, int b) { return a * b; });
accumulate(input_iterator_begin, input_iterator_end, 1,
           [](int a, int b, associative) { return a * b; });
```

The first function call summarizes the input range in conventional way, while the second one applies the more effective algorithm which exploits the associativity.

## CONCLUSION

Multicore programming is an interesting new way of programming. Although the current C++ programming language does not contain any construct to write multithreaded programs, many extensions and libraries can be used. The next standard of C++ includes constructs for parallel program execution. Unfortunately, these constructs are at low level.

In this paper we argue for higher lever constructs at level of the preferred C++ Standard Template Library. We implemented special functors and adaptors that support different kinds of evaluations in a multithreaded way. Associative operations are taken advantage of, thus STL algorithms are overloaded on their operation's associativity, even it is defined as functor or lambda expression.

## ACKNOWLEDGMENTS

The Project is supported by the European Union and co-financed by the European Social Fund (grant agreement no. TÁMOP 4.2.1./B-09/1/KMR-2010-0003).

## REFERENCES

1. D. Abrahams, A. Gurtovoy: *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*, Addison-Wesley, Reading, MA., 2004.
2. A. Alexandrescu, *Modern C++ Design*, Addison-Wesley, Reading, MA., 2001.
3. J. D. Foley, A. van Dam, S. K. Feiner, J. F. Hughes: *Computer Graphics, Principles and Practice*, Addison-Wesley, Reading, MA., 1990.
4. J. Järvi, and J. Freeman, "C++ Lambda Expressions and Closures",in *Sci. Comput. Programming* **75(9)**, pp. 762–772.
5. S. Meyers, *Effective STL - 50 Specific Specific Ways to Improve Your Use of the Standard Template Library*, Addison-Wesley, Reading, MA., 2001.
6. B. Stroustrup, *The C++ Programming Language (Special Edition)*, Addison-Wesley, Reading, MA., 2000.
7. Z. Szűgyi, and N. Pataki, "A More Efficient and Type-Safe Version of FastFlow", in *Proceedings of Workshop on Generative Programming 2011*, pp. 24–37.
8. Z. Szűgyi, and M. Török, and N. Pataki, "Towards a Multicore C++ Standard Template Library", in *Proceedings of Workshop on Generative Programming 2011*, pp. 38–48.
9. M. Torgersen, "The Expression Problem Revisited – Four New Solutions Using Generics", in *Proceedings of European Conference on Object-Oriented Programming (ECOOP) 2004, Lecture Notes in Comput. Sci.* **3086**, pp. 123–143.