DOI: 10.2298/CSIS111231027S

High-level Multicore Programming with C++11

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
{lupin,tmark,patakino,kto}@caesar.elte.hu

Abstract. Nowadays, one of the most important challenges in programming is the efficient usage of multicore processors. All modern programming languages support multicore programming at native or library level. C++11, the next standard of the C++ programming language, also supports multithreading at a low level. In this paper we argue for some extensions of the C++ Standard Template Library based on the features of C++11. These extensions enhance the standard library to be more powerful in the multicore realm. Our approach is based on functors and lambda expressions, which are major extensions in the language. We contribute three case studies: how to efficiently compose functors in pipelines, how to evaluate boolean operators in parallel, and how to efficiently accumulate over associative functors.

Keywords: multicore programming, C++.

1. Introduction

The new standard of the C++ programming language supports parallel computation. Strictly speaking, it supports only low level constructs [20]. 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 [2, 5, 17]. These libraries are complex and robust, their structure can be different from that of other ones and they have their own coding styles. Thus, if the programmer wants to use one of these libraries, first she needs to spend a lot of time to get familiar with it to be able to use it properly [16]. Our goal was to extend the standard library of C++ to support high level parallelization techniques. In our solution we made an effort to extend it only slightly, and to allow simple usage of our library for a programmer familiar with the STL.

The *C++ Standard Template Library* (STL) was developed by *generic programming* approach [3]. 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 [14]. C++ STL is widely-used inasmuch as it is a very handy, standard C++ library that contains useful 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 [4]. 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 which can work together with the existing containers. Iterators bridge the gap between containers and algorithms [11]. The expression problem [21] is solved with this approach. STL also includes adaptor types which transform standard elements of the library for a different functionality [12]. Adaptors can modify the interface of a container, transform streams into iterators, modify the behavior of functors etc.

Functor objects make STL more flexible as they enable the execution of user-defined code parts inside the library [13]. Basically, functors are usually simple classes with an `operator()`. Inside the library `operator()`s are called to execute user-defined code snippets. This can call 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 property searching, they can define operations to be executed on elements of collections.

C++11, the next standard of C++, includes a new feature called *lambda functions* or *lambda expressions* [9]. Lambda expressions are able to express the functionality of a function call operator without writing explicit functor types. The call of an algorithm and the inner logic is not 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 one should prefer standard library to other ones. C++ programmers are familiar with the STL. Unfortunately, whereas the STL is preeminent in a sequential realm, it is not aware of multicore environment [19].

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. Different functor-related techniques are implemented to make STL a more advanced, multicore supporting library. Using our library those programmers who are familiar with STL can easily adopt our extensions, without the need to spend much time to learn it. The source code of the library can be downloaded from the `http://kp.elte.hu/STLpar` URL.

The rest of this paper is organized as follows. Section 2 shows how a pipeline can be effectively implemented with functors. Section 3 describes the evaluation of composite predicates in a multithreaded way. A solution which is able to select a faster evaluation technique if we use an associative computation on huge amount of data is presented in section 4. Finally, section 5 concludes the paper.

2. Pipeline

The algorithm `for_each` of STL applies a functor to each element in a 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 create this kind of special functor called `parallel_compose`. This functor adapts two unary functors to a functor composition, and processes them in the following way. Assume that `parallel_compose` adapts the functors `f` and `g`, and the input parameter is `x`. The result of the computation is $g(f(x))$. However, after $f(x)$ is processed, the result value is passed to the functor `g` in a new thread. Hereby while the functor `g` computes its result, the functor `f` can start to process the next input data.

One input of `parallel_compose` can be an 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.

2.1. Implementation details

The core of our implementation is the `parallel_compose` functor adaptor. Its constructor receives two unary functors that are playing role on a functor composition. Then it wraps the second argument by a thread wrapper class (described later) and starts it to run in a new thread. However, the algorithms of STL can freely copy functors, by definition. This behavior is ineligible for us because we do not want to copy a thread. To avoid this situation we allocate the thread dynamically, and store it in a type `shared_ptr`, which is a smart pointer, provided by the new standard of C++ [15]. This ensures that all the temporarily copied `parallel_compose` functors refer to the same thread, and the memory will be deallocated automatically.

The `operator()` of `parallel_compose` invokes its first unary functor to compute the first member of the composition, and sends the result to the thread wrapper.

The member function `join` blocks the execution until the last element is applied on all the stages. After that it destructs the pipeline. When there is more than two stages in the pipeline, – i.e. `parallel_compose` functors are composed in a chain –, `join` must be invoked for all `parallel_compose` functors recursively. The naive method that invokes the `join` member function of the second argument does not work because the second argument of the last `parallel_compose` is a different unary functor, see the example in subsection 2.2, and hence it would cause a compilation error.

We applied the SFINAE (Substitution Failure Is Not An Error) technique [22] to solve this problem. We defined a new type trait to check whether the given type is a `parallel_compose`. Because type traits define a compile-time template-based interface to query or modify the properties of types [10], we can cus-

tomize our code for a given type without any run-time overhead. Then we created two auxiliary join functions that can be seen below:

```
template<typename T>
void join_aux(T& t,
              typename std::enable_if<
                  is_parallel_compose<T>::value,
                  int>::type n = 0)
{
  t.join();
}

template<typename T>
void join_aux(const T&)
{
}
```

The `enable_if` is a template metafunction provided by the new standard. If its first template argument is `true`, it has a public member typedef `type`, equal to its second template argument; otherwise, there is no member typedef. This metafunction is used to conditionally remove the first function `join_aux` from overload resolution based on type traits.

We invoke this `join_aux` function in the following way:

```
void join()
{
  join_aux(second_argument);
}
```

If the `second_argument` itself is a `parallel_compose` functor, then our type trait returns with true, and the metafunction `enable_if` has typedef `type`, thus both definitions of `join_aux` are valid. Since the `second_argument` is not a constant, the first `join_aux` will be selected by the compiler, thus the `join` is invoked on the next `parallel_compose` functor in a chain. However, if the `second_argument` is a different functor, then there is no typedef `type` inside of `enable_if`. This means the definition of the first `join_aux` is invalid, thus it is removed from the overload resolution. In this case only the second `join_aux` is left – the one which does not invoke any `join` –, thus that one will be selected.

The thread wrapper class wraps a unary functor, and runs it in a new thread. It has two main member functions: the `receive` and the `operator()`. The `parallel_compose` sends the data to a thread via member function `receive`, and the `operator()` performs the computation in the new thread. The key parts of these two member functions can be seen below:

```
void receive(const argument_type& a)
{
```

```
  //...
  data_lock();
  data = a;
  data_ready=true;
  //...
}

void operator()()
{
  //...
  while(run)
  {
    wait_for(data_ready);
    if(run) stored_functor(data);
    data_ready = false;
    data_unlock();
  }
  //...
}
```

The `data_lock` is used to prevent the previous stage to overwrite the data before it is computed. It will be only unlocked when the computation of current data is finished. The main loop of `operator()` runs while the logical variable `run` is true. It waits for the data, then performs the computation, and finally unlocks the semaphore to be able to receive the next one.

This class has a `kill` method, which terminates the thread. First it sets `run` to false, and `data_ready` to true. The second one is necessary because the `operator()` may be waiting for `data_ready`. But, because `run` is false, there will be no false computation.

2.2. Example

The example below illustrates the way to apply our pipeline solution in a classical image processing task [6]. The image processing contains three steps: transformation, rasterization and pixel processing. These steps will be the stages of the pipeline, and the input data is a range of triangles. The hereinafter example demonstrates our approach, but it highly simplifies the problem. In real life image processing is a more complex process, and the stages can be split into more substages to improve performance [23].

```
struct transformation
{
  triangle operator()(const triangle& value)
  {
    // ...
  }
```

```
};

struct rasterization
{
  triangle operator()(const triangle& value)
  {
    // ...
  }
};

struct pixel_processing
{
  triangle operator()(const triangle& value)
  {
    // ...
  }
};

for_each(input_iterator_begin,
         input_iterator_end,
         pcompose(transformation(),
                  pcompose(rasterization(),
                           pixel_processing()))
        ).join();
```

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 performs the computation. The `pcompose` is a helper function that creates a `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 (e.g. `std::make_pair`). The algorithm `for_each` returns with the functor. Thus the last function invocation calls the `join` method of the functor. This synchronization step waits for the pipeline to finish the computation.

3. Speculative Functors

There are several algorithms in the STL that take a predicate functor as argument to decide whether an element 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 element, the algorithm will deal with that element. 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 often constructed from functors composed by `logical_and` or `logical_or`.

If the subexpressions composed by `logical_and` are complex, it might worth to evaluate them in parallel. 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 separate threads. If one thread computes the result of its subexpression, we check whether it is `false`. If so, we got the result – thus, we kill the other thread, or drop its result if it is terminated already. Otherwise we wait for the result of the other thread and use both results.

3.1. Implementation details

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_and` receives the predicates in its constructor, which wraps them with a thread wrapper class to ensure they run in separate threads.

The work is done by the `operator()`. The core of its implementation can be seen below:

```
return_type operator()(const argument_type& a)
{
  compute_in_new_thread(f,a);
  compute_in_new_thread(g,a);

  wait_for(impl->has_result_f || impl->has_result_g);

  if(impl->has_result_f)
  {
    if(impl->result_f == false)
    {
      kill(g);
      return false;
    }
    else
    {
      wait_for(impl->has_result_g);
      return impl->result_g;
    }
  }
  else
  {
```

```
    if(impl->result_g == false)
    {
      kill(f);
      return false;
    }
    else
    {
      wait_for(impl->has_result_f);
      return impl->result_f;
    }
  }
}
```

The members of `speculative_logical_and` functor are put into an implementation class and the variable `impl` – a shared pointer pointing to it. This solution ensures that the functor can be copied by the STL, and that all the copies refer to the same implementation. `impl` is added to the thread wrappers, thus the threads can store their results into that. After one thread computes the result it sets its `has_result` variable to true, indicating to the main thread that the data is ready.

The `speculative_logical_and` waits until one thread is ready and checks the result. If it is `false`, the whole result is `false`, thus the other thread can be killed, and `false` will be returned. Otherwise `speculative_logical_and` waits for the result of the other thread, that value will be returned. (That way the first argument of *logical and* is true, thus the result depends on the second argument.)

In practice, `speculative_logical_or` behaves similarly. The only difference is that it kills the slower thread if the result of the faster one is `true`.

3.2. Example

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 entries which were created on 20.03.2011 and the message fits a given regular expression.

```
struct log_entry
{
  std::string username;
  std::string message;
  time_t      timestamp;
  int         priority;
  // ...
};

struct is_proper_date
```

```
{
  bool operator()(const log_entry& le)
  {
    /*compute if le is created on 20.03.2011*/
  }
};

struct has_proper_message
{
  bool operator()(const 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 the previous section.

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

4. Associative Functors

In this section we present an approach to compute an associative operation on a huge amount of data effectively. We improve the `accumulate` algorithm of STL to be as effective as possible [8].

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` [7]. If the operation is associative, we can apply the optimized version of the accumulate algorithm.

A technique is presented to overload algorithms on the associativity of their functor in [19]. This technique includes a trait type called *functor traits*. This type is similar to the iterator traits of STL; functor traits consist of some typedefs. It is possible to overload algorithms on the associativity of the functor based on these typedefs [18].

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 expressions. We need to denote that an operation is associative in a different way.

4.1. Implementation details

In our solution an extra argument of a 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 [19] is chosen – otherwise we take the original one.

The example below shows the way we determine if the lambda expression has that extra argument. In this section we suppose that the range is defined by a pair of *random access iterators*. STL algorithms can be overloaded on iterator category easily [18].

```
template< typename RandomAccesIterator,
          typename T,
          typename BinFunctor>
T accumulate( RandomAccesIterator first,
              RandomAccesIterator last,
              T init,
              BinFunctor bf )
{
  typedef
    T (BinFunctor::*funtype) (T, T, associative) const;

  if( std::is_same< decltype(&BinFunctor::operator()),
                    funtype>::value )
  {
    return associative_accumulate(first, last, init,
      std::bind(bf, std::_1, std::_2, associative());
  }
  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 efficient. That way the selection of the proper algorithm is done at compile-time. When

the `accumulate` is instantiated by an associative functor, that functor technically is a ternary functor, – its third argument refers to the associativity. That case we need to transform it into a binary functor. The `std::bind` does this work, binding a dummy value to the third arguments. This solution is backward compatible to the original functor usage.

The more efficient version of the algorithm uses the following functor for the computation in a distributed way:

```
template <typename Iterator, typename BinFunctor>
struct Accumulate
{
  void operator()(
    Iterator first,
    Iterator last,
    typename
      std::iterator_traits<Iterator>::pointer p )
  {
    typename std::iterator_traits<Iterator>
      ::difference_type diff =
         last - first;

    if ( 2 == diff )
    {
      *p = BinFunctor()( *p, *first );
      *p = BinFunctor()( *p, *(first + 1 ) );
    }
    else if ( 1 == diff )
    {
      *p = BinFunctor()( *p, *first );
    }
    else
    {
      typename
        std::iterator_traits<Iterator>::pointer p1 =
        new std::iterator_traits<Iterator>::value_type;

      typename
        std::iterator_traits<Iterator>::pointer p2 =
        new std::iterator_traits<Iterator>::value_type;

      std::thread t1(Accumulate(),first,first+diff/2,p1);
      std::thread t2(Accumulate(),first+diff/2,last,p2);
      t1.join();
      t2.join();
      *p = BinFunctor()( *p, *p1 );
      *p = BinFunctor()( *p, *p2 );
```

```
      delete p1;
      delete p2;
    }
  }
};
```

If the range has only 1 or 2 elements, the functor calculates the associative operation, otherwise it divides the input range into two smaller ranges and starts the calculation of smaller ranges in separate threads.

The `associative_accumulate` initializes the shared data and starts the parallel computation:

```
template< typename RandomAccessIterator,
          typename T,
          typename BinFunctor>
T associative_accumulate( RandomAccessIterator first,
                          RandomAccessIterator last,
                          T init,
                          BinFunctor bf )
{
  typename
    std::iterator_traits<RandomAccessIterator>::
      pointer p = new T( init );

  std::thread s( Accumulate<RandomAccessIterator,
                            BinFunctor>(),
                 first,
                 last,
                 p );
  s.join();
  T result = *p;
  delete p;
  return result;
}
```

4.2. Example

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.

4.3. Threshold

A more sophisticated and more effective implementation of the evaluation of associative operation uses a threshold parameter. This parameter defines how many elements in the range require the evaluation in a new separate thread. This parameter highly depends on the characteristic of the problem.

As the previous subsection presents, the user may not know that different implementation strategies are available according to the defined operation. However, the user just states that his own operation is associative. How the threshold parameter can be defined by user?

Two different approaches are discussed: if the operation is defined by a functor type or a lambda function.

A functor type can contain special member variables and member function that can be used by the `accumulate`. For convenience, an associative operation base type can be defined. This base type contains the `associative` typedef and the default threshold value. The user has to create a subtype, and he is able to override the threshold value. If the functor type is not subtype of the associative operation base type, but is an associative operation, the previous implementation works.

Lambda expressions cannot contain member variables and member functions. The compiler generates a simple functor class from the definition of lambda, but the generated functor is unavailable for extension. However, the possibility of lambda-defined threshold needs extensive inspection.

However, the threshold value does not belong to the definition of the associative operation from the view of modularity. It belongs to the `accumulate`. This means that `accumulate` has an extra parameter which defines the threshold value. This argument may be defined if the operation is associative. Since associativity is a compile-time information, compilation diagnostics can be emitted if the operation is not associative and threshold parameter is given by the user. If the operation is associative but no threshold is defined, the previous code does work. This scenario does not depend on if the operation is defined by functor or lambda function. Our future work includes the detailed implementation of this approach.

There are other approaches to solve this problem, but we reject them. One of them is that the `associative` type contains the threshold value. This can be a static value which is problematic from the view of parallelism. In the other one, the threshold value is a template argument or a member set by its constructor. Unfortunately, this makes the invocation of algorithm hard to maintain and the algorithm cannot obtain this value effectively.

5. Conclusion

Multicore programming is an interesting new way of programming. Although the current C++ programming language contains no constructs to write multithreaded programs, extensions and libraries can still be used. The next stan-

dard of C++ includes constructs for parallel program execution. Unfortunately, these constructs are at a low level.

In this paper we argue for higher level constructs – ones at the level of the widely used C++ Standard Template Library. We implemented special functors and adaptors which support different kinds of evaluation in a multithreaded way.

- We can build up a pipeline of computations using a functor combinator.
- We can make use of speculative parallelism in the case of complex predicates.
- We can take advantage of associative operations; STL algorithms can be overloaded on their operation's associativity, even if the operation is defined as a lambda expression.

A programmer familiar with the STL can easily adopt our library.

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. Abrahams, D., Gurtovoy, A.: C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond. Addison-Wesley, Reading, MA., USA (2004)
2. Aldinucci, M., Ruggieri, S., Torquati, M.: Porting decision tree algorithms to multicore using FastFlow. In: Proceedings of European Conference in Machine Learning and Knowledge Discovery in Databases (ECML PKDD), Lecture Notes in Computer Science, vol. 6321, pp. 7–23. Springer-Verlag, Berlin Heidelberg New York (2010)
3. Alexandrescu, A.: Modern C++ Design. Addison-Wesley, Reading, MA., USA (2001)
4. Austern, M.H.: Generic Programming and the STL: Using and Extending the C++ Standard Template Library. Addison–Wesley, Reading, MA., USA (1998)
5. Dagum, L., Menon, R.: Openmp: An industry-standard API for shared-memory programming. IEEE Computational Science and Engineering 5, 46–55 (1998)
6. Foley, J.D., van Dam, A., Feiner, S.K., Hughes, J.F.: Computer Graphics, Principles and Practice. Addison-Wesley, Reading, MA., USA (1990)
7. Frigo, M., Halpern, P., Leiserson, C.E., Lewin-Berlin, S.: Reducers and other cilk++ hyperobjects. In: Proceedings of Symposium on Parallel Algorithms and Architectures (SPAA). pp. 79–90 (2009)
8. Gottschling, P., Lumsdaine, A.: Integrating semantics and compilation: using C++ concepts to develop robust and efficient reusable libraries. In: Proceedings of the 7th international conference on Generative programming and component engineering, GPCE 2008. pp. 67–76 (2008)
9. Järvi, J., Freeman, J.: C++ lambda expressions and closures. Science of Computer Programming 75(9), 762–772 (2010)
10. Kalev, D.: The type traits library, [Online]. Available: http://www.informit.com/guides/content.aspx?g=cplusplus&seqNum=276
11. Kozsik, T., Pataki, N., Szűgyi, Z.: C++ Standard Template Library by infinite iterators. Annales Mathematicae et Informaticae 38, 75–86 (2011)

12. Matsuda, M., Sato, M., Ishikawa, Y.: Parallel array class implementation using C++ STL adaptors. In: Proceedings of the Scientific Computing in Object-Oriented Parallel Environments, Lecture Notes in Computer Science, vol. 1343, pp. 113–120. Springer-Verlag, Berlin Heidelberg New York (1997)
13. Meyers, S.: Effective STL - 50 Specific Specific Ways to Improve Your Use of the Standard Template Library. Addison-Wesley, Reading, MA., USA (2001)
14. Stroustrup, B.: The C++ Programming Language (Special Edition). Addison-Wesley, Reading, MA., USA (2000)
15. Stroustrup, B.: The design of C++0x – Reinforcing C++'s proven strengths, while moving into the future. C/C++ Users Journal 23(5) (May 2005)
16. Szűgyi, Z., Pataki, N.: Generative version of the FastFlow multicore library. Electronic Notes in Theoretical Computer Science 279(3), 73–84 (2011)
17. Szűgyi, Z., Pataki, N.: A more efficient and type-safe version of FastFlow. In: Proceedings of Workshop on Generative Programming 2011. pp. 24–37 (2011)
18. Szűgyi, Z., Török, M., Pataki, N.: Multicore C++ Standard Template Library in a generative way. Electronic Notes in Theoretical Computer Science 279(3), 63–72 (2011)
19. Szűgyi, Z., Török, M., Pataki, N.: Towards a multicore C++ Standard Template Library. In: Proceedings of Workshop on Generative Programming 2011. pp. 38–48 (2011)
20. Szűgyi, Z., Török, M., Pataki, N., Kozsik, T.: Multicore C++ Standard Template Library with C++0x. In: NUMERICAL ANALYSIS AND APPLIED MATHEMATICS ICNAAM 2011: International Conference on Numerical Analysis and Applied Mathematics, AIP Conference Proceedings, vol. 1389, pp. 857–860. American Institute of Physics (2011)
21. Torgersen, M.: The expression problem revisited – four new solutions using generics. In: Proceedings of European Conference on Object-Oriented Programming (ECOOP) 2004, Lecture Notes in Computer Science, vol. 3086, pp. 123–143. Springer-Verlag, Berlin Heidelberg New York (2004)
22. Vandevoorde, D., Josuttis, N.M.: C++ Templates – The Complete Guide. Addison–Wesley, Reading, MA., USA (2002)
23. Wei, H., Yu, J., Li, J.: The design and evaluation of hierarchical multi-level parallelisms for h.264 encoder on multi-core architecture. Computer Science and Information Systems 7(1), 189–200 (2010)

Zalán Szűgyi is assistant at Faculty of Informatics, Eötvös Loránd University (Budapest, Hungary) since 2010. He is teaching C++ programming language. His research area includes static analysis of programming languages, multicore programming, and generative programming.

Márk Török is a PhD student at Faculty of Informatics, Eötvös Loránd University (Budapest, Hungary) where he is assistant since 2010. Programming languages and multicore programming belong to the fields of his interest.

Norbert Pataki is assistant at Faculty of Informatics, Eötvös Loránd University (Budapest, Hungary) since 2009. His research area includes programming languages (especially the C++ programming language), multicore programming, software metrics, and generative programming.

Tamás Kozsik received his PhD (summa cum laude) in computer science in 2006 at Eötvös Loránd University (Budapest, Hungary), where he works as associate professor and vice-dean for scientific affairs and international relations of Faculty of Informatics. Since 1992 he has been teaching programming languages, as well as distributed and concurrent programming. His research fields are program analysis and verification, refactoring, type systems and distributed systems. His PhD thesis investigated the integration of logic-based and type system based verification of functional programs.

Received: December 31, 2011; Accepted: May 21, 2012.