Available online at [www.sciencedirect.com](http://www.sciencedirect.com/)



[Electronic Notes in Theoretical Computer Science 279 (3) (2011) 73–84](http://dx.doi.org/10.1016/j.entcs.2011.11.039)

[www.elsevier.com/locate/entcs](http://www.elsevier.com/locate/entcs)

Generative Version of the FastFlow Multicore Library

Zala´n Szu˝gyi[1](#_bookmark0) Norbert Pataki[2](#_bookmark0)

*Department of Programming Languages and Compilers E¨otv¨os Lora´nd University*

*Budapest, Hungary*

**Abstract**

Nowadays, one of the most important challenges in programming is the efficient usage of multicore proces- sors. Many new programming languages and libraries support multicore programming.

FastFlow is one of the most promising multicore C++ libraries. Unfortunately, a design problem occurs in the library. One of the most important methods is pure virtual function in a base class. This method supports the communication between different threads. Although, it cannot be template function because of the virtuality, hence, the threads pass and take argument as a void\* pointer. The base class is not template neither. This is not typesafe approach. We make the library more efficient and safer with the help of generative technologies.

*Keywords:* multicore programming, C++, FastFlow, template

# Introduction

The recent trend to increase core count in processors has led to a renewed inter- est in the design of both methodologies and mechanisms for the effective parallel programming of shared memory computer architectures. Those methodologies are largely based on traditional approaches of parallel computing.

Usually, low-level approaches supplies the programmers only with primitives for flows-of-control management (creation, destruction), their synchronization and data sharing, which are usually accomplished in critical regions accessed in mutual exclusion (mutex). For instance, POSIX thread library can be used to this purpose. Programming parallel complex applications in this way is certainly hard; tuning them for performance is often even harder due to the non-trivial effects induced by memory fences (used to implement mutex) on data replicated in the core’s caches.

1 Email: [lupin@ludens.elte.hu](mailto:lupin@ludens.elte.hu)

2 Email: [patakino@elte.hu](mailto:patakino@elte.hu)

1571-0661 © 2011 Elsevier B.V. Open access under [CC BY-NC-ND license.](http://creativecommons.org/licenses/by-nc-nd/3.0/)

[doi:10.1016/j.entcs.2011.11.039](http://dx.doi.org/10.1016/j.entcs.2011.11.039)

Indeed, memory fences are one of the key sources of performance degradation in communication intensive (e.g. streaming) parallel applications. Avoiding memory fences means not only avoiding locks but also avoiding any kind of atomic operation in memory (e.g. Compare-And-Swap, Fetch-and-Add). While there exists several assessed fence-free solutions for asynchronous symmetric communications, these re- sults cannot be easily extended to asynchronous asymmetric communications that are necessary to support arbitrary streaming networks.

The important approach to ease programmer’s task and improve program effi- ciency consist in to raise the level of abstraction of concurrency management prim- itives. For example, threads might be abstracted out in higher-level entities that can be pooled and scheduled in user space possibly according to specific strategies to minimize cache flushing or maximize load balancing of cores. Synchronization primitives can be also abstracted out and associated to semantically meaningful points of the code, such as function calls and returns, loops, etc.

This kind of abstraction significantly simplify the hand-coding of applications. However, it is still too low-level to effectively automatize the optimization of the par- allel code: the most important weakness here is in the lack of information concerning the intent of the code (idiom recognition); inter-procedural/component optimization further exacerbates the problem.

Recently, there has been a trend of generating programs from high-level spec- ifications. This is called the generative approach, which focuses on synthesizing implementations from higher-level specifications rather than transforming them. From this approach, programmers’ goal is captured by the specification. In addi- tion, technologies for code generation are well developed (staging, partial evaluation, automatic programming, generative programming) [[7](#_bookmark11)]. FastFlow [[4](#_bookmark9)], TBB [[13](#_bookmark15)] and OpenMP [[8](#_bookmark12)] follow this approach. The programmer needs to explicitly define par- allel behaviour by using proper constructs, which clearly bound the interactions among flows-of-control, the read-only data, the associativity of accumulation oper- ations and the concurrent access to shared data structures.

FastFlow is a parallel programming framework for multi-core platforms based upon non-blocking lock-free/fence-free synchronization mechanisms. The frame- work is composed of a stack of layers that progressively abstracts out the program- ming of shared-memory parallel applications. The stack has two different goals: to ease the development of applications and make them very fast and scalable. FastFlow is particularly targeted to the development of streaming applications.

*Templates* are key elements of C++ programming language [[15](#_bookmark16)]. They enable data structures and algorithms be parameterized by types thus capturing com- monalities of abstractions at compile time without performance penalties at run- time. *Generic programming*, a recently emerged programming paradigm for writing reusable components – most cases data structures and algorithms – is implemented in C++ with heavy use of templates.

The worker threads of FastFlow are passing data between each other via void\* pointers, which are then processed by the receiving thread. This is rather necessary for implementation reasons. At the core of the FastFlow framework there is an

abstract class ff node. It has a pure virtual method svc, which is the main method of every thread in the application. However, it is needed to handle any kind of data type, but while it is a virtual function it cannot be template. Although, the base class could be a class template. However, this solution is neither elegant nor type- safe [[17](#_bookmark17)].

In this paper we present our solution about how FastFlow library can be ex- tended to be type-safe. We make the abstract base class be templated and we use static polymorphism instead of dynamic polymorphism by curiously recurring tem- plate pattern. This way in one hand the system will be type-safe, and on the other hand the virtuality of the method can be eliminated, thus the system becomes more effective.

This paper is organized as follows. In section [2](#_bookmark1) the FastFlow library is shown and present typical examples that are risky from the view of program correctness. We present an approach that can be used to overcome this problem by the type system of C++ in section [3](#_bookmark2), the implementation details are described in section [4](#_bookmark3). The sections [5](#_bookmark4) and [6](#_bookmark5) present our pipeline and buffer implementation. In chapter [7](#_bookmark6) we demonstrate the usage of our extended FastFlow library through an example. Finally, this paper concludes in section [8](#_bookmark7).

# FastFlow

FastFlow is one of the most efficient multicore C++ frameworks. It is similar to Intel’s TBB, advocates a pattern-based (skeletal) approach to parallel program- ming. FastFlow aims to provide a set of low-level mechanisms able to support low latency and high-bandwidth data flows in a network of threads running on a SCM. These flows, as typical in streaming applications, are supposed to be mostly undi- rectional and asynchronous. On these architectures, the key issues regard memory fences, which are required to keep the various caches coherent. The library provides the programmer with two basic mechanisms: Multiple-Producer-Multiple-Consumer (MPMC) queues and a memory allocator. The memory allocator is build on top of MPMC queues and can be substituted either with OS standard allocator (paying a performance penalty) or a third-party allocator (e.g. Intel TBB scalable allocator). The key intuition underneath FastFlow is to provide the programmer with lock- free MP queues and MC queues (that can be used in pipeline to build MPMC queues) to support fast streaming networks. Traditionally, MPMC queues are build as passive entities: threads concurrently synchronize to access data. Synchroniza- tions are usually supported by one or more atomic operations (e.g. Compare-And- Swap) that behave as memory fences. FastFlow design follows a different approach: the synchronizations among queue readers or writers are arbitrated by an active en- tity (e.g. a thread). We call these entities Emitter (E) or Collector (C) according to their role; they actually read an item from one or more lock-free Simple-Producer- Simple-Consumer (SPSC) queues and write onto one or more lock-free SPSC queues.

Hence, this requires a memory copy. On the other hand, no atomic operation is needed.

The performance advantage of this solution descend from the higher speed of the copy with respect to the memory fence, that advantage is further increased by avoiding cache invalidation triggered by fences. This also depends on the size and the memory layout of copied data. The former point is addressed using data pointers instead of data, and enforcing that the data is not concurrently written: in many cases this can be derived by the semantics of the skeleton that has been implemented using MPMC queues (as an example this is guaranteed in a stateless farm and many other cases).

When using dynamically allocated memory, the memory allocator plays an im- portant role in term of performance. Dynamic memory allocators (malloc/ free) rely on mutual exclusion locks for protecting the consistency of their shared data structures under multi-threading. Therefore, the use of memory allocator may sub- tly reintroduce the locks in the lock-free application. FastFlow includes its own custom memory allocator, which is specifically optimized for SPMC pattern. The basic assumption is that, in streaming application, typically, one thread allocate memory and one or many other threads free memory. This assumption permits to develop a multi-threaded memory allocator that use SPSC channels between the allocator thread and the generic thread that performs the free, avoiding the use of costly lock based protocols for maintaining the memory consistency of the internal structures. However, the FastFlow allocator is not a general purpose allocator and it has limitations.

Let us consider the following code snippet that takes advantage of FastFlow library: The Emitter feeds the workers with data, now it just send integers. The Worker processes the data, and passes it toward to the other Worker or a Collector, depends on the initialization of FastFlow. Now it just prints the received integer value to the screen, and pass it toward to the Collector. The Collector gathers the computation results done by the Workers. Now it just receives the integer value and does nothing.

// the generic worker struct Worker: ff::ff\_node

{

void \* svc( void \* task )

{

int \* t = (int \*)task;

std::cout << "Worker " << ff\_node::get\_my\_id()

<< " received task " << \*t << "\n"; return task;

}

};

// the gatherer filter

struct Collector: ff::ff\_node

{

void \* svc( void \* task )

{

int \* t = (int \*)task;

if (\*t == -1) return NULL; return task;

}

};

// the load-balancer filter struct Emitter: ff::ff\_node

{

Emitter( int max\_task ):ntask( max\_task ) {}

void \* svc( void \* )

{

int \* task = new int( ntask );

--ntask;

if ( ntask < 0 ) return NULL; return task;

}

private:

int ntask;

};

int main()

{

ff\_farm<> farm;

Emitter E( streamlen ); farm.add\_emitter( &E );

std::vector<ff\_node \*> w;

for( int i = 0; i < nworkers; ++i ) w.push\_back( new Worker );

farm.add\_workers( w );

Collector C; farm.add\_collector( &C );

farm.run\_and\_wait\_end();

}

The communication between the emitter, the collector and the workers is done

via void\* pointers, so arbitrary data can be passed. But this approach is not typesafe at all. FastFlow works this way because the svc method is specified as a pure virtual function in the base class. A template method can be typesafe and elegant solution to pass arbitrary data, however, a (pure) virtual method cannot be template. On the other hand, the abstract base class itself could be template, but the original implementation does not take advantage of this feature.

# Base of the Implementation

Curiously recurring template pattern can be used for static polymorphism [[5](#_bookmark10)]. This idiom is able to simulate dynamic binding without virtual functions [[9](#_bookmark13)]. In this case, the method is selected at compilation time, hence this approach is more effective at run-time, too. This pattern can be used in many ways [[12](#_bookmark14)]. Typical appearance of this approach is the following:

template <class Derived> struct Base

{

void f()

{

static\_cast<Derived\*>( this )->f();

}

};

struct Derived: Base<Derived>

{

void f()

{

// actual implementation

}

};

Certainly, this pattern has limitation: the derived types cannot depend on run- time information. However, in our case, we can distinguish between emitter, collec- tor and worker types at compilation time, thus we do not need to deal with runtime information. Therefore, this limitation has no effect on FastFlow’s usage.

# Implementation Details

In FastFlow the base class of parallel computing is ff node. That class has a pure virtual method called svc, which has a void\* argument. All of the classes take part in parallel computation must be derived from ff node and must implement the member function svc. As the argument type of svc is void\*, it can accept arbitrary data, however, it is not a type safe solution. Inside the function we need to cast the argument to a proper type. Casting to an improper type causes a runtime error. Nevertheless it can be compiled without any error or warning message, making the bugfix more difficult.

Our solution based on curiously recurring template pattern described in section

[3](#_bookmark2). This way, the function svc does not need to be virtual and the type of its argument can be a properly typed pointer. The type of the argument is a template parameter of class ff node. The class ff node has another template parameter which refers to the type of its derived class. The following code snippet shows the declaration of class ff node:

template<class Derived, class T> class ff\_node;

There are three types of nodes in FastFlow: the emitter, the worker and the collector or fallback. The tasks are scheduled by the emitter. The workers com- pute them parallelly and independently. Finally, the collector gathers the results.

The class ff farm represents a farm skeleton. Originally that class has two template arguments, the type of load balancer and the type of gatherer. These template arguments have default value, thus the programmer can skip them and let the FastFlow use the default types. We defined five template arguments of ff farm additionally. The first three template arguments refer to the type of emitter, worker and collector, the fourth one refers the type passed as argument of member function svc, the fifth one is a boolean value which sets whether FastFlow uses circular or non-circular buffer. The last two are the original template arguments. The details of buffers are described in section [6](#_bookmark5). As the ff farm is a child class of ff node, it needs to pass itself to a template argument of ff node. The following code snippet shows the declaration of the class ff farm.

template<class E,

class W, class C, class T,

bool is\_circular = false,

class lb=ff\_loadbalancer<E, W, C, T>, class gt=ff\_gatherer<C, W, T> >

class ff\_farm:

public ff\_node<ff\_farm<E, W, C, T, is\_circular, lb, gt>, T>;

There are several farm schemes exist which do not require emitter or collector types. It is not problem in the original implementation of FastFlow. The pro- grammer just skips to add emitter or collector to the farm. With our solution the programmer needs to specify the type of all kinds of nodes: emitter, worker and collector. To overcome this problem we created a class called Void. This class is also a child of class ff node, and it has a template parameter which denotes the type of the argument of member function svc. This class implements the svc mem-

ber function but in this implementation that function does nothing. Instantiating ff farm, the programmer can use Void type to indicate there is no need for emitter or collector types. The definition of class Void is below:

template<class T>

struct Void : ff::ff\_node<Void, T>

{

T\* svc( T\* task )

{

return 0;

}

};

# Pipelines

The high level pipeline schema is also supported by FastFlow. The frame class of pipeline is called ff pipeline. The pipeline contains stages, one stage can be either a standalone ff node or a farm skeleton. In the original version, the pipeline stores the stages in a vector of ff node\*, where the ff node is the base type of the stages, and the virtual function svc does the computation.

In our solution, where the dynamic polymorphism replaced with static polymor- phism, this approach does not work. While the stage classes have to add themselves as a template argument of ff node, their base types will be different. Hereby they cannot be stored in one vector.

To overcome this problem we defined the class ff pipeline as template class, and its template argument, is a boost::mpl::list [[1](#_bookmark8),[20](#_bookmark18)]. This template argument defines the type of the stages. The vector stores void\* pointers and a template metaprogram – with the help of template argument – will cast them to the proper type on usage. This casting process will be done at compilation time, keeping the type safe, and it is hidden from the programmer.

In the original version of FastFlow the class ff pipeline deals with stages by the interface of ff node and virtual functions. See the example below:

for( int i = 0; i < number\_of\_stages; ++i )

{

stage\_list[i]->do()

}

Our solution applies the following pattern:

template<class typelist> struct ff\_pipeline

{

std::vector<void\*> stages;

void do()

{

do\_aux<

0,

boost::mpl::size<typelist>::value

>::do(stages);

}

template<int I, int N> struct do\_aux

{

static void do(std::vector<void\*>& stages)

{

static\_cast<

typename boost::mpl::at< typelist, boost::mpl::int\_<I>

>::type

>(stages[I])->do();

do\_stages\_aux< I+1,

boost::mpl::size<typelist>::value

>::do(stages);

}

};

template<int N>

struct do\_aux<N, N> /\* specialization \*/

{

static void do(std::vector<void\*>&) { }

};

/\* ... \*/

};

Our solution based on recursive template instantiation [[1](#_bookmark8)]. The struct do aux has

two template arguments: I is the loop variable, and N is the size of typelist. This struct has a static member function called do which casts the corresponding stage to the proper type and invokes its do member function. After that, it instantiates itself with I+1 and N, and calls its do member function recursively. When I equals to N, the specialized version of do aux is selected. While its static member function does nothing, the recursion will be stopped.

The computation starts by the member function do of struct ff pipeline. It instantiates the struct do aux with 0 and the size of the typelist and invokes its static member function. The entities in scope boost::mpl are described in [[20](#_bookmark18)].

The example below shows how to create pipeline in FastFlow:

struct Stage1 : ff\_node<Stage1, my\_data\_type>; struct Stage2 : ff\_node<Stage2, my\_data\_type>;

//...

typedef boost::mpl::vector<Stage1, Stage2>::type stage\_types; ff\_pipeline<stage\_types, my\_data\_type>\* pipe =

new ff\_pipeline<stage\_types, my\_data\_type>(); pipe->add\_stage(new Stage1());

pipe->add\_stage(new Stage2());

The structs Stage1 and Stage2 derived from ff node are the stages of the pipeline and the stage types refers to their type. The ff pipeline is instantiated with that. In the last two rows, that stages are added.

While a template argument of ff pipeline refers to the type of the stages, their structure must be known at compilation time. Therefore it introduces a restriction of the usage. However, in most cases the structure of the pipeline is defined at compilation time, thus it is not a real limitation.

# Buffers

The original version of FastFlow can use either circular or non-circular buffer as a channel between worker threads. These buffers store void\* to be able to handle different types of data. By default the non-circular buffer is used, how- ever, the programmer can select the other one by defining preprocessor macro FF BOUNDED BUFFER. The following preprocessor code snippet shows how the proper buffer is chosen.

#if defined(FF\_BOUNDED\_BUFFER) #define FFBUFFER SWSR\_Ptr\_Buffer

#else // non-circular buffer #define FFBUFFER uSWSR\_Ptr\_Buffer #endif

Later the FFBUFFER macro is used as a buffer type.

The idea of writing preprocessor macros is against the modern development approaches in C++. The code will be less readable, harder to understand, and the preprocessor does not know the syntax of the language. The possible errors are figured out only at compile time (after preprocessing) and the line numbers of the errors (presented by the compiler) may be confusing, because they refer to the place where the preprocessor macros are applied, and not where they are defined.

On the other hand, with our type-safe solution the buffers need to store a pointer to a proper type and not to void\*, thus they need to be template. Defining template buffers with preprocessor macros makes the code more messy.

In our solution we avoid using preprocessor macros to defining the buffers. In- stead we add an extra boolean template argument to ff farm called is circular, to indicate which kind of buffer to be to use. The true value means we want to use circular buffer, and the false means we chose the non-circular one. The default value of that template argument is false.

At the definition of buffer we apply that template argument as a condition on a template metaprogram if [[1](#_bookmark8)], which chooses the proper buffer type during compilation. The following code snippet shows this, where T is the type being stored in the buffer:

if\_<is\_circular, SWSR\_Ptr\_Buffer<T>,

uSWSR\_Ptr\_Buffer<T> >::type buffer;

The definition of if is the following:

template< bool condition, class then\_, class else\_ > struct if\_

{

typedef then\_ type;

};

template< class then\_, class else\_ > struct if\_<false>

{

typedef else\_ type;

}

If the argument condition is true, the general if structure is chosen to be

instantiate, thus the type refers to template type argument then . If the argument condition is false the partially specialized if structure is chosen in the instantia- tion process, thus type defines the template type argument else .

# Example

In this chapter we show the example of chapter [2](#_bookmark1) rewritten with templates. Each class derived from ff node instantiated with itself and the task type int. The function svc in each classes has a properly typed pointer, instead of void\*. The main function creates the farm, the emitter, worker and collector classes, and starts computing.

// the generic worker

struct Worker: ff::ff\_node<Worker, int>

{

int \* svc( int \* task )

{

std::cout << "Worker "

<< ff\_node<Worker, int>::get\_my\_id()

<< " received task " << \*task << "\n"; return task;

}

};

// the gatherer filter

struct Collector: ff::ff\_node<Collector, int>

{

int \* svc( int \* task )

{

if ( \*task == -1 ) return NULL; return task;

}

};

// the load-balancer filter

struct Emitter: ff::ff\_node<Emitter, int>

{

Emitter( int max\_task ):ntask( max\_task ) {}

int \* svc( int \* )

{

int \* task = new int( ntask );

--ntask;

if ( ntask<0 ) return NULL; return task;

}

private:

int ntask;

};

int main()

{

ff\_farm<Emitter, Worker, Collector, int> farm; Emitter e( TASK\_NUM );

Collector c;

std::vector<ff::ff\_node<Worker, int> \*> workers; for( int i = 0; i < WORKER\_NUM; ++i )

workers.push\_back( new Worker() ); farm.add\_emitter( &e ); farm.add\_workers( workers ); farm.add\_collector( &c );

farm.run\_and\_wait\_end();

}

# Conclusion

Multicore programming is an interesting new way of programming. Many C++ libraries support multicore programming, however, Fastflow is one of the most promising ones.

In this paper we pointed out a weakness of the FastFlow multicore programming library of the C++ programming language. The original version of FastFlow uses void\* to communicate the between threads. While this method provides a simple solution to pass arbitrary data, it is not typesafe and increases the chance of make mistakes.

We presented our solution to overcome of this problem. We introduced template arguments to deal with the type of arguments and static polymorphism is applied insted of dynamic polymorphism.

In this paper we do not deal with the speedup of FastFlow because it would

be misleading. We have eliminated virtuality of a very important method in the library, but we have not changed anything in the multicore functionality. The typical conception of efficacy of a multicore library is based on the features of multicore programming and it is unchanged in this case.

Our approach also shows why generative technologies are strengthened among multicore programming, too.

# Acknowledgement

The Project is supported by the European Union and co-financed by the European Social Fund (grant agreement no. TA´MOP 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 (2004).
2. Aldinucci, M., Danelutto, M., Meneghin, M., Kilpatrick, P., Torquati, M.: *Efficient streaming applications on multi-core with FastFlow: the biosequence alignment test-bed*, in Proc. of Intl. Parallel Computing (PARCO) 2009.
3. Aldinucci, M., Meneghin, M., Torquati, M.: *Efficient smith-waterman on multi-core with FastFlow*, In Proc. of Intl. Euromicro PDP 2010: Parallel Distributed and network-based Processing, IEEE, pp. 195–199.
4. Aldinucci, M., Ruggieri, S., Torquati, M.: *Porting Decision Tree Algorithms to Multicore using* *FastFlow*, in Proc. of European Conference in Machine Learning and Knowledge Discovery in Databases (ECML PKDD), LNCS **6321**, pp. 7–23.
5. Alexandrescu, A.: “Modern C++ Design”, Addison-Wesley (2001).
6. Bishof, H., Gorlatsch, S.: *Generic parallel programming using C++ templates and skeletons*, in Proc. of Domain-Specific Program Generation (International Seminar), 2003, Revised Papers, LNCS **3016**,

pp. 107–126.

1. Czarnecki K., Eisenecker, U. W.: “Generative Programming: Methods, Tools and Applications”, Addison-Wesley (2000).
2. Dagum, L., Menon, R.: *OpenMP: An industry-standard API for shared-memory programming*, “IEEE Computational Science and Engineering” **5** (1998), pp. 46–55.
3. Duret-Lutz, A., G´eraud, T., Demaille, A.: *Design patterns for generic programming in C++*, In Proc. of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS), pp. 189–202.
4. Harrison, N., Meiners, J. H.: *The dynamics of changing dynamic memory allocation in a large- scale C++ application*, In Companion To the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (Portland, Oregon, USA, October 22 - 26, 2006) (OOPSLA, 2006), pp. 866–873.
5. Leiserson, C. E.: *The Cilk++ Concurrency Platform*, in Proc. of 46th Annual Design Automation Conference 2009, pp. 26–31.
6. Pataki, N.: *C++ Standard Template Library by safe functors*, in Proc. of the 8th Joint Conference on Mathematics and Computer Science (MaCS 2010), Selected Papers, pp. 363–374.
7. Pheatt, C.: *Intel Threading Building Blocks*, “Journal of Computing Sciences in Colleges” **23(4)**, pp. 298-298.
8. Reinders, J.: “Intel Threading Building Blocks”, O’Reilly (2007).
9. Stroustrup, B.: “The C++ Programming Language”, Special Edition, Addison-Wesley (2000).
10. Szu˝gyi, Z., Pataki, N.: *Sophisticated Methods in C++*, In Proc. of International Scientific Conference on Computer Science and Engineering (CSE 2010), pp. 93–100.
11. Szu˝gyi, Z., Pataki, N.: *A More Efficient and Type-Safe Version of FastFlow*, in Proc. of Workshop on Generative Technologies (WGT 2011), pp. 24–37.
12. Szu˝gyi, Z., Sinkovics, A´., Pataki, N., Porkola´b, Z.: *C++ Metastring Library and its Applications*, In Proc. of Generative and Transformational Techniques in Software Engineering 2009, LNCS **6491**, pp. 461–480.
13. Szu˝gyi, Z., Tor¨ok, M., Pataki, N.: *Towards a Multicore C++ Standard Template Library*, in Proc. of Workshop on Generative Technologies (WGT 2011), pp. 38–48.
14. <http://www.boost.org/doc/libs/1_45_0/libs/mpl/doc/index.html>