High-Level Parallel Programming EDSL
A BOOST libraries use case

Joel Falcou
LRI† – Université Paris Sud XI – Orsay, France
joel.falcou@lri.fr

ABSTRACT
Parallel programming is becoming a concern for a growing audience of developers. Various programming models and tools have been proposed but never became largely accepted in the community. This paper shows how a Embedded Domain Specific Language has been defined using BOOST libraries like MPL and Proto and how it solves various typical problem of high-level parallel programming tools like reusing legacy code or preserving developer habits. This EDSL – QUAFF – is able to map data-flow applications over clusters, multi-cores and CELL processor using algorithmic skeletons. Realistic application is presented and assess QUAFF expressiveness and efficiency.

Categories and Subject Descriptors
D.1.3 [Software]: Concurrent Programming—parallel programming

Keywords
Parallel programming, algorithmic skeletons, embedded domain specific languages

1. INTRODUCTION
As Sutter stated[1], the free lunch is over for sequential programming as concurrency and parallel programming must be taken into account by ever an growing audience of developers. For a decade now, scientific computing applications have become more and more demanding in terms of raw computing power. Moreover, various technology leap like multi-core and heterogeneous multi-cores changed the deal. The upcoming many-core era will feature ready-to-use, affordable high performance computing platforms in a simple desktop machine. But, for non-specialists, writing efficient code for such machines or group of machines is a non-trivial task, as it usually involves dealing with low level APIs like OpenMP, pThread or MPI.

Manipulating such frameworks is difficult and error prone. Deadlocks and other common undesired behaviors make parallel software development very slow compared to classic, sequential one. In a similar way to how Design Patterns[2] are a common way to build reusable, structured sequential software components, various attempts were made to provide a structured framework to design and implement non-trivial applications and to deliver high performance. Parallel Algorithmic Skeletons[3, 4, 5] have been proposed as such a methodology. Skeletons are typical parallel patterns that can be implemented once and for all on a given platform. Building parallel softwares using skeletons then boils down to combining skeletons and sequential code fragments, thus seeing skeletons as Higher-Order Functions.

A large collection of languages[6] and software libraries[3, 7] proposed an implementation of this paradigm and, recently, industrials started to show interest in those methods and provided spot-on application of this paradigm: Google and its distributed data-mining application [8] or MallBa [9], a combinatorial optimization application. Despite the emphasis put on making skeleton easy to use, skeletons are still largely unused in many fields. One possible answer is that users are reluctant to port existing application to the skeleton model as the time to adapt legacy code to the specificities of a given skeleton library or learning a rather exotic languages using unfamiliar paradigms are restrictive factors. In an ideal world, legacy code should be seamlessly integrated into the skeleton application with the least additional work and the skeletons themselves should be directly available in a wide-spread language like C or C++.

The goal of this paper is to show that such a seamless integration is possible by designing a Domain Specific Language for algorithmic skeletons and embedding it in C++ using classic idioms of generative programming like partial evaluation and meta-programming. Instead of rebuilding such features from scratch, we decided to use BOOST libraries like Proto and MPL to embed our language and to perform the various meta-programming tasks needed to interface this EDSL with existing C++ code. The paper is organized as follow. Section 2 will present the overall API and use-cases solved by our library. Section 3 will show how BOOST libraries helped implement QUAFF as an EDSL. Finally, section 4 will quickly presents a complete application developed with QUAFF. We will conclude on future works and research directions in section 5.
2. THE QUAFF LIBRARY

2.1 Quaff programming model

Listing 1 presents a simple algorithm parallelized with QUAFF. In this example, we want to apply the \texttt{comp} function on a vector in a parallel way.

Listing 1: Quaff sample code

```cpp
#include <quaff/quaff.hpp>
using namespace quaff;

void load( vector<float>& d );
void comp( vector<float>& d, vector<float>& r );
void save( vector<float>& d );

int main(int argc, const char* argv[])
{
    initialize(argc, const char* argv[]);
    run( ( seq(load),
            map<16>(seq(comp)),
            seq(save) ) );
    finalize();
}
```

The actual code is split into four parts. First, the user functions are defined naturally. The only limitation is the ordering of their arguments (input first, output last) which is a requirement to enable QUAFF to determine how data should be transferred between processes. Communication support for all POD types and some STL container are provided, thus limiting the marshaling code one needs to write to support custom types; Then the parallel execution environment is initialized at line 10 via the \texttt{initialize} function. From this point, all skeleton expressions will be able to be evaluated and run on the underlying MPI-enabled parallel machine. The application is described as a combination of skeleton constructors on line 12-14.

In this example, we first load data from a file, distribute this data over processors using the \texttt{MAP} skeleton, perform the computation and gather the results which are then saved back on disk. This sample code shows the explicit call to the \texttt{MAP} and \texttt{SEQ} skeleton constructors and the use of the comma operator as the \texttt{CHAIN} skeleton constructor. Note that skeleton constructors can be parameterized by additional information. For instance, \texttt{MAP} takes an additional template parameter that describes on how many processors data will be distributed. Finally, The parallel execution environment is shutdown at Line 17 by the \texttt{finalize} function.

2.2 Supported skeletons

Various implementations of parallel skeletons have been proposed in the literature. As skeletons are usually defined from a parallelization pattern arising from the implementation of various application classes, there is no standard list of skeletons. QUAFF supports a small subset of skeletons which are usually agreed over by the community:

- The \texttt{SEQ} skeleton encapsulates sequential user-defined functions to use them as parameters of skeletons;
- The \texttt{CHAIN} skeleton that call other skeletons in sequence.
- The \texttt{PARDO} skeleton supports \textit{ad hoc} parallelism as advocated by Cole in \cite{cole2002}; \texttt{PARDO PARDO} simply spawns parallel processes with no defined communications schemes.
- The \texttt{PIPELINE} skeleton which is functionally equivalent to parallel function composition;
- The \texttt{FARM} skeleton models irregular, asynchronous data parallelism in which inputs are dynamically distributed to a pool of slave processors using some parameterizable heuristic;
- The \texttt{MAP} skeleton models regular data parallelism in which input data generates a new sub-stream of data on which a given function is applied. These sub-results are then merged back as a new output element.

All those skeletons are directly usable in QUAFF by using the corresponding function. Some of them are also mapped on some operators to make the description of parallel applications easier. For instance, the \texttt{CHAIN}, \texttt{PIPE} and \texttt{PARDO} skeletons are respectively mapped onto the comma operator, the bitwise \texttt{or} and the bitwise \texttt{and} operators.

2.3 Related Work

State of the art skeleton-based parallel programing libraries are built on top of languages like C, C++ or JAVA. The most representative works are:

- The \texttt{ESKEL} library \cite{cole2002} proposed by Murray Cole represents a concrete attempt to embed the skeleton based parallel programming method into the mainstream of parallel programming. It offers various skeletal parallel programming constructs which stay similar to MPI primitives and which are directly usable in C code. However, eSkel low-level API requires to take care of internal implementation details.
- \texttt{MUESLI}, the Münster Skeleton Library \cite{kuchen2005} is a C++ skeleton library proposed by Herbert Kuchen. Based on a platform independent structure, it integrates features from several other systems like the two-tier model of P3L \cite{kuchen2006} and some data parallel skeletons. The main idea is to generate a process topology from the construction of various skeleton classes and to use a distributed container to handle data transmission. This polymorphic C++ skeleton library is interesting as it proposes a high level of abstraction but stays close to a language that is familiar to a large crowd of developers. Moreover, the C++ binding for higher order functions and polymorphic calls ensure that the library is type safe. The main problem is that the overhead due to dynamic polymorphism is rather high (between 20 and 110 \% for rather simple applications).
- \texttt{Lithium} \cite{kuchler2006} is a Java skeleton library for clusters or grid-like networks of machines. The underlying execution model is a macro data-flow one. The choice of Java is motivated by the fact it provides an easy to use, platform-independent, object oriented language.
3. IMPLEMENTATION USING BOOST

In [11], QUAFF was described as a skeleton-based parallel programing library which main originality was to rely on C++ template meta-programming to reduce the overhead traditionally associated with object-oriented implementations of such libraries. The basic idea was to use the C++ template mechanism so that skeleton-based programs are actually expanded at compile-time and generate a new C++ MPI code to be compiled and executed at run-time. However, the design of this version was rather simplistic and was not easily extensible. To make QUAFF more flexible and user-friendly, we decided to redesign it as an Embedded Domain Specific Language. To do so, the skeleton structure extracted from the application definition has to be transformed into an list of executable instructions for a virtual parallel machine as shown on figure 1.

![Figure 1: Quaff code generation process](image)

This process and its associated specific language is described in [12] and relies on specific data structures and a set of operational semantic rules. If languages like metaOCaml or Template Haskell natively support such constructions, C++ needs to use and abuse template meta programing and operator overloading. Expressions Templates [13] have been a tool of choice for such a task but writing complex EDSL by hand can quickly become a daunting task and lead to hard to maintain tools.

To reduce those difficulties, Niebler proposed a compiler construction toolkit for embedded languages called PROTO [14]. It allows developers to specify grammar and semantic actions for EDSL and provide a semi-automatic generation of all the template structures needed to make it work. Compared to hand-written, Expressions Templates based EDSL, designing a new embedded language with PROTO is done at a higher-level of abstraction. As PROTO consumes abstract syntax trees that already conform to the grammar for valid C++ expressions instead of sequences of tokens, the developer only need to fill in the domain specific features of its language. In a way, PROTO hijacks the normal compiler workflow so domain specific code transformations can take place as soon as possible.

We will use PROTO to define a proper tool for building a skeleton tree directly in C++ and implementing the transformation needed to make this description do actual parallel work. This restructuring introduced new challenges:

- **Specify a proper Proto grammar for Quaff API.** Skeleton nesting is usually freely done by the end user. However, some nesting schemes are either incorrect or lead to poor performance. QUAFF grammar tries to prevent as much as possible of these cases;

- **Turn Quaff semantic into Proto transforms.** This require providing a proper meta-programmed version of all the formal data-structures involved in QUAFF semantic and use Fusion and MPL to compute results of semantic rules. As Proto transforms act as semantic actions, we’ll also have to do a proper binding between our new transforms and grammar rules;

- **Generating message-passing API calls from the process network specification.** Once the process network have been generated, we have to properly execute its instruction lists in a proper MPI-enabled environment.

3.1 Defining the skeletons grammar

At a first glance, one can define a rather straightforward grammar ruleset for skeleton based applications:

$$
\Phi ::= \text{seq } f \\
\Sigma ::= \Phi | \text{chain } \Phi_1 \Phi_2 \\
\Gamma ::= \Phi | \text{pipe } \Sigma_1 \Sigma_2 | \text{pardo } \Sigma_1 \Sigma_2 \\
f ::= \text{user-defined function} \\
n ::= \text{integer greater than 1}
$$

However, some nesting like a FARM of PARDO makes no sense. Moreover, Aldinucci showed in [15] that skeleton-based application performances may be increased when they are expressed in a canonical form. In this form, FARM skeleton plays a specific role among common skeletons and skeleton nesting FARMS should be rewritten as FARM skeleton nesting other constructions. To take into account those remarks, we propose a tiered grammar in which we enforce the role of FARM:

$$
\Phi ::= \text{seq } f | \text{chain } \Phi_1 \Phi_2 | \text{map } n \Phi_1 \\
\Gamma ::= \Phi | \text{pipe } \Gamma_1 \Gamma_2 | \text{pardo } \Gamma_1 \Gamma_2 \\
\Sigma ::= \Gamma | \text{farm } n \Gamma \\
f ::= \text{user-defined function} \\
n ::= \text{integer greater than 1}
$$

In this grammar, we separated the skeletons into three tiers. The first tier gathers the skeleton with a strong sequential component. The second tier gather the control-oriented skeletons. The last tier only contains the FARM skeleton.

The corresponding Proto grammars are straightforward and are shown in listing 2. We simply define three grammars we chain one after the other following the tier structure. Notice how the grammar definition is compact and yet easy to read and how using the pre-defined Proto operator matcher for comma, bitwise or and bitwise and is easy. Also, the size of data-parallel skeletons like MAP and FARM is bundled into an integral constant type and is easily matched using proto::_.

3.2 Implementing operational semantic

As seen in [16], QUAFF formal model transforms skeleton-based abstract syntax trees into process networks. To do so, we formally defined a simple structure for process network, process and process descriptor. Those data structures are then compositionally built by applying operational semantic
Listing 2: Quaff grammar rules as Proto construct

```c++
struct t1_skel :
  or_<terminal<user_function<_>, // seq
    comma<t1_skel,t1_skel>, // chain
    unary_expr<map_<>, t1_skel> // map
  > {};

struct t2_skel :
  or_<t1_skel, bitwise_and<t2_skel,t2_skel> // pardo
    , bitwise_or<t2_skel,t2_skel> // pipe
  > {};

struct skeleton :
  or_<t2_skel, unary_expr<farm_<>, skeleton> // farm
  > {};
```

rules over our skeleton AST. Basically, we define a process network as a tuple containing: a list of labeled processes, a list of PID of input processes and a list of PID of output processes. In a very similar fashion, a labeled process is defined as a pair containing the process PID and a process descriptor. Finally, this descriptor contains the list of process PID with which the current process will communicate, the list of instructions to be run on the process virtual machine and a flag for handling asynchronous processes. The first step of our implementation was to define template versions of these data structures. Listings 3 details those classes.

Listing 3: Data structure for process network, process and process descriptor

```c++
template<class P, class I, class O> struct process_network
{
  typedef P process;
  typedef I inputs;
  typedef O outputs;
};

template<class ID, class DESC, class IT, class OT>
struct process
{
  typedef ID pid;
  typedef DESC descriptor;
  typedef IT input_type;
  typedef OT output_type;
};

template<class IPID, class OPID, class CODE, class KIND>
struct descriptor
{
  typedef IPID i_pids;
  typedef OPID o_pids;
  typedef CODE instrs;
  typedef KIND kind;
};
```

In each structure, lists are represented as Fusion::vector while PID are stored as an integral constant type. Concept-checks are also done to prevent improper instantiation of those structures with unrelated types.

The implementation of the semantic rules is then done by building Proto transforms. Each rule is turned into a custom callable transform that manipulate various meta-programmed structures for encoding process network, process and process descriptor. For example, the rule for building a Pardo skeleton out of two process networks is:

\[
\pi_1 = \langle P_1, I_1, O_1 \rangle \quad (i = 1, 2)\\
\pi_1 \parallel \pi_2 = \langle P_1 \cup P_2, I_1 \cup I_2, O_1 \cup O_2 \rangle \quad \text{(Par)}
\]

In this rule, we simply merge the process network list of process, input and output and return a new process network. This rule directly translates in the following code:

Listing 4: The Par meta-programmed semantic rule

```c++
template<class PN1, class PN2> struct rule_par
{
  typedef copy<PN2::process, back_inserter<PN1::process>> proc;
  typedef copy<PN2::inputs, back_inserter<PN1::inputs>> in;
  typedef copy<PN2::outputs, back_inserter<PN1::outputs>> out;
  typedef process_network<proc, in, out> type;
};
```

The structure of this meta-function is straightforward. We first extract the components of each process network and use mpl::copy to merge them. The resulting type is then rebuilt from the newly computed type list. This same meta-function is then called from the associated Proto transforms:

Listing 5: The Par Proto transform

```c++
template<class D> struct apply_rule<Par,D> : transform<apply_rule<Par,D> >
{
  typedef P D proc, D proc;
  typedef P D proc, D proc;
  typedef P D proc, D proc;
  typedef P D proc, D proc;
};
```

In this transform, we first force the transformation of left and right argument of the matched grammar. We proceed this way instead of using proto::fold so we are sure that a proper unique, monotonous labelling of processes are done for each nested SEQ skeleton. Once performed, we call our rule_par meta-function to merge the resulting networks. The environnement template class acts as a small container carrying the latest evaluated process network and the latest PID to use in the incoming transforms. Similar transforms

\footnote{For type-setting concerns, we omit Proto namespace use and typename keywords in the upcoming listings}
are built for each element of the skeleton set and tied to the associated grammar rules using `proto::when`.

The transform call is triggered by the `run` function which is simply written as a call to the skeleton grammar transform using an empty environment:

```cpp
template<class X> static inline void run(skeleton_expression<X> const &)
{
    typedef result_of<skeleton(X, empty_)>::type type;
    type::run();
}
```

3.3 Generating actual parallel code

We now have to see how the `run` method of the `process_network` class actually produce parallel code. For sake of clarity, we will focus on the MPI implementation of QUAFF as the Cell implementation only differs in the kind of communication primitives it uses.

First, when the meta-computed `process_network::run` method is called, it iterates through a `fusion::vector` of `process` instances and calls their own `run` method. Then, as QUAFF skeleton code is run onto all nodes of our parallel machine, we have to select at run-time which code fragment of the process list has to be effectively generated and run on each of those nodes. Listing 7 shows how its done inside the `process::run` method.

In this method, we retrieve the `process` PID from its integral constant and compare it with the run-time process PID. If they match, we then instantiate the needed input and output container built from a `fusion::tuple`, the actual process code from the descriptor and start a loop that will run this code until a termination message is received. Each macro-instructions to run are stored into a `mpl::vector`. They are built onto the following model:

```cpp
struct macro_send
{
    template<class IPID, class OPID> struct apply
    {
        typedef instr_send<OPID> type;
    };
};
```

The descriptor is then able to build a list of instruction instances by walking through the `mpl::vector` and calling their `apply` member. The result is a `fusion::vector` containing polymorphic function objects. The `instr_send` function object is looking like:

```cpp
template<OPID> struct instr_send
{
    typedef void result_type;
    typedef mpl::at_c<OPID, 0>::type dest;
    template<class In, class Out> void operator()(In const &, Out & out) const
    {
        mpi::send(out, dest::value, is_valid(out) ? tag::Data : tag::Stop);
    }
};
```

In this function, we send the contents of the `Out` argument which is a tuple containing all the process output variables. If this tuple contains only valid data – data that isn’t flagged as causing a program termination –, we send a message with a proper MPI tag. If not, we propagate the termination signal to the next process. The `mpi` namespace use compile-time configuration to detect if a proper MPI machine is available or if the code is compiled on a Cell processor. Depending on this configuration, the `send` function is modified accordingly.

4. APPLICATIONS

To assess QUAFF expressiveness and efficiency, we choose to parallelize a realistic application from the computer vision community. Computer vision features various complex, time-consuming algorithms and has been a field of choices for parallelism and architecture[17]. The chosen application performs object detection and tracking in a stereoscopic video stream using a probabilistic algorithm. In this approach, detecting and tracking is done by computing an a posteriori probability density of a state vector representing the tracked object from a series of random observations[18]. To solve such a problem, one can either analytically resolve the Chapman-Kolmogorov prediction equation or use a Monte-Carlo method. Particle filters [19] are an algorithm that models such random processes by a Markovian process. Probability density is then estimated by a pondered, discrete set of observations inside the observation universe. Details of this algorithm are available in [20].

The implementation of such an algorithm uses four functions:
• **generate**: builds the particle distribution from last iteration results;

• **measure**: extract features from the video stream to evaluate each particles’ interest score;

• **sample**: resample the particle set by replicating particles with large weight and trimming particles with small weight;

• **estimate**: compute the average of the particles set to get the current frame estimation.

The parallelization of this algorithm is then based on distributing the particles set over the available processors and then compute the estimated position of the object on the root node which also runs the GUI parts of the application.

Listing 10 presents the QUAFF listing for this application.

```cpp
#include <quaff/quaff.hpp>
using namespace quaff;

#define NPROC 16
typedef std::vector<particles> data_t;

void gui();
void generate( data_t & );
void measure( image const &, data_t const &, data_t & );
void sample( data_t const &, data_t & );
particles estimate( data_t const & );
void update_gui( particles const & );

int main(int argc, const char * argv[]) {
    initialize(argc, argv);
    run( seq(gui)
         & ( map<NPROC>( seq(generate)
                       , seq(measure)
                       , seq(sample)
                       )
         , seq(estimate)
         , seq(update_gui)
         ));
    finalize();
}
```

Figure 2 shows a sample execution of our 3D tracking application. The upper part of the figure shows the two video streams and the projection of the particles distribution. Lower part is the estimated 3D path of the pedestrian projected on the ground plane.

Table 1 shows results obtained on a stereo 640×480×8bits video stream for several size of particle distribution. Execution times and speed-up are given for a single processor machine (PPC 970) and on a 14 nodes cluster of dual-processor Xserve G5[21]. Near real-time performances are achievable with a low number of nodes: a 20 frames per second rate is achieved by using approximately 2000 particles scattered on 14 nodes and a 12 frames per second rate is achieved with nearly 5000 particles. Contrary to the sequential implementation, we’re able to keep up with real time constraints for more than 3000 particles. We can also see that the global speed up scales correctly with the number of particles.
5. CONCLUSION

Defining tools for high-level programming is a non-trivial task as such tools should be both expressive, efficient and easy to use by non-specialists. Such tools can be efficiently implemented as Embedded Domain Specific Languages into a wide-spread language like C++. In this paper, we designed such an EDSL for building Algorithmic Skeletons based applications from a set of simple constructors. Using Proto and other Boost meta-programming libraries, we successfully implemented this EDSL from a formal model. Experimental results show that realistic applications can be built while still delivering high performance.

Besides this pragmatic success, the design and implementation of Quaff proves that Proto claim of being a compiler construction kit is more than correct and fits how such language based tools are built. Quaff structure including a virtual machine, a set of semantic rules and constructors was fitting right away in the Proto model. The amount of code needed to turn this formal model into actual transforms were acceptable and was strongly helped by the facilities offered by MPL and fusion. The whole redesign of Quaff took less than a week and was mostly spent writing and debugging transforms. The main interest of Proto for library designers is that the transition between the formal, grammar and semantic-based definition of the tool to the raw code is really easy.

Other researchs are currently being done to apply such a methodology to other parallel programming tools. For instance, NT2 [22] is a linear algebra EDSL that mimics MATLAB syntax and semantic to provide an easy way to port such MATLAB code into C++. Main features of NT2 also include a pervasive support for parallel architecture ranging from SIMD-enabled processor to multi-processors, multi-cores and soon GPU or Cell processor. In a similar fashion, NT2 started its as a hand-coded EDSL and is currently being rewritten from the ground up using Proto. Preliminary results show that while the code size was reduced by more than 50%, performances and features are preserved. On a larger time frame, work will be done to see how to use such methodology to implement tools based on other parallel programming models like Bulk Synchronous Parallelism and to provide more domain specific interface for such tools.

Acknowledgements

The author wants to thanks Eric Niebler and Stefen Watanabe for their valuables advices and the time they dedicated to answer a large number of our early questions on the Boost mailing-list.

6. REFERENCES