Formal semantics applied to the implementation of a skeleton-based parallel programming library

Joel Falcou and Jocelyn Sérot

LASMEA, UMR6602 UBP/CNRS, Campus des Cézeaux, 63177 Aubière, France.
E-mail: {Joel.FALCOU, Jocelyn.SEROT}@lasmea.univ-bpclermont.fr

1 Introduction

In a previous paper[1], we described QUAFF, a skeleton-based parallel programming library which main originality is to rely on C++ *template* meta-programming[2,3] techniques to significantly reduce the overhead traditionally associated with object-oriented implementations of such libraries. The basic idea is to use the C++ *template* mechanism so that skeleton-based programs are actually run at compile-time and generate a new C+MPI code to be compiled and executed at run-time. The implementation mechanism supporting this compile-time approach to skeleton-based parallel programming was only sketched mainly because the operational semantics of the skeletons were not stated in a formal way, but "hardwired" in a set of complex meta-programs. As a result, changing this semantics or adding a new skeleton was difficult. In this paper, we give a formal model for the QUAFF skeleton system, describe how this model can efficiently be implemented using C++ meta-programming techniques and show how this helps overcoming the aforementioned difficulties. It relies on three formally defined stages. First, the C++ compiler generates an abstract syntax tree representing the parallel structure of the application, from the high-level C++ skeletal program source. Then, this tree is turned into an *abstract process network* by means of a set of *production rules*; this process network encodes, in a platform-independent way, the communication topology and, for each node, the scheduling of communications and computations. Finally the process network is translated into C+MPI code. By contrast to the previous QUAFF implementation, the process network now plays the role of an *explicit intermediate representation*. Adding a new skeleton now only requires giving the set of production rules for expanding the corresponding tree node into a process sub-network. The paper is organized as follows. Section 2 briefly recalls the main features of the QUAFF programming model. Section 3 presents the formal model we defined to turn a skeleton abstract syntax tree into a process network. Section 4 shows how *template* meta-programming is used to implement this model. We conclude with experimental results for this new implementation (section 5) and a brief review of related work (section 6).

2 The QUAFF library

The programming model of QUAFF is a classical skeleton-based one. Skeletons are defined as follows:

$$\begin{array}{lll} \Sigma & ::= & \mathsf{Seq}\ f \mid \mathsf{Pipe}\ \Sigma_1 \ldots \Sigma_n \mid \mathsf{Farm}\ n\ \Sigma \mid \mathsf{Scm}\ n\ f_s\ \Sigma\ f_m \mid \mathsf{Pardo}\ \Sigma_1\ \ldots \Sigma_n \\ f, f_s, f_m & ::= & \textit{sequential, application-specific user-defined C++ functions} \\ n & ::= & integer \geq 1 \end{array}$$

All user-defined functions take at most one argument and return at most one result. The skeleton set is classical. Intuitively, Seq encapsulates sequential user-defined functions in such a way that they can be used as parameters to other skeletons; Pipe and Farm are the usual task-parallel skeletons (with computations performed in stages and under a master/workers scheme respectively); Scm models data-parallel computations: f_s decomposes the input data into a set of (possibly) overlapping data subsets, the inner skeleton processes each subset in parallel and the f_m function merges the sub-results; Pardo models parallel, independent computations, where n distinct tasks are run on n distinct processors. The parallel structure of an application can then be represented by a tree with nodes corresponding to skeletons and leaves to user-defined sequential functions. A distinctive feature of QUAFF – compared to other skeleton-based parallel programming libraries[4–6] – is that this structure is completely described by means of type definitions. This, of course, is the key point allowing optimized message-passing code to be produced *at compile-time*, as will be explained in section 4. Considering a simple application like:

$$\mathcal{A} = \mathsf{Pipe}(\mathsf{Seq}\ f_1,\ \mathsf{Farm}(4,\ \mathsf{Seq}\ w),\ \mathsf{Seq}\ f_2)$$

It's implemented via the the following code using QUAFF:

Listing 1. Sample QUAFF application

```
typedef task<f1, void_ , int   >         F1;
typedef task<w , int   , double>          W;
typedef task<f2, double, void_ >         F2;

run( pipeline( seq(F1), farm<4>(seq(W)), seq(F2)) );
```

Lines 1–3 register user-defined C functions as *tasks* used into skeleton definitions. A QUAFF *task* is defined by specifying a function and a pair of input/output types. The function itself can be either a C-style function or a C++ functor. On line 5, the skeleton structure is defined using the `pipeline` and `farm` skeleton constructors and executed through the `run` function.

With QUAFF , the *same* language is used for describing the parallel structure of the application, writing application-specific sequential functions and as the target implementation language. This method has two advantages. First, programmers do not have to learn a separate language for describing this structure (as is the case with several existing skeleton-based parallel programming systems such as P3L[5] or Skipper[7]). Second, it makes insertion of existing sequential functions into skeletons easier and more efficient since no special foreign function interface is required: they just need to conform to the generic `t_result f(t_arg)` prototype.

3 Formal model

The implementation model of QUAFF is CSP-based. A parallel program is described as a *process network*, *i.e.* a set of processes communicating by channels and each executing a sequence of instructions. In this section, we describe how such a process network can be built from the skeleton tree describing an application by means of a simple process algebra formalized by a set of *production rules*.

3.1 Process network description

Formally, a *process network* (PN) is a triple $\pi = \langle P, I, O \rangle$ where

- P is a set of labeled processes, i.e. pairs (pid, σ) where pid is a (unique) process id and σ a triple containing: a list[a] of *predecessors* (pids of processes p for which a communication channel exists from p to the current process), a list of *successors* (pids of processes p for which a communication channel exists from the current process to p) and a descriptor Δ. We note $\mathcal{L}(\pi)$ the set of pids of a process network π. For a process p, its predecessors, successors and descriptor will be denoted $\mathcal{I}(p)$, $\mathcal{O}(p)$ et $\delta(p)$ respectively.
- $I(\pi) \subseteq \mathcal{L}(\pi)$ denotes the set of *source* processes for the network π (i.e. the set of processes p for which $\mathcal{I}(p) = \emptyset$)
- $O(\pi) \subseteq \mathcal{L}(\pi)$ denotes the set of *sink* processes for the network π (i.e. the set of processes p for which $\mathcal{O}(p) = \emptyset$)

The process descriptor Δ is a pair $(instrs, kind)$ where $instrs$ is a sequence of (abstract) instructions and $kind$ a flag (the meaning of the $kind$ flag will be explained in section 3.2).

$$\begin{array}{lll} \Delta & ::= & \langle \textit{instrs}, \textit{kind} \rangle \\ instrs & ::= & \textit{instr}_1, \ldots, \textit{instr}_n \\ \textit{kind} & ::= & \mathsf{Regular} \mid \mathsf{FarmM} \end{array}$$

The sequence of instructions describing the process behavior is implicitly iterated (processes never terminate). Instructions use *implicit addressing*, with each process holding four variables named vi, vo, q and iws. The instruction set is given below. In the subsequent explanations, p designates the process executing the instruction.

$$\begin{array}{lll} \textit{instr} & ::= & \mathsf{SendTo} \mid \mathsf{RecvFrom} \mid \mathsf{CallFn\ fid} \mid \mathsf{RecvFromAny} \mid \mathsf{SendToQ} \mid \\ & & \mathsf{Ifq}\ \textit{instrs}_1\ \textit{instrs}_2 \mid \mathsf{GetIdleW} \mid \mathsf{UpdateWs} \end{array}$$

The SendTo instruction sends the contents of variable vo to the process whose *pid* is given in $\mathcal{O}(p)$. The RecvFrom instruction receives data from the process whose *pid* is given in $\mathcal{O}(p)$ and puts it in the variable vi. The CallFn instruction performs a computation by calling a sequential function. This function takes one argument (in vi) and produces one result (in vo). The RecvFromAny instruction waits (non-deterministically) data from the set of processes whose pids are given in $\mathcal{I}(p)$. The received data is placed in the variable vi and the pid of the actual sending process in the variable q. The SendToQ instructions sends the contents of variable vo to the process whose pid is given by variable q. The Ifq instruction compares the value contained in variable q to the first pid listed in $\mathcal{I}(p)$. If case of equality, the instruction sequence $\textit{instrs}_1$ is executed; else $\textit{instrs}_2$ is executed. The UpdateWs instruction reads variable q and updates the variable iws accordingly. The variable iws maintains the list of idle workers for FARM master processes. The GetIdleW retrieves a process id from the iws list and places it in the variable q. Together, these two instructions encapsulate the policy used in a FARM skeleton to allocate data to workers. They are not detailed here further.

[a] Note that this is really a list, and not a set, since the order is relevant.

3.2 A basic process network algebra

The following notation will be used. If $\mathcal{E}$ is a set, we denote by $\mathcal{E}[e \leftarrow e']$ the set obtained by replacing e by e' (assuming $\mathcal{E}[e \leftarrow e'] = \mathcal{E}$ if $e \notin \mathcal{E}$). This notation is left-associative: $\mathcal{E}[e \leftarrow e'][f \leftarrow f']$ means $(\mathcal{E}[e \leftarrow e'])[f \leftarrow f']$. If $e_1, \ldots, e_m$ is an indexed subset of $\mathcal{E}$ and $\phi : \mathcal{E} \rightarrow \mathcal{E}$ a function, we will note $\mathcal{E}[e_i \leftarrow \phi(e_i)]_{i=1..m}$ the set $(\ldots((\mathcal{E}[e_1 \leftarrow \phi(e_1)])[e_2 \leftarrow \phi(e_2)])\ldots)[e_m \leftarrow \phi(e_m)]$. Except when explicitly indicated, we will note $I(\pi_k) = \{i_k^1, \ldots, i_k^m\}$ and $O(\pi_k) = \{o_k^1, \ldots, o_k^n\}$. For concision, the lists $\mathcal{I}(o_k^j)$ et $\mathcal{O}(i_k^j)$ will be noted s_k^j et d_k^j respectively. For lists, we define a concatenation operation ++ as usual : if $l_1 = [e_1^1, \ldots, e_1^m]$ and $l_2 = [e_2^1, \ldots, e_2^n]$ then l_1++$l_2 = [e_1^1, \ldots, e_1^m, e_2^1, \ldots; e_2^n]$. The empty list is noted []. The length of list l (resp. cardinal of a set l) is noted $|l|$.

The $\lceil . \rceil$ operator creates a process network containing a single process from a process descriptor, using the function NEW() to provide "fresh" process ids :

$$\frac{\delta \in \Delta \qquad l = \text{NEW}()}{\lceil \delta \rceil = \langle \{(l, \langle [], [], \delta \rangle)\}, \{l\}, \{l\} \rangle} \qquad \text{(SINGL)}$$

The $\bullet$ operation "serializes" two process networks, by connecting outputs of the first to the inputs of the second :

$$\frac{\pi_i = \langle P_i, I_i, O_i \rangle \ (i = 1, 2) \qquad |O_1| = |I_2| = m}{\pi_1 \bullet \pi_2 = \langle (P_1 \cup P_2)[(o_1^j, \sigma) \leftarrow \phi_d((o_1^j, \sigma), i_2^j)]_{j=1\ldots m}[(i_2^j, \sigma) \leftarrow \phi_s((i_2^j, \sigma), o_1^j)]_{j=1\ldots m}, I_1, O_2 \rangle} \qquad \text{(SERIAL)}$$

This rule uses two auxiliary functions ϕ_s and ϕ_d defined as follows :

$\phi_s((p, \langle s, d, \langle \delta, \mathsf{Regular} \rangle \rangle), p') = (p, \langle [p']\text{++}s, d, \langle [\mathsf{RecvFrom}]\text{++}\delta, \mathsf{Regular} \rangle \rangle)$
$\phi_d((p, \langle s, d, \langle \delta, \mathsf{Regular} \rangle \rangle), p') = (p, \langle s, d\text{++}[p'], \langle \delta\text{++}[\mathsf{SendTo}], \mathsf{Regular} \rangle \rangle)$
$\phi_s((p, \langle s, d, \langle \delta, \mathsf{FarmM} \rangle \rangle), p') = (p, \langle [p']\text{++}s, d, \langle \delta, \mathsf{FarmM} \rangle \rangle)$
$\phi_d((p, \langle s, d, \langle \delta, \mathsf{FarmM} \rangle \rangle), p') = (p, \langle s, d\text{++}[p'], \langle \delta, \mathsf{FarmM} \rangle \rangle)$

The function ϕ_s (resp. ϕ_d) adds a process p' as a predecessor (resp. successor) to process p and updates accordingly its instruction list. This involves prepending (resp. appending) a RecvFrom (resp. SendTo) instruction) to this instruction list, *except for* FARM masters (identified by the FarmM kind flag), for which the instruction list is not modified.

The $\|$ operation puts two process networks in parallel, merging their inputs and outputs respectively.

$$\frac{\pi_i = \langle P_i, I_i, O_i \rangle \ (i = 1, 2)}{\pi_1 \parallel \pi_2 = \langle P_1 \cup P_2, I_1 \cup I_2, O_1 \cup O_2 \rangle} \qquad \text{(PAR)}$$

The $\bowtie$ operation merges two process networks by connecting each input and output of the second to the output of the first :

$$\frac{\pi_i = \langle P_i, I_i, O_i\rangle\ (i=1,2) \qquad |O_1| = 1 \qquad |I_2| = m \qquad |O_2| = n}{\begin{array}{c}\pi_1 \bowtie \pi_2 = \langle (P_1 \cup P_2)[(o_1,\sigma) \leftarrow \Phi((o_1,\sigma), I(\pi_2), O(\pi_2))][(i_2^j,\sigma) \leftarrow \phi_s((i_2^j,\sigma), o_1)]_{j=1\ldots m} \\ [(o_2^j,\sigma) \leftarrow \phi_d((i_2^j,\sigma), o_1)]_{j=1\ldots n}, \\ I_1, O_1\rangle\end{array}}$$

(JOIN)

where $\Phi(p, ps_s, ps_d) = \Phi_s(\Phi_d(p, ps_d), ps_s)$ and Φ_s (resp. Φ_d) generalizes the function ϕ_s (resp. ϕ_d) to a list of processes :

$$\Phi_s(p, [p_1, \ldots, p_n]) = \phi_s(\ldots, \phi_s(\phi_s(p, p_1), p_2), \ldots, p_n)$$
$$\Phi_d(p, [p_1, \ldots, p_n]) = \phi_d(\ldots, \phi_d(\phi_d(p, p_1), p_2), \ldots, p_n)$$

Skeletons can now be defined in terms of the operations defined above, using the following conversion function $\mathcal{C}^b$:

$$\begin{aligned}
\mathcal{C}[[\mathsf{Seq}\ f]] &= \lceil f \rceil \\
\mathcal{C}[[\mathsf{Pipe}\ \Sigma_1 \ldots \Sigma_n]] &= \mathcal{C}[[\Sigma_1]] \bullet \ldots \bullet \mathcal{C}[[\Sigma_n]] \\
\mathcal{C}[[\mathsf{Farm}\ n\ \Sigma]] &= \lceil \mathsf{FarmM} \rceil \bowtie (\mathcal{C}[[\Sigma]]_1 \parallel \ldots \parallel \mathcal{C}[[\Sigma]]_n) \\
\mathcal{C}[[\mathsf{Scm}\ m\ f_s\ \Sigma\ f_m]] &= \mathcal{C}[[\mathsf{Seq}\ f_s]] \triangleleft (\mathcal{C}[[\Sigma]]_1 \parallel \ldots \parallel \mathcal{C}[[\Sigma]]_m) \triangleright \mathcal{C}[[\mathsf{Seq}\ f_m]] \\
\mathcal{C}[[\mathsf{Pardo}\ \Sigma_1\ \ldots \Sigma_n]] &= \mathcal{C}[[\Sigma_1]] \parallel \ldots \parallel \mathcal{C}[[\Sigma_n]]
\end{aligned}$$

where FarmM is a process descriptor predefined as :

$$\Delta(\mathsf{FarmM}) = \langle [\mathsf{RecvFromAny}; \mathsf{Ifq}\ [\mathsf{GetIdleW}; \mathsf{SendToQ}]\ [\mathsf{UpdateWs}; \mathsf{SendTo}]], \mathsf{FarmM}\rangle$$

4 Implementation

We now explain how the production rules and the conversion function $\mathcal{C}$ introduced in the previous section can be implemented as a compile-time process. The process itself is sketched on figure 1.

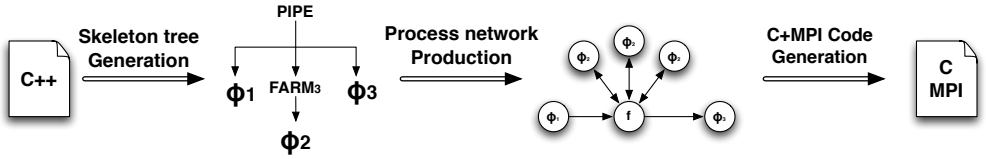

Figure 1. QUAFF code generation process

[b] The production rules for the operators $\triangleleft$ and $\triangleright$, used in the definition of the Scm skeleton have been omitted due to space constraints.

It consists of three steps: generating the skeleton tree, turning this structure into a process network and producing the C+MPI target code. These three steps are carried out at compile-time. The key idea is that each *object* of the formal semantics defined in section 3 is encoded as a *type* in the implementation language. Production rules, in particular, are encoded as *meta-programs* taking arguments and producing results as C++ types. The whole process is illustrated with a very simple application consisting of a two-stages pipeline. Using QUAFF, this application is written:

```
run( pipeline(seq(F1),seq(F2)) );
```

where `F1` and `F2` are registered sequential functions, as illustrated in listing 1.

4.1 Generating the skeleton tree

For each skeleton, the corresponding function at the API level returns a value the type of which is a compile-time representation of the skeleton. Here's, for example, the definition of the `seq` and `farm` functions:

Listing 2. Skeleton constructors for SEQ and FARM

```
template<class F>
Seq<F> seq(const F&) { return Seq<F>(); }

template<int N, class W>
Farm<N,W> farm(const W&) { return Farm<N,W>(); }
```

In the two-stages pipeline example, the call `run()` at the API level needs to call the `pipeline` function, and therefore the `seq` function. This will generate the following residual code, in which the argument to the `run` function is an instance of a type encoding of the skeleton tree:

```
run( Serial< Seq<F1>, Seq<F2> >() );
```

This *template* now carries informations about the skeleton tree in a form usable by our meta-functions.

4.2 Producing the process network

We first give, in listing 3, the type encoding of the *process network*, *labeled process* and *process descriptor* objects. Each of these types is defined as a simple template container, with arguments statically encoding the aggregated objects. In the `process_network` type, the `P`, `I` and `O` fields are compile-time lists of process `ID`s. Technically speaking, process `ID`s themselves are encoded as type-embedded integral constants and type lists are built and manipulated using the `BOOST::MPL` library[3]. In the `process` type, the `input_type` and `output_type` encode the argument and return type of the associated user-defined function. In the `descriptor` type, the `i_pids` and `o_pids` fields encode the list of successors and predecessors respectively and the `instrs` field encodes the list of instructions executed by the process.

Listing 3. process_network, process and descriptor related data types

```
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;
};
```

The `run` function now has to convert the type describing the skeleton tree produced by the previous step into a type describing the corresponding process network (*i.e.* to implement the $\mathcal{C}$ function specified in section 3.2).

Listing 4. The `run` function

```
template<class SKL> void run( const SKL& )
{
  typedef typename convert<SKL>::type p_net;
  p_net::Run();
}
```

This code shows that `run` simply extracts type informations from its *template* parameter and pass it through the `convert` meta-function. This function is statically overloaded for each skeleton constructor. Listing 5 shows the overloaded meta-function for the pipeline skeleton.

Listing 5. `pipeline` *template* conversion

```
template<class S0,class S1,class ID> struct convert<Serial<S0,S1>,ID>
{
  typedef Serial<S1,mpl::void_>                     tail;
  typedef typename convert<S0,ID>::type             proc1;
  typedef typename convert<S0,ID>::new_id           next_id;
  typedef typename convert<tail,next_id>::new_id    new_id;
  typedef typename convert<tail,next_id>::type      proc2;
  typedef typename rule_serial<proc1,proc2>::type   type;
};
```

The `convert` meta-function extracts the skeleton sub-trees from `S0` and `S1`, converts them into process networks, computes a new process ID and applies the appropriate production rule (SERIAL) to generate a new process network embedded in the `type` *typedef*.

The production rules are also implemented as meta-programs. The *template* equivalent of the rule SERIAL defined in section 3.2, for example, is given in listing 6. This template takes as parameters the types encoding the two process networks to serialize. The type encoding the resulting process network is then built incrementally, by means of successive type definitions, each type definition precisely encoding a value definition of the formal production rule and by using MPL meta-function like `transform` which is a meta-programmed iterative function application or `copy` which is used in conjunction with the `back_inserter` manipulator to concatenate two lists of process networks.

Listing 6. The meta-programmed (SERIAL) rule

```
template<class P1, class P2> struct rule_serial
{
  // Get list of processus and I/O from P1 and P2
  typedef typename P1::process                                          proc1;
  typedef typename P2::process                                          proc2;
  typedef typename P1::inputs                                           i1;
  typedef typename P2::inputs                                           i2;
  typedef typename P1::outputs                                          o1;
  typedef typename P2::outputs                                          o2;

  // Add new process graph into the new process network
  typedef typename mpl::transform< proc1, phi_d<_1,o1,i2> >::type       np1;
  typedef typename mpl::transform< proc2, phi_s<_1,i2,o1> >::type       np2;
  typedef typename mpl::copy<np2, mpl::back_inserter<np1> >::type       process;

  // Process network definition
  typedef process_network<process,i1,o2>                                type;
};
```

4.3 Generating parallel application code

The last step consists in turning the process network representation into C+MPI code. This transformation is triggered at the end of the `run` function. The `Run` method of the `process_network` class created by the application of `convert` sequentially instantiates and executes each macro-instruction of its descriptor. The actual process of turning the macro-instructions list into a C+MPI code is based on tuple generation similar to the one used in the previous QUAFF implementation[1]. Each instance is then able to check if its PID matches the actual process rank and executes its code. For our two-stages pipeline, the residual code looks as shown in listing 7

5 Experimental results

We have assessed the impact of this implementation technique by measuring the overhead ρ introduced by QUAFF on the *completion time* over hand-written C+MPI code for both

Listing 7. Generated code for the two stage pipeline

```
if( Rank() == 0 )
{
  do {
    out = F1();
    MPI_Send(&out,1,MPI_INT,1,0,MPI_COMM_WORLD);
  } while( isValid(out) )
}
else if( Rank() == 1 )
{
  do {
    MPI_Recv(&in,1,MPI_INT,0,0,MPI_COMM_WORLD,&s);
    F2(in);
  } while( isValid(in) )
}
```

single skeleton application and when skeletons are nested at arbitrary level. For single skeleton tests, we observe the effect of two parameters: τ, the execution time of the inner sequential function and N, the "size" of the skeleton (number of stages for PIPELINE, number of workers for FARM and SCM). The test case for nesting skeletons involved nesting P FARM skeletons, each having ω workers. Results were obtained on a PowerPC G5 cluster with 30 processors and for $N = 2 - 30$ and $\tau = 1ms, 10ms, 100ms, 1s$.

For PIPELINE, ρ stays under 2%. For FARM and SCM, ρ is no greater than 3% and becomes negligible for $N > 8$ or $\tau > 10ms$. For the nesting test, worst case is obtained with $P = 4$ and $\omega = 2$. In this case, ρ decreases from 7% to 3% when τ increases from $10^{-3}s$ to $1s$.

6 Related work

The idea of relying on a process network as an intermediate representation for skeletal programs is not new; in fact, several implementations of skeleton-based parallel programming libraries, such as P3L[5], implicitly rely on such a representation. But, the process of translating the skeleton tree into such a network has never been formalized before. Aldinucci[8] proposed a formal operational semantics for skeleton-based programs but, contrary to QUAFF , the actual implementation relies on a dynamic runtime. Thus, to our best knowledge, our work is the first to both rely on a formal approach of skeleton compilation while also offering a performance on par with hand-coded C+MPI implementations.

On the other hand, using generative programming and meta-programming for implementing parallel applications and libraries is currently an upcoming trend. Works by Czarnecki *et al.*[9],Puschel and al.[10], Hammond[11], Langhammer and Hermann[12] uses meta-programming in `MetaOCaml`[13] or `Template Haskell` to generate parallel *domain specific languages* for solving problem like signal processing optimizations or parallel process scheduling on MIMD machines thus making generative programming a valid technique to solve realistic problems. Our work can be viewed as a specific application of these general techniques.

7 Conclusion

In this paper, we have shown how generative and meta-programming techniques can be applied to the implementation of a skeleton-based parallel programming library . The resulting library, QUAFF , both offers a high level of abstraction and produces high performance code by performing most of the high to low-level transformations at compile-time rather than run-time. The implementation is derived directly from a set of explicit production rules, in a semantic-oriented style. It is therefore formally sounded and much more amenable to proofs or extensions.

References

1. Joel Falcou, Jocelyn. Sérot, Thierry Chateau, and Jean-Thierry Lapresté. QUAFF: Efficient C++ Design for Parallel Skeletons. *Parallel Computing*, 32:604–615, 2006.
2. Todd Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995. Reprinted in C++ Gems, ed. Stanley Lippman.
3. David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.
4. Herbert Kuchen. A skeleton library. In *Euro-Par '02: Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, pages 620–629, London, UK, 2002. Springer-Verlag.
5. B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P3l: A structured high level programming language and its structured support. *Concurrency: Practice and Experience*, 7:225–255, 1995.
6. M. Cole. *Research Directions in Parallel Functional Programming*, chapter 13, Algorithmic skeletons. Springer, 1999.
7. J. Sérot and D. Ginhac. Skeletons for parallel image processing: an overview of the skipper project. *Parallel Computing*, 28(12):1685–1708, 2002.
8. Marco Aldinucci and Marco Danelutto. Skeleton based parallel programming: functional and parallel semantic in a single shot. *Computer Languages, Systems and Structures*, 2006.
9. K. Czarnecki, J.T. O'Donnell, J. Striegnitz, and W. Taha. Dsl implementation in metaocaml, template haskell and c++. In C. Lengauer, D. Batory, C. Consel, and M. Odersky, editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 51–72. Springer-Verlag, 2004.
10. Markus Puschel, José Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nick Rizzolo. SPIRAL: Code Generation for DSP Transforms. In *Proceedings of the IEEE special issue on "Program Generation, Optimization, and Adaptation"*, 2005.
11. K. Hammond, R. Loogen, and J. Berhold. Automatic Skeletons in Template Haskell. In *Proceedings of 2003 Workshop on High Level Parallel Programming, Paris, France*, June 2003.
12. Christoph A. Herrmann and Tobias Langhammer. Combining partial evaluation and staged interpretation in the implementation of domain-specific languages. *Sci. Comput. Program.*, 62(1):47–65, 2006.
13. MetaOCaml. A compiled, type-safe multi-stage programming language. Available online from http://www.metaocaml.org/, 2003.