QUAFF: efficient C++ design for parallel skeletons

J. Falcou *, J. Sérot, T. Chateau, J.T. Lapresté
LASMEA, UMR6602 UBP/CNRS, Campus des Cézeaux, 63177 Aubière, France

Received 29 December 2005; received in revised form 24 May 2006; accepted 6 June 2006
Available online 2 August 2006

Abstract

We present QUAFF, a new skeleton-based parallel programming library. Its main originality is to rely on C++ template meta-programming techniques to achieve high efficiency. In particular, by performing most of skeleton instantiation and optimization at compile-time, QUAFF can keep the overhead traditionally associated to object-oriented implementations of skeleton-based parallel programming libraries very small. This is not done at the expense of expressivity. This is demonstrated in this paper by several applications, including a full-fledged, realistic real-time vision application.

© 2006 Elsevier B.V. All rights reserved.

Keywords: Parallel skeletons; Template meta-programming; Computer vision; MPI; C++

1. Introduction

Modern parallel programming on MIMD machines is usually carried out using message-passing libraries. These libraries, such as PVM or MPI, provide a standardized, platform independent way to build parallel applications. However, manipulating such low level frameworks is difficult and error prone. Deadlocks and other common undesired behaviors make parallel software development very slow compared to classic, sequential one. Algorithmic skeletons [5,6] have been proposed as a solution to these problems. They are typical parallel patterns that can be implemented once on a given platform. Building parallel software using algorithmic skeletons boils down to combining skeletons and sequential code fragments. In practice, approaches based upon skeletons can be divided into three main classes:

- New languages embedding skeleton patterns in their syntax. This approach can offer good performances but requires the programmer to learn a new language, which can viewed as an obstacle to the adoption of this paradigm.
- Parallel compilers for an existing language. Such compilers aim at identifying parallel structures in existing sequential code and use specific parallel implementation for those structures [20].
- High-level libraries for an existing language. Such solutions are more easily accepted by developers as they allow them to reuse existing code and to work with a familiar environment.

* Corresponding author. Tel.: +33 4 73 407230; fax: +33 4 73 407262.
E-mail address: joel.FALCOU@lasmea.univ-bpclermont.fr (J. Falcou).
We focus here on approaches based upon high-level libraries for an existing language. This is the way taken, for example, by the BSMLLib [17], Lithium [9,2], eSkel [6] and Muesli [15] projects. The most challenging issue, for such a library is to find a good trade-off between readability and efficiency. By readability we mean the ability to express parallelism with a minimum concern for implementation details. By efficiency we mean the ability to produce code whose performances can stay on the par with those obtained using a low-level message passing like MPI. These features are clearly in tension with each other. For example, eSkel can produce very efficient code but at the expense of a rather low-level API (with a lot of MPI-specific idioms visible). By contrast, a library such as Muesli exposes a much more abstract and simple API but incurs a significantly higher overhead at runtime. This overhead can be mostly explained by the fact that this library uses an abstract class hierarchy to embed user-defined tasks within the skeleton structure of the application, resulting in repeated virtual function calls at run-time. This is unfortunate because, within skeleton-based parallel programming models, the overall structure of the application, i.e., the combination of parallel skeletons and user-defined sequential functions is essentially static.

The library described in this paper, named QUAFF, aims at reducing the aforementioned tension. For this, it relies on C++ compilation techniques such as template meta-programming to reduce the runtime overhead of classical object-oriented implementations of skeleton-based parallel programming libraries to the strict minimum, while keeping a high-level of expressivity and readability. QUAFF also enforces the re-usability of legacy code or third party libraries by limiting its impact on existing code.

This paper is organized as follows. The QUAFF programming model is introduced in Section 2, with simple examples. Section 3 presents the implementation, with QUAFF, of a full-fledged vision application, showing its ability to handle complex, realistic situations while still producing very efficient code. Section 4 details the implementation techniques used for turning QUAFF programs into optimized MPI code. Section 5 is a short review of related work and Section 6 concludes the paper.

2. QUAFF programming model

2.1. A first example

In this section, we describe the encoding, with QUAFF, of a very simple application which performs matrix multiplication using a classical domain decomposition approach with a \texttt{scm} (split-compute-merge) skeleton. The QUAFF code of the application appears on listing 1. The corresponding task graph is given on Fig. 1.

Lines 2–4 define the three QUAFF tasks which are passed as arguments to the \texttt{scm} skeleton. A QUAFF task is defined by specifying a function and a pair of input/output types.\footnote{A predefined type – \texttt{none\_t} – is provided to declare tasks without input and/or output (bound to source and sink nodes in the task graph, respectively).} The task function itself can be either a

```
Listing 1: Sample QUAFF application
1 // User-defined tasks registration
2 typedef task<CSlice, none_t, matrix> slice;
3 typedef function(FMatMul, matrix, matrix) mul;
4 typedef task<CMerge, matrix, none_t> merge;
5
6 // Skeleton definition
7 typedef scm<slice, worker< repeat<4, mul> >, merge> app;
8
9 // Application instantiation
10 application< app > myAppInstance;
11 myAppInstance.run();
```

\footnote{These projects are described in Section 5.}
function or a C++ class exposing a proper ( ) operator.\footnote{A \textit{functor} in C++ terminology.} Here the \texttt{slice} task is used to split the input matrix into slices (block of rows here), the \texttt{mul} task multiplies one slice (by a constant matrix in this case) and the \texttt{merge} task merges the resulting blocks to build the result matrix.

The \texttt{task} template and \texttt{function} macro-definition respectively take care of generating all the wrappers needed to call the supplied functor or function from the \texttt{QUAFF} framework. This limits the amount of code to be written to reuse existing or legacy sequential code in parallel applications to the strict minimum. Prototypes of the corresponding functions could be defined as in listing 2. Here, \texttt{CSlice} and \texttt{CMerge} are defined as C++ functors and \texttt{FMatMul} as a C function.

Listing 2: Sample function and functor prototypes

\begin{verbatim}
1 list<matrix> CSlice::operator()();
2 matrix FMatMul (const matrix&);
3 void CMerge::operator()(const list<matrix>&);
\end{verbatim}

Line 7 of listing 1 defines the actual \texttt{scm} skeleton: the first and third arguments specify the splitting and merging tasks, the second argument specifies the computing function and the number of nodes on which this computing function will run in parallel. Lines 10–11 define the application main type and instantiate it.

The key point, and this is actually \texttt{QUAFF} most distinctive feature, is that the application structure (i.e., its task graph) is entirely encoded by means of type definitions. These type definitions will be processed \textit{at compile time} to produce an optimized message-passing code. Details on how this is actually carried out will be given in Section 4.

2.2. \texttt{QUAFF} skeletons

The previous example has exhibited the \texttt{scm} skeleton, a domain decomposition skeleton well suited to fixed data parallelism. \texttt{QUAFF} provides three other parallel skeletons:

- The \texttt{pipeline} skeleton models situations in which a sequence of successive tasks are dispatched and run in parallel on a set of nodes. Each task receives its input from its predecessor and sends its result to the next processor in the pipeline.

\begin{center}
\begin{tabular}{c}
\texttt{SLICE} \hspace{1cm} \texttt{MULTIPLY} \hspace{1cm} \texttt{MERGE} \\
\texttt{MULTIPLY} \hspace{1cm} \texttt{MULTIPLY} \\
\end{tabular}
\end{center}

Fig. 1. Matrix multiplication application graph.
• The \texttt{farm} skeleton handles variable data-parallelism, \textit{i.e.}, situations in which a list of items has to be processed by pool of workers following some kind of load-balancing strategy. Each item is dispatched to the first non-busy \texttt{worker} and a result is sent to implicit collector each time a \texttt{worker} task is completed. Current implementation supports options to specify an explicit \texttt{dealer} and \texttt{collector} tasks [18] to replace the implicit ones.

• The \texttt{pardo} skeleton is a generic \textit{ad hoc} skeleton for running \(N\) different tasks on a subset of \(N\) processors. No implicit communication is provided and all synchronizations or data transfers should be explicitly carried out by the inner tasks. This skeleton is a way to integrate \textit{ad hoc} parallelism as defined in [6] in QUAFF applications.

Apart from these parallel skeletons, QUAFF also provides constructs allowing tasks or skeletons to be composed sequentially or conditionally in applications. Section 3, for instance, will illustrate the use of the \texttt{sequence} construct to build a sequence of tasks. The \texttt{condition} and \texttt{branch} constructs (not illustrated here) are used to provide static \texttt{if...else} and \texttt{switch...case} equivalents. These constructs have a syntax similar to the parallel skeletons one.

2.3. Skeleton nesting

Skeletons nesting is naturally supported in QUAFF. This is because, once instantiated, all skeletons are valid functors and consequently can be used as arguments to other skeleton templates. This is illustrated in Fig. 2 with an application which performs edge detection on streams of images.

Listing 3: Skeleton nesting

```c
1 typedef task< CGetImg , none_t , image > get_img;
2 typedef task< CSobel , image , image > sobel;
3 typedef task< CSaveImg , image , none_t > save_img;
4 typedef farm< worker< repeat< sobel,4 >> > inner;
5 typedef pipeline< stage< get_img, inner, save_img > > pipe;
6 typedef application< pipe > main_app;
```

![Fig. 2. Application graph for a sample skeleton nesting.](image)
The whole application is defined as a pipeline, whose second stage is a farm. Listing 3 shows how this nesting is defined.

As for the first example, lines 1–3 define the application-specific sequential tasks. The inner farm skeleton is defined at line 5, by giving the task computed by each worker and the total number of workers. The pipeline itself is defined at line 6, using the farm skeleton as its second stage.

2.4. Sample results

To assess the validity of our approach, we performed tests on the simple applications previously described in this section. Our goal was to evaluate the overhead introduced by the QUAFF library by comparing the execution time obtained with the QUAFF version of these applications (as given in listing 1 and 3) with those obtained with hand-crafted versions written directly with MPI. All the results presented were obtained on an Apple G5 XServe cluster containing 14 nodes with two 2 GHz processors running Mac OS 10.3 and the LAM/MPI v7.1.2 MPI implementation. Results appear in table Fig. 3. They show that, for these simple applications, the overhead introduced by QUAFF is very small (it does not exceed 3%).

3. Experimental results

Section 2 has demonstrated the ability for QUAFF to produce efficient code for simple applications. We now focus on expressivity issues. In this section the implementation of a realistic application with QUAFF is presented. By realistic, we mean an application solving a “real” problem, by contrast to the code samples in Section 2, whose goal is only to illustrate and demonstrate programming features. This application, taken from the computer vision domain, performs real-time 3D reconstruction from a stereoscopic video stream. It is frequently used in high-level vision tasks such as autonomous robot navigation, motion capture or human-computer interfaces. From a parallel implementation perspective, it raises several interesting challenges. First, it involves both data and control parallelism. Second, it must operate on the fly on digital video streams coming directly from cameras, and not by reading inputs (resp. writing results) on disk. Third, its overall structure follows a client–server model, in which a specific node runs a GUI for real-time user interaction and displaying of results.

An execution snapshot of the corresponding application (as shown by the GUI) is seen on Fig. 4. Video inputs are on the left (right and left cameras) and the set of reconstructed 3D points appears on the right.

3.1. 3D reconstruction algorithm

The algorithm consists in four steps:

1. **Image rectification (RECTIF):** Video frames are warped by mapping epipolar lines onto image pixels rows [11]. This simplifies further point matching by limiting search to a single pixel row.

2. **Key-point detection (HARRIS):** Point of interest to be used as seeds for matching stage are extracted from the rectified stereo pairs by a Harris and Stephen corner detector [13].

3. **Key-point matching (MATCH):** Each 2D key-point from the left image (resp. right image) is matched with a 2D key-point from the right image (resp. left image) by using a maximum likelihood search using a zero normalized cross correlation measure.

4. **3D reconstruction (BUILD):** Correctly matched key-points pairs are then triangulated [14] to output the 3D position of the corresponding point.
3.2. Parallelization strategy

The application is built on a client/server model (see Fig. 5). The “server” part performs the reconstruction algorithm and merge the final 3D points list. This part involves fixed data parallelism and is therefore implemented with a scm skeleton. Each worker in this skeleton runs the 3D reconstruction algorithm (described as a sequence of tasks) on a slice of each input image and results obtained on each slice are merged. The “client” part (executing on node 0) also gets the input images but only takes care of displaying results. An ad hoc synchronization step is performed in the get_frame task to ensure that all computing nodes apply the algorithm on the same pair of images. At the end of the scm skeleton, another ad hoc communication sends the results of the reconstruction to the “client” node.

3.3. Implementation with QUAFF

The application source code is given in listing 4. The reconstruction algorithm sequence is defined at lines 1–2 and inserted into a scm skeleton at line 3. At lines 5–7, we define the sequence of tasks needed to run both the computing tasks and the display task on different nodes.

Fig. 5. 3D reconstruction application graph.
3.4. Performance

Timing results for this application are given in Fig. 6. As in Section 2.4, two numbers are given: one obtained using QUAFF and the other with a hand-crafted version of the application written directly with MPI. The ratio between these numbers gives the overhead introduced by QUAFF. These three numbers are reported for several configurations of the cluster including 2 to 28 processors. The input cameras deliver 640\times 480 images at 30 images/s. The application typically outputs 40,000 reconstructed points per second.

These results call for the following remarks. First, the application is able to process 25 images/s with 24 processors (with a relative speedup of 21.5). This shows that using a high-level library such as QUAFF does not preclude real-time performance. Second, the overhead induced by QUAFF on this application never exceeds 12\%, which is small. Third, the ability offered by QUAFF to reuse existing sequential C/C++ code and to quickly test various parallelization strategies has reduced the implementation effort by one order of magnitude at least (compared to hand-written MPI coding). Another means of assessing the value added by QUAFF is to compare the 10 lines of the QUAFF source code with the 100+ lines of MPI code that the programmer has to write with a direct MPI implementation.

4. QUAFF implementation

As stated in Section 2, the major issue with class-based libraries is the high overhead induced by virtual function calls. Recent compilers are able to reduce this overhead by performing various aggressive optimizations. However, none of them is able to optimize code across function or method boundaries. In the classic polymorphic library model, this leads to very efficient code at the function level, but poor performances at the program level.

One solution is to write code that forces the compiler to generate specific code at specific locations. This can actually be achieved by resorting to various C++ template meta-programming techniques. We propose such a solution applied to the problem of generating efficient code for skeletons.

4.1. Template-based meta-programming

Beside the usual type genericity, the introduction of templates to C++ added a facility whereby the compiler can act as an interpreter. This mechanism differs greatly from C++ preprocessor macros as the whole template
system can be used as a Turing complete language [23]. This makes it possible to write, in a subset of C++, programs which are interpreted at compile time. The first use of this technique was described by Unruh in [21]. He exhibited programs which did not have to be executed but, instead, generated their output at compile time as warning messages. Although this technique might appear as a simple C++ trick, it becomes powerful when combined with normal C++ code. In this hybrid approach, the source code actually contains two programs: the normal C++ run-time program, and a template meta-program which runs at compile time. Template meta-programs encode an algorithm as a set of production rules and force the compiler to follow them.

Various projects [19,1,8] used this feature to create a new kind of libraries that do more than just providing collections of functions or classes. These libraries, called Active Libraries [24], take care of their own optimization by using template meta-programming to guide the compiler. QUAFF is built on this model, using meta-programs to define, generate and optimize skeleton based applications. The next section will illustrate how this is carried out on a simple example.

4.2. Application to skeleton instantiation

Consider the following code:

Listing 5: Sample code using pipeline

```cpp
typedef pipeline< stage<T1,T2,T3> > pipe;
typedef application< pipe > app;
app myApp;
```

It defines a three stages pipeline and an application using this pipeline. The idea is to perform the instantiation of the pipeline structure at compile time. To perform this instantiation, the compiler needs the type of the inner tasks. So, very basically, what is needed is a mean of manipulating types at compile time. Technically, this is challenging, because types are not first class objects in C++. This problem can be solved, however, by resorting to classical C++ idiom known as the typelist [3,22] idiom. It provides a way to store a succession of types into a static container and to provide convenient meta-programs to manipulate them, turning typelist into a complete compile-time list structure. The implementation of typelist is given in listing 6. It is an open-ended, recursive template that can hold any number of types.

Listing 6: Typelist implementation

```cpp
template<class H, class T = null_t> struct typelist {
    typedef H head_t;
    typedef T tail_t;
};
```

Listing 7: Typelist declaration

```cpp
typelist<T1, typelist<T2, typelist<T3> >> stages_t;
```

The typelist idiom can be used to generate the static list of skeletons and tasks describing a parallel application. For example, the stage class in the pipeline definition gathers the task types and build the corresponding typelist for the pipeline example (listing 7).

Listing 8: Tuple declaration

```cpp
template<class TL> struct tuple : public tuple<TL::tail_t> {
    typename TL::head_t mField;
};
template<> struct tuple<null_t> {};
```
The tuple class recursively instantiates the contents of its typelist parameter by using a recursive inheritance pattern. For the pipeline example, the tuple instantiation leads to the following structure.

Listing 9: Pseudocode for tuple instantiation

```c
struct tuple
{
    T1 tuple<T1>::mField;
    T2 tuple<T2>::mField;
    T3 tuple<T3>::mField;
};
```

Using tuple, each skeleton can now instantiate the class stored in its typelist as illustrated on listing 10. Basically, it retrieves the typelist contained into the stage type and builds a tuple instance out of it.

Listing 10: Pipeline class

```c
template<class STG> class pipeline
{
    typedef typename STG::typelist_t list_t;
    typedef tuple<list_t> stages_t;
    stages_t mStages;
};
```

The mStages member will be used whenever the pipeline has to access one of its task. Meta-programs are provided to retrieve a particular elements in a tuple or to run specific task method.

Once the pipeline stage list has been generated, the instantiation of the main application class forces the compiler to inline the skeleton and task code. Using a meta-programmed loop, each element of the tuple corresponding to the skeleton is inlined and the skeleton communication harness is instantiated. For the pipeline example, the resulting code looks like in listing 11.

Listing 11: Inlined pipeline code

```c
void pipeline::run()
{
    if(rank() == 0) send<int>(1, mStages.run<0>());
    if(rank() == 1) send<int>(2, mStages.run<1>(recv<int>(0)));
    if(rank() == 2) mStage.run<2>(recv<int>(1));
}
```

A final inlining step replaces the run method call by the actual task code and resolves the send and recv calls into their corresponding MPI equivalent for atomic types or into a specialized version for user-defined type.

4.3. Task mapping

Once the skeleton has been generated, the top-level run() call starts a method that takes care of mapping each task to a processor (cluster node for example). This requires some kind of process topology to be created. This is a classical problem, for which two kinds of solutions have been proposed:

- **A MPI Group based approach** [12]: A subset of nodes is assigned to each skeleton by using a skeleton dependent heuristic. These subsets are then turned into MPI group. At runtime, each skeleton use the MPI communicator associated with its group to perform communications. Communications between skeletons are performed within special group gathering root nodes of skeleton subsets.
- **A linear approach** [15,16]: Each task is defined by a node ID which is the numerical identifier of the node on which the task will be run. Along with this ID, each task keeps the IDs of the tasks to which it will send
results and from which it will receive input data. When a task is instantiated, IDs are computed and the current node ID is incremented. When skeletons are nested, the input and output node ID of the skeleton is set accordingly to the skeleton communication policy. All communications occur in the `MPI_COMM_WORLD` communicator.

Both approaches lead to acceptable results. Our choice was to use the latter because it fits well with our task definition policy. As we create tasks in a linear way, the actual implementation of this topology deduction algorithm is straightforward. Each time a task constructor is called, the node ID of the task is computed. It also allows us to be able to statically determine the maximum number of nodes used by the application. Fig. 7 shows the task mapping for our current example.

### 4.4. Skeleton optimizations

Optimizing skeletons is needed to increase performances of skeleton-based applications. To do so, the QUAFF code generator runs through skeleton definitions to search for specific patterns.

For example, using a recursive algorithm, QUAFF can rewrite a list of user-defined skeletons stages in case of nesting. This is illustrated in listing 12 which shows a set of meta-programs performing this kind of optimization by merging typelists. These meta-programs read like production rules. We define a general case, the specialized rules for nested pipeline and the terminal case that ends the programs. The basic layout is that for any given typelist, we append the head with an optimized tail. When the meta-program encounters a pipeline, it uses the specialized template that optimizes the pipeline typelist, the remaining of the initial typelist and concatenates them. When the meta-program encounters a `null_t`, it terminates.

**Listing 12: Pipeline optimizer meta-programs**

```cpp
1 template<class H, class T> struct optim< typelist<H,T> >
2 {
3     typedef typelist<H, optim<T>::type_t> type_t;
4 };
5
6 template<class P, class R>
7 struct optim< typelist<pipeline<P>,R> >
8 {
9     typedef optim<P>::type_t head_t;
10    typedef optim<R>::type_t tail_t;
11    typedef append< head_t, tail_t >::type_t type_t;
12 };
13
14 template<> struct optim<null_t>
15 {
16    typedef null_t type_t;
17 };
```

![Fig. 7. Task/Node mapping for a three stage pipeline.](image)
5. Related work

As stated in Section 1, QUAFF is a library-based approach to skeleton-based parallel programming. The most significant projects related to this approach are BSMLlib, Lithium, eSkel and Muesli.

BSMLlib [17] is a library for integrating Bulk Synchronous Parallel (BSP) programming in a functional language (Objective Caml). It extends the underlying lambda-calculus with parallel operations on parallel data structures. Being based on a formal operational semantics, it can be used to predict execution costs.

Lithium [9,2] is a Java library in which parallel programs are written using a set of predefined skeleton classes to be run on a cluster or grid-like network 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 languages.

The ESkel Library [6] 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 to the C/MPI programmer, allowing a large variation in the kind of algorithms that can be developed [7]. ESkel proposes a limited, yet useful set of skeletons. Building applications is easy and integrating ad hoc parallelism is often an appreciable shortcut to circumvent skeleton limitations. However, eSkel exposes a rather low-level API, with a lot of MPI-specific implementation details.

MUESLI, the Münster Skeleton Library [15] 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 [4] and some data parallel skeleton. 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. Kuchen’s approach of a 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 set of skeleton proposed is large enough to accommodate various parallel algorithms. The main problem with Muesli, however, is that the overhead paid for such a high-level API is rather high (in [15] it is reported to be between 20% and 110% for simple applications).

6. Conclusion

In this paper, we have introduced QUAFF, a skeleton-based parallel programming library written in C++. Compared to similar works, QUAFF drastically reduces the runtime overhead by performing most of skeleton expansion and optimization at compile time. This is carried out by relying on template-based meta-programming techniques and without compromising expressivity nor readability. This has been demonstrated on several realistic, complex vision applications, including a real-time 3D reconstruction application and, more recently, a real-time pedestrian tracker based on particle filtering [10].

For these reasons, we claim that QUAFF meets the four guidelines proposed by Cole in [6] to assess skeleton-based parallel programming libraries. First, by relying on a plain C++ interface, QUAFF contributes to the propagation of the skeleton concept with minimal disruption. This is especially true in the computer vision domain – one of our target application domain –, where developers are traditionally reluctant to use another programming language and where reuse of existing code and libraries is mandatory. Second, QUAFF offers several ways to integrate ad hoc parallelism, either by using the par do skeleton or by inserting direct calls to MPI inside user-defined tasks. Third, QUAFF shows the payback of using a skeleton-based approach. In our case, the gain in development effort time has been at least of one order of magnitude (an application such as the one described in Section 3 or in [10] can be implemented in less than 1 day, whereas a hand-crafted MPI implementation required more than a week of development and debugging). Last, and although this has been only suggested in this paper, QUAFF offers a way to accommodate diversity, by passing additional parameters to the existing skeletons (such as the dispatcher/collector option for the farm skeleton).

Ongoing work focuses on two directions. At the application level, we are experimenting with other applications from several domains in order to refine the skeleton set and options. At the implementation level, we are working to further improve the inter-skeleton optimization rules sketched in Section 4.4. A public release of the QUAFF software is planned in the next few months.
References


