Modeling algorithmic skeletons for automatic parallelization using template metaprogramming

Alexis Pereda, David R.C. Hill, Claude Mazel, Bruno Bachelet
Université Clermont Auvergne, CNRS, LIMOS
Clermont-Ferrand, France
{alexis.pereda, david.hill, claude.mazel, bruno.bachelet}@uca.fr

Abstract

This article presents a framework for algorithmic skeletons that aims at representing a whole algorithm, both its sequential and possibly parallelizable parts, in order to enable making global decisions about its implementation. With our modeling, a skeleton is described by an algorithmic structure and a data flow graph, built from the composition of bones and other skeletons. We introduce this notion of bones which represents elementary sequential or parallel patterns whose implementation is available (from the library or designed by well-aware developers), whereas skeletons are automatically implemented from their description. The proposed design, implemented with Template Metaprogramming (TMP), able to operate both at compile- and run-time, allows implementing new bones, describing new skeletons, or simply the instantiation of a skeleton by providing muscles in the form of sequential functions.

Once a skeleton is instantiated, one can decide to generate either a sequential or a parallel code of the algorithm. To optimize the parallelization process, we propose orchestrators, in the form of C++ templates that can analyze a skeleton at compile-time and tune its execution.

A C++ library-based solution is presented, and its mechanisms and usage are illustrated by implementing a GRASPxELS algorithm, a common OR metaheuristic, that enables two levels of parallelism. Performance results are shown to assert that this approach presents negligible run-time overhead.

Keywords – algorithmic skeletons; parallelization; template metaprogramming; task orchestration

1 Introduction

Developing parallel software has become an important issue in order to take advantage of currently available multiple cores hardware. It led to the design of many tools helping developers to design their code using multiple threads with varying levels of abstraction, from low level interfaces (e.g. POSIX Threads) to high level ones, that relieve the developer from directly manipulating threads and rather using parallelization patterns (e.g. OpenMP, Intel TBB). However, these tools often still require the developer to know how parallelization works and source code is interspersed with parallelization specific instructions. Skeleton-based approaches [4] provide parallelization patterns as skeletons with blank parts which the developer can fill using functions, the muscles, usually written unknowingly of the possibility of being inserted into a skeleton. This approach allows a clear separation of the end developer code from the parallelization system.

Our work aims at providing an automatic parallelization utility with the following requirements: being not intrusive in the domain code; not requiring advanced knowledge about parallelization; possibly integration into an existing project; without significant run-time overhead. To meet this list of requirements, we have selected the widely used C++ language with its ability to achieve Template Metaprogramming (TMP) [1]. The design we propose in this paper represents atomic parallel or sequential patterns as "bones" that can be used together to build the algorithm structure. We then pair it with a data flow graph, the links, to build a full skeleton. By setting muscles to a skeleton, one can create a body. Bodies can be used as components of skeletons, but ultimately are to produce an implementation of the described algorithm using our library. Eventually, a task orchestrator can be set to control how the execution is done. Using our library, one can operate at the lowest level and provide new parallelization patterns. One can also write generic skeletons that end developers will use to describe their algorithm to get a parallel program from it. One can eventually write decision algorithms to control how the parallelization is done.

We apply our algorithmic skeleton library to Operational Research (OR), specifically to a commonly used metaheuristic, GRASPxELS [10], in order to solve Traveling Salesman Problem (TSP) instances. This metaheuristic requires various kinds of bones and offers two parallelizable levels, making it an interesting application to our modeling.

Section 2 presents related work. In section 3, we detail the use case that serves to support our study. Then we explain how our modeling fits specifically with our use case,

and more generally with any case, by presenting first how we construct skeletons from bones, and how we define muscles. Eventually, we show our data flow representation that enables flexibility and type safety. Section 4 explains the instantiation process of skeletons into bodies and the code generation process from bodies to the desired algorithm. This section describes the various ways to set muscles and the skeleton nesting mechanism. Then, section 5 discusses about task orchestration for which two strategies are presented. Section 6 shows performance results of our proposed modeling and the associated library.

2 Related work

There are many low level utilities to write parallel code using multithreading. For this execution model, various libraries implementations are available, including the mainstream POSIX Thread, C++ Standard Library Thread, and Boost.Thread. These tools expect that users create and manage the threads themselves. Whereas that allows a high level of flexibility, advanced knowledge and carefulness are required. To avoid working at this level of detail, one can use many tools. The most widely used Application Programming Interface (API) in this domain is OpenMP. Provided as a compiler extension, OpenMP uses annotations to declare parallelizable code segments. Threading Building Blocks (TBB) [12], a C++ template library developed by Intel, provides tools to implement parallel patterns and a run-time system to map tasks to physical cores, preventing oversubscription (i.e. the use of too many threads so the performance is worse than with fewer threads). Oversubscription results of the generation of an excessive amount of threads which can occur with OpenMP. These tools are intrusive and require the multithreading utilities to be mixed with the user code. We can also find languages and code converters [8, 2]. A new language implies re-writing all the user code and is therefore also very intrusive. Code converters require an additional step in the compilation and can limit the expressiveness in the target language.

Algorithmic skeletons [4] address the problem of writing parallel code by defining the overall structure of a parallelizable code segment. Multiple usages of this concept can be found: Skandium [9] is an algorithmic skeleton Java library that demonstrates the interest of this technique in multicore parallelization; Quaff [7] and Muesli [5] are examples of C++ libraries making use of C++ templates, and Template Metaprogramming (TMP), to reduce runtime overhead and maintain a good level of abstraction; SkePU 2 [6] is another example of a C++ library that uses templates to implement algorithmic skeletons and generate parallel software, but it uses a pre-compilation step before compiling with any C++11 compliant compiler. To our knowledge, there is no algorithmic skeleton library that can represent an entire algorithm, including any sequential part of it. This is probably because algorithmic skeletons have been created specifically to ease parallel development. However, this limits the possible code coverage of an algorithmic skeleton library. Representing an algorithm in one single skeleton even if it contains non-parallelizable parts is interesting when it comes to making the decision on how the tasks will be orchestrated over the available cores. Algorithmic skeletons generally define implicitly the communication details [9]. This is a choice that reduces flexibility for the end developer, while making the use of the skeleton possibly simpler.

From these observations, we propose a framework for algorithmic skeletons based on 4 layers.

- The first layer defines the atomic parallel or sequential patterns that will be used to compose algorithms, which we call bones.
- The second layer contains both the structure definition of an algorithm, and a representation of the data flow (the links), that paired together make the skeleton.
- The third layer corresponds to the domain code, that is the code that the end developer will inject into a skeleton, the muscles, to define a body.
- The fourth layer is the task orchestration system that interacts with bones to tune how threads are created and which tasks are assigned to these threads.

3 Modeling skeletons using templates

We propose an algorithmic skeleton C++ library, relying on C++ templates and TMP techniques to represent and process algorithmic skeletons during the compilation process. This makes our library portable and allows us to reduce the execution time overhead implied by the abstraction layer we introduce.

In this section, our choices for modeling algorithmic skeletons with our library are explained. We present here how to design a GRASPxELS algorithm, a common Operational Research (OR) metaheuristic, that will be our use case throughout this document. This algorithm is a combination of two well-known metaheuristics: a Greedy Random Adaptive Search Procedure (GRASP) whose local search is an Evolutionary Local Search (ELS) [10]. This algorithm is interesting because despite of its simplicity, it raises challenges both in the design of skeletons with TMP, and in automating parallelization.

The goal of GRASP (cf. algorithm 1) is to find the best solution S^* for an optimization problem P. Its structure is composed of a loop, repeating a sequence of two tasks: a constructive heuristic that randomly builds a solution, followed by a local search that improves this solution. After the loop, the best solution S^* is selected amongst the solutions S_i found at each iteration. Because the iterations do not depend on each other, they all could be run in parallel.

Algorithm 1 GRASP

```
function GRASP(P)
    for i = 1..N do
        S_i ← constructiveHeuristic(P)
        S_i ← localSearch(P, S_i)
    end for
    S* ← select({S_1, S_2, ..., S_N})
    return S*
end function
```

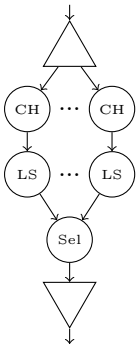

Figure 1: GRASP skeleton

Algorithm 2 ELS

```
function ELS(P, S)
    for i = 1..N do
        for j = 1..M do
            S_j ← mutate(S)
            S_j ← localSearch(P, S_j)
        end for
        S* ← select1({S_1, S_2, ..., S_M})
        S ← select2(S, S*)
    end for
    return S
end function
```

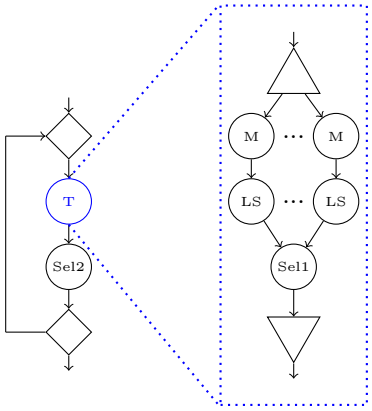

Figure 2: ELS skeleton

3.1 Skeletons, Muscles, and Bones

An algorithmic **skeleton** is a parameterized algorithm where the overall structure is known, but some parts are willingly let unknown (i.e. the parameters of the skeleton, called here the **muscles**). The GRASP skeleton (cf. figure 1) can be modeled as a farm [3], a well-known algorithmic structure, where many tasks are independently performed, here the constructive heuristic (`CH`) followed by the local search (`LS`); and afterwards, the result of one of them is selected, here the best solution (`Sel`).

The **muscles** are the algorithmic elements the developer must provide to a skeleton in order to fill the blanks left (in figure 1, each circle corresponds to a slot for a muscle). These elements can be concrete sequential code (considered as atomic) or skeletons. The GRASPxELS algorithm is built from the GRASP skeleton where the `LS` muscle is implemented by the ELS skeleton (cf. figure 2), which has its own muscles.

The goal of the ELS algorithm (cf. algorithm 2 and figure 2) is to improve a given solution S by generating a set of mutated solutions S_j (muscle `M`), that are improved through a local search procedure (muscle `LS`). The best solution S^* of all S_j is selected (muscle `Sel1`) and becomes the reference solution S if better than the previous S (muscle `Sel2`). This procedure is repeated several times. Notice that the outer loop cannot be parallelized, as each iteration depends on the previous one (data dependency), whereas the inner loop can be. The overall GRASPxELS offers thus the possibility of two levels of parallelization.

In our approach, a skeleton is defined by assembling elementary patterns, which we call **bones**, with muscles and skeletons. For instance, to define the GRASP skeleton, the bone `Farm` is combined with two muscles in sequence, `CH` and `LS`. Contrary to muscles which implementation is sequential when provided by the end developer, and contrary to skeletons which implementation is automatically generated by our library, bones have predefined various implementations (e.g. a sequential code, a multithread code...) usually provided with the library (notice that well-aware users can define bones and their implementations).

3.2 Structure and Links

Most libraries implicitly define how the data transfer is done. For instance, in a bone with two muscles executed in sequence, one can force the return value of the first muscle to be the first argument of the second muscle. If it relieves the developer from deciding for each muscle how it will interact with others, it is also a constraint when it comes to writing muscles because one must comply with the choices done when the library was implemented. We chose to make this explicit and flexible. For that reason, we split the description of a skeleton in two parts: the **structure** that corresponds strictly to the structure of the algorithm, and the **links** that describe the data flow inside the structure.

Listing 1 shows how to define the structure of the GRASP

skeleton with our library. `CH`, `LS` and `Sel` correspond to the muscles in figure 1. The first line makes these muscles unknown at this point. We use two bones, `Serial` which simply executes tasks in sequence, and `FarmSel` which is an association of a classic farm skeleton followed by a selection. Our implementation of `FarmSel` in its selection part targets scalability with high number of threads (i.e. high number of generated outputs to filter), hence it does not strictly correspond to figure 1 because the selection function will not be called with all the generated outputs but only with a pair to compare.

```
1 template<typename CH, typename LS, typename Sel>
2 using SkGraspStructure =
3 S<FarmSel,
4   S<Serial, CH, LS>,
5   Sel
6 >;
```

Listing 1: GRASP skeleton structure definition

Listing 2 shows how to define the links for the GRASP skeleton. Muscles and bones can be considered as callables (i.e. functions, functors or lambdas in C++), meaning that the nature of the parameters they accept and what they return can be described using a function signature. Line 8 states that muscle `Sel` gets two solutions and returns one. In this particular case, it is the `FarmSel` bone that sets which arguments are given to the muscle (here, the arguments are S^* and one of the S_i, for each iteration).

```
1 template<typename Problem, typename Solution>
2 using SkGraspLinks =
3 L<FarmSel, Solution(Problem const&),
4   L<Serial, R<1>(P<0>),
5     Solution(P<0>),
6     Solution(P<0>, R<0>)
7   >,
8   Solution(Solution const&, Solution const&)
9 >;
```

Listing 2: GRASP skeleton links definition

To transfer values from one muscle or bone to another, placeholders are used: `P<i>` means the parameter of index `i` of the upper bone, `R<i>` means the return of the muscle of index `i` in the current bone. For instance, `P<0>` at line 5 means that the argument of muscle `CH` is the first parameter of `Serial` bone, which is also a placeholder for the first parameter of `FarmSel` bone that is the problem P of algorithm 1 to optimize. `R<0>` at line 6 means that the second argument of `LS` is the return value of muscle `CH` (at line 5), which is the solution S_i of the constructive heuristic of algorithm 1.

Each muscle is described as a single function signature. The structural elements, bones `Serial` and `FarmSel`, must keep their organization as defined in the structure of the skeleton. However, they also have a function signature, with the same placeholder mechanism. For instance, `R<1>` at line 4 means that the return value of `Serial` bone will be the return value of muscle `LS`, which is the solution S_i of the local search of algorithm 1.

One last thing to notice is the function signature at line 3 that refers to the whole GRASP skeleton. For that reason, using placeholders for parameters would not make sense. At the opposite, a placeholder for the returned value is possible. But the return value is hard linked in the `FarmSel` bone, thus only the return type can be expressed here.

Once the structure and the links of a skeleton are defined, it is possible to produce a whole parameterized description of a skeleton as in listing 3. The template `SkGrasp` is built from the two templates defining the structure and the links and keeps the muscles unknown, along with the data types. This step makes it easier to define skeletons because it allows describing the structure regardless of how data are transferred.

```
1 template<
2   typename Problem, typename Solution,
3   typename CH, typename LS, typename Sel
4 >
5 using SkGrasp =
6 BuildSkeleton<SkGraspStructure, SkGraspLinks>
7 ::skeleton<
8   Pack<CH, LS, Sel>,
9   Pack<Problem, Solution>
10 >;
```

Listing 3: GRASP skeleton building

4 Implementing skeletons

The primary objective after defining a skeleton is to be able to get a functional program that, given specific muscles, runs the desired algorithm. In this section, we will explain our process for implementing a skeleton with given muscles. In order to continue with the example presented in section 3, we assume that the `SkEls` skeleton has been defined to describe the ELS algorithm (cf. algorithm 2 and figure 2).

4.1 Skeleton Body

The first step toward implementing a skeleton is its instantiation. That means that the muscles, which were unknown, are now set, as well as the data types (cf. listing 4). We call **body** the full instantiation of a skeleton. For instance, the following code instantiates the GRASPxELS algorithm with our library, from the `SkGrasp` skeleton.

```
1 using GRASPxELS = SkGrasp<
2   TspProblem, TspSolution,
3   RGreedy, ELS, FN(selectMin)
4 >;
```

Listing 4: `GRASPxELS` body definition

Instantiating a skeleton simply is instantiating the corresponding template. Here, we are creating an instance of `SkGrasp` to solve a Traveling Salesman Problem (TSP). Line 2 defines the types, respectively of the problem P and the solution S of the GRASP algorithm (cf. algorithm 1). Line 3 sets the muscles, which can be implemented in

multiple ways, as long as they are callable: a functor, a function or a skeleton body. The `RGreedy` muscle, which implements a random greedy constructive heuristic (`CH` in figure 1), is a functor, a common alternative way in C++ to function pointers that presents many advantages[1]. Despite its advantages, writing a functor can be cumbersome, so our library offers the possibility of setting muscles in the form of functions. For instance, the `selectMin` muscle of line 3 can be defined as in listing 5.

```
1 TspSolution selectMin(TspSolution const&lhs,
2                       TspSolution const&rhs) {
3   return lhs.value() < rhs.value()? lhs:rhs;
4 }
```

Listing 5: `selectMin` muscle definition

Internally, our library only manages functors, so muscles in the form of functions have to be transformed into functors, which is done using the `FN` macro[2] (cf. line 3 of listing 4)

The last way to set a muscle is by providing the body of a skeleton, which is not a callable yet, but will be transformed into one in a further step. The template argument `ELS` in our example is actually a body of the `SkEls` skeleton.

Ultimately, the type `GRASPxELS` represents the body of the GRASPxELS algorithm. Until now, all about our skeletons was purely static (i.e. known at compile-time), but there are properties of the skeleton that should be defined dynamically, notably some parameters of the bones and muscles that make up the body. To fill up this information, we need a dynamic object, an instance of the body.

```
1 ELS bodyEls;
2 // parameterization of bodyEls
3
4 GRASPxELS bodyGraspEls;
5 bodyGraspEls.n = 10;
6 bodyGraspEls.task.task<1>() = bodyEls;
```

Listing 6: `GRASPxELS` body instantiation

At line 4 of listing 6, the `GRASPxELS` body is instantiated, and the next lines initialize some of its parameters. Line 5 initializes the parameter N of algorithm 1, which is the parameter `n` of the top `FarmSel` bone of the GRASP skeleton. Line 6 initializes the `LS` muscle of the GRASP skeleton with an instance of the ELS body (cf. line 1), which is the second task (`task<1>`) of the `Serial` bone that is itself the only task (`task`) of the top `FarmSel` bone of the GRASP skeleton.

4.2 Body Implementation

The next step is to obtain an implementation for a body, meaning generating an effective code, depending on the execution context required by the end user. Our library currently deals with two contexts: sequential or parallel (multithread). As explained previously, bones have implementations for each context, and muscles must be provided as sequential code. Skeletons will be analyzed by our library to generate the appropriate code to link muscles and bones.

To get the implementation of a body with our library, template function `implOf` is called with a tag, a static value setting the execution context, and providing the body to implement. Listing 7 shows how to generate a sequential code for the body `bodyGraspEls`.

```
1 auto graspEls = implOf<Sequential>(bodyGraspEls);
```

Listing 7: GRASPxELS sequential implementation

If a parallel implementation is wanted instead, changing the tag from `Sequential` to `Parallel` is sufficient (listing 8).

```
1 auto graspEls = implOf<Parallel>(bodyGraspEls);
```

Listing 8: GRASPxELS parallel implementation

The body being an object, its type can be parsed at compile-time (like expression templates [11]) to analyze the structure of the whole algorithm (which is performed by the *orchestrator* presented in the next section), and its parameters can be tuned at run-time, before (by the `implOf` function) and during (by the orchestrator) the execution of the algorithm.

The ultimate object `graspEls` is a callable whose signature is the one specified for the top-level bone of the GRASP skeleton. One can call it as if it was a function as in listing 9.

```
1 TspProblem problem = /* problem initialization */;
2 TspSolution solution = graspEls(problem);
```

Listing 9: GRASPxELS execution

5 Task orchestration

There is still one important aspect to consider when making a parallel implementation of a body: how the tasks running in parallel should be orchestrated? Here we will study how tasks are assigned to threads on a multicore parallelization context. This section will discuss the relevance of having a facility that makes task orchestration when two levels of parallelization are possible, and present the way our library currently enables it.

As explained previously, the GRASPxELS algorithm presents two loops that can possibly be parallelized: the main loop of GRASP and the inner loop of ELS. More generally, let us consider two intertwined parallelizable loops, the outer loop having N iterations and the inner

[1] `https://isocpp.org/wiki/faq/pointers-to-members#functionoids`

[2] `FN` macro is a syntactic helper due to C++14 compatibility concerns

loop running M tasks, each task taking 1 unit of time to run sequentially. We assume here that there are no CPU, memory or I/O bounds for parallelization. Executed sequentially, the two loops take $M \times N$ units of time. We need to decide here how many threads to assign to each loop, and how to distribute the tasks on these threads, knowing the number K of available cores in a multicore architecture.

5.1 One-Level Orchestration

Let us consider that $K = 4$ cores are available, and that $N = 6$ and $M = 2$. One possible orchestration is to have $t_1 = 4$ threads allocated for the outer loop, with 2 threads running 2 iterations and 2 threads running 1 iteration, as in figure 3. The overall execution time will theoretically be $2 \times 2 = 4$ units. If considering only parallelizing the first level, the theoretical optimal execution time should arise when $K \geq 6$ by allocating $t_1 = 6$ threads.

More generally, for one-level orchestration, $t_1 = \min\{K, N\}$ threads should be allocated. As $\frac{N}{t_1}$ is not always an integer, r_1 threads should run $n_1 + 1$ iterations and $s_1 = t_1 - r_1$ threads should run n_1 iterations, with $n_1 = \lfloor \frac{N}{t_1} \rfloor$ and $r_1 = N - n_1 t_1$. Hence, the overall execution time should be $n_1 M$ if $r_1 = 0$, $(n_1 + 1)M$ otherwise.

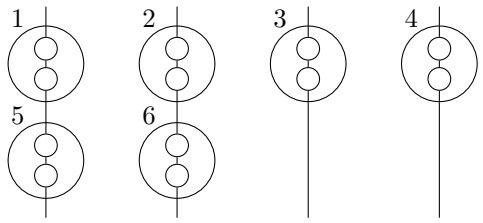

Figure 3: One-level task orchestration

5.2 Two-Level Orchestration

Notice in the previous example (figure 3) that iterations 5 and 6 are not optimally executed. They are allocated on two threads, whereas two cores remain available. The inner loop of each iteration could be executed in parallel, meaning two threads could be assigned to each iteration (cf. figure 4). This way, iterations 1 to 4 should run in 2 units of time, and the iterations 5 and 6 in 1 unit of time, decreasing the theoretical overall execution time to 3 units.

More generally, the outer loop should be split in two loops: loop A, with t_1 allocated threads running n_1 iterations each, and the loop B, with r_1 allocated threads running 1 iteration each. For each loop, A or B, the number K_2 of cores available for the inner loop should be determined: $K_2^A = \lfloor \frac{K}{t_1} \rfloor$ for the first loop and $K_2^B = \lfloor \frac{K}{r_1} \rfloor$ for the second loop. Once K_2 is known, the orchestration of the inner loop is achieved in the same way as the outer loop.

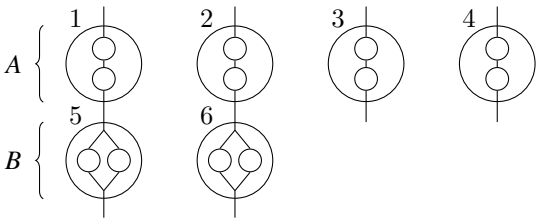

Figure 4: Two-level task orchestration

Table 1 shows how the two-level orchestration operates for $N = 6$ and $M = 5$, with the number of available cores K varying from 1 to 20.

Table 1: Two-level orchestration

	Outer Loop			Inner Loop A				Inner Loop B			
K	t_1	n_1	r_1	K_2	t_2	n_2	r_2	K_2	t_2	n_2	r_2
1	1	6	0	1	1	5	0	-	-	-	-
2	2	3	0	1	1	5	0	-	-	-	-
3	3	2	0	1	1	5	0	-	-	-	-
4	4	1	2	1	1	5	0	2	2	2	1
5	5	1	1	1	1	5	0	5	5	1	0
6	6	1	0	1	1	5	0	-	-	-	-
7	6	1	0	1	1	5	0	-	-	-	-
8	6	1	0	1	1	5	0	-	-	-	-
9	6	1	0	1	1	5	0	-	-	-	-
10	6	1	0	1	1	5	0	-	-	-	-
11	6	1	0	1	1	5	0	-	-	-	-
12	6	1	0	2	2	2	1	-	-	-	-
13	6	1	0	2	2	2	1	-	-	-	-
14	6	1	0	2	2	2	1	-	-	-	-
15	6	1	0	2	2	2	1	-	-	-	-
16	6	1	0	2	2	2	1	-	-	-	-
17	6	1	0	2	2	2	1	-	-	-	-
18	6	1	0	3	3	1	2	-	-	-	-
19	6	1	0	3	3	1	2	-	-	-	-
20	6	1	0	3	3	1	2	-	-	-	-

Figure 5 presents the theoretical speedup of this orchestration on physical cores. It is compared with the one-level orchestration.

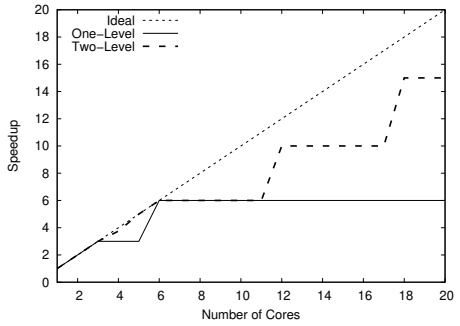

Figure 5: Theoretical speedup of two-level orchestration

5.3 Orchestrator

Our library enables using a task orchestrator that can operate both at compile-time and run-time. During the compilation, it has access to the body type (the same way expression templates proceed [11]) and can therefore

use it to analyze the whole algorithm and make decisions even before the program is being executed. In the case of GRASPxELS, it can detect the potential two levels of parallelization before execution, which allows making better decisions than considering the first level only. The body type is given to the class template implementing an orchestrator as a type template argument to instantiate it. The class generated by this template can aggregate member variables that will hold run-time properties, possibly set by the end developer. An information that any orchestrator will likely have is a limit of concurrent threads, typically bound to the number of available cores, but which can be overwritten. As an example, to implement a two-level orchestration, one will use the run-time information N to tune the parameters (cf. table 1) of the outer and inner loops.

For instance, the `OneLevel` orchestrator implements the behavior we described in section 5.1, that is running in parallel only the first parallelizable level. Listing 10 implements a GRASPxELS using this orchestrator.

```
1 auto orch = makeOrchestrator<OneLevel>(bodyGraspEls);
2 // orchestrator parameterization
3
4 auto graspEls = implOf<Parallel>(bodyGraspEls, orch);
```

Listing 10: GRASPxELS one-level orchestration

The class template representing the orchestrator must be given to the `makeOrchestrator` function, along with the body instance to analyze and tune. The resulting object, `orch`, is an orchestrator instance that will be given to the `implOf` function to be associated with the body instance. This way, when necessary, a bone can send requests to the orchestrator.

Changing the task orchestrator to use one that implements the behavior described in section 5.2 simply consists in using its class template instead of the other one, see listing 11.

```
1 auto orch = makeOrchestrator<TwoLevel>(bodyGraspEls);
2 auto graspEls = implOf<Parallel>(bodyGraspEls, orch);
```

Listing 11: GRASPxELS two-level orchestration

6 Performance results

We aimed at a library implementation of algorithmic skeletons using TMP techniques to minimize the run-time overhead caused by the abstraction layer our tool provides. To validate that our library is not causing a significant overhead, performance measures were run on a handwritten implementation of GRASPxELS applied to TSP instances. The same measures were performed with a GRASPxELS automatically generated from its skeleton and muscles, using two distinct task orchestrators. All tests have been performed on an Intel Xeon CPU E5-2670 v2 at 2.50 GHz, with physical 20 cores, using g++ 8.2.0 with the optimization `O2` flag activated. All figures result of means of 20 runs, where seeds for the random number generation were controlled to ensure repeatability (i.e. that every run, independently of the number of threads allocated, performs the same amount of operations and provide the same result).

Figure 6 shows the comparison between handwritten sequential GRASPxELS and the automatically generated one for a varying number of iterations, from 5 to 30, for the GRASP loop. No significant overhead is noticeable.

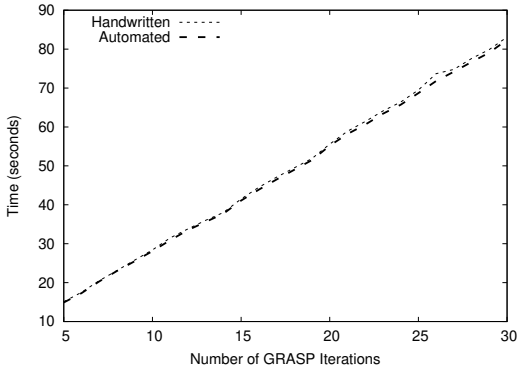

Figure 6: Skeleton overhead for sequential execution

Figure 7 shows the comparison between handwritten parallel GRASPxELS and the automatically generated one using the `TwoLevel` orchestration strategy for a varying number of allotted cores. Again, there is no significant overhead.

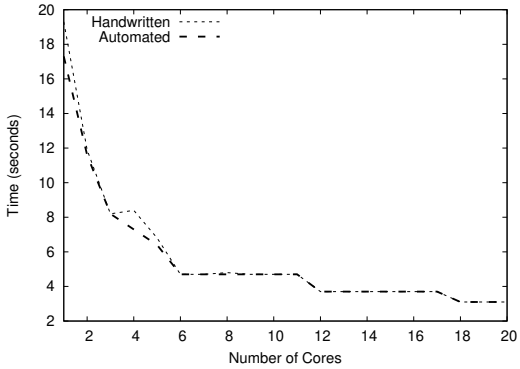

Figure 7: Skeleton overhead for parallel execution

Figure 8 shows the comparison between one-level and two-level orchestration for 2 to 20 cores, as presented in section 5. We can notice improvement when considering two levels of parallelization.

The two-level orchestration assumes that all the iterations of a loop take almost the same amount of time to execute, which should explain that we observe a speedup below the theoretical one. As an orchestrator can also operate at run-time, it is possible to implement a run-time load balancing (e.g. with a thread pool).

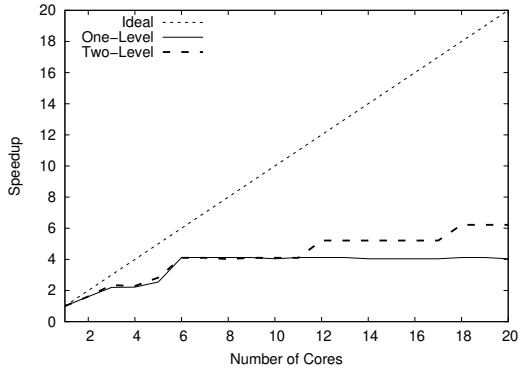

Figure 8: Impact of orchestration

Probably, other two-level orchestrations should be studied, notably one that uses both information N and M to make a decision at first level, contrary to our solution that only uses N. Figure 9 shows nevertheless the relative improvement of the two-level orchestration over the one-level.

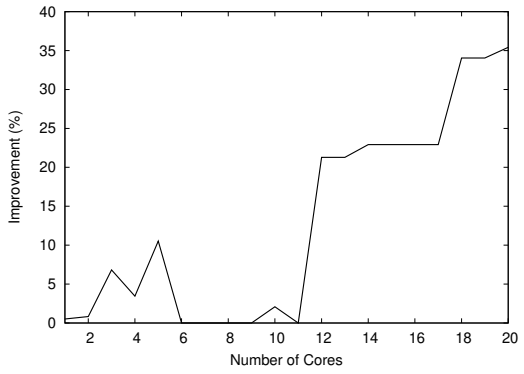

Figure 9: One-level vs two-level orchestrations

7 Conclusion

This paper presents our implementation of algorithmic skeletons which enables a developer to write sequential code that will serve as parts of an automatically parallelized program. After introducing the GRASPxELS metaheuristic, our Operational Research (OR) use case, we presented our modeling. We defined bones: implementations of atomic parallel or sequential patterns combined to describe an algorithm structure. We explained the construction of an algorithmic structure and the definition of its links (the data flow graph) to make a skeleton. We showed how bodies, skeletons instances to which muscles (missing code provided by the end developer) are set, can be used to automatically generate either a sequential or a parallel implementation of the algorithm they represent.

We used C++ templates and Template Metaprogramming (TMP) techniques to implement skeletons so we achieved to minimize the execution time overhead that is generally brought by abstraction layers. Moreover, by strictly using only standard C++, we do not require any step before compilation, making our utility portable and usable with any C++14 compliant compiler. We implemented the GRASPxELS metaheuristic using our library and we produced both a sequential and a parallel implementation, along with handwritten corresponding implementations, so that we could compare the performances. We showed that no significant execution time overhead was measured.

Designing skeletons with TMP allows us to operate on algorithmic structures both at compile-time and at run-time. Our library enables designing orchestrators that are able to analyze the structure of a skeleton during compilation, like expressions templates do, and tune parameters of the final code during execution. We illustrate this possibility with the task orchestration of GRASPxELS that allows two levels of parallelization.

While the library is operational, we still have to extend the set of bones available so any common algorithm can be implemented without requiring the developer to adapt an existing bone or to fully create a new one. Creating new bones will still be possible, for instance to address very specific algorithm particularities in an effective way. Designing more efficient orchestrators, adapted to more general situations than a two-level parallelization, must also be investigated. Another future work will be to introduce tools to facilitate the numerical reproducibility when using random numbers in skeletons. Some steps to reproducibility have already been achieved for this paper in order to validate the performance measures.

References

[1] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. 5. printing. The C++ In-Depth Series. Boston, Mass: Addison-Wesley, 2004. ISBN: 978-0-321-22725-6 (cit. on p. 1).

[2] Amit Barve, Sneha Khomane, Bhagyashree Kulkarni, Shubhangi Katare, and Sonali Ghadage. "A Serial to Parallel C++ Code Converter for Multi-Core Machines". In: *2016 International Conference on ICT in Business Industry Government (ICTBIG)*. 2016, pp. 1–5. ISBN: 978-1-5090-5515-9. DOI: 10.1109/ICTBIG.2016.7892700 (cit. on p. 2).

[3] Duncan K. G. Campbell. "Towards the Classification of Algorithmic Skeletons". In: (1996) (cit. on p. 3).

[4] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Cambridge, MA, USA: MIT Press, 1989. ISBN: 978-0-262-53086-6 (cit. on pp. 1, 2).

[5] Steffen Ernsting and Herbert Kuchen. "Algorithmic Skeletons for Multi-Core, Multi-GPU Systems and Clusters". In: *International Journal of High Performance Computing and Networking* 7.2 (2012), p. 129.

ISSN: 1740-0562, 1740-0570. DOI: 10.1504/IJHPCN.2012.046370 (cit. on p. 2).

[6] August Ernstsson, Lu Li, and Christoph Kessler. "SkePU 2: Flexible and Type-Safe Skeleton Programming for Heterogeneous Parallel Systems". In: *International Journal of Parallel Programming* 46.1 (Feb. 2018), pp. 62–80. ISSN: 0885-7458, 1573-7640. DOI: 10.1007/s10766-017-0490-5 (cit. on p. 2).

[7] J. Falcou, J. Sérot, T. Chateau, and J. T. Lapresté. "Quaff: Efficient C++ Design for Parallel Skeletons". In: *Parallel Computing*. Algorithmic Skeletons 32.7 (Sept. 2006), pp. 604–615. ISSN: 0167-8191. DOI: 10.1016/j.parco.2006.06.001 (cit. on p. 2).

[8] Laxmikant V. Kale and Sanjeev Krishnan. "CHARM++: A Portable Concurrent Object Oriented System Based on C++". In: *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*. ACM Press, 1993, pp. 91–108. ISBN: 978-0-89791-587-8. DOI: 10.1145/165854.165874 (cit. on p. 2).

[9] Mario Leyton and José M. Piquer. "Skandium: Multi-Core Programming with Algorithmic Skeletons". In: *2010 18th Euromicro Conference on Parallel, Distributed and Network-Based Processing*. Pisa: IEEE, Feb. 2010, pp. 289–296. ISBN: 978-1-4244-5673-4. DOI: 10.1109/PDP.2010.26 (cit. on p. 2).

[10] Christian Prins. "A GRASP x Evolutionary Local Search Hybrid for the Vehicle Routing Problem". In: *Bio-Inspired Algorithms for the Vehicle Routing Problem*. Studies in Computational Intelligence. Springer, Berlin, Heidelberg, 2009, pp. 35–53. ISBN: 978-3-540-85152-3. DOI: 10.1007/978-3-540-85152-3_2 (cit. on pp. 1, 2).

[11] Todd Veldhuizen. "Expression Templates". In: *C++ Report* 7 (1995), pp. 26–31 (cit. on pp. 5, 6).

[12] Thomas Willhalm and Nicolae Popovici. "Putting Intel® Threading Building Blocks to Work". In: *Proceedings of the 1st International Workshop on Multicore Software Engineering*. IWMSE '08. New York, NY, USA: ACM, 2008, pp. 3–4. ISBN: 978-1-60558-031-9. DOI: 10.1145/1370082.1370085 (cit. on p. 2).