Parallel algorithmic skeletons for metaheuristics

Alexis Pereda[1], David R.C. Hill[1], Claude Mazel[1], Bruno Bachelet[1]

Université Clermont Auvergne, CNRS, LIMOS, Clermont-Ferrand, France
{alexis.pereda, david.hill, claude.mazel, bruno.bachelet}@uca.fr

Keywords: *algorithmic skeletons, parallelization, metaheuristics, metaprogramming*

1 Introduction

Designing parallel software is a difficult task, but it became essential in modern computing. This is notably true in Operational Research (OR) where many algorithms can benefit greatly from parallelization. It has led to the development of software frameworks that ease the parallel design of OR algorithms (e.g. [1]), based on object-oriented and template programming. New design possibilities arose with the advances of template metaprogramming, especially in the C++ language.

Our goal is to design a library able to produce efficient parallel implementations of an algorithm from the knowledge of its structure with no runtime overhead. We propose to use algorithmic skeletons [2] in order to describe OR algorithms and make them ready for parallelization. For this purpose, metaprogramming techniques are used first to provide facilities to describe an OR algorithm as a composition of algorithmic skeletons, and secondly to analyze and transform the skeleton, at compile-time, into an efficient code to be executed at runtime.

In a first step, we focus on metaheuristics because, as they are generic structures, they naturally adapt to algorithmic skeletons.

2 Modeling algorithmic skeletons

In order to produce a parallel version of an algorithm, we need to get some information about its functioning. As an example, the general scheme of a Greedy Randomized Adaptive Search Procedure (GRASP) for a problem P is shown in algorithm 1. We can represent this algorithm as in figure 1. The circles represent parameterizable parts of the algorithm and the remainder describes the interactions between parts. This kind of modeling is known as an algorithmic skeleton and is commonly used in parallelization [2]. We chose to implement an algorithmic skeleton solution using template metaprogramming in C++ because it enables compile-time code generation, resulting in near zero runtime overhead.

Algorithm 1 GRASP

```
function GRASP(iterMax, P)
    for i = 1..iterMax do
        S ← constructiveHeuristic(P)
        S ← localSearch(S)
    end for
    S* ← best(S)
    return S*
end function
```

GRASP can be split into two main elements: the outer structure, a farm repeating the inner structure and keeping the best result (`Sel`); and a series of a constructive heuristic (`CH`) followed by a local search (`LS`). We have defined several primary skeletons, including `Serial` and `FarmSel`. The first one corresponds directly to a series of tasks. The second one is more complex: it will run its task N times and keep the best result by applying a selector. The task, N and the selector are parameters of the skeleton.

Using these primary skeletons, it is possible to create composite skeletons. The C++ code below is the description of a GRASP using this terminology: it is decomposed in two declarations, the one on the left that describes the structure of the skeleton (i.e. the organization of the tasks), and the other on the right that sets how the different inner tasks interact with each other (i.e. how data are transferred between tasks).

```
template<
  typename CH, typename LS,
  typename Sel
>
using SkelGRASPStructure =
FarmSelStructure<
  SerialStructure<CH, LS>,
  Sel
>;
```

```
template<typename Problem, typename Solution>
using SkelGRASPLinks =
FarmSelLinks<Solution(Problem),
  SerialLinks<R<1>(P<0>),
    Solution(P<0>),
    Solution(R<0>)
  >,
  Solution(Solution, Solution)
>;
```

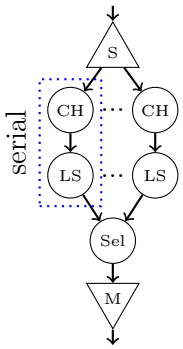

FIG. 1: GRASP skeleton

One interesting aspect of algorithmic skeletons is composability. The example above uses only primary skeletons, but it is possible to use composite skeletons as part of another skeleton, which occurs frequently. For example, as its local search, GRASP could use an Evolutionary Local Search (ELS) which skeleton, shown in figure 2, replaces `LS` in the GRASP skeleton. See the example below:

```
using GRASPxELS = SkelGRASP
< tsp::Problem, tsp::Solution, RGreedy<tsp::Solution>, ELS, SelectMin >;
```

Thanks to the description provided by skeletons, we propose a library able to generate either a sequential or a parallel implementation of the algorithms, tailored to the structure. Our library handles multiple layers of skeletons and multiple levels of parallelism: for example, in the composition GRASPxELS, both the outer structure of GRASP and the inner structure of ELS (framed in figure 2) can be parallelized. This approach allows us to integrate optimization algorithms inside the skeleton-to-code transformation process, for instance to decide on the best way to distribute work among the processing units.

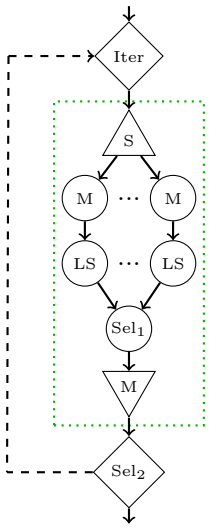

FIG. 2: ELS skeleton

3 Conclusion

We propose a C++ library providing skeletons for common metaheuristics that can be instantiated by giving the missing parts in the form of functions or lambda expressions. The library also allows describing your own skeletons. From the skeleton information, we can generate at compile-time a tailored sequential or parallel implementation. Defining algorithmic skeletons with template metaprogramming makes possible the implementation of compile-time algorithms to optimize the distribution of parallel tasks. All this work is performed by the compiler so it does not imply runtime overhead compared to handwritten implementations. Future work includes developing more primary skeletons and well-known OR skeletons in order to evaluate the library on more complex OR algorithms.

References

[1] S. Cahon, N. Melab, and E.-G. Talbi. “ParadisEO: A Framework for the Reusable Design of Parallel and Distributed Metaheuristics”. In: *Journal of Heuristics* 10.3 (May 2004), pp. 357–380.

[2] J. Darlington et al. “Parallel Programming Using Skeleton Functions”. In: *PARLE ’93 Parallel Architectures and Languages Europe*. Ed. by G. Goos et al. Vol. 694. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 146–160.