

Processing Algorithmic Skeletons at Compile-Time

Alexis Pereda, David R.C. Hill, Claude Mazel, Loïc Yon, Bruno Bachelet

▶ **To cite this version:**

Alexis Pereda, David R.C. Hill, Claude Mazel, Loïc Yon, Bruno Bachelet. Processing Algorithmic Skeletons at Compile-Time. 21ème congrès de la société française de Recherche Opérationnelle et d'Aide à la Décision (ROADEF), Feb 2020, Montpellier, France. hal-02573660v2

HAL Id: hal-02573660
https://hal.science/hal-02573660v2

Submitted on 18 Dec 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Processing algorithmic skeletons at compile-time

Alexis Pereda, David R.C. Hill, Claude Mazel, Loïc Yon, Bruno Bachelet

Université Clermont Auvergne, CNRS, LIMOS, Clermont-Ferrand, France

{alexis.pereda, david.hill, claude.mazel, loic.yon, bruno.bachelet}@uca.fr

Keywords: *algorithmic skeletons, metaheuristics, metaprogramming, compile-time*

1 Introduction

When developing software in Operational Research (OR), one usually aims at getting the shortest execution time. There are multiple means to achieve this, including improving the algorithmic time complexity (finding better suited algorithms and data structures), optimizing the code (e.g. to avoid cache misses), parallelizing the execution... These improvements can be written by the end developer, requiring some coding effort and knowledge, whereas some of them could be achieved automatically.

In [3] we proposed a library to ease the building of parallel implementations of OR algorithms from the knowledge of their structure with no runtime overhead. We chose to use algorithmic skeletons [2] in order to describe metaheuristics and make them ready for parallelization. For this purpose, Template Metaprogramming (TMP) techniques are used first to provide facilities to describe a metaheuristic 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.

C++ TMP allows writing full algorithms to be executed at compile-time, but it is rare for an end developer to use TMP directly, as its syntax greatly differs from the usual language. Hence, we present here some insights of an intermediary library designed to ease the writing of code to process the algorithmic skeletons at compile-time as trees and vectors.

2 Skeletons as trees

To explain our approach, we consider the algorithmic skeleton of the Greedy Randomized Adaptive Search Procedure (GRASP). This algorithm is a parametric structure that can be illustrated as in figure 1 where each circle corresponds to a muscle (i.e. a parameter that is a sequential function to be provided by the end developer) and each dotted box corresponds to a bone (i.e. an elementary predefined sequential or parallel pattern). A farm (`farmsel`) repeats a series (`serial`) of a constructive heuristic (`CH`) followed by a local search (`LS`), and keeps the best result (`Sel`).

Once a skeleton has been described with our skeleton library using TMP [3], we propose to transform it, at compile-time, into a tree (cf. figure 2), as it is a well-known data structure, easier to manipulate than algorithmic skeletons by the end developer. In such a tree, branch nodes are bones (their children being the tasks they execute) and leaves are muscles.

From this structure, it is simple for instance to detect, at compile-time, the number of parallelizable levels of an algorithm, which is necessary to produce an efficient parallel implementation. GRASP has only one parallelizable level (as it only has one `farmsel`, cf. figure 2), whereas GRASPxELS (GRASP with Evolutionary Local Search (ELS) as local search, cf. figure 3) has two levels (as it results in a `farmsel` composed of another `farmsel`).

Our library provides functions to use directly with trees at compile-time, like `TreeAccumulate` that, similarly to the `std::accumulate` function on containers of C++ standard library, goes through all the nodes N of a tree to apply an accumulation function F in order to compute

a value (e.g. the sum of the values of the nodes). Writing such an algorithm is some kind of functional programming based on template recursivity. Here, `TreeAccumulate` calls function F on each node N, from root to leaves, given $R_1, ..., R_n$ the results of `TreeAccumulate` on each child $C_1, ..., C_n$ of N. Such code is hidden from the end developer that will only call the algorithm providing a function for accumulation, which can be compared to Boost.MPL [1].

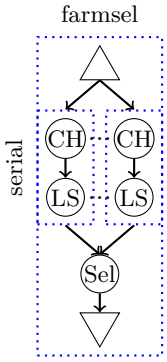


FIG. 1: GRASP skeleton

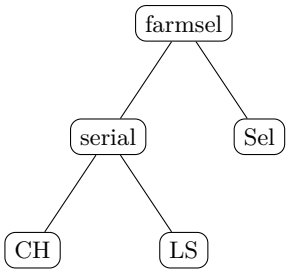


FIG. 2: GRASP skeleton as tree

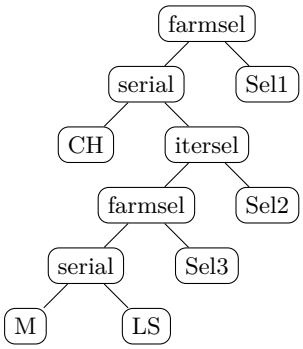


FIG. 3: GRASPxELS skeleton as tree

For instance, counting the parallelizable levels of a skeleton using `TreeAccumulate` requires for the end developer to write an accumulation function $ParLevel(N, R_1, ..., R_n)$ that returns $1 + max(R_1, ..., R_n)$ if the current node is a branch and a `farmsel`, $max(R_1, ..., R_n)$ otherwise.

The tree structure can also be transformed into a vector (storing the nodes in sequence according to a specific order). With this form, it is possible for the end developer to get the number of parallelizable levels of a skeleton by writing a sequence of common accumulate and transform operations on vectors. Both structures (tree and vector) and their algorithms will be presented in TMP code excerpts to show how to write programs executed at compile-time in a form close to the programming language that the end developer is accustomed to.

3 Conclusion

Through our algorithmic skeleton library, we demonstrate the possibility to process information at compile-time using metaprogramming with no runtime overhead on generated program. This is in use, for example, to calculate the number of parallelizable levels in a given algorithm or to apply transformations to a skeleton (e.g. merging serial structures) in order to improve the generated code. This processing is simplified by the usage of an intermediary tool set that allows one to write Template Metaprogramming (TMP) by a means similar to classic run-time algorithms (e.g. transformations and accumulations).

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. 373 pp.

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

[3] Alexis Pereda et al. “Modeling Algorithmic Skeletons for Automatic Parallelization Using Template Metaprogramming”. In: *2019 International Conference on High Performance Computing & Simulation (HPCS)*. Dublin: IEEE, July 2019, pp. 265–272.