Algorithmic skeletons 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

Algorithmic skeletons, introduced by Cole, were designed to ease the development of parallel software. This article presents a way to represent and implement algorithmic skeletons using bones – atomic elements – to build structures, and data flow graphs to link the structures.

We design and implement a library relying on Template Metaprogramming (TMP) to describe and use both skeletons and links to produce automatically either a sequential or a parallel implementation of the algorithm, aiming slight to no run-time overhead compared to handwritten implementations.

Performance results of this library, applied to metaheuristics in Operations Research (OR), are presented to that this approach induces negligible run-time overhead.

Keywords: algorithmic skeletons, parallelization, template metaprogramming.

1 Introduction

Most computers having multiple cores, developing parallel software has become necessary in order to make full use of the available hardware. This has led to the development of numerous tools to ease the implementation of parallel programs, from low level techniques (e.g. specific compilers) to more high level abstractions (e.g. generic libraries).

Our objective is to propose a new parallelization tool that is usable in existing projects (i.e. no new language or specific compiler is needed) and yet not inducing an avoidable run-time overhead when compared to a handwritten implementation. Additionally, we want this tool to provide a clean interface, not requiring to pollute the domain code with parallelization details. This last point oriented our solution towards algorithmic skeletons which enable separation of domain code and parallelization implementation.

The advantage of libraries over most low level techniques is their portability. However, this usually comes with a cost: an abstraction layer producing a less efficient binary than the equivalent handwritten code, which a compiler would be able to produce. Metaprogramming can be seen as an intermediary approach because it makes it possible to produce code without rewriting a full-fledged compiler. Specifically, the C++ language offers Template Metaprogramming (TMP) allowing metaprogramming at library level.

This paper presents our proposal for a new algorithmic skeleton library in C++, using an example application from Operations Research (OR), a Greedy Randomized Adaptive Search Procedure (GRASP) described in algorithm 1. Its goal is to find the best solution S^*, given a problem P. It is done by generating and improving multiple independent solutions, making it suitable for parallelization.

Algorithm 1 GRASP

```
function GRASP($P$)
    for $i = 1..N$ do
        $S_i \leftarrow$ CONSTRUCTIVEHEURISTIC($P$)
        $S_i \leftarrow$ LOCALSEARCH($P, S_i$)
    end for
    $S^* \leftarrow$ SELECT($\{S_1, S_2, ..., S_N\}$)
    return $S^*$
end function
```

2 Algorithmic skeletons

Algorithmic skeletons were designed by Cole [3] to ease the development of parallel software by providing patterns to be used to describe an algorithm so that it will then be automatically parallelized. Numerous tools based on this concept exist, Skandium [6] in Java, working at run-time, Quaff [5] and Muesli [2] in C++, based on TMP, and SkePU 2 [4] which also makes use of TMP but relies on a pre-compilation step. None of these implementations permit the complete representation of an algorithm, including its sequential parts. This effectively reduces the possible code coverage of these algorithmic skeleton tools. Being able to get all this information in a single skeleton enables better results when making decisions about how tasks must be orchestrated.

2.1 Structure

An algorithmic skeleton represents an algorithm whose overall structure is known, but some details can be defined in a later step. This structure must be described by the developer. For this purpose, we provide atomic elements, called bones, each one implementing a specific sequential or parallel pattern such as a sequence of tasks or a farm [1].

Both bones and compound structures can be used to build new structures.

Algorithm 1 presents an algorithm, GRASP, whose structure is composed of a loop, repeating a sequence of 2 tasks (a constructive heuristic (CH), to build a random solution, followed by a local search (LS) that improves the solution), then a sequential instruction that selects (Sel) the best solution. Figure 1 is a representation of this GRASP structure.

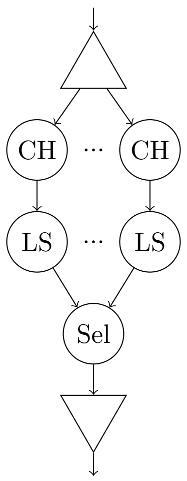

Figure 1: GRASP structure

2.2 Muscles

The circles in figure 1 are slots for tasks yet to be defined. These are called muscles in algorithmic skeleton terminology and are comparable to functions. For our library, these are implemented by regular C++ functions. This demonstrates the separation between domain code, written in muscles, and the parallelization details, written in bones and used through skeletons (i.e. structure and links). It also allows existing code to be easily used with our library.

2.3 Links

Apart from structure, our library requires the developer to define the data flow graph, that we named links, which describe how data is transferred between all tasks executions. This part is usually automatically done by the libraries, however this implies less flexibility and add constraints when defining tasks. In addition, having links explicitly defined enables potential optimizations through TMP and helps producing better implementations (e.g. by avoiding copies).

A task to execute can be either a muscle or a skeleton, in both cases it behaves as a callable. For that reason, links are described using function signatures, which contain the return type and the parameter list. Our library provides placeholder types to make the links. For example, it is possible to write that a muscle accepts as first parameter the return value of its predecessor, or the second argument of its caller. This type is then replaced by the corresponding actual type.

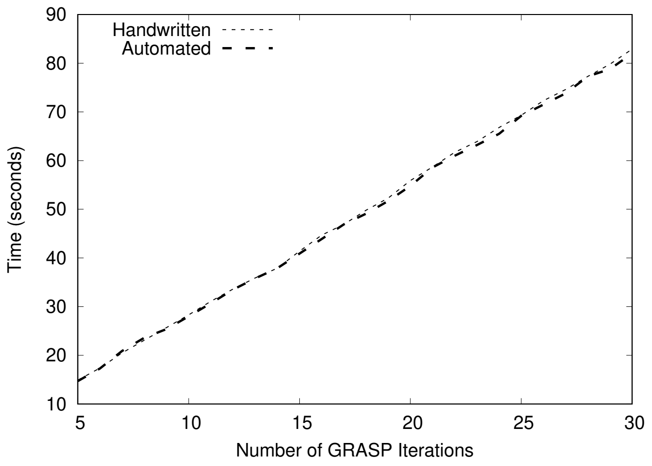

Figure 2: Execution time depending on the number of iterations

3 Performance results

We used Template Metaprogramming (TMP) in order to avoid inducing run-time overhead when compared to a handwritten implementation. To validate that objective, we ran performance measures on two versions of an algorithm to solve Travelling Salesman Problem (TSP) instances: a handwritten one and another generated by our library. The algorithm is a Greedy Randomized Adaptive Search Procedure (GRASP) whose local search is implemented by an Evolutionary Local Search (ELS), named GRASPxELS. It offers two distinct parallelizable levels. These tests have been performed on an Intel Xeon CPU E5-2670 v2 at 2.50 GHz with 20 physical cores and compiled using g++ 8.2.0 with the `O2` optimization 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 provides the same result).

Figure 2 shows the comparison between handwritten and automatically generated GRASPxELS for a sequential implementation, with a varying number of iterations from 5 to 30 for the outer GRASP loop. No significant run-time overhead is noticeable. Figure 3 also results of the comparison between handwritten and automatically generated GRASPxELS but for a parallel implementation, with a fixed number of iterations and a varying number of allotted cores. Similarly, no significant run-time overhead was measured.

Based on these measures, we conclude that we achieved an implementation of algorithmic skeletons that performs

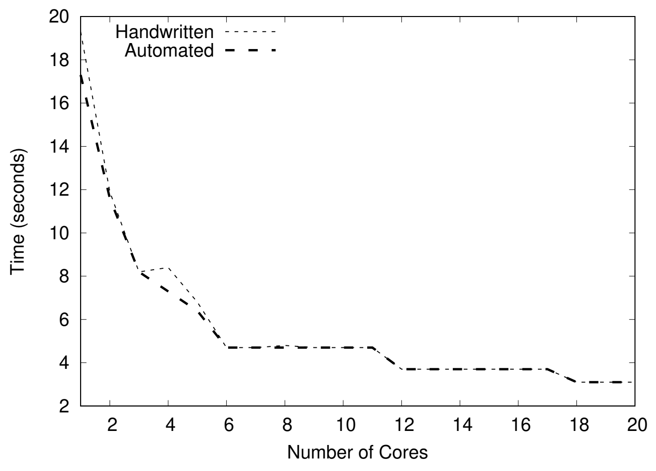

Figure 3: Execution time depending on the number of cores

as well as a handwritten solution. It enables developers to describe their algorithms entirely, sequential as well as parallelizable parts, and to define how the data is transferred between tasks. The described skeletons can then be completed by providing simple functions whose signatures correspond to the slot they try to fill. The knowledge the library has got, thanks to these algorithmic skeletons, enables compile-time algorithm analysis and more suited implementations.

References

[1] Duncan K. G. Campbell. *Towards the Classification of Algorithmic Skeletons.* YCS 276. University of York department of computer science YCS, 1996 (cit. on p. 1).

[2] Philipp Ciechanowicz, Michael Poldner, and Herbert Kuchen. “The Münster Skeleton Library Muesli - a Comprehensive Overview”. In: (2009) (cit. on p. 1).

[3] Murray Cole. “Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming”. In: *Parallel Computing* 30.3 (Mar. 2004), pp. 389–406. ISSN: 01678191. DOI: `10.1016/j.parco.2003.12.002` (cit. on p. 1).

[4] 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. 1).

[5] 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. 1).

[6] 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.* 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP 2010). Pisa: IEEE, Feb. 2010, pp. 289–296. ISBN: 978-1-4244-5673-4. DOI: `10.1109/PDP.2010.26` (cit. on p. 1).