June 2017

A tool to support FastFlow program design

Leonardo GAZZARRI [a] and Marco DANELUTTO [a] [1]
[a] *Dept. Computer Science, Univ. of Pisa*

Abstract. We describe the implementation of RPL, a shell to support structured parallel programming development in FastFlow. The shell provides ways to explore the space of functionally equivalent, alternative implementation of the same parallel applications with different non functional properties. The tool is entirely written in C++ and has been designed in such a way it can be easily extended to take into account new non functional features, refactoring and optimization rules, as well as different parallel patterns. Preliminary experimental results are shown relatively to the code generation part.

Keywords. FastFlow, parallel patterns, algorithmic skeletons, code refactoring, structured parallelism

1. Introduction

Structured parallel programming models have been investigated for a long time in two different research areas, namely HPC (as algorithmic skeletons [1,5,2] and software engineering [8] and recently part of the concepts have been inherited and exploited in widely used parallel programming frameworks, including Intel TBB, Microsoft Parallel Patterns library and, in part, by OpenMP.

In a structured parallel programming framework parallel applications are developed using programming abstractions provided as proper programming constructs of the host programming language. C++ frameworks (FastFlow, Muesli, SkeTo) provide classes modelling parallel patterns, functional frameworks (OSL) provide higher order functions modelling patterns, and so on. These programming abstractions encapsulate most (all) of the details needed to be managed to exploit the parallel patterns. The task of the parallel application programmer therefore mainly consists in picking up the more convenient pattern or the pattern composition among those available/feasible. In a number of significant cases, the framework offers a variety of patterns, each with a number of different non functional parameters (parallelism degree, scheduling policy, load balancing policy, communication mechanisms, etc.) and the very same parallel application may be implemented using different pattern compositions.

Also, different refactoring rules are know that establish *functional* equivalences among different patterns and these rules may be used to derive additional functionally equivalent pattern implementations for a given parallel application. A notable example is the so called *map fusion* rule stating that

> the sequence of two map patterns, one applying function f and another one applying function g over all the items of a given collection (vector, array, tree, ...) is equivalent to a single map pattern applying the composition of f and g: