Parallel Regions using a PDDL Formalization

Claudio Scheer
claudio.scheer@edu.pucrs.br
Master's Degree in Computer Science
Pontifical Catholic University of Rio Grande do Sul - PUCRS
Porto Alegre - RS, Brazil

Abstract

Testing whether a region in a program can be run in parallel is not an easy task. This process takes a lot of time and, consequently, finantial resources. As an approach to this problem, this paper proposes formalizing a compiler with PDDL. With this formalization, it will be possible to reduce the identification of a parallel code to a conventional **search problem**.

There are still many programs that do not exploit the parallel capacity of today's processors. One of the main reasons for this is because the cost to rewrite a program is high. Among the steps involved in the process of migrating a program to a parallel approach, the most expensive is to identify the regions of the program than can be executed in parallel. In addition to identifying the regions, it is necessary to validate whether the parallel execution of that region brings positive results.

According to (del Rio Astorga et al. 2018), loops detection, variable dependencies, identifying whether the arguments are read or written, among other analyzes, are the main patterns that identify a region parallel in a program. Over time, these caracteristics may change and the static analysis will need to be changed.

Therefore, instead of using a static analysis of the source code, the approach in this paper will reduce the problem of identifying a region that can be executed in parallel to a **search problem**. The compiler domain will be formalized using PDDL. In a nutshell, the domain will function as a source code compiler. Section Technical Approach discusses in more detail how the PDDL domain will work.

A PDDL domain describe actions that can be performed in an initial state, to achieve a set of goals. In this paper, the source code will be treated as the initial state. The set of goals will be the complete execution of all instructions in the source code, in the correct order. A problem formalized in PDDL cannot understand source code written in C++ as input, for example. The initial state must be a set of predicates. Hence, the source code provided as an initial state for the planner must be mapped to a set of predicates. This process is the bottleneck of the proposed approach. However, this task can be easily automated (I was thinking of creating this tool, but first I will implement the PDDL domain).

The planner will be reponsible for finding the set of predicates that can be run in parallel. In the Section Results Evaluation, I discuss more about how I will evaluate whether the results were positive or not. In addition, in Section Project Management, I show the schedule that will be followed to execute this proposal.

Technical Approach

Compilers are the programs that convert written source code into executable code. Compilers, in general, are complex programs. In general, the compiler needs to break the source code into tokens[1] and represent it in a tree. Over this tree, the compiler performs some optimizations, such as removing unused variables and unreachable codes. After all this, the compiler can convert the tree into machine instructions.

The compiler must support all instructions provided by the programming language and be able to translate it for every operating systems. It is difficult to develop a compiler. In the proposed approach it will not be necessary to deal with all these complexities. The compiler will be formalized as a PDDL domain, with support for arithmetic and binary operations, functions and loop intructions.

In theory, this approach would work for all programming languages. However, I will create the actions and the predicates from the perspective of a source code written in C++.

There are still some questions to be answered during the execution of the project, as follows:

1. Is the compiler domain capable of handling fluent variables and predicates?
2. Is the compiler domain capable of performing operations with strings?
3. Which planners should I test the compiler domain on?
4. How does a planner find a parallel region?
5. Can I set a weight for the planner to get regions that are really worth running in parallel?

Copyright © 2020, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

[1]A token can be a keyword, a variable name, an operator, etc.

Results Evaluation

As previously described, the goal of the planner will be to execute all instructions in the correct order and find instructions that can be executed in parallel. Hence, to evaluate the correct output of the planner, I will map the predicates back to the source code and validate the parallel execution proposed by the planner.

The main objective of decoding the planner output in a source code is to test whether the parallel execution really brings positive results. A positive result, in the proposed approach, can be understood as the execution of a problem in less time.

Project Management

Task	**Start**	**End**
Understand better compilers	06-01-2020	06-03-2020
Support sum instruction	06-03-2020	06-07-2020
Support proposed instructions	06-08-2020	06-15-2020
Evaluate results	06-16-2020	06-20-2020
Write paper	06-20-2020	06-25-2020

Conclusion

In summary, this approach will reduce the problem of finding a parallel region to a **search problem**. As an output, the planner will execute all instructions of the program and identify the regions that can be executed in parallel. Clearly, these approach needs a validation in terms of testing to find out if the regions are really parallel.

The approach proposed in this paper is not conventional for finding parallel regions. However, if the results are positives, this approach can be used to reduce the time and cost to find these regions.

References

del Rio Astorga, D.; Dolz, M. F.; Sánchez, L. M.; García, J. D.; Danelutto, M.; and Torquati, M. 2018. Finding parallel patterns through static analysis in c++ applications. *The International Journal of High Performance Computing Applications* 32(6):779–788.