Automatic Parallelisation of Rust Programs at Compile Time

Michael Oultram

***Abstract*—A significant amount of research in automatic translation from sequential to parallel source code is focused on FORTRAN and C, both of which are unsafe programming languages. Due to the languages being unsafe, a lot of effort is spent in reconstructing the program, especially with FORTRAN and its GOTO statements. This paper explores the literature's ideas and extract the key information to apply the same transformations to programs written in rust: a safe programming language. TODO: Expand, and cleanup**

I. Introduction

Kish (2002) estimated the end of Moore's Law of miniaturization within 6-8 years or earlier (based on their publication date) and as such, manufacturers have been increasing processors' core count to increase processor performance (Geer 2005). Writing parallelised programs to take advantage of these additional cores has some difficulty and often requires significant changes to the source code. Automatically converting sequential source code into parallelised source code is one solution to this problem, and it is the solution that this paper explores.

II. Literature Review

Most research is for FORTRAN and the DO loops (Banerjee 1993).

TODO: Look at the different models, try to explain the differences

Some people have converted C-to-CUDA (Baskaran, Ramanujam and Sadayappan 2010; Verdoolaege et al. 2013).

III. Problem Details

The rust compiler allows for plugins of different types. A syntax extension plugin can modify the abstract syntax tree of any annotated function. An early lint pass plugin can see any of the functions, with macros expanded, but it cannot edit them. The analysis stage is run by the linter and the modification stage is run by the syntax extension plugin. The rust compiler executes syntax extension plugins first, and then the linter plugins. Analysis stage must come before the modification stage, so compiling is done twice (once for each stage).

When the plugin is loaded, it determines which stage it is by looking for a .auto-parallelize file. If this file does not exist, then it is the analysis stage. If the file does exist, the files content is loaded into a struct using the 'serde_json' crate and the stage is updated to be the modification stage.

A. Analysis stage

On the first compilation, the syntax extension plugin would do nothing. The linter plugin would view the entire abstract data tree and analyse what each statement depends on and modifies. This would create a dependency tree, where any two statements that are independent can be run in parallel. The linter plugin would use this dependency tree to determine which parts should be parallelised, and save this information to a file.

Detecting the end of the analysis stage required some work.

B. Modification stage

On the second compilation, the syntax extension would be able to read the file the linter plugin created on the previous compilation. This lists all the changes required and the syntax plugin can apply those changes to the abstract syntax tree function by function. The linter plugin would also be able to view this file, and it could produce compiler warnings for any function that could be parallelised that is missing an annotation.

TODO: Show example code and the desired transformation

References

Banerjee, Utpal (1993). *Loop transformations for restructuring compilers: the foundations*. Boston: Kluwer Academic Publishers. ISBN: 079239318X.

Baskaran, Muthu, Jj Ramanujam and P Sadayappan (2010). "Automatic C-to-CUDA code generation for affine programs". In: *Compiler Construction*, pp. 244–263.

Geer, David (2005). "Chip makers turn to multicore processors". In: *Computer* 38.5, pp. 11–13.

Kish, Laszlo B (2002). "End of Moore's law: thermal (noise) death of integration in micro and nano electronics". In: *Physics Letters A* 305.3, pp. 144–149.

Verdoolaege, Sven et al. (2013). "Polyhedral parallel code generation for CUDA". In: *ACM Transactions on Architecture and Code Optimization*. ISSN: 15443566. DOI: 10.1145/2400682.2400713.