Automatic Parallelisation of Rust Programs at Compile Time

Project Proposal

1 Problem

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. Is it possible to automate these changes to convert sequential code into parallelised code? Previous attempts at solving this problem include D'Hollander, Zhang and Wang (1998) where they automated parallelisation of sequential FORTRAN code and Baskaran, Ramanujam and Sadayappan (2010) where they automated conversion of sequential C into CUDA code. Both of these approaches use unsafe programming languages significant complexity **TODO: is complexity the right word to use?** to their solutions. Instead, can this problem be solved more easily with a safe programming language like rust (*The Rust Programming Language* 2017)?

2 Approach

My approach involves writing a plugin for the rust compiler. The rust compiler plugin system allows for different types of plugin, which are run at different stages of the compilation. A syntax extension plugin can manipulate the abstract syntax tree of any annotated function, one function at a time. A linter plugin can see the abstract syntax tree of every function without annotations but it cannot manipulate anything.

This problem requires read access to the entire abstract syntax tree, and then modify access to the parallelisable parts. Due to the order of execution, the syntax extension plugin would be executed first and then the linter plugin but I would require them to be executed in the opposite order. To solve this problem, I propose that the program is compiled twice.

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.

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.

2.1 Requirements

I believe that I have access to all the required software and hardware for this project. Below is listed most of what I am planning to use:

- A computer to program on
- Source code of some rust programs to parallelise
- Rust compiler (and it's source code)
- A text editor to write code in (Atom)

2.2 Estimated Timeline

Estimated End Date	**Milestone**
TBD	Create two plugins, a linter and a syntax extension, and get them to communicate via a file
TBD	Get access to the abstract syntax tree inside the linter plugin
TBD	In the linter plugin, analyse each statement to see what variables they depends on and what variables are modified (if any)
TBD	Produce a dependency tree for the entire program based on this analysis
TBD	Look for areas in the tree which do not depend on one another, these areas could be run in parallel
TBD	Save these areas to the shared file used for communication. Make sure that the syntax extension plugin can read this file correctly
TBD	In the syntax extension plugin, first try one statement in parallel to test it works and then try to run everything in parallel
TBD	Run tests against rust programs for program correctness and calculate speedup/slowdown against the serial version.
TBD	In the linter plugin, try to analyse the speed of each parallelisable part. Threads have an overhead so it may be faster to run in serial for some cases. Record this into the shared file.
TBD	In the syntax extension plugin, take into consideration the time information and only run parts in parallel if it would be faster to run in parallel
TBD	Run more tests against rust programs for speedup/slowdown.

2.3 Possible Extensions

If I'm ahead of schedule with the coding section of this project, I could look into parallelising 'if' statements which have a slow condition. Each branch of the 'if' would be run in a separate thread using cloned data. When the condition is finally calculated the incorrect branches would need to be deleted. This kind of parallelisation is different from the project plan as some threads are "thrown away".

Another possible extension upon the previous extension is to utilise the GPU using CUDA in cases where a large number of threads are doing the exact same task on different data. I feel it is very unlikely that I would have time for this extension, and I'm unsure of how much real world code would be written in such a way that it could be automatically run on the GPU efficiently.

3 Evaluation

3.1 Disadvantages to the Chosen Approach

While I think the approach I have chosen is the best option to solve the problem, there are a few downsides to this proposed approach.

To take an existing sequential rust program and compile it requires a few steps. The compiler plugin crate would need to be imported, and every function would need to be annotated so that the plugin has edit access to the abstract syntax tree for that function.

Due to the limitation of rust compiler plugins, the program would need to be compiled twice. The first compile would allow the entire syntax tree to be analysed for parallelisable sections. The second compile would edit the syntax tree to move the parallelisable sections into separate threads.

Project Risks:

- Compiler code too complicated to use
- Size of task is too large
- No idea how to analyse the speed to statements at compile time yet
- “Independent tasks can be run in parallel” is true in my head, but maybe not in practice
- Rust may not give me all the guarantees that I think it would give me

3.2 Other Approaches

There are other approaches to the problem that I am trying to solve (some of which have already been implemented). I will explain these other approaches and why I didn’t choose them.

3.2.1 Make a custom rust compiler

One of the downsides of the chosen approach is that the program has to be compiled twice. I may be able to bypass this limitation by making a fork of the rust compiler instead of using compiler plugins. It would then be possible to examine the entire abstract syntax tree and then edit it afterwards. However, this approach would add significant complexity to the project as the rust compiler code is very complex.

3.2.2 Use a different language from rust

There are many programs out there, written in many different languages. So why rust? Rust is not even in the top 25 of most popular programming languages according to StackOverflow (2017). Using a more popular language would make this tool more useful for more programs. Rust isn’t unpopular as it is the most loved language and the 10th language on the most want to lean list (StackOverflow 2017). The main reason for using rust is the guaranteed memory safety and threads without race conditions which would help significantly with the task.

3.2.3 Manually annotate the parts of the program that are parallelisable

(Dagum and Menon 1998)

3.2.4 Just run the sequential code

Running code in multiple threads has some overhead.

3.3 Measuring Project Success

I can evaluate the successfulness of my solution by measuring the speedup of parallelising the program vs the original serial code. I'll look at existing programs written in rust to see any real world impact.

4 References

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

Dagum, Leonardo and Ramesh Menon (1998). "OpenMP: an industry standard API for shared-memory programming". In: *IEEE computational science and engineering* 5.1, pp. 46–55.

D'Hollander, Erik H, Fubo Zhang and Qi Wang (1998). "The fortran parallel transformer and its programming environment". In: *Information sciences* 106.3-4, pp. 293–317.

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.

StackOverflow (2017). *Stack Overflow Developer Survey 2017*. URL: `https://insights.stackoverflow.com/survey/2017` (visited on 09/10/2017).

The Rust Programming Language (2017). URL: `https://www.rust-lang.org/en-US/` (visited on 09/10/2017).