# 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  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** |
|---|---|
| 30/10/2017 | Create two plugins, a linter and a syntax extension, and get them to communicate via a file |
| 13/11/2017 | Get access to the abstract syntax tree inside the linter plugin |
| 27/11/2017 | In the linter plugin, analyse each statement to see what variables they depends on and what variables are modified (if any) |
| 11/12/2017 | Produce a dependency tree for the entire program based on this analysis |
| 25/12/2017 | Look for areas in the tree which do not depend on one another, these areas could be run in parallel |
| 08/01/2018 | Save these areas to the shared file used for communication. Make sure that the syntax extension plugin can read this file correctly |
| 22/01/2018 | In the syntax extension plugin, first try one statement in parallel to test it works and then try to run everything in parallel |
| 29/01/2018 | Run tests against rust programs for program correctness and calculate speedup/slowdown against the sequential version |
| 12/02/2018 | 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 sequential for some cases. Record this into the shared file |
| 26/03/2018 | 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 |
| 05/03/2018 | 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 branches which have a slow condition (i.e. 'if' statements). Each branch would run isolated in separate thread using cloned data. When the condition is finally calculated the incorrect branches would be terminated. This kind of parallelisation is different from the project plan as some threads are "thrown away".

Another extension upon the previous extension would be to utilise the GPU in cases where the same isolatable task is repeated a large number of times but on different data such that the overhead of moving this task to the GPU is worthwhile. It is very unlikely that I would attempt this extension due to the amount of work required, and I'm unsure about how much real world code exists that could be automatically run on the GPU efficiently.

# 3 Evaluation

To evaluate whether the chosen approach is the best option to solve the problem, it is worth looking at the downsides to this approach, and look other possible approaches. I will also describe how I will evaluate the final solution to see whether it solves the original problem.

## 3.1 Disadvantages to the Chosen Approach

The first downside is that the target program would need to be compiled twice due to the limitation of rust compiler plugins as explained section 2. It might be possible to edit the rust compiler to make it so that the program needs to be compiled once, but this has its own downsides as explained in 3.2.1.

The second downside is installing the compiler plugins. To take an existing sequential rust program and compile it so that it runs in parallel would require a few steps. The compiler plugin crate would need to be imported, and every function would need to be annotated so that the syntax extension plugin has edit access to the abstract syntax tree for each function.

## 3.2 Other Approaches

Below are a few other approaches to the problem that I considered before reaching the chosen approach. Each alternative approach looks at the potential benefits over the chosen one and then explains why I did not choose that alternate approach.

### 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 seems infeasible for me as the rust compiler code complexity would add too much time to the project.

### 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 in the top 25 of most popular programming languages, according to StackOverflow (2017), so the number of existing programs may smaller than a more popular language. Rust is not unpopular however making it the most loved language and the 10th language on the most want to lean list (StackOverflow 2017). If there are not enough existing programs to justify specialising the tool just for rust (which there probably is), the number programs should increase exponentially in the future. The main reason I chose rust is because it is a safe language with guaranteed memory safety and threads without race conditions which would help significantly with the task.

### 3.2.3 Manual annotation of the parts of the program that are parallelisable

Instead of analysing the program for parallelisable parts, let the programmer use annotations to show which functions can be run in parallel, similar to OpenMP developed for C++ (Dagum and Menon 1998). This would be significantly simpler than the chosen approach as the analysation and dependency tree steps of the compilation could essentially be removed. The programmer is responsible for telling the compiler what can and cannot be parallelised. If the programmer accidentally makes a function parallel that should not be run in parallel, then the compiler would either need to spot this error (using the analysis step that was removed) or it would create an incorrect program. This approach would also not allow for automatic translation from sequential to parallel code and so it is not that much better than using threads manually.

### 3.2.4 Just run the sequential code

All of this analysis at compile time (and the fact that it would compile twice in the chosen approach) would increase compile times and running code in multiple threads has some runtime overhead. Is it worth the development time of automating the conversion of sequential code into parallelisable code? This is a valid argument if the number of sequential programs that could benefit from this software is small enough such that it would be faster and easier to redevelop these programs. It must also be true that developing future programs as parallel programs would have the same difficulty as developing it as a sequential program for this argument to hold. Since developing parallel programs is more difficult than developing sequential programs, and there are probably enough programs that could benefit from this software, it is worth the development time to create this piece of software.

## 3.3 Measuring Project Success

I can look at existing programs written in rust to see if there is any real world impact by measuring the speedup of the parallelised program vs the original sequential code. I can also calculate what percentage of the static sequential code has been parallelised, and what percentage of the runtime is using the parallel code. From these statistics, I can evaluate how well my solution solves the problem.

# 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).