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

A. Parallelisation models:

Static parallelism done at compile time: Bondhugula et al. (2008) describes automatic transformations for communication-minimized parallelisation and locality optimization in the Polyhedral Model. It is far too confusing...

Beletska et al. (2011) shows that iteration space slicing extracts more coarse-grained parallelism than Affine Transformation Framework

Speculative parallelism done at run time: There is speculative parallelism which is at run time (Yiapanis, Brown and Luján 2015).

B. Implementations

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

OpenMP (Dagum and Menon 1998; Lam 2011).

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

III. Problem Details

TODO: Write about what exactly I want to parallelise. Which methods from the literature am I following and why?

TODO: Include what rust is, what plugins can do in this section 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 rust compiler executes syntax extension plugins first, and then the linter plugins.

TODO: For optimisations: Describe what is slow and what is required for optimisation. Good example sequential and parallel. Explain why/when faster (i.e. long list). Bad example of sequential. Explain why it cannot be converted

A. Parallel Function Optimisations

If the function's arguments are not modifiable references and the function does not contain an unsafe block, then the function can be run in parallel.

```
fn main() {
    let i = 10;
    // A lot of stuff
    println!("{}", fibinacci(i));
}

fn fibinacci(n: u32) -> u64 {
    match n {
        0    => 0,
        1, 2 => 1,
        _    => fibinacci(n-1) + fibinacci(n-2),
    }
}
```

into

```
fn main() {
    let i = 10;
    let fib = fibinacci_parallel(i);
    // A lot of stuff
    println!("{}", fib.join());
}

fn fibinacci_parallel(n: u32) -> JoinHandle<u64> {
    thread::spawn(move || {
        match n {
            0    => 0,
            1, 2 => 1,
            _    => {
                let n1 = fibinacci_parallel(n-1);
                let n2 = fibinacci_parallel(n-2);
```

```
                n1.join() + n2.join();
            },
        }
    })
}

fn fibinacci(n: u32) -> u64 {
    let fib = fibinacci_parallel(n);
    fib.join()
}
```

These changes allow for fibonacci to start calculating as soon as i is decided, instead of waiting for 'lots of stuff' to be executed. Also n1 and n2 are executed in parallel. fibinacci is changed to use the parallel version and then immediatley tries to get the result. This should allow for any external functions that are not modified to still work.

B. For-Loop Optimisations

If all the loop iterations are independent of each other, then we can run all the iterations at the same time. Example:

```
let mut list = vec![1,2,3,4,5];
for i in 0..list.len() {
    list[i] *= 2;
}
```

could be converted into **TODO: Make code below compile**

```
let mut list = vec![1,2,3,4,5];
// Create a communication channel
let (tx, rx) = mpsc::channel();
let handles = vec![];
// Start all threads
for i in 0..list.len() {
    let element = list[i];
    let tx = tx.clone();
    handles.push(thread::spawn(move || {
        let result = element * 2;
        tx.send((i, result));
    }));
}
// Wait for all threads to be finished
for handle in handles {
    handle.join();
}
// Receive all the results and update list
for _ in 0..list.len() {
    let (i, result) = rx.receive();
    list[i] = result;
}
```

In this simple case, the inner for loop has one operations and so the threaded version would be slower due to overhead. If the operation inside the loop was more complex, and the list was longer, then this conversion would make sense.

Real world example:

```
fn crack_password(dictionary: &Vec<String>,
 ↪  hash_password: String) -> Option<String>{
    for word in dictionary {
        // Hash word using Sha256
        let mut hasher = Sha256::new();
        hasher.input_str(word);
        let hash_word = hasher.result_str();

        // Check if hash matches
        if hash_password == hash_word {
            return Some(word.clone());
        }
    }
    None
}
```

C. Branch Optimisations

In the previous optimisations, all the code that is run in parallel would have be run in sequential normally. This optimisation is for if statements which have a very slow condition. Each side of the branch is run in parallel, and then when the condition is finally worked out, the correct branch is kept.

IV. Design

TODO: Describe how I will use the features of rust plugins The analysis stage is run by the linter and the modification stage is run by the syntax extension plugin. 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.

Beletska, Anna et al. (2011). "Coarse-grained loop parallelization: Iteration Space Slicing vs affine transformations". In: *Parallel Computing*. ISBN: 9780769536804. DOI: 10.1016/j.parco.2010.12.005.

Bondhugula, Uday et al. (2008). "Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. ISBN: 3540787909. DOI: 10.1007/978-3-540-78791-4_9.

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.

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.

Lam, Nam Quang (2011). "A Machine Learning and Compiler-based Approach to Automatically Parallelize Serial Programs Using OpenMP". In:

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.

Yiapanis, Paraskevas, Gavin Brown and Mikel Luján (2015). "Compiler-Driven Software Speculation for Thread-Level Parallelism". In: *ACM Transactions on Programming Languages and Systems* 38.2, pp. 1–45. ISSN: 01640925. DOI: 10.1145/2821505. URL: http://dl.acm.org/citation.cfm?id=2866613.2821505.