Automatic Parallelisation of Rust Programs at Compile Time

Michael Oultram
Student ID: 1428105

Dr Ian Batten
Project Supervisor

***Abstract*—Processors have been gaining more multi-core performance which sequential code cannot take advantage of. One solution to this problem is to automatically convert sequential source code into parallelised source code. The literature for this topic is explored and it is split into two main areas: theoretical models of automatic parallelisation and real-world parallelising compilers. Three sequential elements of Rust programs are converted manually by this paper and a design is outlined to automate these conversions in a new parallelising compiler for the safe programming language Rust.**

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.

TODO: Paragraphs do not link yet

"Rust is a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety" (*The Rust Programming Language* 2017). This brief introduction to Rust will explain the elements of the language necessary for the reader to understand for later sections of this paper. For further understanding of the language, it is recommended that the reader looks at *The Rust Book* (2017).

TODO: Explain the Rust language syntax and memory model

II. Related Work

TODO: Write up and cleanup. Also read the papers

A. Parallelisation Models

1) Static parallelism: Feautrier (1992) describes one model of a parallel program as a set of operations Ω on an initial store, and a partial ordering binary relation Γ also known as a dependency tree. It is shown that this basic model of a parallel is equivalent to affine scheduling, where Ω and Γ are described as linear inequalities.

Bondhugula et al. (2008) describes the affine transformations required to allow for communication-minimized parallelisation and locality optimization.

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

2) Speculative parallelism: Prabhu, Ramalingam and Vaswani (2010) provides an API for C# to allow the programmer to specify areas of speculative parallelism. Yiapanis, Brown and Luján (2015) moves this into the compiler.

B. Parallelisation Implementations

D'Hollander, Zhang and Wang (1998) converts FORTRAN and DO loops. Eigenmann, Hoeflinger and D. Padua (1998) parallelises benchmarks using FORTRAN. Kim et al. (2000) converts FORTRAN using OpenMP.

Rauchwerger and D. A. Padua (1999) uses FORTRAN speculatively on for loops. Quiñones et al. (2005) uses speculation and iteration space slicing together.

OpenMP is a programming interface for shared memory programming. Dagum and Menon (1998) compares OpenMP to alterative parallel programming models. Lam (2011) extends OpenMP using machine learning to automate the parallelisation.

GPUs are another focus due to their number of threads. Baskaran, Ramanujam and Sadayappan (2010) have converted C-to-CUDA. Verdoolaege et al. (2013) generates CUDA code using the Polyhedral model.

III. Problem Details

TODO: Write about what exactly I want to parallelise. Which methods from the literature am I following and why? Include what Rust is, what plugins can do in this section

The literature focuses on unsafe languages such as C/C++ and FORTRAN and as a result, most of their methods revolve around understanding conflicts between iterations. Due to the different memory model in Rust, the information that the literature's approaches tried to gather is readily available.

The Rust compiler allows for plugins of different types but there are two types that are of interest to this problem. Written in the order of execution, they are:

- Syntax Extension: can modify the abstract syntax tree of any annotated function.
- Early Lint Pass: can see abstract syntax tree of each uncompiled function, with macros expanded, without annotations, but it cannot edit them.

To automate the parallelisation of Rust programs, the process must be manually to understand the problem. Below are some examples of sequential code and their manually calculated optimisations that could be automated using a Rust compiler plugin. They are ordered based on their complexity.

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*

In some cases the arguments of a function are known well before the result of the function is required. In sequential programs, only one function is executed at once and so when the result is required, the program switches to that function. This function could be started in the background as soon as the arguments are decided if the program was parallelised. This would allow the result to be ready for when it is requested or at least closer to ready than the pure sequential code. In Rust terms, 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. **TODO: Check this fact**

Example: Algorithm 1 is a program that calculates Fibonacci numbers, written in a very inefficient way. The main method knows that $i = 10$ on the first line, and since it is not mutable, this cannot be changed. But the main method does 'a lot of stuff' which will not affect the `fibonacci` function before calling it. Just running the `fibonacci` function earlier would not improve performance, as now 'a lot of stuff' must wait for `fibonacci` to finish. Ideally, we want to start calculating `fibonacci` at the very beginning at the same time as doing 'a lot of stuff' and wait for the result when we need it.

Algorithm 1 Sequential Fibonacci Function

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

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

Algorithm 2 Parallel Fibonacci Function

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

fn fibonacci_parallel(n: u32) -> JoinHandle<u64> {
    thread::spawn(move || {
        match n {
            0    => 0,
            1, 2 => 1,
            _    => {
                let n1 = fibonacci_parallel(n-1);
                let n2 = fibonacci_parallel(n-2);
                n1.join() + n2.join();
            },
        }
    })
}

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

Algorithm 2 shows one way of converting Algorithm 1 into a more parallelised version. 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. fibonacci 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. However, in most cases, loops are only partially parallelisable.

Example: Algorithm 3 is a real world example where the for loop is combined with an if statement which returns the password for the first valid hash. If we were to naively put the contents of the for loop into separate threads, as shown in Algorithm 4, then we may end up with the wrong result. In the sequential for loop, the first password in the list to match the hash would be returned. In the naive parallel version, the password returned depends on the order the threads are run.

Algorithm 5 shows that we split the for loop into two parts, the hashing part (which is parallelisable) and the verifying part (which is not as parallelisable if we want to keep the order). Algorithm 6 is the final complete parallelisation of Algorithm 3. Each word is hashed in it's own thread and the hash is compared to `hash_password`. The result of

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

Algorithm 3 Sequential Password Cracker

```
fn crack_password(dictionary: &Vec<String>,
                  hash_password: String)
                  -> Option<String> {
    for word in dictionary {
        // Hash word using Sha256
        let mut sha = Sha256::new();
        sha.input_str(word);
        let hash_word = sha.result_str();
        // Check if hash matches
        if hash_password == hash_word {
            return Some(word.clone());
        }
    }
    // No hash matched
    None
}
```

Algorithm 4 Naive Parallel Password Cracker

```
fn crack_password(dictionary: &Vec<String>,
                  hash_password: String)
                  -> Option<String> {
    // Create a communication channel
    let (tx, rx) = mpsc::channel();
    // Start a thread for each dictionary entry
    for i in 0..dictionary.len() {
        let word = dictionary[i].clone();
        let tx = tx.clone();
        thread::spawn(move || {
            // Hash word using Sha256
            let result = {
                let mut sha = Sha256::new();
                sha.input_str(word);
                let hash_word = sha.result_str();
                // Send result via channel
                if hash_password == hash_word {
                    Some(word)
                } else {
                    None
                }
            };
            tx.send(result);
        });
    }
    // Receive up to dictionary.len() results
    for _ in 0..dictionary.len() {
        if let Some(result) = rx.receive() {
            return result;
        }
    }
    // No hash matched
    None
}
```

condition. Each side of the branch is run in parallel, and then when the condition is finally worked out, the correct branch is kept.

TODO: Code Examples

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).

Algorithm 5 Refactored Sequential Password Cracker

```
fn crack_password(dictionary: &Vec<String>,
                  hash_password: String)
                  -> Option<String> {
    let mut hashes = vec![];
    for word in dictionary {
        // Hash word using Sha256
        let mut sha = Sha256::new();
        sha.input_str(word);
        let hash_word = sha.result_str();
        hashes.push(hash_word);
    }
    // Check if hash matches
    for hash_word in hashes {
        if hash_password == hash_word {
            return Some(word.clone());
        }
    }
    // No hash matched
    None
}
```

Algorithm 6 Parallel Password Cracker

```
fn crack_password(dictionary: &Vec<String>,
                  hash_password: String)
                  -> Option<String> {
    // Create a communication channel
    let (tx, rx) = mpsc::channel();
    for i in 0..dictionary.len() {
        let word = dictionary[i].clone();
        let tx = tx.clone();
        thread::spawn(move || {
            // Hash word using Sha256
            let result = {
                let mut sha = Sha256::new();
                sha.input_str(word);
                let hash_word = sha.result_str();
                // Check if hash matches
                if hash_password == hash_word {
                    Some(word)
                } else {
                    None
                }
            };
            tx.send((i, result));
        });
    }
    // Receive all the results
    // Have to return same result as sequential
    let mut results = vec![None; list.len()];
    let mut verified_upto = -1;
    for _ in 0..results.len() {
        // Receive result and store in location
        let (i, result) = rx.receive();
        results[i] = Some(result);
        // Check for final result
        for i in 0..results.len() {
            if let Some(result) = results[i] {
                if let Some(word) = result {
                    return word;
                }
            } else {
                // Have not received i result yet
                break;
            }
        }
    }
    // No hash matched
    None
}
```

Algorithm 7 Sequential Slow If

```
if slow_condition() {
    // True Branch
} else {
    // False Branch
}
```

Algorithm 8 Parallel Slow If

```
let true_branch = thread::spawn(move || {
    // True Branch
});
let false_branch = thread::spawn(move || {
    // False Branch
});
if slow_condition() {
    true_branch.join()
} else {
    false_branch.join()
}
```

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.

References

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.

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.

Eigenmann, Rudolf, Jay Hoeflinger and David Padua (1998). "On the automatic parallelization of the Perfect Benchmarks (R)". In: *IEEE Transactions on Parallel and Distributed Systems* 9.1, pp. 5–23.

Feautrier, Paul (1992). "Some efficient solutions to the affine scheduling problem. I. One-dimensional time". In: *International Journal of Parallel Programming*. ISSN: 08857458. DOI: 10.1007/BF01407835.

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

Kim, Hong Soog et al. (2000). "ICU-PFC: An automatic parallelizing compiler". In: *Proceedings - 4th International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region, HPC-Asia 2000*. ISBN: 0769505902. DOI: 10.1109/HPC.2000.846552.

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:

Prabhu, Prakash, Ganesan Ramalingam and Kapil Vaswani (2010). "Safe programmable speculative parallelism". In: *ACM SIGPLAN Notices*. ISSN: 03621340. DOI: 10.1145/1809028.1806603.

Quiñones, Carlos García et al. (2005). "Mitosis compiler: an infrastructure for speculative threading based on precomputation slices". In: *ACM Sigplan Notices*. ISSN: 0362-1340. DOI: http://doi.acm.org/10.1145/1064978.1065043.

Rauchwerger, Lawrence and David A. Padua (1999). "The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization". In: *IEEE Transactions on Parallel and Distributed Systems*. ISSN: 10459219. DOI: 10.1109/71.752782.

The Rust Book (2017). URL: https://doc.rust-lang.org/book/ (visited on 12/11/2017).

The Rust Programming Language (2017). URL: https://www.rust-lang.org (visited on 12/11/2017).

Verdoolaege, Sven et al. (2013). "Polyhedral parallel code generation for CUDA". In: *ACM Transactions on Archi-*

tecture 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.