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.**

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. One solution to this problem, which is the focus of this paper, is to automatically transform sequential source code into parallelised code. This solution, if achieved, would allow for existing (open-source) sequential programs to take advantage of the new hardware. It would also make developing new programs easier for programmers, as they can just write the easier sequential code and let the compiler make it run in parallel.

Section II examines existing implementations by other authors and finds that most solutions focus on unsafe languages like C++ and FORTRAN. Since a significant proportion of existing programs are written in these languages **TODO: source?**, it makes sense for other authors to focus on these languages. There is a downside to focusing on these unsafe languages, dependency analysis becomes much more difficult. Because of this, the design aspects of this paper will use the safe programming language rust.

"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: Briefly explain the Rust language syntax and memory model

II. Related Work

TODO: Write up and cleanup. Also read the papers

For the parallelisation models part we look at theoretical models of automatic parallelism. The static parallelism subsection shows related work where the schedule is fixed and calculated at 'compile' time. Speculative parallelism **TODO: finish sentence**.

For the parallelisation implementations part **TODO: finish sentence**.

A. Parallelisation Models

1) Static parallelism: Feautrier (1992a) and Feautrier (1992b) 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. Finding a solution where these linear inequalities hold produces a schedule for the program where dependent statements are executed in order. There are some programs where no affine schedule exists. Bondhugula et al. (2008) uses the affine scheduling model on perfectly, and imperfectly nested loops. They describe the transformations needed to minimise the communication between threads, further increasing the performance of the parallelise.

A different method to affine scheduling is iteration space slicing introduced by Pugh and Rosser (1997). "Iteration space slicing takes dependency information as input to find all statement instances from a given loop nest which must be executed to produce the correct result". Pugh and Rosser (1997) shows how this information can be used to transform loops on example programs to produce a real world speedup. Beletska et al. (2011) shows that iteration space slicing extracts more coarse-grained parallelism than affine scheduling.

2) Speculative parallelism: Zhong et al. (2008) shows that some parallelisable optimisations are hidden in loops, such that affine scheduling and iteration space splicing cannot find. They propose a method that runs future loop iterations in parallel with past loop iterations. If a future loop iteration accesses some shared memory space, and then a past iteration modifies that location, the future loop iteration is

'undone' and restarted. It is shown that this method increases the amount of the program that is parallelised.

Prabhu, Ramalingam and Vaswani (2010) provides an API for C# to allow the programmer to manually specify areas of speculative parallelism.

Yiapanis, Brown and Luján (2016) 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.

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. Kim et al. (2000) converts FORTRAN using OpenMP automatically. 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: Which methods from the literature am I following and why?

The literature focuses on unsafe languages such as C/C++ and FORTRAN. As a result, most of their methods revolve around understanding conflicts between statements/loop iterations. Due to the different memory model in Rust, the dependency information that the literature's approaches tried to gather is more 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. This section looks at some examples of sequential rust code and their manually calculated optimisations. To simplify the explanation of how a sequential program could be parallelised, threads are used as if they have no overhead. In reality this is not the case and so in section IV, where this process is automated using a Rust compiler plugin, `thread::spawn` is replaced with something more optimal.

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 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.

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()
}
```

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 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 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

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
}
```

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
}
```

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. This concept can be expanded for when there are multiple branches such as `match` statements.

TODO: Code Examples

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

```
fn slow_if(a: u32, b: u32) {
    if slow_condition(a, b) {
        let c = b - a;
        (a * c) + 5
    } else {
        let c = a * b;
        (c + 19) ^ 2
    }
}
```

Algorithm 8 Parallel Slow If

```
fn slow_if(a: u32, b: u32) {
    let true_branch = {
        let a = a.clone();
        let b = b.clone();
        thread::spawn(move || {
            let c = b - a;
            (a * c) + 5
        })
    };
    let false_branch = {
        let a = a.clone();
        let b = b.clone();
        thread::spawn(move || {
            let c = a * b;
            (c + 19) ^ 2
        })
    };
    if slow_condition(a, b) {
        true_branch.join()
    } else {
        false_branch.join()
    }
}
```

IV. Design Overview

A typical parallelising compiler, such as those described in section II, have a few stages. First the compiler looks at each statement of the source code, and perform dependency analysis to calculate the critical path. Any independent statements can be run in parallel. A scheduling algorithm calculates which order the statements are executed, grouping statements that are dependent on each other into the same task. A compile time performance metric is used on each task to estimate the potential speedup of the parallel version over the sequential version. Due to the overhead of threads, this potential speedup would not actually be achieved. The parallelising compiler takes this into account and will only use the parallel version if it predicts it will really be faster.

This papers design takes elements from a typical parallelising compiler, dividing the task into two main stages, an analysis stage and a modification stage.

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 .autoparallelise 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: Introduce these subsubsections

1) Infinite Thread Problem: The sample programs shown in section III assume that threads have no overhead and were used a a method of describing how a sequential program could be run in parallel. In the real world, these parallelised sample programs produce an unreasonable amount of threads of which most are waiting. This causes the parallel version to perform a lot worse than sequential code.

The initial solution for too many threads is to use a thread-pool so only a fixed number of threads are executed at the same time. This solution would not work in this case as if all the threads are waiting on a future task, which is also waiting for a thread to be free then we get stuck in a deadlock. By tweaking the design of the thread-pool slightly, we can have a fixed the number of tasks and prevent this deadlock.

As the task queue is the main cause of this deadlock, it is removed from this new proposed design which will be referred to as a no-queue thread-pool. Whenever a task is created, it looks for an available thread and that thread starts executing the task in the background. If there is no thread available, then the current thread should execute the task. Also, if the the current thread is waiting for a task to return, then it could execute another task whilst waiting. **TODO: Display this if statement more visually? TODO: Explain why this design won't deadlock like the traditional thread-pool**

To modify the sample programs to use the no-queue thread-pool would be of minimal work; it would require a shared memory space (to gain access to the threads) and instead of calling `thread::spawn`, it would call another function.

2) Performance Analysis:

V. 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* 37.8, pp. 479–497.

Bondhugula, Uday et al. (2008). "Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model". In: *International Conference on Compiler Construction*, pp. 132–146.

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.

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.

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 (1992a). "Some efficient solutions to the affine scheduling problem. I. One-dimensional time". In: *International journal of parallel programming* 21.5, pp. 313–347.

– (1992b). "Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time". In: *International journal of parallel programming* 21.6, pp. 389–420.

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*, pp. 243–246.

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: *Master's Projects* 210.

Prabhu, Prakash, Ganesan Ramalingam and Kapil Vaswani (2010). "Safe programmable speculative parallelism". In: *ACM Sigplan Notices* 45.6, pp. 50–61.

Pugh, William and Evan Rosser (1997). "Iteration space slicing and its application to communication optimization". In: *Proceedings of the 11th international conference on Supercomputing*. ACM, pp. 221–228.

Quiñones, Carlos García et al. (2005). "Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices". In: *ACM Sigplan Notices* 40.6, pp. 269–279.

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 10.2, pp. 160–180.

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 Architecture and Code Optimization (TACO)* 9.4, p. 54.

Yiapanis, Paraskevas, Gavin Brown and Mikel Luján (2016). "Compiler-Driven Software Speculation for Thread-Level Parallelism". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 38.2, pp. 1–45.

Zhong, Hongtao et al. (2008). "Uncovering hidden loop level parallelism in sequential applications". In: *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pp. 290–301.