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. Many solutions exist to this problem including automatic parallelisation at the binary level, automatic parallelisation at the compiler and manually parallelising using annotations. The potential benefits of solving this problem are increased performance for existing programs, as well as making development of new software easier (as programmers do not need to worry about writing parallelised code).**

This paper focuses on automatically converting sequential source code into a parallelised program. The literature is explored and presenting into two main areas: theoretical models of automatic parallelisation and existing real-world parallelising compilers. The author presents a design for a new parallelising compiler for the Rust programming language. To demonstrate the design, three sequential elements of Rust programs are manually converted by this paper: function calls, for-loops and branches.

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 uncompiled sequential programs to take advantage of the new hardware. Developing new programs would be easier for programmers as they can just write the code sequentially and let the compiler parallelise it.

Section II examines the literature to finds that most existing solutions focus on unsafe languages such as C++ and FORTRAN. Since a significant proportion of existing programs are written in these languages, it makes sense for other authors to focus on these languages. There is a downsides to focusing on these unsafe languages, dependency analysis becomes much more difficult. To avoid this downside, the design this paper introduces in section IV uses 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 some of Rust's features 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).

In C++ (or other unsafe languages), a variable can be accessed and modified if it is in scope. In Rust, each variable is immutable by default and if a variable is passed to another function, then that function takes 'ownership' of the variable. The code below would not compile in Rust, but similar code would compile in C++. When `f` is called, it takes ownership of the variable `i`. This essentially moves it out of scope, and so it cannot be moved into `g`.

```
fn main() {
    let mut i = vec![1,2,3];
    f(i);
    g(i);
}
```

To make the code above compile, we have two options. The first option is to let `f` 'borrow' the variable instead of moving the ownership of `i` to `f`. In the example below, variable `i` is 'mutably borrowed' by `f`, and then 'immutably borrowed' by `g`. The syntax of Rust makes it explicit that `f` may modify this variable and `g` cannot. This information expressed is available to the compiler.

```
fn main() {
    let mut i = vec![1,2,3];
    f(&mut i);
    g(&i);
}
```

The second option is to clone `i` instead. This would keep the types of `f` and `g` the same as the first example. In the example below, we make a copy of `i` before moving it to `f`. Any changes that `f` makes to `i` will not be transferred to `g`, unlike the previous example.

```
fn main() {
    let mut i = vec![1,2,3];
    let j = i.clone();
    f(i);
    g(j);
}
```

Note: `i` must be cloned before moving `i` to `f`, or else we would not have access to `i` to clone it.

II. Related Work

A. Parallelisation Models

In this section, 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. It is shown how rearranging loop iterations and optimising memory access patterns for multiple threads can increase performance. The speculative parallelism subsection shows related work where the schedule is more flexible. This kind of parallelism tries to run dependent tasks in parallel and detecting when there is a conflict. When a conflict occurs, some parallel thread is 'undone' and rerun.

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 parallelised code.

An alternative 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 there is some parallelisable parts hidden in loops 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 same 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) introduce two new language constructs for C# to allow the programmer to manually specify areas of the program that can be speculatively parallelised. Yiapanis, Brown and Luján (2016) designs a parallelising compiler which can automatically take advantage of speculative parallelism.

B. Parallelisation Implementations

In this section we look at: parallelising compilers which focus on parallelising FORTRAN programs; OpenMP which is an model for shared memory programming and parallelising compilers which convert sequential CPU code into parallelised GPU code. Some of these parallelising compilers are based off of models described in subsection II-A.

Eigenmann, Hoeflinger and D. Padua (1998) manually parallelises the PERFECT benchmarks for FORTRAN which are compared with the original versions to calculate the potential speedup of an automatic parallelising compiler. D'Hollander, Zhang and Wang (1998) developed a FORTRAN transformer which reconstructs code using `GOTO` statements so that more parallelisms can be detected. It performs dependency analysis and automatically parallelised loops by splitting the task into jobs. These jobs can be split between networked machines to run more jobs concurrently. Rauchwerger and D. A. Padua (1999) introduce a new language construct for FORTRAN programs which allows for run-time speculative parallelism on for loops. Their implementation parallelises some parts of the PERFECT benchmarks which existing parallelising compilers of the time could not find.

Quiñones et al. (2005) introduce the Mitosis compiler which combines speculation with iteration space slicing. There is always only one non-speculative thread which is seen as the base execution; all other threads are speculative. The Mitosis compiler computes the probability of two iterations conflicting. If this probability is low, and there is a spare thread unit, then the loop iteration is executed in parallel. The non-speculative thread detects any conflicts as it is the only thread that can commit results.

Dagum and Menon (1998) introduces a programming interface for shared memory multiprocessors called OpenMP targeted at FORTRAN, C and C++. The programmer annotates the elements of the program that are parallelisable, which the compiler recognises and performs the optimisation. OpenMP is compared to alterative parallel programming models. Kim et al. (2000) introduces the ICP-PFC compiler for FORTRAN which uses the OpenMP model. All loops in the source code are analysed by calculated a dependency matrix. The compiler automatically adds the relevant OpenMP annotations to the loop. Lam (2011) extends OpenMP using machine learning to automate the parallelisation. The system is trained using a set containing programs already parallelised using OpenMP. The knowledge learned is applied to sequential programs to produce parallelised programs.

A CPUs architecture is typically optimised for latency whereas a GPUs architecture is typically optimised for throughput. This can make GPUs perform much better that CPUs for a certain type of task. Baskaran, Ramanujam and Sadayappan (2010) uses the affine transformation model to convert sequential C code into parallelised CUDA code. `For` loops are tiled for efficient execution on the GPU.

III. Problem Details

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 of complex code. Since memory in Rust has 'ownership', this dependency information 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.

Before we can automate the parallelisation of Rust programs, we must look at some example programs to fully understand the problem. The examples were written sequentially and then manually parallelised. The pattern of parallelisation could be applied to different examples, and a design is proposed to automate this in section IV. In these examples, threads are used as if they have no overhead, to simplify the explanation of how a sequential program could be parallelised. In reality this is not the case and so in section IV `thread::spawn` is replaced with something more optimal.

A. Parallel Function Optimisations

In some cases, the function arguments are known before the result of the function is required. When the program is sequential, the program must wait for the result of that function call. If the program was parallelised, this function could be started in the background as soon as the arguments are decided. This would allow the result to be ready for when it is requested or at least closer to ready than the pure sequential version.

Example: Algorithm 1 is a program that calculates Fibonacci numbers in a very inefficient way. The main method calculates `i` on the second line, and since it is not mutable it cannot be changed. The `slow_method` is executed before the `fibonacci` function, even though the two functions are independent. Just running the `fibonacci` function earlier would not improve performance as the `slow_method` would now have to wait for `fibonacci` function to finish. Ideally, `fibonacci` should start as soon as `i` is calculated, and at the same time `slow_method` should be executed.

Algorithm 2 shows one way of converting Algorithm 1 into a more parallelised version. These changes allow for `fibonacci_parallel` to be called as soon as `i` is decided. `fibonacci_parallel` does not block, and will return almost immediately with a `JoinHandle`. The inner thread is executed in the background, allowing `slow_method` to be executed at the same time. Inside the background thread `fibonacci_parallel` is called again which spawns another thread. These changes also allow for `n1` and `n2` to be executed in parallel. Calling `join()` on a `JoinHandle` will block until the thread is terminated, and returns the result of the thread. The `fibonacci` function is changed so that any external code that uses this function will still work (although sequentially).

Algorithm 1 Sequential Fibonacci Function

```
fn main() {
    // Set i = first program argument
    let program_args = std::env::args();
    let i = program_args[0].trim().parse::<u32>();
    slow_method();
    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() {
    // Set i = first program argument
    let program_args = std::env::args();
    let i = program_args[0].trim().parse::<u32>();
    let fib = fibonacci_parallel(i);
    slow_method();
    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

Loop iterations that are independent of each other could be run at the same time to increase performance. It is not always clear how many iteration of a loop may occur, especially while-loops. For-loops are more explicit about how many iterations will occur, but they can still terminate early with a `break` or a `return` statement.

Example: Algorithm 3 is a real world example with a two level for-loop combined with an if-statement which returns the password for the first valid hash. The inner for-loop is

not parallelisable as each iteration reads and modifies the `hash_word` variable. The outer for-loop is parallelisable, but if we were to naively put the contents into separate threads, as shown in Algorithm 4, then we may end up with the wrong result. In the sequential version, the first password in the list to match the hash would be returned but in the naive parallel version, the password returned depends on the order the threads are run. In this example program at most only one iteration could possible enter the if-branch, terminating the for-loop early, but in other examples there could be more than one. Algorithm 5 is the final complete parallelisation of Algorithm 3. Each word is hashed in it's own thread, compared to `hash_password` and returned to be stored in the initial thread. The function does not return a result until all previous iteration threads have also returned.

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 hash_word = word;
        for _ in 0..40 {
            let mut sha = Sha256::new();
            sha.input_str(word);
            hash_word = sha.result_str();
        }
        // Check if hash matches
        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 also been run in sequential normally. This optimisation more speculative than the other optimisations described as some threads are going to be disregarded. Each side of the branch for an if-statement could be run in parallel whilst waiting for the condition. When the condition is finally calculated, the result of the correct branch is kept. Algorithm 7 shows this optimisation applied to Algorithm 6. Ideally, when the condition is calculated and the incorrect branch is still being calculated in a thread, then this thread should be cancelled. This is not easily achievable in Rust so this optimisation is left as a possible extension of the design.

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 performs 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

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 || {
            let result = {
                // Hash word using Sha256
                let mut hash_word = word;
                for _ in 0..40 {
                    let mut sha = Sha256::new();
                    sha.input_str(word);
                    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
}
```

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. The analysis stage is run by a linter plugin and the modification stage is run by a syntax extension plugin. Analysis stage must come before the modification stage, so compiling must be done twice (once for each stage). When the plugin is loaded, it determines which stage it is by looking for a shared JSON file. If this file does not exist, then it is the analysis stage, otherwise the file's content is loaded 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

Algorithm 5 Parallel Password Cracker

```
fn crack_password(dictionary: &Vec<String>,
                  hash_password: String)
                  -> Option<String> {
    let (tx, rx) = mpsc::channel();
    for i in 0..dictionary.len() {
        let word = dictionary[i].clone();
        let tx = tx.clone();
        thread::spawn(move || {
            let result = {
                // Hash word using Sha256
                let mut hash_word = word;
                for _ in 0..40 {
                    let mut sha = Sha256::new();
                    sha.input_str(word);
                    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
    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 6 Sequential Slow If

```
fn f(a: u32, b: u32) -> u32 {
    if slow_condition(a, b) {
        (a * (b - a)) + 5
    } else {
        ((a * b) + 19) ^ 2
    }
}
```

parallel. The linter plugin would use this dependency tree to determine which parts should be parallelised, and save this information to a file.

TODO: Expand

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 modifications applied would be similar to the sample programs shown in section III. Those modifications assumed that threads have no overhead and were used as 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 and most of the threads are just waiting for other threads to return. This synchronisation can cause the parallel version to perform a lot worse than sequential code. The following subsections describe some of the problems with the parallelised sample programs, and provides some solutions.

Algorithm 7 Parallel Slow If

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

1) Infinite Thread Problem: 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.

TODO: Replace this with the new-new design?

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.

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: Parallelising small tasks, or tasks that would require a lot of synchronisation when run in parallel actually run significantly slower than the sequential

code. In these cases, the code should not be parallelised. There is a problem though, how does the compiler know if it is faster parallelised or sequential.

One possible solution is to run the sequential version in one thread, and the parallel version in the other threads. Whichever version finishes first is used, and the other version is cancelled. This approach leaves the performance approach until run-time, but it is a very inefficient solution as the parallel version now has fewer threads.

Another solution is to use some sort of performance metric at compile time to estimate which version will be faster. If we assume each statement takes one unit of execution time, we can attempt to calculate how many time units the sequential and parallel version will take. Since we know that there is an overhead to the parallel version, we should add on some extra time units. The faster version would be the version that is compiled. This solution would not really work as described as the execution time may be dependent on how many iterations of a loop (which isn't necessarily known at compile time). The parallel version is also dependent on the number of cores the machine has, also not known at compile time.

The best solution is a combination of the previous two ideas. The compiler should create an function which estimates both the sequential and the parallel version execution time. This function can take in any run-time conditions i.e. the number of iterations of a loop or the number of cores the machine has. This function should be fast such that the overhead of calculating which version is faster should be negligible. At run-time, this function predicts the faster version using the run-time variables, and the faster version is the only version run.

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 precomputation 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).

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.