Automatic Parallelisation of Rust Programs at Compile Time

Author

Michael Oultram

1428105

Supervisor

Dr Ian Batten

Degree

MSci in Computer Science

September 2014 – June 2018

Institution

School of Computer Science

University of Birmingham

Chapter 1

Preamble

1.1 Abstract

Processors have been gaining more multi-core performance which sequential code cannot take advantage of. Many solutions exist to this problem in the literature including automatic parallelisation at the binary level, at the compiler level and manually annotating parallelisable parts of source code. Solving this problem has many potential benefits: increased performance for existing programs and development of new software that does not need to be parallelised manually by the programmer.

The author introduces a new design for a parallelising compiler for the Rust programming language. The design focuses on safe statement level parallelism using a dependency analysis stage and a scheduling stage. The dependency analysis stage looks at each function individually for statements that use the same variables. The scheduling algorithm takes the dependency tree and tries to run as much as possible in separate threads using a "maximum spanning tree" approach for any statement with multiple dependencies.

The design chosen was implemented using the plugin system for the Rust compiler. Several adaptations were made to the design due to unforeseen complexities with the compiler. The end result was evaluated by testing in detail two example sequential programs as well as testing robustness using randomly generated sequential programs. These tests shows the potential parallelisability of the programs, and one example produces a faster program runtime. Most of the programs tested have a slower runtime, and sometimes the parallel code produced does not even compile. The problems with the implementation are discussed, and how changes to the design could end up with a more successful solution.

All software for this project can be found at
`https://github.com/MichaelOultram/Auto_Parallelise`

1.2 Contents

1 Preamble **2**
1.1 Abstract . . . 2
1.2 Contents . . . 3

2 Introduction **4**
2.1 Related Work . . . 4
2.1.1 Parallelisation Models . . . 4
2.1.2 Parallelisation Implementations . . . 5
2.2 Rust Programming Language . . . 6
2.2.1 Ownership . . . 6
2.2.2 Mutability . . . 7
2.2.3 Unsafe Blocks . . . 7
2.2.4 Threads . . . 8

3 Design **9**
3.1 Analysis Stage . . . 9
3.2 Modification Stage . . . 10

4 Implementation **11**
4.1 How to Implement the Design . . . 11
4.2 Linter Plugin . . . 12
4.3 Syntax Extension Plugin . . . 12

5 Evaluation **16**
5.1 Sequential Programs Parallelised . . . 16
5.1.1 Simple Example . . . 16
5.1.2 Password Cracker . . . 17
5.1.3 Automatically Generated Sequential Programs . . . 20
5.2 Results . . . 22
5.2.1 Simple Example . . . 22
5.2.2 Password Cracker . . . 22
5.2.3 Automatically Generated Sequential Programs . . . 23

6 Discussion **29**
6.1 Achievements . . . 29
6.2 Improvements . . . 29
6.3 Approach Changes . . . 30

7 Conclusion **32**

8 References **33**

9 Appendix **34**
9.1 Running the Code . . . 34
9.2 Rust Compiler Types . . . 34

Chapter 2

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. Ideally we want a simpler way to convert the easier to write (and existing) sequential code into parallelised code that can take advantage of the additional cores. The problem is well established in computer science and has many solutions. The solution that this report focuses on is using a parallelising compiler to convert sequential source code into a parallelised binary. Research done in this field has mostly focused on the C++ language although other researchers have had success using other languages. Some methods require manual annotation of the source code by the programmer to specify which parts of the program are parallelisable whilst others have attempted to automatically detect these areas, but with an unsafe language like C++ this is challenging.

2.1 Related Work

2.1.1 Parallelisation Models

This section looks 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 threads are 'undone' and rerun.

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. 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) use 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) show how this information can be used to transform loops on example

programs to produce a real world speedup. Beletska et al. (2011) show that iteration space slicing extracts more coarse-grained parallelism than affine scheduling.

Speculative parallelism

Zhong et al. (2008) show 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) designed a parallelising compiler which can automatically take advantage of speculative parallelism.

2.1.2 Parallelisation Implementations

This sections looks at: parallelising compilers which focus on parallelising FORTRAN programs; OpenMP which is a model for shared memory programming and parallelising compilers which converts sequential CPU code into parallelised GPU code. Some of these parallelising compilers are based on models described in subsection 2.1.1.

Eigenmann, Hoeflinger and Padua (1998) manually parallelise 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 created parallel loops by splitting the task into jobs. These jobs can be split between networked machines to run more jobs concurrently. Rauchwerger and 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 certain types 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.

2.2 Rust Programming Language

Rust is a fairly new systems programming language with a focus on thread safety. Variables have an ownership and a lifetime which allows only one block to access the variable at once. This is guaranteed at compile time and should make the process of automatically detecting dependencies much easier than some of the approaches described in section 2.1. To test this hypothesis, the author proposes a design for a new paralleling compiler using the Rust programming language. The Rust nightly compiler has a feature which allows for plugins to access the internals of the compiler including modification access of the abstract syntax tree. This feature allows the author to implement their design.

The Rust programming language is similar to C++ but it has some specific features that may not be known to the reader. This section briefly explains features of the language that are used in later sections of the report. If the reader requires more in depth understanding than what is provided, then they should look at the language documentation, *The Rust Book* (2017).

2.2.1 Ownership

In Rust, all variables have an ownership. Only one block can have access to that variable at a time. This is enforced at compile time.

```
1  fn main() {
2      let a = vec![10];
3      f(&a);
4      g(a.clone());
5      h(a);
6      // Cannot access `a` here anymore
7  }
8
9  // 'f' borrows 'a'
10 fn f(a: &Vec<u32>){}
11
12 // 'g' takes owmership of a copy of `a`
13 fn g(a: Vec<u32>){}
14
15 // 'h' takes owmership of `a`
16 fn h(a: Vec<u32>){}
```

Listing 1: Borrowing and moving example

In this example, `a` is a local variable in the `main` method. When `f` is called with parameter `a`, the function borrows that variable. This is similar to call-by-reference from other programming languages. When `h` is called with parameter `a`, the variable is moved to `h`. This is unlike other programming languages as this is not call-by-value. Instead `h` takes ownership of `a`. When `h` is returned, the main method can no longer use `a`. The `g` function also wants to take ownership of the variable `a`, but we want the `main` method to keep the ownership. To do this, `a` is cloned and this copy is moved into `g`. If the type of `a` implemented the `Copy` trait, then the compiler would automatically clone the variable instead of moving, allowing access to `a` after the `h` function call.

2.2.2 Mutability

Variables mutability is declared when the variable is declared. In Rust, variables are immutable by default, but if specified they are mutable. When a variable is borrowed, it can either be immutably borrowed or mutably borrowed (if the variable itself is mutable).

```
fn main() {
    let a = 10;
    let mut b = 20;
    let c = f(&a, &b);
    assert!(a == 10 && b == 20 && c = 30);
    g(&mut b);
    assert!(a == 10 && b == 21 && c = 30);
}

// 'f' immutably borrows 'a' and 'b'
fn f(a: &u32, b: &u32) -> u32 {
    a + b // No semicolon means (a + b) is returned
}

// 'g' mutably borrows 'b'
fn g(b: &mut u32) {
    b += 1;
}
```

Listing 2: Immutable and mutable borrowing

In the `main` method of this example, `a` is an immutable local variable and `b` is a mutable local variable. The `f` function borrows both `a` and `b` immutably. Even though `b` is declared as mutable, it cannot be changed inside `f`. Once `f` returns, `b` becomes mutable again inside the `main` method. The `g` function shows how `b` can be borrowed mutably. Any changes to `b` inside `g` would be reflected in the `main` method as expected.

2.2.3 Unsafe Blocks

Some features required of a systems programming language are not safe. Rust allows the programmer to turn off some of Rust's safety features by using an unsafe block. The most common use of an unsafe block is to modify a mutable static variable but it also allows de-referencing of a raw pointer and calling unsafe functions (i.e. an external c function). Using unsafe blocks may introduce race conditions as two threads could try to modify a global at the same time, and the Rust language would not guarantee an order.

```
static mut global: u32 = 3;
fn main() {
    let a = global;
    assert!(global == 3);
    inc_global();
    assert!(global == 4);
    unsafe {
        global = 5;
    }
    assert!(global == 5);
}

unsafe fn inc_global() {
    global += 1;
}
```

Listing 3: An example using an unsafe block and function to modify a static variable

2.2.4 Threads

Due to ownership model in Rust, only one thread can have access to a variable safely at once. When the thread is spawned, all variables referenced are moved into the thread. A channels are used to communicate between spawned threads and can move a variable from one thread to another. For the variable to be sent via a channel, it must implement the `Send` trait.

```
fn main() {
    // Create a channel
    let (sender, receiver) = ::std::sync::mpsc::channel();
    // Spawn a thread
    let join_handle = ::std::thread::spawn(move || {
        let a = receiver.recv().unwrap();
        2 * a
    });
    // Send 10 down the channel
    sender.send(10);
    // Wait for the thread to end, and get its return value
    let b = join_handle.join().unwrap();
    assert!(b == 20);
}
```

Listing 4: An example using threads and a channel

In this example, a channel is created with a `sender` and `receiver` variable. A new thread is spawned which takes ownership of the `receiver` variable. This thread receives a value `a`, and returns $2 \times$ `a`. The `main` method uses the `sender` variable to send 10 down the channel. It then uses the `join_handle` to wait for the thread to return.

Chapter 3

Design

This chapter covers the original design concepts of my parallelising compiler without realising the full internal details of the Rust compiler. Once I began developing my design, I made adaptations (described in chapter 4) as some elements ended up being more challenging to implement inside the Rust compiler than expected. Before describing the design of my parallelising compiler, we should first look at the structure of the Rust abstract syntax tree. It consists of three main types: blocks, statements and expressions. A block contains a list of statements and a statement is a combination of expressions. Ifs and loops are represented as expressions which can contain a block within itself.

My parallelising compiler is designed to find safe statement level parallelisation within a given block. It is split into two main stages, the analysis stage and the modification stage, each of which are split into several steps. The analysis stage looks at each function in the source code and tries to find parts that could be parallelised. The modification stage takes the parts that can be parallelised and changes the source code so that they run in parallel.

3.1 Analysis Stage

Algorithm 1 Dependency Analysis Algorithm

Require: *block* as a list of statements

```
1:  envs = {}
2:  deps = {}
3:  block_env = {}                          ▷ Block's environment, for unmet statement dependencies
4:  for stmt in block do
5:      stmt_env = Set of variables that stmt uses
6:      envs.push(stmt_env)
7:      stmt_deps = {}                      ▷ Set of array ids representing dependent statements
8:      for i = length(stmt_env) −1 to 0 do ▷ Search backwards for vars in stmt_env
9:          dep_env = envs[i]
10:         for all var where (var is in stmt_env) and (var is in dep_env) do
11:             stmt_deps.push(i)
12:             stmt_env.remove(var)
13:     block_env.push(stmt_env)            ▷ Unmatched variables are part of the block's environment
```

Each statement in a block is analysed individually to provide a set of variables that the statement accesses. This set describes the variable dependencies that the statement has, but it does not describe which statements must be executed before the current one for the program to remain correct. To get this information, the algorithm looks backwards from the current statement for each variable that the current statement requires. The first statement found containing the variable in its environment is added as a statement dependency. Any variables that were not found above the current statement must be defined outside the current block, and so are added as the current blocks environment. This algorithm is shown in Algorithm 1.

3.2 Modification Stage

The dependency tree provided by the analysis stage shows what statements can be run in parallel. Some of these statements have multiple dependencies, all of which must be met before the statement is run. Each of these dependencies could be in a separate thread, and so some synchronisation technique is needed. The dependency tree is converted into a schedule tree so that we know which statements are run in which threads, and where/when synchronisation is required between threads.

The scheduling algorithm got its inspiration from minimum spanning trees. We make the naive assumption that threads have no overhead and so we want to run as much as possible in separate threads. Whenever synchronisation is required, we want to reduce the amount of time that a thread has to wait. If the thread that requires a dependency is slower than the thread that releases the dependency, then no waiting is required. This is different than other minimum spanning tree algorithms and so I refer to it as a "maximum spanning tree" algorithm.

Algorithm 2 Scheduling Algorithm

Require: *block* as a list of statements
Require: *envs* as a list statement environments
Require: *deps* as a list statement dependencies

```
1:  schedule_trees = []
2:  for i = 0 in length(block) - 1 do          ▷ Add all independent statements to separate trees
3:      if length(deps[i]) == 0 then
4:          schedule_trees.push(("Run", i, []))
5:  while not all Statements from block are in schedule_trees do
6:      for stmtid = 0 to length(blocks) - 1 where block[stmtid] is not in schedule_trees do
7:          if all deps[stmtid] are in schedule_trees then
8:              dep_trees = find all ("Run", i, _) in schedule_trees for all i in deps[stmtid]
9:              sort descending dep_trees and deps[stmtid] by depth in schedule_trees
10:             for i = 1 to length(dep_trees) do
11:                 (_, _, subtrees) = dep_trees[i]
12:                 subtrees.push(("SyncTo", stmtid))
13:             (_, _, subtrees) = deps[0]
14:             subtrees.push(("SyncFrom", deps[stmtid][1 :], [("Run", stmtid, [])]))
```

The algorithm starts by looking for any statements with no dependencies. Each of these statements is run in a separate thread. For all the remaining statements, the algorithm selects the set of statements that already have all their dependencies in the schedule. If the statement only has one dependency, then that statement is added directly after that dependency. If the statement has more than one dependency, then the algorithm looks to see which dependency has the longest chain. The thought behind this is that this dependency should be the slowest, and so all the other dependencies should have been finished by this point. To make sure that there are no race conditions, a syncline is introduced from the other dependencies to just before the current statement. This algorithm is repeated until all the statements are in the schedule. Since there cannot be any cyclic dependencies due to the way that the dependency analysis algorithm was designed, this is guaranteed to terminate. This algorithm is shown in Algorithm 2.

Chapter 4

Implementation

This chapter looks at how the design was implemented in practice and the design decisions that had to be adapted due to unforeseen complexities. Each design change is justified with an example of why the original design fails, and alternatives that were considered.

4.1 How to Implement the Design

There were three choices on how I could implement the design: directly modifying the Rust compiler source code and recompiling the compiler; using the Rust compiler plugin system to modify the abstract syntax tree (AST); writing a source to source translation from scratch. Modifying the Rust compiler would give me the flexibility to change any part of the compiler that I needed but it would make seeing my individual contributions very difficult. Also, as the compiler itself is very large and complex, it would take a while to compile from a clean state. Using a Rust compiler plugin would give me less access, but I would only need to compile my plugin. This option still has the downside of dealing with the complex compiler. Writing a source to source translation system from scratch would allow me to avoid touching the Rust compiler and it's complexity. In return, I would have to write code to extract the AST from a source file. I would have to model the ownership/borrowing information to detect when parallelising is possible. From all these choices, I decided to write a Rust compiler plugin as it provides all the ownership/borrowing information. This option requires me to use the full AST whereas the other options had the possibility of using the less verbose high-level intermediate representation (HIR).

The Rust compiler allows for plugins of different types. The two types of plugins of interest are Syntax Extension plugins and Linter plugins. Syntax Extension plugins are executed first in the Rust compiler pipeline, and are generally used to convert macros into code. Linter plugins are run at a later stage, and are generally used to check code style to produce warnings (like unused variable). It has the complete AST of the code with all the macros expanded. All the information required about dependencies is accessible inside a linter plugin. However, once the Rust compiler gets to the linter plugins, the AST can no longer be modified. Syntax extension plugins allow modification of the AST, but they do not have macros expanded. Some dependencies could be missed by trying to analyse the AST at this stage. The solution chosen was to compile the program twice. On the first compile, the syntax extension does nothing and the linter plugin examines the expanded code. The dependency information gathered is saved into a file for the next compile. On the second compile, the syntax extension plugin reads the file to get all the dependency information. Any parallelisable parts are then modified to be run in parallel.

To use the Rust plugin system, each function of the sequential source code should be annotated with `#[autoparallelise]`. This allows the plugins to access that element in the AST. The compiler plugin also needs to be added as a dependency in the `Cargo.toml` file and the main file needs `#![feature(plugin)]` and `#![plugin(auto_parallelise)]` at the top. These annotations do not provide any information about the parallelisability of the source code and are purely just a workaround for the Rust plugin system. This is further explained in section 9.1.

4.2 Linter Plugin

In the design section of the report, the dependency analysis algorithm takes in a block and examines it statement by statement to extract out the variables that are used. Actually doing this in the compiler plugin required a lot more effort than it seemed on the face of it. The compiler calls a function in the plugin for each annotated function. To get the `Block` struct out of this requires expanding the `FnKind` enum. A `Block` contains a list of statements (`Stmt`). The `Stmt` struct contains a `StmtKind` enum. This describes what kind of statement it is, either a local variable binding, an unexpanded macro, or an expression ending with or without a semicolon. For the variable binding, a `Local` struct is given which contains a `Path` representing the variable name. Some variables will be assigned a value at creation and this is represented as an `Expr` struct. Unexpanded macros contain some other types which represent the arguments to the macro, but I did not end up going any deeper into this as the macros will be expanded by this point in the compiler. Statements which are expression with or without a semicolon will give an `Expr` struct. Similar to the `Stmt` struct, the `Expr` struct contains a `ExprKind` enum which represents the 39 different types of expressions. Most of these options contain another `Expr`, but some contain other types like a `Pat` which is the second way a variable can be represented. Almost all of the different cases in `ExprKind` had to be dealt with to explore the tree fully and extract all the variables a statement uses (known as its environment).

Each statement is converted into our representation of the AST so that information can be stored about the dependencies. The original design used two types, an expr to represent a single statement and a block to represent a list of exprs and blocks. This design does not deal a `ExprKind` which contains a `Block` properly (i.e. a for loop). My first idea was to split the `Stmt` into an expr and a block and set the block to depend on the expr. This seemed to work for the dependency analysis stage, but not for the reconstruction stage. It was difficult to distinguish which block is the contents of the `ExprKind` if an external block was also dependent on the expr. I created a third type called exprblock which contains one expr and a list of blocks (as an if-else expression will require more than one block). Once all the statements for a function are converted into our representation, and each statement had an environment, then the environments could be matched up as described in Algorithm 1. The dependencies IDs are stored as array indexes relative to the block.

I realised that just using one environment had a flaw which adds an extra unneeded dependency. If there are two let statements which have the same variable name, the second let statement will be dependent on the first which is not needed. I fixed this by using two environments, one for the variables that the statement depends on, and another environment which lists the variables that the statement releases/produces. As described in section 2.2, a variable's ownership can be moved, and this can be represented by adding it to the requirements environments and not including it in the releases environment. In practice, I used the two environments, but I did not check whether a variable gets moved or not. If the source code provided to my parallelising plugin is correct sequential code, then the variable will not be mentioned in a statement after the point it has been moved.

4.3 Syntax Extension Plugin

The original design did not require two separate compiles. To get the dependency information from the Linter plugin into the Syntax Extension plugin, it was saved to a JSON file using the serde_json library. I could not store the raw AST with my added dependency information as both the JSON library and the Rust compiler are separate crates which I cannot edit. To combat this, I converted the dependency tree into an encoded dependency tree. Each `Stmt` and

`Block` would be represented by a number (a statement id). These structs already have a `NodeId` element which was tried first. Unfortunately, this id was not consistent between compiles. Both of these structs also had a `Span` element which represents the bytes in the source code that the `Stmt` or `Block` represents. Since the source code would not be changed between compiles, this should remain consistent. My statement id became two numbers, extracted from the AST by accessing `span.lo().0` and `span.hi().0`.

When the Syntax Extension plugin detects the JSON file, it loads the encoded dependency information. There is no easy way to decode the dependency information, so I ran the dependency analysis algorithm again on the AST that the Syntax Extension plugin has access to. This time, macros will not be expanded, and so I had to add a new Mac type to represent this. I did not try to examine the `Macro` as I had the encoded dependency information. I copied the encoded dependency information into the dependency tree with unexpanded macros by matching up the statement ids. Later I realised that I should be replacing the dependency information with the encoded dependency information instead of trying to merge it. The reason for this is because the dependency tree with unexpanded macros would have no environment to start off with. When a statement past the macro tries to match up its environment, it will skip checking the macro and match with something above. When the dependencies were merged, the statement would end up depending on two different statements for the same variable (the macro from the expanded dependency tree and then the statement above).

The scheduling algorithm from the design section did not require many changes at all. The only real addition I had to add was an environment to the syncline. This allows the reconstructor to know which variables it should send down the channel, and what variables will be received in the new thread. To do this our representation of the AST needs to be adjusted to store the environment along with the dependency ids for each statement. By storing the environment of each statement, the scheduling algorithm can look at both the releasing and requiring statements to work out which variables should be sent.

Once a schedule has been created, it needs to be turned back into code. First I gathered all of the synclines that are in the schedule, and I created a channel for each of them. The channel name was based on the statement ids of the two points being synced, and the variable names that will be sent down the channel. Each tree in the list of the schedule was fully explored before starting the next tree. They are all given a separate thread, except the last one which uses the current thread. The Rust compiler has a few macros for creating new statements which works well for some static statements, and even some statements where a variable name is changed. But in some cases I had to unwrap the entire statement to change one part and then recreate it all the way up again. This was quite time consuming as some of the functions or structs I wanted to use were private in the compiler so I had to delve deeper to see what I could make, and work from that. When I managed to create the types, and reconstructed all the code, the compiler would not accept my changes. All the error messages it gave me used the original source code, and were not very helpful. I got the compiler to print out the changes I made, as code. I copied this into a separate folder and it compiled just fine. Searching the issues on the Rust compiler GitHub, I came across #46489 which seemed to relate to my issue. Since that issue was not fixed by the time I had to demonstrate my project, I ended up writing a bash script to copy the modified code into a separate file and compile for a third time.

When examining the parallelised source code, I noticed that the incorrect value was returned from a function/block in some cases. Most of the time, when this occurred, the return type would be incorrect and so the parallel code would not compile. The reconstructor puts all the tree elements into separate threads except the last tree which is executed in the main thread. Before the function exits, it joins all of the threads that were created. If nothing more was done, then the return value would be whatever the second to last thread returns. I changed this by

storing the return value of the last tree element into a variable and setting the return value of the function to that. Using this method makes the parallelised source code a lot harder to read, but since it does not need to be human readable, I regard this as acceptable. This works most of the time as the last element in the schedule is usually the correct return value, but not always. To make it work more consistently, I reordered the trees in descending order by the largest statement id value in the tree. The idea behind this is that the return value would be towards the end of the file. Ideally, the dependency analysis stage should be changed to find the return values. This would also remove the need to join all the threads at the end of the function as most of the threads would be guaranteed to be terminated by synclines.

As shown in the literature (section 2.1), some loops can be parallelised to get an increase in speed. Many of the approaches involved reordering nested loops to optimise memory access patterns. I did not follow this approach as this type of parallelism was added very late into my project and so I increased the restrictions to try to get it to work in time. I only attempt to parallelise for loops that iterate over a range of numbers but given more time this could be expanded to more types of loops. Loops that exit early using a break or a return statement are not supported. The scheduler looks at the inner block of the for loop for any external dependencies. These need to be passed between iterations using a separate syncline for each variable per iteration. To minimise the critical section in each iteration, the algorithm searches for the first and last time that the variable is used in the block. The first occurrence receives the variable from the last iteration and the last occurrence sends the variable to the next iteration. If one external variable is received at the start of the block and sent at the end of the block then there is no point in parallelising the loop. The reconstructor places each iteration of the for loop in a separate thread. A new syncline is created in the main thread to send the external dependencies to the first iteration, and another is used to receive the external dependencies back from the last iteration. The external dependencies are sent after the loop has spawned all of the threads, and it immediately waits for the dependencies to be returned. This approach works for some examples, but as shown in chapter 5, it has many flaws.

`for loops`

Chapter 5

Evaluation

Evaluating the success of my solution is difficult. My implemented solution only works for some examples, as only some features of the language are supported. Its focus is on parallelising as much as possible, not necessarily on getting a faster program. To get an idea of how much parallelism was found, the parallel source code was statically examined to find out how many threads and channels are created. These values may be lower than the actual number of threads used, as some thread definitions will be reused (repeated function call, `for loops`, etc.). Each program was compiled sequentially and in parallel, with and without optimisations.

Two manually written sequential programs were parallelised: a simple example and a password cracker. The dependency tree and schedule are analysed to show how the parallelising compiler works. To test the robustness of the compiler, a collection of automatically generated sequential programs were also parallelised. The output of the parallel version was compared with the sequential version to check that the parallelisms have not semantically changed the program.

5.1 Sequential Programs Parallelised

5.1.1 Simple Example

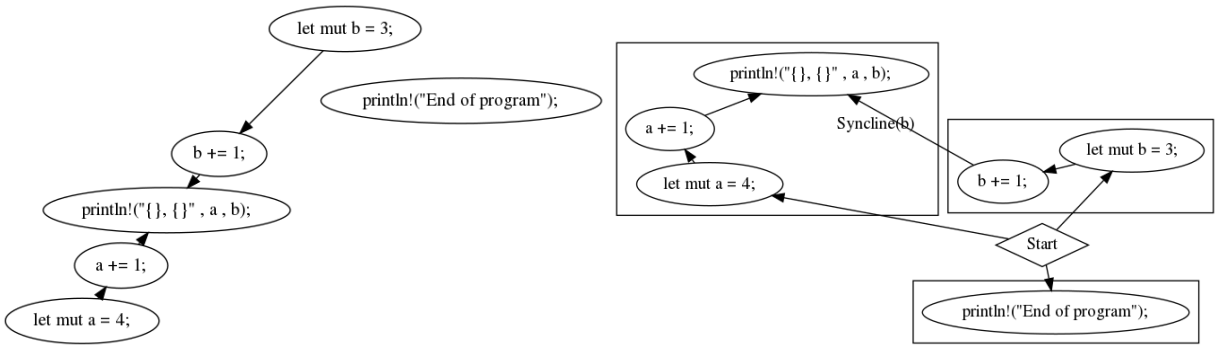
This simple example is intended to be an easy way to show how the parallelising compiler works. It consists of two independent variables, `a` and `b` which are incremented and then printed together. An extra independent print statement is also added at the end to show one potential problem with the dependency analysis.

simple-example/main.rs

```
1  #![feature(plugin)]
2  #![plugin(auto_parallelise())]
3
4  #[autoparallelise]
5  fn main() {
6      let mut a = 4;
7      let mut b = 3;
8      a += 1;
9      b += 1;
10     println!("{}, {}", a, b);
11     println!("End of program");
12 }
```

Listing 5: A simple example program

Figure 5.1a clearly shows `a` and `b` do not need to be inter-weaved. It also shows that the `println!(‘‘End of program");` is completely independent of the other print statement. This may not be intended by the programmer, but there is nothing inside the `println!` macro which states that they must be called in order (there is no variable linking the two print statements). This could be fixed by forcing non-analysed functions to be executed in order, but it may not be an actual dependency and it would reduce the number of parallelisms detected. Because this dependency is not detected, I sorted the outputs of both sequential and parallel programs so that the order does not matter when comparing program outputs. Figure 5.1b shows that



(a) Dependency analysis (b) Schedule

Figure 5.1: Main method from the simple example program

the scheduling algorithm split `a` and `b` into separate threads as well as the "End of program" print statement. The print statement that prints `a` and `b` attached itself to the `a` thread. The `b` variable is sent along the syncline just before the print.

5.1.2 Password Cracker

A password cracker program is a good target for paralleling. Each word in a dictionary is hashed individually and then compared with the hash of a password that the user is trying to crack. My implementation contains three functions: `loadDictionary()`, `hash()` and `main()`. The `loadDictionary()` is not analysed here as the parallelising compiler did not make any changes to this function. This program is on the limits of what my parallelising compiler can do in its current state. Some of the code is written in a semi-specific way to get it to parallelise properly.

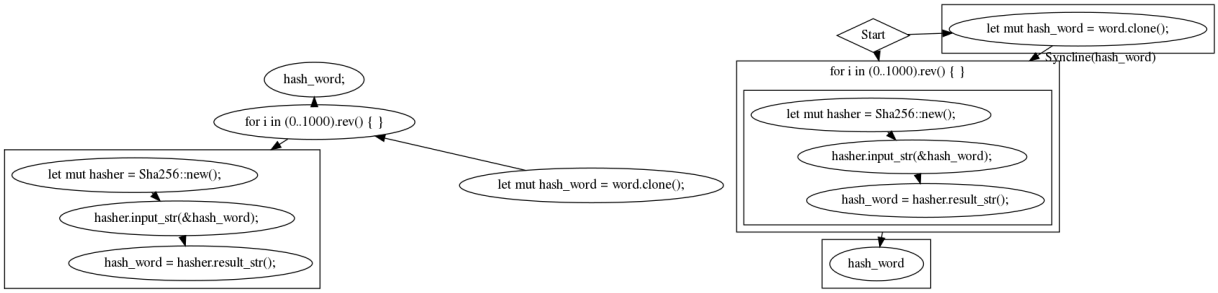
Hash Function

password-cracker/main.rs

```
27  #[autoparallelise]
28  fn hash(word: &String) -> String {
29      let mut hash_word = word.clone();
30      // Hash word using Sha256
31      for i in (0..1000).rev() {
32          let mut hasher = Sha256::new();
33          hasher.input_str(&hash_word);
34          hash_word = hasher.result_str();
35      }
36      hash_word
37  }
```

Listing 6: Hash function of the password cracker program

The word argument is iteratively hashed 1000 times with SHA256 from an external crate. If I let the parallelising compiler try to parallelise this loop, it would produce code that does not compile. `hasher.result_str()` returns a `str` which is normally converted to a `String`. When run in parallel, it tries to send a `str` down a syncline which does not work as `str` does not implement the `Send` trait. This could be fixed by storing the type with the variable name instead of relying on type inference. Another reason for disabling parallelisation of this loop is that it would slow down the program. Most of the statements depend on the previous iteration and so no speedup can be achieved here. Line 32 has no dependencies and could be run in parallel but it is not computationally expensive to call `Sha256::new()` so it is not beneficial. If some performance analysis technique was implemented, then it would notice that it is a bad idea to parallelise this loop. To disable parallelisation of this loop, I added `.rev()` to the range of



(a) Dependency analysis (b) Schedule

Figure 5.2: Hash function from the password cracker program

the loop. This is very hacky, but it breaks one of the extra restrictions I placed on parallelising `for loops` so it will not be parallelised.

Figure 5.2a shows how the inner block is linear for one iteration. The reason that the loop cannot be parallelised effectively is that `hash_word` is modified each iteration. Figure 5.2b shows how linear the iterative hash function is. The only separate thread that is spawned clones the word. The reason that the `for loop` is not inside the same thread is because the `hash_word` dependency is inside the `for loop`. The condition on the `for loop` is independent and could be slow and so is parallelised. In reality, parallelising this loop will be much slower due to thread overhead.

Main Method

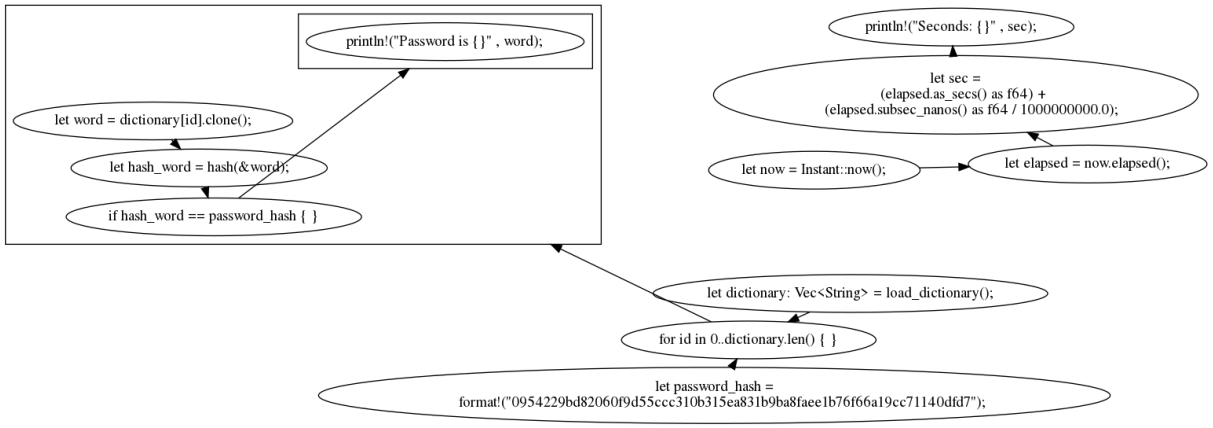
The main method loads the dictionary from a file into a list of strings. Each word in this list is hashed and compared with a password hash. Notice that the `for loop` is not interrupted when it finds the correct word, or even stored; it is just printed. A timing circuit is placed around the function to print how long it takes to find the password.

password-cracker/main.rs

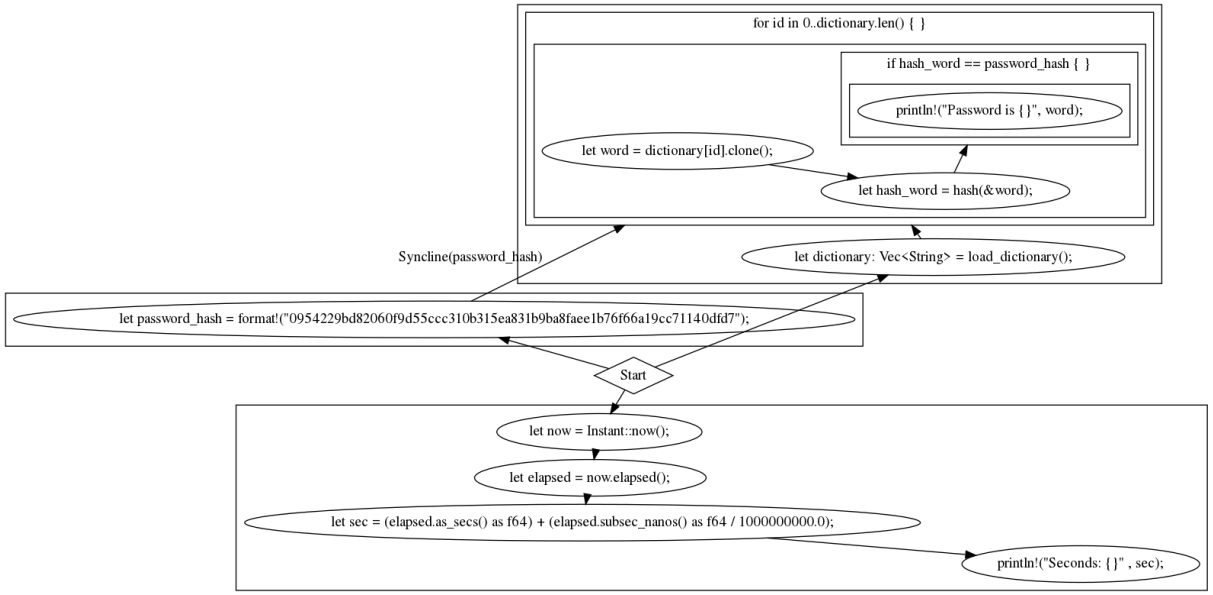
```
39  #[autoparallelise]
40  fn main() {
41      let now = Instant::now();
42
43      let dictionary: Vec<String> = load_dictionary();
44      let password_hash =
     ↪   format!("0954229bd82060f9d55ccc310b315ea831b9ba8faee1b76f66a19cc71140dfd7");
45
46      for id in 0..dictionary.len() {
47          let word = dictionary[id].clone();
48          let hash_word = hash(&word);
49          if hash_word == password_hash {
50              println!("Password is {}", word);
51          }
52      }
53
54      let elapsed = now.elapsed();
55      let sec = (elapsed.as_secs() as f64) + (elapsed.subsec_nanos() as f64 / 1000_000_000.0);
56      println!("Seconds: {}", sec);
57  }
```

Listing 7: Main method of the password cracker program

As you can see from the Figure 5.3a and Figure 5.3b, the timing circuit is a completely independent program. This is probably not intended, and it is a similar problem to the `println!` problem from the simple example. This is a slight difference however, as the order for the time circuit is actually correct but the time that the timing circuit is run is different. There is no real solution I can think of for this issue except hard coding the time crate to be dependent on everything before it and everything after it dependent on it. Hard coding this may be problematic as there



(a) Dependency analysis



(b) Schedule

Figure 5.3: Main method from the password cracker program

may be other functions in other crates which also have similar dependency requirements. I will just remove the timing circuit when comparing the outputs as the time will not be consistent between runs. The schedule (Figure 5.3b) shows that the `password_hash` assignment is placed in its own thread, as the `for loop` condition is independent. The block inside the `for loop` does depend on this value, and so a syncline is setup. It is not clear from the schedule above, but the `for loop` is also parallelised. Figure 5.4 shows how the each iteration of the `for loop` is parallelised and what synclines are required. The `dictionary` and `password_hash` variables are requested on the first line they are required in the iteration and sent to the next iteration when they are no longer needed. In this case, only one statement uses each external dependency, so it is immediately released. The `hash` function is computationally expensive, and can only be started once it has access to its word in the dictionary. The dictionary can be passed very quickly through all of the threads, as accessing the element inside a list is very fast. This allows multiple `hash` functions to be run at the same time. The `password_hash` variable is requested after the `hash` function returns, meaning `id=1` cannot check whether it is equal to the `password_hash` if its `hash` function returns before `id=0`.

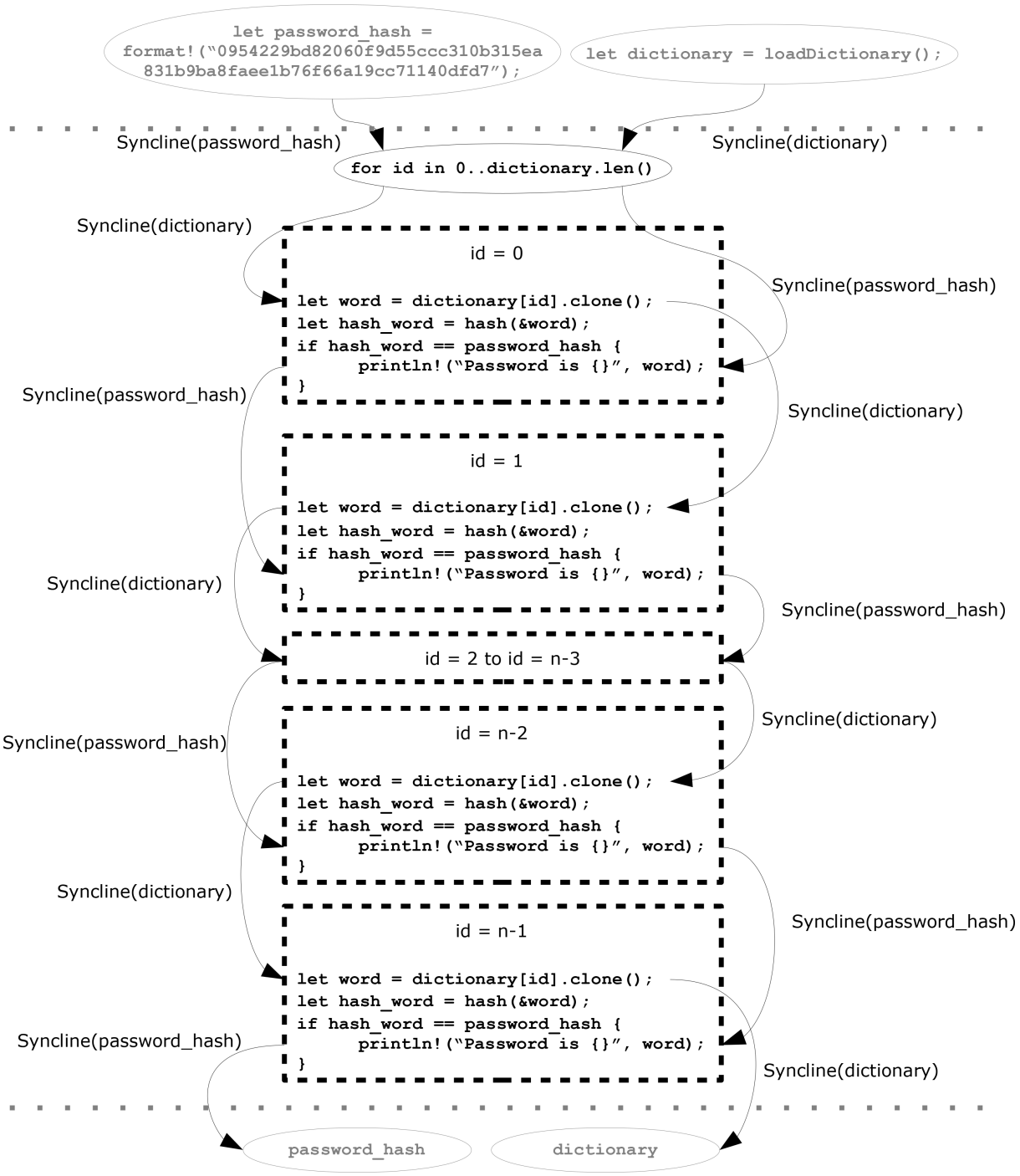


Figure 5.4: A schedule to show how the `for loop` in the main method from the password cracker program is parallelised

5.1.3 Automatically Generated Sequential Programs

To verify that the parallelising compiler can parallelise more that the programs shown above, it was tested against a collection of randomly generated sequential programs with increasing complexity. The first set of programs only contained variable creation, variable assignment and printing arbitrary expressions. A list was added at the beginning of the program which puts the command line arguments into an array. This was used to add some consistent dynamic element that the compiler optimisations would not be able to optimise out. The second set of programs added `for loops` and `if` statements, which allows for nested blocks. I tested

these programs without parallel `for loop` optimisations enabled. The third set uses the same sequential programs as the second set but has parallel `for loop` optimisations enabled. The complexity value varies for each program within each set. This affects how many statements are generated in the sequential source code. Listing 8 is an example of a generated program from the first set. Listing 9 is an example from the second/third set.

```
fn main() {
    let stdin: Vec<i32> = ::std::env::args().skip(1).map(|i| i.parse::<i32>().unwrap()).collect();
    {
        println!("00000000: (stdin[0]) - (stdin[4]) = {:?}", (stdin[0]) - (stdin[4]));
        println!("00000001: (stdin[17]) - (((4) + (1)) - (-4)) = {:?}", (stdin[17]) - (((4) + (1)) - (-4)));
        println!("00000002: (stdin[8]) - (1) = {:?}", (stdin[8]) - (1));
        let mut s: i32 = 0;
        s = stdin[9];
        println!("00000003: ((8) * (9)) - ((s) * (1)) = {:?}", ((8) * (9)) - ((s) * (1)));
        let mut u: i32 = 0;
        println!("00000004: u = {:?}", u);
        let mut a: i32 = 0;
        let mut z: i32 = 0;
        println!("00000005: s = {:?}", s);
        println!("00000006: u = {:?}", u);
        println!("00000007: a = {:?}", a);
        println!("00000008: z = {:?}", z);
    }
}
```

Listing 8: Example generated sequential program from first set with 10 complexity

```
fn main() {
    let stdin: Vec<i32> = ::std::env::args().skip(1).map(|i| i.parse::<i32>().unwrap()).collect();
    {
        println!("00000000: 1 = {:?}", 1);
        for mut y in 0.max(stdin[7])..100.min(stdin[13]) {
            {
                if (((stdin[12]) + (stdin[13])) - (((-1) - (-8)) + (stdin[3]))) < (stdin[3]) {
                    if (-4) < (stdin[7]) {}
                }
                println!("00000001: ((stdin[19]) - ((-7) * (8))) + (((8) - (-5)) * (stdin[4])) = {:?}", ((stdin[19]) - ((-7) * (8))) + (((8) - (-5)) * (stdin[4])));
                let mut d: i32 = 0;
                println!("00000002: d = {:?}", d);
            }
        }
        let mut y: i32 = 0;
        for mut f in 0.max((((-1) + (-3)) - ((-4) * (-1))) + (7))..100.min(y) {
            {
                let mut m: i32 = 0;
                let mut p: i32 = 0;
                if (stdin[1]) < (((stdin[1]) * ((0) + (-4))) - (y)) {
                    println!("00000003: stdin[7] = {:?}", stdin[7]);
                    println!("00000004: y = {:?}", y);
                    println!("00000005: m = {:?}", m);
                    println!("00000006: p = {:?}", p);
                }
                println!("00000007: y = {:?}", y);
                println!("00000008: m = {:?}", m);
                println!("00000009: p = {:?}", p);
            }
        }
        y = y;
        for mut g in 0.max(((y) + ((2) * (3))) - (stdin[14]))..100.min((y) * ((8) * (stdin[15]))) {
            {
                if (y) < (-9) {
                    let mut n: i32 = 0;
                    println!("00000010: y = {:?}", y);
                    println!("00000011: n = {:?}", n);
                }
                println!("00000012: stdin[6] = {:?}", stdin[6]);
```

```
40                  for mut c in 0.max(y)..100.min((((-4) * (-3)) - ((0) + (9))) + (((2) + (1)) +
    ↪  (-2))) {
41                      {
42                          if ((((6) + (2)) - ((7) - (-9))) * (((6) + (-2)) - ((1) * (-10)))) < ((((8)
    ↪  - (4)) - ((-2) - (7))) - (-10)) {
43                              println!("00000013: y = {:?}", y);
44                          }
45                          println!("00000014: y = {:?}", y);
46                      }
47                  }
48                  println!("00000015: y = {:?}", y);
49              }
50          }
51          println!("00000016: y = {:?}", y);
52      }
53  }
```

Listing 9: Example generated sequential program from second and third set with 6 complexity

Each randomly generated sequential program was executed sequentially and in parallel, with and without compiler optimisations to compare the outputs. This is to verify that the program semantics remains the same. The sequential and parallel program was run 1000 times, and their total execution time was recorded. Each parallel program source code was statically analysed to count the number of threads created and the number of synclines used. If the thread was defined inside a `for loop`, it would create more than one thread, but only one thread would be counted. I am not expecting a speedup in the parallel programs as the problem complexity remained fairly small, and is probably not enough to overcome the thread overhead. The more interesting value will be the number of threads and synclines as this gives an indication of how much parallelism the parallelising compiler could find in the sequential program. It gives us a gauge on how fast the parallel version could be if there was no thread overhead.

5.2 Results

5.2.1 Simple Example

Runtime Results	Without Optimisations	With Optimisations
Sequential Version	0.953s	0.992s
Parallel Version	1.131s	1.144s

The results show that the parallel version is slower which is to be expected for this program. The overhead of running this example in parallel is approximately 14.5%.

5.2.2 Password Cracker

Runtime Results	Without Optimisations	With Optimisations
Sequential Version	2m 9.28s	3.975s
Parallel Version Without Parallel For Loops	2m 22.53s	4.485s
Parallel Version With Parallel For Loops	13.724s	0.966s

Due to the larger problem size, the program was only run once instead of the 1000 times that all the other programs were tested with. This is the only program I have that shows a speedup when run in parallel. The parallel version without parallel `for loops` is slightly slower than the sequential version without and with compiler optimisations. Looking at the versions without

compiler optimisations, the sequential version is very slow giving the parallel version with parallel `for loops` a speedup of 9.42. Even when the compiler optimisations are enabled, the parallel version with parallel `for loops` is 4.11 times faster. This speedup shows that the parallelising compiler can beat the Rust compiler under some circumstances. The parallel version with parallel `for loops` did not compile correctly on its own. One of the synclines for the `for loop` sends the dictionary between iterations. Rust has type inference, but it fails to work out the type in this case. I manually annotated the type for this syncline and then it compiled just fine. Originally I attempted to test with a dictionary size of 2150838 words, but later I reduced it to 5000 words. The parallel version with parallel `for loops` tries to hash each word in a separate thread, and so the program was crashing when it could not spawn over 2 million threads. Both these limitations of the design are looked at in chapter 6.

5.2.3 Automatically Generated Sequential Programs

The first set of programs was tested at 8 different complexities in the range 25–200. Figure 5.5, Figure 5.8 and Figure 5.11 shows how many of the programs were successfully compiled. The number of successful compiles for the parallel version is reduced significantly with an increased complexity. There were always 51 programs generated for each complexity, but sometimes the number of sequential programs was lower. We would expect every sequential program to be successful. In the cases it fails, either the sequential code generator had a bug, or the output of the program was different between the sequential and parallel version. Either way, it only occurred 16 out of the 1428 generated programs.

Figure 5.6, Figure 5.9 and Figure 5.12 show the average runtime for each complexity. In almost all of the complexities, the sequential version has a constant time. The parallel version tends to have an increase in runtime when the complexity increases and spikes on some complexities. Since the number of successful parallel compiles is reduced when the complexity is increased, the result becomes less reliable. Some of the spikes in runtime show how badly the parallel version can perform in some situations. Figure 5.7, Figure 5.10 and Figure 5.13 show the number of threads and synclines found when statically analysing the parallel source code. The parallel programs that failed to compile are included in these results. For the first set of generated programs, the increase in complexity increases the number of threads/synclines linearly. The number of synclines increases at a slightly lower rate indicating the more complex the program, the more parallelisable it is. For the second and third set of generated programs, the increase in the number of theads is exponential. For loops are a key target for parallel optimisation, especially if each iteration is separated. Comparing the second set with the third set of generated programs, the number of threads is slightly higher for the third set. The number of synclines are higher for the third set as each variable needs to be passed between iterations of the parallel `for loops`.

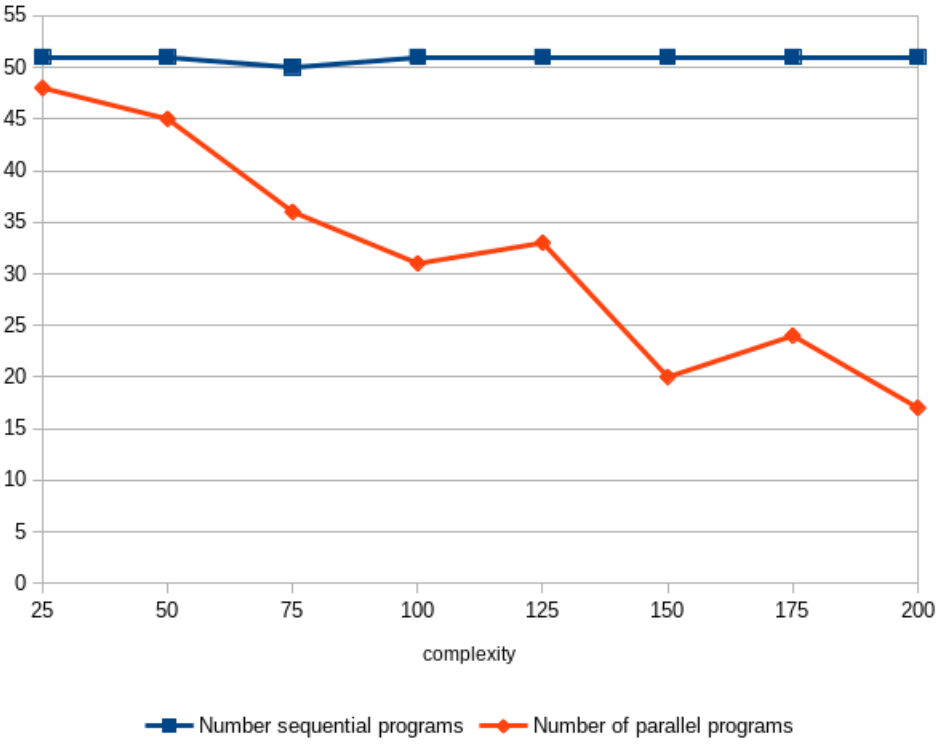


Figure 5.5: Number of programs successfully parallelised for the first program set

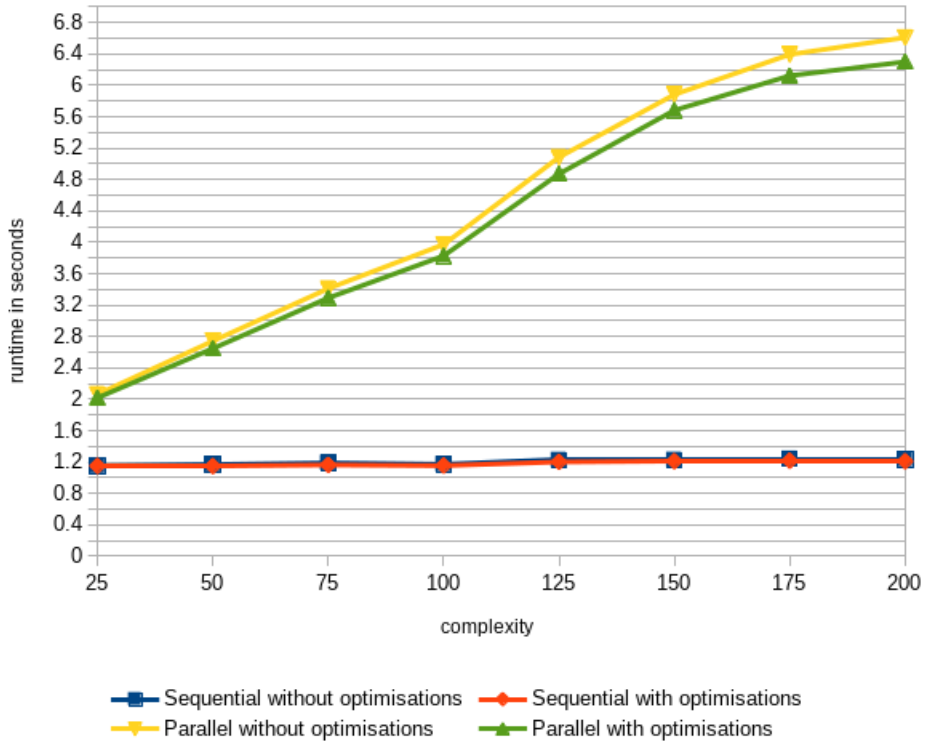


Figure 5.6: Average runtime for the first program set

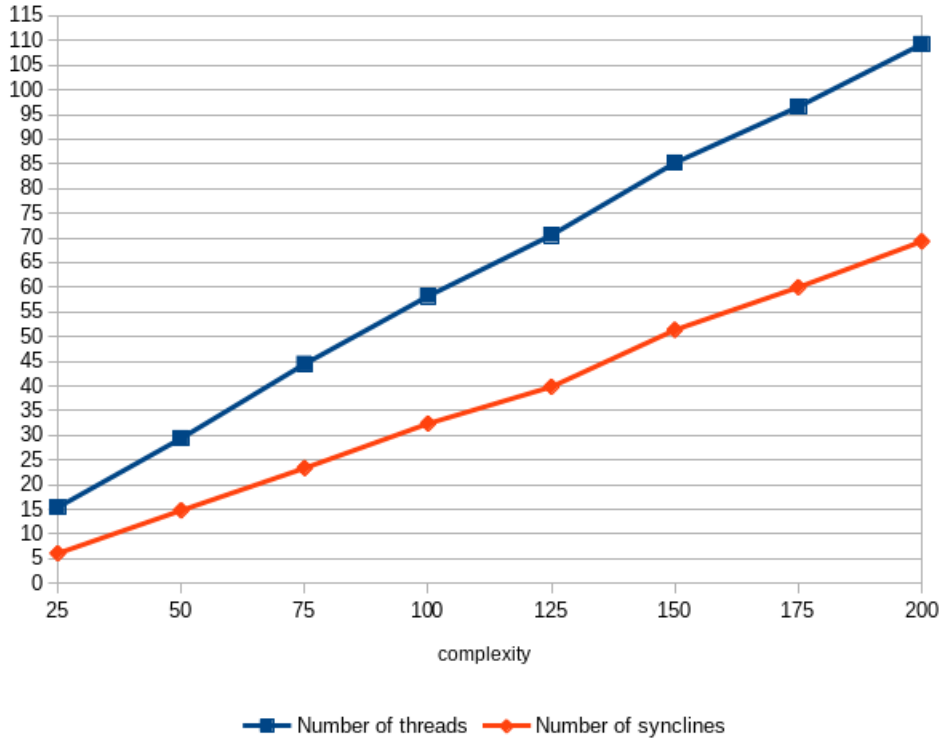


Figure 5.7: Average number of threads and synclines for the first program set

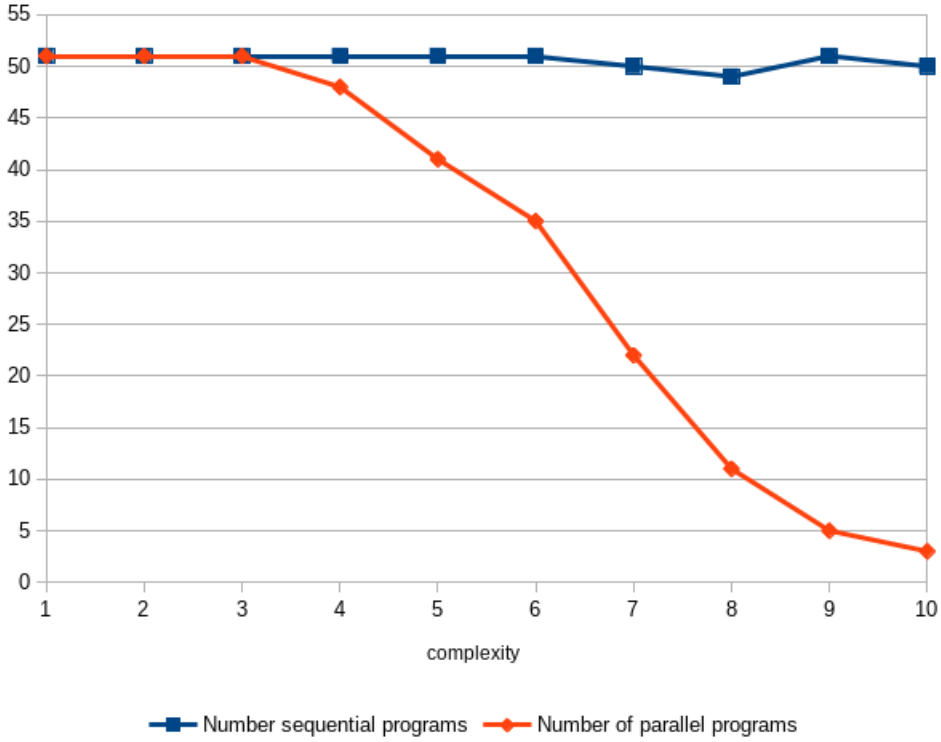


Figure 5.8: Number of programs successfully parallelised for the second program set

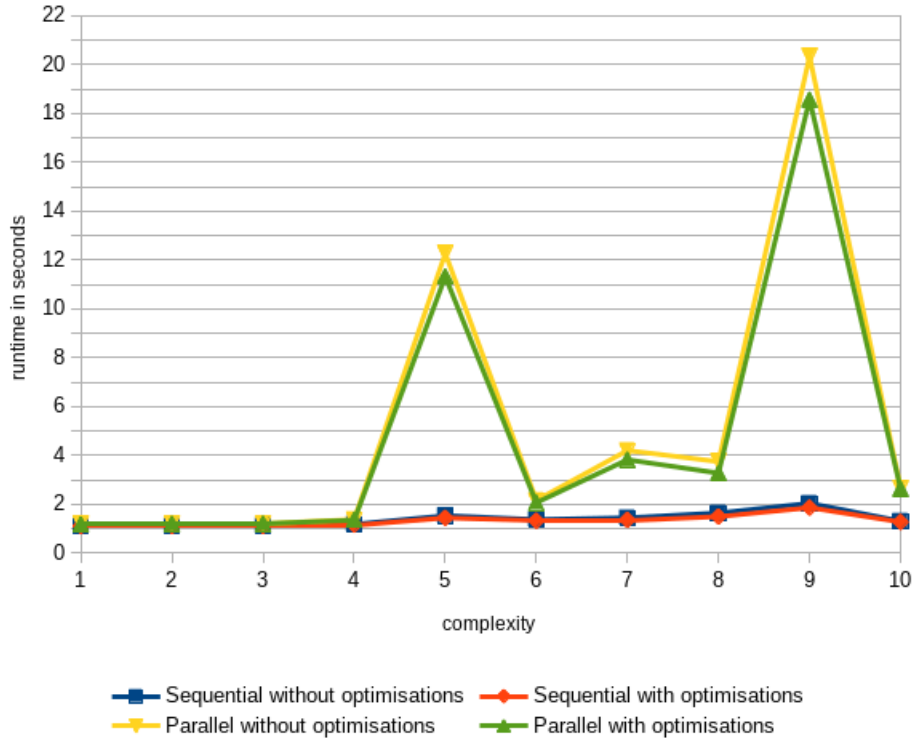


Figure 5.9: Average runtime for the second program set

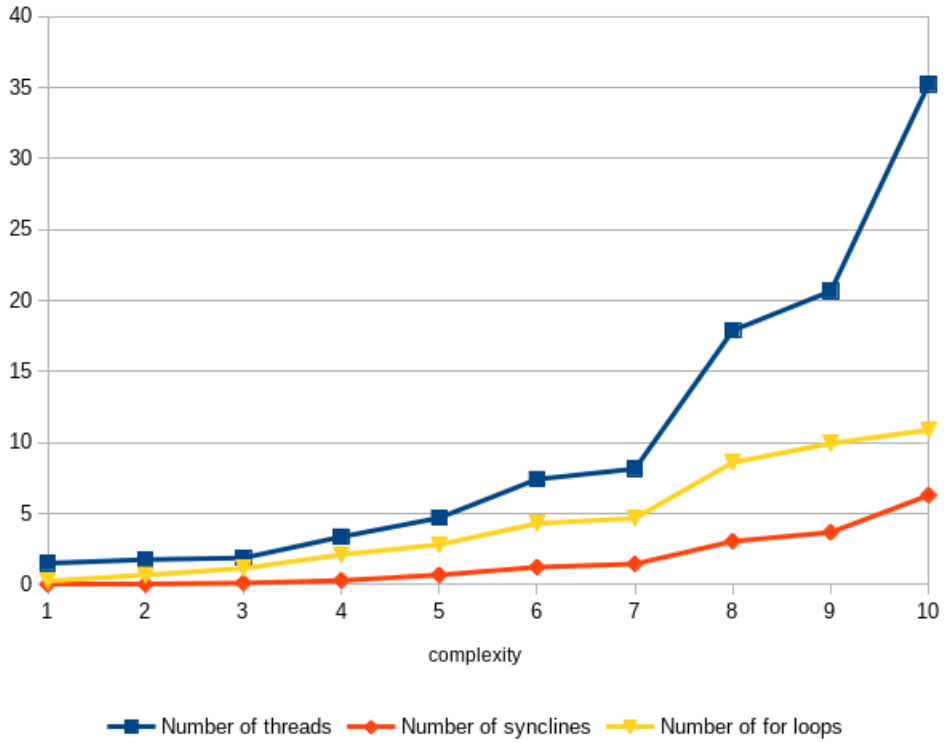


Figure 5.10: Average number of threads and synclines for the second program set

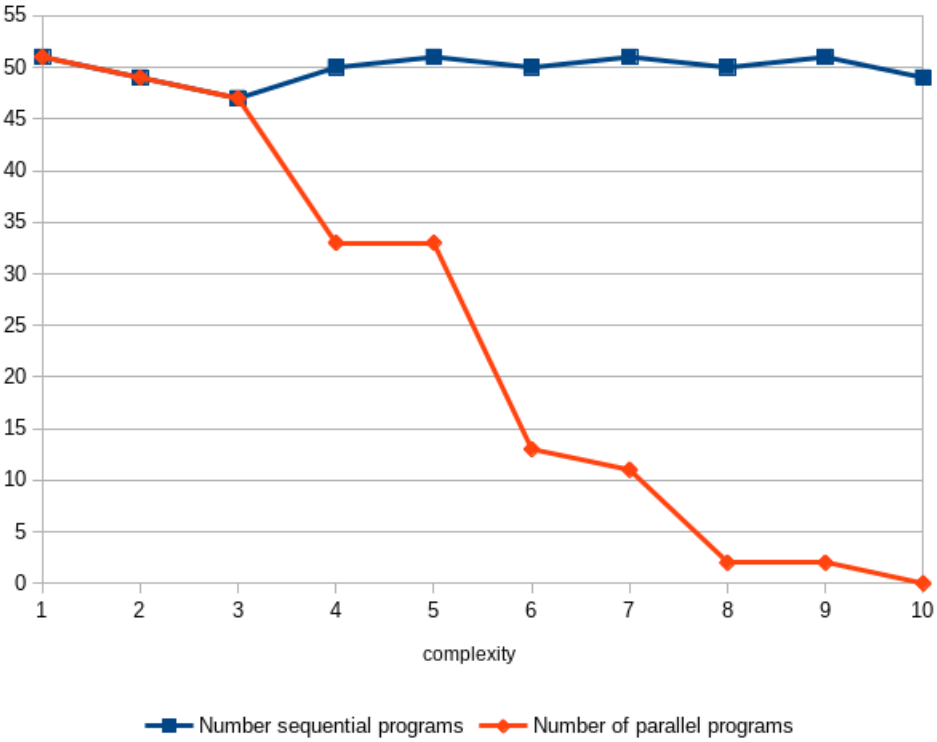


Figure 5.11: Number of programs successfully parallelised for the third program set

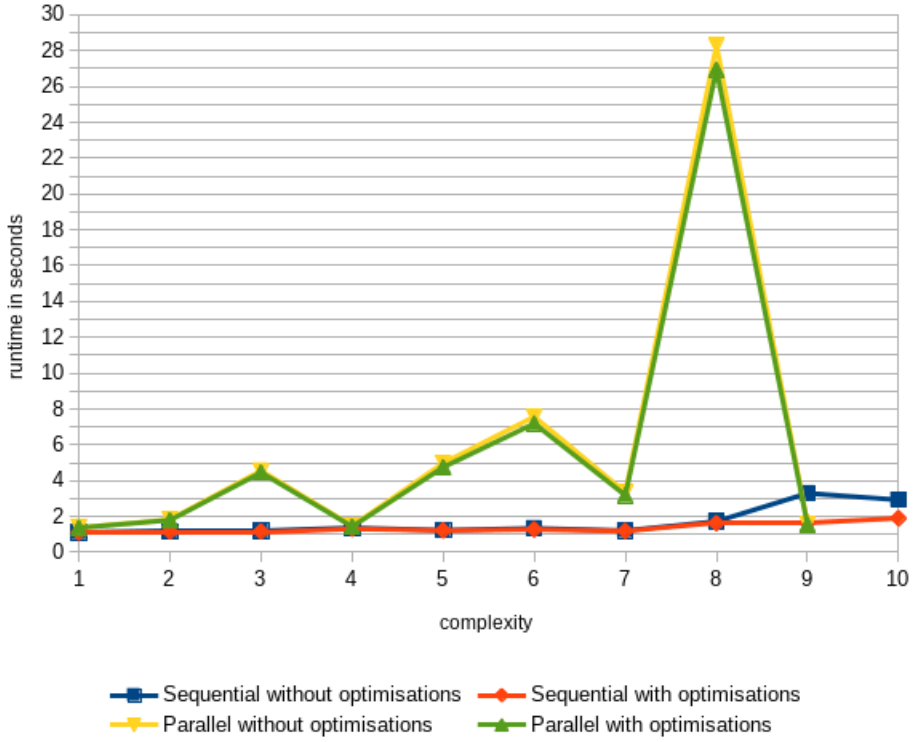


Figure 5.12: Average runtime for the third program set

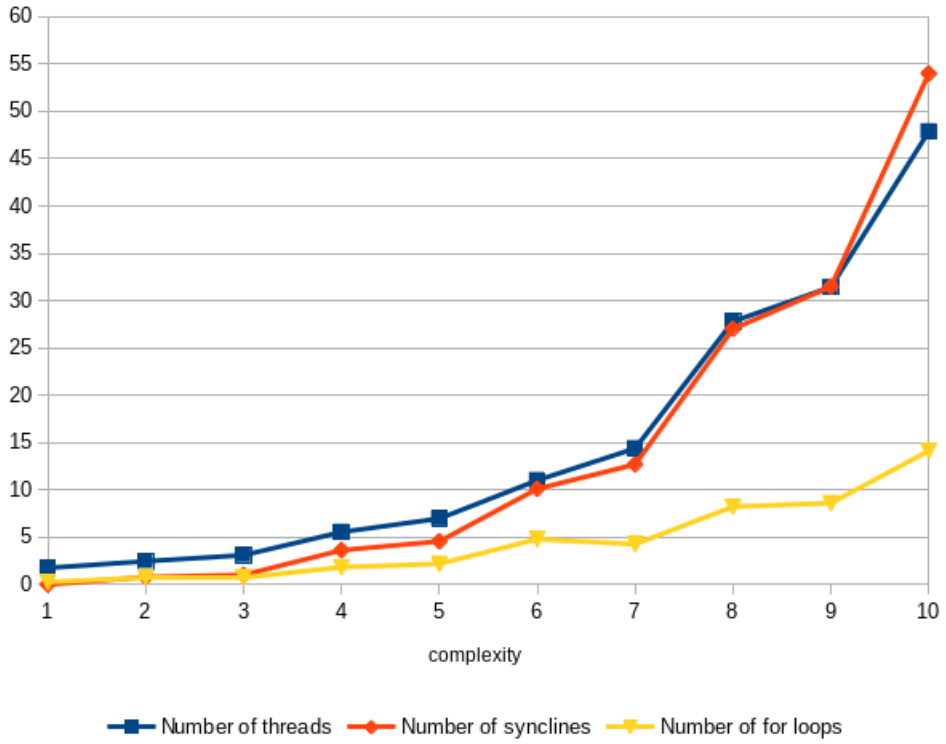


Figure 5.13: Average number of threads and synclines for the third program set

Chapter 6

Discussion

6.1 Achievements

As shown in the evaluation section of this report, my solution manages to separate inter-weaved programs pretty well. It tries to run as much as possible in parallel, and creates synclines for dependencies in different threads. If one statement directly depends on another, then it is placed in the same thread to try to reduce the number of unnecessary threads. Not all programs will compile as not all of the Rust language features are implemented properly in the parallelising compiler. Getting the parallelising compiler to work on simple examples took longer than I originally thought, which ended up delaying the project quite a bit. Once I had it mostly working for simple examples I had the choice of either continuing to fixed more bugs or attempting to parallelise `for loops`. I chose to focus on parallel `for loops` and I am particularly proud of what I achieved as it took a lot of thought to add this into the project. Parallelising `for loops` allows for the parallelising compiler to gain a speedup in the password cracker example. The speedup could also be achieved without parallel `for loops` if the loop was manually unwrapped, but it is unlikely that a programmer would write code this way.

6.2 Improvements

The for loop parallelisation could be vastly improved and extended to work with more types of loops. It is not known how many iteration a while loop will execute at compile time and so it is not known how many threads to spawn. The program could spawn one thread at a time and then check if it should terminate, but this would be slower than sequentially running the loop due to thread overhead. To parallelise this kind of loop, some speculative parallelism would have to take place to run faster. The program could spawn 20 threads each running a separate iteration. Any external dependencies would be passed between iterations, as long as the termination condition is not met. If the termination condition is not met by the end of the 20 threads, then it should spawn another batch. When the termination condition is finally met, all the other threads need to be cancelled. In the literature, speculative parallelism requires undoing the cancelled threads, but in this case the cancelled threads are not touching anything external, and so it is as if they have not been run.

More features of the language could be dealt with to parallelise more programs, more accurately. Unsafe blocks/functions are not dealt with properly in the current version. They allow for modification of a global variable as well as dereferencing a pure pointer value. Since it would be possible for the unsafe block/function to access a variable without using the variables name, and detecting this would be very difficult, the unsafe block/function should be dependent on all statements before it. That way, the program state in the parallel version should be exactly what the sequential program is. All statements after the unsafe block/function should also be dependent on the unsafe block/function to remain consistent and to stop another statement being scheduled at the same time as the unsafe block/function.

When a variable is sent down a channel, it is moved between threads. The variable's type must

implement the `Send` trait to allow this to happen. Most types allow for this, but not all types. The current compiler does not check if the type implements this, or even if the variable is owned and just assumes that it is fine. In the cases that a variable whose type does not implement the `Send` trait is chosen to be sent down the channel the parallel program will not compile.

The compiler does not check whether a variable is mutable or immutable and assumes all variables are mutable. If the compiler checked this, it may be able to find more parallelisms. Immutable variables are not going to change, and so they could be cloned (if the Clone trait is implemented) into each thread where they are used. This would reduce the number of unnecessary synclines and increase the parallel performance. Cloning immutable variables would increase the memory footprint of the runtime parallel program, so some care should be taken not to copy extremely large variables, but in most cases computers have enough memory to cope.

It is important to manage the number of threads being used but the current compiler does not do this. Subsection 5.2.2 shows that the parallel program crashes when too many threads are spawned. One technique to combat this would be to combine threads together. Not all the threads spawned do enough work to overcome the overhead of spawning the thread which makes the parallel program run slower. A performance analysis technique would be needed to estimate whether it is faster of slower to run part of the program in a separate thread. If it would be slower, then the scheduler should combine threads together to reduce the number of synclines. While this technique will help to combat too many threads, and would solve some of the overhead problems, it might not be enough in some large, computationally expensive parallelisable programs. A threadpool could be used to limit the number of threads that are being executed but some care must be taken to not cause a deadlock using this technique. If all the executing threads are waiting for a dependency in a future thread, then that future thread will not be run as the current threads will never finish.

One last improvement I would suggest is to store the type information along with the variable name when deconstructing. Rust has type inference which works most of the time, but as shown in subsection 5.2.2, type inference sometimes fails. By including the type information with the variable, the reconstrutor could annotate each variable with a type in the reconstructed code so that we do not need to rely on type inference.

6.3 Approach Changes

While the approach I took managed to find some parallelisms in the code, there are a few changes to the approach that might find more parallelisms more easily. These changes would be difficult to add into my implementation, and so would require starting essentially from scratch.

The first approach change I wish to suggest is moving away from statement level parallelisms to expression level parallelisms. There may be one statement which adds two slow function calls together. If this is written in one statement, it will execute the two slow functions sequentially but expression level parallelisms could spot this. This change significantly increases the difficulty of all the stages of the parallel compiler, and increases the chance that the program will return the wrong result if the parallelising compiler is written incorrectly.

I would also suggest using the high-level intermediate representation (HIR) instead of using the Rust plugins that I used. Some of the code that my parallelising compiler cannot parallelise can be rewritten slightly to make it parallelisable. The main reason for this is there are many ways to write the same program. With a smaller number of expression types, parallelisms can be more focused as the parallelising compiler would not need to deal with as much syntax sugar.

Not all dependencies are directly visible as variables. In subsection 5.2.2, it is shown that a

common pattern for timing a function call is completely independent of the function call itself when looking at variables alone. The programmer intends for the timing circuit to actually time the function, but the parallel code created does not. Other external functions may also have a shared state (unsafe in Rust, or an unsafe function call) and could produce other similar problems if the order is changed. The unsafe information may not be visible as if a safe function calls an unsafe function, the safe function remains safe. The parallel compiler would need access to all of the library function body to fully evaluate if it is safe to run in a different order. It may not be possible to detect if the functions can be run at a different time without hard coding.

Chapter 7

Conclusion

The problem that I wanted to tackle was automatically converting sequential program into a parallelised program. This is a difficult problem which is well established in the field of computer science. There are many different approaches explored in the literature, but I focused on converting the sequential source code automatically. Rust was chosen as the programming language used due to its memory management features and thread safety. Features of a parallelising compiler which would convert a sequential Rust program into a parallelised Rust program was designed and implemented using a compiler plugin. While this did give me all the information required for dependency analysis, it was very complicated and difficult to access. Using the compiler plugin required me to deal with all of the syntax sugar, dramatically increasing the number of cases I had to deal with. Most of the dependencies are detected by just looking at variables alone.

My implementation could be improved in many ways. Ordering external functions will fix some of the differences between the sequential and parallel program at the cost of forcing some dependencies that might not be necessary. Even with all the problems with my solution, I still managed to gain a speedup on the password cracker program. This shows that there is hope for the approach. If the changes that are described in chapter 6 are made then the parallelising compiler should be more consistent.

Chapter 8

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, Wlodzimierz Bielecki, Albert Cohen, Marek Palkowski and Krzysztof Siedlecki (2011). "Coarse-grained loop parallelization: Iteration space slicing vs affine transformations". In: *Parallel Computing* 37.8, pp. 479–497.

Bondhugula, Uday, Muthu Baskaran, Sriram Krishnamoorthy, Jagannathan Ramanujam, Atanas Rountev and Ponnuswamy Sadayappan (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 (1992). "Some efficient solutions to the affine scheduling problem. I. One-dimensional time". In: *International journal of parallel programming* 21.5, pp. 313–347.

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

Kim, Hong Soog, Young Ha Yoon, Sang Og Na and Dong Soo Han (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, Carlos Madriles, Jesús Sánchez, Pedro Marcuello, Antonio González and Dean M Tullsen (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).

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, Mojtaba Mehrara, Steve Lieberman and Scott Mahlke (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.

Chapter 9

Appendix

9.1 Running the Code

Install rustup and rustc 1.25.0-nightly (0c6091fbd 2018-02-04) compiler:
`curl https://sh.rustup.rs -sSf | sh -s -- -y --default-toolchain nightly-2018-03-05`

Verify the version is correct using:
`rustc --version`

Create a new crate and write sequential code:
`cargo init`

Under `[dependencies]` in `Cargo.toml`, add:
`auto_parallelise={version="*", git="https://github.com/MichaelOultram/Auto_Parallelise/"}`

At the top of your `lib.rs` or `main.rs` file, add:
```
#![feature(plugin)]
#![plugin(auto_parallelise)]
```

At the top of every function, add:
`#[autoparallelise]`

Compile the code once to run the analysis stage:
`cargo build --release`

Normally you would compile the code again with the same command to apply the modifications but due to a bug you must pipe stdout into a file instead:
`cargo build --release > parallel_code.rs`

Normally you would just run the parallelised code but due to the bug, you will need to create a new project and copy `parallel_code.rs` along with any imports. Then you can:
`cargo run --release`

9.2 Rust Compiler Types

rust/src/libsyntax/visit.rs

```
34  pub enum FnKind<'a> {
35      /// fn foo() or extern "Abi" fn foo()
36      ItemFn(Ident, Unsafety, Spanned<Constness>, Abi, &'a Visibility, &'a Block),
```

```
37
38      /// fn foo(&self)
39      Method(Ident, &'a MethodSig, Option<&'a Visibility>, &'a Block),
40
41      /// |x, y| body
42      Closure(&'a Expr),
43  }
```

Listing 10: `FnKind` enum

rust/src/libsyntax/ast.rs

```
489 pub struct Block {
490     /// Statements in a block
491     pub stmts: Vec<Stmt>,
492     pub id: NodeId,
493     /// Distinguishes between `unsafe { ... }` and `{ ... }`
494     pub rules: BlockCheckMode,
495     pub span: Span,
496     pub recovered: bool,
497 }
```

Listing 11: `Block` struct

rust/src/libsyntax/ast.rs

```
781 pub struct Stmt {
782     pub id: NodeId,
783     pub node: StmtKind,
784     pub span: Span,
785 }
```

Listing 12: `Stmt` struct

rust/src/libsyntax/ast.rs

```
815 pub enum StmtKind {
816     /// A local (let) binding.
817     Local(P<Local>),
818
819     /// An item definition.
820     Item(P<Item>),
821
822     /// Expr without trailing semi-colon.
823     Expr(P<Expr>),
824     /// Expr with a trailing semi-colon.
825     Semi(P<Expr>),
826     /// Macro.
827     Mac(P<(Mac, MacStmtStyle, ThinVec<Attribute>)>),
828 }
```

Listing 13: `StmtKind` enum

rust/src/libsyntax/ast.rs

```
899 pub struct Expr {
900     pub id: NodeId,
901     pub node: ExprKind,
902     pub span: Span,
903     pub attrs: ThinVec<Attribute>
904 }
```

Listing 14: `Expr` struct

rust/src/libsyntax/ast.rs

```
987 pub enum ExprKind {
988     /// A `box x` expression.
989     Box(P<Expr>),
990     /// First expr is the place; second expr is the value.
991     InPlace(P<Expr>, P<Expr>),
992     /// An array (`[a, b, c, d]`)
993     Array(Vec<P<Expr>>),
994     /// A function call
995     ///
```

```
996         /// The first field resolves to the function itself,
997         /// and the second field is the list of arguments.
998         /// This also represents calling the constructor of
999         /// tuple-like ADTs such as tuple structs and enum variants.
1000        Call(P<Expr>, Vec<P<Expr>>),
1001        /// A method call (`x.foo::<'static, Bar, Baz>(a, b, c, d)`)
1002        ///
1003        /// The `PathSegment` represents the method name and its generic arguments
1004        /// (within the angle brackets).
1005        /// The first element of the vector of `Expr`s is the expression that evaluates
1006        /// to the object on which the method is being called on (the receiver),
1007        /// and the remaining elements are the rest of the arguments.
1008        /// Thus, `x.foo::<Bar, Baz>(a, b, c, d)` is represented as
1009        /// `ExprKind::MethodCall(PathSegment { foo, [Bar, Baz] }, [x, a, b, c, d])`.
1010        MethodCall(PathSegment, Vec<P<Expr>>),
1011        /// A tuple (`(a, b, c ,d)`)
1012        Tup(Vec<P<Expr>>),
1013        /// A binary operation (For example: `a + b`, `a * b`)
1014        Binary(BinOp, P<Expr>, P<Expr>),
1015        /// A unary operation (For example: `!x`, `*x`)
1016        Unary(UnOp, P<Expr>),
1017        /// A literal (For example: `1`, `"foo"`)
1018        Lit(P<Lit>),
1019        /// A cast (`foo as f64`)
1020        Cast(P<Expr>, P<Ty>),
1021        Type(P<Expr>, P<Ty>),
1022        /// An `if` block, with an optional else block
1023        ///
1024        /// `if expr { block } else { expr }`
1025        If(P<Expr>, P<Block>, Option<P<Expr>>),
1026        /// An `if let` expression with an optional else block
1027        ///
1028        /// `if let pat = expr { block } else { expr }`
1029        ///
1030        /// This is desugared to a `match` expression.
1031        IfLet(P<Pat>, P<Expr>, P<Block>, Option<P<Expr>>),
1032        /// A while loop, with an optional label
1033        ///
1034        /// `'label: while expr { block }`
1035        While(P<Expr>, P<Block>, Option<SpannedIdent>),
1036        /// A while-let loop, with an optional label
1037        ///
1038        /// `'label: while let pat = expr { block }`
1039        ///
1040        /// This is desugared to a combination of `loop` and `match` expressions.
1041        WhileLet(P<Pat>, P<Expr>, P<Block>, Option<SpannedIdent>),
1042        /// A for loop, with an optional label
1043        ///
1044        /// `'label: for pat in expr { block }`
1045        ///
1046        /// This is desugared to a combination of `loop` and `match` expressions.
1047        ForLoop(P<Pat>, P<Expr>, P<Block>, Option<SpannedIdent>),
1048        /// Conditionless loop (can be exited with break, continue, or return)
1049        ///
1050        /// `'label: loop { block }`
1051        Loop(P<Block>, Option<SpannedIdent>),
1052        /// A `match` block.
1053        Match(P<Expr>, Vec<Arm>),
1054        /// A closure (for example, `move |a, b, c| a + b + c`)
1055        ///
1056        /// The final span is the span of the argument block `|...|`
1057        Closure(CaptureBy, P<FnDecl>, P<Expr>, Span),
1058        /// A block (`{ ... }`)
1059        Block(P<Block>),
1060        /// A catch block (`catch { ... }`)
1061        Catch(P<Block>),
1062
1063        /// An assignment (`a = foo()`)
1064        Assign(P<Expr>, P<Expr>),
1065        /// An assignment with an operator
1066        ///
1067        /// For example, `a += 1`.
1068        AssignOp(BinOp, P<Expr>, P<Expr>),
```

```
1069        /// Access of a named struct field (`obj.foo`)
1070        Field(P<Expr>, SpannedIdent),
1071        /// Access of an unnamed field of a struct or tuple-struct
1072        ///
1073        /// For example, `foo.0`.
1074        TupField(P<Expr>, Spanned<usize>),
1075        /// An indexing operation (`foo[2]`)
1076        Index(P<Expr>, P<Expr>),
1077        /// A range (`1..2`, `1..`, `..2`, `1...2`, `1...`, `...2`)
1078        Range(Option<P<Expr>>, Option<P<Expr>>, RangeLimits),
1079
1080        /// Variable reference, possibly containing `::` and/or type
1081        /// parameters, e.g. foo::bar::<baz>.
1082        ///
1083        /// Optionally "qualified",
1084        /// E.g. `<Vec<T> as SomeTrait>::SomeType`.
1085        Path(Option<QSelf>, Path),
1086
1087        /// A referencing operation (`&a` or `&mut a`)
1088        AddrOf(Mutability, P<Expr>),
1089        /// A `break`, with an optional label to break, and an optional expression
1090        Break(Option<SpannedIdent>, Option<P<Expr>>),
1091        /// A `continue`, with an optional label
1092        Continue(Option<SpannedIdent>),
1093        /// A `return`, with an optional value to be returned
1094        Ret(Option<P<Expr>>),
1095
1096        /// Output of the `asm!()` macro
1097        InlineAsm(P<InlineAsm>),
1098
1099        /// A macro invocation; pre-expansion
1100        Mac(Mac),
1101
1102        /// A struct literal expression.
1103        ///
1104        /// For example, `Foo {x: 1, y: 2}`, or
1105        /// `Foo {x: 1, .. base}`, where `base` is the `Option<Expr>`.
1106        Struct(Path, Vec<Field>, Option<P<Expr>>),
1107
1108        /// An array literal constructed from one repeated element.
1109        ///
1110        /// For example, `[1; 5]`. The first expression is the element
1111        /// to be repeated; the second is the number of times to repeat it.
1112        Repeat(P<Expr>, P<Expr>),
1113
1114        /// No-op: used solely so we can pretty-print faithfully
1115        Paren(P<Expr>),
1116
1117        /// `expr?`
1118        Try(P<Expr>),
1119
1120        /// A `yield`, with an optional value to be yielded
1121        Yield(Option<P<Expr>>),
1122    }
```

Listing 15: `ExprKind` enum

rust/src/libsyntax_pos/lib.rs

```
143 pub struct SpanData {
144     pub lo: BytePos,
145     pub hi: BytePos,
146     /// Information about where the macro came from, if this piece of
147     /// code was created by a macro expansion.
148     pub ctxt: SyntaxContext,
149 }
```

Listing 16: `SpanData` struct