INSTITUT FÜR INFORMATIK
der Ludwig-Maximilians-Universität München

A Framework for Observing the Parallel Execution of Rust Programs

Frederic Pascal Sautter

Masterarbeit

Aufgabensteller	Prof. Dr. François Bry
Betreuer	Prof. Dr. François Bry, Thomas Prokosch
Abgabe am	07. Juni 2019

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst habe und keine anderen als die angegebenen Hilfsmittel verwendet habe.

München, den 07. Juni 2019 Frederic Pascal Sautter

Abstract

With computing hardware evolving, processors incorporate more and more processing units. As the number of simultaneously possible computations steadily increases, developing efficient and reliable parallel applications becomes essential in order to take full advantage of the additional computing power. The young programming language Rust aims to ease the development of such high-performance applications. In contrast to Fortran, C, and C++, languages common in the high-performance computing domain, Rust combines low-level control of resources with strong safety guarantees. However, existing parallel performance analysis tools such as Score-P, TAU, and Vampir lack support for Rust.

This work presents the design and implementation of the tRust framework, which allows observing the execution of parallel Rust applications. This framework provides a modified Rust compiler for the automated insertion of probes into the observed program and its dependencies. A run-time library enables the transmission of observation data to a central collector application for persistent storage. This centrally collected data allows for extensive analysis of the run-time behavior of the program.

The tRust framework is evaluated using three different benchmark applications, each targeting one of three common parallelism libraries in Rust. The experiments suggest good performance of the tRust framework. More important is the fact that tRust allows programmers to precisely retrace program execution of concurrent and parallel Rust programs with libraries that implement parallel paradigms not immediately available in the established HPC programming languages Fortran, C, and C++.

Zusammenfassung

Durch laufende Weiterentwicklung von Rechnerhardware erhalten Prozessoren immer mehr parallel arbeitende Recheneinheiten. Da dadurch die Zahl der simultan möglichen Berechnungen steigt, wird die Entwicklung von effizienten und zuverlässigen parallelen Programmen unerlässlich, um die zusätzliche Rechenkapazität voll auszuschöpfen. Die junge Programmiersprache Rust versucht die Entwicklung solcher hochperformanter Anwendungen zu erleichtern. Im Gegensatz zu den Sprachen Fortran, C und C++, die bei der Programmierung von Hochleistungsrechnern vorwiegend eingesetzt werden, kombiniert Rust hardwarenahen Zugriff mit starken Sicherheitsgarantien. Jedoch, fehlt bei existierenden parallelen Leistungsanalysewerkzeugen, Score-P, TAU und Vampir die Unterstützung für Rust.

Diese Arbeit legt den Entwurf und die Umsetzung des tRust Frameworks dar, mit dem der Ablauf paralleler Rust-Anwendungen überwacht werden kann. Das Werkzeug stellt einen modifizierten Rust Compiler zur Verfügung, der das automatisierte Einfügen von Analyseinstruktionen in das zu überwachte Programm und dessen Abhängigkeiten ermöglicht. Eine Laufzeitbibliothek ermöglicht das Übertragen der Überwachungsdaten an eine zentrale Sammelanwendung für die persistente Speicherung. Diese zentral abgelegten Daten können in weiterer Folge für die ausführliche Analyse des Programmablaufs genutzt werden.

Das vorgestellte tRust Werkzeug wird anhand von drei unterschiedlichen Testanwendungen beurteilt, wobei jede Anwendung auf eine von drei geläufigen parallelen Programmbibliotheken zugeschnitten ist. Die Experimente legen nahe, dass tRust eine angemessene Leistung erzielt. Wichtiger ist die Tatsache, dass Entwickler die Möglichkeit erhalten, den Programmablauf von nebenläufigen und parallelen Rust Programmen, die in den üblicherweise im Höchstleistungsbereich eingesetzten Programmiersprachen Fortran, C und C++ nicht unmittelbar zur Verfügung stehen.

Acknowledgments

This thesis would not have been possible without the help and support of many others.

First, I want to thank Prof. Dr. François Bry for his valuable advice and guidance throughout this project. The suggestions and the feedback he provided often helped to concentrate on what is essential.

I want to express my great gratitude to my advisor Thomas Prokosch for his outstanding support and advice at all times. Whenever I got stuck along the way, Thomas encouraged me by providing useful suggestions and ideas. Particularly, his backing during the many meetings helped to focus on and structure my work.

Special thanks go to my parents for their moral support especially in challenging times. Furthermore, I express my heartfelt gratitude to Annabel for her backing at all times. Finally, I thank my friends for taking my mind off things from time to time.

Contents

CHAPTER 1

Introduction

In recent years computing hardware has evolved to a point where almost every computing device contains parallel hardware. From small embedded systems to computer clusters consisting of thousands of nodes, multi-core processors have become the standard in 2019. Chip manufacturers such as Intel and AMD release processors containing up to 56 [6] or 64 [1] cores respectively. Moreover, many systems contain graphics chips, which are made up of an increasing number of compute units. The number of simultaneously possible computations constantly increases. In order to benefit from this additional computing power, appropriate software is required. Software developers need to design applications which effectively leverage parallelism. However, experience has shown that creating reliable and efficient parallel programs is difficult. Parallel computing is significantly more complex, as it introduces new classes of errors such as deadlocks and race-conditions, not possible in serial programs. These errors can be difficult to detect since they often occur under specific circumstances.

Developing programming languages designed for the use in concurrent and parallel environments is one approach to enhance reliability and efficiency of parallel programs. Especially since the systems-level languages commonly used for the development of performance critical applications, in particular Fortran, C and C++ provide, little to no safety guarantees at all. For this reason, the Rust programming language was developed. Rust incorporates elements of conventional systems programming language as well as elements of modern high-level languages [7]. It combines low-level control of resources with strong safety guarantees. Rust's expressive static type system enables the compiler to prevent many memory errors and data races. As a consequence, Rust empowers developers to take advantage of modern parallel hardware [42].

While Rust manages to prevent data races there are other errors it cannot prevent, such as bottlenecks occurring when threads are not properly synchronized. Such errors can only be detected at run-time. In order to analyze and further optimize the complex parallel runtime behavior additional tools are required. In the context of high-performance computing (HPC), there already exist sophisticated tools for the analysis of parallel and distributed execution. Tools like Score-P [37], TAU [56], HPCToolkit [18] and Vampir [48] allow the collection of detailed event data during the programs execution. Based on this data it is possible to gain insight into the programs state during execution, providing in-depth information of its run-time behavior. Accordingly, these tools assist developers in finding most of the remaining errors and performance bottlenecks.

The existing performance analysis tools are built around standard HPC libraries, including Message Passing Interface (MPI), Open Multi-Processing (OpenMP), and pthreads. Besides providing bindings to the named libraries, it is often possible to apply these performance tools directly to applications written in specific programming languages. The languages supported vary between these tools, but all of them support the major three languages in the context of HPC, namely C, C++, and Fortran. Since Rust is still relatively new, it is not supported. Even though Rust wrapper libraries for MPI exist, observation is limited to recording MPI calls, which is not sufficient for extensive performance analysis. This work concentrates on the design and the implementation of the tRust framework which allows observing the execution of parallel Rust applications.

The remainder of this work is structured as follows:
Chapter 2 provides an introduction to the Rust Programming language. It outlines unique features of Rust and gives an overview of the Rust ecosystem.
Chapter 3 covers fundamental concepts such as the architecture of parallel performance tools. Moreover, instrumentation is explained in detail.
Chapter 4 explains design and implementation decisions of the new framework. Therefore, techniques of instrumentation in Rust are described, the data collection architecture is outlined, and instrumented code snippets are explained.
Chapter 5 explains how the proposed framework was tested and evaluates the result of the experiments.
Finally, Chapter 6 gives a conclusion and proposes ideas as well as concepts how this framework can be further improved.

CHAPTER 2

The Rust Programming Language

Rust is a young programming language sponsored by Mozilla Research. Its development is driven by an open source project and backed by a rapidly growing community. A strong indication for its increasing popularity is the fact that as of 2019 Rust is the most loved programming language for the fourth consecutive year according to Stack Overflow Developer Survey [10].

Rust aims to enable low-level resource control while providing high-level guarantees such as type-safety and memory safety. Similar to other common systems programming languages like C or C++, it supports direct hardware access [34] including raw pointer manipulations, and fine-grained control over memory representations. In contrast to C/C++, Rust provides memory safety, avoiding common errors such as dangling pointers, use-after-free, and double-free. Rust achieves this through its strong static type system without depending on a garbage collector. It uses lexical scoping to ensure memory is automatically deallocated when variables go out of scope [42].

This chapter briefly summarizes the key features and concepts of Rust. Section 2.1 explains the more basic Rust syntax. Then ownership and borrowing two concepts unique to Rust are described in Section 2.2. Section 2.3 points out the implications of the ownership models for concurrency and parallelism. Finally, Section 2.4 outlines the ecosystem of Rust and its internal compiler structure.

2.1 Selected Syntax Elements

The syntax of Rust is inspired by C++, Haskell and the ML-family. In Rust, variables are immutable by default. They are defined using a `let` statement. However, if a variable should be mutable, it has to be explicitly stated by adding the keyword `mut` in front of the variable name. In other words, once a value was bound to a name without adding the `mut` keyword it is not possible to change the value of the variable. Listing 2.1 shows the definition of variables. The comments (starting with "`//`") at the end of the last line indicate the error returned by the compiler when trying to mutate a variable not declared as mutable. Treating variables as immutable by default results in compile time errors when a variable is mutated. This helps to prevent bugs where one part of an application relies on the fact that a value will never change, while another component modifies the value, which causes the first part to behave unpredictably. Immutable variables are different from

constants because only immutable variables can be set to the result of an expression that can only be computed at run-time. Listing 2.2 shows the compiler error thrown when trying to assign a value only known at run-time to a constant [35].

```
// Defining a variable (immutable by default)
let var = 5;
// Defining a mutable variable
let mut m_var = 6;

// Change value of mutable variable
m_var = 8;
// Change value of normal (immutable) variable
var = 7;    // ERROR: cannot assign twice to immutable variable `var`
```

Listing 2.1: Defining variables in Rust

```
// Defining a variable (immutable by default)
let im_var = current_time();
// Defining a constant
const CONSTANT1: u32 = 100;
// Not possible
const CONSTANT2: SystemTime = current_time();    // ERROR: calls in
// constants are limited to constant functions,
// tuple structs and tuple variants
```

Listing 2.2: Difference between immutable variables and constants

In Rust the distinction between *statements* and *expressions* plays an important role. *Statment are instructions that perform some action without returning a meaningful value*. Whereas *expressions are instructions that evaluate to a value which is returned.* The second instruction in Listing 2.3 is an expression. However, this expression itself is constructed of multiple smaller expressions, consequently the result of this expression is evaluated from the results of its sub-expressions. In particular the smallest expressions possible are literals, a literal returns its own value. The literal `2` in Listing 2.3 returns the value two. A variable is evaluates to its value, hence the variable `var` evaluates to seven. Consequently, the whole expression returns nine ($9 = 7 + 2$) [35].

In Rust the semicolon "`;`" can be used in order to discard the result of an expression, turning an expression into a statement [13]. Therefore it becomes clear that an expression can be part of a statement. The `let` construct used to define a variable is a statement which expects a "`;`" at its end. The literal `7` in the first instruction in Listing 2.3 evaluates to seven, which is assigned to `var` [35].

```
// Statement
let var = 7;

// Expression
var + 2    // Evaluates to 9
```

Listing 2.3: Statements and expressions

Closely related to these two concepts is the *unit type*, which has exactly one value `()`. This value is returned when no other meaningful value can be returned. Thus it is not the case that statements return no value at all, instead they return `()` referred to as "*unit*"

[13]. From this it follows that every language construct in Rust returns some kind of value, either `()` or some meaningful value. This concept is very important in Rust, as it allows to write very expressive code. For example, it is considered good practice in Rust to omit the `return` keyword. Instead, the return value of a function is the result of the last instruction executed. Furthermore, this holds not only for functions and methods, but for blocks in general: The result of a block is the result of its last evaluated instruction [35].

In Rust functions are defined as follows: `fn func_name(arg1: T, arg2: U, ...) -> R { function body }`. Thus the definition starts with the `fn` keyword followed by the name of the function. Next is a list of the parameters together with their types (`T` and `U`) surrounded by parentheses. If the function returns a value other than the unit value `()` an arrow `->` followed by the return type `R` is added after the closing parenthesis. This is the signature of the function, which is followed by a block containing the function body [35].

In the case of the `positive_power_3` function defined in Listing 2.4 the return value is the result of the `if-else` expression. The `if-else` evaluates to the result of the expression (note no trailing semicolon) in one of its branches, which is the last instruction executed. The second function defined in Listing 2.4 does not return a (meaningful) value, since the statements in both branches of the `if-else` expression just print out the sign of the input. Understanding the basic concept of statements and expressions is vital for understanding the matters of Section 4.3 [35].

```
// Function definition with integer in- and output
fn positive_power_3(x: i32) -> i32 {
    if x < 0 {
        x * x * (-x)
    } else {
        x * x * x
    }
}

// Function definition with integer in- but "no" output
fn sign(x: i32) {    // Short for  fn sign(x: i32) -> () {
    if x < 0 {
        println!("minus");
    } else {
        println!("plus");
    }
}
```

Listing 2.4: Function definitions

Rust allows the definition of *closures*, a language feature often present in functional programming languages. Closures are special anonymous functions. Rust allows for passing closures as arguments, or assigning them to variables for later use. Closures are defined as `|arg1, arg2, ...| { <Function body> }`. Arguments are listed between the two pipes. If the body of the closure consists of one expression, this expression is placed after the second pipe. In cases where the body does not consist of only one expression, it is defined by a normal block similarly to normal functions. In contrast to regular functions closures can capture values from their scope. In line three of Listing 2.5 a closure is defined with a block defining its body. Furthermore, the closure is assigned to the variable `calc`. As a result, the anonymous function now can be called by this name, which is done in the second last line when printing its result to the terminal. In the last line, a closure whose body contains just one expression is defined and passed to a function which takes a closure. In particular, the closure assigned to `calc` uses in its body the two variables `times`

and `add` which were defined earlier without passing them as arguments to the closure. The ability to capture variables from their surrounding scope makes closures a powerful language feature. Especially in the context of concurrency or parallelism closures are used often for defining tasks which should be executed concurrently, such as in Section 2.3 [35].

```
let times = 3;
let add = 7;
let calc = |arg| {
    if add < 9 {
        arg * times + 2 * add
    } else {
        arg * times + add
    }
};
println!("{}", calc(4));
fn_takes_closure(|arg| arg * times);
```

Listing 2.5: Defining and passing closures

2.2 Ownership and Borrowing

As noted earlier, Rust similar to other low-level languages does not have a garbage collector, yet it is rarely necessary to release memory manually [35]. To achieve this, Rust uses a concept of *ownership* integrated into its type system for memory management. In particular, ownership manages the handling of multiple aliases (references to a value in memory). This concept has been gaining popularity among academics as well as mainstream language developers and is Rusts unique feature [34].
Its basic idea is that,

> "[...] although multiple aliases to a resource may exist simultaneously, performing certain actions on the resource (such as reading and writing a memory location) should require a 'right' or 'capability' that is uniquely 'owned' by one alias at any point during the execution of the program." [34]

This is enforced by the following set of rules, which the compiler checks at compile time:

Ownership Rules. According to Steve Klabnik and Carol Nichols [35]

1. *Each value in Rust has a variable that's called its **owner**.*
2. *There can only be one owner at a time.*
3. *When the owner goes out of scope, the value will be dropped.*

First of all, Rust similar to other languages distinguishes between two kinds of values, those stored on the stack and those allocated on the heap. In Rust "Copy" is a marker which tells the compiler data is passed around as *deep copy*. Since the data is stored entirely on the stack, copying it is easy. Values whose size is known at compile time and will not change during execution are "Copy" by default. This includes all primitive data types such as integers, floating-point numbers, characters, and pointers. Besides primitive data types, it is possible to make compound data types "Copy" as well, if they consist of a fixed number of elements which are "Copy" themselves, such as arrays of integers or structs with integer and float fields [35].

Memory management in Rust is based on lexical scopes. A variable is valid for the scope in which it was defined, as soon as the scope ends Rust automatically "drops" the

variable. This is a Rust specific term for invalidating the variable and freeing the memory of its value. In the case of a "Copy" variable, this is straight forward, its value is popped off the stack, releasing its memory. Listing 2.6 shows how variables become invalid when they go out of scope [35].

```
let mut var = 7;
{    // New scope begins
    let inner_var = var;    // value is copied and assigned to inner_var
}    // Scope ends, inner_var is dropped, not affecting var
// Now inner_var is not valid any more
// var is still valid, since its scope hasn't ended
var = var + 1;
var = var + inner_var;    // ERROR: cannot find
                          // value `inner_var` in this scope
```

Listing 2.6: Scops of "Copy" variables

Other values, in particular those not having a fixed size, are allocated on the heap. Copying data located on the heap is supposed to be expensive, thus not the value itself but the reference to the value is copied. This concept, known as *shallow copy*, is common in other languages. This is where Rust leverages the ownership concept [34].

In Listing 2.7 a new string value is assigned to the variable `var1`, by calling the `from` function which is defined on the `String` type indicated by `::`. The `String` type is not "Copy" since string values can grow when adding more characters. According to Rule 1 `var` is now the *owner* of the string value. Next, `var1` is assigned to `var2`. In other languages `var2` would just receive a shallow copy, while `var1` would still be valid. In Rust, this strategy conflicts with Rule 2, it would not be clear which variable is the owner of the value. For this reason, in Rust the ownership of the value is passed to `var2` and `var1` becomes invalid. This process of transferring ownership is referred to as *move*. As in "ownership is moved to ...". In the last line the `into_bytes` method is called on `var1`. Thus, it would be used after it was moved to `var2`, resulting in an error. The first two rules ensure there is only one valid variable to the value at a time - the owner -, the variable's memory can be released by dropping it when it goes out of scope (Rule 3) without running into any dangling pointer errors [35].

```
let var1 = String::from("foo");    // var1 becomes owner of string
let var2 = var1;    // var1 is moved into var2
var1.into_bytes();    // ERROR: use of moved value: `var1`
```

Listing 2.7: Ownership of values and moving ownership

Ownership rules 1 and 2 establish a system of exclusive ownership, which entirely prevents multiple variables referencing the same value in memory (stack and heap). While allowing easy and safe memory management, this is very restrictive. For example, Listing 2.8 defines a function that prints out its string argument [34]. Consider the situation in Listing 2.8, a string value is assigned to a variable `var`, this string should get printed to standard output. It should be possible to still use `var` after calling `swallow` with it. However, the last line produces an error, since the ownership of the string value has been moved into the function and is dropped as soon as the scope of the function ends. Thus passing a value to function in this way always causes the value to be invalid afterward [35].

```
// Function takes ownership of its argument
fn swallow(s: String) {
    println!("{} has been swallowed", s);
}    // s is dropped here

let var = String::from("foo");    // var1 becomes owner of string
swallow(var);    // Ownership is moved into the swallow function
var.into_bytes();    //ERROR: use of moved value: 'var'
```

Listing 2.8: Functions and ownership

In order to gain more expressiveness but not lose its safety Rust incorporates references which allow to refer to a value without taking ownership. References enable aliasing with the following restrictions referred to as *borrowing* [35].

Borrow Rule. According to Steve Klabnik and Carol Nichols [35]
At any given time, it is possible to have either (but not both of)

- *one mutable reference or*
- *any number of immutable references.*

This allows the owner of a value to explicitly hand over references without giving up its ownership. Like regular variables, references (*borrows*) can either be immutable (default) or mutable. For a variable `v` of type `T`, the "*owned*" type, `&v` is a reference to `v` of type `&T`, the "borrowed" type. Accordingly, for mutable variable `m` of type `T`, the "*owned*" type, `&mut m` is a mutable reference to `m` of type `&mut T`, the "borrowed" type [35].

Listing 2.9 defines two functions. The first expects an argument of type `&String` which is an immutable reference type for strings and prints the received string to the terminal. The second function takes an argument of type `&mut String`, which is a mutable reference type for strings and appends an exclamation mark. As the rule for *borrowing* states there is only one active mutable reference allowed at a time. In particular, a mutable reference grants exclusive access to its resource permitting to read or mutate its value. For the duration of the mutable borrow the owner itself is not allowed to modify the value [34]. In contrast, Rust allows multiple immutable references, which grant read-only access, to exist at the same time. Immutable references can be copied without restrictions. Additionally, they can be used as source for new immutable borrows [42].

```
// Function borrows value
fn display(ref_s: &String) {
    println!("The string is: {}", ref_s);
}    // reference ref_s is dropped

// Function mutably borrows value
fn append(mut_ref_s: &mut String) {
    mut_ref_s.push('!');
}    // mutable reference mut_ref_s is dropped
```

Listing 2.9: Functions accepting borrowed values

As stated earlier exclusive ownership is very restrictive: The use of borrowing removes these restrictions. Listing 2.10 shows how the string `"foo"` is passed to the `append` function as mutable borrow, which mutates the string by adding an exclamation mark to the end of the string. Next, the modified value is passed to the `display` function as immutable reference which prints out the string. This is only possible because the mutable borrow used

in the `append` function becomes invalid at the end of the scope of the function. Otherwise, there would be a mutable reference and an immutable reference active at the same time which would result in an error according to the Borrow Rule. After invoking these two functions it is still possible to use the original owned variable `var`. The last character of the string (`'!'`) is removed by calling `pop`.

```
let mut var = String::from("foo");
append(&mut var); // pass as mutable reference
display(&var);    // pass as reference
var.pop();   // Use of var still valid
let ref_var = &var;    // Create reference and assign it to ref_var
let mut_ref_var = &mut var;   // ERROR: cannot borrow 'var' as mutable
                              // because it is also borrowed as immutable
display(ref_var);
```

Listing 2.10: Passing immutably and mutably borrowed values to function calls

The following lines of the example (Listing 2.10) demonstrate how the Borrow Rule prevents the simultaneous existence of mutable and immutable borrows. In line five, an immutable reference of the string is assigned to `ref_var`. Next, a mutable reference to the same string is stored in `mut_ref_var` and afterwards `display` is invoked with `ref_var`. However, already the previous line returns an error. The compiler detects that `ref_var` is needed afterwards for calling `display`, thus it is still active, while the mutable reference `mut_ref_var` is created. This contradicts the Borrow Rule, resulting in a compile-time error.

Borrows, just like regular variables, are dropped as soon as they go out of scope. What would happen if the owner variable of a value goes out of scope before its reference? According to Ownership Rule 3 the value is dropped when the owner goes out of scope. Due to the ownership rules, the compiler can determine when a value is used after it was released, resulting in an error. This can be seen in Listing 2.11, the first line creates an empty vector. In the new scope a string value is assigned to `var`, next a reference to `var` is stored in the vector. This line is where the error occurs. The compiler detects that `var`, the *owner* of the string value goes out of scope, releasing its memory, but the reference stored in `vect` is used afterward in the last line of the example. This would result in an use-after-free memory error. Instead, Rusts compiler throws an error, stating that `var` does not live long enough [35].

```
let mut vect = Vec::new();   // Create empty vector
{    // New scope begins
    let var = String::from("foo");   // Assign string to var
    vect.push(&var);    // ERROR: 'var' does not live long enough
}   // Scope ends, var is dropped
println!("{}", vect[0]);    // Use of reference after value was dropped
```

Listing 2.11: Use of dropped variables

During compilation Rust automatically checks the ownership and borrowing rules. As a consequence, Rust can eliminate common low-level development errors such as dangling pointers and use-after-free without the need for a garbage collector. Furthermore, these rules disallow the unrestricted combination of aliasing and mutation on the same memory location which is required for the occurrence of data races. Thus, Rust provides thread safety which makes Rust ideally suited for modern concurrent and parallel programs [34].

2.3 Concurrency and Parallelism in Rust

Rust was designed with concurrency in mind. The ownership concept eliminates not only common memory errors but also prevents many concurrency errors resulting from data races. These are usually hard to track down. Often errors only occur under specific circumstances, which are difficult to reproduce. By enforcing ownership and borrowing rules the compiler already detects many errors at compile time. For this reason, Rust is well suited for concurrent and parallel programming. However, it is important to note that concurrent programs still need to be designed with caution, as Rust does not eliminate all kinds of bugs [35].

The ownership concept enables Rust to disallow "[...] any unsynchronized, concurrent access to data involving a write.", i.e. data races. In doing so, it helps to avoid other forms of race conditions. For instance, in many situations different memory locations have to be updated atomically. Atomic operations are always performed without interruption. Even if these operations internally consist of multiple operations, execution can not switch to another thread before the atomic operation is finished. When memory locations have to be updated atomically, it is desired that other threads see either all of the changes or none of them. In Rust, granting access via mutable references (`&mut`) to several memory locations ensures updates are performed atomically. The Borrow rule guarantees no other threads have simultaneous read or write access [58].

Another essential feature of Rust which helps to avoid concurrency errors are "Send" and "Sync". Similar to the "Copy" marker they convey information to the compiler. The ownership of types marked with "Send" can be safely transmitted from one thread to another. Most of Rust standard types are "Send". An example for a type not marked as "Send" is `Rc<T>`, which is a reference to some type `T` with reference counting through regular reads and writes. `Rc<T>` is not marked as "Send" because if it is shared between threads, two threads might change the reference count simultaneously, resulting in potentially incorrect reference counts. To support this scenario, Rust provides `Arc<T>`, which uses atomic updates and therefore is "Send" [35].

Types marked as "Sync" can be safely referenced from multiple threads. This means any type `T` is "Sync" if its reference type `&T` can be transmitted safely between threads (is "Send"). Similarly to the "Copy" marker compound data types consisting entirely of "Sync" types are "Sync" themselves. The same holds for "Send". These markers provide additional thread-safety as the compiler will detect if types which are not "Send"or "Sync" are shared between or referenced from multiple threads [35].

Rust provides many features which help to develop safe and fast concurrent or parallel applications. Its philosophy is to catch as many bugs as possible during compilation, preventing many hours of tracing small, subtle errors. Especially for developers new to Rust the long list of compiler errors might be frustrating. However, when a program finally compiles, it is guaranteed to be memory safe and free of data races.

The standard library of Rust supports several approaches of introducing concurrency or parallelism, either by creating child process or by spawning new threads. There are different kinds of thread models. The approach of Rust is that it relies on the operating system for creating new threads. This model is often called *1:1*, because every spawned thread maps exactly to one operating system thread. The standard library of Rust takes this approach because it results in small binaries since thread management is outsourced to the operating system. Low-level languages such as Rust require small binaries as this makes it easier to combine them with other languages [35].

Spawning threads in Rust requires functions defined in the standard library such as the `std::thread::spawn` function shown in Listing 2.12. This function creates a thread, such that the new thread executes the *closure* provided as argument. In the case of the `spawn` function a *closure* with no arguments has to be provided [35].

```
let n = 24;     // Define new variable
let handle = std::thread::spawn(|| {     // Create new thread
    calc(n);                             // Closure body
    // Do some more                      // Closure body
});
handle.join().unwrap();     // Join the child thread
```

Listing 2.12: Spawning threads

When the execution reaches `std::thread::spawn` a new thread is created, which executes the task defined in the closure provided as argument. Additionally, the main thread continues to run and gets a `JoinHandle` [12] as a return value from the `spawn(...)` function call. Listing 2.12 shows how the return value of the `std::thread::spawn` function is assigned to a variable. This handle represents the *owned* connection to the newly created thread. Thus, when it goes out of scope, the connection is dropped, detaching the child thread from the main thread [35].

In order not to lose the connection to the child thread, the main thread has to join the child thread before the `JoinHandle` goes out of scope. Calling the `join` method on handle (last line of Listing 2.12), causes the execution of the main thread to block and to wait for the child thread to finish. The child thread terminates as soon as the execution of the closure is finished. In some cases, the provided closure is defined to perform a calculation which returns a value. This result is returned upon the child threads termination yielded by the `join` method [35].

In summary, calling `join` on the `JoinHandle` forces the main thread to wait for the child threads termination as well as returning the result value computed by the child thread. After `join` returns execution of the main thread will continue freely. In cases where the main thread terminates before the child thread finished execution, the child thread is cleaned up by the operating system. As a consequence, the task executed on the new thread might get interrupted, not being able to finish its task. For instance, when the child thread should calculate some data and write it to a file or send it to some other system, the value might be lost if the thread is interrupted before writing to the file or sending it. This is important to keep in mind as in Section 4.2 this will be the cause of difficulties. [12]

Rusts standard library offers basic support for concurrency such as the `spawn` function, Rusts growing ecosystem provides various first-class libraries supporting multiple models of parallelism. This work will focus on the following three major libraries:

- `crossbeam`
- `rayon`
- `timely`

2.4 Rust Infrastructure and Compiler

Rust, as noted earlier, is a compiled language, which means that Rust source code is translated into executable binaries using a compiler. Depending on platform, operating system and processor architecture, a specific version of the compiler has to be used. Some operating systems, such as Linux or BSD, provide package managers which ease the installation and management of software. However, these managers depend on central repositories which often do not provide the most up-to-date versions of the software. Instead, the Rust ecosystem includes the command line tool *rustup*, "the Rust toolchain installer" [9]. Rustup is the preferred way to install Rust on a system. With `rustup toolchain`

`install stable` it automatically installs the latest stable *toolchain* available for the platform, toolchain in this case refers to a single installation of the Rust compiler. Additionally, it is intended to ease the management of multiple compiler installations, providing specific commands for automatically updating installed compilers, removing installations and switching between them. Beyond that, it allows defining the default compiler installation to be used for a specific directory (`rustup override set nightly-2019-02-07`). In other words, when invoking the rust compiler in a directory, the compiler installation specified for this directory will be used. This is necessary when Rust packages require a specific compiler version for compilation. Beyond that Rustup provides many handy configuration options [9].

The rustup tool provides easy management of multiple Rust compiler installations. The Rust compiler itself is written in Rust and is called *rustc*. It is developed as open-source project together with the language itself on Github [11]. The compiler is invoked with `rustc`. When adding the path to a Rust source file, it will be compiled yielding an executable binary. Rustc provides various options in order to configure the compilation process, such as optimization level, the name of the resulting binary, or linking options. [8]

It is not often necessary to invoke the compiler directly, instead Rusts ecosystem provides a package manager, which is the recommended way for compiling projects. The package manager, called *cargo*, is automatically installed by rustup together with the compiler. This is the main tool Rust developer are confronted with. It provides short commands for scaffolding a basic Rust project (`cargo new <package name>`) as well as compiling (`cargo build`) and running (`cargo run`) it. Instead of passing multiple command-line options to *cargo*, it is configured by means of a configuration file (`Cargo.toml` file). This is where the name of the executable, the package version, dependencies and so on are specified. On the basis of this configuration file *cargo* automatically downloads the suitable version of the dependencies from the central Rust package repository located at `crates.io`, invokes the compiler with the right parameters ensuring all dependencies are compiled and linked correctly [35].

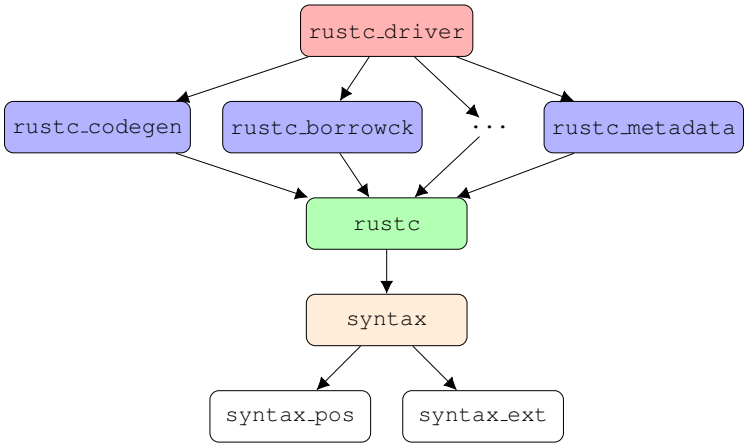

Figure 2.1: Dependency graph of the compilers internal packages [8]

The Framework developed in the course of this work uses an internal compiler interface. For this reason, the structure of the compiler is outlined here.

The Rust compiler is internally composed of multiple packages. Figure 2.1 shows some of these packages and their dependencies. The `rustc_driver` package contains the *compiler driver*, which is the "main" function of the compiler, it ties together the code provided by the other packages, defining the general flow of execution. The packages in the middle

of the dependency tree such as `rustc_borrowck`, `rustc_codegen` and `rustc_metadata` contain major parts of the compiler and export routines which are called from within the compiler driver. The `rustc` package defines common data structure, which are used by `rustc_borrowck` and `rustc_codegen`. These data structures include representations of syntax elements which are defined in the syntax package. The packages below `syntax` contain important routines for the parser [8].

Currently, there is a lot of work being done to transform the Rust compiler from fixed "pass-based" model into a demand driven query model. So far, *rustc* performs a fixed number of passes, each advancing transformation of the entire program by one step. This rigid structure is getting replaced. Regardless of what model is used the compiler has to perform seven steps to transform the provided source code into an executable binary [8].

1. *Parsing Input*: The Rust source files are parsed, creating the AST (abstract syntax tree). The AST is composed of recursive data structures which are intended to closely match Rusts syntax.

2. *Name Resolution, Macro Expansion, and Configuration*: The AST is walked recursively, resolving names and expanding macros.

3. *Lowering to HIR*: The AST is transformed into HIR (Higher-level intermediate representation). The HIR is an altered version of the AST optimized for the following analyses.

4. *Type checking, and subsequent analyses*: Determines types and assigns them to HIR nodes. Additionally type-dependent names, such as method calls are resolved. Subsequently privacy checking is performed.

5. *Lowering to MIR and Postprocessing*: The HIR is transformed into MIR (middle intermediate representation). MIR is optimized for ownership and borrow checking. After transformation, ownership and borrow analysis is performed.

6. *Translation to LLVM and LLVM Optimizations*: MIR is translated to LLVM yielding optimized object files.

7. *Linking*: Links the object files together returning one single executable [8].

The Rust compiler provides interfaces which allow developers to execute additional code between two steps. A detailed explanation of how this works is given in Chapter 4.

CHAPTER 3

Theoretical Foundations

In the course of this chapter, various foundation, needed for the design and the implementation of a parallel performance analysis framework, are explained.

Section 3.1 gives a detailed explanation of instrumentation and its different variants. Afterward, Section 3.2 outlines the architecture of parallel performance analysis tools. Section 3.3 describes three libraries for parallelism common in the HPC domain together with three popular Rust libraries for parallelism. Finally, Section 3.4 outlines three established parallel performance tools.

3.1 Instrumentation

Instrumentation is a technique commonly used to observe and analyze the performance of parallel programs. Instrumentation is the process of inserting additional instructions, often referred to as *probes* or *instrumentation calls*, into the source code. During the execution of an instrumented program these additional instructions trigger events, representing program execution state [55]. By collecting these performance events and by performing measurement on data associated with them developers can obtain valuable characteristics about the run-time behavior of the program [56].

Events can be of two different types. On the one hand *atomic events* together with their associated data represent an independent incident such as, the allocation of a certain size of system memory or the sending of a signal. These events mostly consist of a single instrumentation call. On the other hand *interval events* need at least two instructions and are usually represented by a begin and an end event. These are commonly inserted directly before and after regions of interest, such as performance critical routines or loops to measure the time spent in the enclosed region. [55].

Commonly, instrumentation not only inserts the probes which trigger the aforementioned performance events but also requires some instructions for initialization. Initialization may include the creation of a new process or thread to calculate data associated with performance events. In other instances, data structures might need to be initialized which carry along data depending on multiple performance events. Additionally, instructions for persistently storing the occurring performance events together with their related data might need to be inserted. It might also be required to insert finalization instructions at the

end of the program to clean up data structures and join the created threads or processes [48].

It is important to keep in mind that adding instructions always alters the program to some extent. In order to provide actual help to the developer, instrumentation has to meet certain requirements.

Requirement 1 *The instrumented program must be executable.* Instrumentation does not provide any use if the program under observation cannot be executed. Therefore it is essential that the instrumentation calls themselves are syntactically correct. Furthermore, they have to be inserted into the original code in a way that again results in correct syntax. As instrumentation can be introduced at various stages of the compilation process "syntax" here means the appropriate representation associated with the particular level of source transformation. Accordingly, probes might have to be valid C/C++ or Rust syntax when inserted early on in the process or valid object code when introduced later on [48].

Requirement 2 *The instrumented program must semantically match the original.* It is only possible to draw meaningful conclusions regarding the original program based on the instrumented version when both yield the same results. Thus it is important for the insertions not to alter the programs semantic [48].

Requirement 3 *The execution behaviour of the instrumented program must be as close as possible to the behavior of the original.* It becomes more difficult for developers to draw a meaningful conclusion regarding the run-time behavior of the original program the more its run-time behavior differs from the instrumented version. Introducing additional instructions generally alters the execution time of the program. In particular, the overhead generated by initialization at application start-up, the calculation of data associated with each event, the collection and storage of the event data, as well as the finalization process can cause notable longer execution time. Therefore, the performance analysis system as a whole and especially the instrumentation calls have to be designed to introduce the least possible amount of overhead. Besides the insertion of probes may result in increased compilation time. However, compilation time is less of a concern as the developer is interested in run-time behavior. Still, significantly longer compilation times can prolong development and can be annoying. Thus, keeping compile time within a reasonable margin should not be disregarded completely [48].

In other words, it is crucial that instrumentation alters the program and thus its behavior as least as possible.

It is often difficult to measure the actions of interest directly. The instructions the developer is interested in are often interlaced with other instructions. Consider, the situation `interest_call( func1(), func2() )`. The performance of the function call `interest_call` should be measured but its arguments are calculated inside the call. When inserting probes before and after `interest_call` the time measured includes the time needed to calculate `func1()` and `func2()`. In order to accurately measure such interlaced instructions it is necessary to isolate them which can be challenging. Moreover, isolating these instructions might require significant changes to the program and its structure, which may result in severely altered run-time behavior. As a consequence instrumentation is a balancing act between retrieving detailed measurement which relates poorly to the original program and receiving less detailed data which precisely relates to the original.

Instrumentation calls can be inserted at different stages of the compilation process. At different levels of source-code transformation the available syntactic and semantic information varies. In the following, these different approaches together with advantages and

disadvantages of inserting probes at different stages of the compilation process are described in more detail. [56]

3.1.1 Source-based Instrumentation

In case of source-based instrumentation the developer manually annotates lines of source code with additional instrumentation instructions. These instructions have to be part of an interface provided by a profiling framework and the programming language in question has to be supported by the framework. As an example, Listing 3.1 shows how source-based instrumentation of some function "`foo`" looks like, when instrumented using the Score-P performance measurement infrastructure [37], a framework combining multiple common profiling tools. In particular, the "`SCOREP_USER_FUNC_BEGIN`" instrumentation call indicates the beginning and "`SCOREP_USER_FUNC_END()`" the end of an *interval event*. The two inserted instructions enclose the region of interest which is the body of the "foo" function. Also worth noting is the include statement at the top. This brings the probe functions into scope and is typical for source-based instrumentation [37].

```
#include <scorep/SCOREP_User.h>

void foo()
{
    SCOREP_USER_FUNC_BEGIN()

    // original function body...

    SCOREP_USER_FUNC_END()
}
```

Listing 3.1: Manual instrumentation of a function with the Score-P performance measurement infrastructure. [16]

Source-based instrumentation offers the greatest amount of flexibility, granting the developer full control over which parts of the program should be observed. The developer knows the semantics of the source code in contrast to automated tools. Therefore he knows best which parts of the program need instrumentation. Nevertheless, manual annotation can be very tedious in large projects [56].

3.1.2 Preprocessor-based Instrumentation

Preprocessors are used to alter the source code of a program before compilation. With preprocessor-based instrumentation, a preprocessor automatically infers where to insert instrumentation calls based on predefined regions of interest. After identifying source-code regions which need instrumentation, the preprocessor inserts the appropriate probe instructions. For example, the memory allocation/deallocation tracking package of TAU detects invocations of the `malloc` and `free` routines and redirects them to memory wrapper calls. The wrapper calls add information such as source-code file and line number. As a result, users can track the size of allocated memory during execution [56].

On the one hand, this automated approach reduces flexibility compared to manual insertion. On the other hand, it takes much work off the hands of the developer. In some cases the preprocessor has access to dependencies which allows for inserting instrumentation calls into the source code of libraries or other dependencies [56].

3.1.3 Compiler-based Instrumentation

With compiler-based instrumentation, additional instructions are inserted directly into the object code generated by the compiler. Therefore, modern compilers used in the context of high-performance programming (HPC) often provide extra flags which permit instrumentation of entry and exit points of functions and methods. This approach has a substantial advantage, as the compiler knows the entire mapping of source code to memory locations including dependencies. Furthermore, the compiler has control over large parts of the source-code transformation. This is especially important, since manually inserted probes may conflict with the code optimization done by compilers. In some cases the compiler may remove instrumentation calls during the optimization process, in other cases the probes may prevent the compiler from performing certain optimizations. However, since in this approach the compiler is aware of the additional instructions it can prevent them from being removed while maintaining aggressive source code optimization [56].

3.1.4 Wrapper Library-based Instrumentation

This approach uses special libraries which already contain the necessary instrumentation calls. During compilation the original library is substituted by its corresponding instrumented version. For this reason, the wrapper-library provides a superset of the interface of the original library. Usually, the routines declared in the wrapper-library call the routines of the original library and add the necessary instrumentation calls. In some cases, the original library provides a special interface that allows for intercepting the calls, that the instrumentation calls can be inserted. Furthermore, this grants the wrapper library access to the arguments passed to the routine call. This approach only provides instrumentation for the linked libraries and not for the developed program itself. This approach allows applying instrumentation to a program even though no source-code is available. Also, it is ensured to run relatively stable since the instrumented libraries are well tested [56].

3.1.5 Binary Instrumentation

This approach inserts instrumentation instructions directly into an executable binary after the compiler generated it. Therefore, the executable image can either be rewritten with the added instrumentation calls using a binary rewrite tool [55] or probes can be dynamically inserted into a running program. This enables the insertion of probes for interval events, such as routine and loop entry and exit. Especially the second aspect enables the developer not to have to decide before compilation what routines and loops should be instrumented. Often, only execution reveals which parts of the program are bottlenecks and need further analysis. Thus it is possible to immediately react to the run-time behavior of the program without the need to recompile the entire program, resulting in a smoother development process. However, with this approach it is more challenging to relate specific instruction to source-code constructs [56].

3.2 Architecture of Parallel Performance Tools

Tools used to debug or analyze the performance of parallel applications, often referred to as *parallel performance tools*, originated in the context of high-performance computing (HPC). As HPC is concerned with maximum performance, it early on relied on parallel applications in order to take advantage of increasingly parallel hardware. For the purpose of analyzing parallel programs, performance tools use instrumentation to obtain detailed event data during program execution. In particular, this data contains information about the program state at the time of the event. The performance tool can either store the event

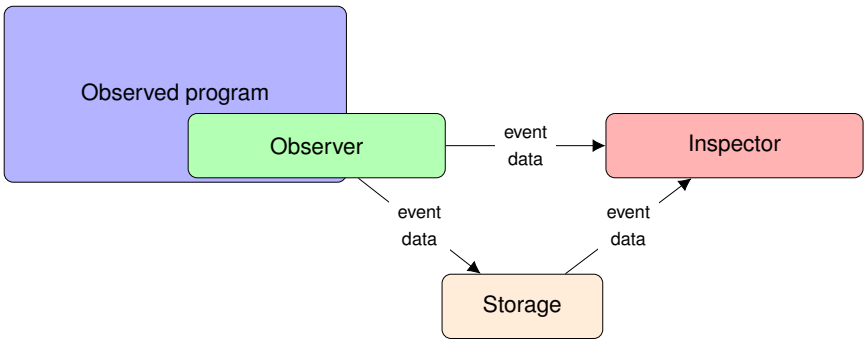

Figure 3.1: Architecture of parallel performance tools

data for later inspection or analyze it right away. The process of storing data and analyzing it after the programs execution is called *offline* or *post mortem* analysis while analyzing the generated data during run-time is referred to as *online* or *live* analysis. Modern parallel performance tools often support both approaches by providing two interfaces, one for storage and one for immediate analysis [37].

Regardless of whether performance tools support online or offline analysis or both they can be differentiated by what kind of event data they generate. *Profiling* tools (*profiler*) aggregate event data during execution. Thus they produce statistics and do not record event timings. Whereas, *tracing* tools (*tracer*) emit the raw event data, including time stamps. However, especially when scaling up the number of threads or processes *tracing* tools can become unusable, as the amount of generated event data may easily exceed storage capacity or the processing capabilities of the online viewing component, reducing its responsiveness to a point where it becomes unusable. Since *profiling* tools only output aggregated data storage and processing capacity is not that big a concern, however in the course of computing statistics information is lost. In particular, the absolute chronology of events is lost due to aggregation of data [39].

Even though profiling and tracing tools process event data differently, the general architecture of parallel performance tools is similar. All of them contain two major components, the *observer* and the *inspector* [33], which are described in the following two sections.

3.2.1 Observer

The *observer* (or *monitor*) component is responsible for monitoring run-time behavior of a program and emitting data about its state. The most common technique to gain insights about the internal behavior of a program during execution is instrumentation. The inserted instruction calls often initialize a run-time environment, which is responsible for computing additional data associated with the performance events or keeping track of data structures used of aggregation, such as the number of times a certain function was called. Additionally, the environment commonly manages the storage of the event data for post mortem analysis or the data transmission for online inspection, as shown in Figure 3.1. These actions are often delegated to a separate process. The inserted probes together with the run-time environment make up the *observer* [38].

Ideally, the behavior of an uninstrumented program matches the behavior of its instrumented counterpart. However, the *observer* is interconnected with the observed program through instrumentation, which is illustrated in Figure 3.1 by the observer node overlapping the node of the observed program. For this reason, not only the instrumentation calls themselves but also any actions performed by the run-time environment such as simple updates of data structures, alter the programs run-time behavior. Even if a large propor-

tion of the computations the observer are performed in separate processes they occupy system resources including CPU time and system memory. If the run-time behaviour of an instrumented program is very different from its uninstrumented counterpart it is difficult to relate observation results to the original program and identify bottlenecks and errors in it. For this reason, it is a critical requirement for the *observer* to generate the least amount of overhead possible [38].

3.2.2 Inspector

The *inspector* component is responsible for the preparation and appropriate presentation of the event data to enable the user to evaluate the monitored data conveniently. In particular, the *inspector* is typically a separate process, which may be running on a different physical machine. Depending on the capabilities of the inspector it can either receive event data directly from the *observer* for online viewing or read event data from storage for *offline* inspection, as shown in Figure 3.1. In the case of online analysis the *observer* forwards the data via some defined interface directly to the inspector. However, the communication between monitor and the presentation component generates additional overhead, which further alters the observed programs run-time behavior. In contrast, the storing of event data generally produces less overhead. Moreover, it allows for computational intensive analysis of the event data, compared to online examination where the *inspector* is required to process the incoming data in real-time [38].

3.3 Parallel Libraries and Interfaces

There exist several well-established libraries for developing parallel applications. These standards and libraries were mainly utilized in applications requiring high performance. However, it is also possible to apply them in less performance critical areas. In the following, the three most common frameworks are introduced and compared to the libraries proposes for parallel computing in Rust.

3.3.1 MPI

The *Message Passing Interface* (*MPI*) is most well known in the context of parallel computing. Its goal is to provide a common standard for developing message-passing applications. For this reason, MPI specifies a library interface and does not provide a library itself [45]. Accordingly, multiple implementations exist with open source versions such as OpenMPI and MVAPICH as well as implementations from hardware vendors optimized for the use on specific hardware such as Intel MPI and IBM BG/Q MPI. MPI is designed to be language independent, yet only implementations in C/C++ and Fortran are officially supported. As its name suggests, MPI proposes the message-passing paradigm, which was originally designed for distributed memory architectures popular during the time of its first release (80s early 90s) [21].

Message-Passing Programming Model In applications implementing the *message-passing programming model* a set of processes exchange data through sending and receiving messages. Figure 3.2 illustrates this concept on the basis of three processes. It is important to note that the communicating processes can reside on different physical machines. In addition, these cooperative operations can be used for synchronization. For instance, a send performed in one process requires a matching receive operation in another process [20].

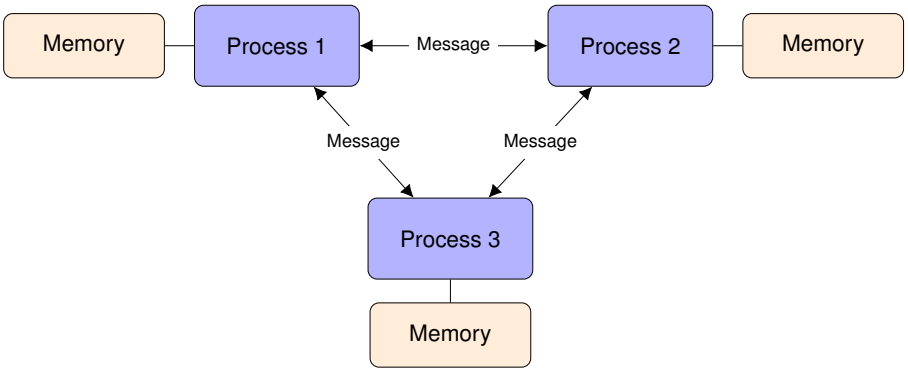

Figure 3.2: Message passing programming model

3.3.2 Crossbeam (Rust)

There are wrapper libraries such as `rsmpi`, which provide Rust bindings to common MPI implementations. However, these wrapper libraries currently only support a subset of the official MPI specification. In some cases, this might be appropriate but when programming in Rust it might be desirable to take full advantage of its safety guarantees by using native Rust libraries. When trying to implement the message passing paradigm in Rust, the most common library to use is `crossbeam`. Crossbeam provides low-level concurrency and parallelism utilities, such as atomics, concurrent data structures, and synchronization primitives [31].

One of its greatest features is its high-performance implementation of channels. Channels are a technique for transmitting data between entities such as threads and processes. They are ideal for implementing message passing between processes. Crossbeam provides several channel variants, all of them can have one sender and one receiver, but multiple sender and receiver are possible as well. The sender can send data through the channel to the receiver, as shown in Listing 3.2. In the case of bounded channels, the sender can only send data if the number of messages within the channel (messages, which were sent but not yet received) is less than the capacity of the channel, otherwise the sender blocks. The channel provided by Crossbeam enable easy synchronization and fast communication between multiple threads or processes [2]. Crossbeam also provides atomics, data structures and synchronization primitives for the development of applications implementing the shared memory or thread paradigm [2].

```
// Creates a channel capacity 5 and
// assigns sending and receiving end to variables
let (sender, receiver) = bounded(5);
// Sends 'Hello'
sender.send("Hello");
// Checks if 'Hello' is received
assert_eq!(receiver.recv(), Ok("Hello"));
```

Listing 3.2: Sending and receiving with Crossbeams channels

Shared Memory Programming Model According to the *shared memory programming model*, multiple execution units have access to a common address space. Data is exchanged through reading from and writing to a shared memory region. This is illustrated in Figure 3.3. In order to coordinate access of multiple threads to shared memory, synchronization mechanisms have to be used. Consequently, in this programming

model execution units work on the same data thus it is shared implicitly, whereas in the message passing model data has to be exchanged explicitly [20].

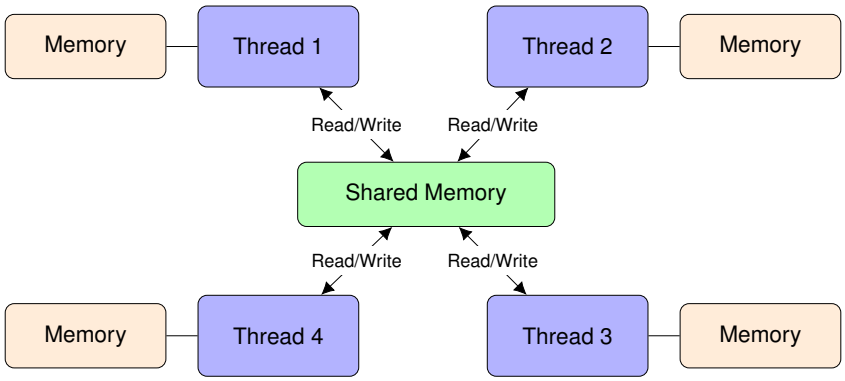

Figure 3.3: Shared memory programming model

3.3.3 POSIX-Threads

Historically, thread implementations from different vendors varied widely, hence developing portable applications which implemented the aforementioned thread programming paradigm was difficult. For this reason, an interface specification for thread management as part of the IEEE POSIX 1003.1c standard was introduced in 1995. Implementations of this standard are referred to as *POSIX* (Portable Operating System Interface for Unix) *Threads* or just *pthreads* [22]. This specification defines low-level data structures and procedures for the C language, allowing synchronization on Unix systems through mutexes and condition variables. Pthreads facilitate the development of portable applications in environments where multiple threads can access a common memory region. For this reason, Pthreads gained popularity with the increasing core count of processors [46].

3.3.4 OpenMP

Another widespread standard used for the development of threaded applications is *Open Multi-Processing* (*OpenMP*). Similar to MPI it is a specification, not an actual library. It specifies directives which can be used by developers. Compared to pthreads the goal of OpenMP is to provide a high-level easy-to-use mechanism of writing new high-performance applications as well as incrementally parallelizing existing serial programs. Therefore, OpenMP specifies certain compiler directives and callable dynamic library procedures. It was originally targeted towards Fortran, but now C and C++ are supported as well. OpenMP follows the fork-join paradigm [26].

Fork-Join Programming Model According to the *fork-join programming model*, every program starts with a main thread. This main thread is colored red in Figure 3.4. It is executed serially until a parallel region is reached. These are program regions which benefit from parallelism. When the execution arrives at such a region the control flow forks and additional threads are spawned such that each thread including the main thread can perform independent computations. When the parallel region ends, the threads are joined and only the main thread resumes execution serially until the next parallel region occurs or the program terminates [53].

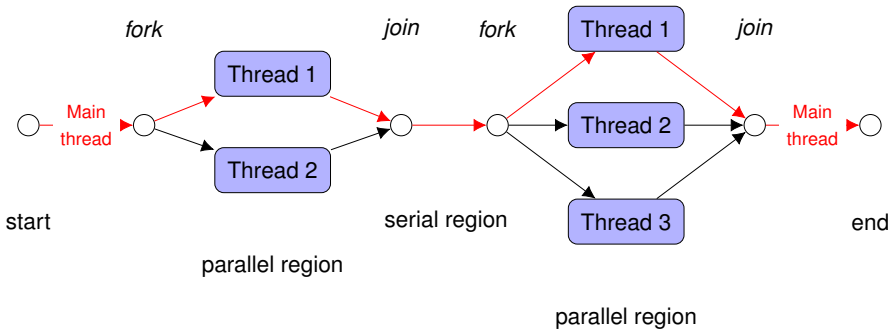

Figure 3.4: Fork-Join programming model

3.3.5 Rayon (Rust)

Rust has a library specifically dedicated to *data parallelism*.

Data Parallel Programming Model The *data parallel programming model* does not make any assumptions about the underlying memory architectures. The message-passing and shared-memory models can be summarized as *task parallelism*, as they both describe how different tasks can be performed simultaneously on multiple threads/processes. In contrast, *data parallelism* proposes data should be distributed among multiple execution units such that each unit gets a portion. Importantly each process/thread performs the same action on its portion of the data. This is illustrated in Figure 3.5 [20].

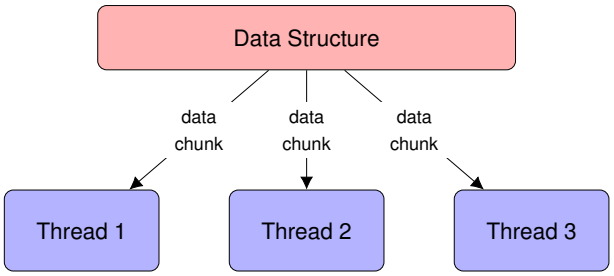

Figure 3.5: Data parallel programming model

The Rust library `rayon` tries to make converting sequential applications into parallel ones easy while guaranteeing the absence of data races. Accordingly, Rayon provides its own thread pool implementation. Developers will not have to deal with setting up a thread pool, this is handled by Rayon. Maintaining a thread pool allows Rayon to quickly assign tasks to the different threads, without having to spawn multiple threads every time parallel executions is desired [3]. The central mechanism of the library for defining tasks which can run in parallel is the `join` function, shown in Listing 3.3. It takes two closures which are potentially executed on the thread pool in parallel. "Potentially" in this case means, Rayon checks during run-time whether processor cores are idle or not. If no idle cores are available, the two tasks are processed sequentially. The idea is similar to OpenMPs parallel regions within the source code calls to `join` indicate which program regions might benefit from parallelism. However, instead of always executing the annotated code in parallel Rayon dynamically decides if it makes sense to perform actions in parallel depending on the available resources [41].

```
// May run task_a and task_b in parallel, returns the result of each task
let (result_a, result_b) = join(|| task_a(), || task_b());
```

Listing 3.3: Rayons join function

```
// Computes total price in euro from multiple dollar prices
let total_euro_price = dollar_prices
    .par_iter()    // Indicates parallel execution possible
    .map(|price| dollar2euro(price))    // May run in parallel
    .sum();    // After parallel execution results are added up
```

Listing 3.4: Rayons parallel iterators

In addition, rayon provides parallel iterators which are convenient wrappers around the `join` function. Parallel iterators allow iterating over data structures so that the specified operation may be performed in parallel on different parts of the data. Listing 3.4 shows an example of the usage of parallel iterators. It calculates the total price in euro from a list of prices in dollar. Rayon not just assigns each thread an equal amount of the data and waits until all threads are finished. Instead, it uses a technique known as *work stealing*. The basic idea is, each thread has its own queue of pending tasks, processing pending tasks on after another until the queue is empty. If one thread finishes its queue earlier it "steals" tasks form the queue of another thread. As a result, no threads are idling and it takes less time to process the entire data structure. Rayon provides various kinds of parallel iterators which all use this technique and enables developers to implement data parallelism in new applications easily [41].

3.3.6 Timely Dataflow (Rust)

While Crossbeam and Rayon allow for easy implementation of common parallel programming models, namely message passing, thread and data parallelism, the library `timely` is the Rust implementation of a new programming model called *Timely Dataflow* [49]. This is a specific version of *dataflow programming*. In recent years, the dataflow model has regained popularity as it is capable of representing parallelism very efficiently [29].

Dataflow Programming Model According to this model, applications are represented as a directed graph. A node corresponds to operations such as addition or subtraction and edges describe the flow of data within the graph. Thus, data arrives on incoming edges and is processed according to the operation defined by the node, and the result is sent on outgoing edges. This is illustrated by Figure 3.6. The data flow model has two properties. First, nodes may perform their operations in parallel, except for one node explicitly depending on the result of another one. Second, results do not depend on the relative ordering in which potentially parallel nodes are executed [25].

This model does not provide an opportunity for nodes to maintain some internal state. To avoid this restriction, Timely Dataflow extends the standard model with timestamps, which correspond to logical points in the computation. As a result Timely dataflow allows stateful dataflow vertices [49].

The central data types of the Timely library are streams which model dataflow. For initialization Timely provides the `execute*` functions which start the dataflow computation on multiple worker threads. All of the `execute*` functions take a closure which defines what operations the workers perform. Moreover, Timely comes with its own communications library which seamlessly handles the transfer of data between workers possibly

Programming model	Library	Supported prog. languages
Message passing	MPI	Fortran, C/C++
Message passing, shared memory/thread	Crossbeam	Rust
Shared memory/thread	Pthreads	C, Rust (spawn/join)
Fork-join	OpenMP	Fortran, C/C++
Data parallel	Rayon	Rust
Timely dataflow	Timely	Rust

Table 3.1: Summary of the presented libraries

spread across multiple physical machines. As a result, it enables developers to build applications which run a single computer as well as large clusters. Besides, the fact that large computations can easily be split up into smaller operations (nodes) Timely allows incremental processing of large amounts of data, which does not entirely fit into memory [44].

Table 3.1 summarizes the six presented libraries listing their supported programming languages and their suggested programming models.

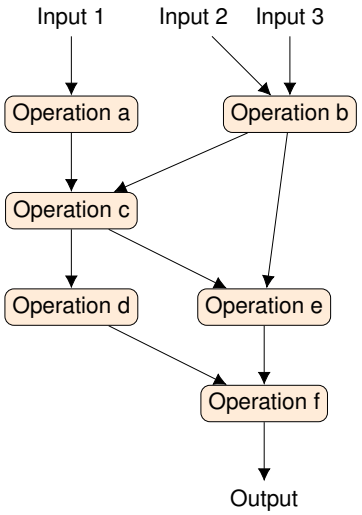

Figure 3.6: Dataflow programming model

3.4 Existing Tools

So far, Section 3.1 explained the most common techniques for extracting internal event data during program execution and Section 3.2 described the general structure of parallel performance tools. This section will present three common tools in detail.

3.4.1 TAU Parallel Performance System

The University of Oregon developed the TAU (Tuning and Analysis Utilities) Parallel Performance System [56] in cooperation with the Research Centre Juelich and the Los Alamos National Laboratory. TAU provides an extensive toolset for instrumentation, measurement, analysis, and visualization of parallel applications. It aims at overcoming portability issues

of other frameworks which force developers to apply different performance tools for different systems. For this reason, TAU offers a broad range of instrumentation mechanisms, allowing users to select those instrumentation methods which are best suited for their environment [56].

The TAU performance tool is organized in three layers, instrumentation, measurement, and analysis. Each layer contains several modules allowing users to configure TAU according to their needs by selecting the appropriate modules. For instrumentation, there are modules available such that all the instrumentation approaches explained in Section 3.1 are included, furthermore adding support for interpreters, component-based software, virtual machines such as Java virtual machine (JVM), combining multiple instrumentation methods, and explicit selection of performance events based on inclusion and exclusion lists. The instrumentation layer communicates with the measurement layer through the TAU measurement API [56].

The measurement component specifies how performance events are handled. Users can choose between two measurement modes: tracing or profiling mode. Depending on the selected measurement form TAU can collect different performance data. The measurement layer is also responsible for adjusting the measurement environment to the underlying system. In particular, TAU determines how many processes it can spawn and how much memory it can occupy without significantly altering the observed applications execution behavior [56].

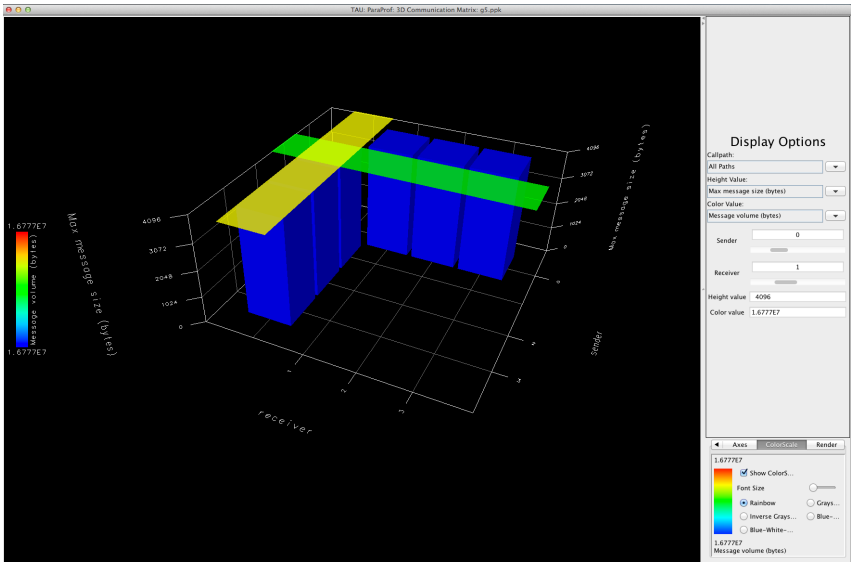

Figure 3.7: ParaProf: 3-D communication matrix view [27]

The analysis and visualization layer offers modules for working with profiling or tracing data. In order to analyze profiling data, TAU comes with the ParaProf tool [23], which provides visualization and analysis capabilities such as the 3-D communication matrix shown in Figure 3.7. This visualization of sender, receiver, and message size allows users to identify communication bottlenecks [27]. Since the focus of TAU lies on portable techniques to obtain performance data and given the fact that there are many tracing analysis tools available TAU does not include its own. Instead, it just provides modules for the conversion of trace data structures allowing the connection of other tools such as Vampir, which is described in the following section [56].

3.4.2 Vampir

Vampir was introduced in 1996 [50] as a visualization environment for the performance analysis of parallel applications. Currently, the Center for Information Services and High-Performance Computing (ZIH) of TU Dresden is responsible for its development. Vampir comes in two versions, Vampir for the use on a single workstation, and VampirServer a distributed variant providing better scalability [48].

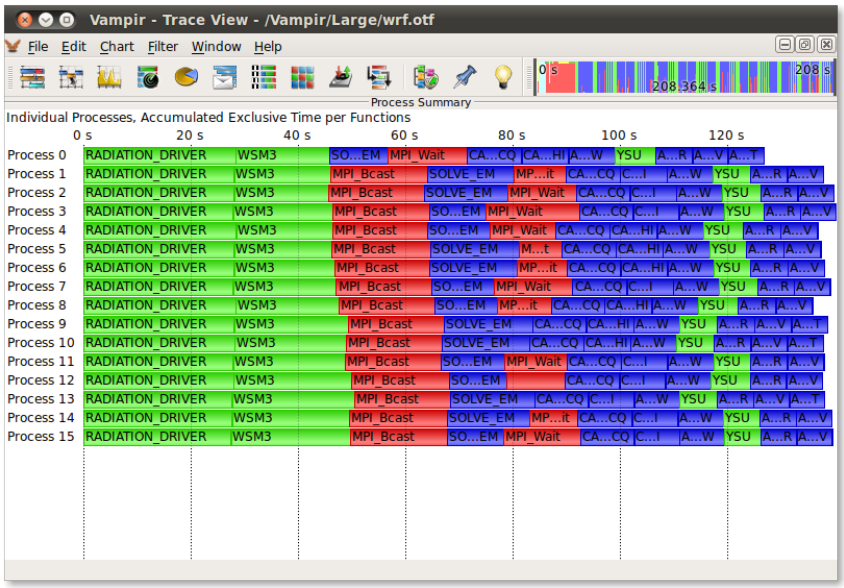

Figure 3.8: Vampir: process summary view [17]

Vampir itself does not include components for producing event data such as instrumentation. Instead, it is intended to be used for post mortem analysis. For this reason, it relies on external tools to store the event data obtained during measurement runs. The recommended framework for instrumentation and event data storage is Score-P, which is described in Section 3.4.3. Vampir supports the import of several trace file formats such as Open Trace Format [36, 28] or EPILOG Trace Format [59]. After loading trace files, Vampir offers various chart- and timeline views. Users can view the different charts based on groups of processes as well as based on individual processes. For instance, the screenshot in Figure 3.8 shows the process summary. It displays the execution time of every function for each process individually. This view may help to discover that one process takes considerably more time to complete a particular calculation. This may indicate the uneven distribution of work among the processes, which might be the cause of a performance bottleneck [17].

While Vimpir targets the analysis of smaller programs, VampirServer aims to provide insight into large scale parallel applications. It utilizes a client/server architecture. The server is a parallel application, which itself uses MPI, pthreads, and sockets for communication, in order to speed up the complex computations performed during the data preparation process. As the server may run on a cluster, it is possible to store the entire trace data in distributed memory. As a result, it is not necessary to copy many data to analyze extensive trace data collections. The client application is connected to the server. It is responsible for visualizing the prepared performance data for analysis by the user. Since the server component renders the visualizations, also low spec workstations and laptops can run the client [48].

3.4.3 Score-P performance measurement infrastructure

The Score-P performance measurement infrastructure [37] is developed in cooperation by the developers of the tools Periscope [24], Scalasca [30], TAU [56], and Vampir [48]. It addresses the fact that multiple performance tools provide different specialized features, and users often utilize multiple tools to take advantage of these specific features. However, many performance tools build on similar base functionalities. To ease the application of several tools and to reduce the redundancy of developing base functionality for each tool development effort was combined. Score-P provides base functionalities such as an instrumentation framework, run-time libraries, and some helper tools [37].

The instrumentation component supports the insertion of probes into programs written in Fortran, C, and C++. It comes with several specific run-time libraries for different programming paradigms. In order to collect event data appropriate for the given programming paradigm Score-P links the application against the according run-time library [37]. Running `scorep mpicc -c myapp.c` in a terminal instruments and compiles `myapp.c`. The `scorep` command is prefixed to the regular compile and link commands. Before the actual build is performed the parallel programming library is automatically detected (MPI in this case), and the corresponding flags are added. By passing options to `scorep` it is possible to configure the desired instrumentation approach. Score-P provides several mechanisms including compiler instrumentation, MPI wrapper-library, OpenMP source code instrumentation, general source code instrumentation through TAU, and by employing other external components pthread and OpenCL instrumentation and CUDA instrumentation [16].

Executing the instrumented application generates performance data. Score-P provides three different interfaces for further processing of the data. In tracing mode, event data is saved to files in the Open Trace format 2 [28] for later inspection using Vampir or Scalasca. In profiling mode, data is stored in special profiling formats which users can view with Scalasca or TAU. Periscope can query measurement information during execution via the online interface when the instrumented application is run with online mode enabled [16].

CHAPTER 4

Framework Design

The language Rust and its ecosystem evolve quickly. As outlined in Chapter 2 Rust provides many features which help to design reliable, high-performance parallel applications. However, the language and its ecosystem are very young, compared to the languages established in the domain of HPC and parallel computing namely Fortran, C, and C++. The existing parallel performance tools were built around these languages and their ecosystems. They offer various instrumentation approaches, allowing users to select the solution best suited for their program and environment. This makes the analysis of parallel applications built with Fortran, C/C++, MPI, or OpenMP as convenient as possible.

Even though using existing performance tools to analyze Rust programs is possible, their benefit is quite limited. As noted earlier wrapper libraries for MPI exist, allowing the recording of MPI calls via wrapper-library instrumentation. Thus, only MPI calls can be observed but not other properties of the program written in Rust. Moreover, only those Rust applications which use MPI can be analyzed. Native Rust libraries are not supported. However, using Rust libraries might be desirable in order to leverage the safety guarantees of Rust. Another important aspect is that the existing performance tools do not fit well with Rust's own ecosystem. Such debugging tools are intended to help developers building high-performance, reliable applications. Therefore, it is essential that using them does not require tedious setup and configuration, which complicates development. For these reasons, the tRust framework is designed and built in the course of this work. In particular, the following requirements were set:

1. *The instrumentation of programs written entirely in Rust must be supported.* In particular, this should be possible without using any other tools and libraries such as MPI or OpenMP.

2. *The instrumentation of dependencies must be possible.* Rust comes with many useful libraries. They often provide high-level wrappers around low-level mechanisms, which allow developers to focus on high-level application design. In order to get detailed insight into the run-time behavior of the program, it is necessary to instrument libraries. tRust should support the three popular Rust libraries for parallel programming: Crossbeam, Rayon, and Timely Dataflow.

3. *It must be easily usable together with existing Rust tools such as Rustup and Cargo.* Building reliable, highly parallel applications is challenging. Debugging tools are intended to

support developers in building such software, therefore it is essential that their setup and configuration does not introduce additional difficulties.

4. *It needs to be extendable.* As Rust and its libraries are young, they still may evolve. Interfaces may be modified or internals may be restructured. Therefore, the framework must be adjustable to support additions and changes.

The following section will outline the instrumentation approaches investigated. Section 4.2 will describe the structure of the run-time environment (observer). Afterward, in Section 4.3 the syntax of the instrumented code is explained in detail. Finally, Section 4.4 describes how event data is collected and stored.

4.1 Technical Implementations of Rust Instrumentation

The tRust framework itself is implemented in Rust. This has multiple advantages. First of all, it simplifies interaction with the observed application. For instance, writing the framework in Rust allows using Rust specific types and data structure for the event data. This enables the observer component to natively process Rust types. Using Rust data types for event data makes sense because it allows for efficient transferring of the data. The event data is produced by a Rust program (application under observation). Thus the event data can be sent directly to the observer component as Rust data structures without needing to convert the data to a common format. This is important, because it keeps the overhead low.

Another reason to implement the framework in Rust is that it allows for convenient use, because the framework would nicely blend into the Rust ecosystem. For example, if it is implemented as a Rust package it could be downloaded and built automatically using Cargo. Consequently, Rust developers do not need to familiarize themselves with new tools and they can use the tools they already know.

A further reason is, implementing the framework in Rust allows to leverage the safety guarantees provided by Rust. Implementing such a framework requires concurrent and parallel programming. The ownership concept together with the other safety features presented in Chapter 2 can help to build an efficient and reliable framework.

One significant difficulty in the design of a performance analysis framework is to provide an instrumentation mechanism for Rust programs which is not as tedious as manual source-based instrumentation. There are two feasible approaches, either recording only the invocations of library routines through wrapper library-based instrumentation or using an automated mechanism for inserting instrumentation calls such as compiler-based instrumentation. However, one goal for this framework was to not only instrument the application itself but also to instrument the dependencies of the observed application conveniently. In the case of source-based instrumentation, this would mean that users would first have to download the source code of the dependencies which should be instrumented. Second, they would have to look through the code and manually annotate the necessary code regions. Third, they would have to compile all the instrumented packages individually. This whole process would be error-prone and take time. Therefore, this approach is cumbersome. Consequently, the instrumentation of dependencies has to be done either in advance or automatically. In other words, the framework provides either already instrumented wrapper-libraries or a mechanism which automatically inserts instrumentation into a projects dependencies. Thus these two options, wrapper-libraries or automated insertion, enable efficiently adding instrumentation calls to both the application and its dependencies. The following Section explains these two approaches.

4.1.1 Wrapper Libraries

This framework aims to support Crossbeam, Rayon, and Timely Dataflow. Therefore, the framework has to provide instrumented versions for these three libraries. Each wrapper-library has to offer a superset of the interface of the original library. In other words, every function method and data structure exposed by the original is implemented in the wrapper-library. Furthermore, it is essential that the function/method signatures of the instrumented and the original version are identical. The functions defined in the wrapper library call the original function passing on the provided arguments. Additionally, before and after the call to the original function instrumentation calls are inserted.

```
// Imports original function and data structures
// with different name to prevent name collision.
use crossbeam::channel::bounded as orig_bounded;
use crossbeam::channel::Sender as Orig_sender;
use crossbeam::channel::Receiver as Orig_receiver;

// Instrumented version
fn bounded<T>(cap: usize) -> (Orig_sender<T>, Orig_receiver<T>) {
    instrumentation_begin();
    let return_val = orig_bounded(cap);    // call to original function
    instrumentation_end();
    return_val
}

// ------------------------------------------------------------------
// Original version in crossbeam
fn bounded<T>(cap: usize) -> (Sender<T>, Receiver<T>) {
    // function body
}
```

Listing 4.1: Definition of a function in a wrapper-library

Listing 4.1 shows how an instrumented function would look like for the creation of a crossbeam channel. The import statement `use crossbeam::channel::bounded as orig_bounded;` brings the original `bounded` method from Crossbeam into scope with a different name. The new name `orig_bounded` is specified after the keyword `as`. This is necessary to avoid name collision of the instrumented version and the original one. After the imports, the new wrapper-function is defined with the same signature as the original (shown at the bottom of the listing). The wrapper-function calls the original function and returns its result performing instrumentation calls in between.

```
// Import of the wrapper-library version
use crossbeaminst::bounded;
// Import of the original version
// use crossbeam::channel::bounded;
```

Listing 4.2: Import of a wrapper library

In order to use such a wrapper-library, the Rust developer has to change the import statement similar to the one in the first line of Listing 4.2. Now, the function can be used exactly like the original function. Commenting out the first line and removing the slashes in the second line ends observation mode so that the original Crossbeam library is used

directly. In order to obtain the wrapper-library the developer specifies it as a dependency and Cargo will automatically download and build it.

This approach might seem straight forward but it has several significant drawbacks. First using only wrapper-libraries results in recording calls to function/methods defined in libraries, but not for code written by the developer himself. This is not sufficient for extensive performance analysis.

Second, this approach only allows the instrumentation of functions/methods which are exposed by the original library but not internal routines. In order to provide instrumentation for internal routines, a considerable amount of the original library would have to be reimplemented because internal routines are private. Reimplementing major parts of these libraries is work and can introduce errors. Moreover, maintenance would become unreasonable. As noted earlier the Rust libraries are not stable and subject to change, therefore keeping the wrapper-libraries up to date is hardly possible. However, ensuring that they are up to date is necessary because they only work if they match the original.

For these reasons, this approach was not pursued further.

4.1.2 Compiler Plugin

As described in section 2.4 the package manager Cargo automatically downloads and compiles all the required dependencies before building the actual application. Thus Cargo has access to the application and all its dependencies, which makes it appropriate for automatically inserting instrumentation into both. When modifying the compilation process to add instrumentation calls Cargo ensures that probes are inserted into the observed program as well as the dependencies.

The Rust compiler rustc has several interfaces which extend its functionality. One of them is the compiler plugin feature. A compiler plugin is a dynamic Rust library which contains user defined functionality. The compiler provides the `plugin_registrar` function to register plugin functions that provide new functionality. There are two main variants of plugins, syntax extensions or lint checks [14].

Syntax extensions are intended for modifying to the syntax of Rust. However, there exist several variants of syntax extensions with different capabilities. Appropriate for inserting probes is the `MultiItemModifier` interface. This interface is implemented by specifying the `expand` method. One of the arguments of the `expand` method is a part of the abstract syntax tree [5].

The *abstract syntax tree* (*AST*) represents the hierarchical structure of a program. It is constructed after parsing the source code. The AST is an ordered tree. Its nodes are operators with the children of the node being the operator's arguments. The leaves of the tree are variables [32]. In the case of Rust the AST is a recursive data structure containing Rust types for the various syntax elements. Three essential types are `Item`, `Expr`, and `Stmt`. `Item` represents top-level constructs such as imports, type definitions, or function definitions. In short, it corresponds to all language constructs which can be defined in the global scope. The `Expr` and `Stmt` data types represent expressions and statements as explained in Section 2.1.

```
// Needed to allow custom inner attributes
#![feature(custom_inner_attributes)]
// Causes the syntax extension to have effect.
#![syntax_extension_name]
```

Listing 4.3: Attributes needed for the syntax extension to have effect

In order to apply the syntax extension to the source code contained in a file the two attributes in Listing 4.3 have to be placed at the beginning of the source file. Inner attributes affect the item inside which they are placed. In the case of Listing 4.3 they are inserted into a module, as every Rust source file represents one module. The inner attribute `#![syntax_extension_name]` ensure that the compiler invokes the `expand` method of the `syntax_extension_name` plugin with the AST representation of this particular file and the instrumentation calls are inserted into the AST.

This approach looked promising, it enables automatic insertion of instrumentation calls, but this solution can not be used for instrumenting dependencies. As a syntax extension only takes effect if the two attributes in Listing 4.3 are present at the top of a source file. The user would have to annotate every source file of his application and its dependencies containing interesting code regions. This requires the user to have substantial knowledge about the internals of the used libraries. For this reason, the approach utilizing syntax extensions was not further pursued.

After discovering that instrumenting dependencies through syntax extensions is not feasible, the lint checks plugin option was examined. Lint checks plugins allow adding new lints which are undesirable source code patterns. The compiler searches the source code for these patterns and prints warnings or throws errors. Lints are intended to retain or enhance code quality by alerting developers when writing possibly harmful code.

New lint checks are defined by implementing one of the methods provided by the interface. The one best suited for inserting instrumentation is the `check_item` method because it is called for every item defined in an application or library. This allows to easily check against a list of names with items which need instrumentation. The advantage of lint checks over syntax extensions is that they have access to the AST without annotating source files with attributes. Consequently, when Cargo invokes the compiler for the application and all its dependencies the defined lint has access to all of the source code.

However, as Lint checks are intended to find undesirable patterns in the source code and not to alter it they can inspect the AST but there is no way to change it. Consequently, this approach does not work either.

4.1.3 Drop-In Compiler

The compiler plugins did not provide the necessary capabilities to satisfy the requirements. However, rustc provides an interface for building a drop-in replacement for the compiler. It allows customizing the compiler driver, the primary function of the compiler. With calls to rustc internal routines it enables falling back on the compilers default behavior. Additionally, it offers hooks which allow to execute additional code in between compilation steps. Therefore, it facilitates the development of a drop-in compiler which adds instrumentation calls during compilation.

The `CompilerCalls` interface provides the hooks, `after_parse`, `after_expand`, `after_hir_lowering`, `after_analysis`, and `compilation_done`. Each hook takes a callback in form of a closure. This closure is executed after the compilation step indicated by the hook. Thus, after parsing source code the callback of the `after_parse` hook is invoked and after macro expansion the `after_expand` callback is run and so on. Which point during the compilation process is best suited for inserting instrumentation calls?

In each compilation step the compiler gains more information about the program. After parsing and the construction of the AST have been accomplished, the compiler is aware of the syntactical structure of the program. After name resolution and expansion are complete, every name can be uniquely related to a definition and after type checking the type of every element is known. Having more information about the program available is beneficial for satisfying the requirements for instrumentation. Especially, the knowledge of types would be beneficial for the correct insertion of probes. However, adding instrumentation

Data name	Data description	Example
absolute name	Resolved name of the function/method	`std::thread::spawn`
AST depth	Depth of the AST node within the AST	`7`
description	Before or after the funtion/method call	`"LOCAL_BEGIN"`
filename	Name of the file containing the corresponding source code	`"main.rs"`
begin line	Beginning line number of the corresponding source code	`67`
end line	Ending line number of the corresponding source code	`70`

Table 4.1: Static event data with description

later in the compilation process also involves manually producing this information for the inserted code. For instance, when instrumenting after type analysis, the instrumentation framework is responsible for performing name resolution, macro expansion, HIR lowering and type checking for the added code. This is necessary because the compiler expects this information to be present later in the compilation process.

In some cases, existing information has to be changed when adding code. It turned out that when inserting instrumentation calls after name resolution has happened the name resolution information is no longer valid. Consequently, the entire name resolution would have to be rerun after instrumentation. Unfortunately, the compiler interface does not provide a mechanism for rerunning compilation steps. Name resolution is a nontrivial process. Reimplementing it would exceed the scope of this work. For this reason, instrumentation calls are inserted after parsing, but before name resolution.

Section A.1 gives a short description of how to set up an environment which automatically uses the drop-in compiler. In order to instrument a Rust program, it has to be compiled using the drop-in compiler. The rustc drop-in parses the entire source code of this program and constructs the AST. Now the callback closure containing the instrumentation component is invoked and first reads a configuration file. This file contains the names of all functions and methods which are the target of instrumentation (see Section A.2). While traversing the AST the instrumentation component checks for certain node types which will be explained in Section 4.3. If a node of such a type is reached the data described in Table 4.1 is computed for this node. The name resolution performed to determine the *absolute name* is not exhaustive. In particular, there exist several situations in which it is not able to resolve names correctly. For example it can not resolve the absolute names of methods, since the actual method invoked depends on the type on which it is called. At this point during the compilation process type information is not available, thus it is not possible to resolve method names. This definitely is a starting point for later improvements. Nevertheless, this resolution is necessary in order to be able to differentiate between functions defined by the user and functions defined in libraries. If the absolute name corresponding to the current AST node matches a name defined in the config file a reference to this AST node together with the computed data is stored in an ordered list of *instrumentation points*. After traversing the entire AST, this list contains all AST nodes which will be instrumented. The data stored together with the node references is called *static data* because it is known at compile time. Section 4.2 explains *dynamic data* which is computed at run-time.

The AST node references stored in the list of instrumentation points are ordered by their *AST depth*. The list is processed serially starting with the AST node with the greatest AST depth and ending with the node with the smallest depth. In other words, child nodes are instrumented before their parent nodes. This is essential for ensuring the validity of the stored references.

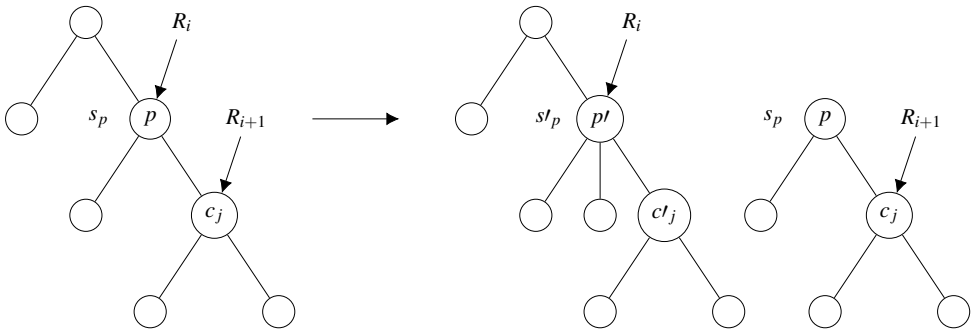

Figure 4.1: AST node replacement and references

To explain why the instrumentation points have to be processed in this particular order, consider the situation shown in Figure 4.1. Generally all of the stored references point to inner AST nodes (not leafs). The stored reference R_i points to AST node p which is the root node of a subtree s_p and reference R_{i+1} points to a child node c_j of p. When instrumenting the parent p before its child c_j, R_{i+1} will become invalid. When probes are inserted at node p the entire subtree s_p is replaced by its instrumented counter part $s\prime_p$, thus the reference R_{i+1} is no longer valid, pointing to the old detached AST tree. Instead R_{i+1} should be pointing to $c\prime_j$. Since the instrumentation framework instruments child nodes before parent nodes this can be avoided. When processing a parent node after its child, the reference to the child in not needed any more because it has been processed already. For this reason it does not matter if the reference to the child node becomes invalid during the replacement of the subtree of its parent node. Processing the stored instrumentation points in this order ensures every AST node references is valid at the time of use and instrumentation is performed correctly for each point.

After inserting probes for every instrumentation point in the list the instrumentation component is finished and the standard compilation process continues producing an instrumented binary.

4.2 Run-Time Instrumentation Architecture

After compiling the program with the Drop-In compiler the resulting binary contains all necessary instrumentation calls. When running the binary the instrumentation calls are invoked and event data is sent via UDP to the collector. The collector is a program which preferably runs on a different physical machine. It receives the event data from the instrumented program and stores the data in an SQLite database. This general structure of the framework is shown in Figure 4.2.

When executing the instrumented binary, first, the global instrumentation environment (`GlobalInstrumentation`) is initialized by reading the same configuration file as the instrumentation component did in the previous section. From the config file it learns the address of the machine running the collector. This information is stored in a global singleton such that all threads have access. Next, the thread-local instrumentation environment is initialized by spawning a helper thread.

When execution reaches an instrumentation call, which is the following function call: `instrument::instrument(static_data)`. The static data is passed to the function as `static_data` object which is constructed entirely of literals inserted during instrumentation. The instrumentation call signals the helper thread by passing on the static data. When the helper thread is signaled it updates the *dynamic data*. This is data such as time stamps

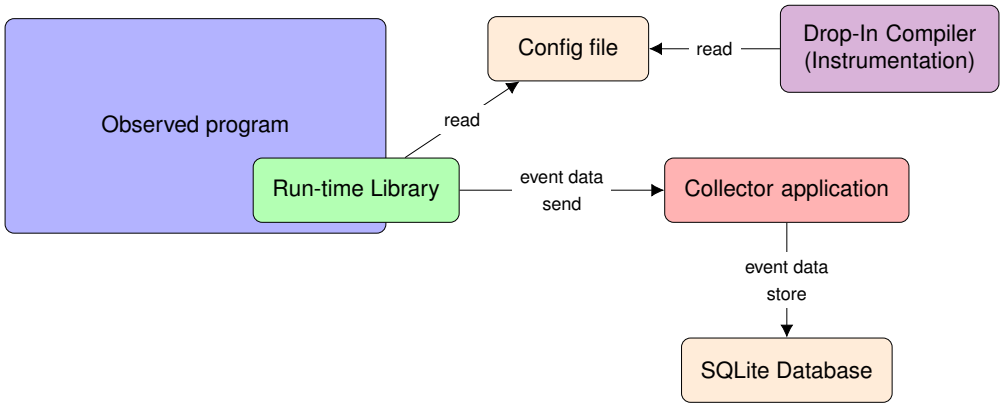

Figure 4.2: Framework architecture

Data name	Data description	Example
system time	Number of nano seconds since an unspecified point in time	`825403805208462`
counter	Number of instrumentation calls invoked by one thread	`37`
pid	Process ID of the current process	`1252`
thread id	Thread ID of the current thread	`ThreadId(1)`
machine id	IP address of the current physical machine	`192.168.178.21`

Table 4.2: Dynamic event data with description

or thread ID which is shown in Table 4.2. The received static data is combined with the updated dynamic data and sent to the collector.

For every thread spawned by the original program, instrumentation spawns one helper thread. Thus every thread of the original program has one corresponding helper thread. These helper threads are responsible for computing the *dynamic data*, encoding and sending the entire event data, *static* and *dynamic data*, via UDP to the collector. Computing dynamic data and sending the data is done for every instrumentation call. Without helper threads, both updating and sending would have to be performed by the original thread, resulting in significantly altered run-time behavior. In order to reduce overhead, these tasks are outsourced to helper threads.

Instrumentation added initialization and finalization routines for the thread-local environment to the original program's main function. Initialization spawns a new helper thread, whereas finalization ensures that the main thread of the original program does not finish before the helper thread sent its last event data. This is necessary to ensure that all of the instrumentation calls are sent. The main thread determines how long the process of the application lives. If the main thread of the application terminates the entire program shuts down, even if helper threads are still running and did not send all event data off.

In summary, the run-time environment consists of helper threads corresponding to each thread of the original program. If an instrumentation call is invoked the corresponding helper thread receives *static data*, updates *dynamic data* and sends both to the collector.

4.3 Instrumentation of Rust Source Code

As described in Section 4.1.3 instrumentation of the references to AST nodes which need instrumentation are stored in a list.

The most interesting nodes for instrumentation are those representing function/method calls because this is where computations are performed. Instrumentation calls are inserted around function/method calls. However, while traversing the AST also other node types are of interest since initializations and finalizations have to be added as well. The following subsections explain the five different types of instrumentation situations, *import statements*, *global scope*, *local scope*, *instrumentation call for functions*, and *instrumentation call for methods*.

4.3.1 Import Statements

All the functions inserted into the program by instrumentation are defined in a library. In order to bring these functions into the scope, this library must be imported. Without this step, the inserted instrumentation calls would be invalid.

```
// Import statement
extern crate instrument;
// The remainder of the program code
// ...
```

Listing 4.4: Instruction inserted for the import statement

The import statement is inserted as the first instruction of the entire program. Therefore, it is attached to the root node as its first child. This is equivalent to the code in Listing 4.4. An `extern crate` statement specifies a dependency on an external library. Additionally, it brings the library into scope, binding it to the identifier `instrument`. This is different from the `use some_library` and `use some_library as short` statements shown earlier, as these only bring the library into scope, but do not specify an external dependency on these libraries. In order to apply these `use` statements, Cargo has to be informed about the dependency. Configuring Cargo to know about this dependency is additional work for the user, which can be avoided by using `extern crate`.

4.3.2 Global Scope Instructions

Initialization and finalization instructions have to be placed at the very beginning and end of the program so that initialization instructions are the first instructions to be executed and finalization instructions are last. For this reason, they need to be inserted into the programs main function. Listing 4.5 shows an instrumented main function. The `global_init` function is responsible for loading the config file and storing the information in a global singleton. Next, `local_init()` initializes a thread-local scope, by spawning a new helper thread for the main thread of the program. It returns a `JoinHandle` which is assigned to a variable. This handle is necessary later for joining the helper thread during finalization. Next, an initial instrumentation event call is placed, marking the start of the application. These three lines make up global initialization.

Before the application finishes finalization is performed. The `clean_up` function call receives the `JoinHandle` of the helper thread and ensures that the main thread waits for its helper thread to finish. Finally, the application exits.

```rust
// Main function of the program
fn main() {
    // Global initialization
    instrument::global_init();
    // Thread-local initialization for main thread
    let instrumentation_local_join_handle = instrument::local_init();
    // Starting instrumentation call
    instrument::instrument(staticData);
    // ----------------
    // ORIGINAL CODE...
    // ----------------
    // Ending instrumentation call
    instrument::instrument(staticData);
    // Finilization of local environment
    instrument::clean_up(instrumentation_local_join_handle);
}
```

Listing 4.5: Instructions inserted for global scope

4.3.3 Local Scope Instructions

The instructions for local scope are inserted whenever a function or method call is encountered which spawns a new thread. They ensure that newly created threads get their corresponding helper threads. For importing as well as global initialization and finalization inserting the appropriate functions was sufficient, in this case, the existing code has to be restructured to ensure the inserted instructions do not alter the program semantically.

```rust
let var = thread::spawn(|| {
    // ----------------
    // ORIGINAL CODE...
    // ----------------
});
// Before instrumentation
// ------------------------------------------------------------------------
// After instrumentation
let var = {
    instrument::instrument(staticData);
    let instrumentation_return_value = thread::spawn(|| {
        let instrumentation_local_join_handle = instrument::local_init();
        instrument::instrument(staticData);
        let instrumentation_return_value = {
            // ----------------
            // ORIGINAL CODE...
            // ----------------
        };
        instrument::instrument(staticData);
        instrumentation_return_value
    });
    instrument::instrument(staticData);
    instrumentation_return_value
};
```

Listing 4.6: Instrumentation of local scope

Consider the `thread::spawn` function from the Rust standard library described in Section 2.3. It takes a closure which contains the code for the new thread and returns a

`JoinHandle` to the created thread. In the first five lines of Listing 4.6 the `JoinHandle` is assigned to a variable via a `let` statement. The call of the `thread::spawn` function is an expression, which evaluates to the `JoinHandle`. Instrumentation has to ensure that after inserting instrumentation calls around the `thread::spawn(...)` function call the `JoinHandle` is returned exactly like before instrumentation.

Generally, any expression can be replaced by another expression yielding the same type. Therefore it is possible to replace the `thread::spawn(...)` function call by a block expression. Inserting a block has two advantages. First, it allows the replacement of this single function call with multiple statements and expressions. Second, it introduces a new scope, thus names defined in this scope shadow existing ones, which reduces the possibility of name collisions. Furthermore, it guarantees any memory allocated for instrumentation within this scope is released at the end of the block. This ensures additional memory allocated for instrumentation is released as soon as possible. This is necessary to keep the memory footprint of the instrumented program as close as possible to its uninstrumented version.

Within the block, a beginning instrumentation call is added (line 10) and then the return value of the original function call is stored in an intermediate return variable (line 11). Next comes an ending instrumentation call, together with the beginning probe it allows to measure the time needed to spawn a new thread. At the bottom of the block (line 23), the result of the `thread::spawn` function call is returned as the result of its variable expression which ensures that the new block expression returns the same value as the single function call did before instrumentation (line 1).

So far, only the instrumentation of the parent thread was explained. Instrumentation of the child thread is inserted inside the body of the closure. The original closure body is moved inside a new block and its result is stored in a variable (line 14), for later return (line 23). This ensures that the returned value of the closure stays the same. Instrumentation calls are added around the assignment in lines 14 till 18 in order to measure the execution time of the computation. However, for the instrumentation calls to send event data the framework requires the new child thread to have its own helper thread. For this reason, the initialization call is placed at the beginning of the closure's body (line 12). In short, the closure's body defined within the function call (line 11) is modified such that the result of the original closure body is assigned to a variable for later return and initialization for the helper thread is added at beginning and end.

```
// Closure definition and assignment
let closure_var = || {
    // Some calculations
};
// Function call with closure name
thread::spawn(closure_var);
```

Listing 4.7: Instrumenting of local scope when closures are define not within the function call

This approach works in most cases, which are those cases in which the closure is defined within the function call, such as in Listing 4.6. As explained in Section 2.1, it is possible to define a closure and assign it to a variable. This allows to define a closure early on in the program and pass it to the `thread::spawn` function by its assigned identifier. This is shown in Listing 4.7. Instrumentation is only triggered for the `thread::spawn` function, thus it only has access to the AST subtree representing the `thread::spawn(...)` function call. However, this expression only contains the identifier of the closure `closure_var`, but not the closure body as this has been defined earlier in the program. At this point, it is not possible to instrument the closure and therefore the newly created thread.

One approach to solve this problem is to define an new closure inside the original `thread::spawn(...)` function call and execute `closure_var` within the new closure. The signature of the new closure must match the signature of the `closure_var` closure, this requires type information. However, instrumentation is performed after parsing thus no type information is available. For this reason, this approach does not work. In order to generically handle all possible situations in which new local scopes are needed, type information is necessary.

It is common practice to define the closure inside the function call, therefore the approach taken by this framework works in the majority of cases.

4.3.4 Instrumentation Calls around Functions

While the three instrumentation situations previously explained, are needed in order to initialize and finalize the run-time environment, function instrumentation is solely for measuring the execution of function calls.

```
1  let var = interest_function(some_func(), 2 + 4, arg3);
2  // Before instrumentation
3  // ------------------------------------------------------------------
4  // After instrumentation
5  let var = {
6      // Extract arguments
7      let instrumentation_argument_var_0 = some_func();
8      let instrumentation_argument_var_1 = 2 + 4;
9      let instrumentation_argument_var_2 = arg3;
10     // Beginning instrumentation call
11     instrument::instrument(staticData);
12     // Function call
13     let instrumentation_return_value = interest_function(
14         instrumentation_argument_var_0,
15         instrumentation_argument_var_1,
16         instrumentation_argument_var_2);
17     // Ending instrumentation call
18     instrument::instrument(staticData);
19     // Return result
20     instrumentation_return_value
21 };
```

Listing 4.8: Instrumentation of function calls

Whenever a function of interest which is listed in the config file is called inside the program, instrumentation calls are added. However, this can not be achieved by simply inserting probes before and after the call. Listing 4.8 shows the original call of function `interest_function`, which expects three arguments. In this example, the first argument is the result of another function call, the second is a simple calculation, and the last is a variable. As described in Section 3.1 it is important that probes measure the function calls of interest and not include other computations. For this reason, it is not sufficient to place instrumentation calls around the `interest_function(...)` function call because measurement would include the evaluation of arguments.

Since instrumentation does not know how expensive the evaluation of the arguments is, they are extracted and stored in variables. This ensures that all arguments are evaluated previous to the `interest_function(...)` call and allows adding the instrumentation calls around it. The result of the call of interest is stored in a variable for later return. Again, a block expression is used to combine all these instructions.

Why are function calls instrumented and not the function definition? Inserting a beginning and ending probe in the function body would ensure that every call of this function is correctly measured. Placing an instrumentation call at the end of the function body, that is where execution returns from the function, is nontrivial. Functions are not always executed sequentially from top to bottom, constructs such as branching allow to return early. To ensure the ending probe is always invoked would require extensive analysis in order to find all points at which the function returns and add an ending probe. Consequently, ensuring correct instrumentation of function definitions is a complex task.

More importantly, when solely instrumenting function definitions it is only possible to record when such a function was invoked and which arguments were passed. However, no information about the caller of the function is available. Consequently, valuable information about the context of the function call including the position (line number and source file) of the function invocation itself cannot be determined. In particular, making caller information available to developers is one of the major advantages of such debugging tools. It is for this reason and the complexity of instrumenting function definitions that this framework settles for instrumenting function calls.

4.3.5 Instrumentation Calls around Methods

The instrumentation of methods allows measuring the execution of method calls. Similar to functions, methods contain computations which makes method calls interesting for analysis.

```
let var = some_expression.first().second().interest_method(
    some_func(),
    2 + 4).last();
// Before instrumentation
// ------------------------------------------------------------------------
// After instrumentation
let var = {
    // Unwind method chain
    let instrumentation_intermediate_var_2 = some_expression;
    let instrumentation_intermediate_var_1 =
        instrumentation_intermediate_var_2.first();
    let instrumentation_intermediate_var_0 =
        instrumentation_intermediate_var_1.second();
    // Extract arguments
    let instrumentation_argument_var_0 = some_func();
    let instrumentation_argument_var_1 = 2 + 4;
    // Beginning instrumentation call
    instrument::instrument(staticData);
    // Method call
    let instrumentation_return_value = instrumentation_intermediate_var_0
        .interest_method(instrumentation_argument_var_0,
            instrumentation_argument_var_1);
    // Ending instrumentation call
    instrument::instrument(staticData);
    // Return result
    instrumentation_return_value
}.last();
```

Listing 4.9: Instrumentation of method calls

Whenever a method of interest is called in a program, instrumentation annotates it with probes. Similar to function instrumentation arguments passed to the method between

the parenthesis have to be extracted to ensure they are evaluated before the instrumented method call. While the number of arguments passed to a method may vary, all methods receive at least one argument, the object they are called on. This object may be the result of another expression which allows chaining multiple methods together. In order to place the instrumentation calls directly around the method of interest, the object on which it is called has to be evaluated before the beginning instrumentation call. To achieve this, the chained methods are unwinded as shown in Listing 4.9 using an intermediate variable to store the result of each method call.

```
let intermediate = vec![1, 2, 3, 4].iter();
instrument::instrument(staticData);
let result = intermediate.map(|elem| elem*2).sum();
//ERROR: temporary value dropped while borrowed
```

Listing 4.10: Temporarily created values

It is necessary to assign the result of each method in the chain to a variable because otherwise values might get released too early. Listing 4.10 shows such an example. In the first line, the `iter` method is called on a vector of four numbers. It returns an iterator over a reference to the vector. However, since the vector itself is never assigned to a variable, it is only created temporarily and is freed at the end of the first line. When calling the map method on the iterator in line three the vector referenced by the iterator has already been released. Rust detects this and throws an error at compile time. Since every one of the chained methods can introduce such a temporary value, the result of each method has to be assigned to a variable to ensure that it is not freed before the method of interest is called.

The remainder of Listing 4.9 (starting with line 14) is structured analogously to the instrumentation of functions.

4.4 Persistent Storage of Trace Data

So far, only the instrumentation and run-time environment have been explained in detail. For collecting and storing the data the collector application is provided. It is independent of the other parts of the framework, which allows the collector to be run on a separate physical machine. This reduces the overhead produced by the framework as the collector has its own machine and does not take away resources of the observed program.

The event data is sent to the collector via UDP. In contrast to the common TCP protocol, UDP uses a connectionless communication model thus no time is spent on establishing connections or error correction such as resending packets. Establishing a new network connection to the collector every time the run-time environment spawns a new helper thread would result in significant overhead. Using UDP, the run-time can send off packets without needing to wait until the message arrives or handle corrupted messages. It does not matter in which order the event data arrives since it contains time stamps which allow later restoring the correct order. Considering the low overhead sporadic packet loss is not a big problem. In typical measurement runs, the collector receives event data of multiple thousand events losing one or two does not affect detailed performance analysis. It should be noted that if the number of parallel threads becomes too large, the amount of sent data can saturate the network link resulting in a large portion of the packets being dropped. Chapter 6 will present a potential solution to this.

Highly parallel applications can generate large amounts of event data. Therefore it is essential that the collector can handle many incoming messages at a time. The collector application is built using *Tokio* [4], a Rust library for developing asynchronous, event-driven network applications. The collector handles incoming messages asynchronously. Therefore

it maintains a thread pool and incoming packets are assigned to threads on the pool for processing. In particular, the messages are decoded and inserted into an SQLite database. The data of one event corresponds to one tuple in the database.

```
$instcollect run_1 -db instrumentation.db --table fractale
```

Listing 4.11: Invokation of the collector application

The collector is started from the command line on the desired machine like it is shown in Listing 4.11. It allows specifying the database file as well as the name of the table in which the event data will be inserted. If the database or the table does not already exist a new database or table is created. The schema currently used for the database is simple. Each event represented by its event data (static and dynamic data) corresponds to one row in the database table. Consequently, the table contains one column for each event data feature. The primary key is constructed of the timestamp and the thread id, as one thread cannot execute two instructions at a time this combination will always be unique.

CHAPTER 5

Experiments and Results

This chapter evaluates the newly build tRust framework. Currently three libraries are supported for instrumentation. In order to appropriately evaluate the new tRust framework, it is necessary to test each of the three supported libraries, Rayon, Crossbeam, and Timely Dataflow. Therefore, three benchmark applications were selected, each using one of the three libraries to demonstrate the most important capabilities of the framework. For evaluation, each benchmark application was first compiled using the drop-in compiler yielding an instrumented binary. During a representative measurement run, the generated event data was recorded using the collector application. Finally, the data stored in the SQLite database was examined and analyzed.

Additionally, the execution time of each benchmark application was evaluated. For this, the instrumented version and the original version of each benchmark were executed four times, recording the elapsed wall clock with the `/usr/bin/time` utility. The timings measured are the *real time*, *user time*, and *system time*. Real time is the wall clock time which elapsed during the execution of the application. User time denotes the time the program spent in user mode. It is important to note that user time is the sum of all used execution units. For instance, if an application has four threads and each thread runs 10 seconds in user mode user time would be 40 seconds. As these four threads are potentially executed simultaneously, the corresponding real time can be less. Last, the system time matches the time the application was executed in kernel mode. The average timings of the four measurement runs of the version with added probes and the version without are compared for analysis of the instrumentation overhead.

The following section describes the environment used for the experiments. In Section 5.2 each benchmark is first explained and then performance analysis results are evaluated.

5.1 Evaluation Environment

As it is recommended to execute the collector component of the framework on a different physical machine to minimize overhead, the evaluation environment consists of two machines, one for compiling and executing the application under observation and one for running the collector.

The machine responsible for running the instrumented application is equipped with eight GB of system memory and one Intel Core i5-4258U processor with two cores and a

total of four threads. The second machine running the collector contains a AMD Ryzen 7 2700 CPU with eight cores and a total of 16 threads as well as 16 GB of memory. The two machines are connected over a switch via Gigabit Ethernet.

5.2 Benchmarks

As described in Section 3.3, each of the supported libraries favors a specific programming model: Rayon is a data parallelism library, Crossbeam supports message passing as well as the thread model, and Timely Dataflow is designed with dataflow programming in mind.

The benchmarks were selected to match these programming models, in order to demonstrate their unique properties.

5.2.1 Creating Vanity Keys

The first benchmark is intended to evaluate the tRust framework in combination with the Crossbeam library. In particular, the instrumentation of the channels implementation of Crossbeam is analyzed. Channels are well suited for implementing the message passing programming model.

This first benchmark is the creation of vanity keys. This is often done to personalize the address of cryptocurrency wallets. Every cryptocurrency user has a wallet with a corresponding address. This address usually is a cryptographic public key. Additionally, every wallet comes with a private key used for signing transactions. In sum, if person *A* wants to pay money to person *B*, person *A* sends money to *B*'s wallet address (public key) and signs the transaction with his (persons *A*) private key [47]. Thus, the public key of the wallet is equivalent to a bank number, which is given away such that people can transfer money to this account. In order to make the public key stand out and make it easier to remember, it is possible to generate a *vanity key* [57]. A vanity key is a cryptographic key with a particular sequence of letters at the beginning, which makes up a word or name, such as the key starting with `pEteR` shown in Listing 5.1. Such a vanity key is generated by brute force, feeding different values to a hash function and comparing the result to a specified sequence of letters.

```
pEteRAmV8Glxu5VJb8w6VfMcYVjenHncqFfw9AAjQfE
```

Listing 5.1: Example vanity key

In this benchmark hashes are generated from three different cryptographic algorithms, SHA2 [51], SHA3 (Keccak) [52], and BLAKE2 [19], all configured to use keys of length 512-bit. In order to leverage Crossbeams channels this application is modeled as a single producer multiple consumer problem. The single producer generates new input values, which are sent through a single channel to the consumers. Every consumer thread computes a hash with its hash function and compares the resulting key against a specified sequence. Because of its producer-consumer structure, this implementation uses four threads, the main thread for the producer and three child threads one for each hash function. For this evaluation the particular sequence of characters which have to be present at the beginning of the hash key is "`cab`". For example, a taxi company which accepts BitCoin payments wants to make its wallet address stand out.

Listing 5.2 shows the relevant instructions of the programs `main` function. In line 6, the `channel::bounded` function (provided by Crossbeam) is called returning the channels sender and receiver. Next the three `consumer_*` functions are called each receiving a copy of the receiver object. Every `consumer_*` function defines one consumer. Listing 5.3 shows one of the consumers, in particular the consumer with Blake2 hash function. In

the beginning, every consumer spawns a new thread as shown in line 35. Afterward, the `recv` method is called inside the `while` condition (line 39), such that the loop terminates when an error is received. After receiving input on the channel the hash key for this input is generated (line 41). The `thread::spawn(...)` function call (line 35), the `send(...)` method call (line 11), and the `recv()` method call inside the `consumer_blake2` function (line 39) as well as the `recv()` calls inside the two other consumer functions are the interesting function/method calls in this application which will be annotated with instrumentation calls.

```
4  fn main() {
5      // Some other instructions
6      let (sender, receiver) = channel::bounded(1);
7      let c_blake2 = consumer_blake2(receiver.clone(), ...);
8      let c_sha3 = consumer_sha3(receiver.clone(), ...);
9      let c_sha2 = consumer_sha2(receiver.clone(), ...);
10     // Producer
11     while sender.send(salt_gen.new_salt()).is_ok() {}
12     // Some other instructions
13 }
```

Listing 5.2: Main function from the examples/vanitykey.rs source file

```
34 fn consumer_blake2(...) -> ... {
35     thread::spawn(move || {
36
37         // Some other instructions
38
39         while let Ok(salt) = receiver.recv() {
40             // Some other instructions
41             let result = hasher.result_reset();
42             // Some other instructions
43         }
44         // Some other instructions
45 })}
```

Listing 5.3: The `consumer_blake2` function form the examples/vanitykey.rs source file

Table 5.1 shows parts of the collected event data in the database after instrumenting the benchmark application and running the instrumented binary. The three dots "..." indicate where rows have been omitted. Some of the column names are abbreviated including cntr (counter), abs_path (absolute path), ast_dth (ast_depth), l_begin (line_begin) and l_end (line_end). Furthermore, the columns for IP address and process ID were removed from this table and the other result tables throughout this chapter. The tables shown in the Section B of the appendix also lack these two columns to save space. The values of these two columns were the same for every row, thus they did not contain valuable information and could be removed. The rows of all event data table are ordered by the time_stamp column.

The `GLOBAL_BEGIN` event shown in row one of Table 5.1 occures first. It originates from global initialization inserted at the beginning of the main function. Afterward, the main thread spawns the three child threads responsible for computing the hash keys, which is indicated by the `LOCAL_BEGIN` events. As the l_begin column of row four indicates the first `LOCAL_BEGIN` event corresponds to the `thread::spawn` call inside the `consumr_blake2` function, shown in line 35 of the source code Listing 5.3. In between the `LOCAL_BEGIN` events the first spawned child thread (`ThreadId(3)`) already calls the `recv` method (row 6) on the channel. According to the l_begin column this event is associated with the `recv`

	time_stamp	cntr	thread_id	abs_path	description	ast_dth	source_file	l_begin	l_end
1	15102611099457	1	ThreadId(1)	main	GLOBAL_BEGIN	2	examples/vanitykey.rs	105	122
2	15102611233964	2	ThreadId(1)	thread::spawn	BEGIN	6	examples/vanitykey.rs	35	52
3	15102611365739	4	ThreadId(1)	thread::spawn	BEGIN	6	examples/vanitykey.rs	60	77
4	15102611443530	1	ThreadId(3)	thread::spawn	LOCAL_BEGIN	6	examples/vanitykey.rs	35	52
5	15102611497545	5	ThreadId(1)	thread::spawn	END	6	examples/vanitykey.rs	60	77
6	15102611602995	2	ThreadId(3)	recv	BEGIN	13	examples/vanitykey.rs	39	39
7	15102611687252	6	ThreadId(1)	thread::spawn	BEGIN	6	examples/vanitykey.rs	85	102
8	15102611810325	3	ThreadId(3)	recv	BEGIN	8	.../crossbeam-...-0.3.8/.../channel.rs	698	698
9	15102611878125	7	ThreadId(1)	thread::spawn	END	6	examples/vanitykey.rs	85	102
10	15102611919797	1	ThreadId(5)	thread::spawn	LOCAL_BEGIN	6	examples/vanitykey.rs	60	77
11	15102612044870	8	ThreadId(1)	send	BEGIN	8	examples/vanitykey.rs	11	11
12	15102612109340	1	ThreadId(7)	thread::spawn	LOCAL_BEGIN	6	examples/vanitykey.rs	85	102
13	15102612121082	2	ThreadId(5)	recv	BEGIN	13	examples/vanitykey.rs	64	64
14	15102612185112	9	ThreadId(1)	send	BEGIN	8	.../crossbeam-...-0.3.8/.../channel.rs	390	390
15	15102612273126	2	ThreadId(7)	recv	BEGIN	13	examples/vanitykey.rs	89	89
16	15102612339682	3	ThreadId(5)	recv	BEGIN	8	.../crossbeam-...-0.3.8/.../channel.rs	698	698
17	15102612459231	10	ThreadId(1)	send	END	8	.../crossbeam-...-0.3.8/.../channel.rs	390	390
18	15102612466290	4	ThreadId(3)	recv	END	8	.../crossbeam-...-0.3.8/.../channel.rs	698	698
19	15102612561254	3	ThreadId(7)	recv	BEGIN	8	.../crossbeam-...-0.3.8/.../channel.rs	698	698
20	15102612680161	5	ThreadId(3)	recv	END	13	examples/vanitykey.rs	39	39
21	15102612690849	11	ThreadId(1)	send	END	8	examples/vanitykey.rs	11	11
22	15102612816204	12	ThreadId(1)	send	BEGIN	8	examples/vanitykey.rs	11	11
23	15102612897457	13	ThreadId(1)	send	BEGIN	8	.../crossbeam-...-0.3.8/.../channel.rs	390	390
24	15102612964779	4	ThreadId(5)	recv	END	8	.../crossbeam-...-0.3.8/.../channel.rs	698	698
25	15102612974639	14	ThreadId(1)	send	END	8	.../crossbeam-...-0.3.8/.../channel.rs	390	390
26	15102613072827	5	ThreadId(5)	recv	END	13	examples/vanitykey.rs	64	64
27	15102613109967	15	ThreadId(1)	send	END	8	examples/vanitykey.rs	11	11
28	...	...	...	...	...	...	...	...	...

Table 5.1: Event data from the vanity key experiment

	Instrumented	**Not Instrumented**
real time	4.3777s	1.3775s
user time	7.3875s	2.1575s
system time	3.6010s	0.0970s

Table 5.2: Average timings of instrumented and not instrumented vanity key application

method call in line 39, which is show in Listing 5.3. However, the main thread representing the producer did not send any data down the channel yet, therefore thread 3 blocks. Following the third invocation of the `thread::spawn` function the main thread sends the first input for hash computation down the channel by calling `send`. The `send` method called in `examples/vanitykey.rs` line eleven invokes a Crossbeam internal `send` method in file `.../channel.rs` (row 14 Table 5.1). Now the `END` event (row 18) of the `recv` method call implies that the thread with `ThreadId(3)` stopped blocking and received the data sent by the producer.

Meanwhile, the two other child threads called the `recv` method on the channel (row 13 and 15 Table 5.1) and are waiting for the producer to send new input down the channel. The `BEGIN` event in row 22 indicates the main thread invoked the `send` method for the second time, such that one of the two blocked threads will be able to continue.

It is evident that this program uses three simultaneous threads as there are three different thread IDs present. Furthermore, analysis of the entire event data reveals that the thread with ID 7 calls the receive method 73 times compared to the thread with ID 3 45 times. Thus thread with ID 7 computes approximately twice as many hash keys as the other thread. Hence the thread with ID 3 spends almost twice as much time computing hash values.

Table 5.2 shows the average execution timings for the instrumented and original application. It reveals that the instrumented version of this benchmark application executes about three times as long as its uninstrumented counterpart. As expected, instrumentation strongly affects the execution behavior of the program. This application incorporates much communication, the producer sends each new input value down the channel such that the

consumer can compute new hash keys. Furthermore, the time elapsing between the send and receive calls is short. Thus, during the execution of the vanity key application, many instrumentation calls occur within a short time interval. This results in high overhead which is indicated by the longer execution time of the instrumented version. However, for debugging purposes a slowdown by the factor of three is usually still acceptable.

5.2.2 Fractal Calculation

This benchmark intends to evaluate the tRust framework in combination with the Rayon library, a library designed for data parallelism. Therefore, the selected task should benefit of data parallelism. For this reason, an embarrassingly parallel code structure is well suited.

A good fit for such a task is the plotting of fractals. The resulting structures show self-symmetry at all scales. In other words, when magnifying a particular part of a fractal, patterns will repeat themselves. All fractals are computed by a recursive function. Depending on the function different fractals arise. The fractal computed here is the Mandelbrot set, which was first discovered by Benoit Mandelbrot [40].

The Mandelbrot set consists of a set of points in the complex plane. Thus each point on this plane is a complex number $c \in \mathbb{C}$. Each number c consists of a real part and an imaginary part. This allows modeling each complex number as a pixel of an image, with the real value being its horizontal position and the imaginary value being its vertical position in the image. When assigning pixels in the Mandelbrot set and pixels not in the Mandelbrot set different colors a visual representation of the Mandelbrot set is generated. In particular a number $c \in \mathbb{C}$ belongs to the Mandelbrot set iff

$$\lim_{n\to\infty} \| z_n \| \nrightarrow \infty \tag{5.1}$$

where $z_n = z_{n-1}^2 + c$ and $z_0 = 0$. The $\| \cdot \|$ sign denotes the Euclidean norm. Consequently, a complex number $c \in \mathbb{C}$ belongs to the set if the result of repeatedly applying this recursive function is bounded. In implementations of this function, it is often difficult to determine whether z_n will eventually diverge or not due to slow divergence at some points. For this reason, implementations usually use a maximum number of iterations m. It has been proven that if $\| z_n \| \geq 2$ z_n will eventually diverge. Therefore, implementations test every iteration if either the result is greater than two or if the maximum number of iterations has been reached, terminating if either condition is met. If $\| z_n \| < 2$ holds throughout all the iterations the pixel (number) belongs to the Mandelbrot set and is colored accordingly [15].

When visualizing the Mandelbrot set the computation described above has to be performed for every pixel of the visualization. However, since pixel values can be computed independently from each other this allows for efficient use of data parallelism. Therefore, this task works well for evaluating the tRust framework in combination with the Rayon library [15].

The particular implementation used for evaluation uses a character matrix instead of an image, with # indicating that the corresponding complex number is in the Mandelbrot set and a whitespace character indicating it is not. The resulting character matrix is shown in Figure 5.1. The computation of each matrix position is parallelized across multiple threads using a parallel iterator from the Rayon libraries, which is called on the matrix data structure.

The first row of Table 5.3 shows the `GLOBAL_BEGIN` instrumentation call is reached first, as it marks the beginning of the entire program. Accordingly, `GLOBAL_END` event (last row) marks the end of the application. The thread ID column shows that the main thread (`Thread(1)`) invokes both of these calls. Additionally, the abs_path column together with the last three columns indicate the location of the function/method call these instrumentation calls are associated with, in the case of the `GLOBAL_*` events, the main function which

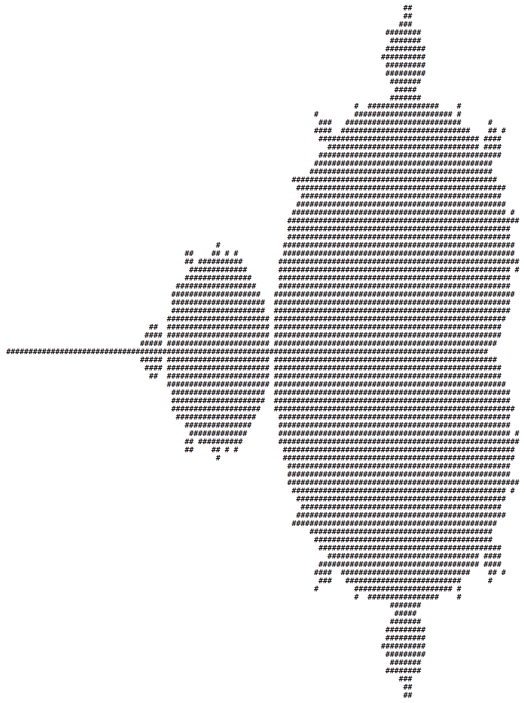

Figure 5.1: Mandelbrot set as 96×96 character matrix

	time_stamp	cntr	thread_id	abs_path	description	ast_dth	source_file	l_begin	l_end
1	8819245541627	1	ThreadId(1)	main	GLOBAL_BEGIN	2	examples/fractal.rs	66	72
2	8819245740176	2	ThreadId(1)	par_iter_mut	BEGIN	8	examples/fractal.rs	30	30
3	8819245932493	3	ThreadId(1)	par_iter_mut	END	8	examples/fractal.rs	30	30
4	8819246984967	4	ThreadId(1)	join_context	BEGIN	14	.../rayon-1.0.3/.../mod.rs	409	415
5	8819257303714	1	ThreadId(7)	join_context	LOCAL_BEGIN	14	.../rayon-1.0.3/.../mod.rs	409	415
6	8819257311026	1	ThreadId(6)	join_context	LOCAL_BEGIN	14	.../rayon-1.0.3/.../mod.rs	409	415
7	...	...	...	...	...	...	...	...	...
8	8830568425424	1	ThreadId(121)	join_context	LOCAL_BEGIN	14	.../rayon-1.0.3/.../mod.rs	409	415
9	8830579068911	2	ThreadId(117)	join_context	LOCAL_END	14	.../rayon-1.0.3/.../mod.rs	409	415
10	8830580207943	1	ThreadId(122)	join_context	LOCAL_BEGIN	14	.../rayon-1.0.3/.../mod.rs	409	415
11	8830696721644	2	ThreadId(102)	join_context	LOCAL_END	14	.../rayon-1.0.3/.../mod.rs	409	415
12	8830697389629	1	ThreadId(123)	join_context	LOCAL_BEGIN	14	.../rayon-1.0.3/.../mod.rs	409	415
13	...	...	...	...	...	...	...	...	...
14	8831787008562	2	ThreadId(121)	join_context	LOCAL_END	14	.../rayon-1.0.3/.../mod.rs	409	415
15	8831787502736	1	ThreadId(131)	join_context	LOCAL_BEGIN	14	.../rayon-1.0.3/.../mod.rs	409	415
16	8831945221673	2	ThreadId(124)	join_context	LOCAL_END	14	.../rayon-1.0.3/.../mod.rs	409	415
17	...	...	...	...	...	...	...	...	...
18	8833695404530	2	ThreadId(135)	join_context	LOCAL_END	14	.../rayon-1.0.3/.../mod.rs	409	415
19	8833696856259	5	ThreadId(1)	join_context	END	14	.../rayon-1.0.3/.../mod.rs	409	415
20	8833705323059	6	ThreadId(1)	main	GLOBAL_END	2	examples/fractal.rs	66	72

Table 5.3: Event data from the fractal experiment

	Instrumented	Not Instrumented
real time	14.2905s	14.3125s
user time	43.0088s	42.9758s
system time	0.3150s	0.2680s

Table 5.4: Average timings of instrumented and not instrumented fractal application

ranges from line 66 to line 77 in file `examples/fractal.rs`. The counter column shows that the `GLOBAL_BEGIN` is the first and `GLOBAL_END` is the sixth instrumentation call invoked by the main thread. If the value of the `ast_depth` column would be one this would indicate both instrumentation calls are invoked at the top level (global scope), instead, the value two points out it is invoked on the second level.

After the initial `GLOBAL_BEGIN` event the calculation of the fractal begins by calling the `par_iter_mut` method on the 96×96 matrix. This method provided by Rayon returns a parallel iterator which allows data-parallel processing of the matrix. Rayon internally uses the `join_context` function for spawning threads and defining workloads. Consequently, after the `BEGIN` event of the `join_context` function the `LOCAL_BEGIN` event occurs inside a newly spawned thread, which is shown by the new thread ID (`Thread(7)`). As it is a Rayon internal routine it is invoked inside the library source code, shown in the `source_file` column. Execution goes on with multiple calls to `join_context`.

The time between a `LOCAL_BEGIN` and `LOCAL_END` is the time a thread needs to process its assigned data portion of the data. For example, thread 121 (`Thread(121)`) starts at time stamp 8830568425424 and finishes at time stamp 8831787008562, thus the duration is 1218583138 nanoseconds or approximately 1.2 seconds. The event data reveals that some threads take longer than others, in particular, `Thread(121)` with 1.2 seconds needed a lot of time as threads calculate on average for 0.8 seconds. Nevertheless, there are threads which are finished almost immediately.

Table 5.4 shows the average execution timings for the instrumented and original application. It reveals that the real run-time of the instrumented version is slightly less than the original version. However, the fact that the application with added probes has greater user and system timings indicates that the instrumentation introduces overhead. However, less than one percent difference between the user times of both versions is well in the margin of error. The system time of the instrumented version is 17 percent longer than the original version, which is probably due to the spawning of helper threads. In sum, the overhead of the instrumented version is very low. It is important to note that this implementation of fractals has few instrumentation calls since data portions are assigned to threads and these threads spend long times processing the data before receiving new data. Thus not much time is spent on communication between threads.

5.2.3 PageRank

The last benchmark task is intended for evaluating the tRust framework in combination with the Timely Dataflow library. As described in Section 3.3.6 this library implements the timely dataflow programming model. This allows for efficient processing of streaming data and flows. An excellent showcase for the Timely Dataflow library is the PageRank algorithm developed by Larry Page and Sergey Brin [54].

PageRank is a graph-based algorithm initially designed to rate the relevance of web pages in the world wide web. The nodes in a directed graph represent websites. An edge points from a node x to another node y if a link exists pointing from the web site corresponding to x to the website associated with node y. Let X be the set of all websites containing links to a site y. Consequently, the rank of this web page y is high if the sum of the ranks in

	time_stamp	cntr	thread_id	abs_path	description	ast_dth	source_file	l_begin	l_end
1	135346766088904	1	ThreadId(1)	main	GLOBAL_BEGIN	2	examples/pagerank.rs	8	200
2	135346766235263	2	ThreadId(1)	timely::execute_from_args	BEGIN	7	examples/pagerank.rs	9	198
3	135346779188063	3	ThreadId(1)	timely::execute_from_args	END	7	examples/pagerank.rs	9	198
4	135346779657673	1	ThreadId(5)	timely::execute_from_args	LOCAL_BEGIN	7	examples/pagerank.rs	9	198
5	135346779685915	1	ThreadId(6)	timely::execute_from_args	LOCAL_BEGIN	7	examples/pagerank.rs	9	198
6	135346779708353	1	ThreadId(7)	timely::execute_from_args	LOCAL_BEGIN	7	examples/pagerank.rs	9	198
7	135346784083029	2	ThreadId(7)	send	BEGIN	16	examples/pagerank.rs	181	181
8	135346784115904	2	ThreadId(5)	send	BEGIN	16	examples/pagerank.rs	181	181
9	135346784145464	2	ThreadId(6)	send	BEGIN	16	examples/pagerank.rs	181	181
10	135346784300237	3	ThreadId(7)	send	END	16	examples/pagerank.rs	181	181
11	135346784386752	3	ThreadId(5)	send	END	16	examples/pagerank.rs	181	181
12	135346784466515	3	ThreadId(6)	send	END	16	examples/pagerank.rs	181	181
13	...	...	...	...	...	...	...	...	...
14	135355143246504	33318	ThreadId(7)	send	BEGIN	10	.../timely_com...-0.8.0/.../process.rs	106	106
15	135355143338154	33732	ThreadId(6)	send	BEGIN	10	.../timely_com...-0.8.0/.../process.rs	106	106
16	135355143390904	33940	ThreadId(5)	send	BEGIN	10	.../timely_com...-0.8.0/.../process.rs	106	106
17	135355143484226	33319	ThreadId(7)	send	END	10	.../timely_com...-0.8.0/.../process.rs	106	106
18	135355143593585	33733	ThreadId(6)	send	END	10	.../timely_com...-0.8.0/.../process.rs	106	106
19	135355143635724	33941	ThreadId(5)	send	END	10	.../timely_com...-0.8.0/.../process.rs	106	106
20	135355143782434	33734	ThreadId(6)	send	BEGIN	10	.../timely_com...-0.8.0/.../process.rs	106	106
21	135355143834537	33320	ThreadId(7)	send	BEGIN	10	.../timely_com...-0.8.0/.../process.rs	106	106
22	135355143851795	33942	ThreadId(5)	send	BEGIN	10	.../timely_com...-0.8.0/.../process.rs	106	106
23	135355143959577	33735	ThreadId(6)	send	END	10	.../timely_com...-0.8.0/.../process.rs	106	106
24	135355144033779	33321	ThreadId(7)	send	END	10	.../timely_com...-0.8.0/.../process.rs	106	106

Table 5.5: Event data from the PageRank experiment

X is high [54].

The simple rank $r(y)$ of a website y is calculated according to the slightly simplified recursive PageRank formula:

$$r(y) = d \sum_{x \in X} \frac{r(x)}{|x|} \tag{5.2}$$

where $|x|$ denotes the number of outgoing links from site x and $d \in [0,1]$ is a factor used for normalization, so that the total rank of all web pages (all nodes in the graph) is constant. The ranks for all nodes (websites) in the graph are computed by starting with an arbitrary set of initial ranks and iteratively computing this equation until the ranks converge [54].

The PageRank benchmark application [43] was executed with three worker threads in addition to the main thread, which is indicated by the four different thread ids present in the thread_id column of event data Table 5.5. Again, the first event is the `GLOBAL_BEGIN` event followed by the `BEGIN` event of the `timely::execute_from_args` function, which immediately returns as the `End` event occurs right after. This indicates it is an asynchronous function. Now the worker threads are spawned which is indicated by the three consecutive `LOCAL_BEGIN` events.

After all worker threads are spawned, each of them sends data by calling the `send` method, which is invoked in line 181 of the benchmark program. The `send` method call in `examples/pagerank.rs` invokes an internal `send` method of the Timely library. The trace goes on with alternating calls to `send` from the program and the library.

This PageRank implementation, in combination with this particular instrumentation, produces a lot of instrumentation data compared to the other two experiments.

	Instrumented	**Not Instrumented**
real time	8.9238s	1.1845s
user time	9.8815s	3.0015s
system time	6.6593s	0.0795s

Table 5.6: Average timings of instrumented and not instrumented PageRank application

When the instrumented binary sends this many event data packets to the collector machine, it appears a considerable amount of packages are lost. The machine running the collector has to process this large quantity of incoming packets sufficiently fast. Generally,

UDP Packets arrive at the network interface of the machine. From there they are moved into the socket receive buffer. Applications listening for incoming data are responsible for retrieving the packets form this buffer before new ones arrive. It appears the collector does not process network traffic fast enough to make room for new incoming packets. Hence, packets are dropped and cannot be stored in the database.

Furthermore, the timings in Table 5.6 show that execution of the instrumented program takes approximately eight times as long as its not instrumented counterpart. The very many probes invoked during execution introduces a significant amount of overhead.

CHAPTER 6

Conclusion and Future Work

As the number of simultaneously possible computations steadily increases, developing efficient and reliable parallel applications becomes essential in order to take full advantage of the additional computing power. The young programming language Rust aims to ease the development of such high-performance applications. However, existing performance analysis tools lack support for Rust. The purpose of this work was to fill this gap by designing and implementing a framework which allows observing the execution of parallel Rust applications. Therefore, in the first part of this work, different instrumentation approaches were explored in order to find a technique capable of conveniently instrumenting an application and its dependencies. Next, a run-time environment designed for minimal overhead was conceived and implemented. For efficient storage of event data, a collector application was implemented. Finally, the developed framework was evaluated using three different benchmark application, each targeting one of three libraries common in Rust.

The evaluation demonstrated that the tRust framework is capable of correctly instrumenting Rust applications and their dependencies yielding an executable binary. Furthermore, the collected event data enables developers to precisely retrace program execution behavior of concurrent and parallel programs implementing various programming paradigms not immediately available in the established HPC programming languages Fortran, C, and C++. Experiments suggest good performance of tRust in combination with the Crossbeam library as it produces fine-grained event data, which helps to understand complex interleaved communication. For applications depending on the Rayon library tRust adds very little overhead. In combination with Timely Dataflow, the evaluation revealed that some work is still required in very exacting applications in order to cope with the excessive amount of data produced by the worker threads.

Existing tools have been developed for many years, continually improving stability and adding new features. Implementing more features or exploring other design approaches would have gone beyond the scope of this work. This holds possibilities for further improvements. Designing a parallel performance analysis framework involves many difficulties. Especially, determining ways for generically inserting instrumentation so that the result is syntactically and semantically correct turned out to be challenging. In some situations, correct instrumentation is not possible without type information. Adding instrumentation after the type analysis compilation step, is a prerequisite for inserting probes in several more complex code constructs as well as to strengthen resolution of method names.

The tRust framework transmits the event data from the instrumented program to the

collector via UDP. During evaluation only one collector process was running. However, it is possible to run multiple collector applications either on one machine or on multiple physical machines spreading the load across multiple entities, which may reduce packet loss. Furthermore, the standardized UDP interface allows easily connecting inspector applications for live viewing by simply listening for incoming UDP packets.

The collector stores the event data in a single table which is simple and efficient. Nevertheless, relational databases such as SQLite allow complex schemas in order to structure data. A more elaborate data structure could ease the later analysis of event data.

Another possible refinement of this work would be the introduction of mechanisms for user-defined events or conditions which limit the execution of instrumentation calls. This grants users more fine-grained control over how much event data is produced and where events are triggered. Adding such features to tRust could help reduce the overwhelming amount of generated event data in some cases.

In ongoing and future work, tRust could be improved and extended by one or several of the features mentioned above to facilitate the development of reliable parallel Rust applications.

APPENDIX A

Usage of the tRust Framework

A.1 Setting up a Rustup Toolchain

For Cargo to use the drop-in compiler provided by tRust it is necessary to register a custom toolchain with rustup. The following describes how to set up a custom toolchain on Ubuntu. It is important that this is run after the drop-in compiler was built.

1. Create a new directory which will later contain the custom toolchain.

```
$ mkdir ~/.rust_custom_toolchains
```

2. Copy the entire toolchain used to build the drop-in compiler to the newly created directory.

```
$ cp -R ~/.rustup/toolchains/nightly-2019-02-07-x86_64-unknown-linux-gnu
    ~/.rust_custom_toolchains/rustinst
```

3. Copy the executable binary of the drop-in compiler inside the `bin` directory of the new toolchain.

```
$ cp path/to/where/built/rustc-dropin/is/rustc
    ~/.rust_custom_toolchains/rustinst/bin/
```

4. Use `rustup` to register the new *rustinst* toolchain.

```
$ rustup toolchain link rustinst ~/.rust_custom_toolchains/rustinst
```

5. Finally, override the default toolchain for the working directory containing the user program such that Cargo will automatically use the new *rustinst* toolchain.

```
$ rustup override set rustinst
```

A.2 Description of the Configuration File

The configuration file containing the functions and methods of interest as well as the address (IP and port) of the machine running the collector application has to be stored in `~/.rust_inst/instconfig.toml`. As the file extension indicates the file is formatted as TOML (Tom's Obvious, Minimal Language), a common file format for configuration files in the Rust ecosystem. The various options for configuring tRust are explained in the following:

```
machine_id = "192.168.86.76"
collector_ip = "192.168.86.71"
collector_port = 8080
code_2_monitor = [
    ["", "ExternCrateItem"],
    ["main", "GlobalScope"],
    ["std::thread::spawn", "LocalScope"],
    ["par_iter_mut", "LocalScope"],
    ["rayon::join", "LocalScope"],
    ["join_context", "LocalScope"],
    ["crossbeam_channel::bounded", "InstCallForFunction"],
    ["send", "InstCallForMethod"],
    ["recv", "InstCallForMethod"],
    ["timely::execute_from_args", "LocalScope"],
    ["receive", "InstCallForMethod"],
]
```

Listing A.1: Example configuration file

machine_id specifies the IP address of the current system. This address is sent as part of the static data to the collector application.

collector_ip specifies the IP address of the machine running the collector application.

collector_port specifies the port on which the collector application is listening.

code_2_monitor specifies all the functions and methods which should receive instrumentation. Each function or method is specified by its absolute name and the kind of instrumentation it should receive.
`["absoult_func_or_method_name", "instrumentation_kind"]`

In the following the different instrumentation kinds are explained:

ExternCrateItem defines the import statement. This has to be present in the config file at all times for tRust to work correctly.

GlobalScope defines which function in the application should be used for global initialization and finalization.

LocalScope defines functions and methods which introduces a new thread-local scope.

InstCallForFunction defines functions which should receive measurement instrumentation calls.

InstCallForMethod defines methods which should receive measurement instrumentation calls.

APPENDIX B

Experiments

In the following three extracts of the stored event data one for each benchmark application are presented. In all three tables the columns for IP address and process ID are omitted.

time_stamp	counter	thread_id	absolute_path	description	ast_depth	source_file	lines_begin	lines_end
8819245541627	1	ThreadId(1)	main	GLOBAL_BEGIN	2	examples/fractal.rs	66	72
8819245740176	2	ThreadId(1)	par_iter_mut	BEGIN	8	examples/fractal.rs	30	30
8819245932493	3	ThreadId(1)	par_iter_mut	END	8	examples/fractal.rs	30	30
8819246984967	4	ThreadId(1)	join_context	BEGIN	14	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/rayon-1.0.3/src/iter/plumbing/mod.rs	409	415
8819257303714	1	ThreadId(7)	join_context	LOCAL_BEGIN	14	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/rayon-1.0.3/src/iter/plumbing/mod.rs	409	415
8819257311026	1	ThreadId(6)	join_context	LOCAL_BEGIN	14	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/rayon-1.0.3/src/iter/plumbing/mod.rs	409	415
8819257408429	2	ThreadId(6)	join_context	BEGIN	14	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/rayon-1.0.3/src/iter/plumbing/mod.rs	409	415
8819257522508	2	ThreadId(7)	join_context	BEGIN	14	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/rayon-1.0.3/src/iter/plumbing/mod.rs	409	415
8819257803673	1	ThreadId(9)	join_context	LOCAL_BEGIN	14	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/rayon-1.0.3/src/iter/plumbing/mod.rs	409	415
8819258106241	1	ThreadId(8)	join_context	LOCAL_BEGIN	14	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/rayon-1.0.3/src/iter/plumbing/mod.rs	409	415
8819258124868	2	ThreadId(9)	join_context	BEGIN	14	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/rayon-1.0.3/src/iter/plumbing/mod.rs	409	415
8819258219048	1	ThreadId(11)	join_context	LOCAL_BEGIN	14	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/rayon-1.0.3/src/iter/plumbing/mod.rs	409	415
8819258372139	1	ThreadId(10)	join_context	LOCAL_BEGIN	14	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/rayon-1.0.3/src/iter/plumbing/mod.rs	409	415
8819258379742	2	ThreadId(8)	join_context	BEGIN	14	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/rayon-1.0.3/src/iter/plumbing/mod.rs	409	415
8819258457598	1	ThreadId(12)	join_context	LOCAL_BEGIN	14	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/rayon-1.0.3/src/iter/plumbing/mod.rs	409	415
8819258511719	2	ThreadId(11)	join_context	BEGIN	14	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/rayon-1.0.3/src/iter/plumbing/mod.rs	409	415
8819258731144	2	ThreadId(10)	join_context	BEGIN	14	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/rayon-1.0.3/src/iter/plumbing/mod.rs	409	415
8819258842097	2	ThreadId(12)	join_context	BEGIN	14	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/rayon-1.0.3/src/iter/plumbing/mod.rs	409	415
8819259086527	1	ThreadId(13)	join_context	LOCAL_BEGIN	14	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/rayon-1.0.3/src/iter/plumbing/mod.rs	409	415
8819259166533	1	ThreadId(14)	join_context	LOCAL_BEGIN	14	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/rayon-1.0.3/src/iter/plumbing/mod.rs	409	415
8819259399442	1	ThreadId(15)	join_context	LOCAL_BEGIN	14	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/rayon-1.0.3/src/iter/plumbing/mod.rs	409	415
8819259554293	2	ThreadId(13)	join_context	BEGIN	14	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/rayon-1.0.3/src/iter/plumbing/mod.rs	409	415
8819259573769	1	ThreadId(16)	join_context	LOCAL_BEGIN	14	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/rayon-1.0.3/src/iter/plumbing/mod.rs	409	415
...	...	...	...	...	...	...	...	...
8831787502736	1	ThreadId(131)	join_context	LOCAL_BEGIN	14	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/rayon-1.0.3/src/iter/plumbing/mod.rs	409	415
8831945221673	2	ThreadId(124)	join_context	LOCAL_END	14	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/rayon-1.0.3/src/iter/plumbing/mod.rs	409	415
8831947809312	1	ThreadId(132)	join_context	LOCAL_BEGIN	14	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/rayon-1.0.3/src/iter/plumbing/mod.rs	409	415
8831947935250	2	ThreadId(132)	join_context	BEGIN	14	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/rayon-1.0.3/src/iter/plumbing/mod.rs	409	415
8831949591740	1	ThreadId(133)	join_context	LOCAL_BEGIN	14	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/rayon-1.0.3/src/iter/plumbing/mod.rs	409	415
8832361650791	2	ThreadId(129)	join_context	LOCAL_END	14	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/rayon-1.0.3/src/iter/plumbing/mod.rs	409	415
8832362127743	1	ThreadId(134)	join_context	LOCAL_BEGIN	14	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/rayon-1.0.3/src/iter/plumbing/mod.rs	409	415
8832931734879	2	ThreadId(130)	join_context	LOCAL_END	14	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/rayon-1.0.3/src/iter/plumbing/mod.rs	409	415
8832932544513	1	ThreadId(135)	join_context	LOCAL_BEGIN	14	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/rayon-1.0.3/src/iter/plumbing/mod.rs	409	415
8832982202328	2	ThreadId(131)	join_context	LOCAL_END	14	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/rayon-1.0.3/src/iter/plumbing/mod.rs	409	415
8832983208198	2	ThreadId(134)	join_context	LOCAL_END	14	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/rayon-1.0.3/src/iter/plumbing/mod.rs	409	415
8833131678987	2	ThreadId(133)	join_context	LOCAL_END	14	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/rayon-1.0.3/src/iter/plumbing/mod.rs	409	415
8833695404530	2	ThreadId(135)	join_context	LOCAL_END	14	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/rayon-1.0.3/src/iter/plumbing/mod.rs	409	415
8833696856259	5	ThreadId(1)	join_context	END	14	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/rayon-1.0.3/src/iter/plumbing/mod.rs	409	415
8833705323059	6	ThreadId(1)	main	GLOBAL_END	2	examples/fractal.rs	66	72

Table B.1: Extract of the event data from the fractal experiment

time_stamp	counter	thread_id	absolute_path	description	ast_depth	source_file	lines_begin	lines_end
15102611099457	1	ThreadId(1)	main	GLOBAL_BEGIN	2	examples/vanitykey.rs	105	122
15102611233964	2	ThreadId(1)	std::thread::spawn	BEGIN	6	examples/vanitykey.rs	35	52
15102611365739	4	ThreadId(1)	std::thread::spawn	BEGIN	6	examples/vanitykey.rs	60	77
15102611443530	1	ThreadId(3)	std::thread::spawn	LOCAL_BEGIN	6	examples/vanitykey.rs	35	52
15102611497545	5	ThreadId(1)	std::thread::spawn	END	6	examples/vanitykey.rs	60	77
15102611602995	2	ThreadId(3)	recv	BEGIN	13	examples/vanitykey.rs	39	39
15102611687252	6	ThreadId(1)	std::thread::spawn	BEGIN	6	examples/vanitykey.rs	85	102
15102611810325	3	ThreadId(3)	recv	BEGIN	8	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/crossbeam-channel-0.3.8/src/channel.rs	698	698
15102611878125	7	ThreadId(1)	std::thread::spawn	END	6	examples/vanitykey.rs	85	102
15102611919797	1	ThreadId(5)	std::thread::spawn	LOCAL_BEGIN	6	examples/vanitykey.rs	60	77
15102612044870	8	ThreadId(1)	send	BEGIN	8	examples/vanitykey.rs	11	11
15102612109340	1	ThreadId(7)	std::thread::spawn	LOCAL_BEGIN	6	examples/vanitykey.rs	85	102
15102612121082	2	ThreadId(5)	recv	BEGIN	13	examples/vanitykey.rs	64	64
15102612185112	9	ThreadId(1)	send	BEGIN	8	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/crossbeam-channel-0.3.8/src/channel.rs	390	390
15102612273126	2	ThreadId(7)	recv	BEGIN	13	examples/vanitykey.rs	89	89
15102612339682	3	ThreadId(5)	recv	BEGIN	8	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/crossbeam-channel-0.3.8/src/channel.rs	698	698
15102612459231	10	ThreadId(1)	send	END	8	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/crossbeam-channel-0.3.8/src/channel.rs	390	390
15102612466290	4	ThreadId(3)	recv	END	8	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/crossbeam-channel-0.3.8/src/channel.rs	698	698
15102612561254	3	ThreadId(7)	recv	BEGIN	8	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/crossbeam-channel-0.3.8/src/channel.rs	698	698
15102612680161	5	ThreadId(3)	recv	END	13	examples/vanitykey.rs	39	39
15102612690849	11	ThreadId(1)	send	END	8	examples/vanitykey.rs	11	11
15102612816204	12	ThreadId(1)	send	BEGIN	8	examples/vanitykey.rs	11	11
15102612897457	13	ThreadId(1)	send	BEGIN	8	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/crossbeam-channel-0.3.8/src/channel.rs	390	390
15102612964779	4	ThreadId(5)	recv	END	8	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/crossbeam-channel-0.3.8/src/channel.rs	698	698
15102612974639	14	ThreadId(1)	send	END	8	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/crossbeam-channel-0.3.8/src/channel.rs	390	390
15102613072827	5	ThreadId(5)	recv	END	13	examples/vanitykey.rs	64	64
15102613109967	15	ThreadId(1)	send	END	8	examples/vanitykey.rs	11	11
15102613169290	6	ThreadId(3)	recv	BEGIN	13	examples/vanitykey.rs	39	39
15102613262615	16	ThreadId(1)	send	BEGIN	8	examples/vanitykey.rs	11	11
15102613328427	7	ThreadId(3)	recv	BEGIN	8	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/crossbeam-channel-0.3.8/src/channel.rs	698	698
15102613378912	17	ThreadId(1)	send	BEGIN	8	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/crossbeam-channel-0.3.8/src/channel.rs	390	390
15102613433520	6	ThreadId(5)	recv	BEGIN	13	examples/vanitykey.rs	64	64
15102613447466	8	ThreadId(3)	recv	END	8	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/crossbeam-channel-0.3.8/src/channel.rs	698	698
15102613522809	18	ThreadId(1)	send	END	8	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/crossbeam-channel-0.3.8/src/channel.rs	390	390
15102613591581	9	ThreadId(3)	recv	END	13	examples/vanitykey.rs	39	39
15102613617474	7	ThreadId(5)	recv	BEGIN	8	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/crossbeam-channel-0.3.8/src/channel.rs	698	698
15102613662654	19	ThreadId(1)	send	END	8	examples/vanitykey.rs	11	11
...	...	...	...	...	...	...	...	...

Table B.2: Extract of the event data from the vanity key experiment

time_stamp	counter	thread_id	absolute_path	description	ast_depth	source_file	lines_begin	lines_end
135346766088904	1	ThreadId(1)	main	GLOBAL_BEGIN	2	examples/pagerank.rs	8	200
135346766235263	2	ThreadId(1)	timely::execute_from_args	BEGIN	7	examples/pagerank.rs	9	198
135346779188063	3	ThreadId(1)	timely::execute_from_args	END	7	examples/pagerank.rs	9	198
135346779657673	1	ThreadId(5)	timely::execute_from_args	LOCAL_BEGIN	7	examples/pagerank.rs	9	198
135346779685915	1	ThreadId(6)	timely::execute_from_args	LOCAL_BEGIN	7	examples/pagerank.rs	9	198
135346779708353	1	ThreadId(7)	timely::execute_from_args	LOCAL_BEGIN	7	examples/pagerank.rs	9	198
135346784083029	2	ThreadId(7)	send	BEGIN	16	examples/pagerank.rs	181	181
135346784115904	2	ThreadId(5)	send	BEGIN	16	examples/pagerank.rs	181	181
135346784145464	2	ThreadId(6)	send	BEGIN	16	examples/pagerank.rs	181	181
135346784300237	3	ThreadId(7)	send	END	16	examples/pagerank.rs	181	181
135346784386752	3	ThreadId(5)	send	END	16	examples/pagerank.rs	181	181
135346784466515	3	ThreadId(6)	send	END	16	examples/pagerank.rs	181	181
135346784595838	4	ThreadId(7)	send	BEGIN	10	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/timely_communication-0.8.0/src/allocator/process.rs	106	106
135346784644335	4	ThreadId(5)	send	BEGIN	10	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/timely_communication-0.8.0/src/allocator/process.rs	106	106
135346784719418	4	ThreadId(6)	send	BEGIN	10	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/timely_communication-0.8.0/src/allocator/process.rs	106	106
135346784930350	5	ThreadId(7)	send	END	10	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/timely_communication-0.8.0/src/allocator/process.rs	106	106
135346785002327	5	ThreadId(5)	send	END	10	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/timely_communication-0.8.0/src/allocator/process.rs	106	106
135346785133497	5	ThreadId(6)	send	END	10	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/timely_communication-0.8.0/src/allocator/process.rs	106	106
...	...	...	...	...	...	...	...	...
135346790580289	20	ThreadId(6)	send	BEGIN	16	examples/pagerank.rs	191	191
135346790742408	20	ThreadId(7)	send	BEGIN	16	examples/pagerank.rs	191	191
135346790809692	20	ThreadId(5)	send	BEGIN	16	examples/pagerank.rs	191	191
135346790885283	21	ThreadId(6)	send	END	16	examples/pagerank.rs	191	191
135346791068863	21	ThreadId(7)	send	END	16	examples/pagerank.rs	191	191
135346791185827	21	ThreadId(5)	send	END	16	examples/pagerank.rs	191	191
135346791479357	22	ThreadId(6)	send	BEGIN	16	examples/pagerank.rs	192	192
135346791635678	22	ThreadId(7)	send	BEGIN	16	examples/pagerank.rs	192	192
135346791704087	22	ThreadId(5)	send	BEGIN	16	examples/pagerank.rs	192	192
135346792544283	23	ThreadId(6)	send	END	16	examples/pagerank.rs	192	192
135346792776098	23	ThreadId(7)	send	END	16	examples/pagerank.rs	192	192
135346792790100	23	ThreadId(5)	send	END	16	examples/pagerank.rs	192	192
...	...	...	...	...	...	...	...	...
135355143246504	33318	ThreadId(7)	send	BEGIN	10	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/timely_communication-0.8.0/src/allocator/process.rs	106	106
135355143338154	33732	ThreadId(6)	send	BEGIN	10	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/timely_communication-0.8.0/src/allocator/process.rs	106	106
135355143390904	33940	ThreadId(5)	send	BEGIN	10	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/timely_communication-0.8.0/src/allocator/process.rs	106	106
135355143484226	33319	ThreadId(7)	send	END	10	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/timely_communication-0.8.0/src/allocator/process.rs	106	106
135355143593585	33733	ThreadId(6)	send	END	10	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/timely_communication-0.8.0/src/allocator/process.rs	106	106
135355143635724	33941	ThreadId(5)	send	END	10	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/timely_communication-0.8.0/src/allocator/process.rs	106	106
135355143782434	33734	ThreadId(6)	send	BEGIN	10	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/timely_communication-0.8.0/src/allocator/process.rs	106	106
135355143834537	33320	ThreadId(7)	send	BEGIN	10	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/timely_communication-0.8.0/src/allocator/process.rs	106	106
135355143851795	33942	ThreadId(5)	send	BEGIN	10	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/timely_communication-0.8.0/src/allocator/process.rs	106	106
135355143959577	33735	ThreadId(6)	send	END	10	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/timely_communication-0.8.0/src/allocator/process.rs	106	106
135355144033779	33321	ThreadId(7)	send	END	10	/.../.cargo/registry/src/github.com-1ecc6299db9ec823/timely_communication-0.8.0/src/allocator/process.rs	106	106

Table B.3: Extract of the event data from the PageRank experiment

List of Figures

List of Tables

Listings

Bibliography

[1] AMD Takes High-Performance Datacenter Computing to the Next Horizon. https://www.amd.com/en/press-releases/2018-11-06-amd-takes-high-performance-datacenter-computing-to-the-next-horizon. Accessed: 2019-05-27.

[2] Crate crossbeam. https://docs.rs/crossbeam/0.7.1/crossbeam/index.html. Accessed: 2019-06-01.

[3] Crate Rayon. https://docs.rs/rayon/1.0.3/rayon/. Accessed: 2019-05-30.

[4] Crate Tokio. https://docs.rs/tokio/0.1.20/tokio/. Accessed: 2019-05-30.

[5] Enum syntax::ext::base::SyntaxExtension. https://doc.rust-lang.org/nightly/nightly-rustc/syntax/ext/base/enum.SyntaxExtension.html. Accessed: 2019-06-03.

[6] Intel Announces Broadest Product Portfolio for Moving, Storing and Processing Data. https://newsroom.intel.com/news-releases/intel-data-centric-launch/. Accessed: 2019-05-27.

[7] Rust language. https://research.mozilla.org/rust/. Accessed: 2019-05-30.

[8] Rustc Guide - High-level overview of the compiler source. https://rust-lang.github.io/rustc-guide/high-level-overview.html. Accessed: 2019-06-03.

[9] rustup: the Rust toolchain installer - README. https://github.com/rust-lang/rustup.rs/blob/master/README.md. Accessed: 2019-06-04.

[10] Stack Overflow - Developer Survey Results 2019. https://insights.stackoverflow.com/survey/2019. Accessed: 2019-06-04.

[11] The Rust Programming Language. https://github.com/rust-lang/rust. Accessed: 2019-06-03.

[12] The Rust Standard Library Documentation - JoinHandle. https://doc.rust-lang.org/std/thread/struct.JoinHandle.html. Accessed: 2019-06-04.

[13] The Rust Standard Library Documentation - Primitive Type unit. https://doc.rust-lang.org/std/primitive.unit.html. Accessed: 2019-06-04.

[14] The Unstable Book - plugin. https://doc.rust-lang.org/unstable-book/language-features/plugin.html. Accessed: 2019-06-03.

[15] Wolfram Mathworld - Mandelbrot Set. `http://mathworld.wolfram.com/MandelbrotSet.html`. Accessed: 2019-05-30.

[16] *Score-P User Manual*, 4.1 edition, October 2018. Accessed: 2019-02-25.

[17] *Vampir 9 - User Manual*, 9.5 edition, June 2018. Accessed: 2019-05-03.

[18] L. Adhianto, S. Banerjee, M. W. Fagan, M. Krentel, G. Marin, J. M. Mellor-Crummey, and N. R. Tallent. HPCTOOLKIT: tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[19] J.-P. Aumasson, S. Neves, Z. Wilcox-O'Hearn, and C. Winnerlein. Blake2: simpler, smaller, fast as md5. In *International Conference on Applied Cryptography and Network Security*, pages 119–135. Springer, 2013.

[20] B. Barney and L. L. N. Laboratory. Introduction to Parallel Computing. `https://computing.llnl.gov/tutorials/parallel_comp/`. Accessed: 2019-04-26.

[21] B. Barney and L. L. N. Laboratory. Message Passing Interface (MPI). `https://computing.llnl.gov/tutorials/mpi/`. Accessed: 2019-04-26.

[22] B. Barney and L. L. N. Laboratory. POSIX Threads Programming. `https://computing.llnl.gov/tutorials/pthreads/`. Accessed: April 2019-04-29.

[23] R. Bell, A. D. Malony, and S. Shende. Paraprof: A portable, extensible, and scalable tool for parallel performance profile analysis. In H. Kosch, L. Böszörményi, and H. Hellwagner, editors, *Euro-Par 2003. Parallel Processing, 9th International Euro-Par Conference, Klagenfurt, Austria, August 26-29, 2003. Proceedings*, volume 2790 of *Lecture Notes in Computer Science*, pages 17–26. Springer, 2003.

[24] S. Benedict, V. Petkov, and M. Gerndt. PERISCOPE: an online-based distributed performance analysis tool. In M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel, editors, *Tools for High Performance Computing 2009 - Proceedings of the 3rd International Workshop on Parallel Tools for High Performance Computing, September 2009, ZIH, Dresden*, pages 1–16. Springer, 2009.

[25] D. E. Culler. Dataflow architectures. *Annual review of computer science*, 1(1):225–253, 1986.

[26] L. Dagum and R. Menon. Openmp: An industry-standard api for shared-memory programming. *Computing in Science & Engineering*, (1):46–55, 1998.

[27] Department of Computer and Information Science, University of Oregon Advanced Computing Laboratory, LANL, NM Research Centre Juelich, ZAM, Germany. *TAU User Guide*, 2.28.1 edition, April 2019. Accessed: 2019-05-03.

[28] D. Eschweiler, M. Wagner, M. Geimer, A. Knüpfer, W. E. Nagel, and F. Wolf. Open trace format 2: The next generation of scalable trace formats and support libraries. In K. D. Bosschere, E. H. D'Hollander, G. R. Joubert, D. A. Padua, F. J. Peters, and M. Sawyer, editors, *Applications, Tools and Techniques on the Road to Exascale Computing, Proceedings of the conference ParCo 2011, 31 August - 3 September 2011, Ghent, Belgium*, volume 22 of *Advances in Parallel Computing*, pages 481–490. IOS Press, 2011.

[29] V. Gajinov, S. Stipic, I. Eric, O. S. Unsal, E. Ayguadé, and A. Cristal. Dash: A benchmark suite for hybrid dataflow and shared memory programming models. *Parallel Computing*, 45:18–48, 2015.

[30] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr. The scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, 2010.

[31] S. Glavina. Lock-free Rust: Crossbeam in 2019. `https://stjepang.github.io/2019/01/29/lock-free-rust-crossbeam-in-2019.html`. Accessed: 2019-05-30.

[32] R. Harper. *Abstract Syntax*, page 3–11. Cambridge University Press, 2 edition, 2016.

[33] A. Hondroudakis. Performance analysis tools for parallel programs. *Edinburgh Parallel Computer Centre, The University of Edinburgh*, 1995.

[34] R. Jung, J. Jourdan, R. Krebbers, and D. Dreyer. Rustbelt: securing the foundations of the rust programming language. *PACMPL*, 2(POPL):66:1–66:34, 2018.

[35] S. Klabnik and C. Nichols. *The Rust Programming Language*. No Starch Press, 2018.

[36] A. Knüpfer, R. Brendel, H. Brunst, H. Mix, and W. E. Nagel. Introducing the open trace format (OTF). In V. N. Alexandrov, G. D. van Albada, P. M. A. Sloot, and J. J. Dongarra, editors, *Computational Science - ICCS 2006, 6th International Conference, Reading, UK, May 28-31, 2006, Proceedings, Part II*, volume 3992 of *Lecture Notes in Computer Science*, pages 526–533. Springer, 2006.

[37] A. Knüpfer, C. Rössel, D. an Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. D. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf. Score-p: A joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir. In H. Brunst, M. S. Müller, W. E. Nagel, and M. M. Resch, editors, *Tools for High Performance Computing 2011 - Proceedings of the 5th International Workshop on Parallel Tools for High Performance Computing, ZIH, Dresden, September 2011*, pages 79–91. Springer, 2011.

[38] D. Kranzlmüller, S. Grabner, and J. Volkert. Debugging with the MAD environment. *Parallel Computing*, 23(1-2):199–217, 1997.

[39] J. Labarta, J. Giménez, E. Martínez, P. González, H. Servat, G. Llort, and X. Aguilar. Scalability of tracing and visualization tools. In G. R. Joubert, W. E. Nagel, F. J. Peters, O. G. Plata, P. Tirado, and E. L. Zapata, editors, *Parallel Computing: Current & Future Issues of High-End Computing, Proceedings of the International Conference ParCo 2005, 13-16 September 2005, Department of Computer Architecture, University of Malaga, Spain*, volume 33 of *John von Neumann Institute for Computing Series*, pages 869–876. Central Institute for Applied Mathematics, Jülich, Germany, 2005.

[40] B. B. Mandelbrot. *The fractal geometry of nature*, volume 173. WH freeman New York, 1983.

[41] N. Matsakis. Rayon: data parallelism in Rust. `http://smallcultfollowing.com/babysteps/blog/2015/12/18/rayon-data-parallelism-in-rust/`. Accessed: 2019-05-30.

[42] N. D. Matsakis and F. S. K. II. The rust language. In M. Feldman and S. T. Taft, editors, *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology, HILT 2014, Portland, Oregon, USA, October 18-21, 2014*, pages 103–104. ACM, 2014.

[43] F. McSherry. PageRank example. https://github.com/TimelyDataflow/timely-dataflow/blob/59f438d338e3395b547453c3104c6ffa0ac8b52e/timely/examples/pagerank.rs. Accessed: 2019-06-04.

[44] F. McSherry. Timely Dataflow. https://timelydataflow.github.io/timely-dataflow/introduction.html. Accessed: 2019-05-30.

[45] Message Passing Interface Forum. MPI: A message-passing interface standard, version 3.1. Technical report, High Performance Computing Center Stuttgart (HLRS), June 2015.

[46] M. Mitchell, J. Oldham, and A. Samuel. *Advanced linux programming*. New Riders Publishing, 2001.

[47] U. Mukhopadhyay, A. Skjellum, O. Hambolu, J. Oakley, L. Yu, and R. R. Brooks. A brief survey of cryptocurrency systems. In *14th Annual Conference on Privacy, Security and Trust, PST 2016, Auckland, New Zealand, December 12-14, 2016*, pages 745–752. IEEE, 2016.

[48] M. S. Müller, A. Knüpfer, M. Jurenz, M. Lieber, H. Brunst, H. Mix, and W. E. Nagel. Developing scalable applications with vampir, vampirserver and vampirtrace. In C. H. Bischof, H. M. Bücker, P. Gibbon, G. R. Joubert, T. Lippert, B. Mohr, and F. J. Peters, editors, *Parallel Computing: Architectures, Algorithms and Applications, ParCo 2007, Forschungszentrum Jülich and RWTH Aachen University, Germany, 4-7 September 2007*, volume 15 of *Advances in Parallel Computing*, pages 637–644. IOS Press, 2007.

[49] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In M. Kaminsky and M. Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 439–455. ACM, 2013.

[50] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 1:69–80, 1996.

[51] N. I. of Standards and Technology. Fips pub 180-2: Secure hash standard. *Federal Information Processing Standards Publication*, 2002.

[52] N. I. of Standards and Technology. Fips pub 202: Sha-3 standard: Permutation-based hash and extendable-output functions. *Federal Information Processing Standards Publication*, 2015.

[53] OpenMP Architecture Review Board. Openmp application program interface, version 5.0. Technical report, November 2018. Accessed: 2019-03-27.

[54] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[55] S. Parker, J. Mellor-Crummey, D. H. Ahn, H. Jagode, H. Brunst, S. Shende, A. D. Malony, D. Lecomber, J. V. DelSignore Jr, R. Tschüter, et al. 2 performance analysis and debugging tools at scale. *Exascale Scientific Applications: Scalability and Performance Portability*, page 17, 2017.

[56] S. Shende and A. D. Malony. The tau parallel performance system. *IJHPCA*, 20(2):287–311, 2006.

[57] C. Team. Vanity addresses. https://medium.com/coinbundle/vanity-addresses-857fa4fb44be, 2018. Accessed: 2019-05-25.

[58] A. Turon. Fearless Concurrency with Rust. `https://blog.rust-lang.org/2015/04/10/Fearless-Concurrency.html`, 2015. Accessed: 2019-06-04.

[59] F. Wolf and B. Mohr. Epilog binary trace-data format (version 1.1), 2004.