Parallel Programming in C++, Go, Julia and Rust

Konstantin Papesh

BACHELORARBEIT

Nr. 1720307111-A

eingereicht am
Fachhochschul-Bachelorstudiengang

Software Engineering

in Hagenberg

im Mai 2020

Advisor:

Peter Kulczycki

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

This printed thesis is identical with the electronic version submitted.

Hagenberg, May 30, 2020

Konstantin Papesh

Contents

Preface

> Don't cry because it happened!
> Smile because it's over!
>
> Common saying, slightly altered

This thesis was written in more or less turbulent times. COVID-19 hit Austria and the globe, resulting in a nearly surreal world where people were not able to get together, shops and restaurants were closed and everything was on lockdown. While in theory that meant that a lot of time could be spent on writing this document, it was harder than ever to get oneself to write. Spending the whole day, the whole week in your apartment may sound easy, but even though you may not realise it, it differs greatly from your normal weekday.

I still hope that no quality was lost due to circumstances happening while researching and writing this document. And I hope, dear reader, that you are able to get up, go out and get yourself a coffee without any restrictions. Enjoy it, it may not be always that easy.

Furthermore, I would like thank my advisor, Peter Kulczycki, for mentoring me and helping me write this thesis. I also express gratitude to the whole Hagenberg teaching staff, which did an superb job of lecturing and conveying the inner workings of software engineering and everything that surrounds it. Lastly, thanks to the companies Dallmayr and J.Hornig, which made writing this thesis and studying in general a lot easier.

Abstract

Parallel Programming in C++, Go, Julia and Rust compares four languages in how they handle parallel programming and what they offer in functionality to aid the programmer in writing a thread-safe program. Comparisons will be done on *Multithreading*, *Message Passing* and *Memory Management*.

In every chapter, the general concept is introduced and explained. Then the implementation of the language will be described and visualised via code and graphs. Finally, the languages are compared between each other based on execution time, compile time, memory usage and ease of use.

Kurzfassung

Parallel Programming in C++, Go, Julia and Rust vergleicht vier Sprachen wie sie parallele Programmierung handhaben und welche Funktionalität sie dem Programmierer bieten, um effizient beim Schreiben eines thread-sicheren Programmes zu helfen. Verglichen werden *Multithreading*, *Nachrichtenaustausch* und *Speicherverwaltung*.

In jedem Kapitel wird das generelle Konzept vorgestellt und erklärt. Danach werden die Implementierungen der Sprachen beschrieben und mithilfe von Programmcode und Graphen visualisiert. Schlussendlich werden die Sprachen untereinander basierend auf Laufzeit, Kompilierzeit, Speicherauslastung und Handhabung verglichen.

Chapter 1

Introduction

1.1 Goals

The goal of this document is to provide a better understanding of current concepts of parallel programming in programming languages. What are the problems programmers face when implementing parallel programs, how do languages try to solve these problems, what features do they offer and how efficient these solutions are implemented.

While all languages aim to serve a general-purpose, they all have different philosophies while trying to accomplish that goal. C++ is by far the oldest language and therefore has a long list of things it has to be backwards compatible to, Rust has a rather unique memory model, Go has an unconventional approach to most things, for example, the lack of generics, and Julia is written like an interpreted language but still compiles its code to binary just-in-time.

1.2 Parallel Programming

Parallel programming describes the possibility to run two parts of the code at the same time. This must be differentiated from *concurrent* programming. Running a program concurrently from the user perspective may seem like a parallel program. On closer inspection, concurrently running threads are interweaved, meaning that they are switched from and to in time slices by the scheduler and that they are not running both at the same point in time. This means that concurrent programs can run on single-core systems. Contrary, parallel programs are actually executed at the same point in time and therefore require two or more cores. (Schmidt et al. 2017)

While parallel programming has always been a part of programming itself, it gained traction in recent years due to widespread usage of multi-core systems for conventional customers with the release of programming languages with concurrency in mind like Go in 2012, but also additions to existing languages like the *Task Parallel Library* to C# .NET 4.0 in 2010 (Microsoft 2013). These additions try to mitigate problems programmers face when working in a parallel environment like how to spawn multiple, simultaneous executing code pieces and how to share information between these parts of code.

This shift happened because waiting for processors to get faster was not a viable

option for program performance enhancement anymore. This is due to physical limitations being reached like heat dissipation and power consumption. Instead, current processors are not getting faster, instead, they are implementing more cores. In 2020, there are processors available with 6 cores and 12 threads, while their base frequency did not increase, staying at about 3 to 3.8 GHz still. But non-multithreaded programs can only make use of one of these threads. Therefore, the programmer or ideally, the programming language itself, must implement concurrency to effectively use today's CPUs. (Sutter 2005)

1.3 Structure of the thesis

In Chapter 2, all used languages are introduced, when they were released and what their primary objectives are. Chapter 3 compares the languages on the multithreading features *coroutines*, *tasks* and *threads*. Chapter 4 consists of a comparison of *channels*. *Locks*, *Condition Variables* and *Atomics* are compared in Chapter 5. A short summary of the thesis is given in Chapter 6. In Appendix A, the code used for the benchmarks in Chapter 3, Chapter 4 and Chapter 5 are listed. Appendix B lists the content of the included CD.

Chapter 2

Language Introductions

2.1 C++

C++ was designed as a general-purpose programming language and was first released in 1985 but thanks to triennial updates to the standard new features and concepts are added regularly. Its most recent release as of this date is C++17 which added parallel algorithms to the *Standard Library* (ISO/IEC 2017). This library is included in virtually every C++ implementation and makes C++ a versatile language. Its roots start with the programming language *C* which was designed to work close to hardware and thus C++ also earned this trait. This makes C++ often used in embedded systems programming where speed and efficiency are necessary but also removes comfort features like garbage collection. Despite its age, C++ is still commonly used by developers, as a survey by Stack Overflow (2019) shows C++ being used by 23.5% of the participants.

C++ gained usable threading in its C++11 standard release which introduced numerous threading tools, foremost `std::thread` which allows spawning parallel threads in a standardised way.

While most of its compilers and documentation are available for free and are open-source, C++ is the only compared language which is standardised by ISO/IEC and whose specification can not be read for free.

```
#include <iostream>

int main() {
    std::cout << "Hello World!" << std::endl;
    return 0;
}
```

2.2 Rust

Rust first appeared in 2010 as a competitor to C++ with its main selling points being type safety and extended concurrency support (Rust Team 2019). It tries to be as fast and efficient as C++ while still accomplishing these goals. Rust tries to incorporate contemporary features such as a dependency manager, *Cargo*, and a formatter, *Rustfmt*, to provide consistent code style throughout the project, but else relies on established techniques (Rust Team 2019). Similarly to C++, Rust doesn't feature a garbage collector

but instead uses *Ownership* to guarantee memory safety (Klabnik and Nichols 2018). Most of its concurrency features were added in Rust 1.0, released in May 2015.

According to Stack Overflow (2019), Rust is the most loved language currently on the market.

```
fn main() {
    println!("Hello World!");
}
```

2.3 Go

Go first appeared in 2009 with its first version 1.0 released in March 2012. It was created at Google as an answer to more recent programming problems like multi-core processing, networking and cluster computing while still retaining relevance in general-purpose programming (Pike 2012).

While not having a primary ancestor language, it incorporated ideas from a multitude of languages, primary Alef, Oberon-2 and C. Its primary reference in regards to parallel programming comes from CSP, an ancestor to Alef. (Donovan and Kernighan 2015) Go's approach to parallel programming often differs from other languages, being primarily based on channel-based communication and lightweight, concurrent function calls called *goroutines*. Go already provided most threading utilities at its inception.

```
package main

import "fmt"

func main() {
    fmt.Println("Hello World!")
}
```

2.4 Julia

The most recent programming language compared in this thesis with its release in February 2012. Its vision is to be as fast as C but with a focus on mathematical computations and data visualization (Bezanson, Karpinski, et al. 2012). Contrary to the other programming languages compared, Julia is a dynamically typed language even though it is possible to specify the type of a variable. It is also the only programming language which will not compile to an executable, instead, Julia programs rely on the Julia runtime to run. Julia is written similarly to Python or other dynamic, interpreted languages while retaining the speed of compiled languages (Julia 2020b). Julia already provided most threading utilities at its inception.

```
println("Hello World!")
```

2.5 Test Setup

All benchmarks were executed on following system specs unless noted differently.

2.5.1 System Specifications

Operating System	KDE neon 5.18
Kernel Version	4.15.0
System Type	x64
Processor	Intel Core i5-2500K
RAM	8GB

2.5.2 Compiler Versions

Compiler	**Version**
g++	9.3.0
rustc	1.40.0
go	1.10.4
julia	1.3.1

2.5.3 Approach

Times were measured by the linux command line tool `multitime` combined with the parameters `-l`, `-q` and `-n 10`. `-l` will output memory information about the run, `-q` will omit the output produced by the benchmarked program and `-n` is the amount of runs. To ease benchmarking, the command and its parameters were *aliased* to `timebench`. For example, a benchmarking run of g++ was done by the command `timebench g++-9 example.cpp` which in turn expanded to `multitime -q -l -n 10 g++-9 example.cpp`.

C++ and Rust compilation times were measured without optimisations while the respective executables were benchmarked with optimisations enabled at level 2 for both languages. This was done by first compiling the programs with the flag `-O` for Rust and `-o2` for C++ and then benchmarking them. Go and Julia lack such compile optimisations. Compilation times of Julia were measured by simply calling `precompile([TASK], ())` once and measuring the time the runtime takes to compile the function. Execution times were measured inside of Julia by the provided `@time` command of Julia. This command was called 10 times and the average of all runs was calculated.

Some features tested required external libraries to be used by Rust. To just measure the compilation time of the file itself, another command was used to just append an empty line to the source file of Rust. Thus, cargo only compiled the changed source file again. For example, `timebench -r 'echo "" » ./src/main.rs'  /.cargo/bin/rustup run stable cargo build` was used to measure just the compile time of `main.rs`.

Chapter 3

Multithreading

3.1 Coroutines

Coroutines, in general, are functions that can suspend and pick up execution midway through the function itself. This means that coroutines have a state from which they continue at their next invocation. Coroutines do not spawn additional threads and thus are fairly quick compared to other multithreading methods described later in this chapter. (Rust Team 2020a)

3.1.1 C++

C++ will introduce coroutines in C++20 (ISO/IEC 2020). With the inclusion of coroutines into C++ three new keywords were introduced:

- `co_await` can be used to suspend the coroutine and return to the caller. The coroutine itself will wait on the value after the `co_await` and then continue on running. (ISO/IEC 2020)
- `co_yield` on the other hand works like a common yield-statement by returning the value after the `co_yield` keyword and suspending the current coroutine. (ISO/IEC 2020)
- `co_return` completes the execution of a coroutine returning a value. This keyword was introduced to be used as a replacement for `return` as `return` cannot be used in coroutines. (ISO/IEC 2020)

Functions are automatically compiled into coroutines by the compiler if they contain one of the mentioned keywords. (ISO/IEC 2020)

Listing 3.1: coroutines.cpp

```
#include <iostream>
#include <experimental/coroutine>

std::experimental::generator<int> generator(int n = 0) {
    while(true)
      co_yield n++;
}

int main()
```

```
{
    auto numbers = generator();
    for(int i = 0; i <= 20; i++) std::cout << numbers << std::endl;
    return 0;
}
```

3.1.2 Rust

In Rust, coroutines are called *generators* and are currently labelled a *experimental feature* of the compiler. Thus it is currently not possible to code generators in production Rust, instead, the nightly experimental compiler must be used. (Rust Team 2020a)

At this moment it is also not possible for generators to have arguments. Internally they are implemented as state machines so the whole function is split up into multiple states, with each state representing the lines of code between the `yield` expressions. (Rust Team 2020a)

There are however user-implementations available as *crates*.[1] Crates are the equivalent of modules in Rust.

Listing 3.2: coroutine.rs

```
#![feature(generators, generator_trait)]

use std::ops::{Generator, GeneratorState};
use std::pin::Pin;

fn main() {
    let mut generator = || {
        for i in 0..20 {
            yield i;
        }
        return;
    };

    loop {
        match Pin::new(&mut generator).resume() {
            GeneratorState::Yielded(x) => {println!("{}", x.to_string())}
            GeneratorState::Complete(()) => {break;},
        }
    }
}
```

3.1.3 Go

Go does not offer `yield` or *coroutines* as a functionality itself. But it is possible to mimic the functionality with goroutines and channels thanks to the blocking nature of the channel and the concurrency of the goroutine. For the programmer, it acts like a coroutine.

Listing 3.3: coroutine.go

```
package main
```

[1] For example *genawaiter*(https://crates.io/crates/genawaiter)

```go

import "fmt"

func numberator(limit int) <-chan int {
  channel := make(chan int)
  go func() {
    for i := 0; i < limit; i++ {
      channel <- i
    }
    fmt.Println("Yield finished, exiting")
    close(channel)
  }()
  return channel
}

func main() {
  for i := range numberator(20) {
    fmt.Println(i)
  }
}
```

3.1.4 Julia

In Julia, the keyword `yield` is used in the *Base* class to call a context switch (Julia 2020d). Therefore, to get the functionality of coroutines, one has to use a *channel*, similar to *Go*. These channels act like iterators but call back to the function which returns the next value, thus acting like a coroutine. This also means that one can use a *for-in* loop on the coroutine.

Listing 3.4: coroutine.jl

```julia
iterations = 20

function numberator(limit)
  Channel() do channel
    for i in 0:limit-1
      put!(channel, i)
    end
    println("Yield finished, exiting")
  end
end

n = numberator(iterations)

for i in n
  println(i)
end
```

3.1.5 Comparison

Create a stateful function which on each function call returns the next value ascending from 0 to 100000. For Rust, version 1.42.0-nightly has been used as coroutines are not on the stable branch yet and C++ is completely missing as g++ 9.3.0 does not support yield statements.

Language	Execution Time [S]	SD Time	Memory Usage [MB]	SD Memory
Rust	0.069	0.001	1,991	54
Go	0.234	0.006	2,392	39
Julia	0.222	0.001	6,097	0

Table 3.1: Coroutine execution metrics.

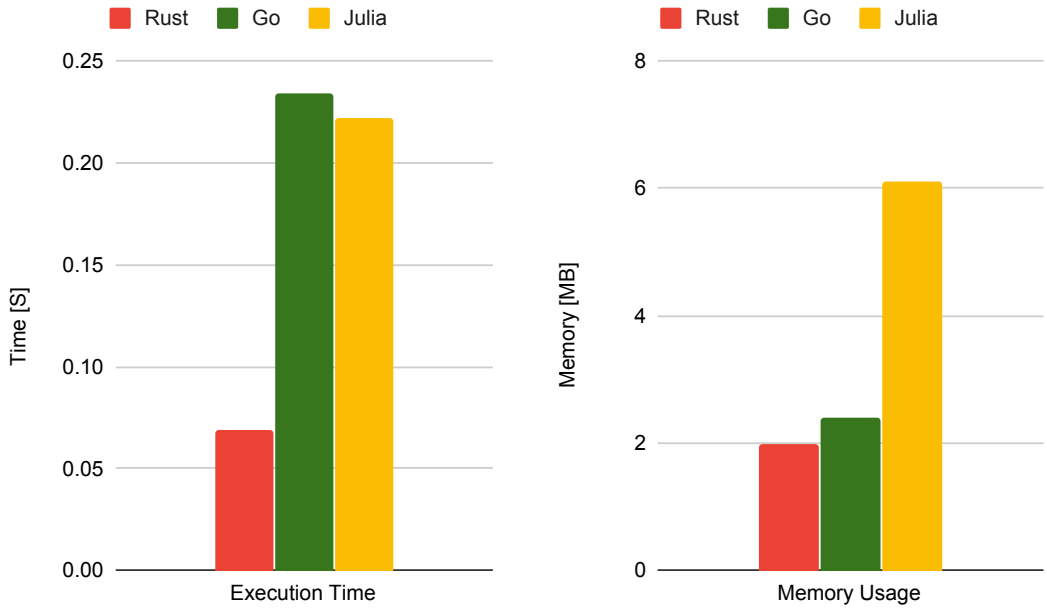

Figure 3.1: Coroutine execution graphs.

Language	Compile Time [S]	SD Time	Memory Usage [MB]	SD Memory
Rust	0.236	0.013	125,294	1,192
Go	0.074	0.006	17,944	373
Julia	0.202	0.01	148,144	110

Table 3.2: Coroutine compilation metrics.

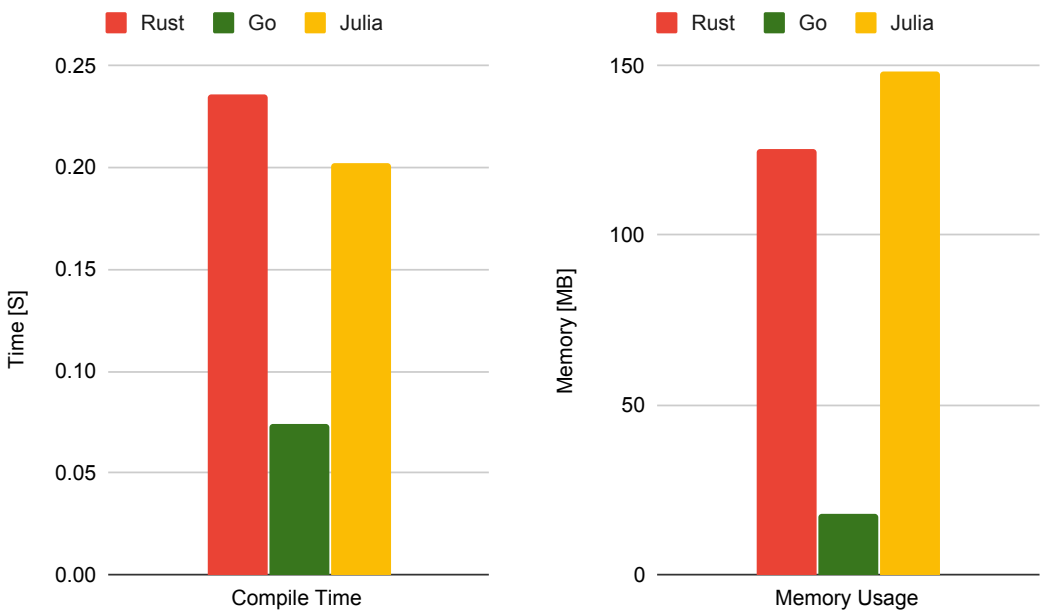

Figure 3.2: Coroutine compilation graphs.

Execution & Compilation

Executing coroutines shows that Rust takes the most efficient approach, both in memory and speed (see fig. 3.1 and tab. 3.1), executing the coroutine in just 0.07 seconds and using under 2MB of memory at max. Go and Julia both take the same time to execute, but Julia requires almost three times as much memory as Rust and Go, maxing out at 6MB.

Compared based on compiling (see fig. 3.2 and tab. 3.2), Go takes only a third of the compile time of the other languages with 0.074 seconds to compile the program once. Go also needs the least amount of memory with a max of 18MB used while Rust uses 125MB and Julia the most amount with over 150MB RAM used.

Ease of use

When comparing readability C++ and Rust offer the `yield` keyword which increases readability. C++ tries to not break backwards compatibility by prefixing all coroutine keywords with `co_` which is unconventional and may lead to confusion.

Go and Julia currently do not support `yield`-statements but this functionality can be sufficiently recreated by using other in-build tools like channels.

As for feature completion, all programming languages offer the same basic functionality except *Rust* which does not offer the possibility to give arguments to the coroutine.

3.2 Tasks

Tasks are functions which are executed concurrent or parallel from the calling function. The function call immediately returns while its execution continues. Additionally a *future* is provided which will return the computed return value as soon as it is available. (Schmidt et al. 2017)

3.2.1 C++

Tasks were introduced in the C++11 standard with the `<future>` header. It introduced functions and classes that allow for asynchronous function execution. For this the function `std::async` is introduced which is passed the callable object, its parameters and optionally a launch policy which determines how the function should be evaluated. In the current C++ 20 standard there are two `std::launch` policy flags available. (cppreference.com 2020a)

- `std::launch::async` will launch the given function asyncronously. This means internally a `std::thread` is spawned and the result of the function (an exception or a value) is then saved into the `std::future` which is returned by the `std::async` function. (cppreference.com 2020a)
- `std::launch::deferred` will *lazy evaluate* the function thus the function will be executed the first time the `std::future` is accessed. The function will be run synchronously on the requesting thread. The result will be saved in the future and any further requests will immediately return the saved value. (cppreference.com 2020a)

The standard allows additional flags in compiler implementations.(cppreference.com 2018) One of these additional flags may also be the default behaviour for `std::async`. It is also up to the compiler what to do when multiple flags are set. If no flags are set the behaviour is undefined. (cppreference.com 2020a)

Example

Listing 3.5: task.cpp

```
#include <iostream>
#include <atomic>
#include <future>
#include <sstream>
#include <chrono>

std::atomic_int i;

std::string printTime(std::string name) {
  auto curTime = std::time(0);
  std::stringstream id;
  id << std::this_thread::get_id();

    // Doing work
    std::this_thread::sleep_for((std::chrono::milliseconds(1000)));

```

```
17   return "This is the " + std::to_string(++i) + " invocation at time " + std::
       to_string(curTime) + " on thread " + id.str() + " with the name " + name;
18 }
19
20 int main()
21 {
22     auto deferred = std::async(std::launch::deferred, printTime, "deferred");
23     auto async = std::async(std::launch::async, printTime, "async");
24     std::cout << "Instantly continuing on with the main thread" << std::endl;
25
26     auto begin = std::chrono::high_resolution_clock::now();
27     std::cout << printTime("sync") << std::endl;
28     std::cout << "Time taken for sync in ms: " << std::chrono::duration_cast<std::
       chrono::milliseconds>(std::chrono::high_resolution_clock::now() - begin).count()
        << std::endl;
29
30     begin = std::chrono::high_resolution_clock::now();
31     std::cout << deferred.get() << std::endl;
32     std::cout << "Time taken for deferred in ms: " << std::chrono::duration_cast<std
       ::chrono::milliseconds>(std::chrono::high_resolution_clock::now() - begin).count
       () << std::endl;
33
34     begin = std::chrono::high_resolution_clock::now();
35     std::cout << async.get() << std::endl;
36     std::cout << "Time taken for async in microseconds: " << std::chrono::
       duration_cast<std::chrono::microseconds>(std::chrono::high_resolution_clock::now
       () - begin).count() << std::endl;
37 }
```

3.2.2 Rust

Rust only provides a trait called `std::future::Future` in its standard library. But this `Future` acts more as a guideline for libraries as it does not work the same way as the futures in other programming languages. Instead, it is only an asynchronous wrapper for functions and does not run asynchronously itself. (Rust Team 2020b)

Currently, *async-await* is only a *Minimum Viable Product* and is expected to be extended in the future. (Matsakis 2019)

`Futures` must be actively polled to complete (Rust Team 2020b). Thus, the user has to either implement a thread which polls the future or use an *async runtime*. These runtimes handle the `std::future::Future` and offer the user easier access to the result.

Listing 3.6: task.rs

```
use std::time::{SystemTime, UNIX_EPOCH};
use std::sync::atomic::{AtomicUsize, Ordering};
use futures::executor::{block_on, ThreadPool};
use futures::task::SpawnExt;

static I: AtomicUsize = AtomicUsize::new(0);

fn print_time(name: String) -> String {
    let time_now = SystemTime::now();
    let result = time_now.duration_since(UNIX_EPOCH).unwrap();
    I.fetch_add(1, Ordering::SeqCst);

    // Doing work
    std::thread::sleep(std::time::Duration::from_millis(1000));
    return format!("This is the {} invocation at time {:?} with the name {}", I.load
    (Ordering::SeqCst), time, name);
}

async fn async_main() {
    let pool = ThreadPool::new().unwrap();
    let task = pool.spawn_with_handle(async { print_time("async".to_string())});
    println!("Instantly continuing on with the main thread");

    let begin = std::time::Instant::now();
    println!("{:?}", sync);
    println!("Time taken for sync in ms: {}", now.elapsed().as_millis());

    begin = std::time::Instant::now();
    println!("{:?}", task.await.unwrap());
    println!("Time taken for async in microseconds: {}", now.elapsed().as_micros());
}

fn main() {
    block_on(async_main());
}
```

3.2.3 Go

In *Go* tasks are called *goroutines*. These are scheduled by the go runtime and may or may not run on another kernel thread. (Deshpande, Sponsler, and Weiss 2012) Compared to C++, Go does not offer the concept of *futures*, thus any return value from the original function will be discarded (Google 2019). To retrieve the return value of goroutines, channels must be used.

Goroutines itself are invoked by calling a function prefixed with the keyword *go*.

Example

Listing 3.7: task.go

```
package main

import "fmt"
import "time"
```

```
import "strconv"

var i = 0

func printTime(name string) string {
  i++
  result := time.Now().UnixNano() / 100000000000

  // Doing work
  time.Sleep(1000 * time.Millisecond)

  return "This is the " + strconv.Itoa(i) + " invocation at the time " + strconv.
    FormatInt(result, 10) + " with the name " + name
}


func main() {
  future := make(chan string)
    go func() { future <- printTime("async") }()
    fmt.Println("Instantly continuing on with the main thread")

    begin := time.Now()
  fmt.Println(printTime("sync"))
    fmt.Printf("Time taken for sync in ms: %v\n", time.Now().Sub(begin))

    begin = time.Now()

    fmt.Println(<- future)
    fmt.Printf("Time taken for async in ms: %v\n", time.Now().Sub(begin))
}
```

3.2.4 Julia

Julia only recently added *composable multi-threading* in July 2019 (Bezanson, Nash, and Pamnany 2019). Using a comparable syntax to *go*, function calls can easily be made asynchronous by prefixing them with `Threads.@spawn`. This macro will return a task which can be waited or fetched on. (Julia 2020c)

Currently, the `@spawn` macro is considered experimental (Julia 2020c) and also does not support task migration between system threads, meaning that a task started on one thread must continue to be scheduled on the same thread due to thread-local variables (Bezanson, Nash, and Pamnany 2019).

Listing 3.8: task.jl

```
import Dates
i = 0

function printTime(name)
  global i+=1
  moment=Dates.now()

  # Doing work
  sleep(1)
  return string("This is the ", i, " invocation at the time ", Dates.value(moment) ,
     " with the name " , name)
end
```

```
12
13 task = Threads.@spawn printTime("async")
14 println("Instantly continuing on with the main thread")
15
16 start = Dates.now()
17 println(printTime("sync"))
18 println("Time taken for sync in ms: ", Dates.now() - start)
19
20 start = Dates.now()
21 println(fetch(task))
22 println("Time taken for async in ms: ", Dates.now() - start)
```

3.2.5 Comparison

Spawn 10.000 tasks and calculate the square root of 1337. When finished spawning all tasks wait for the results of all tasks before exiting.

Language	Execution Time [S]	SD Time	Memory Usage [MB]	SD Memory
C++	0.249	0.003	88,054	43
Rust	0.021	0.003	3,778	127
Go	0.019	0.001	2,635	139
Julia	0.007	0.002	7,422	0

Table 3.3: Task execution metrics.

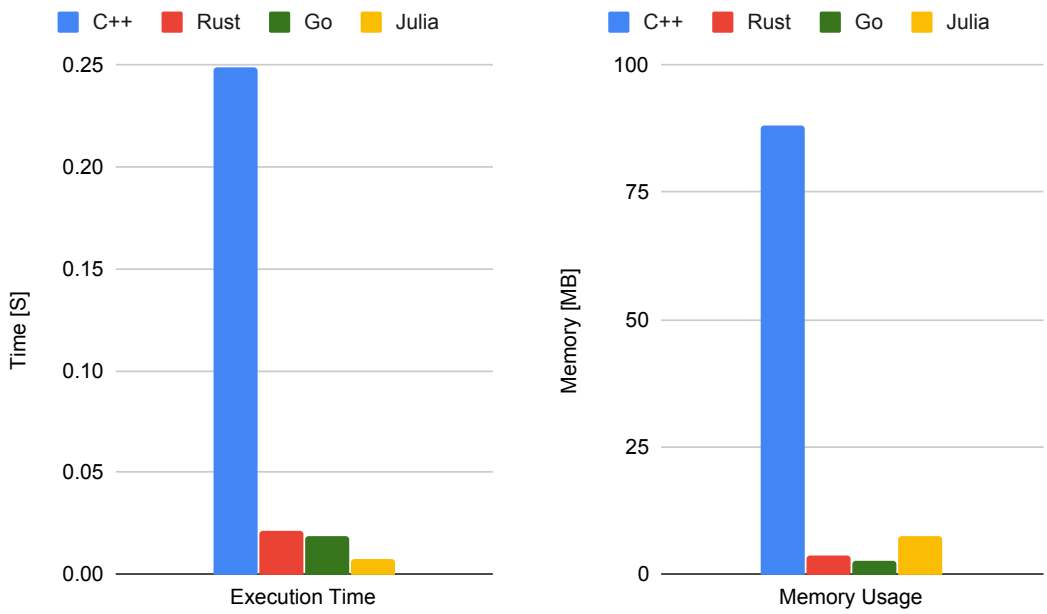

Figure 3.3: Task execution graphs.

Language	Compile Time [S]	SD Time	Memory Usage [MB]	SD Memory
C++	0.999	0.007	115,073	57
Rust	0.403	0.014	141,650	211
Go	0.08	0.005	15,435	401
Julia	0.198	0.014	156,240	72

Table 3.4: Task compilation metrics.

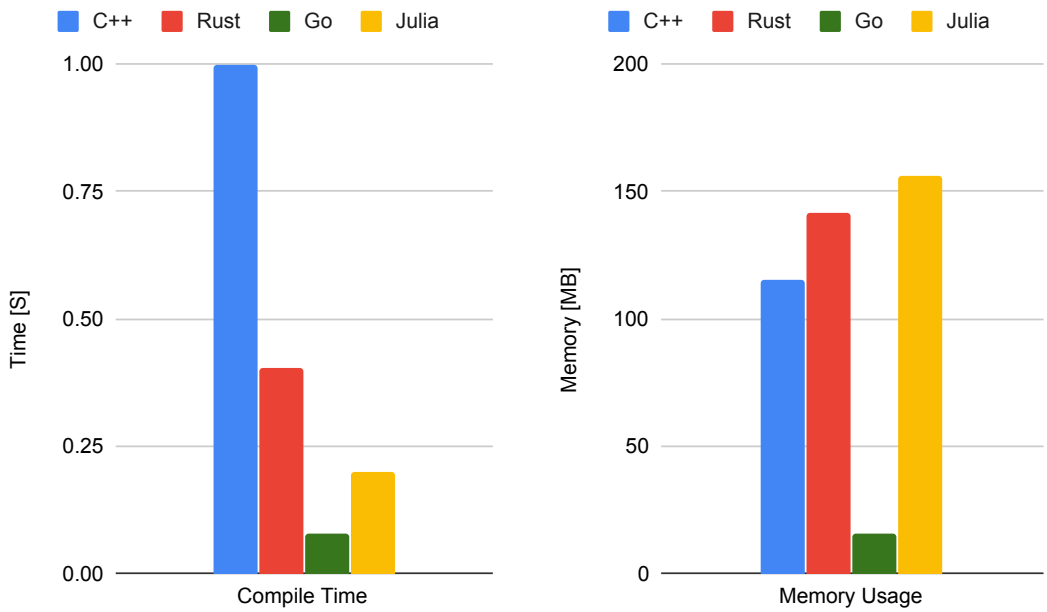

Figure 3.4: Task compilation graphs.

Execution & Compilation

When executing tasks, Julia provides the fastest execution time with just 0.007 seconds (see fig. 3.3 and tab. 3.3), given that the code is precompiled and executing inside the already running Julia runtime starting tasks is not time-consuming for Julia. Of the languages with standalone executables, Go provides the fastest execution time with 0.019 seconds, closely before Rust with 0.021 seconds. As C++ does not offer a dedicated thread pool and thus has to spawn threads to execute tasks. C++'s execution time is the longest at 0.249 seconds. The memory footprint of C++ is the highest with 88MB used when spawning tasks, much higher than the memory usages of the other languages.

Once again Go is the fastest compiling language with a compilation time of 0.080 seconds (see fig. 3.4 and tab. 3.4), followed by Julia with 0.198 seconds. Next is Rust with a total compile time of 0.403 seconds. The increase in compilation time is probably because of the used external library. Last is C++ with 1 second of compile time.

Memory usage differs greatly between Go and the other languages. While C++, Rust

and Julia have nearly the same max memory usage with 115MB, 140MB and 156MB, Go only maxes out at about 15MB.

Ease of use

Julia provides the easiest usage of tasks as functions itself must only be prefixed with `Threads.@spawn` to be executed concurrently, comparable to Go's approach of prefixing `go` to the function. But Go does not offer to directly retrieve the calculated value, instead, the user has to rely on *channels* which add another layer of complexity.

C++ offers an object-oriented `get()` to retrieve the value directly from the `Future`-object. It also offers the biggest variety of launch policies for tasks although this also adds ambiguity as it is up to the implementation how the default behaviour will be.

Rust's approach is the most unconventional and complex one with only providing the user with an interface for futures. The future itself will not be executed asynchronously like in other languages and instead must be actively polled by a user-implemented mechanism or a library. This tasks the user with finding and setting up a runtime in which the future can be executed without blocking the thread.

3.3 Threads

Threads can be seen as the next step up from *tasks*. While tasks and futures are mostly managed by the programming language itself (cppreference.com 2020a; Google 2019), threads are managed by the OS and must be requested from it. This means that the underlying OS must support multithreading. Threads must be differentiated from processes. Processes run in completely separate address spaces while threads have shared memory between them if they belong to the same process. (Schmidt et al. 2017) Users may prefer to use threads over tasks when there is a requirement to have different priority levels within the same process.

3.3.1 C++

C++ threads are directly mapped to OS-threads and are included via the `thread` header. The thread with the passed function is scheduled immediately and will start execution after the OS has spawned the requested thread. When compared with `std::async`, `std::thread` will always spawn an OS-thread and also does not return the original return value of the called function. (cppreference.com 2020e) Instead, either a `std::promise` or a shared variable has to be used.

Example

Listing 3.9: thread.cpp

```
1 #include <thread>
2 #include <string>
3 #include <iostream>
4
5 void sayHello(std::string name) {
6   std::this_thread::sleep_for((std::chrono::milliseconds(5000)));
```

```
  std::cout << "Good morning from " << name << std::endl;
}

void detached() {
  std::thread t1 = std::thread(sayHello, "detached");
  t1.detach();
  std::cout << "Detached will still say Hello, even after the program exits this
    function." << std::endl;
}

void join() {
  std::thread t1 = std::thread(sayHello, "joined");
  t1.join();
  std::cout << "This line will appear after Hello, as join() is blocking." << std::
    endl;
}

void nothing() {
  std::thread t1 = std::thread(sayHello, "detached");
  std::cout << "Without joining or detaching, this function will exit and throw an
    exception as the thread is killed" << std::endl;
}


int main() {
  detached();
  std::this_thread::sleep_for((std::chrono::milliseconds(10000)));
  std::cout << "------------------------" << std::endl;

  join();
  std::this_thread::sleep_for((std::chrono::milliseconds(10000)));
  std::cout << "------------------------" << std::endl;

  nothing();
  std::this_thread::sleep_for((std::chrono::milliseconds(10000)));
  std::cout << "------------------------" << std::endl;
}
```

3.3.2 Rust

Rust allows for OS-threads to be spawned by the programmer. Methods for managing threads are included in the `std::thread` namespace. (Rust Team 2020h)

Similarly to tasks, the `join()`-method returns the last returned value of the executed code packaged into a `std::result::Result`. If the thread panicked, `Err` will instead be returned with the value set to the value of the `panic!`. The only exception is the main thread that, if an unhandled panic occurs, will cause the application to terminate with an error code. (Rust Team 2020h)

As for additional features, Rust allows for threads to be named and to specify their stack sizes. Naming threads will, for example, display the name of the thread when panicking and will also set it as the OS-thread name where applicable. (Rust Team 2020h)

The default stack size for spawned threads is *2MB* but this can be overwritten either by providing a `Builder::stack_size` to the thread *Builder* or by setting the environment variable `RUST_MIN_STACK` to a different value. When both of them are set

the `Builder::stack_size` takes priority. The builder itself can be invoked by calling `thread::Builder::new()`. (Rust Team 2020h)

The spawned thread can also be assigned to a variable as a `std::thread::JoinHandle` on which one can `join`. (Rust Team 2020h)

Example

Listing 3.10: thread.rs

```
fn main() {
    use std::thread;

    let thread = thread::spawn(move || {
        println!("Hello, I'm a new thread!");
    });
    let res = thread.join();
    match res {
        Ok(_) => println!("Thread exited successfully!"),
        Err(_) => println!("Thread exited unsuccessfully!"),
    }

    let thread_builder = thread::Builder::new()
        .name("BuilderThread".to_string())
        .stack_size(4000)
        .spawn(move || {
            println!("I'm the builder thread! And I will fail!");
            panic!("What's that over there?!?");
        });
    let res_builder = thread_builder.unwrap().join();
    match res_builder {
        Ok(_) => println!("Thread exited successfully!"),
        Err(_) => println!("Thread exited unsuccessfully!"),
    }
}
```

3.3.3 Go

Go does not offer threads to OS-threads mapping. Everything is handled by the go runtime, thus one cannot spawn a OS-thread in Go but must instead use a *goroutine*, see Subsection 3.2.3.

3.3.4 Julia

Julia currently does not offer access to native OS-threads instead relying on *Tasks*, see Subsection 3.2.4.

3.3.5 Comparison

Spawn 10.000 threads and calculate the square root of 1337. When finished spawning all threads join all threads before exiting.

Language	Execution Time [S]	SD Time	Memory Usage [MB]	SD Memory
C++	0.234	0.012	86,771	95
Rust	0.245	0.005	87,888	48

Table 3.5: Thread execution metrics.

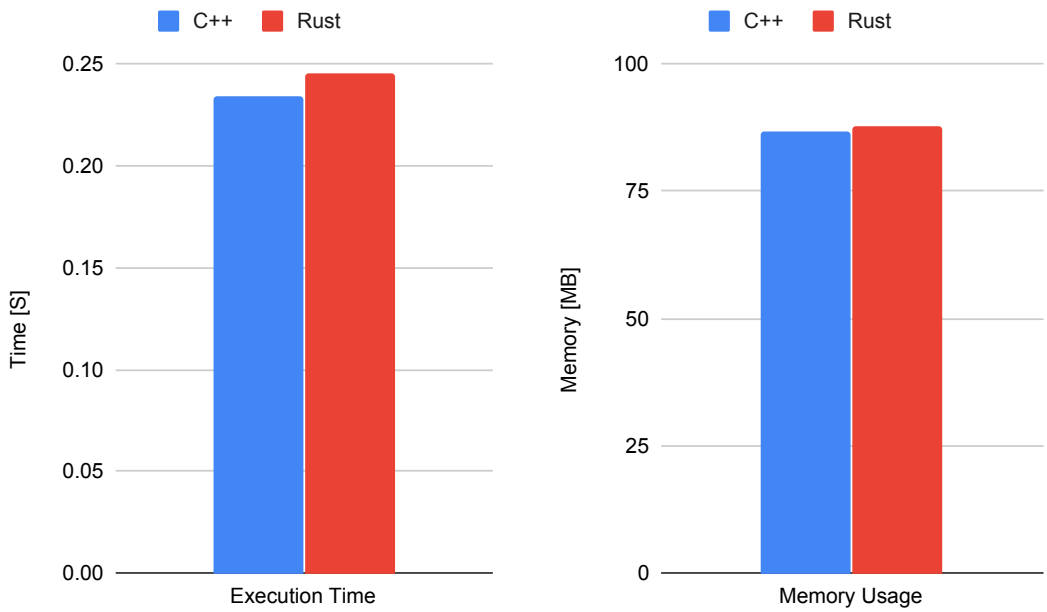

Figure 3.5: Thread execution graphs.

Language	Compile Time [S]	SD Time	Memory Usage [MB]	SD Memory
C++	0.586	0.015	82,041	58
Rust	0.278	0.014	127,641	928

Table 3.6: Thread compilation metrics.

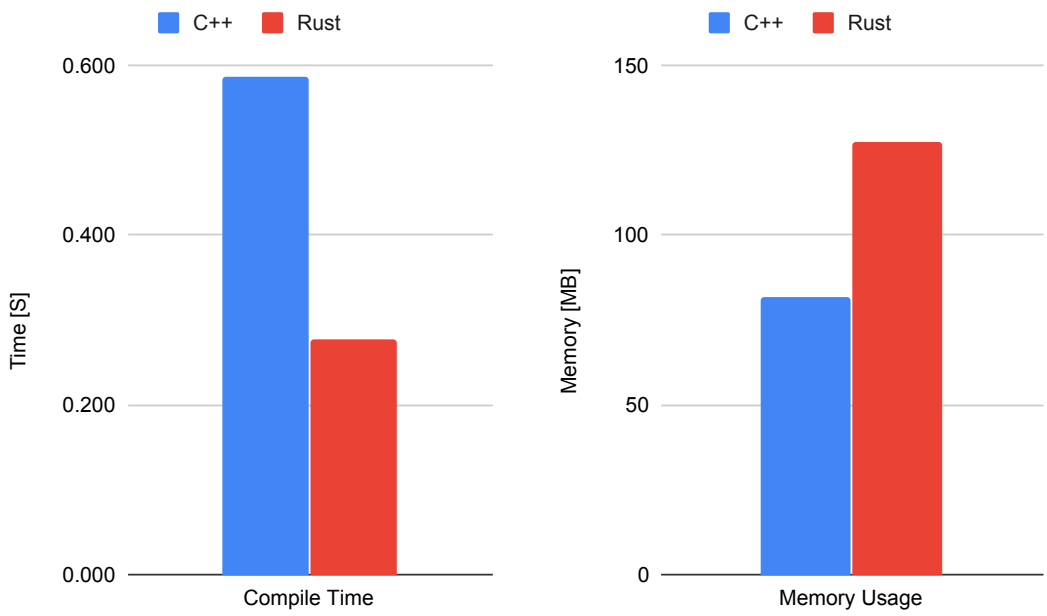

Figure 3.6: Thread compilation graphs.

Execution & Compilation

When comparing the execution time of both programs they are nearly identical with both being around a quarter of a second (see fig. 3.5 and tab. 3.5). Even the memory usage is the same just short of 90MB used. This may be because threads are managed by the OS and thus most of the heavy lifting tasks like spawning the thread and cleaning it up are done by the language-independent OS.

But when comparing threads to tasks there is a big difference between the languages. While Rust gains a lot of speed with tasks by reducing its execution time from 0.245 seconds down to 0.021 seconds, C++ takes even longer with tasks, 0.249 seconds as opposed to pure threads with 0.234 seconds. Memory usage stays the same between the C++ tasks and threads, while in Rust threads take up 90MB of RAM and tasks only 4MB.

Compile times differ between Rust and C++ with C++ taking twice as long as Rust to compile with a compilation time of 0.586 seconds compared to Rusts 0.278 seconds on average (see fig. 3.6 and tab. 3.6). In memory usage, Rust uses over 120MB of memory while C++ uses slightly more than 80MB, even less than the compiled program itself.

Ease of use

Compared to C++, Rust offers additional configuration options for threads via the `thread::Builder` where it is possible to set the stack size and name of the thread while threads in C++ must be differentiated in another way, for example via `std::thread::-get_id`, although this id cannot be customised. (cppreference.com 2020f) Also in Rust threads are automatically detached when spawned while in C++ they must be explicitly

detached from the current thread.

Rust also offers the possibility to check the result of the thread if it exited successfully or if the thread exited with a panic. In C++ exceptions in a thread will call `terminate()` which in its default configuration will call `abort()`, exiting the application unsuccessfully. Therefore exceptions and problems in C++ must be handled inside the thread itself.

Chapter 4

Message Passing

4.1 Channels

One way to pass messages between different threads or processes are channels. Instead of relying on shared memory, channels offer *send*- and *receive*-methods where data can either be copied or moved into them, thus avoiding data races on messages as only one thread can own the message. Channels can either be buffered where the *send*-method only blocks when the buffer is full or they can be atomic where the send-method blocks until a receiver accepts the message. In general, channels work like first-in-first-out queues where the first written value will be the first read value. (Klabnik and Nichols 2018)

4.1.1 C++

In C++ there is no standard library implementation for channels although implementations from other, third-party libraries are available.

4.1.2 Rust

In Rust there is no single channel type, instead, the channel is immediately split at initialization into `Sender` and `Receiver` through multiple return values. However there are two `Sender` types returned by two different functions, `Sender` by `std::sync::mpsc::channel` and `SyncSender` by `std::sync::mpsc::sync_channel`. `Sender` acts as an infinite buffer, thus `send()` will never block the thread. With `std::sync::mpsc::sync_channel`, a buffer can be defined (called `bound`) after which the `send()` function will block. Setting the buffer size to 0 makes the channel act like an unbuffered channel similar to *Go* where values are handed over to the receiver thread atomically. To prevent blocking the sender thread, `SyncSender` has an additional method, `try_send()` which will not block when the requirements for an unblocking `send` are not given, instead, it will return immediately with the failure case (either full buffer or disconnection). Both senders cannot be copied but can rather be cloned, making it possible for multiple threads to send messages. Senders cannot be closed explicitly and instead must be dropped or go out of scope. (Rust Team 2020e)

In both implementations the Receiver acts the same way, blocking until there is a

value in the channel or the channel is closed when using the `recv()` function. Rust also provides a `try_recv` function to poll values without risking to block the channel. Additionally there is a `recv_timeout` function which will act similarly to the receive function but with a timeout which will then return an `Err` instead of blocking forever. (Rust Team 2020e)

It is possible in Rust to iterate over channels making it possible to use channels in for-each loops. Each loop will block until a value is available in the channel. When the channel is closed the loop will exit. (Rust Team 2020e)

In contrast to the `Sender`, `Receivers` cannot be cloned thus only one thread can receive messages from a channel. (Rust Team 2020e)

Rust channels are also strongly typed, thus it is not possible to send untyped values through it. (Rust Team 2020e)

Listing 4.1: message.rs

```rust
use futures::executor::{block_on, ThreadPool};
use futures::task::SpawnExt;

fn channel_reader(std::sync::mpsc::Receiver channel) {
    for elem in channel.iter() {
        println!("Received {}", elem);
    }
    println!("Channel closed, exiting...");
}

fn channel_writer(std::sync::mpsc::SyncSender channel) {
    channel.send(5);
    println!("Wrote to channel: 5");
    channel.send(7);
    println!("Wrote to channel: 7");
    channel.send(12);
    println!("Wrote to channel: 12");
    drop(channel)
}

fn main() {
    let pool = ThreadPool::new().unwrap();

    let (buffered_sender, buffered_receiver) = std::sync::mpsc::sync_channel(10);
    buffered_sender.send(11);
    buffered_sender.send(0);
    buffered_sender.send(12);
    println!("{} {} {}", buffered_receiver.recv(), buffered_receiver.recv(),
    buffered_receiver.recv());

    let (unbuffered_sender, unbuffered_receiver) = std::sync::mpsc::sync_channel(0);
    pool.spawn_with_handle(async { channel_reader(unbuffered_receiver)});
    channel_writer(unbuffered_sender);

}
```

4.1.3 Go

In Go, channels are represented by the `ChannelType`. They can be created with the `make` command and either be buffered or unbuffered. Channels need to have a type but when using `interface{}` it is possible to push different data types into the channel. When constructing the channel without a buffer count or a buffer count equal zero the buffer in Go acts as an unbuffered buffer thus blocking on any write until another thread polls the message. When working with buffered channels the write method will only block after the buffer is full. The read method will block as soon as there are no more values in the channel until either another value is written into the channel or the channel is closed. (Google 2019)

Go also allows for receive-only and send-only channels, these can be created either by assignment or explicit conversion. In Go, there can be multiple senders and receivers acting on one channel. (Google 2019)

As channels can be `nil` and receiving on a `nil` channel blocks forever, one has to check channels for `nil` before receiving on them. Receiving on a closed channel yields the types zero value if the buffer is already empty. One can check if the zero value was sent by a sender or is the result of a closed channel with multiple return, with the first return value being the value and the second return value being a boolean indicating the open state of the channel. (Google 2019)

A feature that no other compared language offers is the `select` statement which acts similarly to a `switch` statement only with channels. Multiple channels can be waited on in the select statement and the channel with a message waiting will be selected. If there are multiple channels with messages one will be chosen at random. The select statement will block if there are no messages in any channel unless there is a `default` case. In that situation, the default case will be executed. As closing a channel in Go results in the default value being returned when querying the channel, `default` cannot be used to check if multiple channels are closed as the select statement will just randomly choose one of the closed channels. (Google 2019)

Listing 4.2: message.go

```
1 package main
2
3 import "fmt"
4
5 func channelReader(channel <-chan int) { // this channel is receive only
6     for elem := range channel { // will block until channel is written into
7         fmt.Println("Received", elem) // will print 5, 7, 10
8     }
9     fmt.Println("Channel closed, exiting...") // will print after channel is closed
10 }d
11
12 func channelWriter(channel chan<- int) { // this channel is send only
13     channel <- 5
14     fmt.Println("Wrote to channel: 5");
15     channel <- 7 // this assignment will block until the value is fetched by
       channelReader
16     fmt.Println("Wrote to channel: 7");
17     channel <- 12
18     fmt.Println("Wrote to channel: 12");
```

```
    close(channel)
}

func multiValDemo(channel <-chan int) {
    for {
        value, open := <- channel
        fmt.Println(value)
        if(!open) {
            fmt.Println("open is true, so value was valid")
            break
        } else {
            fmt.Println("open is false, so value was default value")
        }
    }
}

func fillChannels(a, b chan int) {
    for i := 0; i < 10; i++ {
      a <- i
      b <- i
    }
    close(a)
    close(b)
  }

func selectDemo(a, b chan int) {
  var cur int
  var openA bool
  var openB bool
  for {
    select {
    case cur, openA = <- a:
      if !openA && !openB { // if both channels are closed, return
                return
            }
      if openA {
        fmt.Println("a", cur) // print current number only if channel is open
      }
    case cur, openB = <- b:
            if !openA && !openB { // if both channels are closed, return
                return
            }
      if openB {
        fmt.Println("b", cur)
      }
  }
}

func main() {

  buffered := make(chan interface{}, 10) //buffered, untyped channel

  buffered <- "Test"
  buffered <- 0
  buffered <- 12
  x, y, z := <-buffered, <-buffered, <-buffered
  fmt.Println(x, y, z) // prints "Test 0 12"
```

```
    close(buffered)

    unbuffered := make(chan int, 0)          //unbuffered, typed channel
    go channelReader(unbuffered);
    channelWriter(unbuffered);

    multiAssign := make(chan int, 0)
    go channelWriter(multiAssign)
    multiValDemo(multiAssign)

    a := make(chan int, 11)
  b := make(chan int, 11)
  fillChannels(a, b)
  selectDemo(a, b)
}
```

4.1.4 Julia

Julia also offers buffered and unbuffered channels. They can either be typed or untyped and all variants of channels are created by the same command `Channel`. Thanks to Julia's *Inf* number, it is possible to create an infinite buffered channel. Channels constructed with $size = 0$ are unbuffered and will work the same way as in the other languages atomically handing over the value to the receiver thread. In Julia, the channel itself can also be `fetch`ed where the value is only returned but not removed from the channel itself. This only works on buffered channels. (Julia 2020d)

As in other languages, Julia can iterate over channels until they are closed. Closing can also happen explicitly. (Julia 2020d)

In Julia, tasks and channels are designed to closely work with each other and thus there are functions to explicitly assign tasks to channels and vice versa. The channel constructor allows for passing a function that will execute at the moment of instantiation of the channel. This function will be given the channel as an argument. As the function will only be called once upon creation of the channel itself, and the task should work with incoming data, a loop has to be implemented inside the task which waits on the channel. The task itself will exit as soon as the channel is closed. (Julia 2020d)

Another possibility in Julia is binding channels to tasks, which acts the other way round. When a channel is bound to a task it will automatically be closed as soon as the task terminates. When a channel is bound to more than one task, the first task that terminates will close the channel. (Julia 2020d)

`Channels` themselves are process-local data structures in Julia and will not work across worker boundaries. Julia provides a wrapper `RemoteChannel` which allows different workers to manipulate the same channel. (Julia 2020a)

It is however not possible to create explicit *senders* or *receivers* in Julia. (Julia 2020d)

Listing 4.3: message.jl

```
channelReader(channel) =
begin
    for elem in channel
        println(elem)
    end
```

```
    println("Channel closed, exiting...")
end

channelWriter(channel) =
begin
    put!(channel, 5)
    println("Wrote to channel: 5")
    put!(channel, 7)
    println("Wrote to channel: 7")
    put!(channel, 12)
    println("Wrote to channel: 12")
    close(channel)
end

bindTester(channel) =
begin
    take!(channel) # Blocks until something is written
    println("Got my value, exiting...")
end

buffered = Channel(10)
put!(buffered, "Test")
put!(buffered, 0)
put!(buffered, 12)
x, y, z = take!(buffered), take!(buffered), take!(buffered)
println(x, y, z)
close(buffered)

unbuffered = RemoteChannel(() ->Channel{Int}(channelReader, 0, spawn=true))
# Same as
# unbuffered = RemoteChannel(() ->Channel{Int}(0))
# Threads.@spawn channelReader(unbuffered)
channelWriter(unbuffered)

fetchTest = Channel(10)
put!(fetchTest, 10)
put!(fetchTest, 20)
println(fetch(fetchTest)) # will print 10 but won't remove the item from the channel
println(take!(fetchTest)) # also prints 10 but removes the item from the channel
println(take!(fetchTest)) # will print 20 as 10 was removed

bindTest = Channel(10)
bindTask = Threads.@spawn bindTester(bindTest)
bind(bindTest, bindTask) # Channel will now autoclose when bindTask finishes
println(isopen(bindTest)) # Channel open
put!(bindTest, 5) # writing a value
println(isopen(bindTest)) # Task exited --> Channel closed automatically
```

4.1.5 Comparison

Create an unbuffered channel and send 10.000 messages through it. A task running asynchronously will read the send values and calculate the mean send - read time. In the end, the average send - read time should be displayed.

Language	Send-Receive Time [ns]	SD Time	Memory Usage [MB]	SD Memory
Rust	13,202.8	579.84	2,213	77
Go	506.7	47.38	2,128	73
Julia	4,561.7	293.93	3	0.18

Table 4.1: Messages execution metrics.

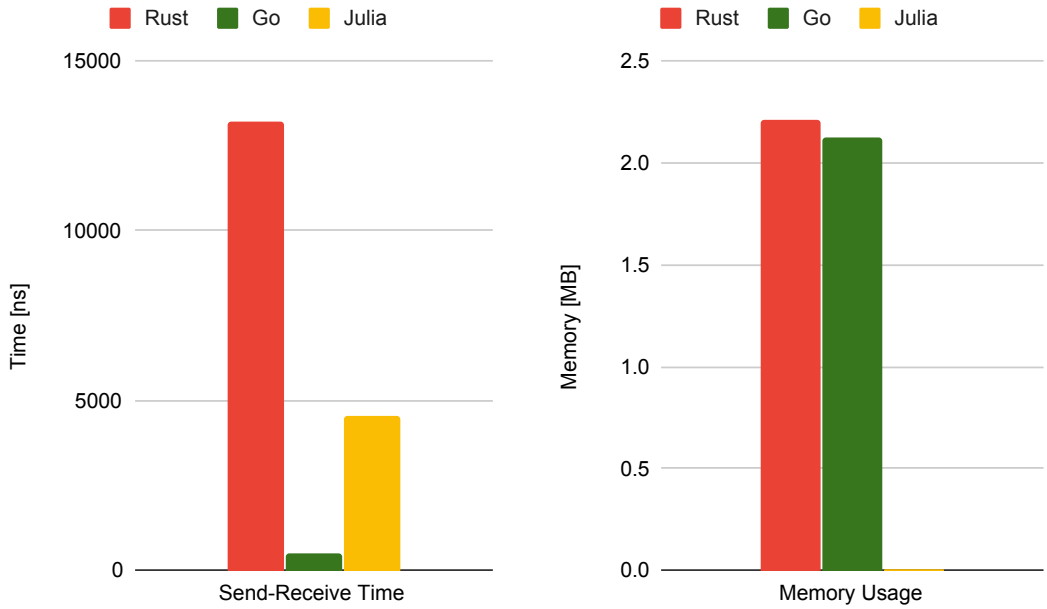

Figure 4.1: Messages execution graphs.

Language	Compile Time [S]	SD Time	Memory Usage [MB]	SD Memory
Rust	0.451	0.004	147,794	147
Go	0.097	0.052	17,754	7,175
Julia	0.229	0.01	153,384	119

Table 4.2: Messages compilation metrics.

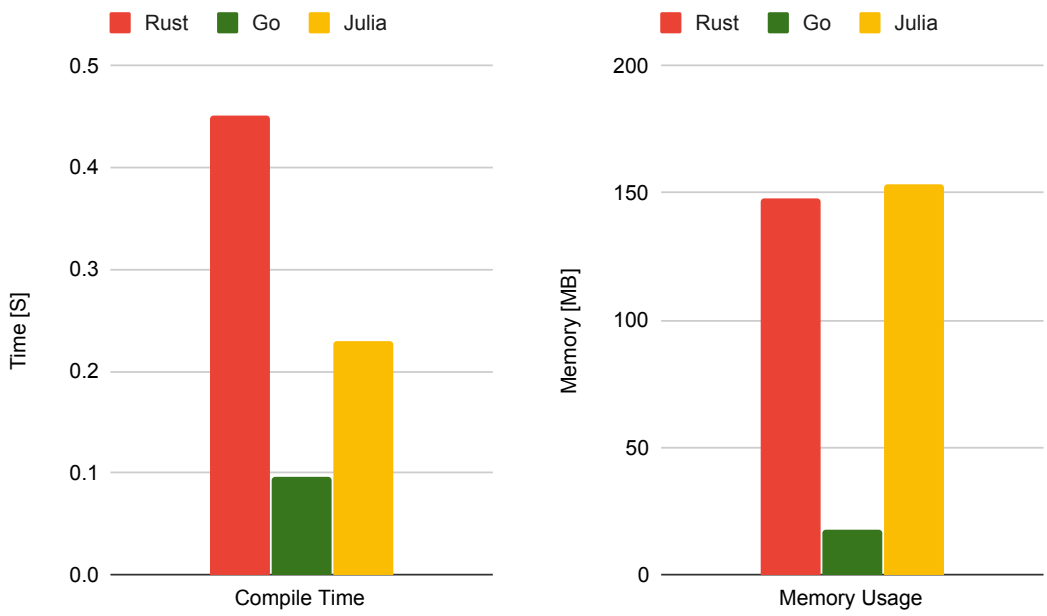

Figure 4.2: Messages compilation graphs.

Execution & Compilation

When comparing the languages based on execution time (see fig. 4.1 and tab. 4.1), Go handles unbuffered channels the fastest taking just 500 nanoseconds on average to fetch the sent value. Next is Julia taking 5 times as long as Go with an average of 4.5 microseconds on average to atomically hand over values. Coming in last is Rust, taking 13 microseconds on average. Memory wise Go and Rust have the same maximum with 2MB of usage while Julia uses just 3kB in its runtime.

When comparing based on compilation (see fig. 4.2 and tab. 4.2), Go is also done the fastest with 97 milliseconds until compilation finished. Julia follows closely behind taking twice that with 200 milliseconds. Rust takes the longest since an external library is required for spawning the tasks handling the receiving of messages, leading to a compile-time of 0.451 seconds. Rust and Julia both have a memory footprint of around 150MB with Rust using close to 148MB and Julia using 153MB. Go has the smallest memory usage of just 18MB.

Ease of use

Go and Julia offer the most features for channels with Go offering multiple return to check the state of the channel and its `select`-statement for selecting a ready value from multiple channels. Go is also the only language to explicitly allow the creation of send-only and receive-only channels while also offering simple channels with both features.

Julia on the other hand tightly knits channels and tasks together with the possibility to close channels automatically as soon as the task finishes or to spawn tasks in the

channel constructor which simplifies tasks acting solely on the channel.

Rust itself offers the same basic functionality as the other languages but there are no additional features compared with the other languages. Rust and Julia take two different approaches in splitting up senders and receivers. While Julia does not allow for explicit senders and receivers, Rust only has the option with the sender and receiver instantly being assigned two different variables through multi-assignment.

Chapter 5

Memory Safety

5.1 Locks

Locks in programming languages inhibit that multiple, concurrent threads can access and alter the same object at the same time. Variables that may be accessed this way can be wrapped in a *mutex* lock, *mutex* standing for *mutual exclusion*. Most commonly, these locks feature a `unlock` and `lock` method. A possible variation is a read-write lock which allows multiple threads to access the variable as read-only while blocking write access to it. (Galowicz 2017)

5.1.1 C++

C++ offers two different mutexes and these can again be split up into specialisations. The first mutex is simply called `mutex` which just offers the methods `lock` and `unlock` and a non-blocking method `try_lock`. Flavours of this lock are

- `timed_mutex` which has the additional methods `try_lock_for` and `try_lock_until` which will timeout after a certain time or at the given timestamp.
- `recursive_mutex` which allows calling `lock` multiple times on the same mutex without blocking, given that the mutex is already locked by that thread. The thread must then `unlock` that mutex as often as it has locked it for it to be unlocked again.
- `recursive_timed_mutex` simply provides both features of `recursive_mutex` and `timed_mutex`.

(Galowicz 2017)

The other flavour is `shared_mutex` which offers the traditional `lock`, `try_lock` and `unlock` methods, but `lock_shared`, `try_lock_shared` and `unlock_shared` as well. These methods lock the mutex in *shared* mode. Multiple threads can hold that shared lock as they will only read the locked variables without changing them. Trying to exclusive lock the mutex will block. Only after all threads with shared locks have unlocked one will be able to lock the mutex in exclusive mode. (Galowicz 2017)

The only other flavour of the `shared_mutex` is `shared_timed_mutex` which adds the time-out methods to the shared mutex (Galowicz 2017) .

C++ also offers wrappers for these mutexes to make unlocking easier as these helpers

will automatically unlock the given mutex without intervention from the programmer. They also help with the coordination of locking multiple mutexes to avoid deadlocking the program by applying a deadlock avoidance strategy. (Galowicz 2017) These wrappers are called *locks* and following locks are provided by C++

- `lock_guard` acts the same as `mutex` only with an auto unlocking feature when going out of scope or being destroyed.
- `scoped_lock` same as `lock_guard` only that it takes 0-x mutexes and locks them all. This helper can be used if multiple mutexes have to be held for the code to work.
- `unique_lock` locks the given mutex in exclusive mode. Also allows for timeouts in the constructor. It is also possible to instruct the helper to not lock the mutex upon construction, only try locking the mutex or to assume the mutex is already locked.
- `shared_lock` works the same as `unique_lock` only in shared mode.

(Galowicz 2017)

Listing 5.1: lock.cpp

```
#include <thread>
#include <mutex>
#include <shared_mutex>
#include <iostream>

void simulateWork() {
    std::cout << "Doing work..." << std::endl;
    std::this_thread::sleep_for(std::chrono::milliseconds(500));
    std::cout << "Work done!" << std::endl;
}

void mutexDemo(std::mutex *mutex) {
    mutex->lock();
    simulateWork(); //simulate work
    mutex->unlock(); // lock must be manually unlocked
}

void lockGuardDemo(std::mutex *mutex) {
    std::cout << "Trying to acquire lock" << std::endl;
    std::lock_guard<std::mutex> lock(*mutex); // blocks until lock is gained
    std::cout << "Lock acquired!" << std::endl;
    simulateWork();
    std::cout << "Mutex will automatically unlock when exiting" << std::endl;
}

void recursiveMutexDemo(std::recursive_mutex *mutex) {
    int maxLocked = 10;
    int timesLocked = 0;
    while(timesLocked < maxLocked) {
        mutex->lock(); // will not block
        timesLocked++;
        std::cout << "Lock already locked for " << timesLocked << " times." << std::
    endl;
    }
    while(timesLocked > 0) {
```

```
        mutex->unlock();
        timesLocked--;
        std::cout << "Lock now locked " << timesLocked << " times." << std::endl;
    }
    std::cout << "Lock is now unlocked again" << std::endl;
}

void timedMutexDemo(std::timed_mutex *mutex) {
    mutex->lock();
    std::cout << "Trying to lock again..." << std::endl;
    if(mutex->try_lock_for(std::chrono::milliseconds(500))) { // this line would be a deadlock for conventional mutexes.
        std::cout << "Gained mutex, this will never be reached!" << std::endl;
    } else {
        std::cout << "Failed to gain mutex, timed out" << std::endl;
    }
    mutex->unlock();
}

void sharedMutexDemo(std::shared_mutex *mutex) {
    std::cout << "Trying to read lock..." << std::endl;
    mutex->lock_shared();
    std::cout << "Reading value..." << std::endl;
    std::this_thread::sleep_for(std::chrono::milliseconds(500));
    std::cout << "Finished reading." << std::endl;
    mutex->unlock_shared(); // lock must be manually unlocked
}

void sharedMutexWriteDemo(std::shared_mutex *mutex) {
    std::cout << "Trying to write lock..." << std::endl;
    std::unique_lock<std::shared_mutex> lock(*mutex);
    std::cout << "Lock happened after all read locks are unlocked." << std::endl;
    simulateWork();
}

int main() {
    {
        std::mutex mutex;
        std::thread mutexDemoThread(mutexDemo, &mutex);
        std::thread lockGuardDemoThread(lockGuardDemo, &mutex);

        mutexDemoThread.join();
        lockGuardDemoThread.join();
    }
    {
        std::recursive_mutex mutex;
        recursiveMutexDemo(&mutex);
    }
    {
        std::timed_mutex mutex;
        timedMutexDemo(&mutex);
    }
    {
        std::shared_mutex mutex;
        std::thread sharedMutexDemoThread1(sharedMutexDemo, &mutex);
        std::thread sharedMutexDemoThread2(sharedMutexDemo, &mutex);
        std::thread sharedMutexDemoThread3(sharedMutexWriteDemo, &mutex);
```

```
        std::thread sharedMutexDemoThread4(sharedMutexDemo, &mutex);

        sharedMutexDemoThread1.join();
        sharedMutexDemoThread2.join();
        sharedMutexDemoThread3.join();
        sharedMutexDemoThread4.join();
    }

    return 0;
}
```

5.1.2 Rust

Rust features the same two types of locks, a simple `Mutex` and a read-write lock `RWLock`. However, in Rust mutexes are closely bound to the variable they protect. At the instantiation of a lock, the protected value is passed as a parameter. The value can only be changed when owning the lock. When accessing the mutex via `lock` the thread will block until the mutex is unlocked. Then a `LockResult` is returned which can be `matched` to `Ok` and `Err`. In case everything is all right the `Ok` branch will return a `MutexGuard` that can be dereferenced and its value changed. In case of `Err`, a `PoisonError` will be returned. This case will happen when the mutex was previously locked but not unlocked properly, for example when an exception happens while the lock is being held. The value in the mutex may, therefore, be invalid or *poisoned*. It is still possible to access the variable by invoking `into_inner` on the `PoisonError` but it is not possible to *unpoison* the mutex. Where C++ uses helpers to determine when to unlock a lock, Rust directly implements these features into the mutex lock. `MutexGuard` will automatically unlock the mutex as soon as it goes out of scope. There is no `unlock` method however. (Rust Team 2020f)

`RWLock` allows for multiple readers or a single writer. `read` calls will only block when the mutex is currently locked in write mode. Currently waiting for write locks on a mutex will not block subsequent read locks, as for the write lock, it will be blocked until all read locks are unlocked. (Rust Team 2020g)

Listing 5.2: lock.rs

```
#![allow(unused_variables)]
#![allow(non_snake_case)]
#![allow(unused_must_use)]

use std::sync::{Arc, Mutex, RwLock};
use std::time::Duration;
use std::thread;

fn simulateWork() {
    println!("Doing work...");
    std::thread::sleep(Duration::from_millis(500));
    println!("Work done!");
}

fn mutexDemo(mutex: Arc<Mutex<i32>>) {
  mutex.lock().unwrap(); // blocks until lock is gained
  simulateWork(); // simulate work
  println!("Mutex will automatically unlock when exiting");
```

```
}

fn poisonDemo(mutex: Arc<Mutex<i32>>) {
  let mutexPanic = mutex.clone();
  let threadPanic = thread::spawn(move || -> () {
    let value = mutexPanic.lock().unwrap();
    panic!(); // thread panics and leaves mutex in poisioned state
  }).join();

  let value = match mutex.lock() {
    Ok(value) => println!("Unpoisoned mutex: {}", *value), // will not happen
    Err(poisoned) => println!("Poisoned mutex: {}", *poisoned.into_inner()), //
     mutex is now poisioned, value is available through method
  };
}

fn sharedMutexDemo(mutex: Arc<RwLock<i32>>) {
  println!("Trying to read lock...");
  let read = mutex.read().unwrap();
  println!("Reading value...");
  std::thread::sleep(std::time::Duration::from_millis(500));
  println!("Finished reading.");
}

fn sharedMutexWriteDemo(mutex: Arc<RwLock<i32>>) {
  println!("Trying to write lock...");
  let mut write = mutex.write().unwrap();
  *write += 1;
  println!("Lock happened after all read locks are unlocked.");
  simulateWork();
}

fn main() {
    {
      let mutex = Arc::new(Mutex::new(0));
      mutexDemo(mutex);
    }
    {
      let mutex = Arc::new(Mutex::new(0));
      poisonDemo(mutex);
    }
    {
      let mutexOrig = Arc::new(RwLock::new(0));
      let mutex1 = mutexOrig.clone();
      let mutex2 = mutexOrig.clone();
      let mutex3 = mutexOrig.clone();
      let mutex4 = mutexOrig.clone();

      let sharedMutexDemoThread1 = thread::spawn(|| {sharedMutexDemo(mutex1)});
      let sharedMutexDemoThread2 = thread::spawn(|| {sharedMutexDemo(mutex2)});
      let sharedMutexDemoThread3 = thread::spawn(|| {sharedMutexWriteDemo(mutex3)});
      let sharedMutexDemoThread4 = thread::spawn(|| {sharedMutexDemo(mutex4)});

      sharedMutexDemoThread1.join();
      sharedMutexDemoThread2.join();
      sharedMutexDemoThread3.join();
      sharedMutexDemoThread4.join();
```

```
    }
}
```

5.1.3 Go

Go also offers the same two kinds of locks, called `Mutex` and `RWMutex`. `Mutex` only offers the most basic methods `Lock` and `Unlock`. Mutex locks are also not bound to a single goroutine so it is possible to lock the mutex in one goroutine and unlock it in a different one. (Google 2020b)

`RWMutex` adds the methods `RLock` and `RUnlock` to the already existing `Mutex` methods. These act the same as C++'s `shared_mutex` allowing multiple read locks on the same mutex from different goroutines. The `RLock` method, however, is not recursive; as soon as one goroutine executes a write lock on it, all following read locks will also block until all initial read locks are unlocked. (Google 2020b)

Go does not provide auto unlocking like C++ does but this functionality can be mimicked by Go's `defer` statement which will execute the following statement as soon as the scope closes. For example `defer Mutex.Unlock()` will unlock the mutex when the statement goes out of scope. (Google 2019)

Listing 5.3: lock.go

```
package main

import "fmt"
import "sync"
import "time"

var waitGroup sync.WaitGroup
func simulateWork() {
    fmt.Println("Doing work..")
    time.Sleep(500 * time.Millisecond)
    fmt.Println("Work done!")
}

func mutexDemo(mutex* sync.Mutex) {
    fmt.Println("Trying to acquire lock")
    mutex.Lock()
    fmt.Println("Lock acquired!")
    simulateWork()
    mutex.Unlock()
    waitGroup.Done()
}

func mutexDeferDemo(mutex* sync.Mutex) {
    fmt.Println("Trying to acquire lock for defer")
    mutex.Lock()
    defer mutex.Unlock() // this method will be called at the exit
    fmt.Println("Lock acquired!")
    simulateWork()
    waitGroup.Done()
}

func sharedMutexDemo(mutex* sync.RWMutex) {
    fmt.Println("Trying to read lock..")
```

```
34     mutex.RLock() // this read will block if it happens after the write lock request
35     fmt.Println("Reading value..")
36     time.Sleep(500 * time.Millisecond)
37     fmt.Println("Finished reading.")
38     mutex.RUnlock()
39     waitGroup.Done()
40 }
41
42 func sharedMutexWriteDemo(mutex* sync.RWMutex) {
43     fmt.Println("Trying to write lock..")
44     mutex.Lock()
45     fmt.Println("Lock happened after all read locks are unlocked.")
46     mutex.Unlock()
47     waitGroup.Done()
48 }
49
50 func main() {
51     {
52         var mutex sync.Mutex
53         waitGroup.Add(2)
54         go mutexDemo(&mutex)
55         go mutexDeferDemo(&mutex)
56         waitGroup.Wait()
57     }
58     {
59         var rwmutex sync.RWMutex
60         waitGroup.Add(4)
61
62         go sharedMutexDemo(&rwmutex)
63         go sharedMutexDemo(&rwmutex)
64         go sharedMutexWriteDemo(&rwmutex)
65         go sharedMutexDemo(&rwmutex)
66         waitGroup.Wait()
67     }
68 }
```

5.1.4 Julia

There are two locks in Julia, `ReentrantLock` and `SpinLock`. `SpinLock` is a low-level primitive lock and is not for general usage (Julia 2020c). Difference between the two locks is that while `SpinLock` performs busy waiting on the lock until it unlocks, `ReentrantLock` will simply block and yield to another task or thread until the lock is unlocked. `ReentrantLock` also supports recursive locking. The lock must then be unlocked as often as it was locked for it to be considered unlocked again. (Julia 2020d)

Both locks support the methods `lock`, `trylock` and `unlock`. `ReentrantLock` additionally supports `lock` und `trylock` with a function as a parameter. When this method gets access to the lock it will execute the function and after the function finishes automatically unlock the lock again. (Julia 2020d)

There is no read-write lock in Julia available although it is possible to implement one while using the interface `AbstractLock` to provide compatibility to the existing locks. (Julia 2020d)

Listing 5.4: lock.jl

```
simulateWork() = begin
    println("Doing work..")
    sleep(0.5)
    println("Work done!")
end

mutexDemo(mutex) = begin
    println("Trying to acquire lock")
    lock(mutex)
    println("Lock acquired!")
    simulateWork()
    unlock(mutex)
end

functionParameterDemo() = begin
    println("Acquired lock, and entered method!")
    simulateWork()
end

mutex = ReentrantLock()
mutexDemoThread = Threads.@spawn mutexDemo(mutex)
mutexDemoThread2 = Threads.@spawn lock(functionParameterDemo, mutex)
fetch(mutexDemoThread)
fetch(mutexDemoThread2)
```

5.1.5 Comparison

Spawn 10 threads. A global counter should be increased by each thread by 1,000,000. Therefore, at the end, the counter should have a value of 10,000,000. For synchronisation, a lock should be used.

Language	Execution Time [S]	SD Time	Memory Usage [MB]	SD Memory
C++	0.813	0.026	3,740	72
Rust	0.88	0.03	2,214	53
Go	1.341	0.03	1,560	24
Julia	0.46	0	155,201	0

Table 5.1: Lock execution metrics.

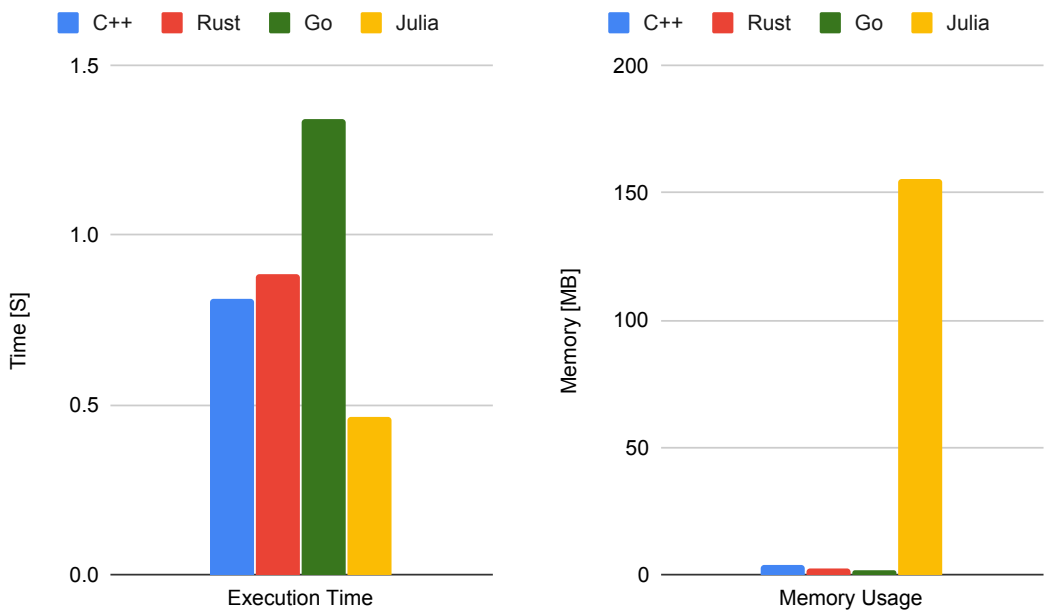

Figure 5.1: Lock execution graphs.

Language	Compile Time [S]	SD Time	Memory Usage [MB]	SD Memory
C++	0.587	0.009	78,350	82
Rust	0.299	0.011	129,037	678
Go	0.086	0.042	17,793	6,780
Julia	0.204	0.01	147,122	665

Table 5.2: Lock compilation metrics.

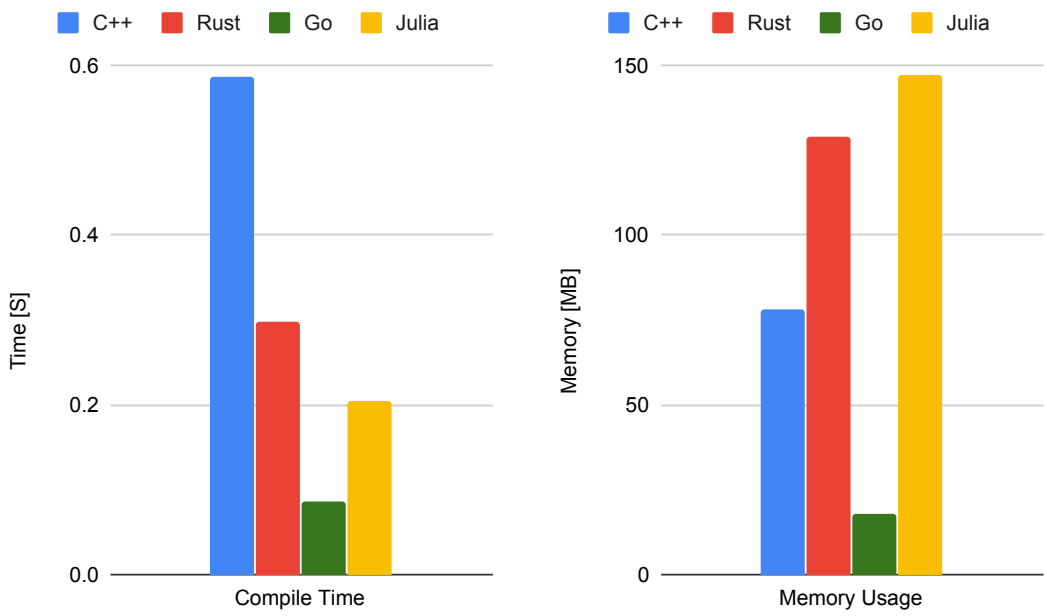

Figure 5.2: Lock compilation graphs.

Execution & Compilation

Julia implements locks the fastest, executing in less than half a second, at the cost of memory usage with over 150MB used, the most of all languages (see fig. 5.1 and tab. 5.1). C++ comes next time wise, executing in 0.8 seconds, but using up 3.7MB. Rust strikes a middle ground with the most balanced memory usage and time consumption, taking still less than a second to execute and using 2.2MB of memory at most. Go, while consuming the least amount of memory, 1.6MB, takes the longest with 1.3 seconds to finish.

In contrast, when compiling, Go is the fastest and most efficient language of all languages compared (see fig. 5.2 and tab. 5.2), compiling within 0.1 seconds, twice as fast as Julia, the second-fastest language. Go also uses by far the least amount of memory, using 18MB while C++ uses 78MB, Rust 130MB and Julia the most with 147MB. C++ spends the most time to compile the program, taking 0.6 seconds.

Ease of use

While all languages support simple locks, Julia is the only language not supporting read-write locks natively. It also does not feature any helper methods for automatic unlocking or deadlock-free access of multiple mutexes, even though a function can be passed to the lock methods which will execute when access to the lock is gained.

Go adds read-write locks to the feature list, but also offers few helper functions. Again, there is no auto unlocking when going out of scope and, differently from all other languages, Go allows unlocking the mutex from other goroutines which, while maybe allowing specific edge cases, may lead to complicated lock-unlock patterns and

deadlocks.

Rust has the tightest connection between locks and the value they protect, as the value is instantiated at the construction of the mutex and is only accessible when unlocking that mutex. Rust also indicates poisoned values very well with its inbuilt match mechanism, where the result of the lock may either be Ok or an Error, immediately notifying the programmer if previous access to the value was interrupted by an exception. Rust also supports out-of-scope unlocking directly in the mutex type and is, therefore, lacking an unlock command.

C++ has the biggest variety of mutexes and mutex helper functions. It offers a great degree of freedom, from simple `mutexes` with `lock` and `unlock` methods to guards that will acquire multiple locks while avoiding deadlocks.

5.2 Condition Variables

Condition variables enable threads to signal events or condition changes to one or more different threads. Threads waiting for a condition to be met *block* on a condition variable until another thread fulfils the condition and signals the condition variable. As condition variables are used to communicate condition changes on shared variables, they are used in conjunction with *mutexes*. The thread wishing to use the condition variable must hold the mutex lock before it waits on the condition. When entering the wait, the mutex is automatically unlocked to be used by the notifying thread. It is also automatically re-acquired when waking up. (Galowicz 2017)

Precautions need to be taken to ensure that the condition is really met. Waking up from waiting on a condition variable does not guarantee that the condition is true, as spurious wakeups can happen. (Galowicz 2017)

5.2.1 C++

`std::condition_variable` in C++ provides three `wait` methods to be used by the thread. `wait` works by providing a held `std::unique_lock` as a parameter. To combat spurious wakeups, an additional and optional `Predicate` can be supplied. This predicate must only return `true` when the condition is really met. When returning `false` the `condition_variable` will automatically reenter the `wait` to wait for the condition to be met. Entering a `wait` without owning the lock or if the lock is differing from the lock of other threads waiting on the same `condition_variable` the behaviour is undefined. (cppreference.com 2020c)

Other variations of `wait` are `wait_for` and `wait_until`. Both offer the same basic functionality as `wait` but with timeouts. `wait_for` will wait until the `Predicate` evaluates to `true` or after `std::chrono::duration` to the initial call to the method. Its return values are differing from the conventional `wait` call. It either returns `std::cv_status::timeout` when the wakeup call resulted from the timer expiring or else `std::cv_status::no_timeout` when the call came from a `notify`. When specifying a predicate the method will return the result of the predicate after the timeout has expired. `wait_until` works by using a `std::chrono::time_point` instead of a `std::chrono::duration`. (cppreference.com 2020c)

For notifications, C++ provides `notify_one` and `notify_all`. `notify_one` will only

wake a single thread waiting on the condition variable while `notify_all` will wake all threads. (cppreference.com 2020c)

Listing 5.5: condvar.cpp

```
#include <thread>
#include <mutex>
#include <iostream>
#include <condition_variable>
#include <atomic>

const int AMOUNT_PINGS = 100000;

std::condition_variable condVar;
std::mutex condMutex;
bool continueExec = false;

std::atomic<bool> finished;
int main() {
    auto waitThread = std::thread([](){
        std::unique_lock<std::mutex> lock(condMutex);
        std::cout << "Entering wait..." << std::endl;
        condVar.wait(lock, []() { return continueExec;});
        std::cout << "Was awoken!" << std::endl;
    });

    std::this_thread::sleep_for(std::chrono::milliseconds(500));
    condMutex.lock();
    continueExec = true;
    condMutex.unlock();
    condVar.notify_all();
    waitThread.join();


    continueExec = false;
    auto timeoutPredicateThread = std::thread([](){
        std::unique_lock<std::mutex> lock(condMutex);
        std::cout << "Entering wait..." << std::endl;
        if(condVar.wait_for(lock, std::chrono::milliseconds(500) ,[]() { return continueExec;}) == false) {
            std::cout << "Timed out, condition is still false!" << std::endl;
        } else {
            std::cout << "Was awoken without timeout!" << std::endl;
        }
    });
    timeoutPredicateThread.join();

    continueExec = false;
    auto timeoutThread = std::thread([](){
        std::unique_lock<std::mutex> lock(condMutex);
        std::cout << "Entering wait..." << std::endl;
        if(condVar.wait_for(lock, std::chrono::milliseconds(500)) == std::cv_status::timeout) {
            std::cout << "Timed out!" << std::endl;
        } else { // std::cv_status::no_timeout
            std::cout << "Was awoken without timeout!" << std::endl;
        }
```

```
    });
    timeoutThread.join();
    return 0;
}
```

5.2.2 Rust

`std::sync::Condvar`s in Rust are not overloaded, methods offering a predicate are suffixed with `_while`. `wait` is also given a `Mutex` but while using different mutexes in C++ yields undefined behaviour, Rust will get into a runtime panic. The method also returns the `LockResult` to check if the mutex has been poisoned. (Rust Team 2020d)

`wait_while` offers the same functionality as `wait` but taking an additional condition which must evaulate to `false`. (Rust Team 2020d)

`wait_timeout` works like C++'s `wait_for` where a `Duration` has to be specified after which the wait will timeout and unblock. The returned `LockResult` also contains a `WaitTimeoutResult` to check if the method returned because of the time running out. `wait_timeout_while` is also available. (Rust Team 2020d)

`notify_one` and `notify_all` unblock the waiting threads. (Rust Team 2020d)

Listing 5.6: condvar.rs

```
#![allow(non_snake_case)]
#![allow(unused_must_use)]

fn main() {
use std::sync::{Arc, Mutex, Condvar};
use std::{thread, time};

    let condVarOrig = Arc::new(Condvar::new());
    let condMutexOrig = Arc::new(Mutex::new(false));

    {
        let condVar = Arc::clone(&condVarOrig);
        let condMutex = Arc::clone(&condMutexOrig);
        let waitThread = thread::spawn(move || {
            let continueExec = condMutex.lock().unwrap();
            println!("Entering wait...");
            condVar.wait_while(continueExec, |continueExec|{return !*continueExec;})
    .unwrap();
            println!("Was awoken!");
        });

        thread::sleep(time::Duration::from_millis(500));
        { // setting continueExec after half a second of waiting
            let mut continueExec = condMutexOrig.lock().unwrap();
            *continueExec = true;
            condVarOrig.notify_all();
        }
        waitThread.join().unwrap();
    }
    { // resetting continueExec
        let mut continueExec = condMutexOrig.lock().unwrap();
        *continueExec = false;
    }
```

```
33
34     {
35         let condVar = Arc::clone(&condVarOrig);
36         let condMutex = Arc::clone(&condMutexOrig);
37         let timeoutPredicateThread = thread::spawn(move || {
38             let continueExec = condMutex.lock().unwrap();
39             println!("Entering wait...");
40             let result = condVar.wait_timeout_while(continueExec, time::Duration::
       from_millis(500), |continueExec|{return !*continueExec;}).unwrap();
41             if result.1.timed_out() { // access WaitTimeoutResult
42                 println!("Timed out!");
43                 if *result.0 {
44                     println!("Condition is true! This will not happen.");
45                 } else {
46                     println!("Condition is still false!");
47                 }
48             } else {
49                 println!("Was awoken!");
50             }
51         });
52
53         timeoutPredicateThread.join().unwrap();
54     }
55
56     {
57         let condVar = Arc::clone(&condVarOrig);
58         let condMutex = Arc::clone(&condMutexOrig);
59         let timeoutThread = thread::spawn(move || {
60             let continueExec = condMutex.lock().unwrap();
61             println!("Entering wait...");
62             let result = condVar.wait_timeout(continueExec, time::Duration::
       from_millis(500)).unwrap();
63             if result.1.timed_out() { // access WaitTimeoutResult
64                 println!("Timed out!");
65             } else {
66                 println!("Was awoken!");
67             }
68         });
69
70         timeoutThread.join().unwrap();
71     }
72
73 }
```

5.2.3 Go

Go uses `Cond` to implement condition variables. Differing from the other languages, the mutex is already required at the instantion of `Cond` as it is a parameter of `NewCond`. (Google 2020b)

Go only offers a single wait method called `Wait` with no timeout methods available. Also differing from the other languages is that in Go there are no spurious wakeups and only `Broadcast` and `Signal` will wake `Wait`. But as the condition may still be changed between checking for it and relocking the lock, `Wait` should still be wrapped in a loop checking for the condition. (Google 2020b)

`Signal` is used in Go to wake a single goroutine while `Broadcast` will wake all goroutines waiting on the condition. (Google 2020b)

Listing 5.7: condvar.go

```
package main

import "fmt"
import "sync"
import "time"

func main() {
    var waitGroup sync.WaitGroup
    waitGroup.Add(1)

    var condMutex sync.Mutex
    condVar := sync.NewCond(&condMutex)
    continueExec := false

    go func() {
        condMutex.Lock()
        fmt.Println("Entering wait...")
        for(!continueExec) {
            condVar.Wait()
        }
        condMutex.Unlock()
        fmt.Println("Was awoken!")
        waitGroup.Done()
    }()
    time.Sleep(500 * time.Millisecond)
    condMutex.Lock()
    continueExec = true
    condVar.Broadcast()
    condMutex.Unlock()

    waitGroup.Wait()
}
```

5.2.4 Julia

Condition variables are implemented within `Base.Threads.Condition` in Julia. As with Go, the lock is already embedded in `Condition` on creation as a parameter in the constructor and the only methods available are one `wait` and one `notify`. (Julia 2020d)

`wait` can be used with `Condition` as a parameter. As only the unlocking and entering the wait happens atomically, but the locking and returning from `wait` does not, it is necessary to wrap the `wait` in a loop where the condition is checked again if it still holds valid before continuing. (Julia 2020d)

`notify` is overloaded allowing to choose if one single thread should be notified or all waiting threads should be awoken. Two unique features of Julia are the possibility to specify a *value* which will be passed to all waiting tasks and to specify whether this value is to be raised as an exception. The return value of `notify` is the number of awoken tasks. (Julia 2020d)

Listing 5.8: condvar.jl

```
condMutex = ReentrantLock()
condVar = Threads.Condition(condMutex)
continueExec = false

waitThread = Threads.@spawn () = begin
    lock(condMutex)
    println("Entering wait...")
    while(!continueExec)
        println(wait(condVar)) # prints "Value"
    end
    unlock(condMutex)
    println("Was awoken!")
end

sleep(0.5)
lock(condMutex)
continueExec = true
notify(condVar, "Value", true, false)
unlock(condMutex)
fetch(waitThread)

#reset continueExec
continueExec = false

errorThread = Threads.@spawn () = begin
    lock(condMutex)
    println("Entering wait...")
    try
        while(!continueExec)
            println(wait(condVar)) # will never print something
        end
    catch e
        println("Catched this: ", e)
    end
    unlock(condMutex)
    println("Was awoken!")
end

sleep(0.5)
lock(condMutex)
continueExec = true
notify(condVar, "Catch me!", true, true) # pass "Catch me!" as error
unlock(condMutex)
fetch(errorThread)
```

5.2.5 Comparison

Spawn 2 threads. One thread should increase a counter by 1 whenever it is requested. A second thread waits and requests an increase of the counter until it has hit 100,000. To synchronise the threads, condition variables should be used.

Language	Execution Time [S]	SD Time	Memory Usage [MB]	SD Memory
C++	0.651	0.054	3,520	79
Rust	0.769	0.018	2,059	43
Go	0.098	0.003	1,550	46
Julia	0.22	0.02	6,098	0

Table 5.3: Condition variable execution metrics.

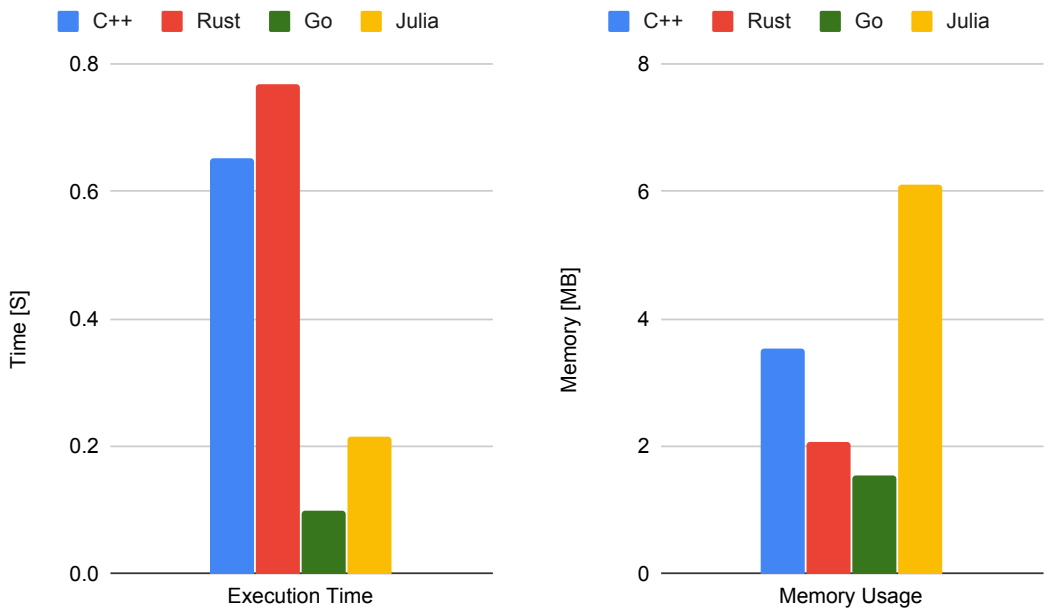

Figure 5.3: Condition variable execution graphs.

Language	Compile Time [S]	SD Time	Memory Usage [MB]	SD Memory
C++	0.577	0.007	80,470	74
Rust	0.308	0.019	128,647	757
Go	0.078	0.008	15,163	218
Julia	0.217	0.01	147,666	144

Table 5.4: Condition variable compilation metrics.

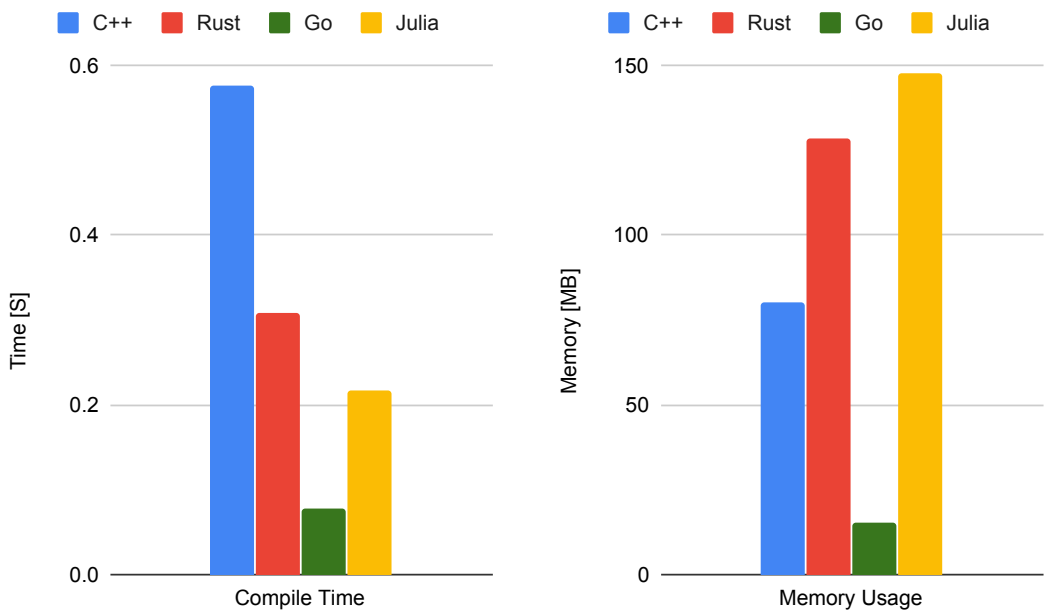

Figure 5.4: Condition variable compilation graphs.

Execution & Compilation

Execution times (see fig. 5.3 and tab. 5.3) are grouped into two groups, Go and Julia, and C++ and Rust. Go executes the fastest in just below 0.1 seconds, followed by Julia with 0.2 seconds. Then there is a big gap and next is C++ with 0.65 seconds, with Rust being the slowest language with 0.77 seconds. Comparing based on memory, Go is again the most efficient language with 1.5MB used, closely followed by Rust with 2 and C++ with 3.5. Julia, even with its runtime, uses 6MB for condition variables.

When compiling (see fig. 5.4 and tab. 5.4), Go too is the fastest and most memory efficient, taking 0.078 seconds and using 15MB while compiling. This means that Go compiles the code faster than it takes to execute the code, same with Rust which takes 0.3 seconds to compile and 0.77 seconds to execute. Julia takes the same for both, 0.2 seconds to both execute and compile. C++ takes the longest to compile at 0.58 seconds until completion. Between the languages, there is a big gap when comparing memory, with Go maxing out at 15MB, and the next language being C++ at 80MB. Rust needs 129MB and Julia including its runtime 148MB to compile the code.

Ease of use

C++ and Rust both offer the same feature set, wherein each `wait` the lock has to be supplied. Both languages also offer condition checking for the wait, C++ as an overloaded parameter and Rust in a separate method by appending `_while` to the original method. Timeouts are also available in both languages, but only Rust takes extra precaution to not have undefined behaviour situations by getting in a runtime panic when using different locks with the same condition or by returning a poisoned

result when necessary.

Go and Julia both require the lock at instantiation of the condition variable, circumventing the problem with differing locks when waiting. They also lack the timeout features the other two languages offer. Extra care has to be taken to not run into a deadlock. There is also no condition checking method present in both languages, so the programmer must use a loop to check if the condition still holds when woken up from a `wait`.

Julia is the only language to not have two different notify methods for signalling one or all waiting threads. Instead, the `notify` function is overloaded with a parameter if one or all threads should be awoken.

5.3 Atomics

As seen in Section 5.1, locks are expensive tools to achieve thread-safety, especially when they are just used to protect increments and decrements of variables. Atomics are data types where behaviour on concurrent read and write is well-defined. These hardware instructions are uninterruptible and therefore need no locking mechanism. These hardware-level instructions are only available on primitive data types. Atomics are also dependent on the CPU with some architectures offering more atomics than others. (Schmidt et al. 2017)

There are a few possible memory orders for atomic operations. Some programming languages allow the programmer to alter the memory order to improve program execution. Supported memory orders of the compared languages are:

- `Relaxed`
- `Acquire`
- `Release`
- `Acquire-Release`
- `Consume`
- `Sequentially Consistent`

(cppreference.com 2020d)

5.3.1 C++

`std::atomics` in C++ are realized via templates with specializations available for certain data types. For the primary template, the used type must only be copy-constructable and copy-assignable. Thus user-defined structs and objects can also be used as atomics. Access to these data types will then be managed with locks. To check if a certain atomic uses mutexes or is lock free the method `is_lock_free` can be used. (cppreference.com 2020b)

There are specializations available for pointer and integral types, adding `fetch_add` and `fetch_sub` for pointer types and `fetch_add`, `fetch_sub`, `fetch_and`, `fetch_or`, `fetch_xor` for integral types. Since C++20, Floating-point types like `float` or `double` also have the methods `fetch_add` and `fetch_sub` added by specialisation. (cppreference.com 2020b)

C++ also offers the type aliases `std::atomic_signed_lock_free` and `std::atomic_unsigned_lock_free` that are guaranteed lock-free and are integral datatypes. `std::atomic_flag` is also guaranteed to be lock-free and can be used as an atomic boolean. It is however not a type alias for an atomic bool as it does not support load and store operations, instead adding the methods `clear` to set the flag to false and `test_and_set` to set the flag to true and return the previous value. (cppreference.com 2020b)

It is possible to change the memory order of certain operations. All memory orders listed above are supported. (cppreference.com 2020b)

There are a few methods that are supported by all atomic types. `store` or a simple `=` assignment atomically store the value in the atomic variable. While it is not possible to change the memory order of the assignment from its default order `std::memory_order_seq_cst`, `store`'s order can be changed to `memory_order_relaxed`, `memory_order_release` or `memory_order_seq_cst`. `load` or simply assigning the value atomically loads the value and returns it. As with `store`, the simple assignement operator uses `memory_order_seq_cst` while `loads` memory order can be set to `memory_order_relaxed`, `memory_order_consume`, `memory_order_acquire`, `memory_order_seq_cst` respectively. (cppreference.com 2020b)

`exchange` assigns the given parameter value to the atomic and returns the old value. The memory order of this method can also be changed. (cppreference.com 2020b)

C++ offers two different compare-and-set methods, `compare_exchange_weak` and `compare_exchange_strong`. `weak` is allowed to fail spuriously but will be faster when used in a loop. Both can be used interchangeably in respect of features. Two values can be passed and if the `expected` parameter matches the current value of the atomic, `desired` will be written into the atomic. The return value of the methods indicates if the value has been set. It is again possible to set a memory order, one can even set the memory order for success and failure case for matching the `expected` parameter. (cppreference.com 2020b)

It is also possible to use atomics as a replacement for `std::condition_variable`s as they also offer the methods `wait`, `notify_one` and `notify_all`. `wait` takes an `old` value and compares it to its current value, if they are equal, it will block until woken up by a notify. The wait will only return when woken up by a notify or spuriously and the old value does not longer match the current value. (cppreference.com 2020b)

Listing 5.9: atomic.cpp

```
#include <atomic>
#include <iostream>

int main() {
    std::atomic<int> counter;
    std::cout << counter.is_lock_free() << std::endl;  // true
    counter.store(0, std::memory_order_release);
    counter.fetch_add(5);
    counter.fetch_sub(3);
    std::cout << counter.load(std::memory_order_acquire) << std::endl; // prints 2
    counter++;
    counter-=3;
    std::cout << counter << std::endl; // prints 0


```

```
16     struct exampleStruct {
17         int a[5]{0};
18     };
19     std::atomic<exampleStruct> example;
20     std::cout << example.is_lock_free() << std::endl; // false
21     exampleStruct nextValue;
22     nextValue.a[3] = 3;
23     nextValue = example.exchange(nextValue);
24     std::cout << example.load().a[3] << std::endl; // prints 3
25     std::cout << nextValue.a[3] << std::endl; // prints 0
26
27
28     std::atomic_flag flag;
29     flag.clear();
30     std::cout << flag.test_and_set() << std::endl; // prints 0
31     std::cout << flag.test_and_set() << std::endl; // prints 1
32
33
34     std::atomic<int> cas;
35     int expectedValue = 3;
36     cas.store(5);
37     cas.compare_exchange_weak(expectedValue, 7); // will not replace value as 3 != 7
38     std::cout << cas.load() << std::endl; // prints 5
39     std::cout << expectedValue << std::endl; // is now set to 5 as when comparison fails,
       expected is set to current value of the atomic
40     cas.compare_exchange_strong(expectedValue, 7); // will exchange
41     std::cout << cas.load() << std::endl; // prints 7
42 }
```

5.3.2 Rust

Atomics in Rust also allows you to specify the ordering of the atomic command, using the same atomic orderings as C++. Contrary to C++, Rust does not have generic atomics, thus only predefined types can be used, these types are however guaranteed to be lock-free. (Rust Team 2020c)

Rust offers atomic types for pointers, integer-types and booleans. All atomic types are prefixed with `Atomic`, for example `AtomicI64`. Available methods depend on the type used. All atomics implement `compare_and_swap` which returns the old value of the atomic. `load`, `store` and `swap` are also available on all types, the same as `compare_exchange` and `compare_exchange_weak`, working the same as C++'s variant where the `weak` variant is allowed to fail spuriously. It is also possible to set the memory orderings for both cases, success and failure, independently. (Rust Team 2020c)

`AtomicBool`'s add the boolean operations `fetch_and`, `fetch_nand`, `fetch_or` and `fetch_xor`. Integer-types, signed and unsigned, also have these operations, and additionally the methods `fetch_add`, `fetch_sub`. `fetch_min` and `fetch_max` are also available, but only as part of the nightly-only experimental API. Also part of this API and a unique feature to Rust is the method `fetch_update` where the current atomic value is fetched, the supplied function `f` applied and then the result saved back to the atomic. The function execution itself is not atomic thus the function may be executed multiple times if the current value changed in the meantime. But the new value will only have the function applied once on it. `fetch_update` takes two orderings, one for fetching the

value and one for setting the new value. The return value is either `Ok(previous_value)` if the passed function returned `Some` or `Err(previous_value)` if `f` returned something else. (Rust Team 2020c)

Listing 5.10: atomic.rs

```
1 #![feature(atomic_min_max)]
2 #![feature(no_more_cas)]
3 #![allow(non_snake_case)]
4 #![allow(unused_must_use)]
5
6 fn main() {
7     use std::sync::atomic::{AtomicI32, Ordering};
8     let counter = AtomicI32::new(0);
9
10     counter.store(0, Ordering::Release);
11     counter.fetch_add(5, Ordering::SeqCst);
12     counter.fetch_sub(3, Ordering::SeqCst);
13     println!("{}", counter.load(Ordering::Acquire)); // prints 2
14
15     let minMax = AtomicI32::new(3);
16     let mut value = minMax.fetch_min(5, Ordering::SeqCst);
17     println!("{}", value); // prints 3
18     value = minMax.fetch_min(1, Ordering::SeqCst);
19     println!("{}", value); // prints 3
20     println!("{}", minMax.load(Ordering::SeqCst)); // prints 1
21
22     let update = AtomicI32::new(0);
23     update.fetch_update(|value| Some(value + 3), Ordering::SeqCst, Ordering::SeqCst)
       ;
24     println!("{}", update.load(Ordering::SeqCst)); // prints 3
25
26     let cas = AtomicI32::new(5);
27     let mut expectedValue = 3;
28     cas.compare_exchange_weak(expectedValue, 7, Ordering::SeqCst, Ordering::SeqCst);
        // will not replace value as 3 != 7
29     println!("{}", cas.load(Ordering::SeqCst)); // prints 5
30     println!("{}", expectedValue); // is still set to 3, in contrast to c++
31     expectedValue = 5;
32     cas.compare_exchange(expectedValue, 7, Ordering::SeqCst, Ordering::SeqCst); //
       will exchange
33     println!("{}", cas.load(Ordering::SeqCst)); // prints 7
34 }
```

5.3.3 Go

Go offers generic atomics in form of `Value` which can store and load a typed value with the help of `interface{}`. `Value` atomics however only offer the methods `Load` and `Store`. (Google 2020a)

Additional methods are available for pointers and integer-types, these atomic operations, however, do not use an atomic type, instead, they operate on the primitive type directly and just atomically change the value of that type. All additional methods are post-fixed with the type they serve, so `StoreInt32` atomically stores `int32`. Besides the always available `Store` and `Load` methods, `Add`, `CompareAndSwap` and just `Swap` are

available. (Google 2020a)

It is not possible in Go to alter the memory orderings as it is automatically managed by the runtime. (Google 2020a)

Listing 5.11: atomic.go

```
1 package main
2
3 import "fmt"
4 import "sync/atomic"
5
6 func main() {
7     var counter int32 = 0
8     atomic.StoreInt32(&counter, 0)
9     atomic.AddInt32(&counter, 5)
10    atomic.AddInt32(&counter, -3)
11    fmt.Println(atomic.LoadInt32(&counter)) // prints 2
12
13    var generic atomic.Value
14    generic.Store("Test")
15    fmt.Println(generic.Load()); // prints "Test"
16
17    var swapInt int64
18    atomic.StoreInt64(&swapInt, 5)
19    fmt.Println(atomic.SwapInt64(&swapInt, 7)) // prints last value, so 5
20    fmt.Println(atomic.LoadInt64(&swapInt)) // prints new value, so 7
21
22    var cas int32
23    var expectedValue int32 = 3
24    atomic.StoreInt32(&cas, 5)
25    atomic.CompareAndSwapInt32(&cas, expectedValue, 7)
26    fmt.Println(atomic.LoadInt32(&cas)) // prints 5
27    fmt.Println(expectedValue) // is still set to 3
28    expectedValue = 5
29    atomic.CompareAndSwapInt32(&cas, expectedValue, 7) // will exchange
30    fmt.Println(atomic.LoadInt32(&cas)) // prints 7
31 }
```

5.3.4 Julia

Similarly to C++, Julia has a generic, atomic type which supports some primitive types. These types are `Bool`, all `Int`s, all `UInt`s and all `Float`s. 128-bit types are not supported on some architectures. (Julia 2020c)

There is no store or load method in Julia, instead the atomic can be accessed by appending `[]` to the variable. This works for getting and setting the variable. All other atomic operations are prepended with the prefix `atomic_`. All atomic operations return the old value of the atomic. (Julia 2020c)

`atomic_cas` is Julia's compare-and-set method, receiving the atomic itself, the compare value `cmp` and `newval`. The atomic will be assigned `newval` if the current value in atomic matches the compare value. `atomic_xchg` exchanges the current value with `newval`. `atomic_add` and `atomic_sub` are available, as well as the boolean operations `atomic_and`, `atomic_nand`, `atomic_or` and `atomic_xor`. `atomic_min` and `atomic_max` compare the current value with the supplied `newval` and then atomically store the min-

imum or maximum of the two values. As only the old value is returned, one has to either compare the returned value with `newval` or re-read the current value in the atomic to get the minimum or maximum of the two numbers. (Julia 2020c)

Listing 5.12: atomic.jl

```
counter = Threads.Atomic{Int32}(0)
counter[] = 0
Threads.atomic_add!(counter, Int32(5))
Threads.atomic_sub!(counter, Int32(3))
println(counter[]) # prints 2

minMax = Threads.Atomic{Int64}(3)
value = Threads.atomic_min!(minMax, 5)
println(value) # prints 3
value = Threads.atomic_min!(minMax, 1)
println(value) # prints 3
println(minMax[]) # prints 1

swapInt = Threads.Atomic{Int64}(0)
swapInt[] = 5
println(Threads.atomic_xchg!(swapInt, 7)) # prints last value, so 5
println(swapInt[]) # prints new value, so 7

cas = Threads.Atomic{Int64}(5)
expectedValue = 3
Threads.atomic_cas!(cas, expectedValue, 7) #will not replace value as 3 != 7
println(cas[]) # prints 5
println(expectedValue) # is still set to 3
expectedValue = 5
Threads.atomic_cas!(cas, expectedValue, 7)
println(cas[]) # prints 7
```

5.3.5 Comparison

Spawn 10 threads. A global counter should be increased by each thread by 1,000,000. Therefore, at the end, the counter should have a value of 10,000,000. For synchronisation an atomic should be used.

Language	Execution Time [S]	SD Time	Memory Usage [MB]	SD Memory
C++	0.189	0.002	3,728	76
Rust	0.19	0.005	2,156	41
Go	0.194	0.015	1,536	22
Julia	0.064	0.002	8.078	0

Table 5.5: Atomic execution metrics.

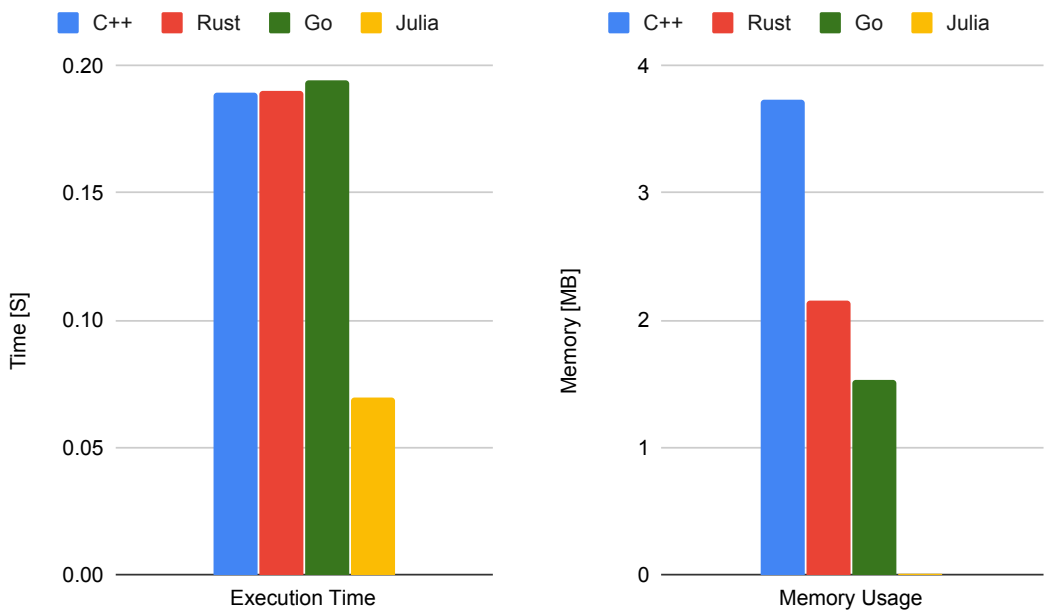

Figure 5.5: Atomic execution graphs.

Language	Compile Time [S]	SD Time	Memory Usage [MB]	SD Memory
C++	0.604	0.011	80,825	87
Rust	0.287	0.019	128,513	507
Go	0.076	0.007	15,227	261
Julia	0.211	0.012	150,525	583

Table 5.6: Atomic compilation metrics.

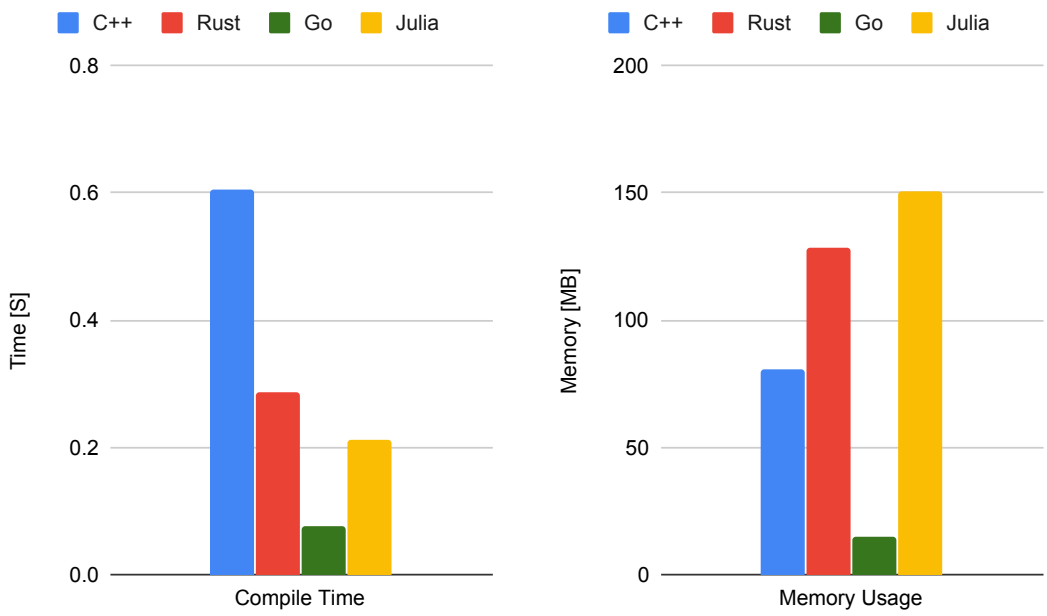

Figure 5.6: Atomic compilation graphs.

Execution & Compilation

All programming languages except Julia take the same amount of time to execute the code (see fig. 5.5 and tab. 5.5). That is because the atomic operations are executed on the hardware itself and are language independent. Only Julia with its runtime has an execution time of 0.06 seconds, more than three times as fast as the other languages. While execution times are the same, memory usage varies between languages with Julia using under 1MB of memory. Next is Go with 1.5MB used, closely followed by Rust with 2.1MB and C++ using nearly twice that with 3.7MB.

Compared, based on compilation time (see fig. 5.6 and tab. 5.6), the fastest language is Go with a compilation speed of just 0.076 seconds. Taking three times as long is Julia with 0.211 seconds, followed by Rust with 0.287 seconds. C++ takes the longest to compile with just above 0.6 seconds until completion. Go also uses the least amount of memory of the four languages with a maximum of 16MB, five times less than the next language, C++, with 80MB used. Most memory is used by Julia with 150MB.

Comparing locks and atomics based on compilation time and memory, there is no visible difference between the two. But execution time decreases greatly when using atomics, with C++, Rust and Julia taking only a fourth of execution time. Go's execution time improves even more, from 1.341 seconds down to 0.194 seconds.

Ease of use

C++ offers the most versatility for atomics. While Rust also provides the developer with the option to alter the memory orderings, and Julia offering generic atomics, only C++ offers both options. C++ also offers the most type aliases for lock-free datatypes.

C++'s atomics are also the only atomics which can act as a replacement for condition variables as they support `wait` and `notify`. Rust offers a similar feature set as C++ with the omission of generic atomics and the condition variable functionality, but also adding one functionality no other language has in the form of `fetch_update`.

Go takes an entirely different approach by not offering atomics and instead relying on atomic operations with addresses of non-atomic data types. Developers must take extra care to not change critical variables in a non-atomic manner as they can also be manipulated without the use of the atomic functions. Go also offers the least functions for atomics, even lacking a subtract function.

Julia offers generic atomics and takes away the conventional `load` and `store` functions, instead accessing the value by appending `[]` to it.

Chapter 6

Closing Remarks

All compared languages offer enough features to effectively design and implement parallel programs. Rust is the only language offering all compared features, ranging from low-level threads to high-level channels. C++, while predating the other languages by decades, still keeps its relevance by implementing most features required to easily make programs execute in parallel. Go and Julia both are a bit more distant from the hardware, apparent by their lack of OS-threads, and lack certain customisation features like launch-policies of tasks or timeouts on condition variables. In return, they compile and run faster than the two other languages.

Personally, the language that fascinated me the most was Julia. Its syntax is similar to Python, it is easy to learn and understand, while offering very fast execution times, something Python may be lacking when compared to compiled languages. Another thing I began to adore was the C++ standard, which provides clear and consistent descriptions of how features should work. While some languages are defined by their reference compiler or only offer a minimalist specification, leaving room for interpretations and misunderstandings, the C++ standard and its process how new features are discussed and implemented leave less room for ambiguities.

A problem faced is the novelty of parallel programming, at least in the compared languages. Some features relevant to this thesis were added while the thesis was being written, making reworks of chapters necessary. Sometimes, I had to rely on proposed changes, like coroutines in C++ (see Subsection 3.1.1), whose specifications may change until they are implemented into the standard itself, outdating the comparison quickly. Also, the description of features was often lacking or even missing as the feature was currently being implemented in the language. Thus, it may be necessary to revisit some chapters, especially *Multithreading* (see Chapter 3), as it may be outdated in a few years after the proposed features are implemented and refined. Still, this thesis offers a glimpse into the status of parallel programming in the four languages C++, Rust, Go and Julia at the beginning of 2020.

Appendix A

Technical Details

A.1 Multithreading - Benchmark

Listing A.1: coroutine.rs

```
#![feature(generators, generator_trait)]

use std::ops::{Generator, GeneratorState};
use std::pin::Pin;

fn main() {
    let mut generator = || {
        for i in 0..100000 {
            yield i;
        }
        return;
    };

    loop {
        match Pin::new(&mut generator).resume() {
            GeneratorState::Yielded(x) => {println!("{}", x.to_string())}
            GeneratorState::Complete(()) => {break;},
        }
    }
}
```

Listing A.2: coroutine.go

```
package main

import "fmt"

func numberator(limit int) <-chan int {
  channel := make(chan int)
  go func() {
    for i := 0; i < limit; i++ {
      channel <- i
    }
    fmt.Println("Yield finished, exiting")
    close(channel)
  }()
```

```
  return channel
}

func main() {
  for i := range numberator(100000) {
    fmt.Println(i)
  }
}
```

Listing A.3: coroutine.jl

```

coroutine() =
    begin
    iterations = 100000

    function numberator(limit)
        Channel() do channel
            for i in 0:limit-1
                put!(channel, i)
            end
        end
    end

    n = numberator(iterations)

    for i in n
        #println(i)
    end
end

precompile(coroutine, ())

for i = 1:10
    @time begin
    coroutine()
    end
end
```

Listing A.4: task.cpp

```
#include <iostream>
#include <future>
#include <vector>
#include <cmath>

int main()
{
    static int AMOUNT_TASKS = 10000;

    std::vector<std::future<double>> tasks;
    for(int i = 0; i < AMOUNT_TASKS; i++) {
        tasks.push_back(std::move(std::async(std::launch::async, []){return sqrt
    (1337);})));
    }

    for(auto& thread : tasks) {
```

```
        std::cout << thread.get();
    }
}
```

Listing A.5: task.rs

```
use futures::executor::{block_on, ThreadPool};
use futures::task::SpawnExt;


fn main() {
        static AMOUNT_TASKS: i32 = 10000;
    let pool = ThreadPool::new().unwrap();

    let mut tasks = vec![];
    for _ in 0..AMOUNT_TASKS {
        tasks.push(pool.spawn_with_handle(async {
            return 1337f64.sqrt();
        }));
    }

    for task in tasks {
        println!("{}", block_on(task.unwrap()));
    }
}
```

Listing A.6: task.go

```
package main

import "fmt"
import "math"


func main() {
    AMOUNT_TASKS := 10000

  future := make(chan float64, AMOUNT_TASKS)
    for i := 0; i < AMOUNT_TASKS; i++ {
        go func() { future <-  math.Sqrt(1337) }()
    }
    for i := 0; i < AMOUNT_TASKS; i++ {
        fmt.Println(<- future)
    }
}
```

Listing A.7: task.jl

```

task() = begin
AMOUNT_TASKS = 10000

tasks = Task[]
for i = 1:AMOUNT_TASKS
    push!(tasks, Threads.@spawn sqrt(1337))
end
for task in tasks
```

```
    fetch(task)
end
end

precompile(task, ())

#=
for i = 1:10
    @time task()
end
=#
```

Listing A.8: thread.cpp

```
#include <thread>
#include <string>
#include <vector>
#include <iostream>
#include <cmath>

int main() {
  static int AMOUNT_THREADS = 10000;

    std::vector<std::thread> threads;
    for(int i = 0; i < AMOUNT_THREADS; i++) {
        threads.push_back(std::thread([](){std::cout << sqrt(1337);}));
    }
    for(auto& thread : threads) {
        thread.join();
    }
}
```

Listing A.9: thread.rs

```
fn main() {
    use std::thread;
    static AMOUNT_THREADS: i32 = 10000;

    let mut threads = vec![];
    for _ in 0..AMOUNT_THREADS {
        threads.push(thread::spawn(move || {
            println!("{}", 1337f64.sqrt());
        }));
    }

    for thread in threads {
        let _ = thread.join();
    }
}
```

A.2 Message Passing - Benchmark

Listing A.10: message.rs

```
use futures::executor::{block_on, ThreadPool};
use futures::task::SpawnExt;
```

```

fn channel_reader(channel: std::sync::mpsc::Receiver<std::time::Instant>) {
    let begin = std::time::Instant::now();
    let mut duration: u128 = 0;
    let mut i: u128 = 0;
    for elem in channel.iter() {
        duration += elem.elapsed().as_nanos();
        i+=1;
    }
    println!("Average duration push - pull: {}ns", duration/i);
    println!("Total time taken: {}ms", begin.elapsed().as_millis());
}

fn main() {
let pool = ThreadPool::new().unwrap();
let task;
{
    let (sender, receiver) = std::sync::mpsc::sync_channel(0);
    task = pool.spawn_with_handle(async { channel_reader(receiver)});
    for _ in 0..10000 {
        sender.send(std::time::Instant::now()).unwrap();
    }
}

block_on(task.unwrap()); //wait for task to finish
}
```

Listing A.11: message.go

```
package main

import "fmt"
import "time"

func channelReader(channel <-chan time.Time) { // this channel is receive only
    begin := time.Now()
    var duration int64
    var i int64
    for elem := range channel { // will block until channel is written into
        duration += time.Since(elem).Nanoseconds()
        i++
    }
    fmt.Println("Average duration push - pull:", duration / i, "ns")
    fmt.Println("Total time taken: ", time.Since(begin))
}

func main() {
  timerChan := make(chan time.Time)
    go channelReader(timerChan)
  for i := 0; i < 10000; i++ {
    timerChan <- time.Now()
  }
  close(timerChan)
  time.Sleep(1000 * time.Millisecond) //as we cannot wait on goroutine
}
```

Listing A.12: message.jl

```
import Dates

task() = begin
channelReader(channel) =
    begin
    start = time_ns()
    duration = 0
    i = 0
    for elem in channel
        duration += time_ns() - elem
        i += 1
    end
    println("Average duration push - pull: ", duration / i, " ns")
    println("Total time taken: ",  start - time_ns())
end

taskref = Ref{Task}();

channel = Channel(channelReader, taskref=taskref, spawn=true)
for i in 10000
    put!(channel, time_ns())
end
close(channel)
fetch(taskref[])
end

precompile(task, ())

#=
for i = 1:10
    @time task()
end
=#
```

A.3 Memory Safety - Benchmark

Listing A.13: lock.cpp

```
#include <thread>
#include <mutex>
#include <iostream>
#include <vector>

const int AMOUNT_THREADS = 10;
const int AMOUNT_ITERATIONS = 1000000;
int counter = 0;
std::mutex mutex;

int main() {
    std::vector<std::thread> threads;
    for(int i = 0; i < AMOUNT_THREADS; i++) {
        threads.push_back(std::thread([](){
            for(int i = 0; i < AMOUNT_ITERATIONS; i++) {
                std::lock_guard<std::mutex> lock(mutex);
                counter++;
            }
```

```
        }));
    }
    for(auto& thread : threads) {
        thread.join();
    }
    std::cout << counter;
    return 0;
}
```

Listing A.14: lock.go

```
package main

import "fmt"
import "sync"

func main() {
    AMOUNT_TASKS := 10
    AMOUNT_ITERATIONS := 1000000
    counter := 0
    var mutex sync.Mutex
    var waitGroup sync.WaitGroup
    waitGroup.Add(AMOUNT_TASKS)

    for i := 0; i < AMOUNT_TASKS; i++ {
        go func() {
            for j := 0; j < AMOUNT_ITERATIONS; j++ {
                mutex.Lock()
                counter++
                mutex.Unlock()
            }
            waitGroup.Done()
        }()
    }
    waitGroup.Wait()
    fmt.Println(counter)
}
```

Listing A.15: lock.rs

```
fn main() {
use std::sync::{Arc, Mutex};
use std::thread;

    static AMOUNT_THREADS: i32 = 10;
    static AMOUNT_ITERATIONS: i32 = 1000000;
    let mutex = Arc::new(Mutex::new(0));

    let mut threads = vec![];
    for _ in 0..AMOUNT_THREADS {
        let mutex = Arc::clone(&mutex);
        threads.push(thread::spawn(move || {
            for _ in 0..AMOUNT_ITERATIONS {
                    let mut counter = mutex.lock().unwrap();
                    *counter += 1;
            }
        }));
```

```
    }

    for thread in threads {
        let _ = thread.join();
    }
    println!("{}", *mutex.lock().unwrap());
}
```

Listing A.16: lock.jl

```
task() = begin
AMOUNT_TASKS = 10
AMOUNT_ITERATIONS = 1000000
mutex = ReentrantLock()
counter = 0

tasks = Task[]
for i = 1:AMOUNT_TASKS
    push!(tasks, Threads.@spawn () = begin
        for j = 1:AMOUNT_ITERATIONS
            lock(mutex)
            counter += 1
            unlock(mutex)
        end
    end)
end
for task in tasks
    fetch(task)
end
println(counter)
end

precompile(task, ())

for i = 1:10
    @time task()
end
```

Listing A.17: condvar.cpp

```
#include <thread>
#include <mutex>
#include <iostream>
#include <condition_variable>
#include <atomic>

const int AMOUNT_PINGS = 100000;

std::condition_variable counterCond;
std::mutex counterMutex;
int counter = 0;

std::condition_variable nextCond;
std::mutex nextMutex;
bool nextValueRequested = true;

std::condition_variable readyCond;
```

```
std::mutex readyMutex;
bool ready = false;

std::atomic<bool> finished;
int main() {
    auto checkerThread = std::thread([](){
        while(true) {
            std::unique_lock<std::mutex> counterLock(counterMutex);
            {
                std::unique_lock<std::mutex> readyLock(readyMutex);
                ready = true;
                readyCond.notify_all();
            }
            if(counter >= AMOUNT_PINGS)
                return;
            {
                std::unique_lock<std::mutex> nextLock(nextMutex);
                nextValueRequested = true;
                nextCond.notify_all();
            }
            counterCond.wait(counterLock);
        }
    });
    {
        std::unique_lock<std::mutex> readyLock(readyMutex);
        readyCond.wait(readyLock, []{return ready;});
    }
    auto adderThread = std::thread([](){
        while(!finished) {
            std::unique_lock<std::mutex> nextLock(nextMutex);
            if(nextValueRequested == true) {
                nextValueRequested = false;
                std::unique_lock<std::mutex> counterLock(counterMutex);
                counter++;
                counterCond.notify_all();
            }
            nextCond.wait(nextLock);
        }
    });
    checkerThread.join();
    finished = true;
    nextCond.notify_all();
    adderThread.join();
    std::cout << counter << std::flush;
    return 0;
}
```

Listing A.18: condvar.go

```
package main

import "fmt"
import "sync"

func main() {
    AMOUNT_PINGS := 100000;

```

```
9     var waitGroup sync.WaitGroup
10    waitGroup.Add(1)
11
12    var counterMutex sync.Mutex
13    counterCond := sync.NewCond(&counterMutex)
14    counter := 0
15
16    var nextMutex sync.Mutex
17    nextCond := sync.NewCond(&nextMutex)
18    nextValueRequested := true
19
20    var readyMutex sync.Mutex
21    readyCond := sync.NewCond(&readyMutex)
22    ready := false
23
24    go func() {
25        for {
26            counterMutex.Lock()
27            readyMutex.Lock()
28            ready = true
29            readyCond.Broadcast()
30            readyMutex.Unlock()
31
32            if(counter >= AMOUNT_PINGS) {
33                waitGroup.Done()
34                return
35            }
36            nextMutex.Lock()
37            nextValueRequested = true
38            nextCond.Broadcast()
39            nextMutex.Unlock()
40            counterCond.Wait()
41            counterMutex.Unlock()
42        }
43    }()
44    readyCond.L.Lock()
45    for !ready {
46        readyCond.Wait()
47    }
48    readyCond.L.Unlock()
49        go func() {
50        for {
51            nextMutex.Lock()
52            if(nextValueRequested == true) {
53                nextValueRequested = false
54                counterMutex.Lock()
55                counter++
56                counterCond.Broadcast()
57                counterMutex.Unlock()
58            }
59            nextCond.Wait()
60            nextMutex.Unlock()
61        }
62    }()
63
64    waitGroup.Wait()
65    fmt.Println(counter)
```

```
}
```

Listing A.19: condvar.rs

```
#![allow(non_snake_case)]
#![allow(unused_must_use)]

fn main() {
use std::sync::{Arc, Mutex, Condvar};
use std::thread;

    static AMOUNT_PINGS: i32 = 100000;
    let counterCondOrig = Arc::new(Condvar::new());
    let counterMutexOrig = Arc::new(Mutex::new(0));

    let nextCondOrig = Arc::new(Condvar::new());
    let nextMutexOrig = Arc::new(Mutex::new(true));

    let readyCondOrig = Arc::new(Condvar::new());
    let readyMutexOrig = Arc::new(Mutex::new(false));

    let nextCond = Arc::clone(&nextCondOrig);
    let nextMutex = Arc::clone(&nextMutexOrig);
    let counterMutex = Arc::clone(&counterMutexOrig);
    let counterCond = Arc::clone(&counterCondOrig);
    let readyCond = Arc::clone(&readyCondOrig);
    let readyMutex = Arc::clone(&readyMutexOrig);
    let checkerThread = thread::spawn(move || {
        loop {
            let counter = counterMutex.lock().unwrap();
            {
                let mut ready = readyMutex.lock().unwrap();
                *ready = true;
                readyCond.notify_all();
            }
            if *counter >= AMOUNT_PINGS
            {
                return;
            }
            {
                let mut next = nextMutex.lock().unwrap();
                *next = true;
                nextCond.notify_all();
            }
            counterCond.wait(counter).unwrap();
        }
    });
    {
        let readyCond = Arc::clone(&readyCondOrig);
        let readyMutex = Arc::clone(&readyMutexOrig);
        let mut ready = readyMutex.lock().unwrap();
        while !*ready {
            ready = readyCond.wait(ready).unwrap();
        }
    }
    let nextCond = Arc::clone(&nextCondOrig);
    let nextMutex = Arc::clone(&nextMutexOrig);
```

```
    let counterMutex = Arc::clone(&counterMutexOrig);
    let counterCond = Arc::clone(&counterCondOrig);
    thread::spawn(move || {
        loop {
            let mut next = nextMutex.lock().unwrap();
            if *next {
                *next = false;
                let mut counter = counterMutex.lock().unwrap();
                *counter += 1;
                counterCond.notify_all();
            }
            nextCond.wait(next).unwrap();
        }
    });
    checkerThread.join().unwrap();
    println!("{}", *counterMutexOrig.lock().unwrap());
}
```

Listing A.20: condvar.jl

```
task() = begin
    AMOUNT_PINGS = 100000

    counterMutex = ReentrantLock()
    counterCond = Threads.Condition(counterMutex)
    counter = 0

    nextMutex = ReentrantLock()
    nextCond = Threads.Condition(nextMutex)
    nextValueRequested = true

    readyMutex = ReentrantLock()
    readyCond = Threads.Condition(readyMutex)
    ready = false

    checkerThread = Threads.@spawn () = begin
        while true
            lock(counterMutex)
            lock(readyMutex)
            ready = true
            notify(readyCond)
            unlock(readyMutex)
            if counter >= AMOUNT_PINGS
                return
            end
            lock(nextMutex)
            nextValueRequested = true
            notify(nextCond)
            unlock(nextMutex)
            wait(counterCond)
            unlock(counterMutex)
        end
    end
    lock(readyMutex)
    while !ready
        wait(readyCond)
    end
```

```
    unlock(readyMutex)
    adderThread = Threads.@spawn () = begin
        while true
            lock(nextMutex)
            if nextValueRequested == true
                nextValueRequested = false
                lock(counterMutex)
                counter += 1
                notify(counterCond)
                unlock(counterMutex)
            end
            wait(nextCond)
            unlock(nextMutex)
        end
    end

    fetch(checkerThread)
    println(counter)
end

precompile(task, ())

# Comment below when just timing the compilation time
for i = 1:10
    @time task()
end
```

Listing A.21: atomic.cpp

```
#include <thread>
#include <atomic>
#include <iostream>
#include <vector>

const int AMOUNT_THREADS = 10;
const int AMOUNT_ITERATIONS = 1000000;
std::atomic<int> counter;

int main() {
    counter.store(0);
    std::vector<std::thread> threads;
    for(int i = 0; i < AMOUNT_THREADS; i++) {
        threads.push_back(std::thread([](){
            for(int i = 0; i < AMOUNT_ITERATIONS; i++) {
                counter.fetch_add(1, std::memory_order_seq_cst);
            }
        }));
    }
    for(auto& thread : threads) {
        thread.join();
    }
    std::cout << counter;
    return 0;
}
```

Listing A.22: atomic.go

```
1 package main
2
3 import "fmt"
4 import "sync"
5 import "sync/atomic"
6
7 func main() {
8     AMOUNT_TASKS := 10
9     AMOUNT_ITERATIONS := 1000000
10    var counter int32 = 0
11    var waitGroup sync.WaitGroup
12    waitGroup.Add(AMOUNT_TASKS)
13
14    for i := 0; i < AMOUNT_TASKS; i++ {
15        go func() {
16            for j := 0; j < AMOUNT_ITERATIONS; j++ {
17                atomic.AddInt32(&counter, 1)
18            }
19            waitGroup.Done()
20        }()
21    }
22    waitGroup.Wait()
23    fmt.Println(atomic.LoadInt32(&counter))
24 }
```

Listing A.23: atomic.rs

```
1 fn main() {
2 use std::sync::{Arc};
3 use std::sync::atomic::{AtomicI32, Ordering};
4 use std::thread;
5     static AMOUNT_THREADS: i32 = 10;
6     static AMOUNT_ITERATIONS: i32 = 1000000;
7     let atomic_global = Arc::new(AtomicI32::new(0));
8
9     let mut threads = vec![];
10    for _ in 0..AMOUNT_THREADS {
11        let atomic = atomic_global.clone();
12        threads.push(thread::spawn(move || {
13            for _ in 0..AMOUNT_ITERATIONS {
14                    atomic.fetch_add(1, Ordering::SeqCst);
15            }
16        }));
17    }
18
19    for thread in threads {
20        let _ = thread.join();
21    }
22    println!("{}", atomic_global.load(Ordering::SeqCst));
23 }
```

Listing A.24: atomic.jl

```
1 atomic() = begin
2 AMOUNT_TASKS = 10
3 AMOUNT_ITERATIONS = 1000000
4 mutex = ReentrantLock()
```

```
5 counter = Threads.Atomic{Int32}(0)
6
7 tasks = Task[]
8 for i = 1:AMOUNT_TASKS
9     push!(tasks, Threads.@spawn () = begin
10         for j = 1:AMOUNT_ITERATIONS
11             Threads.atomic_add!(counter, Int32(1))
12         end
13     end)
14 end
15 for task in tasks
16     fetch(task)
17 end
18 println(counter[])
19 end
20
21 precompile(atomic, ())
22
23 #=
24 for i = 1:10
25     @time atomic()
26 end
27 =#
```

Appendix B

CD-ROM Contents

Format: CD-ROM, Single Layer, ISO9660-Format

B.1 PDF-Files

Path: /

_thesis_EN.pdf	Thesis (Main document)
references.bib	Literature (BibTeX-File)

B.2 Code-Files

Path: /code/multithreading/

benchmarks/	Contain all benchmark code-files
examples/	Contain all example code-files

Path: /code/messages/

benchmarks/	Contain all benchmark code-files
examples/	Contain all example code-files

Path: /code/memory/

benchmarks/	Contain all benchmark code-files
examples/	Contain all example code-files

B.3 Reference-Files

Path: /code/reference/

[AUTHOR]/	All online sources downloaded for offline use

References

Literature

cppreference.com (June 2018). *std::launch - cppreference.com*. URL: https://en.cppreference.com/mwiki/index.php?title=cpp/thread/launch&oldid=103142 (visited on 11/20/2019).

— (Mar. 2020a). *std::async - cppreference.com*. URL: https://en.cppreference.com/mwiki/index.php?title=cpp/thread/async&oldid=117018 (visited on 05/22/2020).

— (Apr. 2020b). *std::atomic - cppreference.com*. URL: https://en.cppreference.com/mwiki/index.php?title=cpp/atomic/atomic&oldid=117865 (visited on 05/15/2020).

— (Apr. 2020c). *std::condition_variable - cppreference.com*. URL: https://en.cppreference.com/mwiki/index.php?title=cpp/thread/condition_variable&oldid=118288 (visited on 05/15/2020).

— (Mar. 2020d). *std::memory_order - cppreference.com*. URL: https://en.cppreference.com/mwiki/index.php?title=cpp/atomic/memory_order&oldid=117145 (visited on 05/22/2020).

— (Jan. 2020e). *std::thread - cppreference.com*. URL: https://en.cppreference.com/mwiki/index.php?title=cpp/thread/thread&oldid=114799 (visited on 05/22/2020).

— (Jan. 2020f). *std::thread::get_id - cppreference.com*. URL: https://en.cppreference.com/mwiki/index.php?title=cpp/thread/thread/get_id&oldid=114791 (visited on 01/24/2020).

Deshpande, Neil, Erica Sponsler, and Nathaniel Weiss (Dec. 2012). *Analysis of the Go runtime scheduler*. Columbia University. URL: http://www.cs.columbia.edu/~aho/cs6998/reports/12-12-11_DeshpandeSponslerWeiss_GO.pdf (visited on 01/02/2020).

Donovan, A.A.A. and B.W. Kernighan (2015). *The Go Programming Language*. Addison-Wesley Professional Computing Series. Pearson Education. URL: https://books.google.at/books?id=SJHvCgAAQBAJ.

Galowicz, J. (2017). *C++17 STL Cookbook*. Packt Publishing. URL: https://books.google.at/books?id=XRBeMQAACAAJ.

Google (July 2019). *The Go Programming Language Specification*. URL: https://web.archive.org/web/20191122052628/https://golang.org/ref/spec (visited on 11/22/2019).

— (2020a). *atomic - The Go Programming Language*. URL: https://web.archive.org/web/20200520052250/https://golang.org/pkg/sync/atomic/ (visited on 05/20/2020).

— (2020b). *sync - The Go Programming Language*. URL: https://web.archive.org/web/20200520093246/https://golang.org/pkg/sync/ (visited on 05/20/2020).

ISO/IEC (Mar. 2017). *Working Draft, Standard for Programming Language C++*. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4659.pdf (visited on 05/12/2020).

— (Mar. 2020). *Programming Languages — C++*. URL: https://isocpp.org/files/papers/N4860.pdf (visited on 05/22/2020).

Julia (2020a). *Distributed Computing - The Julia Language*. Version 1.3. URL: https://docs.julialang.org/en/v1.3/stdlib/Distributed/ (visited on 01/02/2020).

— (2020c). *Multi-Threading - The Julia Language*. Version 1.3. URL: https://docs.julialang.org/en/v1.3/base/multi-threading/ (visited on 05/13/2020).

— (2020d). *Tasks - The Julia Language*. Version 1.3. URL: https://docs.julialang.org/en/v1.3/base/parallel/ (visited on 01/02/2020).

Klabnik, Steve and Carol Nichols (2018). *The Rust Programming Language*. 2018th ed. URL: https://doc.rust-lang.org/1.40.0/book/ (visited on 10/10/2019).

Pike, Rob (Oct. 2012). "Go at Google: Language Design in the Service of Software Engineering". URL: https://talks.golang.org/2012/splash.article (visited on 11/20/2019).

Rust Team (2020a). *generators - The Rust Unstable Book*. Version 1.42.0. URL: https://doc.rust-lang.org/1.42.0/unstable-book/language-features/generators.html (visited on 01/02/2020).

— (2020b). *std::future::Future - Rust*. Version 1.40.0. URL: https://doc.rust-lang.org/1.40.0/std/future/trait.Future.html (visited on 01/02/2020).

— (2020c). *std::sync::Atomic - Rust*. Version 1.40.0. URL: https://doc.rust-lang.org/1.40.0/std/sync/atomic/index.html (visited on 05/15/2020).

— (2020d). *std::sync::Condvar - Rust*. Version 1.42.0. URL: https://doc.rust-lang.org/1.42.0/std/sync/struct.Condvar.html (visited on 05/15/2020).

— (2020e). *std::sync::mpsc - Rust*. Version 1.40.0. URL: https://doc.rust-lang.org/1.40.0/std/sync/mpsc/ (visited on 05/01/2020).

— (2020f). *std::sync::Mutex - Rust*. Version 1.40.0. URL: https://doc.rust-lang.org/1.40.0/std/sync/struct.Mutex.html (visited on 05/14/2020).

— (2020g). *std::sync::RwLock - Rust*. Version 1.40.0. URL: https://doc.rust-lang.org/1.40.0/std/sync/struct.RwLock.html (visited on 05/14/2020).

— (2020h). *std::thread - Rust*. Version 1.40.0. URL: https://doc.rust-lang.org/1.40.0/std/thread/ (visited on 01/02/2020).

Schmidt, Bertil et al. (2017). *Parallel Programming: Concepts and Practice*. Elsevier Science. URL: https://books.google.at/books?id=-y9HDgAAQBAJ.

Sutter, Herb (Mar. 2005). "The Free Lunch Is Over". *Dr. Dobb's Journal* 30.3.

Online sources

Bezanson, Jeff, Stefan Karpinski, et al. (Feb. 2012). *Why We Created Julia*. URL: https://julialang.org/blog/2012/02/why-we-created-julia (visited on 10/26/2019).

Bezanson, Jeff, Jameson Nash, and Kiran Pamnany (July 2019). *Announcing composable multi-threaded parallelism in Julia*. URL: https://julialang.org/blog/2019/07/multithreading (visited on 01/02/2020).

Julia (2020b). *Julia Micro-Benchmarks*. URL: https://julialang.org/benchmarks/ (visited on 05/24/2020).

Matsakis, Niko (Nov. 2019). *Async-await on stable Rust!* URL: https://blog.rust-lang.org/2019/11/07/Async-await-stable.html (visited on 01/02/2020).

Microsoft (Feb. 2013). *What's New in the .NET Framework 4*. URL: https://docs.microsoft.com/en-us/previous-versions/dotnet/netframework-4.0/ms171868(v=vs.100) (visited on 05/22/2020).

Rust Team (2019). *Frequently Asked Questions Rust*. URL: https://prev.rust-lang.org/en-US/faq.html (visited on 10/21/2019).

Stack Overflow (2019). *Stack Overflow Developer Survey 2019*. URL: https://insights.stackoverflow.com/survey/2019 (visited on 10/21/2019).