# Evaluation of performance and productivity metrics of potential programming languages in the HPC environment

**— Bachelor Thesis —**

Division Scientific Computing
Department of Informatics
Faculty of Mathematics, Informatics und Natural Sciences
University of Hamburg

| | |
|---|---|
| Submitted by: | Florian Wilkens |
| E-Mail: | 1wilkens@informatik.uni-hamburg.de |
| Matriculation number: | 6324030 |
| Course of studies: | Software-System-Entwicklung |
| | |
| First assessor: | Prof. Dr. Thomas Ludwig |
| Second assessor: | Sandra Schröder |
| | |
| Advisor: | Michael Kuhn, Sandra Schröder |

Hamburg, March 5, 2015

# Abstract

This thesis aims to analyze new programming languages in the context of HPC. To compare not only speed but also development productivity and general inner metrics, a basic traffic simulation is implemented in C, Mozilla's Rust and Google's Go. These two languages were chosen on their basic promise of performance as well as memory-safety in the case of Rust or easy multhithreaded execution (Go). The implementations are limited to shared-memory parallelism to achieve a fair comparison since the library support for inter-process communication is rather limited at the moment.
Nonetheless the comparison should allow a decent rating of the viability of these two languages in high-performance computing.

# Table of Contents

**1 Introduction** **4**
1.1 Motivation . . . . . 4
1.2 Goals of this thesis . . . . . 5

**2 State of the art** **6**
2.1 Weaknesses of C and Fortran . . . . . 6
2.2 Language candidates . . . . . 6

**3 Approach** **13**
3.1 Overview: Streets4MPI . . . . . 13
3.2 Differences to the base implementation . . . . . 13
3.3 Implementation process . . . . . 15

**4 Implementation** **16**

**5 Evaluation** **17**

**6 Conclusion** **18**

**Bibliography** **19**

**List of Figures** **21**

**List of Tables** **22**

**List of Listings** **23**

**Appendices** **24**

**A Appendix** **25**

# 1. Introduction

*This chapter provides some background information to high-performance computing. The first section describes problems with the currently used programming languages and motivates the search for new candidates. After that the chapter concludes with a quick rundown of the thesis' goals.*

## 1.1. Motivation

The world of high-performance computing is evolving rapidly and programming languages used in this environment are held up to a very high standard. It comes as no surprise that runtime performance is the top priority in language selection when an hour of computation costs thousands of dollars.[Lud11] The focus on raw power led to C and Fortran having an almost monopolistic position in the industry, because their execution speed is nearly unmatched.

However programming in these rather antique languages can be rather difficult. Although they are still in active development, their long lifespans resulted in sometimes unintuitive syntax and large amounts of historical debt. Especially C's *undefined behaviour* often causes inexperienced programmers to write unreliable code which is unnecessarily dependant on implementation details of a specific compiler. Understanding and maintaining these programs requires deep knowledge of memory layout and other technical details.

Considering the fact that scientific applications are often written by scientist without a concrete background in compuer science it is evident that the current situation is less than ideal. There have been various efforts to make programming languages more accessible in the recent years but unfortunately none of the newly emerged ones have been successful in establishing themselves in the HPC community to this day. Although many features and concepts have found their way in newer revision of C and Fortran standards most of them feel tacked (//wording) on and are not well integrated into the core languages.

One example for this is the common practice of testing. Specifically with the growing popularity of test driven development it became vital to the development process to be able to quickly and regularly execute a set of tests to verfiy growing implementations as they are developed. Of course there are also testing frameworks and libraries for Fortran and C but since these languages lack deep integration of testing concepts, they often

require a lot of setup and boilerplate code and are generelly not that pleasant to work with. In contrast for example the Go programming language includes a complete testing framework with the ability to perform benchmarks, execute global setup/teardown work and even do basic output testing. [mai] (// cite vs footnote) Maybe most important all this is available via a single executable `go test` which may be easily integrated in scripts or other parts of the workflow.

While testing is just one example there are a lot of "best practices" and techniques which can greatly increase both developer productivity and code quality but require a language-level integration to work best. Combined with the advancements in type system theory and compiler construction both C and Fortran's feature sets look very dated. With this in mind it is time to evaluate new potential successors of the two giants of high-performance computing.

## 1.2. Goals of this thesis

This thesis aims to evaluate Rust and Go as potential programming languages in the high-performance computing environment. The comparison is based on a port of an existing parallel application orginally written in Python. It was ported to the two language candidates as well as C to have a fair base to compare to. Since libraries for interprocess communication in Rust and Go are nowhere near production-ready this thesis will focus on shared memory parallelization to avoid unfair bias based solely on the quality of the supporting library ecosystem.

The final application is a simplified version of the original but will behave nearly identical. It will use similar input formats and write result to a reduced custom output format for later analysis. To reduce complexity it does not support additional commandline arguments and has limited error handling regarding in- and output.

While performance will be the main concern additional software metrics will also be reviewed to measure the complexity and overall quality of the produced applications. Another aspect to review is the tool support and ease of development.

# 2. State of the art

*This chapter describes the current state of the art in high-performance computing. The dominance of Fortran and C is explained and questioned in* **??**. *After that all considered language candidates are introduced and characterized.*

- state of C and Fortran (section name?)

- technological advancements in low level languages - static analysis - .. -> But no real adaption possible, because language level support is missing (already included in introduction)

## 2.1. Weaknesses of C and Fortran

As stated in section 1.1 high-performance computing is largely dominated by C and Fortran. To understand why a new language is needed it is essential to understand the shortcomings of these programming veterans. (//language?)

The main drawback of both languages is their age. Although new revisions are regularly aproves Fortran and C strive to be backwards compatible for the most part. This has some very serious consequences especially in their respective syntax. A lot of features of newer standards are integrated suboptimally to preserve backwards compatability.

// Candidates here for now might need another chapter for those

## 2.2. Language candidates

As previously stated Go and Rust were chosen to be evaluated in the context of high-performance computing. This section aims to provide a rough overview of all possible language candidates that were considered for further evaluation in this thesis.

## Python

Python is an interpreted general-purpose programming language which aims to be very expressive and flexible. Compared with C and Fortran which sacrifice feature richness for performance, Python's huge standard library combined with the automatic memory management offers a low border of entry and quick prototyping capabilities.

As a matter of fact many introductory computer science courses at universities in the United States recently switched from Java to Python as their first programing language. [Guo14; Lub14] This allows the students to focus on core concepts and algorithms instead of boilerplate code.

```
1  # Function signatures consist only of one keyword (def)
2  def fizzbuzz(start, end):
3      # Nested function definition
4      def fizzbuff_from_int(i):
5          entry = ''
6          if i%3 == 0:
7              entry += "Fizz"
8          if i%5 == 0:
9              entry += "Buzz"
10         # empty string evaluates to false (useable in
               ↪ conditions)
11         if not entry
12             entry = str(i)
13         return entry
14     # List comprehensions are the pythonic way of composing
           ↪ lists
15     return [int_to_fizzbuzz(i) for i in range(start, end+1)]
```

Listing 2.1: FizzBuzz in Pyhon 3.4

In addition to the very extensive standard library the Python community has created a lot of open source projects aiming to support especially scientific applications. There is NumPy [1] which offers efficient implementations for multidimensional arrays and common numeric algorithms like Fourier transforms or MPI4Py [2], an MPI abstraction layer able to interface with various backends like OpenMPI or MPICH. Especially the existance of the latter shows the ongoing attempts to use Python in a cluster environment and there have been successful examples of scientific high-performance applications using these libraries( //need ref ).

Unfortunately dynamic typing and automatic memory management come at a rather high price. The speed of raw numeric algorithms written in plain Python is almost always

[1] http://www.numpy.org
[2] http://www.mpi4py.scipy.org

orders of magnitude slower than implementations in C or Fortran. As a consequence nearly all of the mentioned libraries implement the critical routines in C and focus in optimizing the interop (// wording) experience between the two languages. This often means one needs to make tradeoffs between idiomatic Python - which might not be transferable to the foreign language - and maximum performance. As a result performance critical ython code often looks like it's equivalent written in a statically typed language. The more terseness Python loses because of this, the less desireable it becomes to use in high-performance computing since one could just fall back to C for a similar experience.

In conclusion Python was not chosen to be further evaluated because of the mentioned lack of performance (in pure Python). This might change with some new implementations emerging recently though. Most of the problems discussed here are present in all stable Python implementations today (most notably *Cython* and *PyPy*) but new projects aim to improve the execution speed in various ways. *Medusa* compiles Python code to Google's Dart to make use of the underlying virtual machine. Although these ventures are still in early phases of development, first early benchmarks promise drastic performance improvements. Once Python can achieve similar execution speed to native code it will become a serious competitor in the high performance area.

## Erlang

Erlang is a relatively niche programming language originally designed for the use in telephony applications. It features a high focus on concurrency and a garbage collector which is enabled through the execution inside the BEAM virtual machine. Today it is most often used in soft real-time computing [3] because of it's error tolerance, hot code reload capabilities and lock-free concurrency support.

- brief history?

- code example (not hello world rather show message passing)

- Upsides - Great concurrency - Message passing is default (no locks) - Hot swap? - Downsides - Bad interfacing to other languages - Weird syntax - Limited (community/-support?)

## Go

Go is a relatively new programming language which focusses on simplicity and clarity while not sacrificing too much performance. Initially developed by Google it aims to "make it easy to build simple, reliable and efficient software" (//cite). It is statically typed, offers a garbage collector, basic type inference and a large standard library. Go's syntax is loosely inspired by C but made some major changes like removing the mandatory semicolon at the end of commands and changing the order of types and identifiers. It

[3] see `https://en.wikipedia.org/wiki/Real-time_computing`

was chosen as a candidate because it provides simple concurrency primitives as part of the language (so called *goroutines*) while having a familiar syntax and beeing reasonably performant. [Dox12] It also compiles to native code without external dependencies which makes it usable on cluster computers without many additional libraries installed.

The chosen code example demonstrates two key features which are essential to concurrent programming in Go - the already mentioned goroutines as well as channels used for synchronization purposes. These provide a way to communicate with running goroutines via message passing.

```
1  package main
2
3  import "fmt"
4
5  // Number of gorountines to start
6  const GOROUTINES = 4
7
8  func helloWorldConcurrent() {
9      // Create a channel to track completion
10     c := make(chan int)
11
12     for i := 0; i < GOROUTINES; i++ {
13         // Start a goroutine
14         go func(nr int) {
15             fmt.Printf("Hello from routine %v", nr)
16             // Signalize completion via channel
17             c <- 1
18         }(i)
19     }
20
21     for i := 0; i < GOROUTINES; i++ {
22         // Wait for completion of all goroutines
23         <-c
24     }
25 }
```

Listing 2.2: Go concurrency example

Initially developed for server scenarios Go has seen production use in many different areas. At Google it is used for various internal project such as the download service “dl.google.com” which has been completely rewritten from C++ to Go in 2012. The new version can handle more bandwith while using less memory. It is also noteable that the Go codebase is about half the size of the legacy application with increased test coverage and performance. [Fit13]

While Go's focus on simplicity is admireable it has also been it's greatest point of criticism. The language feature set is very carefully selected and rarely extended. It even misses some of the most natural constructs which a programmer might expect in a reasonably high-level language - the main example for this beeing generics. As of the time of this writing Go does not offer the common concept of generic structs (or classes) and the authors have stated this is not a big priority at the moment.

One other important fact - especially for high-performance computing - is the mandatory garbage collector. Go completey takes the burden of memory management out of the hands of the programmer and relies on the embedded runtime to efficiently perform this job. This makes it impossible to predictably allocate and release memory which can lead to performance loss. This also means the Go runtime has to be statically linked into every application. Although that might not be important for bigger codebases it increases the binary size considerably.

In the end Go was mainly chosen to be evaluated further because of promised "simple" parallelism via goroutines. It will probably not directly compete with C in execution performance but the great toolchain and simplified concurrency might outshine the performance loss.

- Prediction implementation - A bit of syntax weirdness - Relatively quick PoC with decent concurrency aspects - Some fixing/optimization afterwards regarding common concurrency errors -> More time spent after initial PoC but less than in C

## Rust

The last candidate discussed in this chapter is Rust. Developed in the open but strongly backed by Mozilla Rust aims to directly compete with C and C++ as a systems language. It focuses on memory safety which is checked and verified at compile without (or with minimal) impact on runtime performance. Rust compiles to native code using a custom fork of the popular LLVM[4] as backend and is compatible to common tools like **The GNU Project Debugger** (*gdb*) [5] which makes integration into existing workflows a bit easier.

Out of the here discussed languages Rust is closest to C while attempting to fix common mistakes made possible by it's loose standard allowing undefined behaviour. (//wording?) Memory safety is enforced through a very sophisticated model of ownership. It is based on common concepts which are already employed on concurrent applications but integrates them on a langage level and enforces them at compile time. The basic rule is that every resource in an applications (for example allocated memory or file handles) has exactly one *owner* at a time. To share access to a resource one can you use references denoted by a '&'. These can been seen as pointers in C with the additional caveat that they are readonly. To gain mutable access to a resource one must acquire a mutable reference

[4] `http://www.llvm.org`
[5] `http://www.gnu.org/software/gdb/`

via '&mut'. To ensure memory safety a special part of the compiler, the *borrow checker*, validates that there is never more than one mutable reference to the same resource. This effectively prevents mutable aliasing which in turn rules out a whole class of errors like iterator invalidation. It is important to remember that these checks are a "zero cost abstraction" which means they do not have any runtime overhead but enforce additional security at compile time through static analysis.

Another core aspect of Rust are *lifetimes*. As many other programming languages Rust has scopes introduced by blocks for example function and loop bodies or arbitrary scopes opened and closed by curly braces. Combined with the ownership system the compiler can exactly determine when the owner of a resource gets out of scope and call the appropiate destructor (called drop in Rust). This technique is called "Resource acquisition is initialization". [Str94, p. 389] Unlike in C++ it is not limited to stack allocated objects since the compiler can rely on the ownership system to verify that no references to a resource are left when its owner gets out of scope. It is therefore safe to drop.

```
// Immutability per default, Option type built-in -> no null
fn example(val: &i32, mutable: &mut i32) -> Option<String> {
    // Pattern matching
    match *val {
        /* Ranges types (x ... y notation),
         * Powerful macro system (called via 'macro!()') */
        v @ 1 ... 5 => Some(format!("In [1, 5]: {}", v)),
        // Conditional matching
        v if v < 10 => Some(format!("In [6,10): {}", v)),
        // Constant matching
        10          => Some("Exactly 10".to_string()),
        /* Exhaustiveness checks at compile time,
         * '_' matches everything */
        _           => None
    }
    // statements are expressions -> no need for 'return'
}
```

Listing 2.3: Rust example

Although Rust focusses on performance and safety it also adopted some functional concepts like *pattern matching* and the Option type. Combined with range expressions and macros which operate on syntax level codong in Rust often feels like in a scripting language which is just very performant. This was also the main reason it was chosen to be further evaluated. Rust focusses on safety while not sacrificing any performnce in the process. Most of the checks happen at compile time making the resulting binary often close or on par with equivalent C programs. It also has the advantage of beeing still in

development [6] so concepts which did not work out can be quickly changed or completely dropped. But Rust's immatureness is also its greatest weakness. The

- mention cargo

- Prediction Implementation - Moderatly quick PoC without concurrency at first - Nearly only otimization afterwards since compilation secures memory safety -> More time spent before initial PoC than after

## Comparison

| Rust | Python | Erlang | Go |
|---|---|---|---|
| Execution model<br>compiled to native code | interpreted | compiled to bytecode | compiled to native |
| Advantagesree) concurrency support | adv go | adv rust | |
| Disadvantages<br>disadv rust | speed | obscure syntax | mandatory runtim |
| Relative speed<br>speed rust | slow to average | average to fast | speed go |

[6] current version beeing **1.0.0-beta-1** at the time of this writing

# 3. Approach

*The first section of the third chapter describes the existing application the evaluation is based on. In addition the implementation process is illustrated and methods the comparison of languages is based on are introduced.*

## 3.1. Overview: Streets4MPI

As stated in section 1.2 the language evaluation is based on a reimplementation of an existing parallel program originally written in Python. The application in question is *streets4MPI* - a traffic simulation using OpenStreetMap files as input and calculating shortest routes via Dijkstra's algorithm. Streets4MPI was implemented in scope of the module "Parallel Programming Project" in Spring 2012. It was written by Julian Fietkau and Joachim Nitschke and makes heavy use of the various libraries of the Python ecosystem.

//TODO: more description

The original project contained the main simulation as well as a visualization script. [FN12, p. 3] For the purpose of this thesis the visualization was omitted and the evaluation is only based on software quality metrics as well as the result's correctness. //TODO: really cite page?

## 3.2. Differences to the base implementation

Although the evaluated implementations are based on the original Streets4MPI, there are some key differences (mainly to reduce complexity). This section gives a brief overview over the most important aspects that have been changed and describes both the original application as well as the the redeveloped version.

In the remaining part of the thesis the different implementations will be referenced quite frequently. For brevity each application will be called by its name "streets4<language>" for example "streets4go". //wording..

## Input format

The original implementation used the somewhat dated OpenStreetMap XML format [1] as input which is parsed by *imposm.parser* [2]. Streets4MPI then builds a directed graph via the *python-graph* [3] library to base the simulation on.[FN12]

The reimplementations require the input to be in ".osm.pbf" format. This newer version of the OpenStreetMap format is based on Google's *Protocol Buffers* [4] and is superior to the XML variant in both size and speed [Pro]. It also simplifies multilanguage development because the code performing the actual parsing is auto generated from a language independent description file and there are protobuf backends for C, Rust and Go which can perform that generation.

## Simulation

The simulation in streets4MPI is based on randomly generated trips in the directed graph that was build from the input. For these trips the shortest path is calculated by Dijkstra's algorithm (//cite). To avoid oscillation a random factor called "jam tolerance" is introduced. Then after some time has passed in the simulation existing streets get expanded or shut down depending on the usage. The reimplementations to compare also perform trip based simulation but to without the added randomness and street modification. Also Diskstra's algorithm was implemented in all three languages in a benchmarkable way so it can be compared separately. This is done to rule out possible errors in the complete streets4x implementations.

## Concurrency

The base application parallelizes its calculations on multiple processes that communicate via message passing. This is achieved with the aforementioned MPI4Py library which delegates to a native MPI implementation installed on the system if possible to reach maximum performance.

Although Rust as well as Go can integrate decently with existing native code, the reimplementations will be limited to shared memory parallelization on threads. This was mostly decided to evaluate and compare the language inherent concurrency constructs rather than the quality of the forrign funtion interface. To achieve fair comparability *streets4c* will use OpenMP [5] as it is the de facto standard for simple thread parallelization in C.

[1] `http://wiki.openstreetmap.org/wiki/OSM_XML`
[2] `http://imposm.org/docs/imposm.parser/latest/`
[3] `https://code.google.com/p/python-graph/`
[4] `https://developers.google.com/protocol-buffers/`
[5] `http://www.openmp.org`

## 3.3. Implementation process

The implementation process was performed iteratively. Certain milestones were defined and implemented in all three languages. The process only proceeded to the next phase when the previous milestone was reached in all applications. This approach was chosen to allow for a fair comparison of the different stages of development. If the implementations would have been developed one after another to completion, this might have introduced a certain bias to the evaluation because of possible knowledge about the problem aquired in a previous language translating to faster results in the next one.

For each stage various characteristics were captured and compared to highlight the languages' features and performance in the varaious areas

### 0. [Setting up the project]

The first stage of development was to create project skeletons and infrastructure for the future development. The milestone was to have a working environment in place where the sample application could be build and executed. While this is certainly not the most important or even interesting part it did show the differences in comfort between the various toolchains.

### 1. Counting nodes, ways and relations in an .osm.pbf file

The first real milestone was to read a .osm.pbf file and count all nodes, ways and relations in it. This was done to get familiar with the required libraries and the file format in general. The time recorded began from the initial project created in stage 0 and finished after the milestone was reached. As this is the most input and output intensive stage it already showed some key differences between the candidated both in speed and also memory consumption.

# 4. Implementation

*In diesem Kapitel ...*

Milestones: - Count all nodes, ways and relations in hamburg-latest.osm.pbf - Rust: - SLOC: 37 - dev-time: 1989 -> 33,15 min - run-time: - -O0: target/streets4rust ../osm/hamburg-latest.osm.pbf 27,61s user 0,12s system 99% cpu 27,749 total - -O3: target/release/streets4rust ../osm/hamburg-latest.osm.pbf 2,59s user 0,13s system 99% cpu 2,722 total - allocs: total heap usage: 11,373,558 allocs, 11,373,557 frees, 2,186,107,072 bytes allocated - counts: Found 2180418 nodes, 409424 ways and 7182 relations in ../osm/hamburg-latest.osm.pbf - notes: - easy dependency management, huuge optimization gains - Go: - SLOC: 55 (+ helper functions) - dev-time: 1276 -> 21,27 min - run-time: - GOMAXPROCS=1: ./streets4go ../osm/hamburg-latest.osm.pbf 4,85s user 0,05s system 101% cpu 4,846 total - GOMAXPROCS=8: ./streets4go ../osm/hamburg-latest.osm.pbf 9,00s user 0,28s system 672% cpu 1,381 total - allocs: total heap usage: 11,164,068 allocs, 11,000,199 frees, 1,447,543,184 bytes allocated - counts: Found 2180418 nodes, 409424 ways and 7182 relations in ../osm/hamburg-latest.osm.pbf - library parallelizeable, but singlethreaded slower - C: - SLOC: 55 - dev-time: 3078 -> 51,30 min - run-time: - -O0: ./streets4c ../osm/hamburg-latest.osm.pbf 0,95s user 0,07s system 99% cpu 1,017 total - -O3: ./streets4c ../osm/hamburg-latest.osm.pbf 0,92s user 0,08s system 99% cpu 0,994 total - allocs: total heap usage: 2,390,566 allocs, 2,390,566 frees, 372,758,206 bytes allocated - counts: Found 2180418 nodes, 409424 ways and 7182 relations in ../osm/hamburg-latest.osm.pbf - no library available, linking problems, freeing problems, fastest solution (run), slowest solution (dev) - Write basic graph structure - notes: - edges <-> streets: - ID (OSMID) - vertices (from node and to node) - vertices <-> nodes: - ID (OSMID) -

# 5. Evaluation

*In diesem Kapitel ...*

# 6. Conclusion

*In diesem Kapitel ...*

- Only evaluated shared memory -> Multi process implementations -> C: MPI, Rust: MPI via C FFI & opaque pointer, Go: MPI via wrapper? (less idiomatic code)

# Bibliography


[Che+07] Mo Chen et al. *Priority Queues and Dijkstra's Algorithm*. Technical report TR-07-54. The University of Texas at Austin Department of Computer Science, Oct. 12, 2007. URL: http://www3.cs.stonybrook.edu/~rezaul/papers/TR-07-54.pdf.

[Cor+09] Thomas H. Cormen et al. *Introduction to Algorithms, 3rd Edition*. 3rd. The MIT Press, July 2009. ISBN: 9780262033848.

[CT09] Francesco Cesarini and Simon Thompson. *Erlang programming. A Concurrent Approach to Software Development*. Beijing: O'Reilly, June 2009. ISBN: 9780596518189.

[Dox12] Caleb Doxsey. *An Introduction to Programming in Go*. Lexington, KY: CreateSpace Independent Publishing Platform, Sept. 2012. ISBN: 9781478355823 (cit. on p. 9).

[Fit13] Bradley Joseph Fitzpatrick. *dl.google.com: Powered by Go*. July 26, 2013. URL: http://talks.golang.org/2013/oscon-dl.slide (visited on 18. 02. 2015) (cit. on p. 9).

[FN12] Julian Fietkau and Joachim Nitschke. *Project Report: Streets4MPI*. Hamburg, May 21, 2012. URL: http://wr.informatik.uni-hamburg.de/_media/research/labs/2012/2012-05-julian_fietkau_joachim_nitschke-streets4mpi-report.pdf (visited on 18. 02. 2015) (cit. on pp. 13, 14).

[Guo14] Philip Guo. *Python is Now the Most Popular Introductory Teaching Language at Top U.S. Universities*. Oct. 7, 2014. URL: http://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-us-universities/fulltext (visited on 07. 12. 2014) (cit. on p. 7).

[Lub14] Bill Lubanovic. *Introducing Python. Modern Computing in Simple Packages*. 1st ed. Beijing: O'Reilly Media, Inc., Nov. 26, 2014. ISBN: 9781449359362 (cit. on p. 7).

[Lud11] Thomas Ludwig. *The Costs of Science in the Exascale Era*. May 31, 2011. URL: http://perso.ens-lyon.fr/laurent.lefevre/greendaysparis/slides/greendaysparis_Thomas_Ludwig.pdf (visited on 02. 12. 2014) (cit. on p. 4).

[mai] The Go project maintainers. *The Go programming language - Package testing*. URL: http://golang.org/pkg/testing/ (visited on 11. 02. 2015) (cit. on p. 5).

[Pro] The OpenStreetMap Project. *OpenStreeMap Wiki - "PBF Format"*. URL: `http://wiki.openstreetmap.org/wiki/PBF_Format` (visited on 18. 02. 2015) (cit. on p. 14).

[Str94] Bjarne Stroustrup. *The Design and Evolution of C++*. 1st ed. Addison-Wesley Professional, Apr. 1994. ISBN: 9780201543308 (cit. on p. 11).

# List of Figures

# List of Tables

# List of Listings

2.1 FizzBuzz in Pyhon 3.4 . . . . . 7
2.2 Go concurrency example . . . . . 9
2.3 Rust example . . . . . 11

# Appendices

# A. Appendix

- System configuration -> clang/gcc version -> rustc and cargo version (possibly including commit hash!) -> go version and implmentation (cgo?)

# Erklärung

Ich versichere, dass ich die Arbeit selbstständig verfasst und keine anderen, als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internetquellen – benutzt habe, die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

**Optional:** Ich bin mit der Einstellung der Bachelor-Arbeit in den Bestand der Bibliothek des Fachbereichs Informatik einverstanden.

Hamburg, den 26.11.2014 .....