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, April 11, 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

This chapter provides some background information to high-performance computing (HPC). 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 HPC 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 or the underlying machine. 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 (TDD)* 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. [maib] (// 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 HPC.

1.2. Goals of this thesis

This thesis aims to evaluate Rust and Go as potential programming languages in the HPC environment. The comparison is based on a port of an existing parallel application orginally written in Python. It was rewritten in 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 section 2.1. 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. Programming paradigms in Fortran and C

As stated in section 1.1 high-performance computing is largely dominated by C and Fortran and although their trademark is mostly performance these two languages achieve this in very different ways. Unfortunately both approaches are not completely safisfying and could be improved.

Fortran is the traditional choice for HPC applications and it's typesystem is very much accustomed to that area. As the name suggests [1] it was originally developed to allow for easy computation of mathematical formulae on computers. In spite of Fortran beeing one of the oldest programming languages it is actually fairly high-level. It provides functions for many common mathematical operations such as matrix multiplication or trigonometric functions and a builtin datatype for complex numbers. In addition memory management is nearly nonexistent. In earlier versions of Fortran it was not possible to explicitly allocate data and even in programs writtine in newer revisions of the language allocation and memory sharing often only account for a small fraction of the source code.

While this high-level paradigm of scientific programming is certainly well suited for a lot of applications, especially for scientists with mathematical backgrounds, it can also be insufficient in some edge cases. Notably in performance critical sections the functions sometimes are just not good enough and the programmer has to fall back to manual solutions or external libraries. Because Fortran does not offer fine grained control over memory or other resources some algorithms cannot be fully optimized which can limit performance. Of course this is not the general case and normally the compiler can

[1] FORTRAN is an acronym for FORmula TRANslation

generate efficient code but in machine dependant regions like caches or loop unrolling Fortran simply does not give the programmer enough control to finetune every last bit.

C on the other hand approaches performance totally different. Developed as a general purpose language it provides the tools to build efficient mathematical functions and datatypes which in turn require a lot more micromanagent than their equivalents from Fortran.

// remove or rephrase this (maybe as the last aragraph of this section?) 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 HPC. This section aims to provide a rough overview of all 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 programming language. [Guo14; Lub14] This allows the students to focus on core concepts of coding and algorithms instead of distracting 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[2] which offers efficient implementations for multidimensional arrays and common numeric algorithms like Fourier transforms or MPI4Py[3], an 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 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 HPC 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 HPC area.

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

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 Bogdan/Björn's Erlang Abstract Machine (BEAM) virtual machine. Today it is most often used in soft real-time computing [4] because of it's error tolerance, hot code reload capabilities and lock-free concurrency support. [CT09]

Erlang has a vary unique and specialized syntax which is very different from C-like languages. It abstains from using any kind of parentheses as block delimiters and instead uses a mix of periods, semicolons, commas and arrows (`->`). Unfortunately the rules for applying these symbols are not very intuitive and may even seems random for newcomers at times.

One core concept of Erlang is the idea of processes. These lightweight primitives of the language are provided by the virtual machine are neither direct mappings of operating system threads nor processes. One the one hand they are cheap to create and destruct (like threads) but do not share any address space or other state (like processes). Because of this the only way to communicate is through message passing which can be handled via the `receive` keyword and send via the `!` operator. [Arm03; CT09]

```
%% Module example (this must match the filename - '.erl')
-module(example).
%% This module exports two functions: start and codeswitch
%% The number after each function represents the param count
-export([start/0, codeswitch/1]).

start() -> loop(0).

loop(Sum) ->
  % Match on first received message in process mailbox
  receive
    {increment, Count} ->
      loop(Sum+Count);
    {counter, Pid} ->
      % Send current value of Sum to PID
      Pid ! {counter, Sum},
      loop(Sum);
    code_switch ->
      % Explicitly use the latest version of the function
      % => hot code reload
      ?MODULE:codeswitch(Sum)
```

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

```
22 end.
23
24 codeswitch(Sum) -> loop(Sum).
```

Listing 2.2: Erlang example

Conceptionally Erlang offers various constructs known from functional languages like pattern matching, clause based function definition and immutable variables but the language as a whole is not purely functional. Rather each erlang process in itself (ideally) behaves pure (meaning the result of a function depends solely on its input) while the collection of processes interacting which each other through messages of course contain state and side effects.

Erlang was considered as a possible candidate for HPC because of its concurrency capabilities. The fact that processes are a core part of the language and are rather cheap in both creation and destruction seems ideal for high performance applications often demanding enormous amounts of parallelism. Sadly Erlang suffers from what one might call over specialization. The well adapted type system makes it very suited for tasks where concurrency is essential like serverside request management, task scheduling and other services with high connection fluctuation, but "The ease of concurrency doesn't make up for the difficulty in interfacing with other languages." [Dow11] Even advocates of Erlang say they would not use it for regular business logic. In HPC most of the processing time is spent solving numeric problems. These are of course parallelized to increase effectiveness but the concurrency aspect is often not really inherent to the problem itself. Because of this Erlang's concurrency capabilities just do not outweigh it's numeric slowness for traditional HPC problems. [Héb13]

Go

Go is a relatively young 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 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 which are used for synchronization purposes. They provide a way to communicate with running goroutines via message passing.

```go
package main

import "fmt"

// Number of gorountines to start
const GOROUTINES = 4

func helloWorldConcurrent() {
    // Create a channel to track completion
    c := make(chan int)

    for i := 0; i < GOROUTINES; i++ {
        // Start a goroutine
        go func(nr int) {
            fmt.Printf("Hello from routine %v", nr)
            // Signalize completion via channel
            c <- 1
        }(i)
    }

    for i := 0; i < GOROUTINES; i++ {
        // Wait for completion of all goroutines
        <-c
    }
}
```

Listing 2.3: 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 types or functions 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.

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[5] as backend and is compatible to common tools like **The GNU Project Debugger** (*gdb*)[6] 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 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

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

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. //wording

```
// 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 2.4: 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 coding 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[7] so concepts which did not work out can be quickly changed or completely dropped. But Rust's immatureness is also its greatest weakness. The

[7] current version beeing 1.0.0-ALPHA-2 at the time of this writing

Comparison

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

2.3. Related work

The search for new programming languages which are fit for HPC is not a recently developing trend. There have been multiple studies and evaluations but so far none of the proposed languages have gained enough traction to receive widespread adoption. Also most reports focused on the execution performance without really considering additional software metrics or developer productivity. [Nan+13] at least adds lines of code and development time to the equation but both of these metrics only allow for superficial conclusions about the code quality.

From the here presented candidates Go in particular has been compared to other approaches to parallel programming with mixed results. Although its regular execution speed is somewhat lacking [Mit14] showed the highest speedup from parallelization amongst the evaluated languages.

// needs more work

// really mention this here without ? Rust on the other hand has not been seriously evaluated in the HPC environment mostly due to its

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 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 Extensible Markup Language (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 format is based on Google's *Protocol Buffers* [4] and is superior to the XML variant in both size and speed [Proc]. 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 a variant of Diskstra's algorithm was implemented in all three languages in a benchmarkable way so it can be compared separately. The concept is based on the Dijkstra-NoDec algorithm as seen in [Che+07, p. 16] This is done to rule out possible errors in the complete streets4x implementations and be able to directly compare the algorithmic part of the application.

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 Message Passing Interface (MPI) implementation installed on the system. If no supported implementation is found it falls back to a pure Python solution although the native one should be preferred for 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

[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/`

streets4c will use OpenMP [5] as it is the de facto standard for simple thread parallelization in C.

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

3.3.0. [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.

3.3.1. 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 candidates both in speed as well as memory consumption.

3.3.2. Building a basic graph representation for the simulation

The next goal was to conceptionally build the graph and related structures the simulation would later operate on. This involved thinking about the relation between edges and nodes as well as the choice of various containers to store the objects efficiently while also keeping access simple. This milestone tested the language's standard libraries and ability to express functions in a compact way (lambda..)

[5] `http://www.openmp.org`

3.3.3. Verifying structure and algorithm

// wording?!

After the base structure to represent graphs and calculate shortest paths was in place it was time to validate the implementations. Unfortunately the OSM data used in the first phase 3.3.1 contained too much nodes and ways to be able to efficiently verify any computed results. Therefore a small example graph was manually populated and fed to the algorithm. However unlike the final application the start and end nodes were fixed to allow for comparable results.

3.3.4. Benchmarking graph performance

The fourth milestone was preliminary benchmark of the implementations. The basic idea was to parse the data used in phase one and build the representing graph. After that the shortest path algorithm is executed once for each node. The total execution time as well as the time taken for each step (building the graph and calculating shortest paths) should be measured and compared as well as the usual memory statistics from previous phases.

4. Implementation

This chapter describes the implementation process for all three compared languages. It is divided in sections based on the development milestones defined in the previous chapter 3.3.

4.0. Project setup

As all applications written for this thesis have been developed on Linux the setup instructions are for this operating system. They **should** work on *nix as well but there is no definite guarantee this is the case. Also each section assumes the toolchains for the various languages are installed as this is largely different based on what operating system and on Linux which distribution is used. It is therefore not covered in this thesis.

4.0.1. C

//TODO: Add footnotes/links

The buildtool for streets4C is GNU *make* with a simple handcrafted **`Makefile`**. It was chosen to strike a balance between full blown build systems like *Autotools* or *CMake* and manual compilation. The setup steps required for this configuration are relatively straight forward and listed below.

```
$ mkdir -p streets4c
$ cd streets4c
$ vim main.c
$ vim Makefile
$ make && ./streets4c
```

Listing 4.1: Project setup: streets4C

After creatig a new directory for the application a **`Makefile`** and a sourcefile are created. **`main.c`** contains just a bare bones main method while the **`Makefile`** uses basic rules to compile an executable named streets4c with various optimization flags.

All in all the setup in C is quite painless although manual. The only caveat are Makefiles. They may be simple for small projects without real dependencies but as soon as different

source and object files are involved in the compilation process they can get quite confusing. At that point the mentioned build systems might prove their worth in generating the **Makefile**(s) from other configuration files.

4.0.2. Go

//TODO: Add footnotes/links

For Go the choice of buildtool is nonexistent. The language provides the **go** executable which is responsible for nearly the complete development cycle. It can compile your code, install arbitrary Go packages from various sources, run tests and format code just to name the most common features.

This makes Go (the language) extremely convenient since everything you want to do is probably one commandline away. For example to get a dependency one would invoke the tool like so: `go get github.com/petar/GoLLRB/llrb`
This will download the package in source form which can then be imported in any project on that machine via its fully qualified package name.

To achieve this convenience the **go** tool requires some setup work before it can be used for the first time. Because of this this section contains two setup examples.

```
$ mkdir -p streets4go
$ cd streets4go
$ vim main.go
$ go run main.go
```

Listing 4.2: Project setup: streets4Go

Listing 4.2 describes the steps that were taken to create the streets4go project inside the thesis' repository. It is pretty similiar to the C version. A directory gets creates then a source file containing a **main** function is created which can be build and run with a single command.

One thing to remind here is the fact, that this code does not live inside a globally set drectory called **GOPATH**. To be able to download packages only once **go** assumes an environment variable called **GOPATH** to be set to a directory which it has full control over. This directory contains all source files as well a the compiled binaries all stored through a consistent naming scheme. Normally it is assumed that all Go project live inside their own subdirectories of the **GOPATH** but it is possible to avoid this at the cost of some convenience.

The project that was created through the commands of listing 4.2 for example cannot be installed to the system by running `go install` since it does not reside in the correct folder instead one has to copy the compiled binary to a directory in **PATH** manually.

The next listing shows a more realistic workflow for creating a new Go project from scratch without any prior setup required. It assumes one starts in the directory that should be set as **GOPATH** and assumes `www.github.com` as code host which in reality just determines the package name. It is also important to add the export shown in the first line to any inititalization file of your shell or operating system to ensure it always beeing available.

```
$ export GOPATH=$(pwd)
$ mkdir -p src/github.com/<user>/<project>
$ cd src/github.com/<user>/<project>
$ vim main.go
$ go run main.go
```

Listing 4.3: Full setup for new Go projects

4.0.3. Rust

//TODO: Add footnotes/links

Similar to Go also Rust provides its own build system. As mentioned in the candidate introduction Rust installs its own package manager *cargo*. It functions as build system and is also capable of creating new projects. This shortens the setup process considerably as observable in the next listing.

```
$ cargo new --bin streets4rust
$ cd streets4rust
$ cargo run
```

Listing 4.4: Project setup: streets4Rust

With the **new** subcommand a new project gets created. The **--bin** flag tells **cargo** to create an executable project instead of a library which is the default. Thanks to the one command all the initial files and directories are created with one single command. This includes:

- the project directory itself (named like the given project name)
- a **src** directory for source files
- a **target** directory for build results
- a required manifest file named **Cargo.toml** including the given project name
- a sample file inside **src** which is either called **main.rs** for binaries or **lib.rs** for libraries containing some sample code

- and optionally an empty initialized version control repository (**git** or **mercurial** if the corresponding command line option has been passed)

The resulting application is already runnable via `cargo run`[1] and produces some output in **stdout**. This process is extremely convenient and error proof since **cargo** validates all input before executing any task. The **man** pages and help texts are quite thin at the moment but as with everything in the Rust world **cargo** is still beeing developed.

The overall greated advantage however is that the Rust process does not involve any manual text editing. What might sound trivial at first, is actually quite important for newcomers to the language. You do not have to know any syntax to get started with Rust since the generated code already compiles and does something interesting. In the other languages you have to write a valid, minimal program manually to even test the project setup while Rust is ready to go after just one command.

Of course this strategy is not without limitations. To be able to use **cargo** all files and directories have to follow a special pattern. Although the chosen conventions are somewhat common one cannot use arbitrary directory and file names.

4.0.4. Comparison

For newcomers Rust definitely provides the best experience. One can get a valid *Hello world!* application without any prior knowledge which lowers the barrier of entry dramatically. In addition Rust does not require any presetup before the first project. just install the language toolchain (either through the operating system's package manager or the very simple setup script [2]) and start coding.

Go required some intial setup besides the installation but it still quite easy to setup. The **GOPATH** exporting is a small annoyance but it balances out with the benefits the developer gets later down the line like easy dependency management. The syntax is very concise so creating a new source file with a **main** function is still quite fast.

Considering C's long lifespan the tooling for project setup is not very good. Full blown IDEs like Eclipse provide wizards to create all required files but for free standing development with a simple text editor and GNU *make* there is no real automation possible. Naturally it is not hard to create an empty C source file however in this day and age it should not be necessary to manually have to adjust the order of linker flags in the **Makefile** because of obscure warnings in the compilation process.

This probably does not apply to seasoned C developers and one could make the argument that it is inherent to the language's "closeness to the metal". But acknowledging the fact that scientists more often than not see programming as an unwanted necessity to be able to complete their research it is questionable whether this technical know-how should really be required to use a language like C.

//TODO: maybe to wordy?

[1] which is executable anywhere inside the project directory
[2] `https://static.rust-lang.org/rustup.sh`

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

	C	Go	Rust
SLOC	55	55	36
Development time (min)	51:18	21:16	33:09
Execution time (sec)	1.017 (-O0)	4.846 (GOMAXPROCS=1)	27.749 (-O0)
	0.994 (-O3)	1.381 (GOMAXPROCS=8)	2,722 (-O3)
Allocation count	2,390,566	11,164,068[3]	11,373,558
Free count	2,390,566	11,000,199[4]	11,373,557[5]

Table 4.1.: Milestone 1: Counting nodes, ways and relations

4.1.1. C

Preface: For the first real milestone *streets4C* had an important disadvantage. There was no library to conveniently process OpenStreetMap data. Therefore a small abstraction over the offical Protobuf definitions had to be written. The development time for this code located in **`osmpbfreader.c/h`** was not counted towards the total time of the phase to avoid unfair bias just because of a missing library and similarly the source lines of code (SLOC) count does not include this code.

The first phase of development already highlighted many of the common problems encountered when programming in the C language. After finishing the aforementioned library it had to be invluded in the development process which in return meant the **`Makefile`** had to be extended to also compile **`osmpbfreader.c`** and include the resulting object file in assembling the executable binary. This proved harder than expected which can partly be attributed to my lacking expertise with the C compilation process but also confirms the unneeded complexity of such a simple task. In the end the problem was the order in which the sorce files and libraries where passed to the compiler. The libraries were included too early which resulted in "undefined reference to method" error messages. In times where compilers are smart enough to basically rewrite and change code for performance reasons it is completely inexcusable that the order of source arguments to process is still that relevant. The time spent solving these complication errors shows in the statistics for C which is considerably larger than its competitors in this stage.

The other big caveat in working with data was the manual memory management. Since said data is stored in effectively compressed manner in the file additional heap allocations

[3] The memory statistics for Go have not been acquired by valgrind but by **`runtime.MemStats`** this and the fact that Go is garbage collected explain the discrepancy in allocations and frees

[4] See footnote 3 //todo: verify footnote nr in final draft

[5] This is due to a bug in the osmpbf library used. In safe Rust code it is impossible to leak memory

were unavoidable in accessing it. This requires either explicit freeing by the caller or a symmetric deallocation function provided by the library. In the case of Protobuf it is even worse since a client cannot just perform the usual `free(..)` call but has to use the custom free functions generated from the source **`.proto`** format description files. For some intermediate allocations it is possible to limit this to the body of a libaray function but on the real data it shifts additional responsibilities on the caller.

```
/* somewhere in a function */
osmpbf_reader_t *reader = osmpbf_init(<some_path>);

OSMPBF__PrimitiveBlock *pb;
while((pb = get_next_primitive(reader)) != NULL)
{
    for (size_t i = 0; i < pb->n_primitivegroup; i++)
    {
        // access data on the primitive groups
        OSMPBF__PrimitiveGroup *pg = pb->primitivegroup[i];

        /* no need to free pg here since its part
         * of the primitive block pb */
    }

    // cannot use free(pb) here because of Protobuf
    osmpbf__primitive_block__free_unpacked(pb, NULL);
}

// regular free function provided by library
osmpbf_free(reader);
/* remaining part of the function */
```

Listing 4.5: Manual memory management with Protobuf in C

Considering this fact the SLOC count is still decent. With the help of a clever library interface the overhead for the memory management is comparatively small and the data is even loopable by a **`while`** loop which allows for convenient access and conversion. Also the statistics clearly show why C is still that dominant in the HPC area. With low allocation counts [6] and superior singlethreaded(!) performance C is the clear winner in the performance area for this first milestone.

[6] Although these are also caused by the simplicity of the custom osmpbfreader abstraction

4.1.2. Go

To parse the .osm.pbf files *streets4Go* uses an existing library simply called **osmpbf** [7]. The library follows common Go "best practices" which makes it easy to use. Internally goroutines are used to decode data in parallel which can then be retrieved through a **Decoder** struct. The naming of the struct and the corresponding methods follow the conventions of the official Decoder types of the Go standard library. This adherence to conventions directly shows in the development time which is the shortes amongst the candidates for this first phase.

```
1  package main
2
3  import (
4      "fmt"
5      "io"
6      "log"
7      "os"
8      "runtime"
9
10     "github.com/qedus/osmpbf" // <- add the import
11 )
12
13 func main() {
14     /* .. */
15     decoder := osmpbf.NewDecoder(someFile)  // <- use some
           ↪ type or function from the package
16     /* .. */
17 }
```

Listing 4.6: Dependency management in Go

Dependency management was very easy and intuitive. As mentioned in the candidate introduction `go get` was used to download the library and a simple import statement was enough to pull in the necessary code (see Listing 4.6). One caveat here is once again Go stricts compilation rules. Since an unused import is a compiler error an/my (//wording?) editor plugin kept deleting the prematurely inserted import statement as part of the saving process. While the auto fix style of tools like **gofmt** and **goimports** is certainly helpful for fixing common formatting errors, the loss of control for the developer takes some time to get used to. (//really include that?)

Another interesting recorded statistic is the count of source lines of code. This count exposes one of the criticisms commonly directed at Go - verbose error handling. Although the code is semantically simpler (no manual memory management, higher level

[7] `https://github.com/qedus/osmpbf`

language constructs) the SLOC count is in fact identical to that of *streets4C*. This is the result of the common four line idiom to handle errors. A function that could fail typically returns two values. The desired result and an error value. If the function failed to execute successfully the error value will indicate the source of the failed execution. Otherwise this value will be **nil** signalling a failure free completion. This pattern is used three times in this simple first phase alone which results in 12 lines.

```
func SomeIOFunction(path string) {
    file, err := os.Open(path)
    if err != nil {
        log.Fatal(err)  // os.Open returned an error
    }
    err = pkg.SomeIOFunc(file)
    if err != nil {
        log.Fatal(err) // rinse and repeat
    }
}
```

Listing 4.7: Idiomatic error handling in Go

Considering the aforementioned simplicity *streets4Go*'s performance characteristics are very promising. Although in its basic form about four to five times slower than the C solution the parallelized version achieves similar performance. This version was only included since the library was already based on a variable number of goroutines which made the parallelzation a matter of changing an environment variable in the Go runtime. While this change required only the addition of a single line, the C abstraction **osmpbfreader** might not even be parallelizable without considerable changes to its architecture. This truly shows the power of language level parallelization mechanics and confirms the choice of Go as a candidate in this evaluation.

4.1.3. Rust

streets4Rust also had the advantage of an existing library to use for decoding which is called *osmpbfreader-rs* [8]. Similar to Go the dependency management was extremely convenient and simple. The only changes necessary were an added line in the Cargo manifest (Cargo.toml) and an **extern crate osmpbfreader;** in the crate root **main.rs**. After that `cargo build` downloaded the dependency (which in this case meant cloning the git repository) and integrated it into the compilation process.

Compared to C and Go *streets4Rust* required a medium amount of development time and had the lowest SLOC count in this phase. This can mainly be attributed to the library's use of common Rust idioms and structures like **iterators** and **enums**. Unlike

[8] `https://github.com/texitoi/osmpbfreader-rs`

C enums, which are basically named integer constants, the Rust variant provides a lot more features like beeing useable in pattern matching expressions. The next listing shows the complete decode part of this stage which is very compact and easy to understand. //add implementation details of the lib? (chained interators)

```
/* in main() */
for block in pbf_reader.primitive_blocks().map(|b|
  ↪ b.unwrap()) {
    for obj in blocks::iter(&block) {
        match obj {
            objects::OsmObj::Node(_)        => nodes += 1,
            objects::OsmObj::Way(_)         => ways += 1,
            objects::OsmObj::Relation(_)    => rels += 1
        }
    }
}
/* remaining part of main() */
```

Listing 4.8: decoding in Rust

The function `blocks::iter` returns an enum value which gets pattern matched on to determine which counter should get incremented. While this example does not actually use any fields of the objects it would be a simple change to destructure the enum values and retrieve the structures containing the data.

The execution time highlights another important factor in regards to Rust's matureness as a language. The optimized version is more than ten times faster then the binary produced by default options. This is mostly due to the fact that the Rust LLVM frontend produces bloated byte code which does not get optimized on regular builds. That is also the reason release builds take substanstially longer. It simply takes more time to optimize (and therefore often shrink) LLVM Intermediate Representation (LLVM IR) instead of emitting less code in the first place. Although the code generation gets improved steadily it is not a big focus until version `1.0` is released but the Rust core team knows about the issue and it is a high priority after said release.

Nontheless the release build shows the power of LLVM's various optimization passes. *streets4Rust* achieves the second best single threaded performance after C with a run time of 2.72 seconds which is impressive considering the vastly shorter development time and lowest SLOC count across all candidates.

4.1.4. Comparison

4.2. Building a basic graph representation for the simulation

The second milestone was to develop a graph structure to represent the street network in memory. Like *streets4MPI* random nodes from this data would then be fed to Dijkstra's SSP algorithm to simulate trips. Since all applications should be parallelized later on the immutable data (such as the edge lengths, IDs and adjecency lists) needed to be stored separately from the changing data the algorithm required (such as distance and parent arrays). To achieve this all implementations have a **`graph`** structure holding the immutable data and a **`dijkstagraph`** strcuture to store volatile data alongside some kind of reference (or pointer) to a graph object.

Since this milestone included a preliminary implementation of the actual algorithm it required the use of a priority queue which tests the languages libraries. //TODO: remove this?

	C	Go	Rust
SLOC (total)	385	196	170
Development time (hours)	02:30:32	01:06:06	01:14:28

Table 4.2.: Milestone 2: Building a basic graph representation

4.2.1. C

As seen in Table 4.2 this stage resulted in a much higher SLOC count for C. This is due to the fact that development took place in another source files. To encapsulate graph functionality properly a new file called **`graph.c`** was created. Following established conventions this meant also creating a matching header (**`graph.h`**) to be able to use the code in the main application. While this separation is decently useful to not have to clobber you source file with "uninteresting" structure definitions it also introduces a fair bit of redundancy. Functions are declared in the header and implemented in the source files which means the signature appears twice. In addition C had the unfortunate problem of not having a proper implementation of a priority queue easily available which required the addition of another source file / header combination (**`util.c/h`**). This increased the SLOC count even further and added some additional development time as well.

Implementation wise the graph representation was very straight forward. Similar to the mathematical representation a **`graph`** consists of an array of **`node`**s and **`edge`**s. To be able to map from IDs to array indices two hash tables were added which map from

long to int. The **dgraph** structure can be created with a pointer to an existing **graph** and is then able to execute Dijkstra's SSP algorithm.

```c
struct node_t
{
    long osm_id;
    double lon, lat;

    GHashTable *adj; // == adjecent edges/nodes
};

struct edge_t
{
    long osm_id;
    int length;
    int max_speed;
    int driving_time; // == edge weight
};

struct graph_t
{
    int n_nodes, n_edges;
    node *nodes;
    edge *edges;

    GHashTable *node_idx;
    GHashTable *edge_idx;
};

struct dgraph_t
{
    graph g;
    pqueue pq;

    int cur; // == index of current node to explore
    int *dist;
    int *parents;
};
```

Listing 4.9: Graph representation in C

//TODO: add paragraph about the choice of glib

All structures contain the expected data. Only the **cur** field in **dgraph** seems a little inconvenient. It needed to be added since **glib**'s **GHashTable** only support operations

on all key-value-pairs via a function pointer with a single extra argument. Since the algorithm at on particular required access to the currently explored node's index as well as the distance and parent arrays the index needed to be stored in the struct itself.

While the additional field was a minor inconvenience other problematic aspects were the verbose access and missing typesafety of the used **GHashTable**s. Since C naturally is not typesafe and does not allow for true generic programming via type parameters nearly all generic code is written using `void*`

4.2.2. Go

```
type Node struct {
    osmID    int64
    lon, lat float64

    adj map[int]int    // == adjecent edges/nodes
}

type Edge struct {
    osmID       int64
    length      int
    drivingTime uint   // == edge weight
    maxSpeed    uint8
}

type Graph struct {
    nodes []Node
    edges []Edge

    nodeIdx, edgeIdx map[int64]int
}

type DijkstraGraph struct {
    g *Graph
    pq PriorityQueue

    dist    []uint
    parents []int
}
```

Listing 4.10: Graph representation in Go

4.2.3. Rust

```rust
pub struct Node {
    pub osm_id: i64,
    pub lon: f64,
    pub lat: f64,

    pub adj: HashMap<usize, usize> // == adjecent edges/nodes
}

pub struct Edge {
    pub osm_id: i64,
    pub length: u32,
    pub max_speed: u8,
    pub driving_time: u32 // == edge weight
}

pub struct Graph {
    pub nodes: Vec<Node>,
    pub edges: Vec<Edge>,

    pub nodes_idx: HashMap<i64, usize>,
    pub edges_idx: HashMap<i64, usize>
}

pub struct DijkstraGraph<'a> {
    pub graph: &'a Graph,
    pub queue: BinaryHeap<NodeState>,

    pub dist: Vec<u32>,
    pub parents: Vec<usize>
}
```

Listing 4.11: Graph representation in Rust

	C	Go	Rust
SLOC (total)	637	268	232
Development time (hours)	01:53:30	01:16:49	01:04:38
Execution time (seconds)	0.004 (-O0)	0.296	0.007 (-O0)
	0.003 (-O3)		0.005 (-O3)
Allocation count	108	519[9]	47
Free count	106[10]	169[11]	47
Allocation amount (bytes)	7,868[12]	53,016	22,792

Table 4.3.: Milestone 3: Verifying the implementation

4.2.4. Comparison

4.3. Verifying structure and algorithm

4.3.1. C

4.3.2. Go

4.3.3. Rust

4.3.4. Comparison

[9] See footnote 3

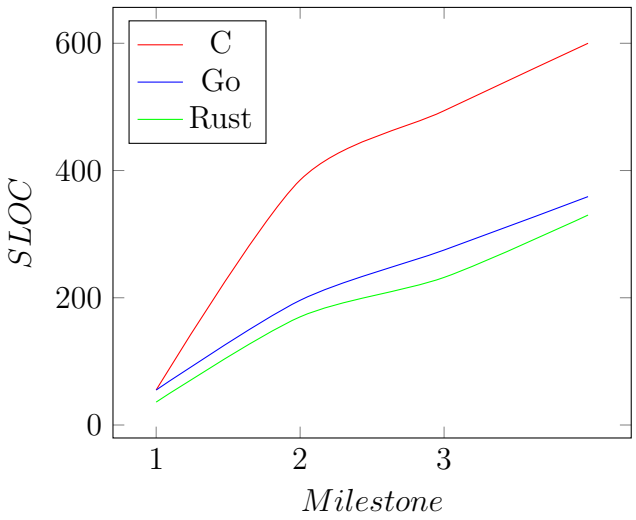
[10] Due to the use of GLib some global state remains reachable after exiting. This is likely intended behaviour and not a memory leak.
add link?

[11] See footnote 3

[12] 2,036 bytes were in use at exit see footnote xx

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

[Arm03] Joe Armstrong. "Making reliable distributed systems in the presence of sodware errors". dissertation. Stockholm, Sweden: The Royal Institute of Technology, Department of Microelectronics and Information Technology, Dec. 2003. URL: http://www.erlang.org/download/armstrong_thesis_2003.pdf (visited on 13. 03. 2015) (cit. on p. 9).

[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 (cit. on p. 16).

[con] OpenStreetMap contributors. *OpenStreeMap - About*. URL: https://www.openstreetmap.org/about (visited on 06. 04. 2015).

[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 (cit. on p. 9).

[Dow11] Malcolm Dowse. *Erlang and First-Person Shooters*. June 2011. URL: http://www.erlang-factory.com/upload/presentations/395/ErlangandFirstPersonShooters.pdf (visited on 13. 03. 2015) (cit. on p. 10).

[Dox12] Caleb Doxsey. *An Introduction to Programming in Go*. Lexington, KY: CreateSpace Independent Publishing Platform, Sept. 2012. ISBN: 9781478355823 (cit. on p. 10).

[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. 11).

[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. 15, 16).

[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).

[Héb13] Fred Hébert. *Learn you some Erlang for great good! a beginner's guide*. San Francisco: No Starch Press, 2013. ISBN: 9781593274351. URL: http://learnyousomeerlang.com/content (cit. on p. 10).

[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).

[maia] The Go project maintainers. *The Go Programming Language - Effective Go*. URL: https://golang.org/doc/effective_go.html (visited on 06. 04. 2015) (cit. on p. 42).

[maib] 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).

[Mit14] Sparsh Mittal. "A Study of Successive Over-relaxation Method Parallelisation over Modern HPC Languages". In: *International Journal of High Performance Computing and Networking* 7.4 (June 2014), pp. 292–298. ISSN: 1740-0562. DOI: 10.1504/IJHPCN.2014.062731 (cit. on p. 14).

[MS] ANL Mathematics and Computer Science. *The Message Passing Interface (MPI) standard*. URL: http://www.mcs.anl.gov/research/projects/mpi/ (visited on 06. 04. 2015) (cit. on p. 42).

[Nan+13] Sebastian Nanz et al. "Benchmarking Usability and Performance of Multicore Languages". In: *Empirical Software Engineering and Measurement, 2013 ACM / IEEE International Symposium on*. Oct. 2013, pp. 183–192. DOI: 10.1109/ESEM.2013.10 (cit. on p. 14).

[Proa] LLVM Project. *LLVM Language Reference Manual*. URL: http://llvm.org/docs/LangRef.html (visited on 06. 04. 2015) (cit. on p. 42).

[Prob] LLVM Project. *The LLVM Compiler Infrastructure - Overview*. URL: http://llvm.org (visited on 06. 04. 2015) (cit. on p. 42).

[Proc] The OpenStreetMap Project. *OpenStreeMap Wiki - "PBF Format"*. URL: http://wiki.openstreetmap.org/wiki/PBF_Format (visited on 18. 02. 2015) (cit. on p. 16).

[SKP06] Sayantan Sur, Matthew J. Koop, and Dhabaleswar K. Panda. "High-performance and Scalable MPI over InfiniBand with Reduced Memory Usage: An In-depth Performance Analysis". In: *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. SC '06. Tampa, Florida: ACM, 2006. ISBN: 0-7695-2700-0. DOI: 10.1145/1188455.1188565. URL: http://doi.acm.org/10.1145/1188455.1188565 (cit. on p. 42).

[Str94] Bjarne Stroustrup. *The Design and Evolution of C++*. 1st ed. Addison-Wesley Professional, Apr. 1994. ISBN: 9780201543308 (cit. on p. 12).

List of Figures

List of Tables

List of Listings

Appendices

A. Glossary

BEAM

Bogdan/Björn's Erlang Abstract Machine

Bogdan/Björn's Erlang Abstract Machine

The virtual machine which runs Erlang. It loads bytecode which is converted directly to threaded native code and executed.

goroutine

A lightweigth concurrently executing function which gets multiplexed into OS threads by the Go runtime [maia]

HPC

high-performance computing

LLVM

The LLVM Compiler Infrastructure Project (formerly short for Low Level Virtual Machine) is an umbrella project for various compiler and other low-level tools. LLVM Core is the primary subproject and a set of libraries for code generation and optimization for various platforms. [Prob]

LLVM Intermediate Representation

A low level programming language similar to assembly. It is the code representation LLVM uses in its Core libraries. LLVM IR is platform-agnostic with the "capability of representing 'all' high-level languages cleanly." [Proa]

LLVM IR

LLVM Intermediate Representation

Message Passing Interface

The Message Passing Interface standard is a library specification developed by a committee of vendors, implementors and users. It is the current cominant model used in high-performance computing and implementations for many platforms (both commercial and free) are available including bindings for various programming languages. [MS; SKP06]

MPI

Message Passing Interface

SLOC

source lines of code

TDD

test-driven development

XML

Extensible Markup Language

B. System configuration

```
Linux florian-arch 3.19.3-3-ARCH #1 SMP PREEMPT Wed Apr 8
   ↪ 14:10:00 CEST 2015 x86_64 GNU/Linux
```

Listing B.1: Output of `uname -a`

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                8
On-line CPU(s) list:   0-7
Thread(s) per core:    2
Core(s) per socket:    4
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 42
Model name:            Intel(R) Core(TM) i7-2630QM CPU @
   ↪ 2.00GHz
Stepping:              7
CPU MHz:               810.625
CPU max MHz:           2900,0000
CPU min MHz:           800,0000
BogoMIPS:              3992.48
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              6144K
NUMA node0 CPU(s):     0-7
```

Listing B.2: Output of `lscpu`

C. Software versions

```
gcc (GCC) 4.9.2 20150304 (prerelease)
Copyright (C) 2014 Free Software Foundation, Inc.
This is free software; see the source for copying
   ↪ conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A
   ↪ PARTICULAR PURPOSE.
```

Listing C.1: Output of `gcc --version`

```
go version go1.4.2 linux/amd64
```

Listing C.2: Output of `go version`

```
rustc 1.0.0-nightly (93f7fe32d 2015-04-10) (built
   ↪ 2015-04-11)
```

Listing C.3: Output of `rustc --version`

```
cargo 0.0.1-pre-nightly (72c2e3d 2015-04-10) (built
   ↪ 2015-04-10)
```

Listing C.4: Output of `cargo --version`

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 .....