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

— Bachelor Thesis —

Research group 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 25, 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 field, 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 dependent 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. In contrast Fortran does not require the same amount of technical knowledge but also limits the programmer in fine grained resource control. Both approaches are not ideal and the situation could be improved by a language offering both control and high-level abstractions while keeping up with Fortran and C's execution performance.

Also considering the fact that scientific applications are often written by scientist without a strong background in computer 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 on and are not well integrated into the core language.

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 generally 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, perform global setup/teardown work and even do basic output verification [maib]. Most importantly 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 review 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 three implementations of a shortest path algorithm in the two language candidates as well as C. The idea is based on an existing parallel application called *streets4MPI* which was written in Python. It simulates ongoing traffic in a geographical area creating heatmaps as a result. The programs written for this thesis implement the computational intensive part which is the shortest path calculation to be able to review Go and Rust's performance characteristics as well as development productivity and tooling support. Since libraries for interprocess communication in Rust and Go are nowhere near production-ready this thesis will focus on shared memory parallelization only to avoid unfair bias based solely on the quality of the supporting library ecosystem.

To reduce complexity the implementations perform no real error handling nor produce any usable simulation output. They simply perform Dijkstra's algorithm in the most language ideomatic way which can optionally be parallelized. While raw performance will be the main criteria, additional software metrics will also be reviewed to measure the complexity and overall quality of the produced applications. In addition the general development experience especially for newcomers to the respective languages will be judged to be able to evaluate the usefulness for scientist not specialized in programming.

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. After that all considered language candidates are introduced and characterized.

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 its 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 being 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 written 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 generate efficient code but in machine dependent 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 differently. Developed as a general purpose language it provides the tools to build efficient mathematical functions and

[1] FORTRAN is an acronym for FORmula TRANslation

datatypes which in turn require a lot more micromanagment than their equivalents in Fortran. This allows the programmer to carefully tweak each operation to achieve maximum performance at the cost of high-level abstractions. Thus C is often the language of choice for computer scientists when performance is the main concern but it is rather ill-suited for people without broad knowledge about memory and other machine internals.

The main drawback of both languages is their age. Even though new revisions are regularly accepted Fortran and C strive to be backwards compatible for the most part. This has some very serious consequences especially in their respective syntaxes. A lot of features of newer standards are integrated suboptimally to preserve backwards compatability. Newer languages can take advantage of all past research without having to adhere to outdated idioms and patterns.

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. The following Listing demonstrates just a few of Python's core features which make it a great first programming language to learn.

```
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 existence 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 as seen in [WFV14].

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 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 Python code often looks like its 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.

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

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

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 its error tolerance, hot code reload capabilities and lock-free concurrency support [CT09].

Erlang has a very 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 seem 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 and 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 sent via the `!` operator [Arm03; CT09].

```
1  %% Module example (this must match the filename - '.erl')
2  -module(example).
3  %% This module exports two functions: start and codeswitch
4  %% The number after each function represents the param count
5  -export([start/0, codeswitch/1]).
6
7  start() -> loop(0).
8
9  loop(Sum) ->
10   % Match on first received message in process mailbox
11   receive
12     {increment, Count} ->
13       loop(Sum+Count);
14     {counter, Pid} ->
15       % Send current value of Sum to PID
16       Pid ! {counter, Sum},
17       loop(Sum);
18     code_switch ->
19       % Explicitly use the latest version of the function
20       % => hot code reload
21       ?MODULE:codeswitch(Sum)
22 end.
23
24 codeswitch(Sum) -> loop(Sum).
```

Listing 2.2: Erlang example

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

Listing 2.2 illustrates some of these key features like code reloading and message passing. Further mode 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 purely (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 its numeric slowness for traditional HPC problems [Héb13].

Go

Go is a relatively young programming language which focuses 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 reaching reasone performance [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. The Listing below features a simple example writing multiple messages concurrently and using these channels to prevent premature exit of the parent thread.

```
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 2.3: 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 its 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 linked into every application. To prevent additional dependencies on target machines the language

designers chose to link all libraries statically including the runtime. 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 languages discussed here Rust is closest to C while attempting to fix common mistakes made possible by its loose standard allowing undefined behaviour. Memory safety is enforced through a very sophisticated model of ownership tracking. It is based on common concepts which are already employed on concurrent applications but integrates them on a language level and enforces them at compile time. The basic rule is that every resource in an application (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 (or at least minimal) 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 such as 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 and can be safely freed.

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

```rust
// 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 focuses on performance and safety it also adopted some functional concepts like *pattern matching* and the **`Option`** type as demonstrated in Listing 2.4. 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 targets safety without 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 being 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 language is still changing every day which means code written today might not compile tomorrow. However the breaking changes are getting less as the first stable release is scheduled to be issued on 2015-05-15. Rust 1.0.0 is guaranteed to be backwards compatible for all later versions so he language should soon be ready for production use. Meanwhile the toolchain is already quite impressive. In addition to the compiler the default installation also contains a package manager called **`cargo`**. It is able to fetch dependencies from git repositories or the central package repository located on `https://crates.io` and can build complex projects including linking to native C libraries. It is obviously still in development same as the language but the feature set is already very broad.

Rust was chosen to be evaluated further because it should be able to match C's execution speed while providing additional memory safety and modern language features. Even if the performance is not completely on par the productivity gains should still be substantial.

[7] The current version is `1.0.0-beta.2` at the time of this writing

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 candidates presented here Go in particular has been compared to traditional HPC languages with mixed results. Although its regular execution speed is somewhat lacking [Mit14] showed the highest speedup from parallelization amongst the evaluated languages which is very promising considering high concurrency scenarios like cluster computing. Rust on the other hand has not been seriously evaluated in the HPC context probably due to it still being developed.

3. Approach

The first section of the third chapter describes the existing application this evaluation is based on. In addition the various stages of the development process are roughly illustrated.

3.1. Overview: *streets4MPI*

As stated in section 1.2 the concept for the implementations to compare is inspired by *streets4MPI*, which was implemented to evaluate Python's usefulness for "computational intensive parallel applications" [FN12, p.3]. It was written by Julian Fietkau and Joachim Nitschke in scope of the module "Parallel Programming" in Spring 2012 and makes heavy use of the various libraries of the Python ecosystem.

streets4MPI parses OpenStreetMap (OSM) input data, builds a directed graph and repeatedly computes shortest paths for a set amount of "trips" (randomly chosen node pairs from the graph) in the graph. Over time it gradually modifies the graph based on results of previous iterations to emulate structural changes in the traffic network in the simulated area. The results can then optionally be written to a custom output format which is visualizable by an additional script [FN12].

As mentioned the applications written in scope of this thesis perform only the graph calculations discarding the produced results. Consequently the implementations do not produce any data besides the runtime statistics externally acquired by benchmark tools.

3.2. Differences to streets4MPI

Although the evaluated implementations are based on the original *streets4MPI*, there are some key implementational differences. This section gives a brief overview over the most important aspects that have been changed and describes both the original application's functionality as well as the derived implementations.

In the remaining part of the thesis the different applications will be referenced quite frequently. For brevity the language implementations to compare will be called by the following scheme: "streets4<language>". The Go version for example is called "streets4go".

Input format

The original *streets4MPI* uses the somewhat dated OSM Extensible Markup Language (XML) format[1] as input which is parsed by *imposm.parser*[2]. It then builds a directed graph via the *python-graph*[3] library to base the simulation on [FN12].

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

Simulation

The simulation in the base application is based on randomly picked node pairs from the source graph. For these trips the shortest path is calculated by Dijkstra's algorithm as seen in [Cor+09] and a random factor called "jam tolerance" is introduced to avoid oscillation in between iterations [FN12]. Then after some time has passed in the simulation, existing streets get expanded or shut down depending on their usage.

The compared implementations of this thesis also perform trip based simulation but without the added randomness and street modification. The concrete algorithm is a variant of the Dijkstra-NoDec algorithm as seen in [Che+07, p. 16]. It was mainly chosen because of its reduced complexity in required data structures which again reduces complexity and scope. The algorithm is implemented separately in all three languages so it could theoretically get benchmarked standalone to get clearer results. Mainly because of time constraints this was not attempted in this thesis.

Concurrency

streets4MPI 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 but 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 their foreign funtion interfaces. To achieve a fair comparison *streets4c*

[1] `http://wiki.openstreetmap.org/wiki/OSM_XML`
[2] `http://imposm.org/docs/imposm.parser/latest/`
[3] `https://code.google.com/p/python-graph/`

will use *OpenMP* [4] as it is the de facto standard for simple thread parallelization in C. Of course this solution might not match the performance of hand optimized implementations parallelized with the help of *pthreads* but since the focus is on simple concurrency in the context of scientific applications *OpenMP* was selected as the framework of choice.

3.3. Implementation process

The implementation process was performed iteratively. Certain milestones were defined and implemented in all three languages. The process only advanced 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 (or in any other arbitrary order), 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 various areas. While the main development and test runs were performed on a laptop the final application was run on a high performance machine provided by the research group Scientific Computing to compare scalability beyond common desktop level processors.

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 built 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.

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

3.3.2. Building a basic graph representation

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 expressiveness in terms of typed containers.

3.3.3. Verifying structure and algorithm

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 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.

3.3.4. Benchmarking graph performance

The fourth milestone was preliminary benchmark of the implementations. The basic idea was to parse the OSM 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.

3.3.5. Benchmarking parallel execution

The fifth and final stage consisted of modifying the existing benchmark to operate in parallel via threading and benchmarking the results for various configurations. While all the development and previous benchmarks were performed on a personal laptop the final benchmarks were taken on a computation node of the research group to gather relevant results in high concurrency situations.

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

All applications written for this thesis have been developed on Linux thus the setup instructions are for this operating system. They *should* work on *nix as well but there is no 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
source lines of code (SLOC)	163	55	36
Development time (hours)	00:51:18	00:21:16	00:33:09
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 Protocol Buffers 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 however the SLOC count includes the additional code since it was essential to this stage.

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 OSM data was the manual memory management. Since said data is stored in effectively compressed manner in the file additional heap

[3] The memory statistics for Go have not been acquired by valgrind but by **`runtime.MemStats`**. The fact that Go is garbage collected explains 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 very hard to leak memory (usually involving reference cycles or something similar).

allocations 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 Protocol Buffers 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 partially 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 OSM 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: OSM 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.

Nonetheless 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

The second milestone was to develop a graph structure to represent the street network in memory. Like in *streets4MPI* random nodes from this data would then be fed to Dijkstra's shortest path algorithm to simulate trips. Since all applications should be parallelized later on the immutable data (such as the edge lengths, OSM 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 **`dijkstragraph`** structure to store volatile data for the algorithm 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 was not directly available in all languages. Considering this fact the third milestone already highlighted some differences in comprehensiveness of the different standard libraries.

	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

// TODO: shorten this? other sections are shorter

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 newly written code in the main application. While this separation is decently useful to not have to clobber you source file with 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.

At this point it became clear the C version would not be created dependency free. Advanced data structures such as hash tables or growing arrays are essential when

properly modelling a graph and the choice was made to use the popular *GLib*[9] to provide these types. It is a commonly used library containing data structures, threading routines, conversion functions or macros and much more. Since both Rust and Go's standard library are mch more comprehensive then C's the addition of GLib to the project is easily justified.

Implementing the graph representation itself was very straight forward. Similar to the mathematical representation a **graph** in *streets4C* consists of an array of **node**s and **edge**s. To be able to map from OSM IDs to array indices two hash tables were added using `long`s as keys and `int`s as values. The **dgraph** structure can be created with a pointer to an existing **graph** and is then able to execute Dijkstra's SSSP algorithm.

```
1  struct node_t
2  {
3      long osm_id;
4      double lon, lat;
5
6      GHashTable *adj; // == adjecent edges/nodes
7  };
8
9  struct edge_t
10 {
11     long osm_id;
12     int length;
13     int max_speed;
14     int driving_time; // == edge weight
15 };
16
17 struct graph_t
18 {
19     int n_nodes, n_edges;
20     node *nodes;
21     edge *edges;
22
23     GHashTable *node_idx;
24     GHashTable *edge_idx;
25 };
26
27 struct dgraph_t
28 {
29     graph g;
30     pqueue pq;
31
```

[9] https://developer.gnome.org/glib/stable/

```
32      int cur; // == index of current node to explore
33      int *dist;
34      int *parents;
35  };
```

Listing 4.9: Graph representation in C

All structures contain little more than the expected data bseides the **cur** field in **dgraph**. 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 code bloat and added unsafety intrduced by the use of **GHashTable**s[10]. Since C is not typesafe by design and does not allow for true generic programming via type parameters nearly all generic code is written using `void*`. This leads to very verbose code because of the high amount of casts involved when accessing or storing values inside the data structures mentioned earlier.

Another complication was the use of integers as keys in **GHashTable**. It requires both key an value to be a gpointer (which is a platform independant void pointer) which forces the programmer to either allocate the integer key on the heap or explicitly cast it to the correct type. This works well using a macro provided by GLib until the number zero appears as a value because it represent the **NULL** pointer which GHashTable also uses to indicate a key was not found in the hash table.

The implementation of Dijkstra's algorithm was not particularly hard only more verbose than expected. As mentioned the GHashTable only provides iterative access through an extra function. As a consequence the step commonly referred to as *relax edge* is contained in a separate function that get passed to **g_hash_table_foreach**. In combination with the conversion macros and temporary variables the code bloats up.

All in all the experience was poor compared to the other languages. The verboseness and missing safety lead to the highest development time and SLOC count by far. The time was spent debugging some obscure errors introduced by the excessive casting which might have been avoided by a more sophisticated typesystem.

4.2.2. Go

For Go a similar approach was chosen. The graph is again mostly composed of two arrays holding all nodes and edges. However unlike the native C variants Go's **slices** and **maps** are dynamically growing which means the constructor function of the graph does not require capacity parameters. In general the development process was once again

[10] The hash table implementation provided by GLib

very smooth and simple which shows in the short time spent and the low SLOC count.

```
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

As the listing shows the structures are nearly identical to their C counterparts. Only the current node index in DijkstraGraph was not required since Go allows for much better iteration through maps. It is also interesting to note that Go supports (and even encourages) the declaration of multiple fields of the same type on the same line. Although this was used only two times in the snippet it shrinks the line count while keeping the code understandable since two fields with identical types are often related anyway.

As stated in the introductory part Dijkstra's algorithm depedends on a priority queue. Despite the fact that Go's standard library does not directly provide a ready-to-use implementation thereof the required steps to achieve this were minimal. The package

container\heap[11] offers a convenient way to work with any kind of heap. The only restriction is that the underlying data structure implements a special interface containing common operations used to *heapify* the stored data. Since interfaces are implicitly implemented on all structures which present the necessary methods it was a simple task to create a full featured priority queue on top of a slice by writing just four trivial methods.

```
type NodeState struct {
    cost uint
    idx  int
}

type PriorityQueue []NodeState

func (self PriorityQueue) Len() int {
    return len(self)
}
func (self PriorityQueue) Less(i, j int) bool {
    return self[i].cost < self[j].cost
}

func (self PriorityQueue) Swap(i, j int) {
    self[i], self[j] = self[j], self[i]
}

func (self *PriorityQueue) Push(x interface{}) {
    *self = append(*self, x.(NodeState))
}

func (self *PriorityQueue) Pop() (popped interface{}) {
    popped = (*self)[len(*self)-1]
    *self = (*self)[:len(*self)-1]
    return
}
```

Listing 4.11: Priority queue in Go

While the heap implementation was provided by the standard library (which is likely to be correct) it required the custom methods of Listing 4.11 to be correct. At this point Go's builtin test functionality came in handy. All it took to test the custom implementation was to create another file called **util_test.go** (adding the suffix "_test" to the already existing file **util.go**) and write a simple test. No import besides the *testing* package were needed since the code resided in the same package as the main application and all

[11] http://golang.org/pkg/container/heap/

test got executed with a single call of `go test` from the commandline. In contrast the C implementation of the queue required the setup of an additional source file including a regular main function which then had to manually compiled and run. In addition some basic error formatting and output[12] had to be written to properly locate potential errors in the implementation. Although all test related statistics are not counted in either language, Go's testing workflow is clearly superior to the manual, errorprone C approach.

All things considered this milestone was easily implemented in Go. The builtin container datastructures simplified the structure definitions while the provided heap implementation had a very low entry barrier and produced quick results. This reflect in the statistics which are on par with the Rust version discussed in the next section.
technically a subsection?

4.2.3. Rust

The original plan for the Rust implementation was to use direct references between nodes and edges of the graph to allow for easy navigation during the algorithm. Combined with the guarantees the typesystem offers it seemed to be a unique approach offering both convenient access and memory safety. Unfortunately this approach was quickly dismissed since it would have essentially created circular datastructures. While those are definitely possible to implement, it takes some **`unsafe`** marked code and a lot of careful interface design to retain the aforementioned safety. Due to once again time restrictions a architecture to the Go and C variant was implemented

The interesting differences in contrast to the previously introduced structures are located in **`DijkstraGraph`**. The **`queue`** field has the type **`BinaryHeap`** which is located in the standard library. This already shows that Rust is the only language out of the candidates which contains a complete implementation of this datastructure as part of the core libraries. While a priority queue is certainly not an essential component of every program it was required for this algorithm and having it available right from the start was beneficial to the development.

```
1 pub struct DijkstraGraph<'a> {
2     pub graph: &'a Graph,
3     pub queue: BinaryHeap<NodeState>,
4
5     pub dist: Vec<u32>,
6     pub parents: Vec<usize>
7 }
```

Listing 4.12: DijkstraGraph in Rust

[12] Which potentially could also contain errors(!)

The other interesting part is the type of the `graph` field. As mentioned earlier the struct calculating the shortest paths needs a reference to the immutable graph data. Ideally one would like to encode this immutability in the type itself. This is where Rust's typesystem shines. As mentioned in the language introduction regular references only allow read access. This means DijkstraGraph cannot (accidentally or intentionally) modify the referenced Graph instance or any of its fields just because the reference does not allow this. This comes in handy later in a parallel scenario where multiple threads are reading data from the graph while calculating shortest paths. The readonly reference (in Rust terms "a shared borrow") ensures no data races can happen when accessing the graph concurrently.

From a statistics standpoint Rust is evenly matched with Go. While *streets4Go* took a little less time to write, *streets4Rust* has a few less lines. This mostly came down to the heap implementations being available in the standard library (which means less code had to be written) and the mentioned deviation from the original implementation plan, adding some additional development time.

4.2.4. Comparison

Although this milestone did not contain any performance measurements it clearly highlighted and emphasized the original argument for a new language in high-performance computing. In scenarios where complex data structures beyond a simple array are required C fails to deliver an easy development experience. This was mostly due to the lack of "true" generic programming limiting the expressiveness of the implemented structures and algorithms. Since all casts in C are unsafe anyway but required to enable genericity, one slight type error, which a rigid typesystem might have been prevented from even compiling, can cause segmentation faults which are hard to trace and correct. This clearly underlines that C is not the optimal choice for developing complex high performance applications.

Go and Rust performed equally well in this stage. Both include a typesystem suited to safely use generic containers and provide a sufficient standard library for a decent implementation of a shortest path algorithm. Although Go's generics are limited to builtin types like slices and maps this was not an issue in this stage since no generic methods had to be written. Rust had the unique advantage to be able to express application semantics (graph data is immutable to the algorithm) in the typesystem. Although that did not solve any immediate problems in the implementation it can help to prevent a whole class of defects as described in the previous section.

4.3. Verifying structure and algorithm

The next goal was to verify the implemented algorithms on some sample data. To achieve this a sample graph with ten nodes and about 15 edges was constructed followed by a

shortest path calculation for each node. Although performance was measured it was not the core focus of this stage since the input data was very small and not representative of the OSM data. Nonetheless the execution time reveals some interesting differences between the competitors.

	C	Go	Rust
SLOC (total)	633	275	232
Development time (hours)	01:53:30	01:16:49	01:04:38
Execution time (seconds)	0.004 (-O0)	0.686	0.007 (-O0)
	0.003 (-O3)		0.005 (-O3)
Allocation count	108	519	47
Free count	106[13]	169	47
Allocation amount (bytes)	7,868[14]	53,016	22,792

Table 4.3.: Milestone 3: Verifying the implementation

4.3.1. C

Unsurprisingly the C implementation has the lowest execution time among the compared languages. Unfortunately the performance was once again paid with a high development time following the trend from previous milestones. In this phase a lot of time was invested into debugging the custom priority queue implementation. Although the there was a simple test performed in the last stage the real data revealed a bug when queuing zero indices. Similar to the GHashTable the zero index was casted to a void pointer and treated as null which caused errors during the pop operation later on. Unfortunately the defect manifested in the typical C style with a nondescriptive segmentation fault.

Performancewise C proves once again why it is one of the two major players in HPC. The execution time is unbeaten (even unoptimized) and the allocation amount is the lowest among the contestants by far. As explained in the annotation the mismatch in **`malloc`** and **`free`** calls can be explained by the inclusion of GLib. For its advanced features like memory pooling it retains some global state which *valgrind* mistakenly classifies as a potentital leak.

4.3.2. Go

The verification in Go took a little bit longer than expected. Although the implementation itself was quickly completed it exposed some errors in the original graph structure. The main problem was the initialization of the graph slices. The previous implementation

[13] Due to the use of GLib some global state remains reachable after exiting. This is likely intended behaviour and not a memory leak (see: `http://stackoverflow.com/a/4256967`).

[14] 2,036 bytes were in use at exit see footnote 14

used the builtin function **make** to create a slice with an initial capacity. When adding nodes to the slice later another builtin function **append** was used under the assumption the slice would be empty initially. This was not the case since Go had just filled the whole slice with empty node objects. This caused errors later down the line when these empty objects were used in Dijkstra's algorithm. The problem was later solved by changing the creation function from **make** to **new**. This method just creates a new array and lets the slice point into it reallocating later if neccessary.

While the actual change in code was minimal the origin of this defect are interesting. As mentioned above all three functions interacting with the slice are built into the language itself. This approach was explicitly chose to make common operations (like creating, retrieving the length or capacity) on common types (like slices, arrays and maps) more accessible. Unfortunately these functions obviously have slightly different meanings on different types resulting in some unexpected behaviour. This is certainly something which can be picked up when using Go for extended periods of time but for newcomers especially if can cause some confusion and while the new variant with **new** works it is unclear whether this is the ideomatic way to create growing slices.

From a performance standpoint *streets4go* falls short compared to the other implementations. This can mostly be explained by the Go runtime. It needs some initial setup time and memory which increases the allocation amount and prolongs execution time. Since this was a very small benchmark only used to validate the implementations these little "static costs" make up a much higher percentage of the total statistics.

4.3.3. Rust

During the implementation an effort was made to randomize the order of languages between milestones. It just happened that Rust was the last candidate in this phase since an update in the nightly version of the compiler broke the Protocol Buffers package on which the OSM library depended on. Although the author was quick to update the code to the changes there was a downtime of about three to four days where the development could not continue since the other versions were finished but *streets4Rust* did not build. Although this was the only case where the code was majorly broken for larger timespan it still effectively halted the whole process. Luckily the first stable release is scheduled for shortly after the deadline of this thesis so this should not really be a problem later on.

In this case it was even an advantage that the Rust version was developed last since it revealed a critical error in the other implementations. When creating the sample graph all data was derived from indices of two for loops. The assumption was that edges created in the second loop would only reference existing nodes created in the first one. Since both other implementations did not crash or produce any errors the creation code was not thoroughly verified. Running the same sample data through the Rust application revealed the error. The **add_edge** method did not check whether the edge ids passed as arguments were previously added to the graph. This is mandatory since the ids get converted to array indices to be able to add the nodes in the respecting adjecency lists.

A map lookup in Go or C is achieved via the indexing operator which then returns and the value element associated with the given key. Obviously this operation can fail when the given key is not found in the map. While both C and Go indicate this error case with a return of zero the possibility of failure is directly encoded in the corresponding Rust. Instead of simply returning the value it returns an **`Option`** which is a Rust enumeration is either containing a value or **`None`**. This type is a perfect fit for functions which might fail to return the desired value since it shifts the responsibility to deal with the failure to the callee which can potentially recover or otherwise abort completely. The following listings show the id to index conversion in Go and Rust highlighting the differences.

```
func (g *Graph) AddEdge(n1, n2 int64, e *Edge) {
    // [..]
    // link up adjecents
    n1_idx, n2_idx := g.nodeIdx[n1], g.nodeIdx[n2]
    // [..]
}
```

Listing 4.13: Map lookup in Go

```
pub fn add_edge(&mut self, n1_id: i64, n2_id: i64, e: Edge)
    ↪ {
    // [..]
    // link up adjecents
    let n1_idx = self.nodes_idx.get(&n1_id).unwrap();
    let n2_idx = self.nodes_idx.get(&n2_id).unwrap();
    // [..]
}
```

Listing 4.14: Map lookup in Rust

Although not shown here the C version behaves identical to the Go version but uses a static function **`g_hash_table_lookup`**. While the indexing seems more convenient is does not offer precise feedback over the success state of the operation. In this context this is especially critical since zero is a semantically valid index to retrieve. As mentioned above the return value of Rust's **`HashMap.get`** is an **`Option`** and as such it has to be "unwrapped" to get the contained value. This method panics the thread if called on a **`None`** value which is exactly what happened when the application processed the sample data. Further investigation then revealed a missing check whether the key is contained in the map which got silently ignored in both other applications. This is a good exaple of how a sophisticated typesystem can prevent potential errors through descriptive types.

The statistics for this milestone are once again very promising for Rust. With the lowest SLOC count as well as develoment time it still remains competitive with the execution performance of C. The allocation count also hints that Rust's vectors reallocate larget than the GHashTable from GLib. While the count of function calls is smaller the amount of memory allocated is larger.

4.3.4. Comparison

This milestone highlighted the importance of strong typesystems in particular. They can prevent bugs which would otherwise require intensive testing to be even noticed in the first place. In addition C once again comes in last in terms of developer productivity and while the performance benefit is still in its favor Rust reaches a similar speed with less implementation time. While Go is certainly not as fast as the other two languages as of yet it is a very comfortable language. // feels short/wrong?

4.4. Sequential benchmark

In this milestone runtime performance came back into the main focus. Since the algorithms at this point were proven to work correctly it could now be applied to real geographical data. The main technical challenge here was to efficiently process the input file while ideally directly filling the graph with the accumulated data. The caveat was the handling of OSM `ways` which are later represented by one or more edges in the graph. The input format lists all ids of the nodes which are part of the way but these nodes might not have been processed and added to the graph yet. This forced the implementations to retain the ways' data in some way and construct edges from that data in a second step.

	C	Go	Rust
SLOC (total)	757	359	292
Development time (hours)	01:14:32	00:56:16	00:45:20
Execution time (hours)	08:34:01 (-O3)	09:08:19	07:31:37 (-O3)
Memory usage (MB)[15]	994	1551	2235

Table 4.4.: Milestone 4: Sequential benchmark

4.4.1. C

For *streets4C* the most time was spent on dealing with the edges. As mentioned in the introductory part they need to be saved and added later. This either required knowing the amount of edges beforehand (to be able to preallocate an array large enough to store all information) or to use a dynamically growing array to store them as they get processed. Since the amount of edges is not stored in the OSM file, the first approach would have to read the whole input file twice to count edges (and ideally nodes too) first and then parse the actual data in the second run. To prevent this the second design as implemented.

[15] Obtained via htop (`http://hisham.hm/htop/`) at the time of shortest path calculation

Of course reallocating arrays are not part of the C language or standard library and the application had to rely on GLib once again. The used types were **GPtrArray** to store pointers to the heap allocated *node* and *edge* structures and the regular **GArray** to store the OSM ids of the constructed edges. With this implementation the file only needed to get read once creating the actual values and counts (useful to pass to the graph creation method as capacities later) in the process.

Another option would have been to rewrite part of the graph structure to use GArrays internally for storing edges and nodes in the first place. This idea was not realized to keep the number of external data structures in the graph representation minimal. However that change would have simplified this milestone considerably.

The performance statistics from this phase contains the first real surprise. *streets4C* does not have the lowest execution time. Rust outshines the C implementation by more than an hour. This might reflect a suboptimal architecture on the C side which can be traced back to my limited experience with the language. However this might very well be reflective of a scientist with similar limited programming skills. Considering the development time and SLOC count the result is even more alarming. The redeeming factor of the C variant is the memory footprint which is the lowest among the three languages. Although memory is typically not as critical as processing time it is still an important criteria when evaluating HPC applications.

4.4.2. Go

This phase did not offer any difficult technical challenges for the Go implementation. Nodes could be added right as they were encountered while parsing whereas edges were temporarly stored in a growing slice and were appended in a second pass.

An essential function for this milestone was the calculation of the length between two nodes based on latitude and longitude. Based on *streets4MPI* the haversine formula[16] was chosen to perform this calculation. This kind of mathematical formulae can be implemented very compactly in Go. Especially the assignments of multiple variables in the same line helps readability and reduces the amount of lines required.

```
1 const conversionFactor = math.Pi / 180
2
3 func Rad(deg float64) float64 {
4     return deg * conversionFactor
5 }
6
7 func HaversineLength(n1, n2 *Node) float64 {
8     lat1, lon1, lat2, lon2 :=
9       Rad(n1.lat), Rad(n1.lon), Rad(n2.lat), Rad(n2.lon)
```

[16] http://en.wikipedia.org/wiki/Haversine_formula

```
10      dlat, dlon := lat2-lat1, lon2-lon1
11      a := math.Pow(math.Sin(dlat/2), 2) + math.Cos(lat1) *
12          math.Cos(lat2) * math.Pow(math.Sin(dlon/2), 2)
13      c := 2 * math.Asin(math.Sqrt(a))
14      return 6367000 * c // distance in m
15  }
```

Listing 4.15: Haversine formula in Go

While the listing is a bit small in width the advantages should be clearly visible. The multiline assignment are very compact and are useful for these shortlived intermediate results. In Go all identifiers are located on the left side of the assignment while a multi assignment in C consists of multiple assignment separated with a comma. This mix of identifiers and values takes some additional time to mentally parse - a problem which the Go way does not suffer from.

In the performance comparison Go comes in at the third place as expected but the gap to C is actually not as big predicted. Interestingly *streets4Go* has only the second highest memory consumption despite being garbage collected and therefore lacking deterministic destruction. This differentiates Go from other garbage collected languages like *C#* or *Java* which encourage "allocations hevy programming" under the premise that memory cannot leak and can therefore be allocated often.

4.4.3. Rust

The Rust implementation follows the same pattern as the Go version. Nodes are added to the graph as they are decoded while edges are stored in a vector to add later. There was a caveat though where Rust's ownership tracking typesystem

After the input data has been processed the **edges** vector contains all edges found in the file. Which means in Rust terms that the vector *owns* all objects inside it. On the other hand the **add_edge** method also needs to take ownership of the edge argument passed to it. This was a deliberate choice while designing the interface since the graph should own all edges and nodes it consists of. This creates a conflict because indexing the vector only returns a reference to the object. The solution was to use a moving iterator to remove the objects from the vector to be able to pass them to **add_edge**. In addition the two node ids of the edge, which were stored in additional vectors, needed to be retrieved the same way. To achieve this the two iterators were zipped and as a consequence all required variables could be in scope at the same time.

```
1   fn benchmark_osm(osm_path: &Path) {
2       // [.. initial setup ..]
3       let mut g = Graph::new();
4       let mut edges = Vec::new();
```

```
5       let mut indices = Vec::new();
6
7       // [.. parse nodes and edges ..]
8
9       // Add edges to graph
10      for (mut e, (n1, n2)) in edges.into_iter()
11          .zip(indices.into_iter()) {
12          e.length = graph::haversine_length(
13              &g.nodes[g.nodes_idx[&n1]],
14              &g.nodes[g.nodes_idx[&n2]]) as u32;
15          g.add_edge(n1, n2, e);
16      }
17      // [.. perform calculations ..]
18  }
```

Listing 4.16: Zipped iterators in Rust

The **Iterator.zip** method takes an iterator and returns a composite iterator which yields elements from the two as a pair. Here **indices** is a vector of pairs of i64 (Rust's 64-bit integer) so the iterator created by zip yields a pair of type **(edge, (i64, i64))**. Also the edge length had to be calculated before the edge could be added to the graph. This meant the **mut** keywork had to be added in front of the edge variable. The resulting expression for (mut e, (n1, n2)) in edges.into_iter().zip(indices.into_iter()) looks very complicated but is actually pretty simple when taken apart. These tangled statements are probably the major disadvantage of the complex typesystem. There are a lot of sigils which sometimes even have double meanings.

The performance comparison was specifically interesting in this stage because Rust was actually ahead of C for the first time. It is also remarkable that the development time and SLOC count are the lowest of the three languages compared. This means the usual tradeoff between performance and productivity does not apply in this particular case and Rust straightup beats Go and C in all three tracked categories which is impressive.

4.4.4. Comparison

4.5. Parallel benchmark

Last but not least it was time to parallelize the calculations. Because the algorithm itself is not very easy to execute concurrently the choice was made to calculate multiple shortest paths in different threads. Also the main loop over the first 100.000 nodes was already in place from the previous milestone and so only this block had to be updated to run in parallel.

It is also important that the parallelized code does not make any use of the computed results as the previous versions did not do that either. In concurrent scenarios this

becomes a separate problem because of synchronization issues. Although it is not included in the implementations each of the following sections will describe possible strategies to deal with the calculated results.

	C	Go	Rust
SLOC (incl. previous stages)	777	381	314
SLOC (final)	668	285	253
Development time (hours)	00:08:11	00:07:56	00:27:23
Execution time (4 threads) (hours)	03:22:21 (-O3)	03:47:30	02:32:06 (-O3)
Memory usage (MB)[17]	994	1551	2235

Table 4.5.: Milestone 5: Parallel benchmark

4.5.1. C

As mentioned in the previous chapter OpenMP was the framework of choice to parallelize streets4C. Given this there were two possible approaches to consider. Either the `for` pragma applied to the existing loop or a regular `omp parallel` block.

4.5.2. Go

streets4Go obviously used goroutines for concurrent execution. As the development time shows the implementation was really simple and convenient. Similar to the C variant the only required change

4.5.3. Rust

4.5.4. Comparison

4.6. Preparing execution on the high performance machine

[17] Obtained via htop (`http://hisham.hm/htop/`) at the time of shortest path calculation

5. Evaluation

This chapter provides the analysis of the statistics gathered from the final implementations. As stated in the introduction the evaluation considers raw performance characteristics as well as developer productivity from the previous chapter.

Preface: All data in this chapter was gathered from a high performance computer by courtesy of the research group . The machine has access to 48 singlecore processors and 128 GB of memory. It is therefore ideal to compare shared memory performance on a large scale.

5.1. Performance

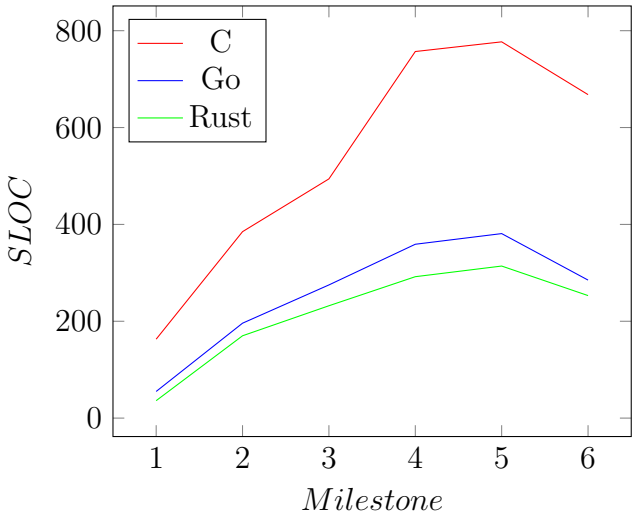
threads/goroutines	C	Go	Rust
1	xx:xx:xx	16:48:19	14:15:06
2	xx:xx:xx	10:21:36	09:12:47
4	xx:xx:xx	05:58:35	05:09:56
8	xx:xx:xx	03:01:54	02:49:35
12	xx:xx:xx	02:06:08	01:55:33
24	xx:xx:xx	01:13:47	01:03:34
48	xx:xx:xx	00:53:54	00:44:54

Table 5.1.: Execution time of the final applications (100.000 nodes)

threads/goroutines	C	Go	Rust
1	1.0000	1.0000	1.0000
2	xx:xx:xx	1.6221	1.5469
4	xx:xx:xx	2.8119	2.7590
8	xx:xx:xx	5.5432	5.0424
12	xx:xx:xx	7.9941	7.4003
24	xx:xx:xx	13.6659	13.4520
48	xx:xx:xx	18.7072	19.0445

Table 5.2.: Parallel speedup of the final applications (100.000 nodes)

5.2. Additional metrics / productivity



6. Conclusion

In diesem Kapitel ...

6.1. Improvements and future work

Although this evaluation already yielded some interesting results regarding new programming languages in HPC there are still lots of potential research topics in this area. Also there are some missed opportunities for a more complete result which were just not considered or skipped due to time constraints. This section addresses both of these points and.. //TODO: end this sentence

6.1.1. Code remarks to *streets4x*

6.1.2. Limitation to shared memory

While thread based concurrency is certainly an important aspect in HPC the dominant model is distributed memory communicating via message passing. This technique was not evaluated in this thesis because of missing library support but the general performance of the languages should still be applicable. As both Rust and Go have good capabilities to link to native C libraries it might be possible to use a standard MPI implementation today. A complete library written in the target language should be preferred whenever available though because interfacing with C often restricts the types which can be used. When these libraries have matured enough (if ever) it might be very valuable to reassess the candidates in the HPC context.

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).

[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.

[Dev] Google Developers. *Protocol Buffers*. URL: https://developers.google.com/protocol-buffers/ (visited on 21. 04. 2015).

[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).

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

[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).

[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).

[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).

[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.

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

[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).

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

[maib] The Go project maintainers. *The Go Programming Language - Package testing*. URL: http://golang.org/pkg/testing/ (visited on 11. 02. 2015).

[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.

[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).

[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.

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

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

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

[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.

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

[WFV14] F.D. Witherden, A.M. Farrington, and P.E. Vincent. "PyFR: An open source framework for solving advection–diffusion type problems on streaming architectures using the flux reconstruction approach". In: *Computer Physics Communications* 185.11 (2014), pp. 3028–3040. ISSN: 0010-4655. DOI: 10.1016/j.cpc.2014.07.011. URL: http://www.sciencedirect.com/science/article/pii/S0010465514002549.

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

iterator invalidation

A common problem in languages without automatic memory management which can occur when an iterator is used on a mutable container. For example when iterating over a dynamically growing vector which reallocates itself, the pending iterator pointer can become dangling. Thus making it effectively unusable or invalid.

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

OpenStreetMap

OpenStreetMap is a collaborative project which aims collect and maintain geographical data about roads, trails, railways stations and more. As the name suggests the data is provided openly under the Open Data Commons Open Database License [1]

OSM

OpenStreetMap

Protocol Buffers

"Protocol buffers are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data" [Dev]

Single Source Shortest Path

A common graph problem searching for shortest paths between nodes of a graph. As the name suggests this type of problem states the use a single node as starting point and aims to determine shortest paths to all remaining nodes of the graph. Dijkstra's algorithm is commonly used for graphs with nonnegative edge weights while the Bellman-Ford-Algorithm can even handle that case.

SLOC

source lines of code

SSSP

Single Source Shortest Path

TDD

test-driven development

XML

Extensible Markup Language

[1] `http://opendatacommons.org/licenses/odbl/`

B. System configuration

Development laptop

```
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:               799.921
CPU max MHz:           2900,0000
CPU min MHz:           800,0000
BogoMIPS:              3992.36
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`

Cluster

Listing B.3: Output of `uname -a`

Listing B.4: 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 1.1.0-nightly (21f278a68 2015-04-23) (built
   ↪ 2015-04-24)
```

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

```
cargo 0.2.0-nightly (dac600c 2015-04-22) (built 2015-04-23)
```

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 .....