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 28, 2015

Abstract

This thesis aims to analyze new programming languages in the context of high-performance computing (HPC). In contrast to many other evaluations the focus is not only on **performance** but also on developer **productivity metrics**. The two new languages Go and Rust are compared with C as it is one of the two commonly used languages in HPC next to Fortran.

The base for the evaluation is a shortest path calculation based on real world geographical data which is parallelized for shared memory concurrency. An implementation of this concept was written in all three languages to compare multiple productivity and performance metrics like **execution time**, **tooling support**, **memory consumption** and **development time** across different phases.

Although the results are not comprehensive enough to invalidate C as a leading language in HPC they clearly show that both Rust and Go offer tremendous **productivity gain** compared to C with **similar performance**. Additional work is required to further validate these results as future reseach topics are listed at the end of the thesis.

Table of Contents

1. Introduction

This chapter provides some background information to 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 high-performance computing is evolving rapidly and programming languages used in this environment are held up to a very high performance standard. This is not surprising 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 accumulated over the past centuries. Especially C's *undefined behavior* 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 verify 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 lowering developer productivity. In contrast, for example, the Go programming language includes a complete testing framework with the ability to perform benchmarks, perform global setup/tear-down work and even do basic output verification [maic].

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 heat-maps 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 based on multiple criteria. Since libraries for inter-process communication in Rust and Go are nowhere near production-ready this thesis will focus on shared memory parallelization. Additionally unfair bias based solely on the quality of the supporting library ecosystem should be avoided.

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 idiomatic way which can optionally be parallelized. While raw performance will be the main criteria, additional productivity metrics will also be reviewed to rate the general development experience. Another focus will be the barrier of entry for newcomers to the respective languages which is important for scientist less proficient in programming.

1.3. Structure

This first chapter briefly motivated the search for new languages in HPC and outlined the goals of the thesis. The second chapter *State of the Art* describes common programming paradigms in C and Fortran and introduces the various languages which were

considered for further evaluation. The following chapter *Concept* describes the original case study *streets4MPI* which the evaluation is based on, illustrates the various phases of the implementation process and mentions some related work. The fourth chapter *Implementation* describes each implementation milestone in detail providing and briefly comparing intermediate results. The fifth chapter *Evaluation* compares the various criteria for both performance and productivity and judges them accordingly. The final chapter *Conclusion* summarizes the results of the evaluation and lists some possible improvements and future work.

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 satisfying and could be improved.

Fortran (originally an acronym for FORmula TRANslation) is the traditional choice for scientific applications like climate simulations. As the name suggests 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 intrinsic functions for many common mathematical operations such as matrix multiplication or trigonometric functions and a built-in datatype for complex numbers. In addition, memory management is nearly nonexistent. In earlier versions of Fortran it was not possible to explicitly allocate data. 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 intrinsic functions sometimes are just not fast 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 datatypes which in turn require a lot more micromanagement 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 compatibility. 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. Listing 2.1 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 Python 3.4

In addition to the very extensive standard library the Python community has created a lot of open source projects aiming to support especially scientific applications. There is NumPy[1] which offers efficient implementations for multidimensional arrays and common numeric algorithms like Fourier transforms or MPI4Py[2], an Message Passing Interface (MPI) 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. 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.

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 *CPython*[3] and *PyPy*[4]) but new projects aim to improve the execution speed in various ways. *Medusa*[5] compiles Python code to Google's *Dart*[6] 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 specific purpose programming language originally designed for the use in telephony applications. It features a high focus on concurrency and a garbage

[1] http://www.numpy.org
[2] http://www.mpi4py.scipy.org
[3] https://www.python.org
[4] http://www.pypy.org
[5] https://github.com/rahul080327/medusa
[6] https://www.dartlang.org/

collector which is enabled through the execution inside the Bogdan/Björn's Erlang Abstract Machine (BEAM) virtual machine. Today it is most often used in soft real-time computing[7] 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

[7] 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. Each Erlang process in itself behaves purely (meaning the result of a function depends solely on its input). The collection of processes interacting with each other through messages 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" [maia]. 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.

Listing 2.3 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.

```go
package main

import "fmt"

// Number of gorountines to start
const GOROUTINES = 4

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

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

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

Listing 2.3: Go concurrency example

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

While Go's focus on simplicity is admireable it has also been 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. At 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 it provides easy to use parallel constructs, the aforementioned goroutines. It will probably not directly compete with C in execution performance but the great toolchain and simplified concurrency might top 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*[8] as backend and is compatible to common tools like *The GNU Project Debugger* (*gdb*)[9] which makes integration into existing workflows a bit easier. Compared to the discussed here languages int this chapter Rust is closest to C while attempting to fix common mistakes made possible by its loose standard allowing undefined behavior.

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 constraint 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 only 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

[8] `http://www.llvm.org`
[9] `http://www.gnu.org/software/gdb/`

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.

```
// 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 nearly identical to an equivalent C program. It also has the advantage of being still in development[10] so concepts which did not work out can be quickly changed or completely dropped.

But the immatureness of Rust 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

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

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 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 similar to native code the **productivity gains** should still be substantial.

3. Concept

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

3.1. Overview of the Case Study *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. Figure 3.1 provides a rough overview about the architecture of *streets4MPI*.

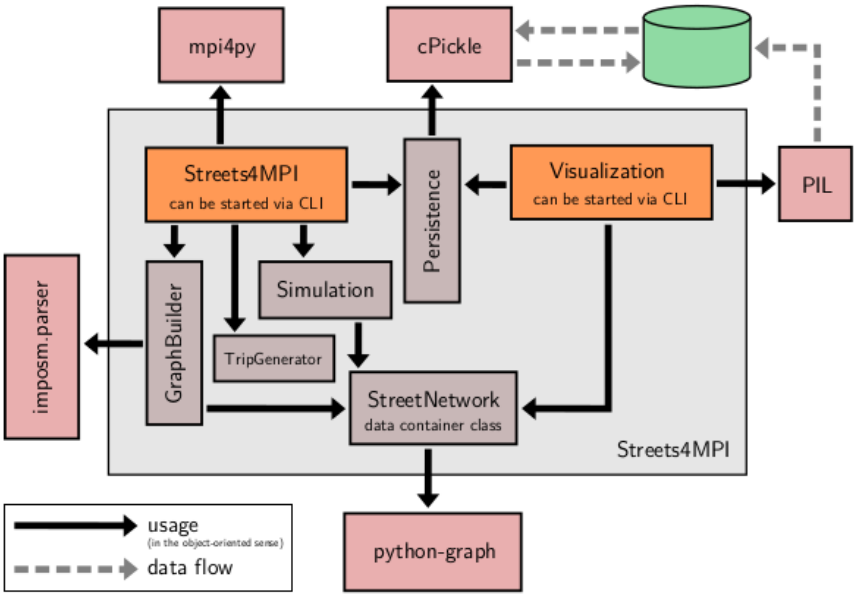


Figure 3.1.: Architecture overview: Streets4MPI [FN12, p. 9]

The *GraphBuilder* class parses OpenStreetMap (OSM) input data and builds a directed graph which is stored in the *StreetNetwork*. The *Simulation* than uses this data and

repeatedly computes shortest paths for a set amount of *trips* (randomly chosen node pairs from 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 Persistence class then optionally writes to results to a custom output format which is visualizable by an additional script [FN12].

3.2. Differences and Limitations

Although the evaluated applications are based on the original *streets4MPI*, there are some key differences in the implementation. This section gives a brief overview over the most important aspects that have been changed. The first paragraph of each subsection describes the original application's functionality while the second highlights differences and limitations in the evaluated 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 Single Source Shortest Path (SSSP) algorithm as seen in [Cor+09]. Also a random factor called "jam tolerance" is introduced to avoid oscillation between two high traffic routes in alternating

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

iterations [FN12]. Then after some time has passed in the simulation, existing streets get expanded or shut down depending on their usage to simulate road construction.

The compared implementations of this thesis also perform trip based simulation but without the added randomness and street modification. Also the edge weights are not dynamically recalculated in each iterations. Instead the street's length is calculated once from the corrdinates of the corresponding nodes and used as edge weigth directly. The concrete algorithm is a variant of the `Dijkstra-NoDec` SSSP algorithm as seen in [Che+07, p. 16]. It was mainly chosen because of its reduced complexity in required data structures. The algorithm is implemented separately in all three languages so it could theoretically get benchmarked standalone to get clearer results. This was not attempted in scope of the thesis because of time constraints.

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 MPI implementation installed on the system. If no supported implementation is found it falls back to a pure Python solution. Results have show that the native one should be preferred in order to achieve maximum performance [FN12].

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* 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 phases 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.

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

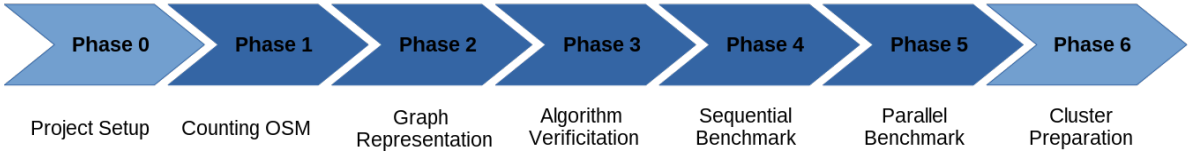


Figure 3.2.: Milestone overview

Figure 3.2 shows the different milestones in order of completion. For each phase 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. In the following sections each milestone is briefly described.

Setting up the Project

The first phase 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 might show the differences in comfort between the various toolchains.

Counting Nodes, Ways and Relations

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 phase 0 and finished after the milestone was reached. As this is the most input and output intensive phase it should reveal some key differences between the candidates both in speed as well as memory consumption.

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. In addition the shortest path algorithm had to be implemented. This meant a priority queue had to be available as the algorithm relies on that data structure to store nodes which have yet to be processed. This milestone therefore tested the language's standard libraries and expressiveness in terms of typed containers.

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.

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.

Benchmarking Parallel Execution

The fifth phase 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.

Cluster Preperation

The final milestone was to prepare the implementations for the execution on the cluster provided by the research group. As this was a remote environment with some key differences to the development laptop the implementations had to be prepared and slightly changed.

3.4. Overview of evaluated Criteria

For the evaluation of the three languages multiple criteria have been selected. While some of them are directly quantifiable such as development time others are rated subjectively based on experiences from the implementation process. This is mostly true for the productivity metrics. It is important to note that not all statistics apply to all milestones. The following list introduces the reviewed criteria and briefly describes them.

- Performance
  - Execution Time
    The time to complete the task of the milestone
  - Memory Footprint
    Total memory consumption as well as allocation and free counts
- Productivity
  - SLOC Count
    Source lines of code to roughly estimate the code's complexity and maintainability. Tracked in all milestones
  - Development Time
    Time required to implementation the desired functionality. Tracked in all milestones
  - Resource Management *Amout of work required to properly manage resources like memory, file handle or threads in a given language*
  - Tooling Support
    Tooling support for common tasks throughout the development process. This includes the compiler, dependency management, project setup automation and many more
  - Library Ecosystem
    Available libraries for the given language considering common data structures, algorithms or mathematical functions. Includes the quality of the language's standard library
  - Parallelization Effort
    Amount of work required to parallelize an existing sequential application

As these statistics were tracked during the implementation itself the next chapter directly lists and evaluates intermediate results for each milestone. In contrast Chapter 5 evaluates the final performance outcomes from the cluster benchmarks as well as the gathered productivity metrics.

3.5. 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] adds lines of code and development time to the equation but both of these metrics only allow for superficial conclusions about code quality and productivity.

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.

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 **??**. *The last section briefly describes the preparation process for the final benchmarks.*

4.0. Project Setup

All applications written for this thesis have been developed on Linux as it is the predominant operating system in HPC. They *should* compile and run 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 which Linux distribution is used. It is therefore not covered in this thesis.

4.0.1. C

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*[1] or *CMake*[2] and manual compilation. The setup steps required for this configuration are relatively straight forward and shown in Listing 4.1.

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

Listing 4.1: Project setup: streets4C

After generating 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.

[1] `http://www.gnu.org/software/software.html`
[2] `http://www.cmake.org`

All in all the setup in C is quite simple although it has to be performed manually. The only potential problem are `Makefile`s. They may be easy enough 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

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 code, install arbitrary Go packages from various sources, run tests and format source files just to name the most common features.

This makes Go extremely convenient since only one command is required to perform multiple common actions in the development cycle. 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's repository. It is pretty similar to the C version. A directory gets created then a source file containing a **main** function is created which can be built and run with a single command. Unfortunately this variant does not follow the guidelines for project layout as described in the official documentation because the code does not live inside the globally unique **GOPATH** folder.

To be able to download packages only once the **go** commandline utility assumes an environment variable called **GOPATH** is configured to point 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 projects 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.

Listing 4.3 shows a more realistic workflow for creating a new Go project from scratch without any prior setup required. It expects the programmer to start in the directory that should be set as **GOPATH** and uses **GitHub** 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 is accessible everywhere.

```
$ 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

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 Listing 4.4.

```
$ 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**[3] 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 incomplete at the moment but as with everything in the Rust world **cargo** is still in active development.

The overall greatest 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. In the other languages ones has 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 up and running without any prior knowledge which lowers the barrier of entry dramatically. In addition Rust does not require any presetup before the first project. After installing the language toolchain (either through the operating system's package manager or the very simple setup script[4]) the language is completely configured and the first project can be created.

Go requires some initial setup besides the installation but is 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 support** 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 the compiler and linker usability is still years behind other modern toolchains. One example is linking libraries where the developer can decide between potentially unneeded libraries being included in the application (with default settings) or having to carefully order the linker arguments (with the special flag **--as--needed**) which is tedious when new dependencies get added later on.

This probably does not apply to experienced C developers and one could make the argument that it is inherent to the language's low level nature. But acknowledging the fact that scientists of other fields 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.

[3] Which is executable anywhere inside the project directory

[4] `https://static.rust-lang.org/rustup.sh`

4.1. Counting Nodes, Ways and Relations

	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[5,6]	11,373,558
Free count	2,390,566	11,000,199	11,373,557[7]

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

4.1.1. C

For the first real milestone *streets4C* had an important disadvantage. There was no library to conveniently process OSM 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 phase.

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 included in the development process which. This meant the extending the existing `Makefile` in order 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 the author's lacking expertise with the C compilation process but also confirms the unneeded complexity of such a simple task.

Ultimately the problem was the order in which the source files and libraries were passed to the compiler and linker. The libraries were included too early which resulted in "undefined reference to method" error messages because the aforementioned linker flag **`--as--needed`** was enabled per default by the Linux distribution. In this mode the linker effectively treats passed objects files as completed when no missing symbols are found after the unit has been processed and therefore ignores them in the further linking process. As a result the arguments have to carefully match the dependency hierarchy to not accidentally remove a critical library early on so that later files cannot use their symbols.

[5] The memory statistics for Go have not been acquired by *valgrind* but by **`runtime.MemStats`**

[6] The fact that Go is garbage collected explains the discrepancy in allocations and frees

[7] 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).

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. Meanwhile other toolchains show that it is definitely possible to accept arguments in arbitrary order and perform the required analysis whether to include a given library in a second pass. This effectively combines the best of both inferior strategies the C linker currently supports. The **time** spent solving these compilation errors shows in the statistics for C which is considerably larger than its competitors in this phase.

The other big caveat in working with OSM data was the **manual memory management**. Since data is stored in effectively compressed manner in the file additional heap 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 freeing functions generated from the source **`.proto`** format description files. For some intermediate allocations it is possible to limit this to the body of a library function but on the real data it shifts additional responsibilities on the caller.

```
1  /* somewhere in a function */
2  osmpbf_reader_t *reader = osmpbf_init(<some_path>);
3
4  OSMPBF__PrimitiveBlock *pb;
5  while((pb = get_next_primitive(reader)) != NULL)
6  {
7      for (size_t i = 0; i < pb->n_primitivegroup; i++)
8      {
9          // access data on the primitive groups
10         OSMPBF__PrimitiveGroup *pg = pb->primitivegroup[i];
11
12         /* no need to free pg here since its part
13          * of the primitive block pb */
14     }
15
16     // cannot use free(pb) here because of Protobuf
17     osmpbf__primitive_block__free_unpacked(pb, NULL);
18 }
19
20 // regular free function provided by library
21 osmpbf_free(reader);
22 /* remaining part of the function */
```

Listing 4.5: Manual memory management with Protobuf in C

Listing 4.5 shows this overhead introduced by the mandatory call to **`osmpbf__primitive_block__free_unpacked`** in line 17. This results in some very asymmetric interface

design since the parsing library has to rely on the client application to explicitly call the correct free function from the Protocol Buffers library. While this approach is acceptable for regular allocations via the C standard library, it is a problem here since the allocating function's name **`get_next_primitive`** does not directly imply a heap allocation (and the resulting need to free it later).

Considering this fact the **SLOC count** shown in Table 4.1 is still decent. With the help of a clever library interface the overhead for the memory management is comparatively small and the data can be iterated 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**[8] and superior single threaded performance C is the clear winner in the **performance** area for this first milestone.

4.1.2. Go

To parse the .osm.pbf files *streets4Go* uses an **existing library** simply called *osmpbf*[9]. 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** listed in Table 4.1 which is the shortest amongst the candidates for this first phase.

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

Listing 4.6: Dependency management in Go

[8] Although these are also partially caused by the simplicity of the custom osmpbfreader abstraction
[9] `https://github.com/qedus/osmpbf`

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 are once again Go's strict compilation rules. Since an unused import is a compiler error an 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.

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 language constructs) the **SLOC count** is in fact identical to that of *streets4C*. This is partially 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 successful completion. This pattern is used three times in this simple first phase alone which results in 12 lines.

```
1  func SomeIOFunction(path string) {
2      file, err := os.Open(path)
3      if err != nil {
4          log.Fatal(err)  // os.Open returned an error
5      }
6      err = pkg.SomeIOFunc(file)
7      if err != nil {
8          log.Fatal(err) // rinse and repeat
9      }
10 }
```

Listing 4.7: Idiomatic error handling in Go

Considering the aforementioned simplicity *streets4Go*'s performance characteristics as shown in Table 4.1 are very promising. Although in its basic form about four to five times slower than the C solution the parallelized version achieves similar performance to *streets4C*. This version was only included since the library was already based on a variable number of goroutines. This meant parallelization could be achieved by simply 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*[10]. 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 as Table 4.1 highlights. This can mainly be attributed to the library's use of common Rust idioms and structures like **`iterators`** and **`enumerations`**. Unlike their C equivalent, which are basically named integer constants, Rust enumerations are real types. This means they can be used in pattern matching expression and act as a target for method implementations similar to structures. Listing 4.8 shows the complete decoding part of this phase which is very compact and easy to understand thanks to Rust's high level constructs.

```
1  /* in main() */
2  for block in pbf_reader.primitive_blocks().map(|b|
   ↪ b.unwrap()) {
3      for obj in blocks::iter(&block) {
4          match obj {
5              objects::OsmObj::Node(_)         => nodes += 1,
6              objects::OsmObj::Way(_)          => ways += 1,
7              objects::OsmObj::Relation(_)     => rels += 1
8          }
9      }
10 }
11 /* remaining part of main() */
```

Listing 4.8: OSM decoding in Rust

The function **`blocks::iter`** (see line 3) 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 rom within.

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 (see Table 4.1). This is mostly due to the fact that the Rust LLVM frontend produces mediocre byte code which does not get optimized on regular builds. That is also the reason release builds take substantially longer. It simply takes

[10] `https://github.com/textitoi/osmpbfreader-rs`

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

The first phase already showed some severe differences in **performance** between the evaluated languages. Table 4.1 shows C is the fastest language as expected with Rust reaching similar single threaded performance. While Go was considerably in the single threaded variant it was simple to parallelize thanks to goroutines and achieved similar performance to C. Of course this is not fair comparison but the simplicity of the change shows the good integration of this parallel construct into the Go language.

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 SSSP algorithm to simulate trips. Since all applications should be parallelized later on the immutable data (such as the edge lengths, OSM IDs and adjacency lists) needed to be stored separately from the changing data the algorithm required (such as distance and parent arrays). To achieve this all implementations provide 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

As seen in Table 4.2 this phase 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 waste important space with structure definitions in the main source file 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 that 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*[11] 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 much 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 with the IDs as keys (of type `long`) and corresponding indices as values (type `int`). 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;          // == edge weight
13     int max_speed;
```

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

```
14      int driving_time;
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
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 besides the **cur** field in **dgraph**. It had to be added since Glib's **GHashTable** only supports operations on all key-value-pairs via a function pointer with a single extra argument. Since the algorithm requires access to the currently explored node's index as well as the distance and parent arrays the index needs to be stored in the struct itself.

While the extra field was a minor inconvenience other problematic aspects were the high amount of verboseness and additional unsafety introduced by the use of **GHashTable**s[12]. Since C is not typesafe by design and also 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 a **GHashTable** or **GArray**.

Another complication was the use of integers as keys in **GHashTable**. It requires both key and value to be a gpointer (which is a platform independent 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. Although there is an extended

[12] The hash table implementation provided by GLib

function which is able to indicate to caller whether the return value is **NULL** because it was stored that way or because it was not found, this problem could have been avoided by a better API design.

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. As Table 4.2 shows 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 type system.

4.2.2. Go

In Go the graph is again mostly composed of two arrays holding all nodes and edges. However Go's **slices** and **maps** are dynamically growing. This means the constructor function of the graph does not require capacity parameters to initialize these fields since they can reallocate if necessary. In general the development process was once again very smooth and simple which shows in the short **time** spent and the low **SLOC count** (see Table 4.2).

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

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

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

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

    nodeIdx, edgeIdx map[int64]int
}
```

```
21
22 type DijkstraGraph struct {
23     g *Graph
24     pq PriorityQueue
25
26     dist    []uint
27     parents []int
28 }
```

Listing 4.10: Graph representation in Go

Listing 4.10 shows that 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 depends 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**[13] 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. This is illustrated in Listing 4.11.

```
1  type PriorityQueue []NodeState
2
3  type NodeState struct {
4      cost uint
5      idx  int
6  }
7
8  func (self PriorityQueue) Len() int {
9      return len(self)
10 }
11 func (self PriorityQueue) Less(i, j int) bool {
12     return self[i].cost < self[j].cost
13 }
14 func (self PriorityQueue) Swap(i, j int) {
```

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

```
15      self[i], self[j] = self[j], self[i]
16  }
17  func (self *PriorityQueue) Push(x interface{}) {
18      *self = append(*self, x.(NodeState))
19  }
20  func (self *PriorityQueue) Pop() (popped interface{}) {
21      popped = (*self)[len(*self)-1]
22      *self = (*self)[:len(*self)-1]
23      return
24  }
```

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 described in Listing 4.11 to be correct. At this point Go's **built-in test functionality** came in handy. All it took to test the custom implementation was to create another file called **`util_test.go`** (the suffix "_test.go" is mandatory) 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 tests got executed with a single call of `go test`. In contrast the C implementation required the setup of an additional source file including a regular main function which then had to be manually compiled and run. In addition some basic error formatting and output 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 **automated testing workflow** is clearly superior to the manual, error prone C approach.

All things considered this milestone was easily implemented in Go. The built-in **container data structures** simplified the structure definitions while the provided heap implementation had a very low entry barrier and produced quick results. As Table 4.2 shows this is reflected in the statistics which are on par with the Rust version discussed in the next section.

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 type system 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 data structures. 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 an architecture similar 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 **data structure** 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. Listing 4.12 illustrated the resulting structure definitions.

```
pub struct DijkstraGraph<'a> {
    pub graph: &'a Graph,
    pub queue: BinaryHeap<NodeState>,
    pub dist: Vec<u32>,
    pub parents: Vec<usize>
}
```

Listing 4.12: DijkstraGraph in Rust

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 type system 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 read-only reference (in Rust terms a *shared borrow*) ensures **no data races** can happen when accessing the graph concurrently.

From a productivity standpoint Rust is evenly matched with Go as Table 4.2 clearly shows. 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 can cause segmentation faults which are hard to trace and correct. A rigid type system might have prevented the code from even compiling in

the first place. This clearly underlines that C is not the optimal choice for developing complex high performance applications.

Go and Rust performed equally well in this phase. Both include a type system 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 built-in types like slices and maps this was not an issue in this phase 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 **type system**. 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 phase 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[14]	169	47
Allocation amount (bytes)	7,868[15]	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

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

[15] 2,036 bytes were in use at exit see footnote 14

there was a simple test performed in the last phase 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.

Considering performance C proves once again why it is one of the two major players in HPC. Table 4.3 shows that the **execution time** is still 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 potential 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 used the built-in function `make` to create a slice with an initial capacity. When adding nodes to the slice later another built-in 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 necessary.

While the actual change in code was minimal the origin of this defect is interesting. As mentioned above all three functions interacting with the slice are built into the language itself. This approach was explicitly chosen 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 behavior. This is certainly something which can be picked up when using Go for extended periods of time. But for newcomers especially it can cause some confusion and while the new variant with `new` works it is unclear whether this is the idiomatic way to create growing slices.

From a **performance** standpoint *streets4go* falls short compared to the other implementations as Table 4.3 highlights. 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 so 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 of the dependency 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 adjacency 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 Listings 4.13 and 4.14 show the id to index conversion in Go and Rust highlighting the differences.

```
1 func (g *Graph) AddEdge(n1, n2 int64, e *Edge) {
2     // [..]
3     // link up adjecents
4     n1_idx, n2_idx := g.nodeIdx[n1], g.nodeIdx[n2]
5     // [..]
6 }
```

Listing 4.13: Map lookup in Go

```rust
pub fn add_edge(&mut self, n1: i64, n2: i64, e: Edge) {
    // [..]
    // link up adjecents
    let n1_idx = self.nodes_idx.get(&n1).unwrap();
    let n2_idx = self.nodes_idx.get(&n2).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 it 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 example of how a **sophisticated type system** can **prevent potential errors** through descriptive types.

The statistics for this milestone are once again very promising for Rust as Table 4.3 proves. With the lowest **SLOC count** as well as **development time** it still remains competitive with the **execution performance** of C. The **allocation count** also hints that Rust's vectors probably have a bigger reallocation factor 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 type systems in particular. They can prevent bugs which would otherwise require intensive testing to be even noticed in the first place. Rust shines in this discipline. Its type system not only pretty much guarantees **memory safety** but also allows libraries to **encode usage semantics into function signatures** preventing some cases of possible misuse. 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 which took only about half as long to implement. While Go is certainly not as fast as the other two languages as of yet it is a very comfortable language and allows for some decent **productivity gains**.

4.4. Benchmarking Graph Performance

In this milestone **runtime performance** came back into the main focus. Since the algorithms at this point were proven to work correctly they 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 problem was 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 this metadata 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)[16]	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 was chosen and implemented.

Self-reallocating arrays are **not part** of the C language or **standard library** and so 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.

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

The **performance statistics** from this phase contain the first real surprise. As Table 4.4 clearly shows *streets4C* does not have the lowest **execution time** and 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 the author's limited experience with the language. However this might very well be reflective of a scientist with similarly limited programming skills. Considering the **development time** and **SLOC** count the result is even more alarming. The redeeming factor for C 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 and has to be taken into account.

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 temporarily 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[17] 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.

While the Listing 4.15 is a bit small in width the advantages should be clearly visible. The multiline assignments 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.

```
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)
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)
```

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

```
    c := 2 * math.Asin(math.Sqrt(a))
    return 6367000 * c // distance in m
}
```

Listing 4.15: Haversine formula in Go

In the **performance** comparison shown in Table 4.4 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 *allocation heavy 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. During the implementation though there was a small problem caused by Rust's ownership model. It essentially prevented the application from freely moving around the structures generated from the input data as ownership of these instances had to be passed to the graph correctly.

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 ownership of the edge instance could be transferred to the graph while the two 64-bit integers **`n1`** and **`n2`** are implicitly copied.[18] Listing 4.16 illustrates this process.

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

    // [.. parse nodes and edges ..]
```

[18] This happens because **`i64`**, like all primitives, implements the **`Copy`** trait allowing it be copied instead of moved

```
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 (see line 11) 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` keyword 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 type system. There are a lot of sigils which sometimes even have double meanings. Ultimately though it has to be noted that the compiler messages are very helpful even in these seemingly complex situations. They are a big reason why the development time for this milestone is the lowest despite the difficulties with the ownership. It takes some time to be able to parse through the sheer amount of information the compiler prints in case of a problem but once that is done the messages help to identify all kinds of errors rather quickly.

The **performance** comparison was specifically interesting in this phase because Rust was actually ahead of C for the first time (see Table 4.4). 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 straight up beats Go and C in three important categories which is impressive.

4.4.4. Comparison

This milestone really highlighted the power of Rust. On the one hand it increases **productivity** through high level abstractions saving valuable **development time**. On the other hand these abstractions produce very efficient machine code even beating C in **execution time** by about 33%. Unfortunately they also consume about twice the **amount of memory**. Since the structures contain nearly the same fields it is unclear

what causes this big difference exactly but considering Rust's current development state these results are promising and might even improve with optimizations when the language reaches a stable release.

Although the results for the C implementation as seen in Table 4.4 are certainly still optimizable and may be the consequence of a subotimal architecture, the statistics undeniably show that it is *possible* to write **faster applications in less time** using Rust instead of C with moderate experience in both languages. Go also produces decent results especially in the **productivity** sections. Only slightly slower than C *streets4Go* took vastly less **development time** and consumes less **memory** than the Rust version.

4.5. Benchmarking Parallel Execution

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 to note 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 in an idiomatic way.

The resulting code from this milestone was later benchmarked on a cluster computer of the research group. To lower binary size all applications were stripped of unneeded code for this final phase. This mostly meant removing the incremental parts from previous milestone as they were not required for the final versions. The result are two SLOC count categories for this phase. The first is the logical continuation from the previous phases counting all lines written up to this point while the second represents the final amount of lines required after all old code had been removed and the benchmark function was inlined into the main function. This number really shows how much lines were needed to successfully implement the application in each language.

	C	Go	Rust
SLOC (incl. previous phases)	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 (4 threads) (MB)[19]	1213	1693	2501

Table 4.5.: Milestone 5: Parallel benchmark

The **execution time** and **memory usage** were measured on the development laptop with 4 threads to have a quick overview about the general performance of each application. Although it is not representative of the final results the tracked statistics were the result of the same code that was later extensively run on the high performance machine.

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 calculating offsets manually. To have finer grained control over the threads the seconds strategy was favored. Listing 4.17 shows the resulting **`omp parallel`** block with some print statements removed for brevity.

```
1  static void benchmark_osm(char* path)
2  {
3      // [.. parse input data ..]
4
5      // Setup OpenMP
6      omp_set_dynamic(0);
7      omp_set_num_threads(NUM_THREADS);
8
9      #pragma omp parallel
10     {
11         // Calculate limits for this thread
12         int id = omp_get_thread_num();
13         int first = id * nodes_per_thread;
14         int last = first + nodes_per_thread;
15
16         dgraph dg = new_dgraph(g);
17         for (int i = first; i < last; i++)
18         {
19             dijkstra(dg, i); // Calculate shortest paths
20         }
21         free_dgraph(dg);
22     }
23
24     // [.. final cleanup ..]
25 }
```

Listing 4.17: Parallelization with OpenMP

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

The required changes were less than expected which proves why OpenMP is a popular choice for thread based parallelization in C. The existing loop was surrounded with the correct pragma and of course each thread has to calculate new limits for the loop based on its id. Luckily the **`dgraph`** structure was designed with concurrency in mind and as such can be instantiated independently in each thread. The **`dijkstra`** function is then called in a loop to calculate shortest paths for all nodes this thread was assigned to process and results are discarded immediately.[20]

The idiomatic way in C to save and use the resulting data would be writing to a shared array. Assuming the nodes processed by each thread are disjoint the array access can be safely shared for writing. While the benchmarked implementation actually does not guarantee this for all amounts of threads (because of remainders in certain configurations) it could easily be changed to use this pattern. It is also possible to create a result array in each thread and then later manually reduce it to a global one via an **`omp critical`** section. While this approach would theoretically support overlapping node lists in different threads it is not very useful in this context because the affected values would just be overwritten. Also the use of a critical block obviously impacts performance quite heavily because threads might have to wait for each other to finish the global update.

Table 4.5 shows once again C as only second fastest implementation. While these results were only gathered from a single run they still show an ongoing trend from the previous phase. The short **development time** can be attributed to OpenMP's matureness as a framework reducing the changes to a bare minimum. Only beaten by Go (which basically chosen for a promise of simple concurrency) this result is very positive and is a welcome contrasts to previous milestones.

4.5.2. Go

streets4Go obviously used goroutines for concurrent execution. As predicted the changes to parallelize the application were minimal and simple. A for loop was added to start up the required amount of goroutines which then calculate their limits based on the outer loop index and start executing the algorithm. Listing 4.18 shows the relevant lines responsible for concurrent execution without print statements.

```
1 func benchmarkOsm(path string) {
2     // [.. parse input data ..]
3
4     // To wait goroutines later on
5     var wg sync.WaitGroup
6     runtime.GOMAXPROCS(NUM_THREADS)
7
```

[20] This is not directly clear from the code but the calculating function only stores results in the struct and since the loop instantly continues to the next node they are overwritten during the next call

```
8       for i := 0; i < NUM_THREADS; i++ {
9           wg.Add(1)   // Increase WaitGroup counter
10          go func(id int) {
11              defer wg.Done() // Decrease counter when the
                    ↪ goroutine exits
12
13              dg := FromGraph(g)
14              first := id * nodes_per_thread
15              last := first + nodes_per_thread
16
17              for n := first; n < last; n++ {
18                  dg.Dijkstra(n)
19              }
20          }(i)
21      }
22
23      wg.Wait()
24  }
```

Listing 4.18: Parallelization with goroutines

The existing for loop was surrounded by a second one starting up goroutines to work concurrently. In typical fashion these goroutines get their function body passed as an anonymous function with the loop argument passed as a parameter. While it would be possible to extract the code into a separate function and call it with the **go** keyword this solution is much easier to understand and really highlights how anonymous functionc can enhance code readability. To prevent the parent thread from returning prematurely a **WaitGroup** is added which effectively joins all running goroutines before exiting.

The common way in Go to share memory is by communicating over channels. As the data from the graph was only read it was easier to just share the memory read-only but for concurrent writes channels are definitely preferred. The actual implementation would create a channel pair in the parent thread and then pass a copy of the transmitting end to all goroutines. When a calculation is finished the result is written to the channel and the loop continues. The parent thread can wait to retrieve results from the receiving end of the channel and store or directly use them for further calculations.

Considering **performance** Table 4.5 once again places Go slightly behind C in the with lower **memory usage** than Rust. Considering the low amount of work required to achieve this runtime it is safe to say Go fulfilled the promise of *simple concurrency*.

4.5.3. Rust

While both other languages had the advantage of having a framework of some sort at their disposal the Rust solution creates raw threads to parallelize the execution. This

shows in the **development time** which is more than three times as high as the time spent on the other implementations. Listing 4.19 below includes the relevant parts from the parallel version of the benchmark function.

```
1  fn benchmark_osm(osm_path: &Path) {
2      // [.. parse input data ..]
3      let graph = &g; // immutable reference to copy into the
           ↪ closures below
4
5      let num_nodes = NUM_NODES / NUM_THREADS;
6      let mut guards = Vec::with_capacity(NUM_THREADS);
7
8      // Spawn NUM_THREADS amount of worker threads
9      for id in 0..NUM_THREADS {
10         guards.push(thread::scoped(move || {
11             let first = id * num_nodes;
12             let last = first + num_nodes;
13             let mut dg = DijkstraGraph::from_graph(graph);
14
15             for n in first..last {
16                 dg.dijkstra(n);
17             }
18         }));
19     }
20
21     // Join all threads before exiting
22     for g in guards {
23         g.join();
24     }
25 }
```

Listing 4.19: Parallelization with threads in Rust

Similar to the Go variant an outer loop is added responsible for spawning child threads which calculate shortest paths concurrently. Again comparable to Go each thread receives its work through an anonymous function. As the keyword **`move`** suggests all variables used inside this function are essentially moved inside the thread and cannot be used after that. Luckily references and primitives implement the **`Copy`** which allows them to be copied instead. To be able to copy a reference to the graph data into the thread stack it has to be explicitly bound (line 3). Otherwise the use of **`&g`** would cause a move of the graph itself. After the threads have been spawned the main thread calls join on all **`JoinGuard`**s to ensure the child threads do not outlive it. This strategy is similar to Go's **`WaitGroup`** but in Rust the **`thread::scoped`** function returns a JoinGuard instance. It is therefore not as optional as the WaitGroup since the caller has to handle the return

value in some way. This once again shows Rust's power to **encode usage semantics into return values**.

As noted above the Rust solution directly uses threads. The language provides the **tools to build efficient concurrency abstractions** but does not offer a complete framework like OpenMP. Since the language is still young these frameworks will most likely be developed by the community in after a stable release. Of course C also provides raw threading capabilities but in Go concurrency is completely abstracted into goroutines and the programmer is forced to use those.

There are multiple ways to retrieve calculation results. One choice are channels which Rust's standard library also contains. In this case the solution would be identical to the Go strategy effectively cloning the transmitting end of the channel for each thread. Another option is shared writing supported by locks or mutexes both of which are also part of the standard library. Since scoped threads (which are used in this implementation) are able to share stack data with their owning thread it is possible to write to a shared variable when locking accordingly. This approach would be comparable to the first possibility proposed for C above with the additional safety benefits of locking.

Like the previous phase the parallelized Rust variant is the fastest among the compared implementations (see Table 4.5). And once again this **performance** comes at the price of high **memory consumption**. However it is interesting to note that the **memory footprint** only increased slightly when using four threads. This hints at a suboptimal data layout in the immutable structures of the application which are only allocated once even in parallel scenarios.

4.5.4. Comparison

This final milestone offered some insight about the **difficulty of parallelization** in the three languages. As expected Go shines here with the lowest **development time** proving goroutines are a **simple yet powerful abstraction for concurrency**. C surprisingly also took very little effort thanks to OpenMP. Combined with the language inherent speed this makes for a very good result only beaten by Rust in **execution time** which is again in a leading position. However the **memory footprint** is also very high roughly doubling C in this aspect. Although these are only intermediate results they show a trend with Rust beeing ahead in **both performance and productivity**.

4.6. Preparing Execution on the High Performance Machine

The results for the next chapter were gathered from a machine provided by the research group. Since access to this machine was only available via Secure Shell (SSH) and had different libraries installed the code of the various implementations had to be prepared

to be able to perform the benchmarks. Although no numeric statistics were tracked they are briefly described here as they were certainly a part of the development process.

C

When first started on the cluster the binary compiled on the development laptop did not run correctly. The first problem was the version of the libgomp *streets4C* was linked against. This was the result of a newer compiler version on the development laptop compared to the remote machine which uses a Long Term Support (LTS) distribution and therefore generally older software. The solution was to simply compile the application on the remote machine from sources.

Unfortunately not all sources were easily available and as a consequence two libraries were only linked in binary form. This probably introduced a small performance penalty as the libraries were not fully optimized but due to time restrictions this approach had to be chosen. The two libraries in question were the Protocol Buffers C runtime `libprotobuf-c.a` and a custom fork of the OSM library `libosmpbf.a`[21]. These two files were compiled on the development laptop and then copied to the cluster. A better tool for **dependency management** would have eliminated this problem.

Go

When preparing the Go implementation for the benchmarks it came in handy the a Go executable is always linked statically **without any further dependencies**. Since the remote machine and the development shared the same architecture the process was as simple as copying the clustern executing it. Considering that the binary was never recompiled and could not take advantage of special processor specific features the **performance** results from the next chapter are very promising.

Rust

Porting the Rust version posed an interesting challenge. All executable who link against the Rust standard library require at least a C standard library installed.[22] While there is a minimum required version the compiler takes advantage of newer versions installed on the compiling system and in turn raises the requirement. This means an executable linking against a newer version of `glibc` is not executable with older versions. But the same code can be compiled on the target system without problems. This strategy was employed on the remote machine. It is worth mentioning that the **excellent toolchain** around *cargo* was very helpful in the process. In contrast to the C variant where some

[21] Located at `https://github.com/MrFloya/libosmpbf`

[22] Currently onle the GNU `glibc` is supported but that will change. See `https://github.com/rust-lang/rfcs/issues/625` for further information

libraries were not available and had to copied from the development laptop *cargo* was able to **fetch all dependencies and build them without any problems**.

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. Both areas are evaluated based on results from the previous chapter.

All data in this chapter was gathered from a high performance computer by courtesy of the research group Scientific Computing. The machine has access to four 12-core processors and 128 GB of memory. It is therefore ideal to compare shared memory performance on a large scale.

5.1. Performance

In high-performance computing the most important criteria when evaluating a language is **performance**. The important statistic that was tracked to compare performance is **execution time**. The benchmarks that were performed on the development laptop also roughly measured **memory usage** but that proved difficult to automate on the remote machine. It is therefore not directly included in this final evaluation. Table 5.1 shows the benchmark results in varying concurrency scenarios from single threaded execution up to 48 calculating in parallel.

threads/goroutines	C	Go	Rust
1	21:51:18	16:48:19	14:15:06
2	12:29:56	10:21:36	09:12:47
4	07:16:34	05:58:35	05:09:56
8	04:13:04	03:01:54	02:49:35
12	03:17:28	02:06:08	01:55:33
24	02:06:08	01:13:47	01:03:34
48	01:21:58	00:53:54	00:44:54

Table 5.1.: Execution time of the final applications (100K nodes)

These results already contain the first real surprise. C was chosen as a comparative baseline, since it is one of the two big programming languages in HPC and is the slowest

of the three compared languages in all configurations. In contrast the preliminary benchmarks on the development laptop showed C at least in second place in the performance comparison. As briefly mentioned in the previous chapter this performance regression might have been caused by the two unoptimized libraries that were compiled on the development laptop and copied to the cluster. However this shows that C is still very much compiler and machine dependent.

In contrast the Go binary that was also compiled on the development laptop was executed without any changes on the target machine and shows great result even reaching **similar performance** to Rust in the **high concurrency configurations**. This shows that a garbage collected language is not immediately unsuitable for use in HPC. Combined with the portability caused by full static linking Go might very well be suited for cluster computations on nodes with a minimum of system libraries installed.

Finally Rust demonstrates that it might be a **competent successor** to C in HPC. It is the **fastest language** out of the compared three across all scenarios while providing additional **memory safety** through its unique type system. It prevented multiple errors from compiling in the Rust implementation throughout the whole development process. On one occasion it even revealed an error which had gone unnoticed in both C and Go. This really highlights how static analysis can provide **safety without sacrificing performance**.

Another important statistic to compare is the **parallel speedup**. A slow **execution time** alone does not mean a language is completely unfit for HPC because the implementation might simply be flawed to begin with. If this is the case the application can still offer **above average speedups** making it viable for **high concurrency scenarios**. Table 5.2 lists the achieved speedup for each language in the same configurations as above.

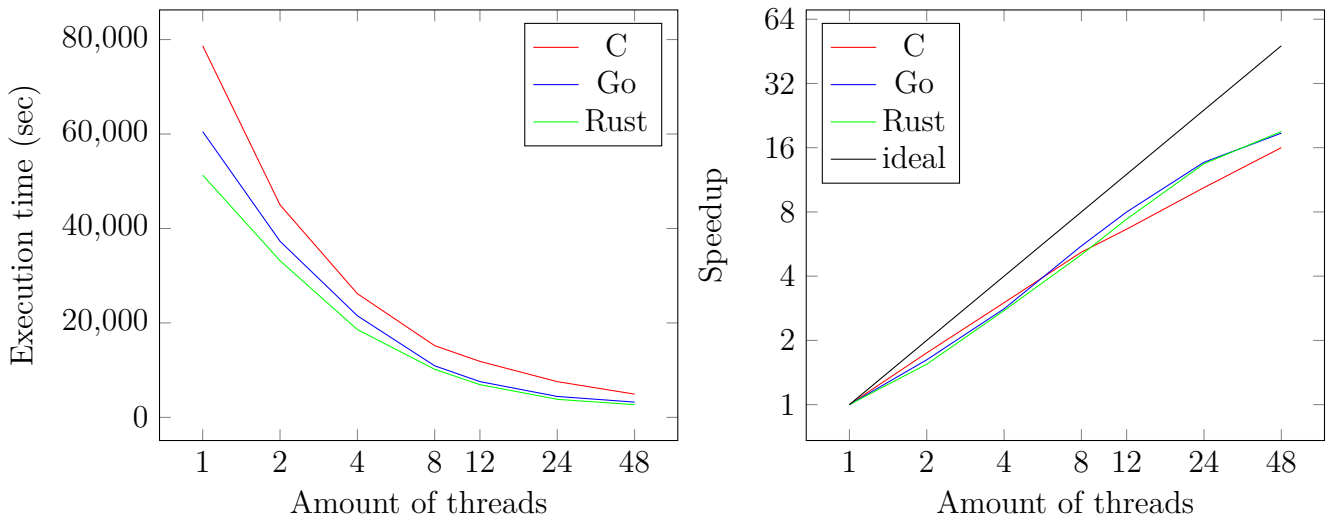
threads/goroutines	C	Go	Rust
1	1.0000	1.0000	1.0000
2	1.7486	1.6221	1.5469
4	3.0037	2.8119	2.7590
8	5.1816	5.5432	5.0424
12	6.6406	7.9941	7.4003
24	10.3961	13.6659	13.4520
48	15.9980	18.7072	19.0445

Table 5.2.: Parallel speedup of the final applications (100K nodes)

Again the results are interesting for multiple reasons. C scales very well up to four threads but falls off quite heavily. Rust and Go are evenly matched with Go scaling better up to the final benchmark with 48 threads where it gets beaten slightly by Rust. It would be interesting to see the trend continue here but unfortunately the target machine *only* offered 48 logical cores. C' **strong scaling in the lower thread counts** shows

OpenMP's efficiency in generating threaded code for **common desktop scenarios**. In the **high concurrency configurations** though the scaling diminishes resulting in a big discrepancy of about 3 (~13%). For these use cases it might be worthwhile to implement **custom parallelization with *pthreads***.[1] This approach will most likely result in much higher **development time** but might yield better **performance results** for higher thread counts.

Rust and Go scale both comparably well as Table 5.2 shows. Although the **final speedup** of about 19 is extremly impressive for 48 threads it is still a serious improvement about the serial version. It is also important to note that the threads share the work with statically which means there is no load balancing once the threads are started. This results in some threads **exiting early** effectively **reducing the speedup**. To solve this the work could be **dynamically distributed** for example through a queue like construct which threads use to retrieve new tasks. A possible implementation could be channel based as both Go and Rust offer those as part of the standard library.



(a) Execution time

(b) Parallel speedup

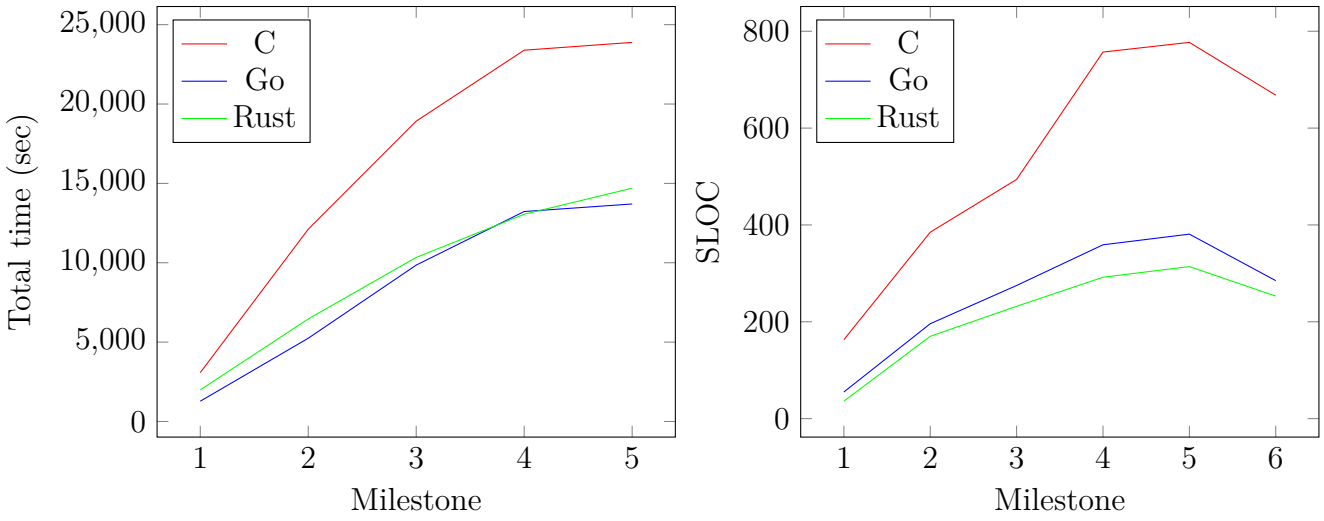
Figure 5.1.: Performance metrics across the various milestones

Figure 5.1 compares the **execution time** and **speedup side by side**. As expected no implementation reaches linear speedup but there are large differences between the languages. C offers a nearly constant growth with the least amount variation but has the lowest scaling overall. Rust and Go on the other hand fluctuate in the lower thread counts and achieve similar goo results in the **high concurrency configurations**. As mentioned multiple times Rust offers the best **overall performance** in both **execution time** and **parallel speedup**.

[1] `http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/pthread.h.html`

5.2. Productivity and additional Metrics

Next to **performance** the second main criterion evaluated in this thesis is **developer productivity**. The two numeric statistics tracked to compare this category are **SLOC count** and **development time**. While the **SLOC count** is certainly not the best code quality metric it allows for some basic conclusions. Less lines can contain potentially less errors, lowering maintenance costs, and *should* take less **time to develop**. This is obviously not always the case but Figure 5.2 confirm this correlation for the evaluated implementations.



(a) Development time

(b) SLOC counts

Figure 5.2.: Productivity metrics across the various milestones

Comparing **developer productivity** C comes in last by a large margin. The high **development time** can be traced back to the **manual implementation of common data structures** and the **high amount of memory and type related errors** encountered during the development. This naturally lead to a much higher **SLOC count** which was increased further by code duplication caused by the conventional header files. It is important to note here that the **SLOC count** for C is also partially caused by the dominant bracing style. While Go forces the programmer to set all curly braces on the same line as the preceding statement the C implementations were written in a different style always placing curly braces on new lines. Rust follows the Go rules but only encourages them as a convention allowing for other styles to be used. This in turn produces some extra lines on the C side while Rust and Go remain unaffected. However the difference is still way too significant to be only attributed to the style choice.

Rust and Go on the other hand both allow for substantial **productivity gains** as Figure 5.2 clearly shows. There seems to be just a little **tradeoff** between the two tracked criteria with Rust requiring slightly less **lines of code** while Go is a little **faster**

to develop in. Although the results are so close together that it does not matter a lot when compared to C. While these changes are certainly largely caused by the languages themselves especially the lower **development time** is also caused by the **superior tooling**. Both Go and Rust offer **excellent tool support** for **dependency management** and other parts of the build process like **testing**. Especially for compilations on foreign machines this is invaluable because the application is mostly **independent from systemwide installed libraries**.

All these results reinforce the inital motivation from the Introduction for a new successor to C in the context of high-performance computing. Considering **productivity only**, both Go and Rust offer **excellent advantages** over C.

6. Conclusion

In this final chapter a short summary is given and the thesis concludes with a brief description of possible improvements and future work.

6.1. Summary

The results from the previous chapter indicate that C might not be the best choice for high performance applications anymore. While C has been known to be a **lesser productive language** (mostly but not exclusively due to **manual memory management**) there was usually a **tradeoff between performance and productivity**. Since in HPC **low execution times are the highest goal** the **lower productivity** was accepted to eventually gain superior speed. This thesis proves that **this tradeoff does not neccessarily have to exist anymore** as new languages provide **higher productivity** paired with **equal or better performance** characteristics. While the produced results are certainly not comprehensive enough to completely invalidate C as *the* high performance language to use they still show that there are possible successors with Go and Rust.

Go as a garbage collected languages allows the programmer to **forget about memory management**. This can be helpful especially for newcomers because they are able to focus on the core functionality. Similarly scientists of other fields are able to just **focus on their research** and expressing it in code without having to think about **allocations and memory freeing**. **Concurrency is an built into the language** and goroutines allow for simple parallelization in shared memory scenarios. All this is achieved **without the typical loss of performance** as the **execution time** statistics show.

Although Go's results are impressive in itself Rust's benchmarks look even more promising. As the **fastest among the three evaluated languages** it is already worth considering for adoption but **combined with the equally good productivity metrics Rust is a serious contender for the next big language in high-performance computing**. The **tooling support** is already quite good and improving every day and the **standard library provides a complete toolbox for various kind of concurrency abstractions**. Unfortunately it is currently also **limited to shared memory parallelization** (similar to Go) but this might change afte the language gets a first stable release.

6.2. 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 areas and concludes this thesis.

General Code Quality

As mentioned a few times in the thesis the quality of the implementations might not be optimal especially in the case *streets4C*. This can mostly be attributed to the author's lack of equal experience in the three evaluated languages. While this is to a point intentional, as scientists might not be proficient in these languages either, the implementations could be reviewed by language experts to really demonstrate the absolute **highest possible performance**.

Limited Benchmark Configurations

Although the benchmark results presented in the previous chapter allow for some decent conclusions, the configurations could have been varied some more. Especially the **problem size**, which was fixed at 100K nodes, **could have been varied** for multiple thread amounts. Another possible comparison could include **different compilers for C and Go** (Rust currently only has one reference implementation) or multiple compiler and library versions revealing possible regressions across versions.

Limitation to shared Memory

While thread based concurrency is certainly an important aspect in HPC the **dominant model** is **distributed memory with communication 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).

[dev] OpenBSD developers. *OpenSSH Project Goals*. URL: http://www.openssh.com/goals.html (visited on 22. 04. 2015).

[Dow11] Malcolm Dowse. *Erlang and First-Person Shooters*. June 2011. URL: http://www.erlang-factory.com/upload/presentations/395/ErlangandFirst-PersonShooters.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.

[Ini] The Open Source Initiative. *About the Open Source Initiative*. URL: http://opensource.org/about (visited on 25. 04. 2015).

[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*. URL: https://golang.org/ (visited on 27. 04. 2015).

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

[maic] 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

API

Application Programming Interface

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 [maib]

HPC

high-performance computing

intrinsic

An intrinsic function is a function in a programming language which is handled specifically by the compiler. This is often used to optimize common operations for the target processor.

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

Long Term Support

Mostly applied to Linux distributions, a Long Time Support version is a specific release with an extended support cycle. This means no new feaures get added but security patches are backported. LTS versions therefore offer better stability and security for mission-critical systems.

loop unrolling

Loop unrolling or loop unwinding is a common optimization used by compilers trading binary-size for speed. A loop gets transformed into the seperate instructions to avoid the overhead of the loop control instructions.

LTS

Long Term Support

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

Open Source Initiative

The Open Source Initiative is a non-profit organization based in California which was "formed to educate about and advocate for the benefits of open source and to build bridges among different constituencies in the open source community" [Ini].

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].

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

OSI

Open Source Initiative

OSM

OpenStreetMap

Protocol Buffers

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

Secure Shell

Secure Shell is an encrypted network protocol aimed to replace clear text password transmitting protocols like telnet, rlogin and ftp. [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

source lines of code

A software metric counting the number of lines in a program's source code. Often used to roughly estimate developer productivity and maintainability

SSH

Secure Shell

SSSP

Single Source Shortest Path

TDD

test-driven development

XML

Extensible Markup Language

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:               1252.890
CPU max MHz:           2900,0000
CPU min MHz:           800,0000
BogoMIPS:              3992.48
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              6144K
NUMA node0 CPU(s):     0-7
```

Listing B.2: Output of `lscpu`

Cluster

```
Linux magny1 3.8.0-44-generic #66~precise1-Ubuntu SMP Tue
  ↪ Jul 15 04:01:04 UTC 2014 x86_64 x86_64 x86_64 GNU/Linux
```

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

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                48
On-line CPU(s) list:   0-47
Thread(s) per core:    1
Core(s) per socket:    12
Socket(s):             4
NUMA node(s):          8
Vendor ID:             AuthenticAMD
CPU family:            16
Model:                 9
Stepping:              1
CPU MHz:               800.000
BogoMIPS:              3800.11
Virtualization:        AMD-V
L1d cache:             64K
L1i cache:             64K
L2 cache:              512K
L3 cache:              5118K
NUMA node0 CPU(s):     0-5
NUMA node1 CPU(s):     6-11
NUMA node2 CPU(s):     12-17
NUMA node3 CPU(s):     18-23
NUMA node4 CPU(s):     24-29
NUMA node5 CPU(s):     30-35
NUMA node6 CPU(s):     36-41
NUMA node7 CPU(s):     42-47
```

Listing B.4: Output of `lscpu`

C. Software versions

These are the compiler and toolchain versions which were used to develop and compile all code in this thesis.

```
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 (da623844a 2015-04-25) (built
   ↪ 2015-04-26)
```

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

```
cargo 0.2.0-nightly (dac600c 2015-04-22) (built 2015-04-24)
```

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

D. Final notes

All source code is available at `https://github.com/mrfloya/thesis-ba` under various Open Source Initiative (OSI) approved licenses. The versions containing all code from intermediate milestones are in a separate branch called **`incremental`**. The **`master`** branch only consists of the final variants that were used in the benchmarks on the cluster.

The Rust implementation was compiled with the version shown in Listing C.3. Unfortunately it as not possible to compile the code with the current beta version at the time of this writing (1.0.0-beta.2). Because of this the application will most likely not compile with 1.0.0 either. The code in the repository mentioned above will be updated as soon it compiles with a stable Rust release.

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 28.04.2015 .....