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

— Bachelor Thesis —

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

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

Hamburg, February 20, 2015

Abstract

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

Table of Contents

1. Introduction

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

1.1. Motivation

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

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

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

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

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

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

1.2. Goals of this thesis

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

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

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

2. State of the art

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

- state of C and Fortran (section name?)

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

2.1. Weaknesses of C and Fortran

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

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

2.2. Candidates

This section aims to provide a rough overview of possible candidates that were considered for further evaluation in this thesis.

Python

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

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

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

Listing 2.1: FizzBuzz in Pyhon 3.4

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

Unfortunately dynamic typing and automatic memory management come at a rather high price. The speed of raw numeric algorithms written in plain Python is almost always orders of magnitude slower than implementations in C or Fortran. As a consequence nearly all of the mentioned libraries implement the critical routines in C and focus in optimizing the interop (// wording) experience. 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 it's equivalent written in a statically typed language and the more terseness

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

Python loses because of this the less desireable it becomes to use in high-performance computing since one could just fall back to C for a similar experience.

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

Erlang

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

- brief history?

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

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

Go

Go is a relatively new programming language which focusses on simplicity and clarity while not sacrificing too much performance. Initially developed by Google it aims to "make it easy to build simple, reliable and efficient software" (//cite). It is statically typed, offers a garbage collector, basic type inference and a large standard library. Go's syntax is loosely inspired by C but made some major changes like removing the mandatory semicolon at the end of commands and changing the order of types and identifiers. It was chosen as a candidate because it provides simple concurrency primitives as part of the language (so called *goroutines*) while having a familiar syntax and beeing reasonably performant. [Dox12] It also compiles to native code without external dependencies which makes it usable on cluster computers without many additional libraries installed.

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

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

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

Listing 2.2: Go concurrency example

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

All these statistics show the core focus of the language // simplicity and avoidance of boilerplate

- brief history

- Prediction implementation - A bit of syntax weirdness - Relatively quick PoC with decent concurrency aspects - Some fixing/optimization afterwards regarding common

concurrency errors -> More time spent after initial PoC but less than in C

Rust

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

Out of the here discussed languages Rust is closest to C while attempting to fix common mistakes made possible by it's loose standard allowing undefined behaviour. (//wording?) Memory safety is enforced through a very sophisticated model of ownership. It is based on common concepts which are already employed on concurrent applications. The basic rule is that every piece of allocated memory is *owned* by *one* entity in the program (typically a variable) and only the owner can change the contents of that memory. To allow more complex algorithms values can be borrowed

- Key features

- Up/Downside

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

Comparison

Rust	Python	Erlang	Go
Execution model compiled to native code	interpreted	compiled to bytecode	compiled
Advantages adv rust	low barrier of entry	builtin (lockfree) concurrency support	adv go
Disadvantages disadv rust	speed	obscure syntax	mandator
Relative speed speed rust	slow to average	average to fast	speed go

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

3. Approach

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

3.1. Overview: Streets4MPI

As stated in section 1.2 the language evaluation is based on a reimplementation of an existing parallel program originally written in Python. The application in question is *streets4MPI* - a traffic simulation using OpenStreetMap files as input and calculating shortest routes via Dijkstra's algorithm. Streets4MPI was implemented in scope of the module "Parallel Programming Project" in Spring 2012. It was written by Julian Fietkau and Joachim Nitschke and makes heavy use of the various libraries of the Python ecosystem. The original project contained the main simulation as well as a visualization script. [FN12, p. 3] For the purpose of this thesis the visualization was omitted and the evaluation is only based on software quality metrics as well as the result's correctness. //TODO: really cite page?

3.2. Differences to the base implementation

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

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

Input format

The original implementation used the somewhat dated OpenStreetMap XML format [1] as input which is parsed by *imposm.parser* [2]. Streets4MPI then builds a directed graph

[1] `http://wiki.openstreetmap.org/wiki/OSM_XML`
[2] `http://imposm.org/docs/imposm.parser/latest/`

via the *python-graph* [3] library to base the simulation on.[FN12]

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

Simulation

Streets4MPI performs

Concurrency

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

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

3.3. Implementation process

[3] `https://code.google.com/p/python-graph/`
[4] `https://developers.google.com/protocol-buffers/`
[5] `http://www.openmp.org`

4. Implementation

In diesem Kapitel ...

5. Evaluation

In diesem Kapitel ...

6. Conclusion

In diesem Kapitel ...

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

Bibliography

[Ces09] Francesco Cesarini. *Erlang programming. A Concurrent Approach to Software Development*. Beijing: O'Reilly, June 2009. ISBN: 978-0-5965-1818-9.

[Dox12] Caleb Doxsey. *An Introduction to Programming in Go*. Lexington, KY: CreateSpace, 2012. ISBN: 978-1-4783-5582-3 (cit. on p. 8).

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

[FN12] Julian Fietkau and Joachim Nitschke. *Project Report: Streets4MPI*. Hamburg, May 21, 2012. URL: `http://wr.informatik.uni-hamburg.de/_media/research/labs/2012/2012-05-julian_fietkau_joachim_nitschke-streets4mpi-report.pdf` (visited on 18. 02. 2015) (cit. on pp. 11, 12).

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

[Lub14] Bill Lubanovic. *Introducing Python. Modern Computing in Simple Packages*. 1st ed. Beijing: O'Reilly Media, Inc., Nov. 26, 2014. ISBN: 978-1-4493-5936-2 (cit. on p. 7).

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

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

[Pro] The OpenStreetMap Project. *OpenStreeMap Wiki - "PBF Format"*. URL: `http://wiki.openstreetmap.org/wiki/PBF_Format` (visited on 18. 02. 2015) (cit. on p. 12).

List of Figures

List of Tables

List of Listings

Appendices

A. Appendix

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

Erklärung

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

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

Hamburg, den 26.11.2014 .....