GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Seminar Report

Go in High Performance Computing

Valerius Mattfeld

MatrNr: 11580056

Supervisor: Jonathan Decker

Georg-August-Universität Göttingen
Institute of Computer Science

April 22, 2024

Abstract

Go is a popular, statically typed and compiled language with an easy to learn syntax and unique concurrency model. Therefore, it could provide some beneficial contributions to the High-Performance Computing (HPC) ecosystem. This report investigates the use cases and practicality of Go in modern cloud computing in regard to serverless functions, containerization, orchestration and HPC. The challenge resides in providing Go with a robust Message Passing Interface (MPI) library, which effectively manages the language's typical memory model and garbage collection, as it gets in the way of harnessing Go's potential in HPC. Currently, research and development in the HPC application area is stagnated and fully focused on cloud infrastructure. The HPC ecosystem could benefit from a language with excellent concurrency support on worker nodes. A wrapper library with an accessible API, which manages the language given caveats, could unlock Go's potential in HPC and improve the developer experience.

Declaration on the use of ChatGPT and comparable tools in the context of examinations

In this work I have used ChatGPT or another AI as follows:

- ☐ Not at all
- ☐ During brainstorming
- ☐ When creating the outline
- ☐ To write individual passages, altogether to the extent of 0% of the entire text
- ☐ For the development of software source texts
- ☐ For optimizing or restructuring software source texts
- ☐ For proofreading or optimizing
- ☑ Further, namely: The usage of semanticscolar.org[1] for searching and discovering scientific literature. LanguageTool.org[2] for finding typos and incorrect spelling.

I hereby declare that I have stated all uses completely.
Missing or incorrect information will be considered as an attempt to cheat.

[1] `https://www.semanticscholar.org/`
[2] `https://languagetool.org/`

Contents

List of Tables

List of Figures

List of Listings

List of Abbreviations

API Application Programming Interface

CUDA Compute Unified Device Architecture

FFI Foreign-Function Interface

FaaS Functions-as-a-Service

HPC High-Performance Computing

HTTP Hypertext Transfer Protocol

JSON JavaScript Object Notation

MPI Message Passing Interface

SQL Structured Query Language

XML Extended Markup Language

1 Introduction

This report explores the Go programming language over various fields, starting with its core benefits, its use in cloud computing, and its role in HPC. It is a statically compiled language with a unique concurrency and memory model.[23c] Additionally, Go embodies a simple syntax alongside an in-build garbage collector.[23c] This report investigates the language's potential for HPC applications.

The report introduces the Go programming language with its origins and core traits, and benefits in section 3. Then continue to section 4 with Go's relevance in container virtualization and usage in HPC. After that, the report discusses the suitability of Go in HPC as an application language along with its available MPI wrappers and the implied challenges in section 5-6.

Lastly, in section 7 the report explores the usage of Go-based serverless function frameworks in combination with Kubernetes, which attempt to process HPC workloads in a modern fashion.

2 Background and Motivation

When it comes to the usage of Go in HPC, and, by extension, MPI applications, the available source material is sparsely populated. Aside from one source, namely [BW14], no serious attempts or utilization of Go with MPI beside some unmaintained and outdated OpenMPI [ope] wrapper libraries are present on the internet or research databases.

Since Go is an accessible, statically typed, and compiled language with a rich standard library with a modern concurrency model, it lately became quite popular among developers. [Cas22; 23c]

Having a language capable of an easy-to-use concurrency model could benefit the development of faster, and single-node parallelism-focused HPC applications. With that, the question arises, why Go does have a limited popularity when it comes to MPI related runtime applications and research.

Furthermore, when viable, Go could enable novice-programmers to find a stepping-stone into HPC application development without having advanced knowledge of lower-level programming languages like C or C++.

Moreover, with the push of moving HPC workloads into Kubernetes clusters, [PA22; LG22] and the rising demand for serverless functions, [Sch+21] Go could offer an entry point for developers and researchers planning to contribute to this kind of projects.

3 Benefits of using Go in Cloud Environments

This section briefly elaborates on the origins of the Go programming language. After that, we briefly examine the language's core traits and benefits. Finally, we will explore how Go can benefit applications in becoming more scalable and cost-effective.

3.1 The Go Programming Language

The Go programming language, initially authored by Robert Pike in 2007, is a compiled programming language. [god24]

Since then, it has been under active development by Google and Open-Source maintainers. [god24]

The motivation for creating Go as a programming language was to resolve issues and criticism regarding programming languages at the time while maintaining their core benefits. [god24] Those benefits included the static typing of the C programming language and the readability of Python while creating a design that includes networking and multiprocessing capabilities in a language-native manner. Go was released in 2012 with version 1.0.[god24]

Go's most vital benefits include being a language with simple syntax, type inference, fast compile times, and state-of-the-art concurrency design.[23c; god24] This concurrency design is language native and partly shown by using the `go` keyword to start a green thread for an asynchronous function call; an example can be found in Listing 2. An example of the Go syntax and a simple *"Hello"* application can be found in Listing 1.

```
1  package main // entrypoint package
2
3  import "fmt"
4
5  func main() { // application entry point
6      fmt.Println("Hello")
7  }
```

Listing 1: "Hello" in Go

```
1  package main
2
3  func routine() {
4      // ...
5  }
6
7  func main() {
8      go routine()
9  }
```

Listing 2: Calling a function as a `goroutine`

Furthermore, Go offers in-build tools, like a race-checker, benchmarkers, a minimal package manager, profilers, static code analysis, and a rich standard library, which enhances the developer's experience in writing Go applications.[god24; 23c] Another interesting trait is that Go has a language-specific memory model.[god24; Bit22]

Go's compiled nature results in benefits, like fast startup times and seamless deployment on the target OS architecture, since it omits the need for an interpreter. [god24]

This results in the advantage that, while Go is a garbage-collected language, it generally outperforms interpreted languages.

3.2 Use-Cases

Go's robust ecosystem, including support for industry-relevant technologies like Hypertext Transfer Protocol (HTTP), Extended Markup Language (XML), JavaScript Object Notation (JSON), and Structured Query Language (SQL) databases, makes it, in combination with the benefits mentioned earlier, a popular choice in big tech.[god]

Some concretizations include but are not limited to Go making up over 75% of project code at the Cloud Native Computing Foundation, its applications in Google Cloud - natively supporting it through their product line, powering the serverless Credit-Offer-API at Capital One, and providing the vehicle for Dropbox's business logic for performance-critical backend sections.[god]

Go shines when used with containerization software, like Docker and, by extension, the container-orchestration software Kubernetes. This will be elaborated in section 4.

3.3 Scalability and Cost-Effectiveness

As mentioned above, Go's benefits make it a fitting choice for cloud-based applications.

This is particularly advantageous when leveraging features such as `goroutines` and `channels`, which enables the creation of easily maintainable microservices.[23c]

Furthermore, due to its memory model and garbage collection, Go's effective resource allocation allows for the more efficient assignment of available hardware quotas.[God24] This results in enhanced scalability and cost efficiency within cloud environments.[god; God24]

Lastly, the easy syntax and rich ecosystem, due to the package manager's decentralized nature — it only requires a reference to the library's git repository — allows quick and independent development of robust applications.[God24]

4 Go in Container Virtualization

When it comes to virtualization and Go, some names immediately come to mind: Kubernetes and Docker.

4.1 Docker

Docker[3], being a containerization and, by extension, virtualization software, is of crucial importance for the modern tech industry. It comes with several benefits, such as easy packaging, portability, and scalability of applications. Docker does not require a Hypervisor in order to run containers. It utilizes the underlying operating system directly.[Par+16] Dockers' engine is written in Go, which plays a key role in its efficient performance and resource management, due to the benefits of the language mentioned above.[23d; 22]

[3] `https://docker.com/`

4.2 Kubernetes

Kubernetes, on the other hand, is an open-source container orchestration platform written to a severe extent in Go, that automates many of the manual processes involved in deploying, managing, and scaling containerized applications.[Kub] Important to note is that Kubernetes is compatible with Docker containers, but not limited to them. It is also capable of running Singularity-based containers, which are used in the HPC sector as well, [PA22]. Go allows the concurrent and management and orchestration process of those containers or pods, making it a suitable programming language to scale a Kubernetes infrastructure.

Since both, Docker and Kubernetes, are written in Go, one might assume, that Go is a highly suitable language for container virtualization and orchestration tasks, possibly a great choice for HPC applications.

5 Usage of Go in HPC

This section briefly discusses the process of choosing a programming language for HPC and discusses the viability of Go as a fitting language in that aspect.

5.1 Choosing a programming language for HPC

In HPC applications, using the right programming language can significantly impact the yield of results concerning the resources used.

Languages like FORTRAN turned out to be quite adequate for its usage in HPC, [Loh10]. In fact, [Loh10] et al. state that programming challenges stem mostly from factors other than the programming language, which is FORTRAN itself. The author, however, does not elaborate on the performance implications of interpreted languages used in HPC. Moreover, [Loh10] encourages that readability, variability, and maintainability are crucial factors when developing HPC applications.

The popularity of statically typed and compiled languages like C and C++ in HPC is indicated by the popularity of MPI SDKs and their respective wrappers written in them. Those include, but are not limited to:

- MPICH, [mpi]
- OpenMPI, [ope]
- mpi4py, a Python-Wrapper for the aforementioned, [DF21]
- rsmpi, a Rust-Wrapper for the aforementioned, [24]

Since a strong performance boost compared to interpreted or runtime-dependent languages like Java is contested [Mer+15] et al., C, C++, and FORTRAN enable the developer of fine-grained control on how the program works down to the deepest level.

Returning to [Loh10] et al., [Vir10] argues that given a fixed timeframe for developers, a language like Java can outperform a C application. [Vir10] also argues, similarly to [Mer+15], that a runtime-dependent language, in that case Java, is not significantly less performant than C.

Choosing a language based on the use case and the application requirements may be wiser rather than just finding the perfect language for every use case. Using a lower-level language like C or C++ for HPC is possibly more efficient when there is no strict timeframe, high requirements on throughput, and capabilities for optimization.

Otherwise, one may consider an interpreted or runtime-dependent language.

5.2 Go as a programming language for HPC

At first glance, Go presents itself as a compelling option for developing HPC applications, particularly when the demands do not necessitate intricate control over low-level instructions.

This assertion gains confidence when considering the general usage of MPI libraries in many computational tasks, where Go has its own libraries in the form of Go-MPI [Weg23]and gompi [Bro23].

However, an apparent drawback emerges in the absence of native GPU support within Go.

This limitation can prove inhibitive for applications requiring the usage of GPUs for accelerated computation.

Despite this, innovative strides have been made to bridge this gap by leveraging Compute Unified Device Architecture (CUDA) and OpenGL[4] through C interfaces in conjunction with MPI implementations such as MPICH and Open MPI [BW14].

Moreover, Go's modern and straightforward approach to application development makes it an attractive candidate for swiftly prototyping and deploying HPC solutions, particularly those not bound by stringent performance requirements.

By bypassing the overhead associated with interpreters or runtime environments, Go appears as a potentially faster alternative to languages like Java or Python for certain HPC use cases.

The modern and easy approach to writing applications on Go makes it a fitting candidate for rapidly developed, non-performance-critical HPC applications.

Omitting the need for an interpreter or a runtime, Go can be an even faster solution than Java or Python implementations.

6 Go Libraries in HPC

This section delves into the difficulties of interfacing with MPI libraries through language wrappers and the challenges Go's garbage collection poses in HPC contexts.

It shows how Rust and Go interact with C and how performance issues and incomplete functionality mappings are often associated with such wrappers.

Furthermore, it discusses the complexities of binding C functions in Go, as well as the implications for cross-platform compatibility. Potential issues encountered in distributed computing environments are illustrated by the impact of Go's shared memory model and garbage collection on HPC applications. The section shows how important language choice and memory management strategies are when making HPC applications.

[4] `https://www.opengl.org/`

6.1 MPI Library Wrappers

Almost every language has the capability to use an Foreign-Function Interface (FFI) to interface with lower-level languages like C, or C++. This is especially important when it comes to HPC for enabling other languages to interface with MPI libraries, like `OpenMPI` or `mpich`.

A library wrapper essentially packages the base library, which in that case would be `OpenMPI` or `mpich`, in such a way that the functionality is accessible inside the target language. This is done by using the FFI aspect of the target language to link the functions of the underlying library to their respective counterpart. This allows the invocation of the functions and capabilities of the underlying library from the target language, running the compiled libraries code with the defined parameters, essentially forwarding the parameters to the library for processing.

One example, calling Rust from C, is [Gou], interfacing with a C program over Rusts' function definitions. Listing 3 shows, how a simple addition function is defined in Rust. The C counterpart, displayed in Listing 4, calling the Rust-defined function, matching the data-types from a Rust-compiled shared library; effectively outsourcing the function call processing to Rust while running the C program.

```rust
#[no_mangle]
pub extern "C" fn addition(a: u32, b: u32) -> u32 {
    a + b
}
```

Listing 3: Defining the `addition` function and business logic in Rust, [Gou]

```c
#include <stdio.h>
#include <stdint.h>
#include <inttypes.h>
extern uint32_t addition(uint32_t, uint32_t);
int main(void) {
    uint32_t sum = addition(1, 2);
    printf("%" PRIu32 "\n", sum);
}
```

Listing 4: Interfacing with the Rust-implemented function in C, adding two unsigned 32-bit integers, [Gou]

Applying the same idea, but going from the C/C++ implementations of `OpenMPI` or `mpich`, libraries like `mpi4py` or `rsmpi`, as mentioned above, allow the respective languages to leverage the embedded libraries' capabilities.

One crucial implication, which results from this programming pattern is, that the invocation of the embedded functions is always proxied by the overlaying language, which is possibly slower, effecting the performance of the HPC application compared to pure C/C++ implementations using those libraries directly. Furthermore, library wrappers

tend to be incomplete in their functionality-mapping, as `rsmpi` and `gosl` indicate, [24] [Cpm].

Binding C functions in Go is a fairly tricky process. It involves the usage of the `cgo` command, which comes with the Go development suite.[God24]

First, it involves employing the C pseudo-package and defining the corresponding data types for their counterparts in the embedded library. Then, the developer needs to respect the libraries and C-specific traits when mapping the Application Programming Interface (API) to every method, respecting the behavior and data-types of the underlying functions.[God24]

Listing 5 shows, how gompi achieves this by interfacing with `OpenMPI`. Compatibility flags are displayed at the top of the snippet, while the build-process, flags, and the start of the pseudo-package follow.

```
1  //go:build !windows
2  // +build !windows
3
4  //go:generate stringer -type=DataType
5  //go:generate stringer -type=Op
6  // ...
7  /*
8  #include "mpi.h"
9
10 MPI_Comm     World     = MPI_COMM_WORLD;
11 MPI_Status*  StIgnore  = MPI_STATUS_IGNORE;
12
13 #define DOUBLE_COMPLEX double complex
14 */
15 import "C"
```

Listing 5: Go Interfacing with OpenMPI, [Bro23]

The potential drawback of tying C code to Go may pertain to its compatibility across multiple platforms, as it becomes bound to the platform compatibility of the underlying C codes.

In summary, the complexity of interfacing with C libraries in Go is heavily dependent on the embedded counterpart. To create a properly defined mapping, it is necessary to know the difficulties and behavior of the functionality and aspects of the library beforehand. In the context of MPI, it is imperative that the developer is cognizant of the characteristics of the underlying library and its interaction with the HPC cluster, and may be able to modify the Go source code to align with this behavior. Furthermore, when incorporating C code into Go, it may compromise its compatibility with various platforms.

6.2 Garbage Collection and Shared Memory in Go for HPC

Characteristic of Go's concurrency model is its shared memory model and built-in garbage collection. Since a green-thread-similar model is utilized inside the goroutines, the require-

ment for a memory-model capable of allowing such concurrency became a necessity.[God]

A part of the official Go documentation discusses a crucial aspect of Go programming: handling memory safely and effectively.[God] It discusses Go's features to ensure correct memory handling, decreasing the probability of memory-related errors like segmentation faults and buffer overflows. Moreover, Go distinguishes between "kinds" of memory access; for example, read-like and write-like operations being handled with different priorities, mitigating the chance of data races. This allows a seamless and safe development of concurrent applications. However, an essential context in the aspect of HPC is that the shared memory model refers to being shared between threads, not machines.

Since multiple goroutines do have access to the same memory space, garbage-collection will also be handled on the same space, [God; Bit22].

This introduces the problem of the garbage collector not being able to effectively assess an object's lifetime when used in HPC with MPI libraries. This problem was observed last year during the development of a traffic simulator, initially written in Go for HPC.[5]

In the referenced report, some objects representing a vehicle in that simulation were garbage collected because they were sent to other machines inside the cluster. The origin machine had no way of knowing if the corresponding vehicle still existed. That issue displayed an example of why choosing the right language for the requirements becomes so crucial.

To summarize, Go allows the development of memory-safe and efficient applications on a single machine but reaches its limitations when being deployed onto HPC clusters with high throughput since the automatic garbage-collection process needs to be considered and managed by the developer.

7 Serverless Functions in HPC with Go

In this section, we explore what serverless functions are and how they are usually deployed. Subsequently, we examine prominent Functions-as-a-Service (FaaS) frameworks. Finally, we examine solutions for HPC involving serverless functions for processing workloads.

7.1 Serverless functions

A new trend in cloud computing with rising popularity is the utilization of serverless functions.[Sch+21] Serverless functions are a model in which the cloud provider dynamically manages the allocation of available hardware resources.[Sch+21] These functions are initiated by events or requests and execute code provided by the user, usually by an endpoint invocation.[Sch+21]

Several key traits define the characteristics of serverless applications.

First, we have event-driven execution of the stateless function. This ensures that only necessary actions are undertaken, thereby aiding in the efficient utilization of resources.[Sch+21]

Subsequently, the automated scaling of the available resources to a specific deployment facilitates dynamic scaling without any manual intervention, as demonstrated in [Sch+21].

[5] `https://hps.vi4io.org/_media/teaching/summer_term_2023/pchpc-student/valerius_mattfeld_bianca_vetter_report.pdf`

Serverless functions are usually put in container images, which make them portable and separate from other programs, [Oak+18]. The functions can be used in various programming languages, [Sch+21].

7.2 Functions-as-a-Service Frameworks

Since various Open Source FaaS frameworks utilize the advantages of Kubernetes, those frameworks stand out the most. Some notable examples are:

- `knative.dev` (supporting languages like Go, Elixir, Java, etc.), [23a]
- `nuclio.io` — With a data science focus and completely written in Go, [23b]
- `openfaas.com` — Also using Go, [ope23]
- `fission.io` — Built with Go, [fis23]

Since Kubernetes is written in Go, those frameworks might choose Go as their primary language because they stay within one ecosystem and language.

7.3 Serverless-Functions in HPC

With Kubernetes being the main platform for investigating serverless functions for modern high-performance computing, two crucial issues emerge: Response Times and Scheduling.[LG22; DKK22; PA22]

[LG22] and [PA22] explore different approaches to resolve those issues.

[LG22] points out, that firstly, Kubernetes sees HPC workloads as batch jobs. The author explains that the performance nuances of containerized HPC workloads are not yet fully explored. Moreover, the author goes into detail, that Kubernetes-native batch-jobs are not designed for supporting a fully featured HPC application efficiently, and that Kubernetes schedules the jobs not in workloads, but in pods. To resolve this, the author mentions a tool, Kubeflow MPI[6], which helps to launch HPC workloads. In combination with fine-grained Kubernetes policies, [LG22] can produce a reduction in response times of HPC workloads by 35% in Kubernetes, though basing those results on small-scales MPI jobs for single nodes. Lastly, [LG22], emphasizes, that a precise control over mixed workloads, e.g., I/O applications proves to be a challenge to be resolved.

An alternative approach is presented by [PA22], who presents an alternative architecture, "Shoc", which is designed specifically for HPC workloads, while maintaining serverless access for the function invocators.

[PA22] builds the architecture of Shoc on Singularity[KSB17] and Docker in combination with the scheduling and resource-management capabilities of Kubernetes. The author states, that this architecture allows CPU as well as data-intensive workloads, without the need of a complex HPC infrastructure. Moreover, Shoc seems to bypass the scheduling limitations, like [LG22] discussed. Furthermore, [PA22] mentions, that Kubernetes' ability to user non-Docker containers, like Singularity-based containers, as aforementioned, could be beneficial to the HPC world, since they are more common. Leveraging the

[6] `https://www.kubeflow.org/docs/components/training/mpi/`

`kube-autoscaler`, the Shoc architecture can instantiate and join nodes on demand, allowing massive scaling.[PA22]

In summary, serverless functions with a Kubernetes base represent a modern alternative to traditional HPC infrastructures. Though, critical issues need to be resolved for those approaches to become a mature and reliable option; like the response-time problem mentioned above, and more testing for mixed workloads.

8 Discussion

We have explored Go from its origins, syntax design, and key benefits through to its use-cases in real-life applications and HPC. The result is, that Go appears to be a scalable and cost-effective language which is also primarily used in Container virtualization and orchestration.

Subsequently, we have explored possible candidates for HPC programming languages and its requirements to program an effective a performant application. During this, we have assessed that it is a requirement-based decision, depending on the skill and experience of the development team as well as time resources to optimize for lower-level languages like C, and, if not, another language. We categorized Go in between lower-level languages and interpreted languages like Java.

After that, we examined MPI library wrappers for the most prominent libraries - OpenMPI and mpich - and how wrappers generally work. We took those findings into account when briefly analyzing a Go MPI wrapper library for its structure in regard of interfacing with one of those two libraries. While Go having a suboptimal process of binding to C libraries in general, we further accessed that guaranteeing bug-safety, with importance of memory access problems, the developer of the bindings library must have an intricate comprehension of the workings of HPC clusters and MPI libraries in general. Furthermore, the bindings must be implemented in such a way, that it circumvents the default garbage collection mechanisms of Go to avoid accidental object removal from a worker machine.

Lastly, we have explored the concept of using Serverless Functions in HPC clusters and Kubernetes with HPC loads in the context of Go to find a reasoning for the popularity of that language in this regard. We found that two problems are prominent, when using Kubernetes with HPC workloads, which are response time and scheduling. Two papers explored this project, one of which indicated that a cluster with the right set of policies can successfully reduce response times for small single-node workloads by up to 35%, and the other, that a composition of tools named "Shoc" can provide an architecture capable of scaling, processing CPU intensive and data-intensive workloads, while bypassing scheduling limitations of the previous paper. All authors mentioned in section 7.3 criticize a severe lack of research in that area.

While Go turns out to be an excellent choice for developing concurrency-based applications, its simple syntax and other benefits come with some caveats when it stays in context to HPC. While its garbage collection, shared memory model and compiled nature make it a delightful language to write applications in, those features are disruptive when developing applications for HPC, especially when it comes to interfacing with C libraries like OpenMPI. Because it's basically forcing the developer to manage memory to a certain

degree, its usefulness in that regard is questionable compared to C/C++ or Rust.

`mpi4py`, for example, is a mature MPI library for Python, providing similar development speeds to Go, while omitting compilation shenanigans as well as the manual process of managing object garbage collection from shared memory. Furthermore, `rsmpi`, though it still being incomplete, is easier to compile and - when observing the Rust part - safe to write, while providing the same benefit of a compiled executable.

Those insights indicate that Go appears to be unbeatable when it comes to infrastructure code, but provides little upsides when it comes to actual HPC applications. Outdated, unmaintained, undocumented, and incomplete wrapper libraries make it challenging to develop applications with Go, which mitigates the entire teleology of the language - it being simple, accessible and fast.

Since Go's concurrency capabilities could still be leveraged for HPC, a properly developed wrapper could unlock the potential of that language. Writing a "wrapper-wrapper library", for example, a Go library utilizing `rsmpi`, could reduce the issues of memory safety and compilation difficulties when developing HPC applications.

9 Conclusion

While Go is an accessible and comparably performant language, currently, as of the time of writing, it is not a suitable language to write HPC applications in. This is due to the difficult implementation of a fitting MPI library into Go's language design, namely its garbage-collector in combination with its single-machine memory model. Go can introduce bugs, when the library developer and HPC application developer do not fully understand the underlying architecture of the libraries and how HPC works in general.

The lack of research in that regard, as well as the lack of maintenance for corresponding wrapper libraries, can be traced back to those issues.

However, it fits perfectly with infrastructure code and is heavily in use with modern containerization and orchestration software. This is where Go in the aspect of HPC shines - as an infrastructure management language.

Go could possibly benefit and leverage its potential with a proper wrapper library, perhaps implementing a safe underlying wrapper written in Rust with `rsmpi`.

References

[22] *Which Part of Docker Written in the GO Language? - General Discussions*. Docker Community Forums. Oct. 8, 2022. URL: https://forums.docker.com/t/which-part-of-docker-written-in-the-go-language/129912 (visited on 04/22/2024).

[23a] *Knative Documentation*. Knative, May 31, 2023. URL: https://github.com/knative/docs (visited on 05/31/2023).

[23b] *Nuclio - "Serverless" for Real-Time Events and Data Processing*. nuclio, May 30, 2023. URL: https://github.com/nuclio/nuclio (visited on 05/31/2023).

[23c] *The Go Programming Language*. Go, July 5, 2023. URL: https://github.com/golang/go (visited on 07/05/2023).

[23d] *The Moby Project*. Moby, July 5, 2023. URL: https://github.com/moby/moby (visited on 07/05/2023).

[24] *Rsmpi/Rsmpi*. rsmpi, Apr. 12, 2024. URL: https://github.com/rsmpi/rsmpi (visited on 04/20/2024).

[Bit22] Frederico Bittencourt. *Concurrency in Go: Shared Memory*. Oct. 15, 2022. URL: https://blog.fredrb.com/2022/10/15/go-concurrency-shared-memory/ (visited on 04/21/2024).

[Bro23] Seth Bromberger. *Sbromberger/Gompi*. Sept. 5, 2023. URL: https://github.com/sbromberger/gompi (visited on 10/14/2023).

[BW14] Alexander Beifuss and Johann Weging. *A Golang Wrapper for MPI*. Universtiy of Hamburg, Apr. 4, 2014.

[Cas22] David Cassel. *What Made Golang So Popular? The Language's Creators Look Back*. The New Stack. May 29, 2022. URL: https://thenewstack.io/what-made-golang-so-popular-the-languages-creators-look-back/ (visited on 04/22/2024).

[Cpm] Cpmech/Gosl. *Mpi Package - Github.Com/Cpmech/Gosl/Mpi - Go Packages*. URL: https://pkg.go.dev/github.com/cpmech/gosl/mpi#section-readme (visited on 07/06/2023).

[DF21] Lisandro Dalcin and Yao-Lung L. Fang. *Mpi4py: Status Update After 12 Years of Development*. 2021. DOI: 10.1109/mcse.2021.3083216. URL: https://github.com/mpi4py/mpi4py (visited on 04/20/2024).

[DKK22] Jonathan Decker, Piotr Kasprzak, and Julian Martin Kunkel. "Performance Evaluation of Open-Source Serverless Platforms for Kubernetes". In: *Algorithms* 15.7 (2022).
"High-performance computing (HPC) clusters can profit from improved serverless resource sharing capabilities compared to reservation-based systems such as Slurm." (Decker et al., 2022, p. 1)
"However, before running self-hosted serverless platforms in HPC becomes a viable option, serverless platforms must be able to deliver a decent level of performance." (Decker et al., 2022, p. 1)

"Other researchers have already pointed out that there is a distinct lack of studies in the area of comparative benchmarks on serverless platforms, especially for open-source self-hosted platforms." (Decker et al., 2022, p. 1). ISSN: 1999-4893. DOI: 10.3390/a15070234. URL: https://www.mdpi.com/1999-4893/15/7/234.

[fis23] fission. *Fission/Fission: Fast and Simple Serverless Functions for Kubernetes*. 2023. URL: https://github.com/fission/fission (visited on 05/31/2023).

[God] Homepage Go.dev. *The Go Memory Model - The Go Programming Language*. URL: https://go.dev/ref/mem (visited on 04/21/2024).

[god] go.dev. *Go for Cloud & Network Services - The Go Programming Language*. URL: https://go.dev/solutions/cloud (visited on 04/19/2024).

[God24] Go.dev. *Cgo Command - Cmd/Cgo - Go Packages*.
"Cgo enables the creation of Go packages that call C code." (Go.dev, 2024, p. 1)
"To use cgo write normal Go code that imports a pseudo-package "C". The Go code can then refer to types such as C.size_t, variables such as C.stdout, or functions such as C.putchar." (Go.dev, 2024, p. 1)
"These may then be referred to from Go code as though they were defined in the package "C"." (Go.dev, 2024, p. 2)
"For security reasons, only a limited set of flags are allowed, notably -D, -U, -I, and -l. To allow additional flags, set CGO_CFLAGS_ALLOW to a regular expression matching the new flags." (Go.dev, 2024, p. 2). 2024. URL: https://pkg.go.dev/cmd/cgo (visited on 04/21/2024).

[god24] go.dev. *Frequently Asked Questions (FAQ) - The Go Programming Language*. FAQ. 2024. URL: https://go.dev/doc/faq (visited on 04/22/2024).

[Gou] Jake Goulding. *Basics - The Rust FFI Omnibus*. URL: https://jakegoulding.com/rust-ffi-omnibus/basics/ (visited on 04/21/2024).

[KSB17] Gregory M. Kurtzer, Vanessa Sochat, and Michael W. Bauer. "Singularity: Scientific Containers for Mobility of Compute". In: *PLOS ONE* 12.5 (May 11, 2017). Ed. by Attila Gursoy, e0177459. ISSN: 1932-6203. DOI: 10.1371/journal.pone.0177459. URL: https://dx.plos.org/10.1371/journal.pone.0177459 (visited on 04/21/2024).

[Kub] Kubernetes. *Kubernetes Documentation*. Kubernetes. URL: https://kubernetes.io/docs/home/ (visited on 07/05/2023).

[LG22] Peini Liu and Jordi Guitart. *Fine-Grained Scheduling for Containerized HPC Workloads in Kubernetes Clusters*.
"However, scheduling policies that consider the performance nuances of containerized High Performance Computing (HPC) workloads have not been well-explored yet" (Liu and Guitart, 2022, p. 1)
"Our results show that our fine-grained scheduling policies outperform baseline and baseline with CPU/memory affinity enabled policies, reducing the overall response time by 35% and 19%, respectively, and also improving the makespan by 34% and 11%, respectively. They also provide better usability and flexibility to specify HPC workloads than other comparable HPC Cloud frameworks, while providing better scheduling efficiency thanks to their multi-layered approach." (Liu and Guitart, 2022, p. 1)

"HPC workloads are considered as batch jobs in Kubernetes." (Liu and Guitart, 2022, p. 2)
"An HPC workload is specified as a launcher and one or multiple workers. Each launcher or worker is a container that can be executed as a Pod and run in parallel in a Kubernetes cluster. However, the original Kubernetes batch jobs are not designed for supporting the HPC applications efficiently." (Liu and Guitart, 2022, p. 2)
"Also, the Kubernetes default scheduler does not schedule jobs but individual Pods." (Liu and Guitart, 2022, p. 2)
"Kubeflow MPI operator [10] provides a better specification for MPI jobs which defines an MPI 'Launcher' and an MPI 'Worker'. In most cases, all the MPI worker processes will be launched in this 'Worker' container." (Liu and Guitart, 2022, p. 2)
"Our results show that the proposed fine-grained policies can reduce the response time of HPC workloads up to 35%, as well as improve the makespan up to 34%. Although our benchmarks are small-scaled MPI jobs that fit in a single node," (Liu and Guitart, 2022, p. 10)
"e.g. for network applications, one would probably use coarse-grained granularity within each node to exploit fast shared-memory communication, whereas CPU-bound applications could use fine-grained granularity to exploit affinity." (Liu and Guitart, 2022, p. 10)
"In the future, we will enhance our fine-grained policies for the scheduling of mixed HPC-AI workloads on Kubernetes, and to consider other application profiles such as I/O applications. Moreover, we will evaluate them in larger-scale scenarios." (Liu and Guitart, 2022, p. 10). Nov. 21, 2022. DOI: 10.48550/arXiv.2211.11487. arXiv: 2211.11487 [cs]. URL: http://arxiv.org/abs/2211.11487 (visited on 07/05/2023). preprint.

[Loh10] Eugene Loh. "The Ideal HPC Programming Language: Maybe It's Fortran. Or Maybe It Just Doesn't Matter." In: *Queue* 8.6 (June 1, 2010).
"Fortran, which is arguably still the primary language in HPC—proved remarkably adequate. Programming challenges stem mostly from other factors." (Loh, 2010, p. 1)
"We rewrote a number of HPC benchmarks and applications using modern Fortran in a way that took into account the human costs of software development: programmability and associated characteristics such as readability, verifiability, and maintainability." (Loh, 2010, p. 2)
"Most of all, the HPC community could well benefit from a community-wide effort to emphasize programmability and human productivity." (Loh, 2010, p. 9), pp. 30–38. ISSN: 1542-7730. DOI: 10.1145/1810226.1820518. URL: https://dl.acm.org/doi/10.1145/1810226.1820518 (visited on 04/20/2024).

[Mer+15] Juan-J. Merelo et al. *There Is No Fast Lunch: An Examination of the Running Speed of Evolutionary Algorithms in Several Languages*. Nov. 3, 2015. DOI: 10.48550/arXiv.1511.01088. arXiv: 1511.01088 [cs]. URL: http://arxiv.org/abs/1511.01088 (visited on 04/20/2024). preprint.

[mpi] mpich.org. *MPICH | High-Performance Portable MPI*. URL: https://www.mpich.org/ (visited on 04/20/2024).

[Oak+18] Edward Oakes et al. "SOCK: Rapid Task Provisioning with Serverless-Optimized Containers". In: USENIX Annual Technical Conference. [TLDR] This work analyzes Linux container primitives, identifying scalability bottlenecks related to storage and network isolation, and implements SOCK, a container system optimized for serverless workloads. July 11, 2018. URL: https://www.semanticscholar.org/paper/SOCK%3A-Rapid-Task-Provisioning-with-Containers-Oakes-Yang/8352e3ba20cb0fd48e2514bc1948e68108943e39 (visited on 04/21/2024).

[ope] openmpi.org. *Open MPI: Open Source High Performance Computing*. URL: https://www.open-mpi.org/ (visited on 04/20/2024).

[ope23] openfaas. *Openfaas/Faas: OpenFaaS - Serverless Functions Made Simple*. 2023. URL: https://github.com/openfaas/faas (visited on 05/31/2023).

[PA22] Davit Petrosyan and Hrachya Astsatryan. "Serverless High-Performance Computing over Cloud". In: *Cybernetics and Information Technologies* 22.3 (Sept. 1, 2022).
"The article aims to present an architecture that enables HPC workloads to be serverless over the cloud (Shoc), one of the most critical cloud capabilities for HPC workloads." (Petrosyan and Astsatryan, 2022, p. 82)
"On one hand, Shoc utilizes the abstraction power of container technologies like Singularity and Docker, combined with the scheduling and resource management capabilities of Kubernetes." (Petrosyan and Astsatryan, 2022, p. 82)
"On the other hand, Shoc allows running any CPU-intensive and data-intensive workloads in the cloud without needing to manage HPC infrastructure, complex software, and hardware environment deployments" (Petrosyan and Astsatryan, 2022, p. 82)
"Shoc architecture has overcome the limitations of scheduling serverless HPC workloads on the clouds." (Petrosyan and Astsatryan, 2022, p. 85)
"Kubernetes has rich resource management and workload scheduling functionality. Policy-based scheduling and declarative resource requirements allow building a serverless HPC solution to run HPC workloads over Kubernetes. On the other hand, Kubernetes supports several container runtime environments other than Docker. Therefore, it could be easily implemented to orchestrate Singularity-based containers, more common in the HPC world." (Petrosyan and Astsatryan, 2022, p. 86)
"4.2. Containerization The Shoc architecture can use any technology compatible with Kubernetes. However, for practical reasons, Shoc uses Docker and Singularity as the primary container runtimes for the Shoc architecture. One of the most critical advantages of Shoc architecture is that the end-user is unaware of any containers, pods, and other infrastructure-level complexities. To achieve this, Shoc provides back-end services that containerize target programs. In this case, the end-user submits an executable to the Shoc system, and a particular back-end service containerizes the given executable along with its dependencies. This level of abstraction allows containerization of any workload with the Container runtime of choice. Thus, whether it is an Open-MPI executable with dependencies or a Java-based Spark application, it gets containerized the same way. The container image of the workload is then pushed into a uniqu" (Petrosyan and Astsatryan, 2022, p. 87)

"88 registry for further reference. This containerization method is used as a foundation of well-known serverless systems (function-as-a-service, etc.)." (Petrosyan and Astsatryan, 2022, p. 88)
"The kube-autoscaler feature allows the Kubernetes cluster to instantiate and join a node on demand, enabling Shoc to support massive scale infrastructures. This way, if Shoc is given several Kubernetes cluster references managed by various Cloud providers (public or private), it will be ready to scale up or down based on the actual resource usage. This makes Shoc a serverless system, as underlying resources are allocated and de-allocated without the involvement of the Shoc system or a human operation." (Petrosyan and Astsatryan, 2022, p. 88)
"The article presents the architecture of the Shoc system. Shoc architecture aims to advance seamless cloud infrastructure usage for running HPC workloads by benefiting from modern cloud technologies. It adds serverless experience to the enduser and takes out the complexity of deploying HPC infrastructures. The proposed methodology expands existing high-performance computing technologies with containerization enabling seamless scaling, deployment, and clustering capabilities." (Petrosyan and Astsatryan, 2022, p. 91), pp. 82–92. DOI: 10.2478/cait-2022-0029. URL: https://sciendo.com/article/10.2478/cait-2022-0029 (visited on 04/21/2024).

[Par+16] Fawaz Paraiso et al. "Model-Driven Management of Docker Containers". In: *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)* (2016), pp. 718–725. URL: https://api.semanticscholar.org/CorpusID:17464333.

[Sch+21] Johann Schleier-Smith et al. "What Serverless Computing Is and Should Become: The next Phase of Cloud Computing". In: *Communications of the ACM* 64.5 (Apr. 26, 2021), pp. 76–84. ISSN: 0001-0782. DOI: 10.1145/3406011. URL: https://dl.acm.org/doi/10.1145/3406011 (visited on 04/21/2024).

[Vir10] Patrick Viry. *Java for HPC (High-Performance Computing)*.
""HPC developers and users usually want to use Java in their projects" [...] Indeed, Java has many advantages over traditional HPC languages:

- faster development
- higher code reliability
- portability
- adaptative run-time optimization" (Viry, 2010, p. 2)

"Since the language is cleaner, it is easier for developers to concentrate on performance optimization (rather than, say, chasing memory-allocation bugs):" (Viry, 2010, p. 2)
""In Mare Nostrum the Java version runs about four times faster than the C version [...]. Obviously, an optimisation of the C code should make it much more efficient, to at least the level of the Java code. However, this shows how the same developer did a quicker and better job in Java (a language that, unlike C, he was unfamiliar with)" [1]." (Viry, 2010, p. 2). The dress of thought. Sept. 3, 2010. URL: https://ateji.blogspot.com/2010/09/java-for-high-performance-computing.html (visited on 04/20/2024).

[Weg23] Johann Weging. *Go-Mpi*. Feb. 5, 2023. URL: https://github.com/yoo/go-mpi (visited on 07/06/2023).

A Code samples