Language Interoperability: Improving Multi-Language Systems with Bindings

CS4991 Capstone Report, 2025

Ahbey Mesfin
Computer Science
The University of Virginia
School of Engineering and Applied Science
Charlottesville, Virginia USA
ttw8ea@virginia.edu

ABSTRACT
Modern software development often involves multiple programming languages, each chosen for its suitability for a specific task. However, this multilingual approach introduces complex interfaces between languages, leading to decreased performance. Language interoperability—the ability for different programming languages to work seamlessly together within a single program - presents a potential solution to these bottlenecks. One method to achieve interoperability is through language binding, which allows code written in one language to directly interact with code written in another. My work illustrates the benefits of interoperability through my work experience at the MITRE Corporation, and how similar approaches can be taken to improve other existing applications. One potential avenue to expand this work is to compare and contrast bindings with other interoperability methods (such as Foreign Function Interfaces), and determine when one is more appropriate than the other.

1. INTRODUCTION

Today, large-scale software systems are rarely confined to a single programming language - instead, developers leverage multiple languages, selecting each for its particular strengths, for a given task within the larger system. This tendency harbors some complications, however with regard to communication across languages. Without the proper interoperability mechanisms in place, developers must rely on inefficient communication mediums (such as file-based data transfer or data serialization), or sometimes even duplicate implementations in different languages.

Language interoperability offers a solution to this problem by enabling near seamless communication between languages within a single unified program. One commonly used method to enable interoperability is language binding, which allows code written in one language to directly interact with data structures and call functions from another. My experience at the MITRE Corporation provides one example of how interoperability can benefit large-scale systems written in multiple programming languages. Expanding on this work, I explore how bindings compare with other interoperability methods to determine the most effective approach depending on the application needs.

2. RELATED WORKS

Cross-language interoperability is, and has been, an existing challenge in software development. Early attempts to solve this issue, such as the Common Object Request Broker Architecture (COBRA) and Component Object Model (COM), introduced significant complexity and performance overhead (Chisnall, 2013). Such work by the

authors investigates why older solutions aren't expandable or scalable to today's demands. Another important note in this investigation is the difference in object models across programming languages. For example, object-oriented languages differ in how they implement inheritance and memory management, making seamless integration very complex. Furthermore, challenges arise when handling exceptions and garbage collection across languages.

Another work implements a VM, TruffleVM, to be an example of how to elegantly enable interoperability (Grimmer et al, 2018). TruffleVM is a language-agnostic mechanism for cross-language interoperability and is not locked to a fixed set of languages. The authors go into detail about the development, challenges, and technical details of TruffleVM. Such work provides a basis for the potential of language interoperability and gives more examples to compare Bindings to (as a Virtual Machine is distinct from Bindings or FFIs), providing a more well-rounded understanding of when bindings are appropriate.

3. PROPOSAL DESIGN

This experiment will examine the performance of different interoperability methods in the setting of heavy computational load, specifically, matrix multiplication (of different dimensions) will be used.

3.1 Overview of Design

This project evaluates the efficacy of different language interoperability techniques by implementing "toy" (simple) programs in two different languages and measuring the performance differences between them. The primary focus is on comparing language bindings with another interoperability approach, such as file-based data transfer and data serialization. The results will provide some insight regarding when one method is preferable over another, depending on the context of the application's use-case.

3.2 Experimental Setup

The experiment entails the development of two minimal programs that execute identical computational tasks but differ in how they exchange data between languages. One program will use language bindings (C++ with Python bindings via the pybind11 library), and the other will use an alternative technique. The performance of both approaches will be measured and analyzed numerically and graphically.

3.2.1 Programming Language Selection

To analyze interoperability effectively, two languages will be selected based on their relevance in real-world multi-language systems. The two that have been selected are Python and C++. C++ is commonly used in performance-critical applications, while Python is used for scripting.

3.2.2 Methods of Interoperability

The two methods of interoperability to be compared are Language bindings and some form of Data transfer (sockets, shared memory, or file-based communication).

3.3 Performance Evaluation

To measure the efficiency of each method, key performance metrics will be logged. First, Execution time for the program - the time taken from start to finish to complete the task - will be recorded. Second, the data transfer latency—the overhead created when transmitting data between the two languages—will be considered. Finally, the ease of implementation will be weighed. That is, my subjective experience of developing the program with the respective interoperability method will be assessed regarding how difficult it was to accomplish. A series of tests will be conducted, where the programs perform identical computational tasks (for

example, numerical simulations or matrix multiplications).

3.4 Expected Challenges

A number of challenges may arise during implementation, including compatibility issues between languages. Data representation, memory management, and error handling are all large items that differ among languages, which may be tricky to circumvent. Another challenge is ensuring that the tests are as fair as possible. Ensuring my own hardware does not impact the performance of the programs (since hardware may happen to run faster in one instance than another) is an important consideration.

4. RESULTS

The two approaches were compared on the basis of matrix multiplication speeds. Identical dimensions for the matrices were used in both approaches. The results show an unexpected result: while the bindings approach (will refer to as Pybind11 from here) significantly outperformed the file-based approach for small matrices, the advantage quickly diminishes as the matrix size grows. Notably, for large matrices (1000x1000 and beyond), the file-based method is faster, with execution times growing at a much slower rate than the Pybind11 approach. Please refer to Fig 1 below to observe this trend.

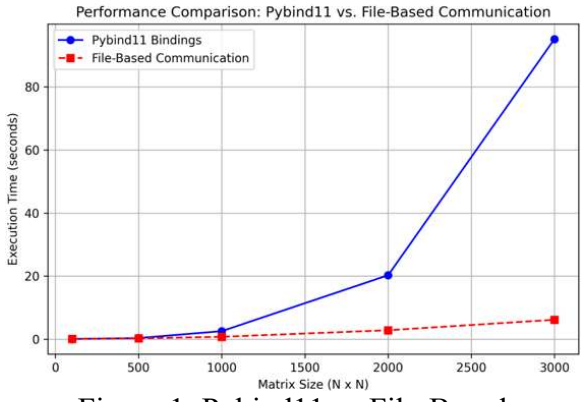

Figure 1: Pybind11 vs File-Based Communication Runtimes

To further quantify the differences in computational speed between the two approaches, a metric called *speedup* will be considered. Speedup is simply the ratio of the time it takes between the two approaches, specifically, the file-based approach divided by the Pybind11 approach (when speedup > 1, Pybind11 is faster, when speedup < 1, file-based is faster). This metric simply shows how many times faster Pybind11 is than the file-based approach. After executing roughly 50 epochs and taking the average, it is clear that the Pybind11 approach is appropriate for smaller matrices, but scales poorly. For example, the speedup for a 100x00 matrix computation was 24.5. This means that Pybind11 was 24.5x faster in computing the matrix product than the file-based approach. On the other hand, the speedup for very large matrices (3000x3000) was .06, meaning that Pybind11 was .06x faster (or 16.67x slower) than its counterpart. Please refer to Fig 2 for a full list of the execution times and speedup factors with respect to the different matrix dimensions.

Matrix Size	Pybind11 Time (sec)	Fille-Based Time (sec)	Speedup Ratio
100x100	0.002805	0.068737	24.51x
500x500	0.310283	0.227768	0.73x
1000x1000	2.494159	0.711464	0.29x
2000x2000	20.252831	2.782751	0.14x
3000x30000	95.236415	6.117981	0.06x

Figure 2: Pybind11 vs File-Based Communication Times and Speedups

As can be seen in Fig 1 and Fig 2, the Pybind11 approach has an almost exponential increase in runtime as the computational load increases. In short, the file-based approach remains relatively efficient across all tested sizes. While it does require writing and reading matrices from the disk, the execution time scales almost linearly, resulting in much better performance for large matrices. These experiments suggest that the overhead of transferring data via files is less important than the cost of executing Pybind11 functions at scale.

Pybind11, while being slower for larger matrices, was very simple to implement. The documentation, community, and support are very active - allowing developers such as myself to pick up the tool and get started relatively quickly. File-based data transfer was a bit tricky to implement, as there is no "tool" being used here - just standard libraries.

5. CONCLUSION

This project provides an important performance analysis on two different methods for integrating C++ matrix multiplications with Python: file-based data transfer and Pybind11 bindings. While Pybind11 was expected to be the superior approach due to in-memory execution, the results show that file-based execution outperforms Pybind11 for larger matrices. These findings challenge the assumptions held by me and programmers in general about bindings and highlight the need for further optimization when using Pybind11 for large-scale problems.

Furthermore, the results of this study offer some valuable insights for developers choosing between efficiency and usability in C++-Python interoperability. Pybind11, despite the surprising difference in execution time, remains as a useful tool for low-latency and small-scale operations, while file-based communication might be a better, scalable solution for larger datasets. These findings can be easily applied to machine learning and scientific computing, as both of these fields heavily involve large matrices and, in some cases, communication between codebases written in different languages.

6. FUTURE WORK

To further refine this work, I would like to compare these interoperability methods with other existing ones (such as FFIs and sockets), to get a better understanding on how other options fare. The most important expansion that can be done on this work is to compare more tasks. This report only concerns matrix multiplication, but this is an extremely narrow slice of what programs do in the real world. There is a possibility that Pybind11 could perform much better than the file-based approach if a task other than matrix multiplication was tested. These two changes, including other interoperability methods and testing different computational tasks, would greatly enhance the generalizability and applicability of my work.

REFERENCES

Chisnall, D. (2013). The challenge of cross-language interoperability. *Commun. ACM*, *56*(12), 50–56. https://doi.org/10.1145/2534706.2534719

Grimmer, M., Schatz, R., Seaton, C., Würthinger, T., Luján, M., & Mössenböck, H. (2018). Cross-language interoperability in a multi-language runtime. *ACM Trans. Program. Lang. Syst.*, *40*(2), 8:1-8:43. https://doi.org/10.1145/3201898