MPI4All: Universal Binding Generation for MPI Parallel Programming

César Piñeiro[1][0000−0001−6490−7128], Alvaro Vazquez[2][0000−0002−4719−8099], and Juan C. Pichel[2,3][0000−0001−9505−6493]

[1] CITIC, Universidade da Coruña, Spain
`cesar.pomar@udc.es`
[2] Electronics and Computer Science Department, Universidade de Santiago de Compostela, Spain
`alvaro.vazquez@usc.es`
[3] CITIUS, Universidade de Santiago de Compostela, Spain
`juancarlos.pichel@usc.es`

Abstract. Message Passing Interface (MPI) is the predominant and most extensively utilized programming model in the High Performance Computing (HPC) area. The standard only provides bindings for the low-level programming languages C, C++, and Fortran. While efforts are being made to offer MPI bindings for other programming languages, the support provided may be limited, potentially resulting in functionality gaps, performance overhead, and compatibility problems. To deal with those issues, we introduce MPI4All, a novel tool aimed at simplifying the process of creating efficient MPI bindings for any programming language. MPI4All is not dependent on the MPI implementation, and adding support for new languages does not require significant effort. The current version of MPI4All includes binding generators for Java and Go programming languages. We demonstrate their good performance with respect to other state-of-the-art approaches.

Keywords: Parallel computing · MPI · Bindings · Java · Go.

1 Introduction

Message Passing Interface (MPI) [13] is the most widely used and dominant programming model in High Performance Computing (HPC). In MPI, processes make explicit calls to library routines defined by the MPI standard to communicate data between two or more processes. These routines include both point-to-point (two party) and collective (many party) communications. Quality implementations can be found from prominent open-source projects like MPICH [5] and OpenMPI [6], as well as from software and hardware vendors of HPC systems (for instance, Intel MPI).

Traditionally, in the pursuit of raw performance, HPC has been closely tied to software development using low-level compiled languages such as C/C++ and Fortran. However, in contemporary science and research, high-level programming languages like Python, Go, and Julia play a crucial role. They offer researchers a user-friendly platform for developing complex algorithms, analyzing vast datasets, and implementing sophisticated models, even for those with limited programming experience. In parallel programming, high-level languages face limitations in their support for MPI. These languages prioritize abstraction and ease of use, which can conflict with the low-level nature of MPI. While efforts exist to provide MPI bindings for these languages, support may be limited, leading to gaps in functionality, performance overhead and compatibility issues.

In this paper, we introduce MPI4All[4], a novel tool designed to simplify the creation of MPI bindings for any programming language. Adding support for a new language only requires writing a *generator* code (in any language) that maps MPI C macros, functions, and data types, automatically obtained by MPI4All, to the target language. It is important to note that unlike other approaches, MPI4All is not tied to a specific MPI implementation (such as OpenMPI or MPICH), it is compatible with all of them. Another important limitation of the existent MPI bindings is that they do not support the complete MPI API, or due to their lack of support, they only implement functions for old MPI versions. In our case, if MPI were to release a new version, running MPI4All again would suffice to generate complete API bindings for the desired MPI implementation and language. In other words, we will obtain the bindings for the new MPI version in seconds. Therefore, once there is an implementation of the *generator* code for a target language, it can be reused.

As illustrative examples, currently MPI4All provides bindings for Java and Go programming languages. We have evaluated them with respect to other state-of-the-art bindings when running different MPI applications to demonstrate the efficiency and completeness of the ones generated by our tool.

The remainder of this paper is organized as follows. Section 2 discusses related work, Section 3 explains the MPI4All architecture and how to create MPI bindings for a particular language using our tool. Section 4 shows the performance evaluation of the Java and Go MPI bindings built using MPI4All. Finally, the conclusions derived from this work are presented.

2 Related Work

The MPI Standard [13] only provides bindings for C and Fortran programming languages[5]. Overall, people interested in MPI features but who do not use either C or Fortran have been relying in unofficial MPI-like solutions. Languages supported by these implementations include C++, Java, C#, Go, Julia, Python, among others. However, some of these implementations are not currently maintained and only provide support for an outdated MPI version or implement a reduced set of characteristics.

Java is one of the most popular object-oriented languages and is widely used, for instance, in Big Data processing. Therefore, many efforts have been done

[4] It is publicly available at `https://github.com/citiususc/mpi4all`

[5] C++ bindings where introduced in the MPI 2.0 specification but have been removed since the MPI 3.0 specification.

in using Java for parallel programming. Mainly, proposed Java MPI libraries subscribe to one of these three approaches:

– Relying on standalone Java APIs in order to provide a fully portable solution. Though it was followed by some MPI libraries, it finally proved out not to be practical since all of the MPI features need to be re-implemented.
– Relying on native MPI libraries for communication using the Java Native Interface (JNI), as it was adopted by solutions like mpiJava [10], and more recently the OpenMPI [16] and MVAPICH2 [7] official Java bindings. JNI allows Java programs to invoke functions and methods written in other languages including C.
– A hybrid approach, taken by MPJ Express [8] and FastMPJ [15], where message-passing libraries have custom device and network layers implemented in Java combined with JNI communication devices that call native methods.

The third approach, of implementing the MPI standard in Java aiming for high-performance communications, requires substantial development and maintenance effort. As a consequence, any change in the network and/or MPI standard version implies modifications in the code at low-level. On the other hand, the second approach allows easier development and maintenance. In order to get high-performance for Java MPI libraries, it keeps the Java layer as minimal as possible and uses JNI to invoke MPI methods implemented by native production-quality MPI libraries. The downside is that the JNI introduces a substantial amount of time overhead due to additional memory copying operations and requires recompiling the native code when porting the application to a new computer.

Currently, OpenMPI Java and FastMPJ are practically the only two well-maintained Java MPI libraries in the community. The MVAPICH2 Java implementation [7] is still in a maturation phase.

In the absence of a standard API for Java, older implementations [8, 10] follow the mpiJava 1.2. API proposed by the Java Grande Forum (JGF) in late 90s. FastMPJ [15] has support for both mpiJava 1.2 and the MPJ API, a minor upgrade to the mpiJava 1.2 API. On the other hand, the Java OpenMPI library [16] implements a custom API that is an extension of the MPJ API. Likewise, the Java MVAPICH2 bindings have adopted the OpenMPI Java API in order to facilitate end users. Though MPI4All follows the official MPI C interface style, a simple wrapper would solve the hypothetical necessity to comply with any of the proposed Java APIs.

There are other languages that include some kind of support for MPI. For example, Python supports MPI through the MPI4Py implementation [11], which underlies on the standard MPI-2 C++ bindings. The last version is compatible with both Python 2 and Python 3, and it supports various MPI-2 implementations like OpenMPI, MPICH, and Intel MPI. JuliaMPI.jl [9] is a Julia package for MPI. Though it supports up to MPI 3.1 many features are not yet available. Also, it does currently not support high performance networks such as InfiniBand, which limits its scalability to large problems. There were also several attempts to implement MPI bindings for Go programming language [1,3,4],

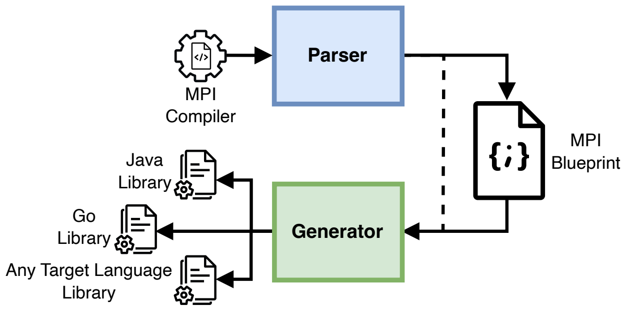

Fig. 1. Architecture of MPI4All.

but available distributions implement the MPI Standard only partially or stop keeping updating to new MPI versions.

3 MPI4All

In this section we will explain in detail the MPI4All architecture and how to proceed in order to build MPI bindings for a particular target language. MPI4All is composed of two distinct modules: the *Parser* and the *Generator*, as depicted in Figure 1. The *Parser* takes an MPI compiler installed on the system as input. Its output, known as the *blueprint*, can be saved in JSON format. This blueprint is then utilized by the *Generator* to produce bindings for a particular programming language. The modular design allows the *Parser* and *Generator* to function independently, with the ability to exchange the blueprint file as needed.

3.1 Parser

The Parser module is in charge of collecting functions, data types and variables from an MPI implementation (for example, MPICH) and then creating a structured blueprint to organize all the extracted information. The information is obtained from the MPI compiler, which by default searches for the compiler in the system's PATH using the common names (mpicc, mpicxx, mpicpp, etc.), although the user can specify the compiler to use. Once the compiler is detected, the parsing task is carried out in two stages: extraction and typing.

The extraction phase involves retrieving the functions, types, and variables defined by MPI, that are identifiable by the prefix `MPI`. For example, functions like `MPI_Send` and `MPI_Recv`, data types like `MPI_Comm`, and variables like `MPI_COMM_WORLD` fall under this category. First, the preprocessor-defined macros are examined, followed by the symbols. During this initial phase, distinguishing between a variable and a data type is challenging, but the functions can be correctly detected.

```
{
  "macros": [
    ...
    {
      "raw": "#define MPI_COMM_WORLD ((MPI_Comm)0x44000000)",
      "name": "MPI_COMM_WORLD",
      "value": "((MPI_Comm)0x44000000)",
      "type": "MPI_Comm",
      "var": true
    },
    ...
  ],
  "functions": [
    ...
    {
      "header": "int MPI_Send (const void *, int, MPI_Datatype, int, int, MPI_Comm)",
      "rtype": "int",
      "name": "MPI_Send",
      "args": [
    {
      "type": "const void *",
      "name": "buf"
    },
    {
      "type": "int",
      "name": "count"
    },
    {
      "type": "MPI_Datatype",
      "name": "datatype"
    },
    {
      "type": "int",
      "name": "dest"
    },
    {
      "type": "int",
      "name": "tag"
    },
    {
      "type": "MPI_Comm",
      "name": "comm"
    }
    ]
    },
  ...
  ],
  "types": {
    ...
    "int": "4",
    ...
    "MPI_Comm": "int",
    ...
  },
}
```

Fig. 2. Example of a blueprint fragment using MPICH 4.1.

The typing phase serves to identify the type of the extracted information. This allows us to differentiate between variables and types, assigning the appropriate type to variables, as well as to the parameters and return values of each function. This process is carried out through different C and C++ tests that are compiled and executed. If a test fails to compile or compiles but fails to execute correctly, it indicates, for instance, whether we are dealing with a variable or a data type. Furthermore, this process identifies type size and aliases to gather all the necessary information, which may be required in the *Generator* module.

Figure 2 shows an example of a blueprint fragment generated with MPICH v4.1. In this example, we can see three sections: *macros*, *functions*, and *types*. The *macros* store preprocessor definitions, where we can observe the macro name, its associated type, and whether it is a variable or defines a MPI-specific type. Regarding *functions*, the most important data includes the function name, the types of arguments and return, and additional information such as parameter names or their C headers, which are useful for improving the readability of the target bindings. Finally, the *types* section contains information about all types used in macros and functions. These types are first mapped to native language types, and in the case of native types, the number of bytes they occupy is indicated. This can assist languages in finding type equivalences based on their names and sizes.

3.2 Generator

The Generator is in charge of generating the source code for a particular language bindings following a blueprint. Its goal is mapping the MPI C macros, functions, and data types included in the blueprint to the target language. The generation process is different for each programming language and employs different strategies for code generation. The current implementation of MPI4All includes generator scripts for Java and Go. However, since generators have no dependencies between them, they can be implemented as independent projects, allowing for the creation of new generators by three party users using a blueprint. It is not mandatory for the generator implementation to be in the same programming language as the target bindings. For instance, MPI4All, implemented in Python, provides generators for Go and Java programmed in that language.

As we commented previously, the implementation of a generator must take into account the interoperability of C with the target language. As illustrative examples, we will describe the design in terms of the implementation of the Java and Go generator scripts, which can be generalized to support other languages. The interoperability between C and Go is facilitated by the `cgo` tool, which allows easy calling of C functions from Go and vice versa. Likewise, in Java, the new Foreign Function Interface (FFI) allows for bidirectional interaction between Java and C, facilitating the integration of native code into Java applications. In the Go approach, interoperability is facilitated by including C headers directly into Go code, allowing seamless interaction between the two languages. This integration is achieved by compiling the combined codebase, ensuring that Go and C components work harmoniously together. In contrast, Java's FFI interacts with pre-compiled native libraries. These libraries are linked dynamically at runtime, enabling Java to access and utilize functions defined within the C code. Java communicates with these libraries using the symbols they contain, providing a bridge for the execution of native code within Java applications.

Macros and data types. In Go and Java, in any case, it is not possible to directly call MPI C functions because macros defined in the headers cannot be accessed. Go lacks access to compiler macros, and in Java, such information is removed after compilation. This behavior is common to other programming languages.

To deal with this issue, MPI4All uses a hybrid strategy, which requires two steps. Firstly, it generates a C auxiliary library. The primary function of this library is to convert all macros stored in the blueprint into data types or variables. For instance, while languages like Java require a separate file for this purpose, Go allows embedding C code directly within a string in the code. Nonetheless, the procedure still involves iterating over all macros in the blueprint and generating C code using the following pattern:

– Variable:

```
[type] [PREFIX][name] = [name];
```

```
1  #include<mpi.h>
2
3  MPI_Comm GO_MPI_COMM_WORLD = MPI_COMM_WORLD;
4  ...
5  MPI_Datatype GO_MPI_DOUBLE = MPI_DOUBLE;
6  ...
7  int GO_MPI_THREAD_SINGLE = MPI_THREAD_SINGLE;
8  ...
9  typedef int GO_MPI_Fint;
10 ...
```

Fig. 3. Example of auxiliary C code output from the generator script.

```
1  var MPI_COMM_WORLD = C.GO_MPI_COMM_WORLD;
2  ...
3  var MPI_DOUBLE = C.GO_MPI_DOUBLE;
4  ...
5  var MPI_THREAD_SINGLE = C.GO_MPI_THREAD_SINGLE;
6  ...
7  type MPI_Fint = C.GO_MPI_Fint;
8  ...
9  type MPI_Comm = C.MPI_Comm;
10 ...
```

Fig. 4. Example of Go macro binding output from the generator script.

– Data type:

```
typedef [type] [PREFIX][name];
```

where the pattern names correspond to blueprint fields and `PREFIX` represents any chosen value defined in the generator.

Once applied to all macros in the blueprint, we would have a result similar to Figure 3. Note that it is necessary to include the MPI header (line 1) to compile the library. Once the symbols corresponding to the macros are generated, we can use them from the target language.

The process in the second step is similar to the previous one, iterating over all the macros in the blueprint but generating code in the target language as shown in Figure 4 for Go. Note that in the process, variables in the target language with the same name than the macro in C are assigned to the corresponding ones in the C auxiliary library. In this phase, we can also iterate over the data types defined in the blueprint and generate them along with the macros following the same procedure (line 9).

The second step for Java is slightly more complicated because it is not possible to map compiled data types in the native C library. The FFI uses a class called `MemorySegment`, which represents a memory address range of a known size using a `MemoryLayout`. Consequently, MPI4All defines MPI types as Java classes that extend a common class containing the `MemorySegment` that emulate different types. The `MemoryLayout` size of the `MemorySegment` are determined through the *types* section of the blueprint. The variables are defined as instances of those classes and are assigned to static attributes. In the case of classes representing primitive types, these are converted to simplify the API.

```go
...
func MPI_Send(buf unsafe.Pointer /*(const void *)*/, count C_int, datatype
      C_MPI_Datatype, dest C_int, tag C_int, comm C_MPI_Comm) error {
    return mpi_check(C.MPI_Send(buf, count, datatype, dest, tag, comm))
}
...
```

Fig. 5. Example of Go function binding output from the generator script.

Functions. The final task of the generator script is to map the functions as described in the blueprint to the target language. This involves generating each function using the syntax and conventions of the considered language, using the types specified in the previous step. In the function body, we call the corresponding C function, ensuring conversion of arguments and return types, and taking care of C error codes. Figure 5 presents the Go code for the `MPI_Send` function. Parameters are seamlessly passed to the C function without conversion, as we have defined their types as aliases of C types. Additionally, we have introduced the auxiliary function `mpi_check` to handle return values, converting MPI function return codes into `error` when they are not equal to 0.

The equivalent process in Java involves more steps. First, we need to locate the function symbol within the auxiliary C library generated previously. Then, we define the type for each parameter and return value using `MemoryLayout`. Finally, we can use the function definition to invoke it from Java. While primitive types are automatically converted into `MemorySegment` using the function layout, complex types must have their layout defined manually. Similar to Go, an auxiliary function `mpiCheck` is defined in Java. This function will check the return code of the MPI function and throw a `RuntimeException`, analogous to how Go returns an `error`.

4 Experimental Evaluation

Next we will evaluate the performance of the bindings for Go and Java that currently can be generated by MPI4All. With that goal in mind, we will compare their performance with respect to other state-of-the-art solutions. Since MPI4All is agnostic regarding the MPI implementation considered, unlike other approaches, we will prove that it is capable of generating bindings for OpenMPI and Intel MPI, for example.

Experiments were conducted using up to 8 computing nodes of the FinisTerrae III [2] supercomputer installed at CESGA (Spain). Each node contains two 32-core Intel Xeon Ice Lake 8352Y @2.2GHz processors and 256 GB of memory interconnected with Infiniband HDR 100. A 100Gb Ethernet network is also available on all nodes. It is a Linux cluster running Rocky Linux v8.4 (kernel v4.18.0).

4.1 Java

In this section we present the evaluation results carried out over three different Java MPI implementations. To provide an estimation of MPI4All performance we ran some of the NAS parallel benchmarks ported to Java described in [12]. We have introduced a few modifications in the source code in order to adapt them to the MPI4All Java bindings. The subset of the Java NAS parallel benchmarks selected for evaluation purposes is the following:

- CG – It uses a Conjugate Gradient method to compute approximations to the smallest eigenvalues of a sparse unstructured matrix.
- EP – This benchmark, short for Embarrassingly Parallel, is designed to measure the performance of a parallel application that consists of independent tasks that can be executed concurrently without any communication or synchronization between them.
- FT – It contains the computational kernel of a 3D Fast Fourier Transform (FFT).
- SP – It is a simulated Computational Fluid Dynamics (CFD) application. It solves a Scalar Pentadiagonal system of linear equations.
- MG – It uses a V-cycle Multi Grid method to compute the solution of the 3D scalar Poisson equation.

We use class D benchmarks, which correspond to considerably large problem sizes. A strong scaling test was conducted for each benchmark using up to 128 cores (in 4 nodes, 32 cores per node) of the Finisterrae III cluster and the Infiniband HDR 100 interconnection. We measured the performance using JDK 21.0.1, and Java bindings were generated for OpenMPI v4.1. Each measurement was computed as the median of five executions. Note that each new JDK release starting from version 19 has required to generate new MPI4All Java bindings. This is due to the fact that the Java FFI is currently a preview feature of the Java Platform still subject to changes. This API will be upgraded to permanent features in the next JDK 22 release, so it is expected to remain stable. Nevertheless, in our case the JDK update barely involved a change in a couple lines of code in the generation process of MPI4All Java bindings.

Also, for comparison purposes we selected and ran the same subset of the NAS parallel benchmarks over two representative MPI Java implementations: the FastMPJ library [15], and the official OpenMPI Java bindings [16]. The former uses JNI to invoke networking native library primitives (including Infiniband ones) while the later uses JNI to call MPI C primitives.

We represent in Figure 6 the speedups obtained for both FastMPJ library and the OpenMPI Java bindings using as reference the execution times measured in the corresponding MPI4All tests.

Overall, the OpenMPI Java bindings present degraded performance figures mainly due to the overhead introduced by the JNI calls to the MPI native library. MPI4All invokes the same MPI C library functions using FFI. Therefore, in this case, the performance improvement of the MPI4All Java bindings comes from using FFI instead of JNI to minimize the cost associated with copying data from Java to C.

On the other hand, FastMPJ presents better performance numbers than the generated MPI4All Java bindings except for the MG benchmark, where MPI4All

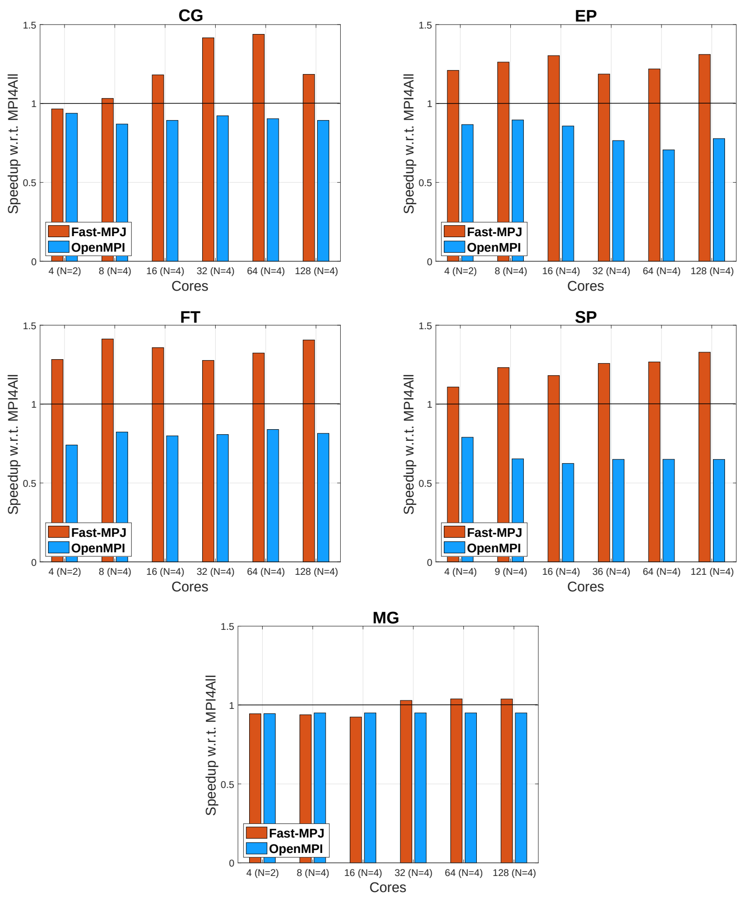

Fig. 6. Speedup of the different Java MPI implementations when executing the NAS Parallel Benchmarks (class D) using as reference the bindings for OpenMPI generated by MPI4All. N is the number of computing nodes.

is competitive with FastMPJ. FastMPJ relies on a highly efficient Java implementation of MPI-like functions aiming to provide a similar performance as native MPI implementations. However, current version only provides support for MPI-2, and upgrading or incorporating new functionalities requires a huge code programming effort.

4.2 Go

In this section, we will evaluate the performance obtained by the MPI Go bindings generated by MPI4All. In particular, we have generated bindings for Intel MPI version 2021.10.0. Go version 1.20.4 was used for compiling and deploying purposes. In the experiments we have considered Ember [14], which is a

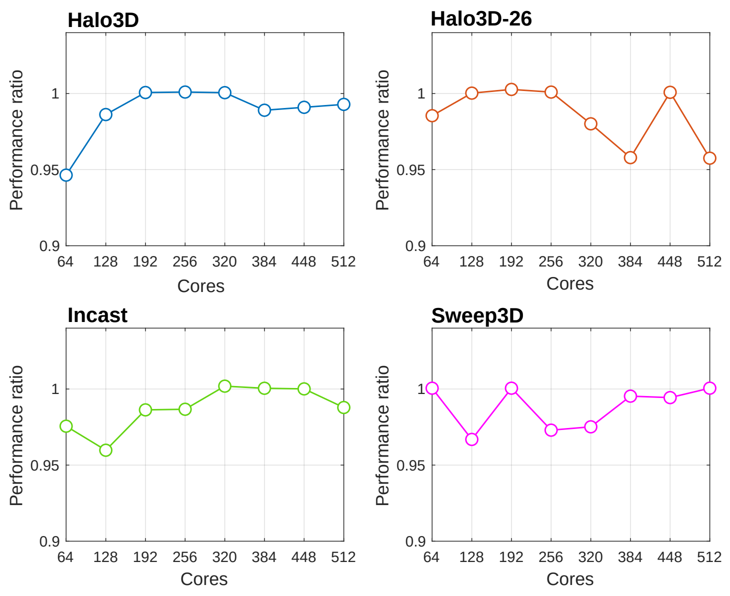

Fig. 7. Performance ratio between the original MPI C implementation and the Go one that uses the language bindings generated by MPI4All when running the Ember benchmarks.

communication pattern library developed at Sandia National Labs (USA). It is part of the Exascale Computing Project (ECP) proxy applications suite[6]. The Ember code was originally implemented in C using MPI and represents simplified communication patterns that are relevant to extreme scaled supercomputing systems. Four communication patterns were studied:

- Halo3D: It performs a structured nearest neighbor communication. In this pattern, each MPI rank communicates with ranks that are adjacent to it in each cartesian dimension. The *halo* exchanged is the data on each face.
- Halo3D-26: In this pattern, each MPI rank communicates with ranks along each cartesian face, as well as each edge of the local grid and each vertex. It represents a typical unstructured nearest neighbor communication.
- Incast: The purpose of this benchmark is to represent small collections of nodes which attempt to simultaneously send messages to the same remote node, similar to some parallel I/O systems.
- Sweep3: There are many scientific applications that have a strong level of dependencies, which affect their communication patterns. This benchmark attempts to mimic that behavior by decomposing a 3D data domain over a 2D array of MPI ranks.

[6] `http://proxyapps.exascaleproject.org/ecp-proxy-apps-suite`

Our experimentation focuses on comparing the performance of the original C Ember benchmark with our Go port implementation, which utilizes MPI4All bindings (see Figure 7). Through this comparison, we aim to evaluate the efficacy of our ported solution against the established C implementation. The experiment was conducted using up to 8 nodes of the Finisterrae III cluster (comprising a total of 512 cores, 64 per node). The benchmarking process involved running each of the four communication patterns, both in C and Go, a total of five times. Subsequently, the median of the values obtained in each run for each pattern and language was taken.

Figure 7 shows that the MPI4All-generated Go bindings achieve similar performance comparable to the native Intel C MPI library for all four communication patterns since the ratio remained very close to 1 regardless of the number of nodes used. It also indicates that in terms of performance, Go and C are highly similar. One notable distinction between both languages is Go's garbage collector. However, given the absence of memory creation and destruction during the communication process, it should not significantly impact the measurements. The seamless integration of C into Go has enabled MPI4All to produce a binding library with performance nearly identical to that of a C implementation.

On the other hand, as we pointed out in Section 2, there are some other MPI Go bindings proposals [1, 3, 4]. However, to the best of our knowledge, none of them can implement the Ember benchmark in Go. This limitation arises from both the absence of necessary functionalities such as asynchronous communications and the lack of support for Intel MPI.

5 Conclusions

In this paper, we introduced MPI4All[7], an innovative tool aimed at facilitating the creation of MPI bindings for any programming language. Adding support for new languages merely requires the development of an script code that maps MPI C macros, functions, and data types to the target language. It is noteworthy that unlike other approaches, MPI4All is not bound to a specific MPI implementation; it is compatible with all of them. Once the script for some programming language is available, generating bindings for the complete API of any MPI implementation (and version) takes seconds. This assures completeness and avoids maintenance problems.

MPI4All includes scripts for generating MPI bindings for Go and Java programming languages. We evaluated them in terms of performance compared to other state-of-the-art solutions. The MPI4All Java bindings for OpenMPI clearly outperform the official Java OpenMPI bindings when running several NAS benchmarks. Although their performance is lower compared to FastMPJ, it is important to note that FastMPJ only supports MPI-2 routines. Regarding Go, the MPI4All bindings for Intel MPI achieve very similar performance to the native Intel C MPI library when running the Ember benchmarks.

[7] It is publicly available at `https://github.com/citiususc/mpi4all`

Acknowledgments. The authors would like to thank Guillermo López Taboada and Roberto Expósito for providing access to the FastMPJ library. This work was supported by Xunta de Galicia [ED431G 2019/04, ED431F 2020/08, ED431C 2022/16]; MICINN [PLEC2021-007662, PID2022-137061OB-C22, PID2022-141027NB-C22]; and European Regional Development Fund (ERDF). Authors also wish to thank CESGA (Galicia, Spain) for providing access to their supercomputing facilities

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. A Golang Wrapper for MPI. `https://github.com/yoo/go-mpi` [Online; accessed 26 feb 2024]
2. CESGA (Galician Supercomputing Center) - Computing Infrastructures. `https://www.cesga.es/en/infrastructures/computing/` [Online; accessed 26 feb 2024]
3. GoMPI: Message Passing Interface for Parallel Computing. `https://github.com/sbromberger/gompi` [Online; accessed 26 feb 2024]
4. MPI-binding package for Golang. `https://github.com/marcusthierfelder/mpi` [Online; accessed 26 feb 2024]
5. MPICH. `https://www.mpich.org`, [Online; accessed 26 feb 2024]
6. Open MPI. `https://www.open-mpi.org/`, [Online; accessed 26 feb 2024]
7. Al-Attar, K., Shafi, A., Subramoni, H., Panda, D.K.: Towards Java-based HPC using the MVAPICH2 Library: Early Experiences. In: IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). pp. 510–519 (2022)
8. Baker, M., Carpenter, B., Shafi, A.: MPJ Express: Towards Thread Safe Java HPC. In: 2006 IEEE International Conference on Cluster Computing. pp. 1–10 (2006)
9. Byrne, S., Wilcox, L.C., Churavy, V.: MPI. jl: Julia bindings for the Message Passing Interface. Proceedings of the JuliaCon Conferences **1**(1), 68 (2021)
10. Carpenter, B., Getov, V., Judd, G., Skjellum, A., Fox, G.: MPJ: MPI-like message passing for Java. Concurrency: Practice and Experience **12**(11), 1019–1038 (09 2000)
11. Dalcin, L., Fang, Y.L.L.: mpi4py: Status update after 12 years of development. Computing in Science & Engineering **23**(4), 47–54 (2021)
12. Mallón, D.A., Taboada, G.L., Touriño, J., Doallo, R.: NPB-MPJ: NAS Parallel Benchmarks Implementation for Message-Passing in Java. In: 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing. pp. 181–190 (2009)
13. Message Passing Interface Forum: MPI: A message-passing interface standard version 4.1. `https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf` [Online; accessed 26 feb 2024] (2023)
14. Sandia National Laboratories: Ember Communication Pattern Library (2018), `https://github.com/sstsimulator/ember` [Online; accessed 26 feb 2024]
15. Taboada, G.L., Touriño, J., Doallo, R.: F-MPJ: scalable Java message-passing communications on parallel systems. The Journal of Supercomputing **60**, 117–140 (2012)
16. Vega-Gisbert, O., Roman, J.E., Squyres, J.M.: Design and implementation of Java bindings in Open MPI. Parallel Computing **59**, 1–20 (2016)