


Towards Universal MPI Bindings for Enhanced New Language Support

César Piñeiro ^{1,2}[0000-0001-6490-7128], Álvaro Vázquez²[0000-0002-4719-8099],
and Juan C. Pichel^{1,2}[0000-0001-9505-6493]

¹ CITIUS, Universidade de Santiago de Compostela, Spain
cesaralfredo.pineiro@usc.es, juancarlos.pichel@usc.es

² Electronics and Computer Science Department, Universidade de Santiago de
Compostela, Spain
alvaro.vazquez@usc.es

Abstract. In the field of High Performance Computing (HPC), Message Passing Interface (MPI) is the most widely used and prevalent programming model. Only the low-level programming languages C, C++, and Fortran have bindings available in the standard. Although there are attempts to provide MPI bindings for other programming languages, these may be limited, which could lead to incompatibilities, performance overhead, and functional gaps. To address those problems, we present MPI4All, a brand-new tool designed to make the process of developing effective MPI bindings for any programming language more straightforward. Support for additional languages can be added with little difficulty, and MPI4All is independent of the MPI implementation. Programming language binding generators for Go and Java are included in the most recent version of MPI4All. We demonstrate their good performance results with respect to other state-of-the-art approaches. This work is an extended version of the ICCS-2024 conference paper [20].

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

1 Introduction

The Message Passing Interface (MPI) [15] stands as the most prevalent and dominant programming model within High Performance Computing (HPC). In the MPI framework, processes explicitly invoke library routines specified by the MPI standard to facilitate data exchange between two or more processes. These routines encompass both point-to-point (two-party) and collective (many-party) communications. High-quality implementations are available from well-known open-source initiatives like MPICH [5] and OpenMPI [6], as well as from software and hardware providers in the HPC sector (such as Intel MPI).

Historically, the quest for raw performance has closely associated HPC with software development utilizing low-level compiled languages, including C/C++ and Fortran. Nevertheless, in modern scientific research, high-level programming languages like Python, Go, and Julia are increasingly significant. These

languages provide researchers with an accessible platform for crafting intricate algorithms, examining large datasets, and deploying advanced models, even for individuals with minimal programming expertise. However, in the realm of parallel programming, high-level languages encounter challenges in their support for MPI. These languages emphasize abstraction and user-friendliness, which can sometimes conflict with the low-level requirements of MPI. Although there are initiatives aimed at offering MPI bindings for these languages, the level of support may be inadequate, resulting in functionality gaps, performance overhead, and compatibility challenges.

This paper presents an extended version of our previous conference paper [20], which introduced MPI4All³, an innovative tool aimed at simplifying the creation of MPI bindings for various programming languages. While the initial paper outlined the fundamental concepts of MPI4All, this version includes significant new features, such as a new experimental results section, an extended related work, detailed explanations of how the bindings were generated, and practical examples demonstrating how to implement applications using MPI4All. Note that adding support for new languages in MPI4All 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. In this manner, unlike other approaches, MPI4All is not tied to a specific MPI implementation (such as OpenMPI or MPICH) and is compatible with all of them. Another significant limitation of existing MPI bindings is that they do not support the complete MPI API, or, due to their lack of support, they only implement functions for older MPI versions. In our case, if MPI were to release a new version, rerunning MPI4All would suffice to generate complete API bindings for the desired MPI implementation and target language. In other words, we can obtain the bindings for the new MPI version in seconds. Therefore, once a *generator* code for a target language is implemented, it can be reused.

As illustrative examples, MPI4All currently provides bindings for the Java and Go programming languages. We have evaluated these bindings against other state-of-the-art options while running different MPI scientific applications to demonstrate the efficiency and completeness of those 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 provides usage examples of the API bindings and Section 5 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 [15] only provides bindings for C and Fortran programming languages. Overall, people interested in MPI features but who do not use either

³ It is publicly available at <https://github.com/citiususc/mpi4all>

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.

The support for C++ in MPI was removed in MPI 3.0 specification due to the high maintenance cost and minimal advantages over the C interface. Although the C++ bindings offered little additional functionality compared to modern C++ practices, they significantly burdened the MPI specification. The introduction of ISO C++11 brought substantial improvements to the language, reigniting interest in create modern C++ bindings for MPI. As a result, support has been maintained in various MPI implementations, such as MPICH, while libraries like Boost.MPI [9] offer a C++ friendly interface to the standard MPI. Moreover, there are community proposals focused on refining and enhancing the C++ interface to better align with contemporary language standards and capabilities [21].

Java is one of the most widely used object-oriented programming languages, particularly in applications such as Big Data processing. Typically, Big Data applications process large data sets in parallel using the MapReduce programming model, implemented in well-known open-source Java-based frameworks like Hadoop or Spark. However, there has also been interest in utilizing Java for HPC applications. Moreover, the convergence of HPC and Big Data has received increasing attention in recent years, leading to the emergence of frameworks that integrate compute-intensive MPI tasks with conventional data-intensive MapReduce operations to develop hybrid applications [19].

Consequently, significant efforts have been made to utilize Java for parallel programming. Proposed Java MPI libraries generally follow to one of the following two approaches:

- **Java Bindings of Native MPI Libraries via JNI:** These type of methods implement Java bindings for communication with native MPI libraries through the Java Native Interface (JNI). Solutions such as mpiJava [12], along with the more recent official Java bindings for OpenMPI [18] and MVAPICH2 [7], exemplify this approach. JNI enables Java programs to call functions and methods written in other programming languages, including C.
- **Standalone Java Messaging Libraries for portability:** This approach, employed by MPJ Express [10] and FastMPJ [17], aims to provide a fully Java portable solution. These methods implement a message-passing API on top of a Java communication middleware, usually based on low-level Java communication APIs such as Java Remote Method Invocation (RMI) or Java sockets. Specific high-performance solutions for low-latency networks, such as InfiniBand, have been implemented, for instance, through Java Fast Sockets [17].

The second approach requires a lot of development and continuous maintenance because it uses Java to implement the MPI standard for high-performance

communication. This means that any updates to the MPI standard version or the network require low-level code changes.

On the other hand, the first method makes development and upkeep easier. It minimizes the Java layer and uses JNI to call MPI methods provided by native, production-quality MPI libraries in order to achieve high-performance Java MPI libraries. Nevertheless, JNI introduces considerable time overhead due to extra memory copying and requires recompiling the native code when migrating the application to a new machine.

One way to mitigate the amount of code required could involve using an intermediate language like SWIG (Simplified Wrapper and Interface Generator) [23], which allows automatic generation of bindings for multiple languages. However, the main issue is that using SWIG for MPI would require implementing and maintaining the entire standard within SWIG. Additionally, the number of supported languages is quite limited, and in the case of Java, it generates JNI-based code, making it incompatible with the Foreign Function Interface (FFI) approach explored in this work as a replacement for JNI due to the novelty of this approach.

Currently, OpenMPI Java and FastMPJ are essentially the only two well-maintained Java MPI libraries available in the community. The MVAPICH2 Java implementation [7, 8] is still in the maturation phase. In the absence of a standard API for Java, older implementations [10, 12] follow the mpiJava 1.2. API proposed by the Java Grande Forum (JGF) in late 90s. FastMPJ [17] has support for both mpiJava 1.2 and the MPJ API, a minor upgrade to the mpiJava 1.2 API. Conversely, the Java OpenMPI library [18] implements a custom API that extends the MPJ API. Similarly, the Java MVAPICH2 bindings have adopted the OpenMPI Java API to streamline usability for end users. Though MPI4All follows the official MPI C interface style, a simple wrapper could address the hypothetical requirement for compliance 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 [13], 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. Additionally, newer alternatives such as mpiPython [22] have emerged, providing enhanced performance along with a communication API that closely resembles C. This design choice facilitates a smoother transition for users familiar with traditional MPI implementations, allowing them to leverage Python’s ease of use while benefiting from optimized performance.

Finally, JuliaMPI.jl [11] 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], but available distributions implement the MPI Standard only partially or stop keeping updating to new MPI versions.

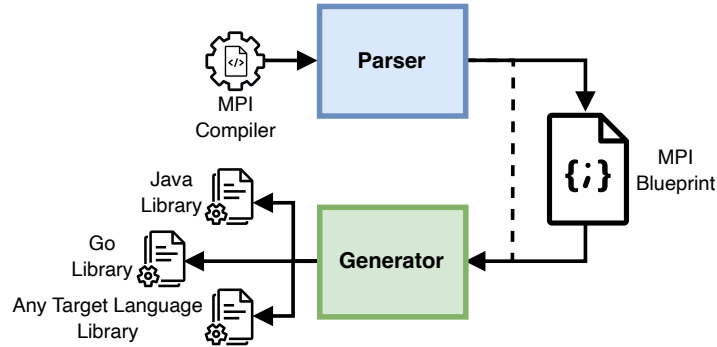


Fig. 1. Architecture of MPI4All.

3 MPI4All

This section will provide a detailed explanation of the MPI4All architecture and the steps involved in creating MPI bindings for specific target languages.

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 designed to collect variables, functions, and data types from an MPI implementation (like MPICH), and then organizing all the extracted data into a structured blueprint. The information is retrieved from the MPI compiler, which, by default, looks for the compiler in the system's PATH using common names (mpicc, mpicxx, mpicpp, etc.), although the user has the option to specify a different compiler. Once the compiler is identified, the parsing process takes place in two phases: 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.

The purpose of the typing stage is to determine the kind of data that has been extracted. This enables us to distinguish between types and variables, giving

```

1  {
2  "macros": [
3  ...
4  {
5  "raw": "#define MPI_COMM_WORLD
        ↪ ((MPI_Comm)0x44000000)",
6  "name": "MPI_COMM_WORLD",
7  "value": "((MPI_Comm)0x44000000)",
8  "type": "MPI_Comm",
9  "var": true
10 },
11 ...
12 ],
13 "functions": [
14 ...
15 {
16 "header": "int MPI_Send (const void
        ↪ *, int, MPI_Datatype, int,
        ↪ int, MPI_Comm)",
17 "rtype": "int",
18 "name": "MPI_Send",
19 "args": [
20 {
21 "type": "const void *",
22 "name": "buf"
23 },
24 {
25 "type": "int",
26 "name": "count"
27 },
28
29 {
30 "type": "MPI_Datatype",
31 "name": "datatype"
32 },
33 {
34 "type": "int",
35 "name": "dest"
36 },
37 {
38 "type": "int",
39 "name": "tag"
40 },
41 {
42 "type": "MPI_Comm",
43 "name": "comm"
44 }
45 ],
46 ...
47 ],
48 "types": {
49 ...
50 "int": "4",
51 ...
52 "MPI_Comm": "int",
53 ...
54 },
55 }

```

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

each function’s parameters and return values, as well as its variables, the proper type. Several C and C++ tests that are compiled and run are used to carry out this process. For example, it can reveal whether we are working with a variable or a data type if a test fails to compile or compiles but fails to run correctly. Additionally, in order to collect all the data that might be needed in the Generator module, this process determines type size and aliases.

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 responsible for creating source code based on a blueprint for a specific language binding. Its objective is to translate MPI C data types, functions, and macros included in the blueprint into the target language. Each programming language has a unique generation process that employs various code generation techniques. Each programming language has a different generation process that uses a different set of techniques to generate code. Generator scripts for Go and Java are part of the current MPI4All implementation. However, since the generators are independent, they can be developed as separate projects, allowing third-party users to use a blueprint to create new generators. The target bindings and generator implementations do not always need to be written in the same language. For example, MPI4All offers generators for Go and Java that are written in Python.

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. Because these libraries are dynamically linked during runtime, Java can access and make use of functions that are defined in the C code. Java uses the symbols these libraries contain to communicate with them, creating a bridge that allows native code to run within Java applications.

Macros and data types. Go and Java do not allow calling MPI C functions directly because the macros defined in the headers cannot be accessed. In Java, this type of data is removed during compilation, and Go lacks access to compiler macros as well. This limitation also applies 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. As illustrated in Figure 5, the FFI uses a class called `MemorySegment`, which represents a range of memory addresses of known size, this memory can either be newly or mapped from existing memory. Consequently, `MPI4All` defines MPI types as Java classes that extend a common class `Type` (line 4) containing the `MemorySegment` that emulate different types. The `SymbolLookup` class allows a symbol to be looked up by its name and returns the `MemorySegment` without size, using the `reinterpret`


```

1  ...
2  private static SymbolLookup lib = SymbolLookup.loaderLookup();
3  ...
4  public class Type {
5      public final MemorySegment ms;
6
7      Type(String n, int sz) {
8          this.ms = lib.find(n).get().reinterpret(sz);
9      }
10 }
11 ...
12 public class MPI_Comm extends Type{
13
14     MPI_Comm(String n){
15         super(n, (int)layout(/*size*/4).byteSize());
16     }
17 }
18 ...
19 public static final MPI_Comm MPI_COMM_WORLD = new MPI_Comm("J_MPI_COMM_WORLD");
20 ...

```

Fig. 5. Example of Java binding output from the generator script.

function, the appropriate size is assigned to the segment (line 8). The size of the `MemorySegment` are determined through the *types*, and this size is stored in the corresponding class representing the MPI type (line 15). The variables are defined as instances of those classes (line 19) and are assigned to static attributes. In the case of classes representing primitive types, these are converted to simplify the API.

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 6 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, as shown in Figure 7. First, we need to locate the function symbol within the auxiliary C library generated previously (line 5), the `SymbolLookup` gets the symbol and the `Linker` creates an object to perform the call. Then, we define the type for each parameter and return value using `MemoryLayout` (line 13). Finally, we can use the function definition to invoke it from Java (line 15). 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

```

1  ...
2  func MPI_Send(buf unsafe.Pointer /*(const void *)*/ , count C_int, datatype
3     C_MPI_Datatype, dest C_int, tag C_int, comm C_MPI_Comm) error {
4     return mpi_check(C.MPI_Send(buf, count, datatype, dest, tag, comm))
5  }
6  ...

```

Fig. 6. Example of Go function binding output from the generator script.

```

1  ...
2  private static SymbolLookup lib = SymbolLookup.loaderLookup();
3  private static final Linker linker = Linker.nativeLinker();
4  ...
5  private static MethodHandle findMethod(String name, FunctionDescriptor function){
6     java.util.Optional<MemorySegment> ms = lib.find(name);
7     if (ms.isEmpty()){
8         return null;
9     }
10    return linker.downcallHandle(ms.get(), function);
11 }
12 ...
13 private static final MethodHandle C_MPI_SEND = findMethod("MPI_Send",
14     FunctionDescriptor.of(ValueLayout.JAVA_INT, ValueLayout.ADDRESS,
15     ValueLayout.JAVA_INT, layout(4), ValueLayout.JAVA_INT, ValueLayout.JAVA_INT,
16     layout(4)));
17 ...
18 public static void MPI_Send(/*(const void *)*/ C_pointer<Void> buf, int count,
19     MPI_Datatype datatype, int dest, int tag, MPI_Comm comm) {
20     mpiCheck(()->C_MPI_SEND.invoke(buf.ms, count, datatype.ms, dest, tag, comm.ms));
21 }
22 ...

```

Fig. 7. Example of Java function binding output from the generator script.

MPI function and throw a `RuntimeException`, analogous to how Go returns an error.

4 Application Programming

This section outlines the process of implementing an MPI4All application, using the calculation of π through the integral approximation method as an illustrative example. The example demonstrates how to use MPI bindings for both Java and Go, guiding users through the steps required to integrate MPI library into applications written in these languages.

Figures 8 and 9 show an example of a parallelized calculation of π using MPI in Go and Java. The program first initializes the MPI environment with `MPI_Init` (line 4 Go, line 5 Java) and retrieves the rank of the current process and the total number of processes (lines 7-8 Go, lines 9-10 Java). The total number of rectangles n is defined, and the width h of each rectangle is calculated. Each process then computes its local sum by iterating over a subset of the rectangles (lines 14-15 Go, lines 17-18 Java), where the function $f(x) = \sqrt{1-x^2}$ is evaluated at the midpoint of each rectangle. The loop is distributed across processes by using the rank to determine which subset of the rectangles each process works

```

1  import ("fmt"; "math"; "mpi")
2
3  func check_error(err error) {if err != nil {panic(err)}}
4
5  func main() {
6      check_error(mpi.MPI_Init(nil, nil))
7
8      var rank, size mpi.C_int
9      check_error(mpi.MPI_Comm_rank(mpi.MPI_COMM_WORLD, &rank))
10     check_error(mpi.MPI_Comm_size(mpi.MPI_COMM_WORLD, &size))
11
12     const n = 1000000
13     h := 1.0 / n // Width of each rectangle
14     local_sum, total_sum := 0.0, 0.0
15     for i := int(rank) + 1; i <= n; i += int(size) {
16         x := (float64(i) - 0.5) * h // Midpoint for rectangle
17         local_sum += math.Sqrt(1.0 - x*x) // f(x)
18     }
19
20     check_error(mpi.MPI_Reduce(mpi.P(&local_sum), mpi.P(&total_sum), 1,
21         mpi.MPI_DOUBLE, mpi.MPI_SUM, 0, mpi.MPI_COMM_WORLD))
22
23     if rank == 0 {
24         fmt.Println("Estimated value of pi:", total_sum*h*4)
25     }
26     check_error(mpi.MPI_Finalize())
27 }

```

Fig. 8. Example of π calculation using Go bindings.

on. After the local sums are computed, the program uses `MPI_Reduce` (line 18 Go, line 24 Java) to combine the results from all processes, summing the local contributions on the master process with rank 0. Finally, the master process calculates and prints the approximation of π . The program concludes by finalizing the MPI environment with `MPI_Finalize` (line 23 Go, line 30 Java).

The Go code shown in Figure 8 is very similar to MPI programs written in C, with a few key differences due to Go's language characteristics. One main difference is that MPI functions in Go return a Go error type instead of an integer error code as in C. This aligns error handling with standard Go practices; we use `check_error` to trigger a panic if an error occurs. Another important difference is that MPI integers in Go must be declared as `mpi.C_int` (line 8), rather than the default Go `int`, because the number of bits in a Go `int` may differ from the C `int` used by the MPI library. This ensures compatibility with the MPI library, avoiding potential issues related to mismatched integer sizes. Additionally, functions like `MPI_Reduce` (line 20), which in C take a void pointer, require the use of an auxiliary function `mpi.P` to convert a Go pointer into a C pointer. This function is only for objects, Go slice/array must use `mpi.PA` to expose the memory instead of the Go object that represents the collections.

In the Java MPI code shown in Figure 9, there are some differences due to Java's memory management and the way Java interacts with native libraries. One key difference is the use of `Mpi.C_int` (lines 8-9) for MPI integers instead of Java's `int`. This is necessary because Java integers are managed by the garbage collector and do not have a fixed physical memory address that can be passed

```

1  import org.mpi.Mpi;
2  import java.lang.Math;
3
4  public class Main {
5      public static void main(String[] args) throws Throwable {
6          Mpi.MPI_Init(Mpi.C_pointer.NULL.cast(), Mpi.MPI_ARGVS_NULL);
7
8          var c_rank = Mpi.C_int.alloc();
9          var c_size = Mpi.C_int.alloc();
10         Mpi.MPI_Comm_rank(Mpi.MPI_COMM_WORLD, c_rank.pointer());
11         Mpi.MPI_Comm_size(Mpi.MPI_COMM_WORLD, c_size.pointer());
12
13         int n = 1000000;
14         int rank = c_rank.get(), size = c_size.get();
15         double h = 1.0 / n; // Width of each rectangle
16         double local_sum = 0.0;
17         for (int i = rank + 1; i <= n; i += size) {
18             double x = (i - 0.5) * h; // Midpoint for rectangle
19             local_sum += Math.sqrt(1.0 - x * x); // f(x)
20         }
21
22         var c_local_sum = Mpi.C_double.alloc();
23         var c_total_sum = Mpi.C_double.alloc();
24         c_local_sum.set(local_sum);
25         Mpi.MPI_Reduce(c_local_sum.pointer().cast(), c_total_sum.pointer().cast(), 1,
26             Mpi.MPI_DOUBLE, Mpi.MPI_SUM, 0, Mpi.MPI_COMM_WORLD);
27
28         if (rank == 0){
29             System.out.println("Estimated value of pi:" + c_total_sum.get() * h * 4);
30         }
31         Mpi.MPI_Finalize();
32     }
33 }

```

Fig. 9. Example of π calculation using Java bindings.

directly to C functions. In this case, `Mpi.C_int` acts as a wrapper that allocates memory outside of the Java heap, providing a direct pointer that can be passed to C-based MPI functions. Additionally, the primitive values stored in `Mpi.C_int` (or `Mpi.C_double` for floating-point numbers) are accessed and modified using getter (lines 14 and 28) and setter methods (line 24). Finally, when dealing with pointers in MPI functions like `MPI_Reduce` (line 25), the `pointer` method retrieves the memory address of the variable, and the `cast` method converts it to the `void*` type required by the function. In Java, error handling is managed through exceptions, which are automatically thrown and can be caught using try-catch blocks.

5 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 FinisTerae 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).

5.1 Java

In this section, we present the evaluation results obtained from testing four different Java MPI implementations. To estimate the performance of MPI4All, we ran a selection of the NAS parallel benchmarks, which have been ported to Java as described in the work by Mallón et al. [14]. We made some modifications to the source code in order to adapt these benchmarks to the MPI4All Java bindings. The specific subset of Java NAS parallel benchmarks chosen for the evaluation included five key benchmarks. The first benchmark, CG, implements a Conjugate Gradient method, which is used to compute approximations to the smallest eigenvalues of a sparse unstructured matrix. This benchmark is useful for testing iterative solver performance on large, sparse systems. The second benchmark, EP (Embarrassingly Parallel), is designed to measure the performance of parallel applications where independent tasks can be executed concurrently without requiring any communication or synchronization between them. It is ideal for systems where tasks can be easily split and distributed. FT, the third benchmark, focuses on the computational kernel of a 3D Fast Fourier Transform (FFT), testing the system’s ability to handle complex operations related to signal processing and scientific computing. The fourth benchmark, SP, simulates a Computational Fluid Dynamics (CFD) application. It solves a Scalar Pentadiagonal system of linear equations, which is typically encountered in modeling fluid flow or heat transfer. Note that the SP benchmark requires a square number of processors to run, rather than the more usual power-of-two configuration. Finally, MG utilizes a V-cycle Multi-Grid method to solve the 3D scalar Poisson equation. This benchmark is essential for evaluating the performance of hierarchical grid-based solvers, which are commonly used for problems involving partial differential equations. Together, these benchmarks provide a comprehensive assessment of the MPI4All Java bindings across different types of parallel computing tasks.

We used class D benchmarks, which correspond to considerably large problem sizes. A strong scaling test was conducted for each benchmark, except for the SP, using up to 128 cores (in 4 nodes, 32 cores per node), of the Finisterrae III cluster and the Infiniband HDR 100 interconnection. The corresponding strong scaling test for SP benchmark was carried out with various configurations of a square number of cores up to 121 (11^2). Performance was measured using JDK 22.0.1, and Java bindings were generated for OpenMPI v4.1 and IntelMPI v.2021.3.0. 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 was preview feature of the Java Platform subject to changes. This API has been upgraded to a permanent feature in the JDK 22 release, so it is expected to remain stable. Nevertheless, in our case the JDK update barely involves 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 [17], and the official OpenMPI Java bindings [18]. The former uses JNI to invoke networking native library primitives (including Infini-band ones) while the later uses JNI to call MPI C primitives. It is worth noting that the NAS benchmarks were also evaluated in the original papers introducing FastMPJ and the official OpenMPI Java bindings.

Figure 10 shows the speedups obtained for the FastMPJ library and the OpenMPI and IntelMPI bindings generated by MPI4All, using the official OpenMPI Java bindings as a reference. The scalability analysis reveals varying performance across different benchmarks. In the CG benchmark, MPI4All-IntelMPI stands out, achieving a $3.5\times$ speedup with 128 cores, while FastMPJ also surpasses OpenMPI, though to a lesser extent. MPI4All-OpenMPI remains close to the baseline but performs slightly better. For the EP benchmark, FastMPJ performs best, reaching up to $1.7\times$ speedup at higher core counts, with MPI4All-IntelMPI showing significant improvements, especially as core counts increase. Meanwhile, MPI4All-OpenMPI shows moderate benefits. In the FT benchmark, MPI4All-IntelMPI demonstrates excellent scalability, with FastMPJ following closely, while MPI4All-OpenMPI shows only minor gains over OpenMPI. In the SP benchmark, all bindings clearly outperform OpenMPI, with MPI4All-IntelMPI leading, achieving speedups up to $2.2\times$. Lastly, in the MG benchmark, MPI4All-OpenMPI shows modest improvements over the reference, and MPI4All-IntelMPI improves performance only at 64 and 128 cores. However, for this benchmark, FastMPJ performs worse than OpenMPI.

Therefore, when comparing the overall behavior of the four Java MPI bindings, their scalability and performance reveal clear distinctions. MPI4All-IntelMPI consistently shows the best scalability, performing significantly better than the other bindings as the number of cores increases. FastMPJ also delivers strong performance, though it slightly lags behind MPI4All-IntelMPI at higher core counts. However, a key drawback is that the current version of FastMPJ only supports MPI-2, and upgrading or incorporating new functionalities would require significant programming effort. MPI4All-OpenMPI generally mirrors the performance of the reference OpenMPI, offering similar results with only minor improvements in scalability. In contrast, the reference OpenMPI is outperformed by the other bindings, showing degraded performance mainly due to the overhead introduced by JNI calls to the MPI native library. Notably, the MPI4All bindings invoke the same MPI C library functions using FFI. Therefore, the performance improvement of the MPI4All Java bindings comes from using FFI

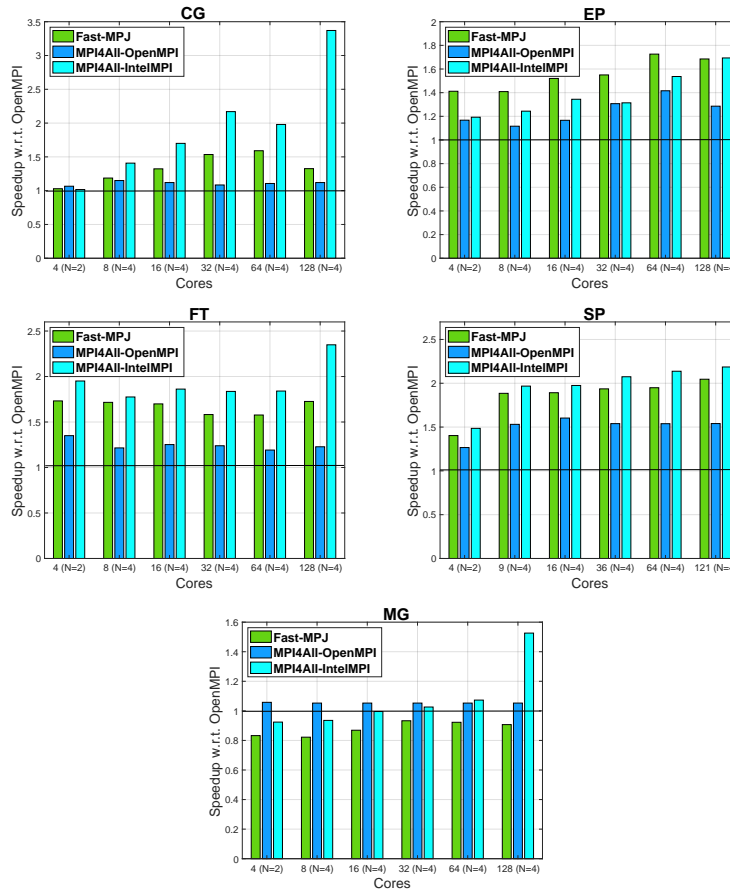


Fig. 10. Speedup of the different Java MPI implementations when executing the NAS Parallel Benchmarks (class D) using as reference the official OpenMPI Java bindings. N is the number of computing nodes.

instead of JNI, which minimizes the cost associated with copying data from Java to C.

5.2 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 our previous work [20], we have focused on evaluating the performance of communication patterns using the Ember benchmark suite⁴, originally devel-

⁴ <http://proxyapps.exascaleproject.org/ecp-proxy-apps-suite>

oped in C with MPI, which includes four key communication patterns: *Halo3D* (structured nearest neighbor communication), *Halo3D-26* (unstructured nearest neighbor communication), *Incast* (small groups of nodes sending messages to a single node, typical in parallel I/O systems), and *Sweep3* (communication with strong dependencies, mimicking scientific applications). Our experiments compared the performance of the C implementation with our Go port using MPI4All bindings. Results demonstrated that the Go implementation achieved performance nearly identical to the C version, with performance ratios close to 1 across all patterns, highlighting the efficiency of the MPI4All bindings. Despite Go’s garbage collector, the absence of memory allocation during communication minimized any performance impact.

In this extended study, we shift our focus from communication patterns to evaluating real Go-based scientific applications. We assess three key applications: the Jacobi method, a molecular dynamics simulation (md), and merge sort (msort). The *Jacobi method*, an iterative algorithm for solving systems of linear equations $A \cdot x = b$, was executed with a square matrix A of size $14k \times 14k$. We performed up to 1,000 iterations, with a stopping criterion based on an error tolerance of 1×10^{-6} . The *molecular dynamics simulation (md)* aimed to study particle motion over time. In this simulation, positions, velocities, and accelerations of 64k particles were updated using the velocity Verlet integration scheme, modeling their interaction through a central pair potential. Finally, the *merge sort (msort)* algorithm, a divide-and-conquer approach to sorting, recursively splits an array into smaller subarrays until each contains a single element, then merges them back together in sorted order. We applied this process to sort twenty different arrays, each containing 1 billion double-precision numbers, achieving a time complexity of $O(n \log n)$.

These applications are highly relevant for the scientific community. The *Jacobi method* is crucial for solving large systems of linear equations that frequently arise in simulations related to engineering and physical sciences, such as computational fluid dynamics and structural analysis. The *molecular dynamics simulation* is pivotal in studying the behavior of complex biological systems, such as protein folding and drug interactions, as well as material properties at the atomic level, which is essential for materials design and nanotechnology. Additionally, *merge sort* is foundational for processing large datasets generated in scientific experiments and simulations, allowing researchers to efficiently analyze and visualize data. As a consequence, this expanded evaluation will offer deeper insights into Go’s performance in high-performance computing scenarios.

Our experiments focus on comparing the performance of the original C code of the applications with our Go port implementation, which utilizes MPI4All bindings (see Figures 11 and 12). The performance analysis of the three parallel applications—Jacobi, Molecular Dynamics and Merge Sort—implemented in C and Go demonstrates significant speedups relative to their sequential execution, with varying degrees of improvement as the number of cores increases to 512. In Jacobi, C achieves a speedup of around $276\times$ when using 512 cores, whereas Go achieves a speedup of about $265\times$. At 512 cores, Go is approximately 13.3%

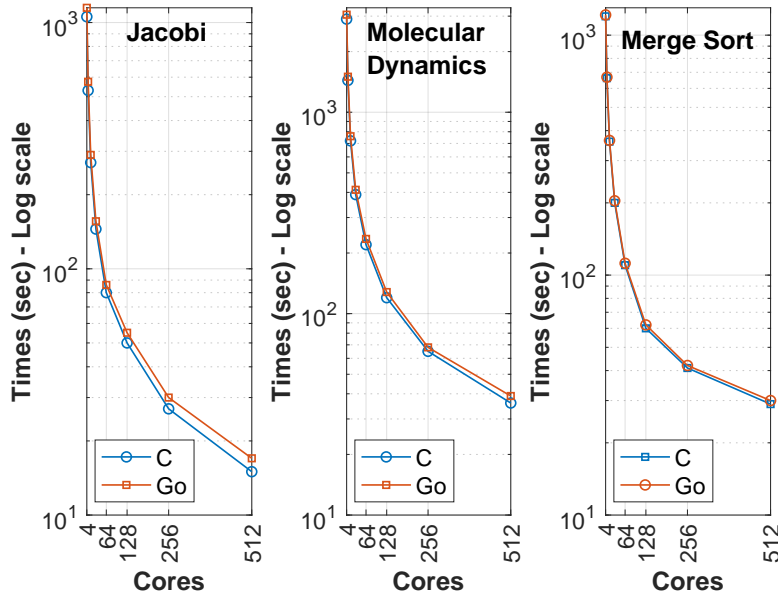


Fig. 11. Scalability of the original MPI C implementation of the scientific applications and the Go one that uses the bindings generated by MPI4All.

slower than C, with the C/Go ratio decreasing from 0.92 to 0.88, further reflecting Go's relatively slower performance. For Molecular Dynamics, C achieves a speedup of roughly $315\times$ with 512 cores, while Go achieves a speedup of about $305\times$. At 512 cores, Go is approximately 8.3% slower than C, with the C/Go ratio decreasing from 0.95 to 0.92, demonstrating a consistent gap in performance. For Merge Sort, C achieves a speedup of approximately $139\times$, while Go speedup is about $136\times$ using 512 cores. At this level, Go is approximately 3.4% slower than C, with the C/Go ratio decreasing from 0.99 at 1 core to 0.97 at 512 cores, indicating consistent performance across the board. Overall, both C and Go exhibit substantial speedups in parallel execution compared to sequential performance, with significant improvements observed across all applications as the number of cores increases. However, as expected, C consistently outperforms Go, with performance differences at 512 cores being up to 13.3% in favor of C in the Jacobi application. In any case, the seamless integration of C into Go has enabled MPI4All to produce a binding library with performance close to that of a C implementation.

Note that while other MPI Go bindings exist [1,3,4], none can fully implement the above applications due to limitations in asynchronous communication and Intel MPI support.

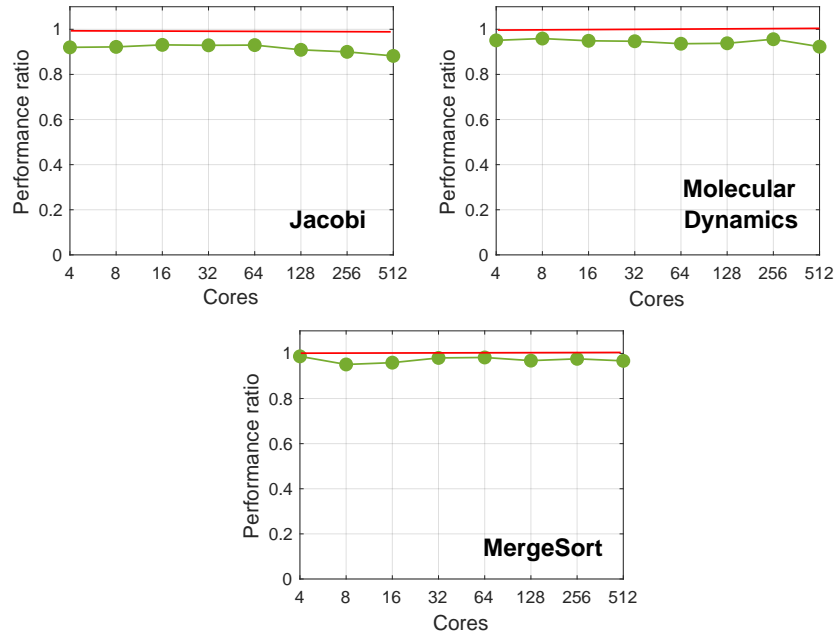


Fig. 12. Performance ratio between the original MPI C implementation of the scientific applications and the Go one that uses the bindings generated by MPI4All.

6 Conclusions

In this paper, we describe in detail MPI4All⁵, an innovative tool aimed at facilitating the creation of MPI bindings for any programming language. Adding support for new languages merely requires developing a script that maps MPI C macros, functions, and data types to the target language. Notably, unlike other approaches, MPI4All is not tied to a specific MPI implementation; it is compatible with all of them. Once the script for a programming language is available, generating bindings for the complete API of any MPI implementation (and version) takes seconds. This ensures completeness and avoids maintenance issues.

MPI4All includes scripts for generating MPI bindings for the Go and Java programming languages. We evaluated these bindings in terms of performance compared to other state-of-the-art solutions. The MPI4All Java bindings for OpenMPI and IntelMPI clearly outperform the official Java OpenMPI bindings when running several NAS benchmarks. While FastMPJ also provides consistent performance, it is important to note that it only supports MPI-2 routines. Regarding Go, the MPI4All bindings for Intel MPI achieve performance similar

⁵ It is publicly available at <https://github.com/citiususc/mpi4all>

to that of the native Intel C MPI library when running several relevant scientific applications.

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-C2, 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 11 dec 2024]
2. CESA (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 11 dec 2024]
4. MPI-binding package for Golang. <https://github.com/marcsthierfelder/mpl> [Online; accessed 11 dec 2024]
5. MPICH. <https://www.mpich.org>, [Online; accessed 11 dec 2024]
6. Open MPI. <https://www.open-mpi.org/>, [Online; accessed 11 dec 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. Panda, D.K., Subramoni, H., Chu, C.H., Bayatpour, M.: The MVAPICH project: Transforming research into high-performance MPI library for HPC community. *Journal of Computational Science* **52**, 101208 (2021).
9. MPICH. <https://www.boost.org>, [Online; accessed 30 sep 2024]
10. 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)
11. 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)
12. 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)
13. Dalcin, L., Fang, Y.L.L.: mpi4py: Status update after 12 years of development. *Computing in Science & Engineering* **23**(4), 47–54 (2021)
14. 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)
15. 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 11 dec 2024] (2023)

16. Sandia National Laboratories: Ember Communication Pattern Library (2018), <https://github.com/sstsimulator/ember> [Online; accessed 11 dec 2024]
17. 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)
18. Vega-Gisbert, O., Roman, J.E., Squyres, J.M.: Design and implementation of Java bindings in Open MPI. *Parallel Computing* **59**, 1–20 (2016)
19. Piñeiro, César, Pichel, Juan C.: A unified framework to improve the interoperability between HPC and Big Data languages and programming models, *Future Generation Computer Systems* **134**, 123–139 (2022).
20. Piñeiro, César, Vazquez, Alvaro, Pichel, Juan C.: MPI4All: Universal Binding Generation for MPI Parallel Programming. In: Franco, L., de Mulatier, Célia, Paszynski, M., Krzhizhanovskaya, V.V., Dongarra, J.J., Sloot, P.M.A. (eds.) *Computational Science – ICCS 2024*, pp. 196–208. Springer Nature Switzerland, Cham (2024), ISBN 978-3-031-63749-0.
21. S. Ghosh, C. Alsobrooks, M. Rüfenacht, A. Skjellum, P. V. Bangalore and A. Lumsdaine, "Towards Modern C++ Language Support for MPI," 2021 Workshop on Exascale MPI (ExaMPI), St. Louis, MO, USA, 2021, pp. 27-35, doi: 10.1109/ExaMPI54564.2021.00009.
22. H. Park, J. DeNio, J. Choi and H. Lee, "mpiPython: A Robust Python MPI Binding," 2020 3rd International Conference on Information and Computer Technologies (ICICT), San Jose, CA, USA, 2020, pp. 96-101, doi: 10.1109/ICICT50521.2020.00023.
23. Beazley, D.M.: SWIG: an easy to use tool for integrating scripting languages with C and C++. In: *Proceedings of the 4th Conference on USENIX Tcl/Tk Workshop, 1996 - Volume 4*, pp. 15. USENIX Association (1996)