# EMPI: Enhanced Message Passing Interface in Modern C++

Majid Salimi Beni, Luigi Crisci, and Biagio Cosenza
*Department of Computer Science, University of Salerno, Salerno, Italy*
Email: {msalimibeni, lcrisci, bcosenza}@unisa.it

***Abstract*—Message Passing Interface (MPI) is a well-known standard for programming distributed and HPC systems. While the community has been continuously improving MPI to address the requirements of next-generation architectures and applications, its interface has not substantially evolved. In fact, MPI only provides an interface to C and Fortran and does not support recent features of modern C++. Moreover, MPI programs are error-prone and subject to different syntactic and semantic errors.**

**This paper introduces EMPI, an Enhanced Message Passing Interface based on modern C++, which is directly mapped to the OpenMPI implementation and exploits modern C++ for safe and efficient distributed programming. EMPI proposes novel C++ RAII-based semantics and constant specialization to prevent error-prone code patterns such as parameter mismatch, and reduce the overhead of handling multiple objects and per-invocation time. Consequently, EMPI programs are safer: six out of nine well-known MPI error patterns do not occur while correctly using EMPI semantics. Experimental results on five microbenchmarks and two applications on a large-scale cluster using up to 1024 processes show that EMPI's performance is very similar to native MPI and considerably faster than the MPL C++ interface.**

***Index Terms*—Message Passing Interface (MPI), Modern C++, Programming Models, High Performance Computing**

## I. Introduction

The *Message Passing Interface* (MPI) is the *de facto* standard for programming distributed memory systems. Since the first definition of the standard in 1994 [1], development and standardization efforts have been continuous and led to several improvements in terms of topology [2], remote memory access [3], accelerator support [4], fault tolerance [5] and more.

Although the standard has evolved in terms of features, its interface — which provides a set of routines directly callable from C, C++, and Fortran — has not significantly changed. Its programming interface has been recognized to be error-prone [6]: MPI programs are subject to syntactic errors such as incorrect arguments, lost or dropped requests, data type and tag mismatching, wrong buffer usage, as well as semantic errors such as displacement and index out of range errors. Therefore, many researchers have investigated MPI code error detection based on static [7] and runtime [8, 9] analysis.

On the other hand, modern C++ has considerably expanded in recent years and has become the reference programming language for many high-productive, high-performance programming frameworks such as SYCL [10], KoKKoS [11], RAJA [12], and Celerity [13]. In particular, the combination of programming techniques such as RAII[1] and SFINAE[2], together with new language features such as Lambda functions, Constant expressions, CTAD[3] (C++17), Constraints and Concepts (C++20), have drastically enhanced the capability and productivity of C++ programs.

Recent work such as BoostMPI [14], MPL [15, 16], and MPP [17] provide a high-level C++ interface to the MPI library; however, they do not provide advanced features such as error mitigation techniques, constant static checks, or implicit waits for asynchronous calls. On top of that, existing C++ message passing interfaces are a direct mapping to the C-based MPI interface. This, potentially, is a limitation since it does not allow for across-the-stack improvements, e.g., by removing unneeded runtime checks for better performance. Therefore, to implement such optimizations, the interface must go beyond the C interface and interact with the lower layers of the MPI implementations.

Our work aims to use modern C++ to provide a high-level message passing interface. In contrast to the state-of-the-art, our interface is built on top of the OpenMPI [18] implementation. Thanks to the flexibility and programmability of C++ and interacting directly with the lower layers of OpenMPI, our work reduces the complexity of the code and prevents some of the most common programming errors. Moreover, it provides the potential to translate some of the checks related to constant parameters from runtime to compile-time.

This paper presents the Enhanced Message Passing Interface (EMPI)[4], a new message passing interface based on modern C++, and makes the following contributions:

- We implement EMPI, a modern C++ interface for message passing built on top of a customized version of OpenMPI, which allows for flexible and efficient mapping of EMPI semantics to communication primitives.
- Through EMPI, we propose two new RAII-based semantics, *program context* and *message group*, that enhance programmability, reduce the code's complexity, and prevent some error-prone code patterns such as lacking a matching wait for asynchronous calls. EMPI also introduces an efficient request-handling method for asynchronous calls in each *message group* that reduces the overheads of handling multiple request objects.

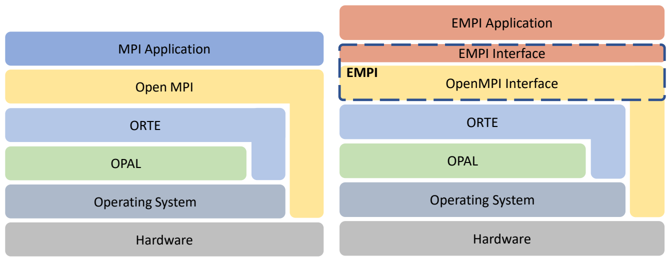
[1]Resource Acquisition Is Initialization
[2]Substitution Failure Is Not An Error
[3]Class Template Argument Deduction
[4]The ongoing project can be accessed at https://github.com/unisa-hpc/empi

- We present EMPI's *constant specialization* mechanism within *message groups*, which allows for (optionally) specifying EMPI function arguments as compile-time constants to prevent parameter mismatches and reduce the per-call invocation time.
- We experimentally evaluated EMPI in terms of potential check times saved by our methodology against equivalent MPI functions. Finally, we evaluated the performance of EMPI on five micro-benchmarks and two real-world applications against OpenMPI and MPL, the state-of-the-art C++ MPI binding.

The rest of the paper is organized as follows. Section II gives an overview of EMPI, then presents its interface semantics: *program context*, *message group*, and circular request pool. Section III focuses on profiling MPI run-time checks and explains the message group *constant specialization* mechanism, which allows the removal of the runtime checks. Section IV presents the experimental evaluation. Related work is presented in section V, and section VI concludes the paper. The Artifacts of the paper are described in section VIII.

## II. EMPI Overview

EMPI is a C++ library that aims to enhance the MPI programming model's interface while providing competitive performance. Its objective is to provide a more modern and straightforward interface to MPI while exploiting C++ language features to reduce programming errors and improve performance. It is developed using C++20, exploiting several modern C++ features.



(a) OpenMPI (b) EMPI

Fig. 1: OpenMPI and EMPI abstraction layer architectures.

Figure 1 compares the abstraction layer architecture of EMPI against OpenMPI (OMPI). As shown in this figure, the EMPI's lightweight interface is directly coupled with the OpenMPI interface and can interact with underneath layers. This allows EMPI, for instance, to skip some runtime checks, which are always performed when interacting with the MPI interface (unless specific flags are used). Therefore, the user application is developed based on EMPI's C++ semantics and compiled with the customized version of OpenMPI that calls *unchecked* functions. In this customized OMPI, each communication primitive has a corresponding unchecked version, which delivers the same functionality as the original function, with the difference that some of the runtime checks are skipped. For example, for `MPI_Bcast`, there is a corresponding `Unchecked_MPI_Bcast`.

Nevertheless, an EMPI application is still able to call standard MPI functions if *unchecked* semantics are not used. This means that the EMPI user application can still compile on all the standard MPI implementations, but it can benefit the optimizations mentioned in this paper only if compiled with our customized OMPI.

To clarify, figure 2 compares the call sequences of OMPI and EMPI. In OMPI, typically, when the user program calls an MPI primitive (e.g., `MPI_Send`), it performs several checks on all the passed parameters to this function, e.g., buffers, data size, data type, tag, communicator, etc. Then, it performs the main communication part (`send` in this example), which accomplishes the data transfer. In EMPI, however, when a communication primitive is called from the user's application, it is mapped to an *unchecked* function. This function performs only the checks that are not removable, i.e., cannot be inferred statically, and finally performs the communication. The following sections describe the details of EMPI's unchecked semantics.

### A. *Program Context*

RAII is a popular C++ programming discipline that provides safe management of system resources. It systematically encapsulates program resources in classes and performs all acquisitions and releases of such resources within constructors and destructors of the corresponding class. By making resource acquisition and release in this way, RAII not only provides safe resource management, which has been identified to be a challenge in MPI [19], but also minimizes the risk of leaking resources by cleaning them up at the end of their usage.

Considering the complications of the MPI programming model and the high possibility of carrying out simple programming errors, RAII can highly contribute to reducing the possibility of errors by avoiding error-prone code patterns while sometimes reducing the number of lines of code. In fact, EMPI proposes several RAII-based semantics that improve programmability and remove those error-prone patterns.

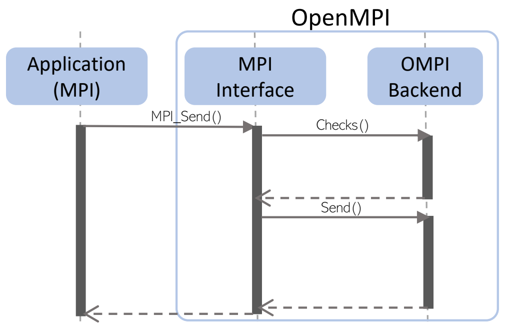
In MPI, a program starts with `MPI_Init()` and ends with `MPI_Finalize()`. The first use case of RAII in EMPI has been the characterization of *program context*, which defines the initialization and finalization of the EMPI program. In this way, `MPI_Init()` and `MPI_Finalize()` functions are replaced with one line of code to create the EMPI *program context*, and the user codes are written within this context. An example of such context creation is shown in Listing 1. By defining this semantic, we ensure not to forget finalizing the EMPI program.

```
using namespace empi;
Context ctx(&argc, &argv);
```

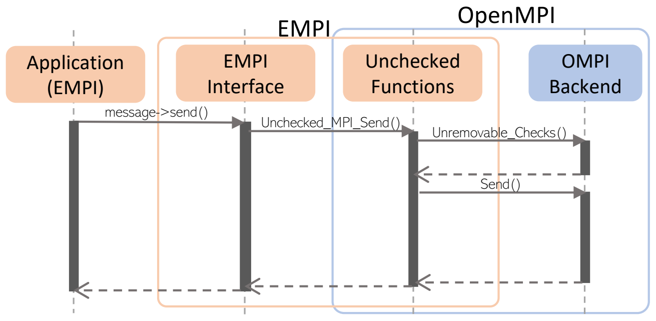
Listing 1: EMPI program context definition.

### B. *Message Group*

In MPI, it is very common to have program regions where all communication primitives share one or more parameters.



(a) MPI interface over OpenMPI call sequences.



(b) EMPI with unchecked semantic call sequences.

Fig. 2: OpenMPI and EMPI call sequences for a `send` operation.

For example, the communicator, the message size and the message type may be the same for a group of messages; similarly, it is common to have a pair of matching send-receive that share the same message tag. By exploiting *non-type template parameters*, SFINAE, and C++20 Concepts, we propose the *message group* semantic, which retains some communications that have some parameters in common. The idea behind implementing the *message group* semantic is to make EMPI able to specialize constant variables whose values can be set dynamically during the program's runtime. In fact, a *message group* is an object that holds a communicator and provides an interface to EMPI functionalities. Hence, each MPI function is overloaded in a way that a user can selectively specify parameters as compile-time constants or at runtime.

For the *message group* interface, EMPI provides a `run` function to be used for a list of EMPI calls that share one or more parameters. It takes a lambda function with a *message group handler* object as its input. This *message group handler* object is created by the *message group* and can be specialized with several compile-time parameters such as type, message size, and message tag. Within the lambda, the *message group handler* invokes the EMPI calls, and each call shares the compile-time parameters specified in the *message group handler* definition. Thanks to the *message group* semantic, the constant parameters can be specified (and checked) at compile time rather than runtime. On the programmability side, we pass fewer parameters to the EMPI communication functions, reducing the possibility of errors and parameter mismatches between coupled communication functions (e.g., send and receive). On the performance side, this technique improves performance by enabling us to shift some of the checks related to those constant parameters from runtime to compile-time. Accordingly, such checks are done only once within the *message group* constructor rather than per invocation.

Listing 2 illustrates *message group* creation within the previously-defined program context (in listing 1). As shown, the main communication happens in the `run` function, with three compile-time arguments (`datatype,tag,size`) shared between all the calls within the lambda. The user can invoke the EMPI primitives and collectives through the *message group* and the *message group handler*.

```
message_group = ctx.create_message_group(comm);
message_group->run(
  [&](MessageGroupHandler <datatype,tag,size> &mgh){
   // Do Work and Communication
});
```

Listing 2: Message group creation in EMPI, with three compile-time arguments.

## C. Implicit Wait for Asynchronous Calls

One common MPI programming mistake is forgetting to match waits with non-blocking calls. To address this problem, in EMPI, each asynchronous call is implicitly recorded by the message group in a request pool. When the message group is destroyed, a *wait* is called on all the dangling calls. This allows the programmer to nest C++ scopes and execute EMPI asynchronous calls in RAII fashion while ensuring all the requests are implicitly completed outside the current block.

For this purpose, we have defined `run_and_wait` function for the message groups, to be replaced with `run` function (in listing 2), in which an implicit `wait_all()` is called automatically at the end of lambda. In this way, we not only remove some lines of code related to waiting for each asynchronous call but also guarantee to wait for all the asynchronous calls at the end of the scope. Listing 3 presents an example of EMPI for `Ibcast` collective with `run_and_wait`. In this example, data type and data size are constant within the message group, and the `wait_all()` is omitted due to the usage of `run_and_wait`.

```
message_group->run_and_wait([&](MessageGroupHandler<char,
     notag, N> &mgh) {
  mgh.Ibcast(message, 0);
  // Do Some Work
}); //implicit wait here
```

Listing 3: EMPI Implicit wait example.

## D. Explicit Wait and Request Handling

As an alternative to the implicit wait, EMPI provides a way to explicitly define a wait within a message group. In this case, we use a normal `run` lambda, but instead of relying on the automatic *wait-on-exit* semantic, we explicitly call the `waitall` function in the message group. Using non-implicit wait in a message group, EMPI takes care of the MPI request objects that are usually associated with non-blocking communications. Hence, for each asynchronous communication, EMPI will automatically store the new request object in a *request pool*. Later within the same message group, when a new request object is demanded, the first available request in the pool is returned. Likewise, when a request has been fulfilled (e.g., a *wait* was called on it), it is marked as available and can be reused later in the rest of the current message group. In this way, and by reusing the requests, we minimize the overheads of creating and deleting multiple requests within each message group for asynchronous calls.

An example of explicit wait for asynchronous calls in EMPI is shown in listing 4. This example illustrates a one-dimensional stencil that takes advantage of the explicit wait at the end of each iteration of the `for` loop. The *request pool* is internally handled within `Isend` and `Irecv`. In this example, at the end of each iteration, (explicit) `waitall()` waits for all the communications to be finished. However, instead of releasing the request objects, they could be reused in subsequent iterations. Therefore, only four requests are allocated in the message group, reused in different iterations, and deleted at the end of that message group.

```
message_group->run(
  [&](MessageGroupHandler<char, Tag{0}, n> &mgh) {
  for (auto iter=0; iter<max_iter; iter++){
    mgh.Irecv(rbuff, prev);
    mgh.Irecv(rbuff, next);
    mgh.Isend(Sbuff, prev);
    mgh.Isend(Sbuff, next);
    mgh.waitall(); // Explicit wait
  }
});
```

Listing 4: Explicit wait example with a 1-dimensional stencil.

The usage of *request pool* within the implementation of `Isend` function is shown in listing 5. When there is a need for a request, calling `request_pool->get_req()` returns the first available request in the request pool to be re-used. Notice that in this listing, instead of calling `MPI_Isend`, `MPI_IUsend` that is an unchecked equivalence, is called, and some of the parameters' checks required by this function are specialized with the `requires` keyword.

```
template<typename K>
requires (is_valid_container<T,K> || is_valid_pointer<T,K>)
      && has_size_v<SIZE> && has_tag_v<TAG>
shared_ptr<async_event>& Isend(K&& data, int dest)
{
  // The request pool returns the first available request
  auto&& event = request_pool->get_req();
  // Call the unchecked OMPI Send
  event->res = MPI_IUsend(details::get_underlying_pointer(
     data), SIZE, details::mpi_type<T>::get_type(),dest, TAG
     .value, communicator, event->request.get());
  return event; //Reference to the request
}
```

Listing 5: EMPI implementation with concepts and request pool usage in the `Isend` function.

## *E. Semantics Overview*

Table I summarizes EMPI's principal classes and semantics. Worth noting that, in addition to the features mentioned, EMPI provides direct support for contiguous STL containers, such as `std::vector` and `std::array`. It also exposes a pointer interface that can be used for other data types.

| **Semantic** | **Scope** |
|---|---|
| Program Context | Initializes and manages the EMPI environment, and enables message groups creation |
| Message Group | Wraps a communicator, exposes EMPI functions, and creates Message Group Handlers |
| Message Group Handler | Represents compile-time constant parameters (i.e. tag,sizes,type), offers constant specialization, and invokes EMPI primitives |
| Implicit Wait Handler | Implicitly waits and handles the asynchronous EMPI calls by wrapping request objects |
| Request Handling | In a request pool, stores async events and provides collective functions over them, handles dangling requests |

TABLE I: Principal EMPI semantics.

As described in this section, EMPI minimizes the possibility of some programming errors by utilizing novel semantics. Table II summarizes a list of well-known MPI programming errors [20]: six out of nine error patterns are prevented by correctly utilizing EMPI semantics.

| **Error type** | **Error explanation** | **Avoided by EMPI** | **Avoidance Strategy** |
|---|---|---|---|
| Type mismatch | Buffer type and specified MPI type do not match | ✓ | Message Group |
| Incorrect buffer referencing | Buffer is not correctly referenced when passed to an MPI function | ✕ | - |
| Invalid argument type | i.e. non-integer type used where only integer types are allowed | ✓ | Message Group |
| Unmatched P2P call | Unmatched point-to-point call | ✕ | - |
| Unreachable call | Unreachable calls caused by deadlocks from blocking MPI calls | ✕ | - |
| Double non-blocking | Double request usage of non-blocking calls without intermediate wait | ✓ | Request Handling |
| Unmatched wait | Waiting for a request that was never used by a non-blocking call | ✓ | Request Handling |
| Missing wait | Non-blocking call without matching wait | ✓ | Implicit Wait |
| No Init/Finalize | Forgetting to put MPI_Init and MPI_Finalize | ✓ | Program Context |

TABLE II: The list of some common MPI programming errors and corresponding EMPI's avoidance strategies.

## III. Message Group Constant Specialization

In MPI applications, a group of communications very often use the same parameters, e.g., data type, tag, size, or communicator. To some extent, these parameters can be treated as constant values for those communications within each *message group handler*. EMPI constant specialization represents constant parameters whose values can be set dynamically during the execution of the program. In other words, the values of these constants are fixed within the scope of a message group.

Exploiting EMPI's constant specialization in message groups prevents passing multiple parameters to EMPI's communication primitives and therefore reduces the possibility of programming errors. This semantic also enables us to benefit from diminishing the checking overheads within each communication function.

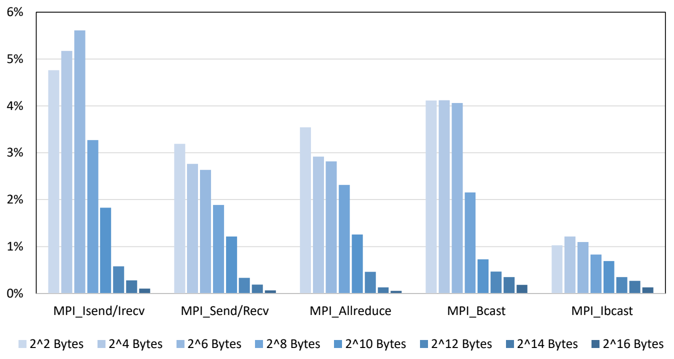
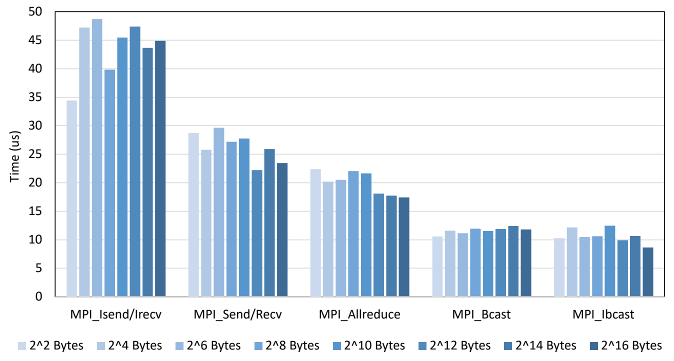


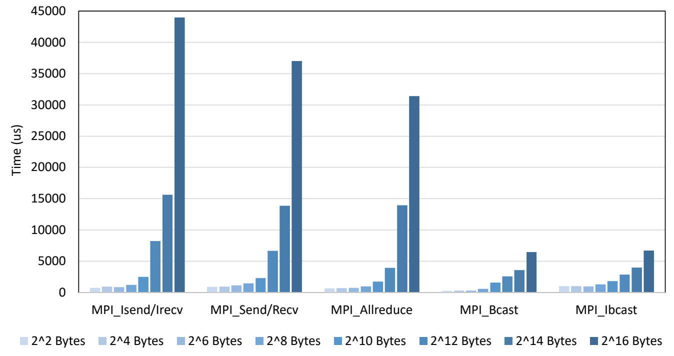
Fig. 3: Percentages of checks to the overall time taken by different OMPI function invocations on 2 processes.

Considering the OMPI's interface communication primitives, each function consists of several checks plus the main operation, which performs the communication (As demonstrated in figure 2). These checks are primarily to check if the data type, message size, and communicator are defined for the required buffers and if they are accessible. Also, they check the validity of the value of parameters like tag, data type, buffers, communicators, etc., passed to each call. Although most of the invocation time of each MPI communication call is spent during the data transfer (main operation), a small fraction of each MPI call is spent while doing all the above-mentioned checks. Therefore, utilizing constant specialization, we skip some of the checks in each message group for the constant parameters. In fact, EMPI performs those only once in the *message group*'s constructor. For this, the *unchecked functions* are called in EMPI as the backend communication functions within the *message group*. These unchecked functions are implemented in EMPI's OpenMPI and carry out what the normal function does, except they skip some time-consuming checks handled by the *message group*.

Before making any effort to eliminate some checks using the constant specialization, we need to investigate if the amount of time spent during these checks is considerable. For this, we designed a profiling system to measure the accurate times of each phase of the functions on a single node. Since the



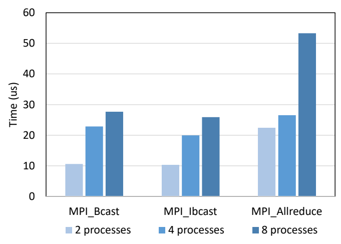
(a) Checks times only



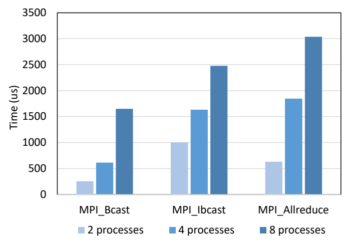
(b) Main communication time only

Fig. 4: Taken times by the two phases of OMPI communication functions with different data sizes on two processes.

measured times are in micro/nano seconds, we report the summation times of 1000 iterations of each communication function. Figure 3 shows the percentage of the checks' times to the overall time taken by some OMPI calls while transferring different data sizes on two processes. As indicated in this figure, a small but still considerable fraction of each OMPI call's time is spent in runtime checks. It reaches around 6% in Isend/Irecv while operating on small messages. However, by increasing the data size, the communication time becomes dominant, and the checks times fraction becomes smaller.



(a) Checks times only



(b) Main communication time only

Fig. 5: Taken times by the two phases of OMPI communication functions with $2^2$ Bytes of data and different numbers of MPI processes.

As a deeper look inside the functions, figure 4 distinguishes between checks' times and main operation time. Figure 4a

```
MPI_Init(&argc, &argv);
MPI_Comm_size(comm, &procs);
MPI_Comm_rank(comm, &myid);
if (myid == 0) {
  MPI_Send(arr, n, MPI_CHAR, 1, 0, comm);
  MPI_Recv(arr, n, MPI_CHAR, 1, tag, comm, &status);
} else { // Node rank 1
  MPI_Recv(myarr, n, MPI_CHAR, 0, tag, comm, &status);
  MPI_Send(myarr, n, MPI_CHAR, 0, 1, comm);
}
MPI_Barrier(comm);
MPI_Finalize();
```

Listing 6: MPI ping-pong example.

```
Context ctx(&argc, &argv);
ctx.create_message_group(comm)->run(
  [&](MessageGroupHandler<char,Tag{0},n> &mgh) {
  if(mgh.rank() == 0){
    mgh.send(message,1);
    mgh.recv(message,1,status);
  }else{
    mgh.recv(message,0,status);
    mgh.send(message,0);
  }
  mgh.barrier();
});
```

Listing 7: EMPI ping-pong example.

```
MPI_Init(&argc, &argv);
MPI_Comm_size(comm, &procs);
MPI_Comm_rank(comm, &myid);
int tag1 = 0;
int tag2 = 1;
MPI_Irecv(buf[0], 1, MPI_INT, prev, tag1, comm, &reqs[2]);
MPI_Irecv(buf[1], 1, MPI_INT, next, tag2, comm, &reqs[3]);
MPI_Isend(&rank, 1, MPI_INT, prev, tag2, comm, &reqs[0]);
MPI_Isend(&rank, 1, MPI_INT, next, tag1, comm, &reqs[1]);
 { /* do some work */ }
MPI_Waitall(4, reqs, stats);
MPI_Finalize();
```

Listing 8: MPI 1-dimensional communication asynchronous example.

```
Context ctx(&argc, &argv);
ctx.create_message_group(comm)->run_and_wait(
  [&](MessageGroupHandler<double,notag,size> &mgh){
  Tag tag1{0};
  Tag tag2{1};
  mgh.Irecv(&buf[0], message_group->prev(), tag1);
  mgh.Irecv(&buf[1], message_group->next(), tag2);
  mgh.Isend(&rank, message_group->prev(), tag2);
  mgh.Isend(&rank, message_group->next(), tag1);
 { /* do some work */ }
}); // Waitall is implicit here
```

Listing 9: EMPI 1-dimensional communication asynchronous example.

shows the checks times-only for different OMPI calls, while figure 4b presents the times taken only by the main operation of the functions. Although the checking times are almost constant for all the functions and do not scale with the message size, the main operation (communication) times increase with the increment of message size for all the functions. That is why the percentage of checks' times to the overall time decreases with the increment of message size in figure 3. Note that there is a slight variation in the check and communication times with different message sizes due to the performance variability issue [21].

To figure out how much time is spent performing the checks when increasing the number of processes, we performed a scalability test with constant input data size ($2^2$ bytes) for each collective. As shown in figure 5, both checks' times and communication times increase with increasing the number of processes for the three collectives. However, checks times do not grow the same as communication times since they are much smaller (smaller than 25 us for all of them), and even if they increase, their proportion would still be much smaller than the communication times.

Overall, this is evident that an important percentage of each OMPI call's time is spent while performing runtime checks. Moreover, the check times do not scale with changing the message size of the corresponding call, and this time is invariant for different message sizes. Nevertheless, checks times increase with increasing the number of processes primarily because some checks have to be done for each participant process in that communication which slightly contributes to the increment of overall checks times. But regarding the high growth of the communication time at the same time, when having more processes, checking times' share of the overall time gets smaller. Therefore, by specializing the constant parameters within message groups, the checks related to those specialized parameters are omitted for all the calls inside that *message group*, and a small performance gain is achieved. Regardless, this gain would be higher while operating on smaller message sizes and fewer processes.

## IV. Experimental Evaluation

In this section, we experimentally evaluate EMPI in terms of interface and performance. In particular, we compared the EMPI interface against the MPI classical interface along with EMPI's performance against MPI (over OpenMPI) and MPL (the state-of-the-art) on five micro-benchmarks and two mini-applications on a large-scale cluster.

### A. Experimental Setup

The experiments are performed on Marconi100 [22] at the CINECA supercomputing center [23]. Marconi100, with 55 racks, is an IBM Power9-based system with 980 nodes (plus 8 login nodes). Each node is equipped with two IBM AC922 CPUs with 16 cores (32 cores/node and 31,616 cores overall) at 2.6 (3.1 turbo) GHz, four NVIDIA Volta V100 GPUs with 16GB and Nvlink 2.0, and 256 GB of memory/node (252,928 GB overall). The internal network is Mellanox InfiniBand EDR Dragonfly+ 100Gb/s. The operating system is Red Hat Enterprise 7.6, and all the codes are compiled with gnu 10.3.0 and OpenMPI 4.1 using the `-O3` flag.

## B. *Interface Evaluation*

We compare EMPI and MPI interfaces with selected code examples that illustrate the EMPI features and their advantages against the MPI interface. Listings 6 and 7 compare a ping-pong example in MPI and EMPI using two processes. In the EMPI example, the program has one *program context* that consists of a *message group*. All the communications within this *message group* share the same communicator `comm`, and all the functionality happens within the *message group*'s `run` function. Moreover, `MessageGroupHandler` object (`mgh`) handles the communications and specializes the constant parameters (data size and type, in this example) for the *message group*. In this example, in the `send` and `recv`, the programmer only needs to specify the data and the source or destination. In this way, we not only reduce the possibility of parameter invalid or mismatch between paired functions but also reduce the code complexity.

In listings 8 and 9, we illustrate the implementations of a 1-dimensional asynchronous message exchange with both MPI and EMPI. Likewise, in this EMPI example, message type and size are specialized as constants inside the message group handler. Also, by using the `run_and_wait` function, there is no need to put any explicit wait at the end of the message group. The functions `prev()` and `next()` in this code snippet, respectively, return the previous and next MPI rank in the current *message group*.

## C. *Performance Evaluation*

This section focuses on the performance of EMPI and evaluates it against pure MPI (OpenMPI) as the baseline, and MPL, as the state-of-the-art in C++ message passing interface. For this purpose, we primarily take a set of micro-benchmarks from OSU [24], then compare their performance on different message sizes. Afterward, we keep the message size constant and evaluate it with varying numbers of processes. In addition to the micro-benchmarks, we evaluate two applications, LULESH[5] [25] and Vibrating String [26], and assess them on hundreds of MPI processes. Each experiment is repeated 1000 times for the micro-benchmarks and 100 times for the applications, and the average time is reported. Moreover, to consider the variability in the results mostly originating from potential network noises, we show the error bars for the inter-node experiments.

*1) Micro-benchmarks:* For the micro-benchmarks comparison, we take five micro-benchmarks, including blocking and non-blocking peer-to-peer, Bcast, Ibcast, and Allreduce, and compare their performance on two processes with different message sizes, as shown in figure 6. For all the micro-benchmarks, except for Allreduce, EMPI performs very similarly to OMPI and always performs better than MPL. Specifically, it performs 15% faster than Send/Recv MPL for $2^{16}$ bytes.

In this figure, for smaller messages, the EMPI's performance is 34%, 7%, 27%, 56%, and 25% higher than MPL

[5]Livermore Unstructured Lagrange Explicit Shock Hydrodynamics

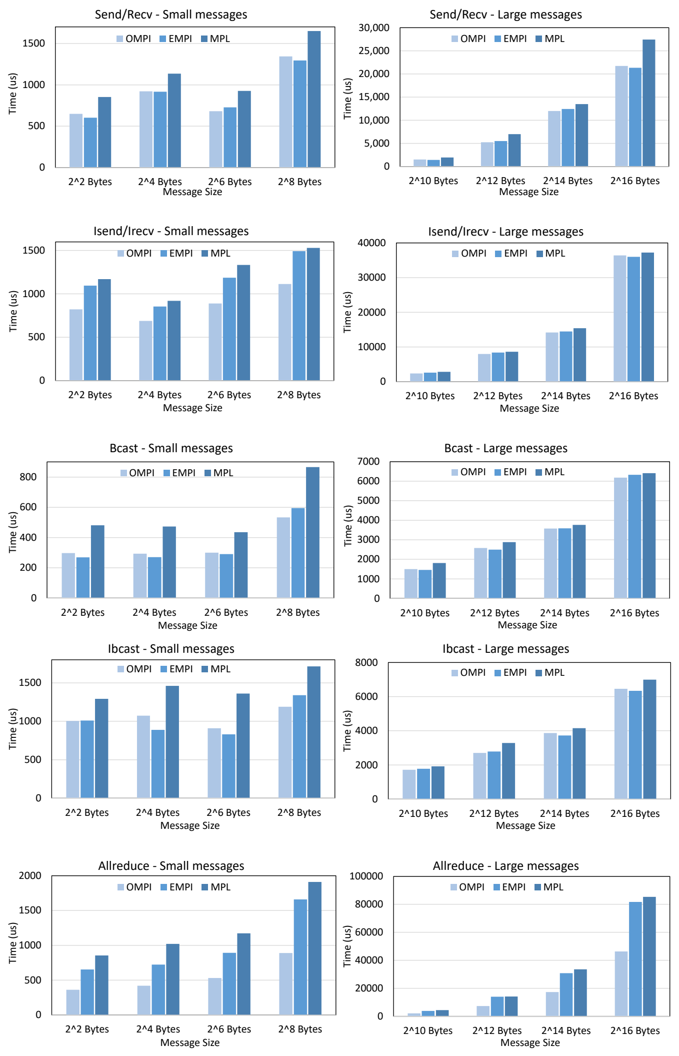
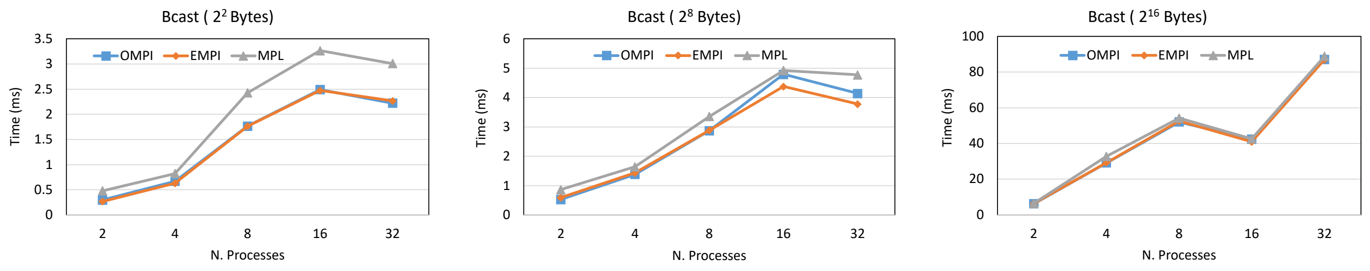


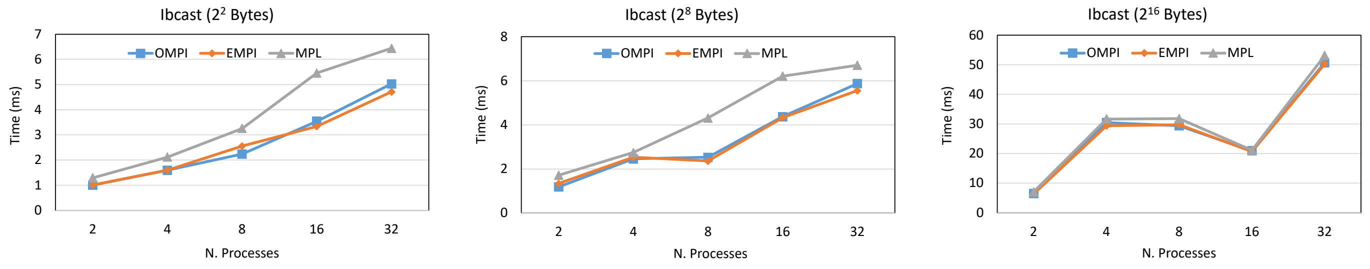
Fig. 6: OMPI, EMPI and MPL on 2 MPI processes on a node.

for Send/Recv, Isend/Irecv, Allreduce, Bcast, Ibcast on $2^2$ bytes. This performance improvement is because of the gain from calling *unchecked* functions underneath, which shows to be more fruitful when dealing with small messages. In Allreduce, EMPI's slowness of OMPI is mainly because of this collective's usage of two buffers, which makes the efforts of function invocations for passing parameters, and buffer handling larger than the other collectives.

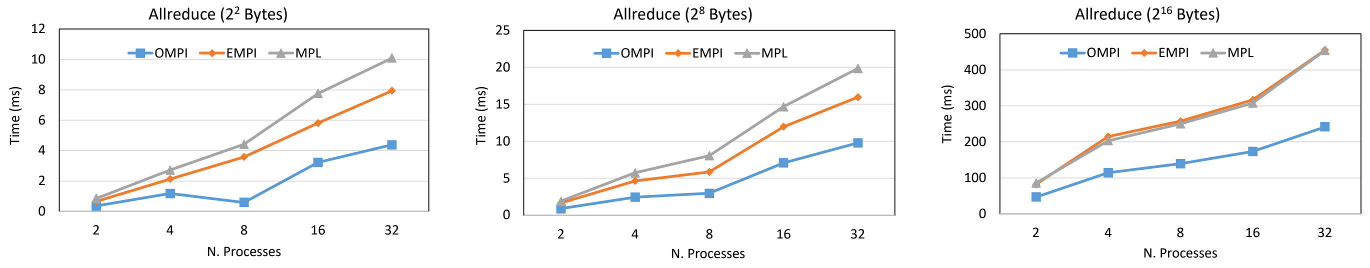
In figure 7, for the collective micro-benchmarks, we evaluate the performances of EMPI, MPL, and OMPI on different numbers of processes within a node. This evaluation is done with three different sizes: $2^2$, $2^8$, and $2^{16}$ bytes. For Bcast and Ibcast, with $2^2$ and $2^8$ bytes, EMPI performs almost the same as OMPI, while MPL performs slower, and there is a gap between their performances. Nevertheless, in $2^{16}$ bytes of Bcast and Ibcast, all three interfaces show similar latencies for different numbers of processes. Because with increasing the scalability, either the advantages of removing checks or libraries overheads get less effective and the communication



(a) Broadcast



(b) Ibcast



(c) Allreduce

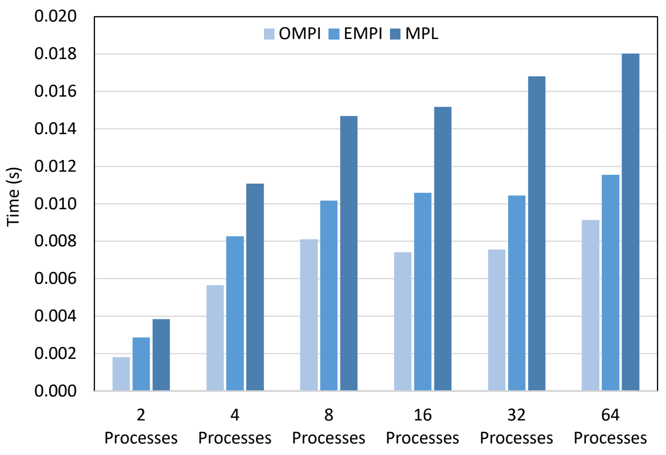
Fig. 7: OMPI, EMPI and MPL with different number of processes on a node and with three different message sizes.

time dominates all the optimizations and overheads of the library.

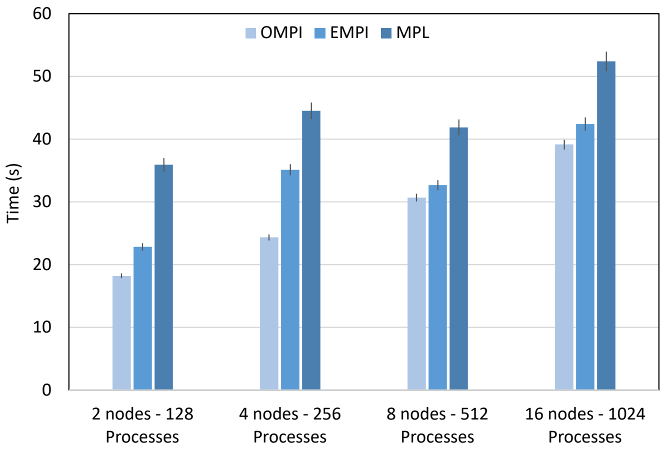
In Allreduce, however, EMPI performs slower than OMPI for the three sizes, although it is still faster than MPL for $2^2$ and $2^8$ bytes. In $2^{16}$ bytes, EMPI Allreduce performs the same as MPL and it becomes slower than OMPI. This is mainly because of C++ function invocation overheads considering that all the parameters are passed by reference and cannot be instantiated at compile time. Also, Allreduce handles two buffers which makes the invocation overheads more than other collectives. It is also an un-rooted algorithm in which the same checks are performed for all ranks on both intra and inter-communicators. While increasing the scale, each process requires some time to create, check, and manage its buffers, and using C++ semantic to handle data imposes some overhead. Similarly, the performance gain from unchecked functions gets less fruitful when increasing the number of processes and message size.

*2) Vibrating String:* The Vibrating String is a mini-application that solves the time-dependent one-dimensional wave equation to obtain the displacement of a vibrating string via a finite difference discretization and explicit time stepping. The MPI implementation of this mini-application mainly performs one-dimensional non-blocking communications. The strong scaling results are shown in figure 8 with 1001 total number of grid points on both single-node (8a) and multi-node (8b) environments with up to 1024 processes (1-16 nodes).

As demonstrated in figure 8, for different numbers of processes, EMPI performs faster than MPL, and its execution time does not drastically change from 4 to 64 processes. In contrast, in MPL, when increasing the number of processes, the execution time slightly increases (EMPI is 29% faster than MPL for 4 processes and is 44% faster on 64 processes), which is because of the lower internal overheads and using the unchecked communications by EMPI. Additionally, on more than one node, although the execution times of all three interfaces get closer, EMPI is still more efficient than MPL and is 21% faster than MPL with 1024 processes. The reason behind the slowness of EMPI compared to OMPI is the overheads of function invocations within the EMPI layer that overcomes the benefits of using it. However, EMPI is still showing a higher performance than MPL on different number of processes.



(a) Single node



(b) Multi-node

Fig. 8: OMPI, EMPI and MPL for the Vibrating String mini-application.

*3) LULESH:* LULESH is a proxy application from the shock hydrodynamics area. Using MPI two-sided non-blocking APIs, within each iteration, all the processes communicate with all their neighbors in a three-dimensional domain and exchange their ghost fields (boundary data).

Figure 9 presents the weak-scaling performance comparison of LULESH implemented with OMPI, MPL, and EMPI on 8–1000 processes (1-16 nodes) with the default problem size of 27,000 and 10 internal iterations. In this figure, both MPL and EMPI demonstrate a performance close to the OMPI. However, EMPI is performing even faster than OMPI when increasing the number of processes; with 1000 processes, it is 8% and 13% faster than OMPI and MPL, consecutively. This slight performance gain is primarily because of the application's communications behavior. In LULESH, iteratively, each process does a non-blocking data exchange with all of its neighbors in a 3-dimensional space. All the exchanged messages are of the same size and type, and the communicator does not change for them, as well. Hence, this application is the best fit for EMPI, and its characteristics allow the application to benefit both the request pool and constant specialization. Therefore, in this example, when there are higher numbers of processes — which means more asynchronous calls — the benefits of using the EMPI interface dominate its overheads, and EMPI shows higher performance.

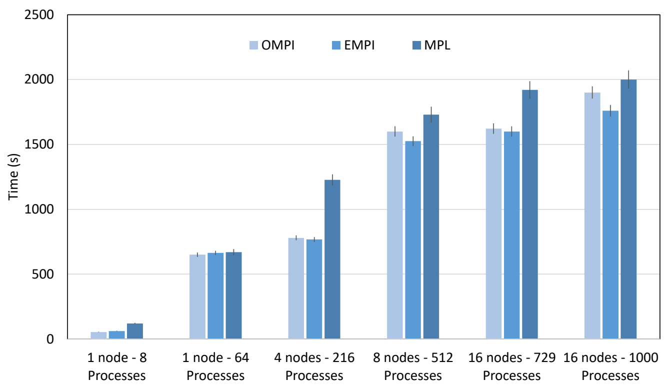


Fig. 9: OMPI, EMPI and MPL for the LULESH proxy application.

## V. Related Work

In recent years, the C++ programming language has become a popular choice for developing high performance applications. With the emergence of heterogeneous computing systems and the evolution of the C++ standard to what is called modern C++, some novel C++-based heterogeneous programming models such as Kokkos [11], and SYCL [10] have been designed to reach portability across multiple target devices. These models have also been successful for real-world scientific applications [27]. In line with these efforts, there have also been efforts to exploit modern C++ features in MPI.

The efforts to bring a C++ binding to MPI started since the development of the first MPI versions [28, 29, 28]. Skjellum et al. [29] were among the first who tried to introduce a C++ binding for MPI by proposing MPI++. Their goal was to make their implementation syntactically and semantically consistent with the C interface. Later, Object-oriented MPI (OOMPI) [28] was introduced, which was an attempt to provide a C++ class library for MPI and bring object-oriented into MPI. By bringing together OOMPI's object-oriented design and its ease of use and MPI++'s new features of C++, a merged system was proposed [30]. This merged system resulted in a standard binding within the MPI-2 standard. Kambadur et al. [14] proposed BoostMPI in which they investigated several solutions to modernize the C++ interface to MPI and provide a more natural syntax. Hence, they tried to provide seamless support for C++ standard library constructs and user-defined types. They also show latency improvement by using function specialization and serializing the data types. In line with these efforts, MPP [17] was introduced, which was an advanced C++ interface to MPI. In this work, they brought together some ideas of OOMPI and BoostMPI, resulting in a lightweight header-only C++ interface. They focused on point-to-point communications and integration of user data types, relying on native MPI Datatypes for better performance.

Recent research investigated the use of C++20 and MPI 3. Ruefenacht et al. [31] discuss how the C-based MPI interface is holding MPI back as a whole. They suggest a performance-portable modern C++ language interface for MPI to maintain productivity, performance, and interoperability. Ghosh et al. [16] proposed a prototypical interface derived from MPL [15], an MPI-based open-source C++17 library. For this purpose, they removed the MPI-unrelated features of MPL to make it more lightweight, and considered handling point-to-point/collective communication, derived data types, communication completion, and communicator interfaces.

Table III compares EMPI with state-of-the-art. With respect to state-of-the-art, MPP is the only work that provides both performance features and error mitigation strategies. MPP's error mitigation is limited to inferring some information required by MPI routines at compile-time. Also, MPP supports peer-to-peer communications and does not investigate collective communications. In another recent work, MPL, there is no error mitigation strategy. They only compare their work's performance against MPI and provide some insights and future directions of such MPI and C++ bindings. In contrast to these two works, EMPI investigates several error reduction techniques and is capable of removing six out of nine well-known MPI error patterns [20]. It also evaluates the performance against both MPI and MPL and shows considerably higher performance than MPL. In comparison to related work, EMPI relies on OpenMPI to deliver more efficiency and performance.

| Method | C++ Programming Style | Perf. Evaluation | Error reduction |
|---|---|---|---|
| Skjellum et al. [29] | Basic object-oriented MPI library | × | × |
| OOMPI [28] | Object-oriented stream-based MPI | × | × |
| BoostMPI [14] | C++03 object-oriented MPI library with data serialization mechanism | ✓ | × |
| MPP [17] | Lightweight C++11 header-only, object-oriented MPI library | ✓ | ✓ |
| Ruefenacht et al. [31] | In-depth analysis of C++ MPI capabilities | × | × |
| Ghosh et al. (MPL) [16] | C++17 header-only MPI library with complex data layout support | ✓ | × |
| **EMPI** | **C++20, low-overhead OMPI-mapping, message passing interface with message grouping** | ✓ | ✓ |

TABLE III: Comparison with the state of the art.

## VI. Discussion and Conclusion

In this paper, we proposed EMPI, which is a modern C++-based message passing interface implemented on top of OpenMPI that attempts to enhance MPI's interface. The proposed work improves programmability thanks to C++ features such as RAII and SFINAE and makes the application code less error-prone: six out of nine well-known MPI error patterns are totally avoided by correctly exploiting EMPI. Moreover, by proposing two new semantics, program context and message group, EMPI contributes to decreasing the code's complexity and reduces the number of parameters passed to each communication call by compile-time constant specialization.

Relying on a customized implementation of OpenMPI that skips some unnecessary checks within each MPI communication call, EMPI offers a comparative performance to MPI (OpenMPI) and a higher performance than MPL, the state-of-the-art. In fact, it showed that it could achieve a near-MPI performance for different micro-benchmarks and applications with different message sizes and hundreds of processes on a large-scale compute cluster. It, however, should be noted that currently, EMPI does not provide support for all new MPI features, and it only handles some basic features, such as peer-to-peer and collective communications and basic data types. As future work, we are working to integrate more modern C++ and MPI features in EMPI while investigating to reduce possible overheads to make it able to be used for more use-cases and applications.

All in all, modern C++ has shown to be an excellent choice for implementing a high-level interface for message passing libraries. It not only provides a better programming experience to the users by reducing the possibility of errors and reducing complexity but also can improve overall performance. Worth noting that adopting a C++ interface is straightforward. Mainly, due to the growing popularity of C++ as a language for scientific and high-performance computing, it seems the right time for the MPI community to develop a standard C++ interface for MPI to meet the evolving needs of the HPC community. Regarding the results achieved in this paper, we show that it is very hard to get higher performance than C-based MPI by only enhancing the interface. Therefore, the MPI backend layers must undergo some modifications (e.g. handling some checks statically, enabled by C++) to improve the efficiency of C++ binding.

## VII. Acknowledgements

This research has been partially funded by the European High-Performance Computing Joint Undertaking (JU) under grant agreement No. 956137 (LIGATE project).

## References

[1] Lyndon Clarke, Ian Glendinning, and Rolf Hempel. "The MPI message passing interface standard". In: (1994), pp. 213–218.

[2] Stephane Bouhrour, Thibaut Pepin, and Julien Jaeger. "Towards leveraging collective performance with the support of MPI 4.0 features in MPC". In: *Parallel Computing* 109 (2022), p. 10286.

[3] Torsten Hoefler, James Dinan, Rajeev Thakur, Brian Barrett, Pavan Balaji, William Gropp, and Keith Underwood. "Remote memory access programming in MPI-3". In: *ACM Transactions on Parallel Computing (TOPC)* 2.2 (2015), pp. 1–26.

[4] K Suresh, K Khorassani, C Chen, B Ramesh, M Abduljabbar, A Shafi, and DK Panda. "Network assisted non-contiguous transfers for GPU-aware MPI libraries". In: *2022 IEEE Symposium on High-Performance Interconnects (HOTI)*. 2022, pp. 13–20.

[5] Nuria Losada, Patricia González, María J Martín, George Bosilca, Aurélien Bouteiller, and Keita Teranishi. "Fault tolerance of MPI applications in exascale systems: The ULFM solution". In: *Future Generation Computer Systems* 106 (2020), pp. 467–481.

[6] Tobias Hilbrich, Joachim Protze, Martin Schulz, Bronis R de Supinski, and Matthias S Müller. "MPI runtime error detection with MUST: advances in deadlock detection". In: *Scientific Programming* 21.3-4 (2013), pp. 109–121.

[7] Jan-Patrick Lehr, Tim Jammer, and Christian Bischof. "MPI-corrbench: Towards an MPI correctness benchmark suite". In: *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*. 2021, pp. 69–80.

[8] Tassadit Célia Aitkaci, Marc Sergent, Emmanuelle Saillard, Denis Barthou, and Guillaume Papauré. "Dynamic Data Race Detection for MPI-RMA Programs". In: *EuroMPI 2021-European MPI Users's Group Meeting*. 2021.

[9] Rana A Alnemari, Mai A Fadel, and Fathy Eassa. "Integrating static and dynamic analysis techniques for detecting dynamic errors in MPI programs". In: *International Journal of Computer Science and Mobile Computing* 7.4 (2018), pp. 141–147.

[10] Ben Ashbaugh, Alexey Bader, James Brodman, Jeff Hammond, Michael Kinsner, John Pennycook, Roland Schulz, and Jason Sewall. "Data parallel c++ enhancing SYCL through extensions for productivity and performance". In: *Proceedings of the International Workshop on OpenCL*. 2020, pp. 1–2.

[11] H Carter Edwards, Christian R Trott, and Daniel Sunderland. "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns". In: *Journal of parallel and distributed computing* 74.12 (2014), pp. 3202–3216.

[12] David A Beckingsale, Jason Burmark, Rich Hornung, Holger Jones, William Killian, Adam J Kunen, Olga Pearce, Peter Robinson, Brian S Ryujin, and Thomas RW Scogland. "RAJA: Portable performance for large-scale scientific applications". In: *2019 IEEE/ACM international workshop on performance, portability and productivity in hpc (p3hpc)*. IEEE. 2019, pp. 71–81.

[13] Peter Thoman, Philip Salzmann, Biagio Cosenza, and Thomas Fahringer. "Celerity: High-level c++ for accelerator clusters". In: *European Conference on Parallel Processing*. Springer. 2019, pp. 291–303.

[14] Prabhanjan Kambadur, Douglas Gregor, Andrew Lumsdaine, and Amey Dharurkar. "Modernizing the C++ interface to MPI". In: *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer. 2006, pp. 266–274.

[15] *GitHub - rabauke/mpl: A C++17 message passing library based on MPI*. https://github.com/rabauke/mpl. (Visited on 12/06/2022).

[16] Sayan Ghosh, Clara Alsobrooks, Martin Rüfenacht, Anthony Skjellum, Purushotham V Bangalore, and Andrew Lumsdaine. "Towards modern C++ language support for MPI". In: *2021 Workshop on Exascale MPI (ExaMPI)*. IEEE. 2021, pp. 27–35.

[17] Simone Pellegrini, Radu Prodan, and Thomas Fahringer. "A lightweight C++ interface to MPI". In: *2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing*. IEEE. 2012, pp. 3–10.

[18] Richard L Graham, Timothy S Woodall, and Jeffrey M Squyres. "Open MPI: A flexible high performance MPI". In: *International Conference on Parallel Processing and Applied Mathematics*. Springer. 2005, pp. 228–239.

[19] Dominik Huber, Maximilian Streubel, Isaías Comprés, Martin Schulz, Martin Schreiber, and Howard Pritchard. "Towards dynamic resource management with MPI sessions and PMIx". In: *EuroMPI/USA'22: 29th European MPI Users' Group Meeting*. 2022, pp. 57–67.

[20] Alexander Droste, Michael Kuhn, and Thomas Ludwig. "MPI-checker: static analysis for MPI". In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. 2015, pp. 1–10.

[21] Majid Salimi Beni and Biagio Cosenza. "An Analysis of Performance Variability on Dragonfly+ topology". In: *2022 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE. 2022, pp. 500–501.

[22] *Marconi100, The new accelerated system*. URL: https://www.hpc.cineca.it/hardware/marconi100 (visited on 12/06/2022).

[23] *Top500, MARCONI-100*. https://www.top500.org/system/179845/. (Visited on 12/06/2022).

[24] *OSU Micro-Benchmarks 5.8*. 2021. URL: https://mvapich.cse.ohio-state.edu/benchmarks/ (visited on 12/06/2022).

[25] Ian Karlin, Abhinav Bhatele, Jeff Keasler, Bradford L Chamberlain, Jonathan Cohen, Zachary DeVito, Riyaz Haque, Dan Laney, Edward Luke, Felix Wang, et al. "Exploring traditional and emerging parallel programming models using a proxy application". In: *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE. 2013, pp. 919–932.

[26] Gerald F Wheeler and William P Crummett. "The vibrating string controversy". In: *American Journal of Physics* 55.1 (1987), pp. 33–37.

[27] Luigi Crisci, Majid Salimi Beni, Biagio Cosenza, Nicolò Scipione, Davide Gadioli, Emanuele Vitali, Gianluca Palermo, and Andrea Beccari. "Towards a portable drug discovery pipeline with SYCL 2020". In: *International Workshop on OpenCL*. 2022, pp. 1–2.

[28] Brian C McCandless, Jeffrey M Squyres, and Andrew Lumsdaine. "Object oriented MPI (OOMPI): a class library for the message passing interface". In: *Proceedings. Second MPI Developer's Conference*. IEEE. 1996, pp. 87–94.

[29] Anthony Skjellum, Ziyang Lu, Purushotham V Bangalore, and Nathan Doss. "Explicit parallel programming in C++ based on the Message-Passing Interface (MPI)". In: *Parallel Programming Using C+* (1995), pp. 767–776.

[30] Anthony Skjellum, Diane G Wooley, Ziyang Lu, Michael Wolf, Purushotham V Bangalore, Andrew Lumsdaine, Jeffrey M Squyres, and Brian McCandless. "Object-oriented analysis and design of the Message Passing Interface". In: *Concurrency and Computation: Practice and Experience* 13.4 (2001), pp. 245–292.

[31] Martin Ruefenacht, Derek Schafer, Anthony Skjellum, and Purushotham V Bangalore. "MPIs language bindings are holding MPI back". In: *arXiv preprint arXiv:2107.10566* (2021).

## VIII. Artifact Appendix

EMPI is an Enhanced Message Passing Interface based on modern C++, which is directly mapped to a customized version of OpenMPI (OMPI) implementation and exploits modern C++ for safe and efficient distributed programming. The artifact consists of two parts: first, the customized OMPI, which adds a new version of point-to-point and collective communications without runtime checks. Second is the EMPI, which is the programming interface built on top of the customized OMPI. Both the customized OMPI and EMPI source codes are open-source and publicly available at https://doi.org/10.5281/zenodo.7727977 In this section, we present a brief description of the artifact. More details are presented in the README file of the repository.

**Important Notice:** The codes in the paper are tested on a Supercomputer (Marconi100 @CINECA supercomputing center), and its specifications are described in the paper; however, due to the unique specifications of this system (job scheduler, hardware, etc.) and considering the fact that not everyone might have access to such a system, in this artifact description, we provide all the details on how to build and run the code on a local machine.

### *A. Artifact Checklist and Requirements*

**Hardware:** In addition to the Marconi100 (equipped with IBM Power 9 CPUs), it was also tested on a single-node machine with two Intel Xeon Gold 5218 at 2.30 GHz, with 64 cores and 200 GB of main memory. It, however, should be compiled on any CPU (with multiple cores) capable of compiling OpenMPI 4.1.

**Software and OS:** Tested with Ubuntu Server 20.04 operating system, and all the codes are compiled with C++20 (e.g., gcc 12). The requirements for building the codes are as follows:

- Python 3.8.10
  - command
  - argparse
- M4 1.4.18
- Autoconf 2.69
- Automake 1.16.1
- Libtool
- Flex 2.6.4

### *B. Environment Setup and Build process*

**Step-1: Setup the basic environment:** To facilitate installing the requirements and dependencies, their installation script is placed in `script/` directory and can be installed with the following command:

```
pip3 install -r
empi/scripts/requirements.txt
```

Then, and before the building process, make sure to source the `set-env.sh` to set up the environment as follows:

```
source set_env.sh
```

**Step-2: Setup the customized OMPI:** There are some dependencies related to OpenMPI. For ease of use, all these dependencies are included under the folder `sources/`, and by running the script `build_openmpi.sh`, you can unpack and compile OpenMPI and the dependencies as follows.

```
source build_openmpi.sh
```

The OpenMPI code is located under `deps/openmpi`.

**Step-3: Setup EMPI and the Microbenchmarks:** For building EMPI, follow the instructions below:

```
cd empi
mkdir build
cd build
cmake .. -DCMAKE_BUILD_TYPE=Release
    -DCMAKE_CXX_COMPILER=
    $WORKSPACE/deps/openmpi/bin/mpicxx
    -DCMAKE_CXX_FLAGS='-O3
    -ffast-math -march=native
    -I$WORKSPACE/deps/openmpi/include'
make -j
```

**Step-4: Building LULESH:** The LULESH application can be found under `empi/benchmarks/LULESH`. LULESH uses the CMake variable `-DWITH_MPI=[1|0]` to enable the MPI implementation and `-DUSE_EMPI=[1|0]` to enable the EMPI version. To build LULESH with EMPI, type:

```
cd $WORKSPACE/empi/benchmarks/LULESH
mkdir build
cd build
cmake .. -DCMAKE_BUILD_TYPE=Release
    -DCMAKE_CXX_COMPILER=
    $WORKSPACE/deps/openmpi/bin/mpicxx
    -DCMAKE_CXX_FLAGS='-O3
    -ffast-math -march=native
    -I$WORKSPACE/deps/openmpi/include'
    -DWITH_MPI=1 -DUSE_EMPI=1
    -DEMPI_PATH=$WORKSPACE/empi/include
make -j
```

### *C. Running the Experiments*

There are six python scripts under `empi/scripts/` naming:

- `minibench.py`
- `lulesh.py`
- `vibrating_string.py`
- `paper_minibench.py`

- `paper_lulesh.py`
- `paper_vibrating_string.py`

The first three will launch the microbenchmarks and applications while taking some command line arguments to customize each run and manually specify the input parameters. The last three will replicate the paper's experiments by launching the apps with the paper's specified sizes and the number of processors. In this case, the input parameters passed by the user are ignored.

To get more information about the command line parameters to configure the benchmarks and applications execution, type `python [minibench.py | lulesh.py | vibrating_string.py] --help`.

For example, to run all the micro-benchmarks with:

- 4 Processors (`--num_proc=4`)
- 4 Bytes of message size (`--size=2`)
- 10 Inner iterations per benchmark (`--app_iter=10`)
- 20 Outer repetitions per each mpirun (`--app_restart=20`)
- Root permissions (if you are root)
- Default dependencies path

You can run the experiments with the following command, and for the applications, you just need to replace the `minibench.py` python script with the related script mentioned before.

```
python3 minibench.py
    --num_proc=4
    --size=2
    --app_iter=10
    --app_restart=20
    --root
```

Reminding that the parameters only need to be passed to the first three scripts under `empi/scripts/` and the ones with `paper_` prefix can be executed without passing any parameters.