# Brief Announcement: (Near) Zero-Overhead C++ Bindings for MPI*

Demian Hespe
Researcher
Munich, Germany
demian.hespe@outlook.com

Lukas Hübner†
Karlsruhe Institute of Technology
Karlsruhe, Germany
huebner@kit.edu

Florian Kurpicz
Karlsruhe Institute of Technology
Karlsruhe, Germany
kurpicz@kit.edu

Peter Sanders
Karlsruhe Institute of Technology
Karlsruhe, Germany
sanders@kit.edu

Matthias Schimek
Karlsruhe Institute of Technology
Karlsruhe, Germany
schimek@kit.edu

Daniel Seemaier
Karlsruhe Institute of Technology
Karlsruhe, Germany
daniel.seemaier@kit.edu

Tim Niklas Uhl‡
Karlsruhe Institute of Technology
Karlsruhe, Germany
uhl@kit.edu

## ABSTRACT

The Message-Passing Interface (MPI) and C++ form the backbone of high-performance computing and algorithmic research in the field of distributed-memory computing, but MPI only provides C and Fortran bindings. This provides good language interoperability, but higher-level programming languages make development quicker and less error-prone.

We propose novel C++ language bindings designed to cover the whole range of abstraction levels from low-level MPI calls to convenient STL-style bindings, where most parameters are inferred from a small subset of the full parameter set. This allows for both rapid prototyping and fine-tuning of distributed code with predictable runtime behavior and memory management. Using template-metaprogramming, only code paths required for computing missing parameters are generated at compile time, which results in (near) zero-overhead bindings.

## CCS CONCEPTS

• **Computing methodologies** → **Distributed algorithms**; *Massively parallel algorithms*; **Parallel programming languages**; • **Software and its engineering** → *Software libraries and repositories.*

## KEYWORDS

MPI; Modern C++; Parallel Programming Libraries

*A full version [7] is available at https://doi.org/10.48550/arXiv.2404.05610. The presented library is available at https://github.com/kamping-site/kamping.
†Also with Heidelberg Institute for Theoretical Studies.
‡Corresponding author.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
*SPAA '24, June 17–21, 2024, Nantes, France*
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0416-1/24/06
https://doi.org/10.1145/3626183.3660260

**ACM Reference Format:**
Demian Hespe, Lukas Hübner, Florian Kurpicz, Peter Sanders, Matthias Schimek, Daniel Seemaier, and Tim Niklas Uhl. 2024. Brief Announcement: (Near) Zero-Overhead C++ Bindings for MPI. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '24), June 17–21, 2024, Nantes, France.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3626183.3660260

## 1 INTRODUCTION

The first version of the Message-Passing Interface (MPI) was proposed by the Message-Passing Interface Forum in 1994 [9] with the goal to standardize a portable, flexible, and efficient standard for message-passing. Today, it is the backbone of most HPC applications. While the majority of them is written in C++ [8], MPI's syntax and semantics are designed around C and Fortran. While this allows for calling MPI from C++ code, the semantics do not fit well with modern C++ language features. This makes developing MPI application in C++ unintuitive and error-prone [10].

MPI 2.0 (1997) introduced C++ bindings, which were deprecated with MPI 2.2 (2009). With version 3.0 (2012), the bindings have been removed entirely, because they only added minimal functionality over the C bindings while adding significant maintenance complexity to the MPI specification [9].

Since then, there have been various efforts in designing new C++ interfaces. Notable libraries include *Boost.MPI* [6], *RWTH-MPI* [4] and *MPL* [1], which has recently been considered as a starting point for new C++ language bindings by the newly formed MPI working group on language bindings [5]. For a detailed overview of related work we refer the reader to our technical report [7]. While all of them agree that introducing compatibility with STL containers, automatic datatype deduction and an object oriented interface are key building blocks of such bindings, each new library chooses its own level of abstraction, requiring different amounts of boilerplate code. This may come at a performance penalty [5] and may introduce additional sources of errors. To solve this we introduce **KaMPIng** (Karlsruhe MPI next generation), a library of novel C++ bindings for MPI.

The library's main goal is to cover the complete range of abstraction levels over MPI calls, which makes it both easy to use but also introduces nearly no overhead compared to using the C bindings. Each parameter of an MPI call can either be provided directly by the user or is computed by KaMPIng. It further offers complete control over memory allocations. Because all this is achieved using template meta-programming, only the code paths programmers would have to write themselves are generated, which makes these new bindings near zero-overhead.

## 2 OVERVIEW AND DESIGN

Similar to most previous bindings, KaMPIng represents MPI objects such as communicators, requests and statuses as classes and operations on them as member functions. Resource management is achieved using *RAII* (Resource Acquisition Is Initialization) which is a commonly used C++ idiom. Also, C++ datatypes are mapped to MPI types at compile time, which prevents type matching errors. STL containers which allow access to the underlying contiguous memory are directly supported, i.e., every container which models the `std::contiguous_range` concept. Raw pointers are supported via `std::span` as proposed in the C++ core guidelines.

One of KaMPIng's distinct features is parameter handling. A common source of programming errors in MPI stems from the complexity of the interface of many communication calls. In particular, variable collective operations (suffixed with `v`) where the amount of data transferred between each processor pair varies, require a large number of parameters. The data to be sent or received is described in terms of a pointer to a memory region, the datatype, the number of elements and its displacements. This makes MPI calls verbose and requires programmers to often consult the documentation for required parameters and ordering. While all parameters are necessary for full flexibility, there exist many use cases where only a small subset of explicitly provided parameters suffices and the rest can be inferred from them.

How we achieve this with near zero-overhead is discussed in Section 2.1. If the number of elements to receive is already known (or provided as a default), it may be desirable to resize containers appropriately, but for highly-tuned applications such hidden allocation may be unfavorable. We therefore propose a flexible allocation control in Section 2.2, which is configurable at compile time. As the MPI standard continuously grows, C++ bindings also need to evolve while maintaining compatibility with existing code and MPI features not covered yet by such bindings. A key aspect here is to keep KaMPIng's core small, but allow easy integration of features via plugins. We discuss this in Section 2.3.

### 2.1 Computation of Default Parameters

As discussed previously, MPI calls often allow for computing useful default values for an operation based on only a small subset of parameters. Consider the case, where we want to perform an `MPI_Allgatherv`, where each rank initially holds an `std::vector` of varying size and we want to join them to a global vector on each rank. Then the send count and datatype can be directly inferred from the vector. The receive counts and displacements can be computed by an `MPI_Allgather` of all send counts and followed by an exclusive prefix sum. While this is a common pattern, none of the other C++ bindings allows to avoid this boilerplate code. Boost.MPI offers various overloaded functions which allow the user to omit explicit displacements, but the counts have to be communicated. RWTH-MPI does provide a default variant which gathers the counts, but it only works with `MPI_IN_PLACE` which requires a different data layout and is therefore not applicable. This leaves us with a situation where the usability of C++ bindings depends on whether the implementors had our particular use case in mind and provided a default option for it.

To address this, we choose an alternative approach inspired by *named parameters*, where parameters passed to a function can be named at the caller site and passed in arbitrary order (as known from languages like Python). This allows us to check for the presence of each parameter and to compute default values only if the respective parameter is omitted, without resorting to many overloads exploring the complete combinatorial explosion of parameters. To avoid runtime overhead, we rely on template meta-programming to only generate the code paths required for computing missing parameters at compile time. To allgather a vector , other bindings require at least 5 lines of code [7], while the KaMPIng version is a one liner:

```
std::vector<T> v_glob = comm.allgatherv(send_buf(v));
```

The use of named parameters also improves readability if multiple arguments are passed to a call.

### 2.2 Controlling Memory Allocation

Previous MPI wrappers have no unified way of controlling memory allocation. They either accept containers which are always resized to fit, or, if resizing is not desired, the user has to pass raw pointers directly. They also offer no control over allocation happening for default parameter computation, but always use the default allocator and `std::vector` for auxiliary data structures.

KaMPIng allows for fine-grained control over memory management. Each parameter accepting a container takes an optional template parameter indicating its *resize policy*, which controls whether it is always resized to fit, resized if it does not have enough space to store the result, or performs no checking and assumes that the memory location pointed to by the container is large enough. Data may either be passed by reference or by value and the user may control which of the internal auxiliary data structures are returned by value. For example, this allows to either receive a message into a newly allocated vector which is returned as a result of the call or into a reusable receive container, which is passed by reference.

If KaMPIng has to create auxiliary data structures to compute missing parameters, the user may either pass preallocated containers to use or can provide the container's type via template parameters. Recall that additional allocation is omitted entirely when parameters are provided by the user.

Again, through the use of template meta-programming, this involves no additional overhead compared to a hand-rolled implementation. This flexibility allows to quickly implement distributed algorithms and then iteratively fine-tune memory allocations and library inferred values. This facilitates an algorithm engineering workflow which involves iterative refinement of implementations and analysis through experimentation.

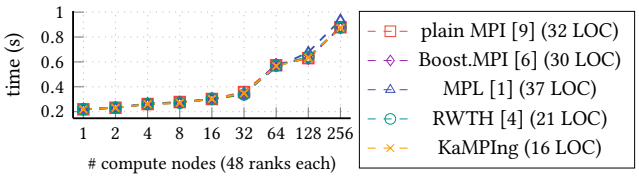


**Figure 1: Running time of and lines of code (LOC) of sample sort using different MPI bindings. KaMPIng has the smallest mean squared error compared to plain MPI.**

## 2.3 Expandability

While the main goal of KaMPIng is to design C++ bindings for MPI which cover most usage scenarios, this is impossible to achieve.

We therefore designed KaMPIng with expandability and compatibility with existing MPI code in mind, to allow easy extension and alteration of it's core features. KaMPIng's plugin interface allows overriding and adding member functions of a communicator object (e.g. collectives) without changing existing application code. For example, this allows enhancing communicators with primitives tailored for irregular sparse all-to-all exchanges [7].

Ease of development for MPI applications could also be massively improved by providing a standard library of distributed algorithms and data structures, but incorporating this in KaMPIng's core would make it overly complex.

Via plugins we can keep KaMPIng's core library small, while providing a base for third party general purpose MPI libraries and keeping maintenance low, in order not to follow the fate of the official MPI C++ interface.

# 3 EVALUATION

To highlight the usability and (near) zero-overhead of our library, we show how to implement a small real world example using KaMPIng. We use a text book implementation of distributed sample sort [11] and reimplemented it using all previously proposed C++ MPI bindings. The full KaMPIng implementation is shown in Fig. 2. [1]

Figure 1 shows the running time of the different implementations using up to 256 compute nodes of SuperMUC-NG. We sort a distributed array of $10^6$ random 64-bit integers per rank.

We see that KaMPIng introduces no additional overhead compared to a hand-rolled implementation in plain MPI or other libraries. The implementation is the shortest, being 3 times shorter than the plain MPI variant. The more concise code improves readability and removes potential for programming errors.

These findings are not limited to sample sort, but also apply to more complex applications in graph processing, suffix array construction and phylogenetic interference [7].

# 4 CONCLUSION

We introduced KaMPIng, a set of novel near zero-overhead C++ MPI bindings. Through configurable inference of parameter defaults, fine-grained allocation control, and a flexible plugin system, it enables rapid prototyping and careful engineering of distributed algorithms.

[1] See https://github.com/kamping-site/kamping-examples for full code.

```
void sort(std::vector<T>& data, Communicator comm) {
  using namespace std;
  const size_t num_samples = 16 * log2(comm.size()) + 1;
  vector<T> lsamples(num_samples);
  sample(data.begin(), data.end(), lsamples.begin(),
         num_samples, mt19937{random_device{}()});
  auto gsamples = comm.allgather(send_buf(lsamples));
  sort(gsamples.begin(), global_samples.end());
  for (size_t i = 0; i < comm.size() - 1; i++) {
    gsamples[i] = global_samples[num_samples * (i + 1)];
  }
  gsamples.resize(comm.size() - 1);
  vector<vector<T>> buckets = build_buckets(data, gsamples);
  data = with_flattened(buckets).call([&](auto... flattened) {
    return comm.alltoallv(std::move(flattened)...);
  });
  sort(data.begin(), data.end());
}
```

**Figure 2: Distributed sample sort using KaMPIng.**

KaMPIng is publicly available, extensively tested, and currently used in multiple research projects. It covers the most commonly used MPI features [8] and we plan to extend standard coverage. We are currently working towards our goal of building a basic algorithmic toolbox on top of KaMPIng, to ease rapid prototyping and analysis of distributed algorithms with a strong focus on performance. With distributed containers and optimized communication primitives, we want to enable lightweight bulk parallel computation inspired by MapReduce [3] and Thrill [2], while not locking the programmer into the walled garden of a particular framework. We strive to establish KaMPIng as a stable core for a whole ecosystem of future general purpose distributed algorithms and applications.

# ACKNOWLEDGMENTS

The authors gratefully acknowledge the Gauss Centre for Supercomputing e.V. (www.gauss-centre.eu) for funding this project by providing computing time on the GCS Supercomputer SuperMUC at Leibniz Supercomputing Centre (www.lrz.de).

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 882500).

This work was supported by a grant from the Ministry of Science, Research and the Arts of Baden-Württemberg (Az: 33-7533.-9-10/20/2) to Peter Sanders and Alexandros Stamatakis.

# REFERENCES

[1] Heiko Bauke. 2015. *MPL - A message passing library.* https://github.com/rabauke/mpl
[2] Timo Bingmann et al. 2016. Thrill: High-performance algorithmic distributed batch data processing with C++. In *IEEE BigData.* https://doi.org/10.1109/BigData.2016.7840603
[3] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified data processing on large clusters. (2004).
[4] Ali Can Demiralp et al. 2022. A C++20 Interface for MPI 4.0. Poster at SC'22.
[5] Sayan Ghosh et al. 2021. Towards Modern C++ Language Support for MPI. In *ExaMPI@SC.* IEEE, 27–35.
[6] Douglas Gregor and Matthias Troyer. 2005–2007. *Boost.MPI.* version 1.84.
[7] Demian Hespe et al. 2024. KaMPIng: Flexible and (Near) Zero-Overhead C++ Bindings for MPI. *CoRR* (2024). http://arxiv.org/abs/2404.05610
[8] Ignacio Laguna et al. 2019. A large-scale study of MPI usage in open-source HPC applications. In *SC.* ACM, 31:1–31:14.
[9] MPI Forum. 2009. MPI: A Message-Passing Interface Standard – Version 2.2.
[10] Martin Ruefenacht et al. 2021. MPI Language Bindings Are Holding MPI Back. *CoRR* (2021). http://arxiv.org/abs/2107.10566v1
[11] Peter Sanders, Kurt Mehlhorn, Martin Dietzfelbinger, and Roman Dementiev. 2019. *Sequential and Parallel Algorithms and Data Structures.* Springer.