KaMPIng: Flexible and (Near) Zero-overhead C++ Bindings for MPI

Demian Hespe *, Lukas Hübner †, Florian Kurpicz †, Peter Sanders †,
Matthias Schimek †‡, Daniel Seemaier †, Christoph Stelz †§, Tim Niklas Uhl †‡
* demian.hespe@outlook.com
†*Institute of Theoretical Informatics*
Karlsruhe Institute of Technology
Karlsruhe, Germany
{huebner, kurpicz, sanders, schimek, daniel.seemaier, uhl}@kit.edu
§christoph.stelz@student.kit.edu
‡ Corresponding authors

***Abstract*—The Message-Passing Interface (MPI) and C++ form the backbone of high-performance computing, but MPI only provides C and Fortran bindings. While this offers great language interoperability, high-level programming languages like C++ make software development quicker and less error-prone.**

We propose novel C++ language bindings that cover all abstraction levels from low-level MPI calls to convenient STL-style bindings, where most parameters are inferred from a small subset of parameters, by bringing named parameters to C++. This enables rapid prototyping and fine-tuning runtime behavior and memory management. A flexible type system and additional safeness guarantees help to prevent programming errors.

By exploiting C++'s template-metaprogramming capabilities, this has (near) zero-overhead, as only required code paths are generated at compile time.

We demonstrate that our library is a strong foundation for a future distributed standard library using multiple application benchmarks, ranging from text-book sorting algorithms to phylogenetic interference.

***Index Terms*—MPI, modern C++, parallel programming libraries**

I. Introduction

The first version of the Message-Passing Interface (MPI) was proposed by the Message-Passing Interface Forum in 1994 [1] 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++ [2], 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 [3].

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 [4].

Since then, there have been various efforts in designing new C++ interfaces. Notable libraries include *Boost.MPI* [5], the MPI bindings by Demiralp et al. [6], and *MPL* [7], which has recently been considered as a starting point for new C++ language bindings by the newly formed MPI working group on language bindings [8].

While all of them agree that introducing compatibility with STL containers, automatic data type 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 [8] and may introduce additional sources of errors.

To solve this, we introduce KaMPIng (Karlsruhe MPI next generation)[1]. Its main goal is to cover the complete range of abstraction levels over MPI calls, which makes it easy to use without introducing significant overhead compared to the C bindings. Parameters of MPI calls can either be provided directly by the user or are computed by KaMPIng. It further offers complete control over memory allocations.

It helps to reduce common sources of programming errors by employing compile-time error checking and prevents invalid memory access for nonblocking communication by introducing a ownership model. A flexible type system supports type-safety by generating type definitions at compile time and enables serialization when needed. 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.

We highlight the flexibility of our library by discussing a variety of different application benchmarks, including both simple examples and more complex applications scenarios. This includes applications in sorting, text processing, graph search and partitioning, and phylogenetic interference. For example, KaMPIng allows us to implement the Prefix Doubling algorithm

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

[1] https://github.com/kamping-site/kamping

for suffix array construction [19] using less than 200 lines of code and to entirely remove the custom MPI abstraction layer of RaxML-NG [29].

Our Contributions

In the following, we briefly list our contributions:

- Detailed overview over previous effort for designing C++ MPI bindings.
- A new approach to parameter handling allowing for a flexible computation of default parameters and fine-grained control over memory management for highly engineered distributed applications.
- Safety for nonblocking MPI communication calls by design.
- Reduction of verbosity and error-proneness of MPI code.
- (Near) zero-overhead compared to using plain MPI.
- Compile-time error checking with human-readable error messages.
- Demonstration of real-world applicability using a large variety of application benchmarks.
- First steps towards integrated and extensible general algorithmic building blocks for distributed computing such as specialized collectives for sparse and low-latency irregular personalized communication.

The remainder of the paper is organized as follows. We begin by providing an overview of existing MPI (C++) language bindings in Section II. In Section III, we present KaMPIng's core features like parameter handling, type deduction and serialization, handling of nonblocking communication, and our approach to ensure easy expandability. We then evaluate the applicability of KaMPIng in multiple different real-world application benchmarks in Section IV. Finally, we present first examples for general distributed computing algorithmic building blocks in Section V and conclude by discussing possibilities for future work in Section VI.

II. Related Work

Since the removal of C++ bindings from the MPI standard, there has been continuous third-party effort on designing C++ language bindings for MPI. The topic is also actively discussed in the MPI forum[2], where the following desired features where prominently mentioned:

- mapping of variables to types, type safety
- safety guarantees for nonblocking communication
- returning data by value
- support for unstructured send data, i.e. a mapping of communication partners to data buffers
- a strong debug mode
- reduction via lambda
- good interaction with C++, especially move semantics and ranges

In the following, we discuss notable libraries and summarize their design and feature set.

[2]https://github.com/mpi-forum/mpi-issues/issues/288

Boost.MPI [5]: This was the first library to enable automatic data type inference and facilitate integration with the STL by also supporting `std::vector` for input and output, and not only raw pointers. Vectors are automatically resized to fit the received data, which prevents invalid memory accesses, but leads to hidden allocation. It supports custom data types by constructing appropriate data types if possible or resorting to serialization. To enable this, users must provide a serialization function compatible with *Boost.Serialization*, specifying all members explicitly. This requires the user to ensure that data type definitions and their serializations remain synchronized. The data types are managed by a global type pool which is queried for each communication call, which ensures proper resource cleanup. The adoption of Boost.MPI has been hindered by it's tight coupling with other Boost libraries and the fact that it performs implicit serialization if data types are not directly supported by MPI [8]. Additionally, it is the only MPI library considered here that is not header-only. While this is usually not a problem with modern build tools, it becomes especially tedious when switching between MPI implementation, as it requires a separate build of Boost.MPI for each, because the MPI standard does not enforce ABI compatibility. Besides KaMPIng, it is the only library that supports mapping STL functors such as `std::plus` to the corresponding built-in MPI constant for reduction operations (which may enable optimization by the MPI implementation) and writing custom reduction operations using a simple lambda. If an MPI error occurs, an exception is thrown. Boost.MPI does not provide any bindings for `MPI_Alltoallv`. It is not actively maintained since 2008 and therefore only supports MPI-1.1 features. There exists a (non-official) rewrite of Boost.MPI, called *boost-mpi3* [9], which aims to extend Boost.MPI to the feature set of MPI-3 and C++ -17. While incorporating support for iterators, MPI one-sided communication and MPI shared memory, it does not substantially improve upon Boost.MPI's design, closely mirroring MPI's C interface.

MPL [7]: Since 2015 Bauke maintains *MPL*, a library providing MPI bindings targeting C++ -17.

MPL introduces a powerful type system based on so called *layouts* to programmatically construct views over chunks of contiguous memory, which can be converted to MPI data types. While this allows for flexible communication in scientific computations, such as halo exchanges, it requires a lot of code for irregular communication patterns often found in discrete algorithms. MPL does not expose the underlying native MPI representation of constructs such as communicators, and types directly, which complicates iterative adaption of existing MPI code to MPL. MPL offers support for custom data types, which requires to define a matching layout, but has no serialization support. It currently does not support error handling.

Recently, MPL has been considered by the members of the newly formed MPI working group on language bindings as a starting point for new C++ bindings [8]. While the authors highlight the simplified interface, which offers abstraction with default parameters using overloads, they also show that MPL incurs overheads for variable size collectives. This is due

to the fact that it does not use the corresponding collective operations directly, but works with derived data types that are costly in some MPI implementations. For example, `gatherv` is implemented based on `MPI_Alltoallw` [8].

RWTH-MPI [6]: Demiralp et al. recently introduced another modern C++ interface, which we call RWTH-MPI in the following. They offer full STL support for send and receive buffers. For each communication call, they provide various overloads using different abstraction levels, which often allow the omission of send or receive counts, and RWTH-MPI employs additional communication to compute them. Automatic resizing of the receive buffer is supported in some cases and can be disabled. For custom types which support reflection based on the PFR library [10], RWTH-MPI constructs appropriate MPI data types automatically. A customization point for mapping C++ types to type definitions for MPI is available, but does not provide automatic type management. Using types with dynamic run time sizes is not supported.

While RWTH-MPI covers the complete MPI standard, large parts directly mirror the C interface without providing additional convenience or safety guarantees.

Beyond C++ : There also exist MPI bindings for other high-level programming languages. The Python bindings provided by *mpi4py* [11, 12] may be considered as the most mature third-party bindings, as they are actively developed for over a decade. It enables transparent sending of Python objects using serialization. While this introduces additional overheads, using mpi4py in conjunction with NumPy arrays allows performance on par with a native C implementation [11]. They provide default values for certain parameters, but only when no additional communication is necessary for that. Finally, rsmpi [13] introduces idiomatic Rust bindings for MPI. While they make using MPI from Rust more ergonomic than using the C interface, the missing default parameters require the writing of much boilerplate code.

III. KaMPIng: 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* [14, E.6][15] (Resource Acquisition Is Initialization) which is a commonly used C++ idiom. Also, C++ data types 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 [14, I.13]. 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 data type, 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 remaining ones can be inferred from them. How we achieve this with near zero-overhead is discussed in Section III-A. In Section III-B, KaMPIng's overall parameter handling concept is explained in more detail. If the number of elements to receive is already known, 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 III-C, which is configurable at compile time.

The missing type introspection features of C require MPI users to explicitly specify the layout of data types. If the type declaration goes out of sync with the actual data layout, this may lead to hard-to-find errors.

In Section III-D we introduce KaMPIng's flexible type system which provides type-safety through compile data type construction, and offers additional versatility through support for runtime-sized types and serialization.

Non-blocking communication in MPI introduces additional sources of errors, as the user has to manually wait for the completion of operations and has to take care to not perform invalid memory accesses before a operation has finished. To address this, we propose memory-safe abstractions which prevent illegal memory accesses through the use of C++ 's ownership model and move semantics in Section III-E.

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 additional features via plugins. We discuss this in Section III-F.

A. 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. As an example, 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 concatenate them to a global vector on each rank. The send count and data type can be directly inferred from the vector's size and `value_type` (see Section III-D for more details on automatic type deduction). Receive counts and displacements can be computed by an `MPI_Allgather` of all send counts followed by an exclusive prefix sum over them (see Fig. 1). While this is a common pattern, none of the other C++ bindings allows to avoid all 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 an overload which gathers the counts internally, but it only works with `MPI_IN_PLACE`, which requires the send data to be already provided at the correct position on each rank. To achieve this the

```
std::vector<T> v = ...; // fill with data

// allgathering a vector using MPI
int size;
MPI_Comm_size(comm, &size);
int n = static_cast<int>(v.size());
std::vector<int> rc(size), rd(size);
MPI_Allgather(&n, 1, MPI_INT, rc.data(), 1, MPI_INT, comm);
std::exclusive_scan(rc.begin(), rc.end(), rd.begin(), 0);
int n_glob = rc().back() + rd.back();
std::vector<T> v_glob(n_glob);
MPI_Allgatherv(v.data(), v_size, MPI_TYPE, v_glob.data(),
 rc.data(), rd.data(), MPI_TYPE, comm);

// allgathering a vector using __MPIng
std::vector<T> v_glob = comm.allgatherv(send_buf(v));
```

Fig. 1. Allgathering of a vector using MPI vs. KaMPIng.

user has to manually exchange count information upfront [3]. This leaves us with a situation where the usability of C++ bindings depends on whether the implementers had this particular use case in mind and provided a default option for it.

To address this problem, 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). Internally, named parameters are realized as *factory functions* [16] which construct lightweight parameter objects encapsulating the parameter type (i.e., send buffer, send counts, ...) and the corresponding data. This allows us to check for the presence of each parameter at compile time 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. Gathering a vector in KaMPIng then becomes a one-liner as shown in Fig. 1. The implementations using other bindings are more verbose (see Table I).

B. Input and Output Parameters

KaMPIng extends MPI's definition of *in(put)*- and *out(put)*-parameters. With an in-parameter, the caller provides data to the wrapped MPI call, such as with `send_buf(data)`. By passing an out-parameter, e.g., `recv_counts_out()`, the caller asks the library to compute the requested parameter and return its result. Most MPI parameters like send displacements, receive counts, etc. can either be passed as in- or out-parameters since they can be internally deduced in many cases using additional computation or communication as outlined in Section III-A.

The parameter type is determined by the named parameter factory functions: To give an example, `send_displs(data)` creates an in-parameter containing the send displacements as specified in `data` whereas `send_displs_out()` creates an out-parameter signaling to return the send displacements by value.

[3]Full example code can be found at https://github.com/kamping-site/kamping-examples/tree/main/include/vector_allgather/

Since one is primarily interested in the receive buffer in MPI calls, this parameter is always implicitly returned by KaMPIng. To retrieve other parameters from the wrapped MPI call one has to explicitly pass the corresponding out-parameter. This is a major improvement over previous MPI libraries, which simply mimic MPI's C-Interface and return output data next to the receive buffer by pointer or reference. This is not in line with the C++ core guidelines which strongly suggest to return output data by value [14, F.20]. Furthermore, it is often unclear which additional parameters are computed by the library, as the overloaded wrapped MPI function calls differ only in the number of function arguments. Combined with the large number of parameters of MPI calls, it is hard to see which argument refers to which parameter when looking at the code.

KaMPIng on the other hand improves this situation in two regards:

1) The caller can decide which non-required parameters they want KaMPIng to compute internally. By the named parameter approach this decision is clearly documented in the source code and correctness does not depend on a common understanding of the parameter order.
2) For each requested out-parameter, the caller can individually decide how the data is returned.

In the following this is illustrated with a call to the wrapped `MPI_Allgatherv` function.

```
auto result = comm.allgatherv(send_buf(v),
                              recv_counts_out());
auto recv_buf = result.extract_recv_buf();
auto counts = result.extract_recv_counts();
```

The above call to **`allgatherv`** returns a result object containing the (implicitly) requested receive buffer and the receive counts. These can then be extracted from the result object using *move* semantics. It is furthermore possible to directly decompose the result object using C++'s *structured bindings* which simplifies the call to

```
auto [recv_buf, counts] = comm.allgatherv(send_buf(v),
                                          recv_counts_out());
```

A shortcoming of returning by value is the redundant memory allocation in case a previously allocated container could be reused instead. In KaMPIng, we offer two solutions for this scenario. For containers supporting C++ move semantics, a previously user allocated container can be simply moved to the underlying call and is then subsequently returned with the result object by value.

```
std::vector<T> tmp = ...;
// tmp is moved to the underlying call where the
// storage is reused for the recv buffer
auto recv_buffer = comm.allgatherv(
                        send_buf(v),
                        recv_buf(std::move(tmp)));
```

If there is no efficient way to support move semantics for a container type, it is also possible to pass the container via reference to the underlying call. The data computed by KaMPIng for the requested parameter will then be written directly to the specified memory-location.

```
std::vector<T> recv_buffer = ...;
// data is written into recv_buffer directly
```

```
comm.allgatherv(send_buf(v),
                recv_buf(recv_buffer));
```

C. 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.

KaMPIng allows for fine-grained control over memory management. Each (out-)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 capacity of the container is large enough, which is the default.

```
std::vector<T> recv_buffer; // has to be resized
std::vector<int> counts(comm.size()); // size large
                                      // enough
comm.allgatherv(send_buf(v),
    recv_buf<resize_to_fit>(recv_buffer),
    recv_counts_out(counts));
```

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.

As all of this relies on template meta-programming, there is 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.

D. Using Custom Types

HPC applications use a variety of data types which need to be communicated. Beyond *basic data types* corresponding directly to C++'s built-in types, MPI allows for complex *derived data types* using type constructors, such as `MPI_Type_create_struct` and `MPI_Type_create_contiguous`.

C's lack of type-introspection forces users to always pass the type explicitly to a communication call, which is both tedious and error-prone, as type definitions need to be kept in sync with the actual data layout.

Template-metaprogramming enables mapping arbitrary C++ types one-to-one to MPI data types (we call this a *static type*). MPI's derived data types, however, form a superset of C++ data types. This is because MPI allows constructing arbitrary type signatures with sizes known only at runtime (we call this a *dynamic type*). KaMPIng provides support for static and dynamic types and offers implicit static type construction without performance pitfalls. Sometimes, applications need to communicate unstructured and complex data types off the critical code path. To support this with minimal code overhead, KaMPIng provides transparent serialization support. In the following, we discuss these aspects in more detail.

```
struct MyType {
    int a;
    double b;
    char c;
    std::array<int, 3> d;
};
namespace __mping {
// using __MPIng's built-in struct serializer
template <>
struct mpi_type_traits<MyType> : struct_type<MyType> {};

// or using an explicitly constructed type
template <>
struct mpi_type_traits<MyType> {
    static constexpr bool has_to_be_committed = true;
    static MPI_Datatype data_type() {
        MPI_Datatype type;
        MPI_Type_create_*(..., &type);
        return type;
    }
};
} // namespace __mping
```

Fig. 2. Defining custom static types using automatic type reflection or a custom type definition.

1) Static derived data types at compile time: KaMPIng maps basic C++ data types to their MPI counterparts and supports complex data types on homogeneous systems if they are *trivially copyable*, i.e., the C++ standard guarantees that they can be copied into a **`char`** array. In this case, we create a contiguous type using `MPI_Type_contiguous` with the appropriate number of bytes, as this provides a sensible default which usually achieves better performance than using `MPI_Type_struct` (see Section III-D4). For all other types, the user can directly provide static type definitions by providing an explicit instantiation of the `mpi_type_traits` template for the desired type, which describes how to construct a matching `MPI_Datatype`, as show in Fig. 2. While this allows for building data types using MPI's type constructors, constructing a correct type for a given C++ struct is error-prone as the programmer has to keep the type-construction calls in sync with the data type. We leverage the *PFR* library [10] to automatically generate MPI type definitions for user-provided structs at compile time. This can be enabled by inheriting from the type constructor when defining the type trait (see Fig. 2).

MPI requires the user to initialize and deallocate non-built-in types. KaMPIng archives both transparently to the user by exploiting the Construct-On-First-Use-Idiom [4].

Existing MPI bindings also support non-built-in static types to some extent. For Boost.MPI, this is dependent on the definition of a serialization function. Similar to KaMPIng, types are managed using a global type pool, but each operation incurs a runtime type lookup. MPL and RWTH-MPI use the Construct-On-First-Use-Idiom to commit types before first use, and MPL also uses PFR to provide automatic type definitions by using reflection. Opposed to KaMPIng, types are not properly freed, which may result in resource leakage. RWTH-MPI allows custom static type definitions, but the user is responsible for committing and freeing types.

[4] https://isocpp.org/wiki/faq/ctors#static-init-order-on-first-use

```
using dict = std::unordered_map<std::string, std::string>;
dict data = ...;
comm.send(send_buf(as_serialized(data)));

dict recv_dict = comm.recv(
  send_buf(as_deserializable<dict>())
);
```

Fig. 3. Usage of KaMPIng's serialization.

2) Support for dynamic types: For data types constructed at runtime, KaMPIng supports passing explicit types to operations directly, by providing optional type parameters.

In KaMPIng, dynamic types can currently only be constructed using MPI's type constructors. MPL on the other hand provides a runtime type interface mirroring MPI's type constructors using the builder pattern, which encapsulates the constructed types in so-called *layouts*. While MPL's collective operations are tightly interleaved with the layout system, which results in verbose code, its type construction is a powerful feature which we plan to integrate as the default way for constructing dynamic types in KaMPIng. RWTH-MPI offers no support for dynamic types.

3) Communicating arbitrary data using serialization: Some applications require communicating non-contiguous data which is (partially) located on the heap, e.g., `std::string` or `std::unordered_map`. These have to be packed to a contiguous buffer before communication. KaMPIng facilitates this by providing serialization support, which is both highly tuneable and transparent to the user. We rely on the popular C++ serialization library *Cereal* [17], which supports STL data types and allows providing serialization routines for custom types. While serialization is transparent to the user, it also has to be explicitly enabled as it usually incurs hidden costs for allocation memory for serialized data and perform (de-)serialization. Through Cereal's flexible design, serialization in KaMPIng is also highly configurable; allowing users to specify custom serialization functions and archives, e.g., binary formats, JSON, or XML. See Fig. 3 for an example on how to use serialization in KaMPIng.

Besides KaMPIng, Boost.MPI is the only library offering serialization support, but is tightly coupled with other Boost libraries and serialization is performed implicitly; if a type is not marked as an MPI data type, serialization is used.

We are convinced, that using serialization implicitly should be avoided by zero-overhead MPI bindings, as hidden serialization incurs hidden runtime and memory overheads.

4) Towards sensible defaults for type construction: By default, KaMPIng maps trivially-copyable types to a type interpreted as a contiguous sequence of bytes. When defining a struct type where the members have alignment gaps, MPI does not include the gap in the communicated data. This requires non-contiguous memory accesses, which may be slower than copying whole memory blocks to the communication hardware (MPI standard [1, §5.1.6]). The standard suggest introducing dummy struct members to fill these gaps, but this requires the user to modify their non-MPI code. By using a type consisting of contiguous bytes when valid with respect to the C++ standard, we enable this more efficient default transparent to the user. Preliminary experiments also confirm this in practice and further highlight that serialization incurs a non-negligible overhead, which is the reason why KaMPIng uses serialization only if explicitly enabled.

```
std::vector<int> v = ...;
auto r1 = comm.isend(
    send_buf_out(std::move(v)), destination(1)
);

v = r1.wait(); // v is moved back to caller after
               // request is complete

auto r2 = comm.irecv<int>(recv_count(42));
std::vector<int> data = r2.wait(); // data only returned
                                   // after request
                                   // is complete
```

Fig. 4. Example of nonblocking safety in KaMPIng.

E. Enabling Safety for Nonblocking Communication

Nonblocking communication in MPI is important for both correct and performant applications. MPI allows to *initiate* an operation, which returns a *request handle* to the user. A user than has to *complete* the request, by either testing for completion of the request using `MPI_Test` or using `MPI_Wait` to block until the request is completed. The semantics of MPI only allow updating send buffers or reading from receive buffers taking part in a previously initiated operation when the corresponding request has been completed. This introduces an additional source for programming errors, as MPI does not hinder users from accessing the memory locations regardless of completion status.

For asynchronous (I/O) operations the C++ standard library provides `std::future`, which allows to query or wait for the result of an asynchronous operation, only returning a value once the operation has completed. Using `std::future` to provide a safe interface for nonblocking communicating is not possible, as they are tied to asynchronous progress happening in the background, which the MPI standard does not guarantee.

To solve this, we introduce a similar concept called a *nonblocking MPI result*, which encapsulates an `MPI_Request` and the data returned by value from the operation, as described in Section III-B. The data is only returned to the user by calling `result.wait()` which internally completes the request. Calling `result.test()` returns an `std::optional`, which only contains the returned data if the request is completed, and otherwise returns `std::nullopt`. This ensures that a user can only access valid received data.

To also prevent unwanted access to send buffers during nonblocking operations, the user can move the data into the call. The nonblocking result then assures that the data lives long enough and it is re-returned to the user upon completion of the call. This happens without any copying of data by relying on C++ 's move semantics. See Fig. 4 for an example.

This is enabled by KaMPIng's distinct parameter handling, making it the first C++ MPI library which provides safety guarantees for nonblocking communication. Opposed to that,

other MPI bindings offer no enhanced safety features for the data involved in a nonblocking call, but only return request handles. Here the user is still responsible for ensuring that no invalid data access happen while an operation is in progress. Only *rsmpi* provides similar guarantees, powered by Rust's ownership model.

Another feature KaMPIng provides to facilitate working with nonblocking MPI are *request pools*, which allow for easy completion of multiple requests. The user only needs to submit the request associated with the call to such a pool. While the current implementation just collects them in an unbounded array, requests pool are designed with extensibility in mind, to enable more sophisticated variants. For example, we are currently working on a request pool with a fixed number of slots, internally maintaining free slots, which allows to limit the number of concurrent nonblocking requests.

F. Extensibility

While the main goal of KaMPIng is to design C++ bindings for MPI which cover most usage scenarios, full coverage is impossible to achieve. We therefore designed it with extensibility and compatibility with existing MPI code in mind, to allow easy extension and alteration of its core features. KaMPIng's plugin interface allows overriding member functions of a communicator object (e.g., collectives) and adding additional functionality without changing existing application code. The library allows plugin implementers to define new named parameters to enable the full named parameter flexibility also for these library extensions. This architecture allows us to 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.

With KaMPIng, we already ship multiple plugins extending the functionality of the current MPI standard (see Section V).

G. More Safety Features for MPI

MPI notifies users of errors by return codes. Here, MPI makes no distinction between failures, such as insufficient buffer space or node failures, which may be recoverable, and usage errors, such as providing invalid parameters.

KaMPIng tries to improve upon this by using three major techniques, following the C++ core guidelines: using exceptions for failures [14, E.2], catching usage errors at compile time whenever possible [14, P.5], and making heavy use of assertions at runtime. While C++ 's template-metaprogramming is notorious for complex and hard-to-read compiler errors, we try to ensure that compile time assertions fail early and provide helpful human-readable error messages. For example, when the user does not provide a required parameter to a collective operation, the error message indicates which parameter is missing during compile time. KaMPIng also includes many runtime assertions verifying MPI invariants, that are grouped in different levels, ranging from lightweight checks to assertions involving additional communication. The assertions can be completely disabled level-by-level. The use of exceptions can also be completley disabled, and KaMPIng allows to override the error handling strategy using the plugin system.

In contrast to that, other MPI bindings either always convert MPI errors to exceptions or do not provide any error handling.

TABLE I
LINES OF CODE FOR EXAMPLES USING KAMPING VS. OTHER BINDINGS[5].

	MPI	Boost.MPI	RWTH-MPI	MPL	**KaMPIng**
vector allgather	14	5	5	12	1
sample sort	32	30	21	37	16
BFS	46	42	32	49	22

IV. INTEGRATING KAMPING INTO REAL-WORLD APPLICATIONS

To highlight the usability of our library, we have integrated KaMPIng into multiple (research) applications, ranging from sorting (sample sort and suffix sorting) over graph algorithms (BFS and label propagation) to a large phylogenetic interference tool. Experiments backing our (near) zero-overhead are executed on up to 256 compute nodes of SuperMUC-NG, where each node is equipped with an Intel Skylake Xeon Platinum 8174 processor with 48 cores. The internal interconnect is a fast OmniPath network with $100\,\mathrm{Gbit/s}$. Our code is compiled with g++-12.2.0 and Intel MPI 2021 using optimization level `-O3`.

A. Sorting

As a first example, we use a textbook distributed sample sort [18] with Fig. 5 showing a prototypical implementation in KaMPIng. We have implemented the sample sort algorithm using all previously discussed C++ MPI bindings in a comparable way, where all shared parts of the code have been extracted to functions and the code has been formatted identically with `clang-format` using the `Google` style template. In this setting, we require only 16 lines of code (LOC) using KaMPIng while the plain MPI example requires 32 LOC. The lines of code for all bindings can be found in Table I. Fig. 6 shows the running time of the different implementations in a weak-scaling experiment. We sort a distributed array with 10^6 64-bit integers per rank, which are drawn uniformly at random.

We see that KaMPIng introduces no additional overhead compared to a hand-rolled implementation in plain MPI or other libraries, but makes the implementation a lot easier to read and to write, while being more flexible.

Suffix Array Construction: For a more complex example we consider an application from text processing: We sort all suffixes of a text lexicographically, i.e., we compute the suffix array [19]. Here, we implemented two algorithms: DCX [20] and Prefix Doubling [19]. Our KaMPIng implementation of DCX requires 1 264 LOC whereas the plain MPI implementation [21] needs 1 396 LOC. The additional 9.5 % of code are mostly due to boilerplate code, e.g., distributing send counts for `MPI_Alltoallv` and the tedious construction of

[5]The source codes for all three examples is available at https://github.com/kamping-site/kamping-examples/.

```cpp
template<typename T>
void sort(std::vector<T>& data, MPI_Comm comm_) {
  using namespace std;
  Communicator comm(comm_);
  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(), gsamples.end());
  for (size_t i = 0; i < comm.size() - 1; i++) {
    gsamples[i] = gsamples[num_samples * (i + 1)];
  }
  gsamples.resize(comm.size() - 1);
  vector<vector<T>> buckets = build_buckets(data, gsamples);
  data.clear();
  vector<int> scounts;
  for (auto &bucket : buckets) {
    data.insert(data.end(), bucket.begin(), bucket.end());
    scounts.push_back(bucket.size());
  }
  data = comm.alltoallv(
    send_buf(data), send_counts(scounts)
  );
  sort(data.begin(), data.end());
}
```

Fig. 5. Distributed sample sort using KaMPIng.

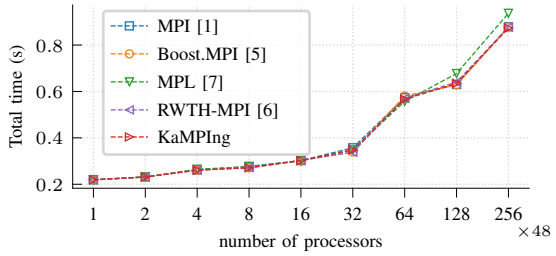

Fig. 6. Running time of sample sort using different MPI bindings.

MPI types. Likewise, our KaMPIng implementation of the Prefix Doubling algorithm requires 163 LOC. An existing plain MPI implementation of the same algorithm [22] needs 426 LOC (not counting the 1 442 LOC for wrapped MPI functionality used by the plain MPI implementation). Even when using a high-level distributed programming framework, implementing the Prefix Doubling algorithm still requires 266 LOC [23].

B. Graph Algorithms

Currently, most state of the art HPC platforms are mainly designed for numerical applications with fairly regular data access and communication patterns. However, data intensive irregular workloads become more and more frequent, for example, in materials science or data analysis tasks such as (human) brain analysis. Graphs are commonly used to represent data sets in these applications and we therefore require efficient distributed graph algorithms either for network analysis or as important building blocks for more complex applications [24, 25, 26]. To demonstrate the applicability of KaMPIng in this setting, we provide a simple distributed breadth-first search (BFS) implementation in Fig. 7. We assume the graph to be distributed among the ranks with each rank holding a subset of the vertices and their incident edges. Locally, the graph is

```cpp
using VId = size_t;
using VBuf = std::vector<VId>;
constexpr VId undef = std::numeric_limits<VId>::max();

bool is_empty(auto &frontier, Communicator const& comm) {
  return comm.allreduce_single(send_buf(frontier.empty()),
                               op(std::logical_and<>{}));
}
VBuf exchange(auto frontier, Communicator const &comm) {
  return with_flattened(frontier, comm.size()).call(
    [&](auto... flattened) {
      return comm.alltoallv(std::move(flattened)...);
    });
}
vector<size_t> bfs(Graph const &g, VId s, MPI_Comm _comm) {
  Communicator comm(_comm);
  VBuf frontier;
  std::unordered_map<int, VBuf> next_frontier;
  if (g.is_local(s)) {
    frontier.push_back(s);
  }
  std::vector<size_t> bfs_levels(g.local_size(), undef);
  size_t bfs_level = 0;
  while(!is_empty(frontier, comm)) {
    expand_frontier(frontier, next_frontier, bfs_levels);
    frontier = exchange(std::move(next_frontier), comm);
    next_frontier.clear();
    ++bfs_level;
  }
  return bfs_levels;
}
```

Fig. 7. Distributed BFS using KaMPIng.

represented as an adjacency array. For each vertex v, the `bfs` returns the distance (number of hops) between the source vertex `s` and v. KaMPIng's utility function **`with_flattened`**`(...)`, which *flattens* a container of nested destination-message pairs by transforming it into a contiguous range while also providing send counts, proves to be especially useful.

As for the sample sort example, we implemented the distributed BFS algorithm using all previously discussed (C++) MPI bindings in a comparable way. The implementations only differ for the frontier exchange and completion logic, which can be implemented in KaMPIng using only 22 lines of code, whereas plain MPI requires 46 lines. Our closest competitor regarding code length is RWTH-MPI with 32 LOC (see Table I for all other bindings) [6].

In Fig. 8, we provide a performance evaluation of the implementations on a variety of different graph families, which is discussed in more detail in Section V-A.

Graph Partitioning. As a more complex showcase, we integrated KaMPIng into the state-of-the-art distributed multilevel graph partitioner dKaMinPar [27], consisting of roughly 30 000 LOC and including its own abstraction layer with specialized graph-specific communication primitives over plain MPI. The partitioner uses size-constrained label propagation to iteratively cluster and contract the input graph, shrinking it down until its size falls below a certain threshold. Due to the project's size, we only focus on this component and compare an implementation based on dKaMinPar's application-specific MPI abstraction layer, a plain MPI-based implementation and a KaMPIng-based

[6]The code considered here is structured slightly differently compared to Fig. 7, which has been shortened for readability. See https://github.com/kamping-site/kamping-examples/tree/main/include/bfs/ for full implementations.

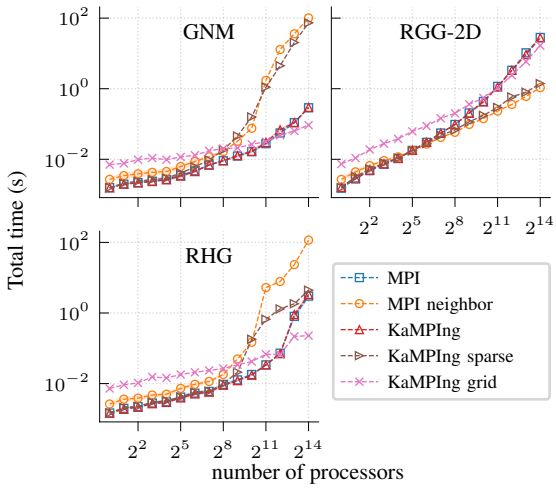

Fig. 8. Running time of BFS using different all-to-all variants.

```
// Before. The self-written mpi_broadcast(...) wrapper and
// serialization/deserialization of data is not shown.
template<typename T>
static void mpi_broadcast(T& obj) {
    if (_num_ranks > 1) {
    size_t size = master() ?
      BinaryStream::serialize(
        _parallel_buf.data(),
        _parallel_buf.capacity(),
        obj)
      : 0;
    mpi_broadcast((void *) &size, sizeof(size_t));
    mpi_broadcast((void *) _parallel_buf.data(), size);
    if (!master()) {
      BinaryStream bs(_parallel_buf.data(), size);
      bs >> obj;
    }
  }
}

// After. KaMPIng provides all required functionality.
template <typename T>
static void mpi_broadcast(T &obj) {
  if (_num_ranks > 1) {
    _comm->bcast(send_recv_buf(as_serialized(obj)));
  }
}
```

Fig. 9. Example of a routine in the RAxML-NG parallelization abstraction layer simplified substantially using KaMPIng. We are able to replace custom serialization logic entirely.

implementation. We have extracted the shared code of all implementations (202 LOC) to a base class and only focus on the MPI-heavy part of the algorithm's implementation. Here, the plain MPI-based implementation (154 LOC) is roughly 17.5% larger than the KaMPIng-based implementation (127 LOC), which in turn is 16.5% larger than the implementation based on dKaMinPar's specialized abstraction layer (106 LOC). We observed the same running times for all variants.

C. Integrating KaMPIng in RAxML-NG

As our largest application benchmark, we consider RAxML-NG. RAxML [28] and its modern rewrite RAxML-NG [29] are widely used real-world programms for phylogenetic inference in the field of bioinformatics with over 50 000 citations. RAxML-NG is written in C++ and uses a custom non-trivial abstraction layer over `pthreads` + MPI parallelism with over 700 lines of code. We use KaMPIng to substantially simplify the MPI part of this abstraction layer; demonstrating that KaMPIng can easily be integrated even in large and well-established scientific codes which use hybrid parallelization (for an example, see Fig. 9). If KaMPIng would have been available at the time, the RAxML-NG developers would have never have needed to write, unit-test, maintain, and document over hundred lines of complex code.

We empirically verified that replacing the abstraction layer with KaMPIng does not incur a measurable performance overhead even though RAxML-NG issued nearly 700 MPI calls per second[7]. The binary's size does also not increase substantially (by 2.5 %); the compilation time increases from 1:15 min to 1:30 min.

V. Towards General Building Blocks for Distributed Computing

Ease of development for MPI applications can be massively improved by providing a standard library of distributed (communication) algorithms and data structures, but incorporating this functionality in KaMPIng's core would make it overly complex. With KaMPIng we ship multiple library extensions (plugins) including an STL-like distributed sorter (see Section IV-A), specialized personalized all-to-all communication, fault-tolerance, and reproducible reduction operations which we will briefly highlight in the following.

A. Sparse and Low-Latency All-To-All communication

All-to-all exchanges are one of the most frequent communication patterns in distributed computing. However, there is a large algorithmic design space for all-to-all communication ranging from algorithms with near optimal communication volume but latency at least linear in the number of processing elements to algorithms following a hypercube communication scheme with logarithmic latency but also a communication volume that is increased by a logarithmic factor [18].

With KaMPIng's `GridCommunicator` plugin we go part of the way of trading communication volume for reduced latency by resorting to two-dimensional grid communication [30]. The processors are organized in a virtual two-dimensional grid and messages are routed in two hops to their destination to achieve a message start-up latency in $\mathcal{O}(\sqrt{p})$, where p denotes the size of the communicator. This enables hardware agnostic latency reduction with asymptotic guarantees, in contrast to the variants provided by most MPI implementations.

Additionally, MPI's standard collectives have not been designed with *sparse* communication patterns in mind. `MPI_Alltoallv`, for example, requires a send counts parameter consisting of an array with one entry for each rank of the communicator. This yields a time complexity which is at least linear in the size of the underlying communi-

[7]The mean running times are less than one standard deviations apart.

cator. To mitigate this problem for static communication patterns, *neighborhood* collectives have been added to MPI-3.0 allowing the user to perform `MPI_Alltoall(v)` and `MPI_Allgather(v)` on a previously defined (sparse) graph topology. However, for applications and algorithms with rapidly changing communication partners, e.g., (dynamic) graph algorithms, the overhead of defining a new communication graph topology prior to every few all-to-all exchanges may impose too much overhead. KaMPIng's `SparseAlltoall` plugin offers a lightweight alternative for these scenarios. It introduces a `sparse_send_buf` named parameter accepting any C++ container consisting of destination-message pairs, like for example `std::unordered_map<int, std::vector<T>>`. For the actual data exchange, the plugin uses the *NBX* algorithm for sparse all-to-all communication by Hoefler et al. [31].

Fig. 8 shows an evaluation of our different all-to-all strategies using the weak-scaling BFS benchmark introduced earlier on three different graph families [32], where each rank holds 2^{12} vertices and 2^{15} edges. Erdős-Rényi graphs possess almost no locality (most edges cross rank boundaries) but a small diameter, whereas random geometric graphs (RGG) are highly local with a high diameter. Regarding locality, random hyperbolic graphs (RHG) range somewhere in between and also have small diameter. In contrast to Erdős-Rényi graphs and RGGs they possess high-degree vertices. In the experiment, we compare different algorithms for the actual frontier exchange in each BFS step: built-in `MPI_Alltoallv` (`mpi`, `KaMPIng`), KaMPIng's sparse all-to-all plugin, KaMPIng's grid all-to-all plugin, and `MPI_Neighbor_alltoallv` (`mpi_neighbor`). For RHGs (and less pronounced for Erdős-Rényi graphs) the most scalable communication method is our grid all-to-all approach. Grid all-to-all also outperforms built-in `MPI_Alltoallv` on RGGs. Due to their high diameter and locality, a competitive performance on RGGs can only be achieved with sparse communication. KaMPIng's sparse all-to-all approach is only slightly slower than `MPI_Neighbor_alltoallv`. Note that when rebuilding MPI's communication graph before each frontier exchange, which simulates dynamic communication patterns to some extent, `MPI_Neighbor_alltoallv` does not scale.

Our base KaMPIng implementation as well as RWTH-MPI and Boost.MPI (omitted in the plot) are always on par with the MPI implementation. MPL (also omitted in the plot) internally uses `MPI_Alltoallw` for all-to-all exchanges and is (considerably) slower than MPI on all configurations, as already discussed previously [8].

B. User-Level Failure Mitigation

With the increasing number of processors in high performance computing clusters, the probability that some processors fail during a computation rises. Handling such failures constitutes a major challenge for future exascale systems [33]. In upcoming systems, we expect a hardware failure to occur every 30 to 60 minutes [34, 35, 36]. The upcoming MPI 5.0 standard enables developers to develop software able to recover from such failures by employing User-Level Failure-Mitigation (ULFM) [37].

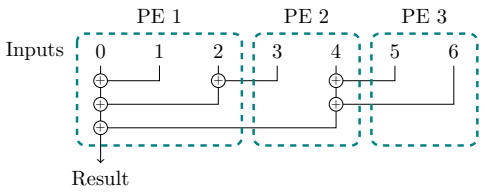

Fig. 10. Reduction tree for 7 elements distributed across 3 ranks. The operations are performed in the order described by the tree.

As one example of the potency of our plugin architecture (Section III-F) we developed an abstraction layer over ULFM supporting all functions of the proposal[8]. As plugins can add custom error handling hooks, this enables users to develop fault-tolerant algorithms using idiomatic C++ exceptions instead of checking return codes.

C. Reproducible Reduce

Reproducibility of results is a key aspect of scientific work. One challenge in distributed computing is to ensure that the results of a computation are consistent across different runs using different numbers of processors. IEEE 754 floating point math is not associative because of rounding errors and thus the order of operations, which often depends on the number of processors, influences the result.

We include an implementation of `MPI_Reduce` as a KaMPIng plugin which fixes the reduction order independent of the number of processors but is faster than a gather + local reduction + broadcast. We use a binary tree scheme inspired by Villa et al. [39] and include various performance improvements (Fig. 10; for details see Stelz [38]). This enables parallel computations while using only a few messages to exchange intermediate results.

Similar to "normal" KaMPIng `reduce`, we support plain-MPI constants, function pointers, and lambda functions as operations. We hope that the availability of a convenient off-the-shelf library method encourages more authors to ensure that their applications produce consistent and reproducible results.

VI. Conclusion

We introduced KaMPIng, a set of novel near zero-overhead C++ MPI bindings. Through configurable inference of parameter defaults, fine-grained memory allocation control, enhanced safety guarantees, and a flexible plugin system, it enables both rapid prototyping and careful engineering of distributed algorithms, which we demonstrated using a variety of benchmarks from different application domains.

KaMPIng is open source, extensively tested, and currently used in multiple research projects. It covers MPI's collective and (nonblocking) point-to-point operations, the most commonly used MPI features [2]. In the future, we plan to extend the standard coverage while also working towards our goal of building a basic algorithmic toolbox on top of it, to ease rapid

[8] https://fault-tolerance.org/

prototyping and analysis of distributed algorithms with a strong focus on performance.

In future releases of KaMPIng, we plan to generalize our all-to-all primitives to higher dimensions and integrate mechanism for message aggregation for request-reply patterns when reading from globally distributed data. With distributed containers, we want to enable lightweight bulk parallel computation inspired by MapReduce [40] and Thrill [41], 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.

Acknowledgment

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).

References

[1] MPI Forum, "MPI: A message-passing interface standard – version 4.1," 2023.

[2] I. Laguna *et al.*, "A large-scale study of MPI usage in open-source HPC applications," in *Intl. Conf. for High Performance Computing, Networking, Storage and Analysis*, 11 2019. [Online]. Available: http://dx.doi.org/10.1145/3295500.3356176

[3] M. Ruefenacht *et al.*, "MPI language bindings are holding MPI back," *CoRR*, 2021. [Online]. Available: http://arxiv.org/abs/2107.10566v1

[4] MPI Forum, "MPI: A message-passing interface standard – version 2.2," 2009.

[5] D. Gregor and M. Troyer, *Boost.MPI*, 2005–2007, version 1.84.

[6] A. C. Demiralp *et al.*, "A C++20 Interface for MPI 4.0," 2022, poster at SC'22.

[7] H. Bauke. (2015) MPL - a message passing library. [Online]. Available: https://github.com/rabauke/mpl

[8] S. Ghosh *et al.*, "Towards modern c++ language support for mpi," in *2021 Workshop on Exascale MPI (ExaMPI)*, 11 2021. [Online]. Available: http://dx.doi.org/10.1109/ExaMPI54564.2021.00009

[9] A. Correa. (2018) boost-mpi3. [Online]. Available: https://gitlab.com/correaa/boost-mpi3

[10] A. Polukhin, "Boost.pfr," 2016. [Online]. Available: https://www.boost.org/doc/libs/1_84_0/doc/html/boost_pfr.html

[11] L. D. Dalcin *et al.*, "Parallel distributed computing using python," *Advances in Water Resources*, vol. 34, no. 9, pp. 1124–1139, 2011. [Online]. Available: http://dx.doi.org/10.1016/j.advwatres.2011.04.013

[12] L. Dalcín and Y. L. Fang, "mpi4py: Status update after 12 years of development," *Comput. Sci. Eng.*, vol. 23, no. 4, pp. 47–54, 2021. [Online]. Available: https://doi.org/10.1109/MCSE.2021.3083216

[13] B. Steinbusch, A. Gaspar, and J. Brown, "rsmpi - MPI bindings for rust," 2015. [Online]. Available: https://github.com/rsmpi/rsmpi

[14] H. S. Bjarne Stroustrup *et al.* (2024) C++ core guidelines. [Online]. Available: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines.html

[15] B. Stroustrup, *The design and evolution of C++*. Addison-Wesley, 1994.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, ser. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.

[17] W. S. Grant and R. Voorhies, "cereal – a C++11 library for serialization," 2017. [Online]. Available: http://uscilab.github.io/cereal/

[18] P. Sanders, K. Mehlhorn, M. Dietzfelbinger, and R. Dementiev, *Sequential and Parallel Algorithms and Data Structures*. Springer, 2019.

[19] U. Manber and E. W. Myers, "Suffix arrays: A new method for on-line string searches," *SIAM J. Comput.*, vol. 22, no. 5, pp. 935–948, 1993.

[20] J. Kärkkäinen, P. Sanders, and S. Burkhardt, "Linear work suffix array construction," *J. ACM*, vol. 53, no. 6, pp. 918–936, 2006.

[21] T. Bingmann, "pdcx, https://github.com/bingmann/pDCX," 2018.

[22] J. Fischer and F. Kurpicz, "Lightweight distributed suffix array construction," in *21st Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 2019, pp. 27–38.

[23] T. Bingmann, S. Gog, and F. Kurpicz, "Scalable construction of text indexes with thrill," in *IEEE BigData*. IEEE, 2018, pp. 634–643.

[24] G. M. Slota, S. Rajamanickam, and K. Madduri, "A case study of complex graph analysis in distributed memory: Implementation and optimization," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2016, pp. 293–302.

[25] M. Davis, G. Efstathiou, C. S. Frenk, and S. D. White, "The evolution of large-scale structure in a universe dominated by cold dark matter," *Astrophysical Journal, Part 1 (ISSN 0004-637X), vol. 292, May 15, 1985, p. 371-394. Research supported by the Science and Engineering Research Council of England and NASA.*, vol. 292, pp. 371–394, 1985.

[26] J. Hötzer, M. Jainta, A. Vondrous, J. Ettrich, A. August, D. Stubenvoll, M. Reichardt, M. Selzer, and B. Nestler, "Phase-field simulations of large-scale microstructures by integrated parallel algorithms," in *High Performance Computing in Science and Engineering '14: Transactions of the High Performance Computing Center, Stuttgart (HLRS) 2014*. Springer, 2015, pp. 629–644.

[27] P. Sanders and D. Seemaier, "Distributed deep multilevel graph partitioning," in *Euro-Par 2023: Parallel Processing*. Springer Nature Switzerland, 2023, pp. 443–457.

[28] A. Stamatakis, "RAxML version 8: a tool for

phylogenetic analysis and post-analysis of large phylogenies," *Bioinformatics*, vol. 30, no. 9, pp. 1312–1313, 01 2014. [Online]. Available: https://doi.org/10.1093/bioinformatics/btu033

[29] A. M. Kozlov, D. Darriba, T. Flouri, B. Morel, and A. Stamatakis, "RAxML-NG: a fast, scalable and user-friendly tool for maximum likelihood phylogenetic inference," *Bioinformatics*, vol. 35, no. 21, pp. 4453–4455, 05 2019. [Online]. Available: https://doi.org/10.1093/bioinformatics/btz305

[30] L. V. Kalé, S. Kumar, and K. Varadarajan, "A framework for collective personalized communication," in *17th International Parallel and Distributed Processing Symposium (IPDPS 2003), 22-26 April 2003, Nice, France, CD-ROM/Abstracts Proceedings*. IEEE Computer Society, 2003, p. 69. [Online]. Available: https://doi.org/10.1109/IPDPS.2003.1213166

[31] T. Hoefler, C. Siebert, and A. Lumsdaine, "Scalable communication protocols for dynamic sparse data exchange," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2010, Bangalore, India, January 9-14, 2010*, R. Govindarajan, D. A. Padua, and M. W. Hall, Eds. ACM, 2010, pp. 159–168. [Online]. Available: https://doi.org/10.1145/1693453.1693476

[32] D. Funke, S. Lamm, U. Meyer, M. Penschuck, P. Sanders, C. Schulz, D. Strash, and M. von Looz, "Communication-free massively distributed graph generation," *J. Parallel Distributed Comput.*, vol. 131, pp. 200–217, 2019. [Online]. Available: https://doi.org/10.1016/j.jpdc.2019.03.011

[33] J. Shalf, S. S. Dosanjh, and J. Morrison, "Exascale computing technology challenges," in *High Performance Computing for Computational Science - VECPAR 2010 - 9th International conference, Berkeley, CA, USA, June 22-25, 2010, Revised Selected Papers*, ser. Lecture Notes in Computer Science, J. M. L. M. Palma, M. J. Daydé, O. Marques, and J. C. Lopes, Eds., vol. 6449. Springer, 2010, pp. 1–25. [Online]. Available: https://doi.org/10.1007/978-3-642-19328-6_1

[34] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir, "Toward exascale resilience: 2014 update," *Supercomput. Front. Innov.*, vol. 1, no. 1, pp. 5–28, 2014. [Online]. Available: https://doi.org/10.14529/jsfi140101

[35] J. Dongarra, T. Herault, and Y. Robert1, *Fault-Tolerance Techniques for High-Performance Computing*, ser. Computer Communications and Networks. Springer, Cham, 2015, no. 1.

[36] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. F. Belak, P. Bose, F. Cappello, B. Carlson, A. A. Chien, P. Coteus, N. DeBardeleben, P. C. Diniz, C. Engelmann, M. Erez, S. Fazzari, A. Geist, R. Gupta, F. Johnson, S. Krishnamoorthy, S. Leyffer, D. Liberty, S. Mitra, T. S. Munson, R. Schreiber, J. Stearley, and E. V. Hensbergen, "Addressing failures in exascale computing," *Int. J. High Perform. Comput. Appl.*, vol. 28, no. 2, pp. 129–173, 2014. [Online]. Available: https://doi.org/10.1177/1094342014522573

[37] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra, "Post-failure recovery of mpi communication capability: Design and rationale," *The International Journal of High Performance Computing Applications*, vol. 27, no. 3, pp. 244–254, 2013. [Online]. Available: https://doi.org/10.1177/1094342013488238

[38] C. Stelz, "Core-Count Independent Reproducible Reduce," Bachelor's Thesis, Karlsruhe Institute of Technology, Apr. 2022. [Online]. Available: https://cme.h-its.org/exelixis/pubs/bachelorChristop.pdf

[39] O. Villa, D. Chavarría-Miranda, V. Gurumoorthi, A. Márquez, and S. Krishnamoorthy, "Effects of floating-point non-associativity on numerical computations on massively multithreaded systems," in *Proceedings of Cray User Group Meeting (CUG)*, vol. 3, 2009. [Online]. Available: https://cug.org/5-publications/proceedings_attendee_lists/CUG09CD/S09_Proceedings/pages/authors/01-5Monday/4C-Villa/villa-paper.pdf

[40] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," 2004.

[41] T. Bingmann *et al.*, "Thrill: High-performance algorithmic distributed batch data processing with C++," in *IEEE BigData*, 2016, pp. 172–183. [Online]. Available: https://doi.org/10.1109/BigData.2016.7840603