Towards Modern C++ Language Support for MPI

Sayan Ghosh*, Clara Alsobrooks†, Martin Rüfenacht¶,
Anthony Skjellum†, Purushotham V. Bangalore‡, Andrew Lumsdaine§

* Pacific Northwest National Laboratory sg0@pnnl.gov
† University of Tennessee at Chattanooga WQX336@mocs.utc.edu, Tony-Skjellum@utc.edu
¶ Leibniz Supercomputing Centre martin.ruefenacht@lrz.de
‡ University of Alabama pvbangalore@ua.edu
§ University of Washington al75@uw.edu

***Abstract*—The C++ programming language has made significant strides in improving performance and productivity across a broad spectrum of applications and hardware. The C++ language bindings to MPI had been deleted since MPI 3.0 (circa 2009) because it reportedly added only minimal functionality over the existing C bindings relative to modern C++ practice while incurring significant amount of maintenance to the MPI standard specification. Two years after the MPI C++ interface was eliminated, the ISO C++11 standard was published, which paved the way for modern C++ through numerous improvements to the core language. Since then, there has been continuous enthusiasm among application developers and the MPI Forum for modern C++ bindings to MPI.**

In this paper, we discuss ongoing efforts of the recently formed MPI working group on language bindings in the context of providing modern C++ (C++11 and beyond) support to MPI. Because of the lack of standardized bindings, C++-based MPI applications will often layer their own custom subsets of C++ MPI functionality on top of lower-level C; application- and/or domain-specific abstractions are subsequently layered on this custom subset. From such efforts, it is apparently a challenge to devise a compact set of C++ bindings over MPI with the "right" level of abstractions to support a variety of application uses cases under the expected performance/memory constraints. However, we are convinced that it is possible to identify and eventually standardize a normative set of C++ bindings to MPI that can provide the basic functionality required by distributed-memory applications. To engage with the broader MPI and C++ communities, we discuss a prototypical interface derived from `mpl`, an open-source C++17 message passing library based on MPI.

***Index Terms*—Message Passing, MPI, Communication, Modern C++, C++17, C++20**

I. Introduction

There is an increasing consensus in industry and research organizations on using C++ for high performance code development by targeting a variety of hardware architectures and application domains. On the other hand, MPI is supported on the widest variety of platforms by all major HPC vendors and can be trivially installed via the appropriate package management system on the most common platforms.

While MPI is one of the most popular and widely portable parallel programming models, high-level abstractions of MPI are rare. Apart from a lack of consensus on the design of prospective high-level interfaces over MPI (data structures are artifacts of algorithms/applications, devising general solutions can be challenging), the default C and Fortran language bindings have mostly satisfied the needs of applications running on supercomputers. However, the scientific applications landscape has changed over the last five years. In the past, data movement statements (with no separation between communication and synchronization) were sparsely used in an application at strategic data transfer points and its position remained relatively unchanged between software releases.

Current data-intensive applications often have irregular communication needs. Hence, data movement operations are spread across application codes and may further require robust support for handling heterogeneous data representations. Also, node heterogeneity and software maturity have set the stage for emerging classes of converged applications, which can establish synergy between classic HPC simulations and Machine Learning (ML). Modern data-intensive workloads arising from Machine Learning and Graph Analytics are driving the design of the next generation of extreme-scale architectures. As the use of HPC systems expands to include data-intensive workloads, there is a long-term need for efficient and productive communication interfaces for supporting the data movement needs of the applications. Thus, applications that have stood the test of time require a broad set of capabilities for achieving sustainable performance and productivity on current and future extreme-scale architectures.

As hardware platforms become more heterogeneous and complex, C++ is becoming the programming model of choice for modern science applications. As such, the performance-critical components of popular ML platforms and current performance portability models (e.g., SYCL/DPC++ [1], KoKKoS [7] and RAJA [3]) are built using modern ISO standardized C++. We are convinced that standardizing C++ language bindings for MPI would encourage the users to rapidly prototype new application communication scenarios and provide the necessary flexibility to enhance existing codes to distributed-memory platforms in a convenient fashion.

To drive the discussion in favor of high-level language support for MPI, consider a simple halo exchange operation on a 1D stencil using the MPI C bindings as shown in Listing 1.

2021 Workshop on Exascale MPI (ExaMPI) | 978-1-6654-1108-0/21/$31.00 ©2021 IEEE | DOI: 10.1109/EXAMPI54564.2021.00009

978-1-6654-1108-0/21/$31.00 ©2021 IEEE
DOI 10.1109/ExaMPI54564.2021.00009

```
MPI_Request* reqs = new MPI_Request[size*2];
if (rank != 0) {
  MPI_Irecv(&x(0, 0), x.num_y(), MPI_DOUBLE, rank - 1, 321,
     MPI_COMM_WORLD, &reqs[c]);
  MPI_Isend(&x(1, 0), x.num_y(), MPI_DOUBLE, rank - 1, 322,
     MPI_COMM_WORLD, &reqs[c++]);
}
if (rank != size-1) {
  MPI_Irecv(&x(x.num_x()-1, 0), x.num_y(), MPI_DOUBLE,
     rank + 1, 322, MPI_COMM_WORLD, &reqs[c]);
  MPI_Isend(&x(x.num_x()-2, 0), x.num_y(), MPI_DOUBLE,
     rank + 1, 321, MPI_COMM_WORLD, &reqs[c++]);
}
MPI_Waitall(c, reqs);
```

Listing 1 Halo exchange using MPI C bindings

Using the previously standardized MPI C++ bindings (since deleted from the standard), the same halo exchange operation can be developed as shown in Listing 2.

```
std::vector<MPI::Request> reqs;
if (rank != 0) {
  reqs.emplace_back(MPI::COMM_WORLD.Irecv(const_cast<double
       *>(&x(0, 0)), x.num_y(), MPI::DOUBLE, rank - 1, 321)
       );
  reqs.emplace_back(MPI::COMM_WORLD.Isend(&x(1, 0), x.num_y
       (), MPI::DOUBLE, rank - 1, 322));
}
if (rank != size-1) {
  reqs.emplace_back(MPI::COMM_WORLD.Irecv(const_cast<double
       *>(&x(x.num_x()-1, 0)), x.num_y(), MPI::DOUBLE, rank
        + 1, 322));
  reqs.emplace_back(MPI::COMM_WORLD.Isend(&x(x.num_x()-2,
       0), x.num_y(), MPI::DOUBLE, rank + 1, 321));
}
MPI::Request::Waitall(reqs.size(), reqs.data());
```

Listing 2 Halo exchange using deleted MPI C++ bindings

While the previous C++ interface can benefit from containers and idioms such as RAII (initialization/cleanup of resources), there are obvious limitations affecting the productivity:

- Lack of modern C++ idioms: generic container interface, operator overloading, iterators/ranges, type traits based conditional compilation, etc.
- Separation of MPI and primitive types, missing automatic type mapping or parameter deduction
- Extensive use of free functions (in `MPI` namespace), as opposed to objects

The MPI C++ interface was officially deleted from the MPI-3 standard (September 2012), and the ISO C++11 standard was published in September 2011. C++11 paved the way for modern C++ through numerous improvements to the core language [15]. Since then, there has been continuous enthusiasm among application developers and the MPI forum for modern C++ bindings to MPI (e.g., Boost.MPI from Indiana University, `mpl` et al.). However, the strength of MPI lies in its ability to address a variety of application communication scenarios, through flexible data type creation and communication models. Existing libraries that provide a set of C++ bindings over MPI primarily differ in the following aspects: 1) Incomplete set of bindings (i.e., offering C++ bindings to only a subset of MPI functions), 2) Handling of derived datatypes (automatic serialization like Boost.MPI, or separate datatype constructors), and, 3) Modern C++ usage.

Using a modern C++ messaging interface such as `mpl`, the halo exchange example can be rewritten as shown in Listing 3.

```
mpl::irequest_pool reqs;
mpl::tag atag = mpl::tag(321), btag = mpl::tag(322);
mpl::contiguous_layout<double> l(x.num_y());
if (rank != 0) {
   reqs.push(comm_world.irecv(&x(0,0),l,rank-1,atag));
   reqs.push(comm_world.isend(&x(1,0),l,rank-1,btag));
}
if (rank != size-1) {
  reqs.push(comm_world.irecv(&x(x.num_x()-1,0),l,rank+1,
       btag));
  reqs.push(comm_world.isend(&x(x.num_x()-2,0),l,rank+1,
       atag));
}
reqs.waitall();
```

Listing 3 Halo exchange using `mpl` C++17 messaging library

Although the respective halo exchange implementations are exactly similar in terms of semantics, the latter implementation is more concise and expressive, in the idioms of the C++ programming language.

The rest of the paper is organized as follows. We briefly explore relevant high level interfaces to MPI in Section II. In Section III, we discuss the existing `mpl` C++17 messaging library, and introduce a compact subset of the `mpl` library for studying the impact of accessing basic MPI functionality using modern C++. Section IV compares the results of the `mpl` subset with MPI through microbenchmark and mini-application evaluations. Section V reviews relevant features in the C++20 standard. New features of the upcoming MPI standards and the potential for C++ support are discussed in Section VI. Finally, Section VII concludes the paper.

II. Existing Interfaces to MPI

C++ bindings for MPI began while MPI-1 was still being ratified [2], [14], [21], [28], [29] and continued until the official introduction of the C++ bindings in MPI-2. After MPI-3 discarded the interface, work evidently resumed. Prior and on-going efforts are noted that sought to provide third-party MPI language interfaces including Java [5], C++ (e.g., [2], [21]), Spark [24], Python [6], and .Net [9].

A. MPJ: MPI-like Message Passing for Java

In Carpenter et al. [5], the MPJ notation for Java was introduced, providing a means to program Java with MPI-1-type operations. This project has been inactive since 2000.

B. Boost MPI

Boost MPI is a historical, header-only library on top of MPI-2 that leverages the Boost library [10]. The goal was evidently to provide a higher-level abstraction like C++ STL. Boost MPI has not been widely used by major MPI applications due to its complex dependencies and use of serialization to handle user defined datatypes. Recent minor updates in 2019 follow minor updates in 2013, and end of active development in 2008.

C. MPP

MPP [25] is a header-only C++ MPI interface that uses some of the object oriented programming features of C++ such as generic programming, type traits, futures, and also

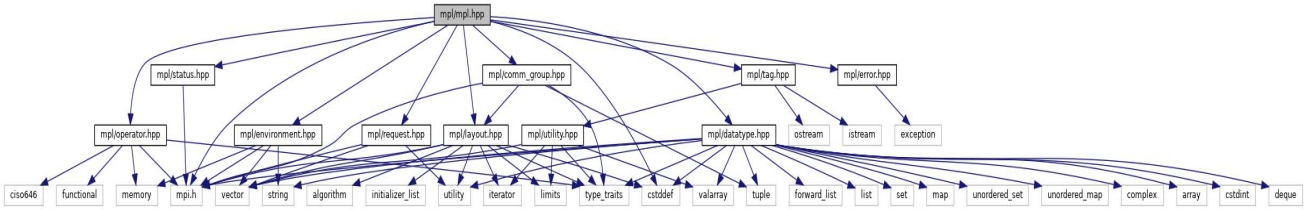


Fig. 1 Dependencies of our prototypical `mpl` subset

supports the use of user-defined datatypes. Initial performance evaluation indicated better performance when compared with Boost MPI. This interface has not been updated since 2013.

D. Ad hoc C++ APIs

A large-scale study of MPI usage in open-source HPC applications [17] found that C++ was the most widely used language among open-source HPC applications. The US DOE ECP software technology capability assessment report also claims high criticality and importance of C++ and MPI [12]. While a prospective C++ interface can bypass the C bindings to MPI and directly invoke the lower-level communication substrate in an implementation, the MPI tools interface (i.e., PMPI [27]/QMPI [8]) currently exposes a C interface. Therefore, compliant MPI implementations require C bindings. A C++ API for MPI needs to invent an efficient workaround for this issue, but that is beyond the scope of this communication.

It appears that layering on top of the C interface is done commonly in C++-based MPI applications, particularly after MPI-3 deprecated the old C++ interface. The applications Comb [4] and HemeLB [20] are two particular examples of this trend.

III. A review of MPL, a new starting point

In this section, we discuss the features of an existing messaging library developed using modern C++, namely `mpl` [11]. Since `mpl` provides a broad set of communication abstractions, we attempt to identify a compact subset of `mpl` to study the software overheads and design issues.

A. As noted above, the `mpl` Messaging Library

The `mpl` library [11] is a C++-17 based, header-only library meant to provide an easy to use MPI interface for C++ developers after the deprecation and removal of the C++ API in MPI 3.1. Its key features are as follows:

- use of `comm` and `group` objects with MPI APIs refactored into member functions (like in the MPI-2 C++ interface)
- reduced argument sets as compared to the C interface, including polymorphic variations with functions like gatherv (e.g., non-root processes specify less arguments)
- return of values and request objects in non-blocking operations, not error codes like C.

The `mpl` library does not support one-sided, dynamic process management, or I/O. Exceptions are thrown for various detected errors but there is no formal connection of the five exceptions it can throw to all the MPI error codes and classes.

Nonblocking operations in `mpl` returns an `mpl::irequest` type object, which wraps an `MPI_Request`. For handling multiple requests, the `mpl::request_pool` class manage `MPI_Request` and `MPI_Status` type containers and the associated routines. To prevent (accidental) copying of request/status objects, the default copy constructors are thwarted.

The `mpl` library does not rely on predefined reduction operation handles (such as `MPI_SUM`, `MPI_PROD`, etc.), and builds a user-defined reduction operation (using `MPI_Op_create`) from C++ lambda expressions or functors. This may lead to overheads if a reduction routine is invoked repeatedly, or it could serve as a vehicle for optimization.

Currently, `mpl` does not support one-sided communication via MPI RMA. However, it is relatively straightforward to develop a parameterized `mpl::window` class and use `mpl::layout` for representing the data. It can be idiomatic to accept C++ `std::memory_order` semantics and translate to the appropriate `MPI_Info` hints for one-sided remote atomic operations.

B. A Prototype `mpl` Subset

We have extracted communication-specific interfaces from the main `mpl` codebase [11] (about 6K LoC, compared to about 10K LoC in main `mpl`), and consider it as a prototypical C++ interface to study MPI-specific language bindings.

The `mpl` library also includes abstractions to aid scientific programming use cases (such as distributed process grid/graph interfaces), which is not part of our prototypical `mpl` interface. Our primary goal was to minimize this library by eliminating the features (and associated files) that are unrelated to base MPI functionality (at present, we are considering point-to-point/collective communication, communication completion, derived data types, and communicator management interfaces). Our next goal is to employ the compact library in studying the applicability of recent C++20 features. Dependencies and the main modules of our prototypical `mpl` interface is shown in Fig. 1.

Our prototypical `mpl` interface eliminates 10 files from the larger `mpl` library. These files contain interfaces related to virtual topologies: `cart_comm`, `dist_graph_comm`, `graph_comm`, `topo_comm`, and `distributed_grid`, as well as the additional interfaces that introduce abstract data types: `displacements`, `flat_memory`, `message`, `ranks`, and a custom `vector` class. In eliminating these files (about 3K LoC), we have minimized `mpl` to a compact

set of bindings that supports point-to-point and collective communication operations. The following interfaces comprise our prototypical `mpl` implementation: comm_group, datatype, environment, error, layout, operator, request, status, tag, utility, and the encompassing header mpl. We also excised additional code from these interfaces. The largest of these files is `comm_group`, which contains the communication classes and overloads, as well as interfaces for communicator management. Table I summarizes the components of the abridged `mpl` implementation.

TABLE I Interfaces and functionality of our prototypical `mpl` subset.

MPL interface	Functionality
Comm group	(Nonblocking) Send/Recv and Collectives, Communicator management
Datatype	Datatype mapping, construction, (de) serialization
Environment	Initialization, thread support, global buffer management (for `MPI_Bsend`)
Error	Default exception class
Layout	Derived datatype constructor
Operator	Logical operators, specialization, `MPI_Op` constructor
Request	Communication initiation/completion, Request object handling (includes request pool management)
Status	(Receiver) Status object handling
Tag	Managing tag, comparison, initialization, etc.
Utility	Helper routines, i.e., checks for type narrowing, integral constants, contiguous type, etc.

The `mpl` library lacks routines for bulk communication scenarios, and relies on the `mpl::layout` constructor to internally manage derives data types. Hence, the count parameter passed to the point-to-point and collective routines is 1. There is also an `mpl::layouts` constructor, to deal with many-to-many collective operations with multiple counts/displacements (for e.g., `alltoallv`). Another option is to use the interface in `mpl` that manages communication states, using generalized requests instead of standard MPI request objects. A relevant example routine (from main `mpl`) is shown in Listing 4.

```
template<typename T>
void isend(const T &data, int dest, tag_t t,
     isend_irecv_state *isend_state, detail::
     contiguous_const_stl_container) const {
  using value_type = typename T::value_type;
  const int count(data.size());
  MPI_Datatype datatype{detail::datatype_traits<
       value_type>::get_datatype()};
  MPI_Request req;
  MPI_Isend(data.size() > 0 ? &data[0] : nullptr, count,
       datatype, dest,static_cast<int>(t), comm_, &req);
  MPI_Status s;
  MPI_Wait(&req, &s);
  isend_state->source = s.MPI_SOURCE;
  isend_state->tag = s.MPI_TAG;
  isend_state->datatype = datatype;
  isend_state->count = 0;
  MPI_Grequest_complete(isend_state->req);
}
```

Listing 4 Bulk communication interface in `mpl` that uses generalized requests

Generalized requests [18] in MPI can be used to define new nonblocking operations. Generalized requests require that the communication progress must occur outside of the context of MPI. We are convinced that using generalized request model to wrap existing nonblocking bulk point-to-point or collective operations is excessive, and the relatively strict progress model can lead to unnecessary overheads. The nonblocking API in the example offers no overlap of computation and communication, which is one of the basic tenets of nonblocking communication. Related routines in `mpl` also spawns a C++ `std::thread` to make progress on a generalized request enabled nonblocking routine, and further allocates a communication state per message on the heap. Hence, we have removed functions that rely on the communication state/generalized request from our version.

IV. Evaluations

We transformed the OSU microbenchmark [19] (based on v5.3.1) and replaced MPI routines with the corresponding `mpl` counterparts for a few commonly used benchmarks. We also updated the LULESH proxy application [16] and replaced MPI with `mpl` to compare real application scenarios.

Experimental platform: The experiments were performed on the NERSC Cori supercomputer (Haswell partition), using cray-mpich/7.7.10 and PrgEnv-intel/6.0.5 (Intel compiler v19.0.3). NERSC Cori is a Cray XC40 system with 16-core Intel Haswell Xeon E5-2698v3 node at 2.3GHz (32 cores/node, two-way), 128GB memory, 40MB L3 cache, and Cray Aries interconnect with the Dragonfly topology.

A. OSU microbenchmark

Results are shown in Figures 2 and 3; `mpl` demonstrates near-MPI performance on single and multiple nodes. Thus, the C++ interface does not add significant overheads to the critical path. We also discuss a few cases where the opposite is true.

The original version of `mpl` from which we derived our work used MPI generalized requests to track nonblocking bulk communication scenarios (where count is greater than one) and collective operations. We are sure that comparison of `mpl` directly with MPI in the OSU microbenchmark is reasonable, despite default derived data type usage in `mpl`. In our prototypical `mpl` interface, we have removed routines that use communication state and generalized requests (refer to Section III).

Currently, in main `mpl`, the way to communicate multiple data points is through derived types. The `mpl::layout` class abstracts the MPI derived datatype management for contiguous and strided data. The abridged code snippet from the updated OSU multi-latency microbenchmark demonstrates `mpl` usage next to MPI in Listing 5. Since `mpl` is built directly on top of MPI (and not a different communication runtime), MPI and `mpl` functions are interoperable.

```
#if defined(USE_MPL_CXX)
   const mpl::communicator &comm_world(mpl::environment::
        comm_world());
   rank = comm_world.rank();
#else
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
#endif
   for(size = 0; size <= MAX_MSG_SIZE; size *= 2) {
#if defined(USE_MPL_CXX)
      mpl::contiguous_layout<char> lo(size);
      mpl::tag tag = mpl::tag();
#endif
      if (rank < pairs) {
```

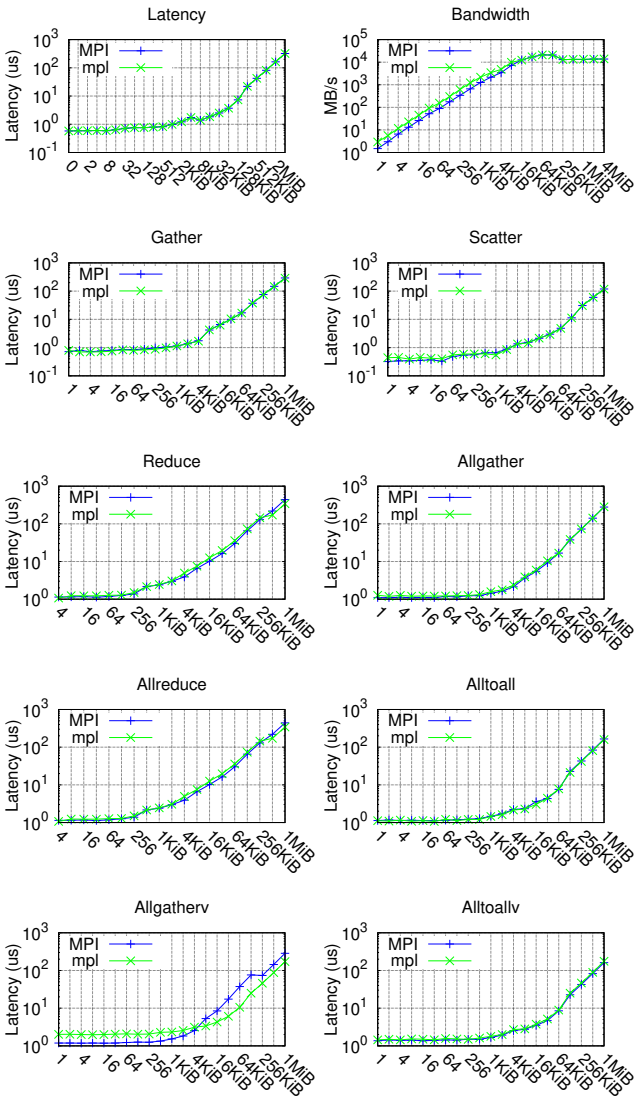


Fig. 2 MPI vs `mpl` on 2 PEs within a node. *X-axis*: Bytes transferred.

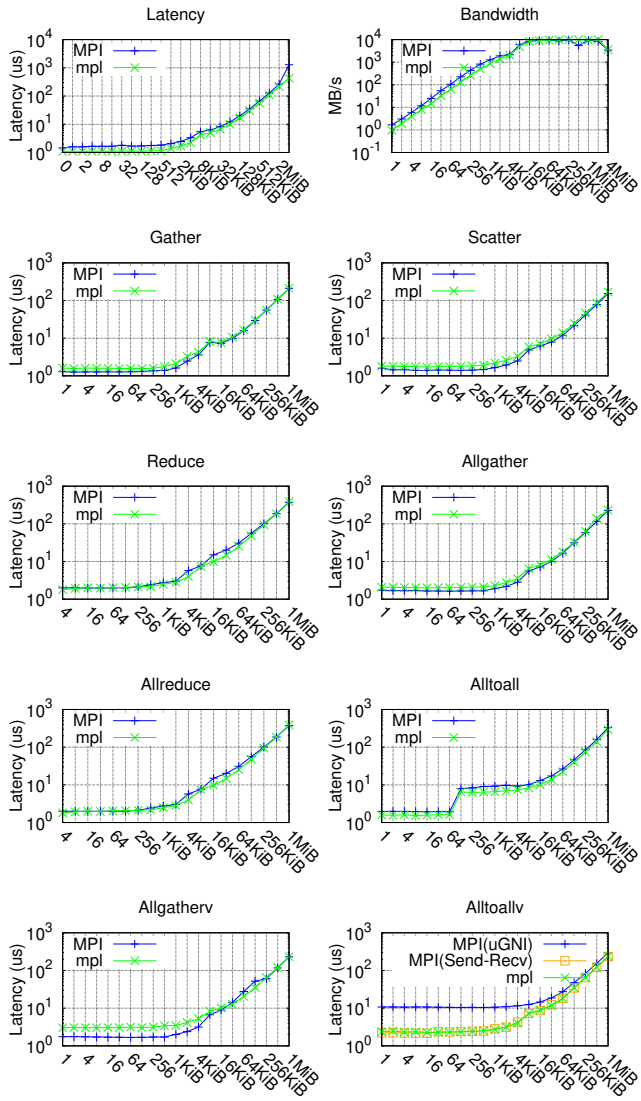


Fig. 3 MPI vs `mpl` on 2 PEs across nodes. *X-axis*: Bytes transferred.

```
        partner = rank + pairs;
#if defined(USE_MPL_CXX)
        comm_world.send(s_buf, lo, partner, tag);
        comm_world.recv(r_buf, lo, partner, tag);
#else
        MPI_Send(s_buf, size, MPI_CHAR, partner, 1,
            MPI_COMM_WORLD);
        MPI_Recv(r_buf, size, MPI_CHAR, partner, 1,
            MPI_COMM_WORLD, &reqstat);
#endif
      }
    }
```

Listing 5 OSU multi-latency case using MPI and `mpl`

For small messages on two nodes, we observed a significant difference between MPI and `mpl` for the `alltoallv` case (Fig. 3). The `mpl` library uses `MPI_Alltoallw` instead of `MPI_Alltoallv` for its `alltoallv` implementations. On the Cray XC40 platform, uGNI optimized algorithms are used for selected MPI collectives (includes `MPI_Alltoallv`) by default. Switching back to a Send-Recv-based all-to-all algorithm (using `export MPICH_GNI_COLL_OPT_OFF=mpi_alltoallv`), we again observe similar performance between MPI and `mpl`. Thus, the performance of `mpl` depends on the quality and configurations of the platform MPI implementation.

Fig. 4 demonstrates benchmark results on 16 nodes (using 32 PEs/node). Once again, we observe similar performance of MPI and `mpl` for the message rate evaluation. For `alltoallv` evaluations on 512 PEs, `mpl` exhibits up to 2.5× more latency compared to `MPI_Alltoallv` (default case, uGNI optimized all-to-all), primarily arising from overheads associated with copying data from `mpl::layouts` instance to integer buffers. The `allgatherv` implementation of `mpl` uses `MPI_Alltoallv` internally instead of `MPI_Allgatherv`, which is the reason behind the performance divergence between MPI and `mpl`. Since the

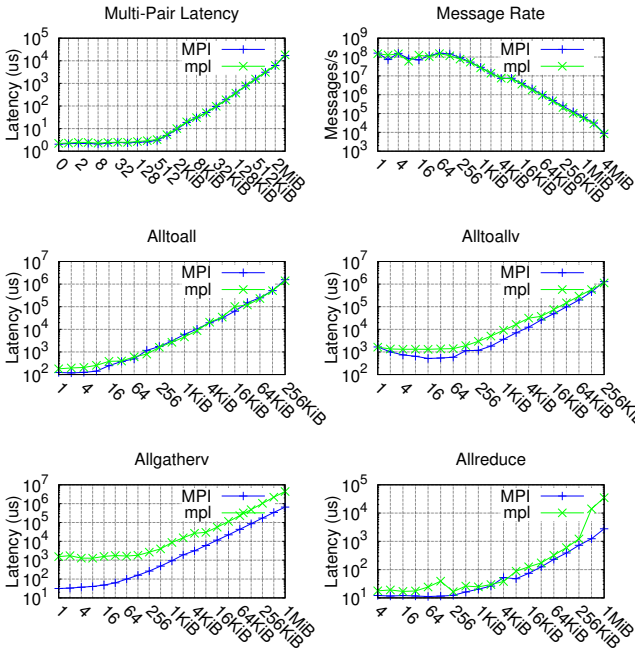


Fig. 4 MPI vs `mpl` on 512 PEs. *X-axis*: Bytes transferred.

`allreduce` implementation of `mpl` requires `MPI_Op` creation (refer to Section III), there are associated overheads. We are convinced that persistent collective operations [23] in the recently approved MPI-4 specification can mitigate such overheads stemming from repeated invocations.

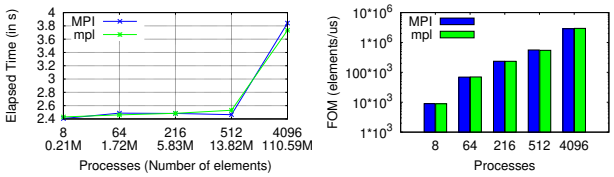
B. LULESH proxy application

The Livermore Unstructured Lagrange Explicit Shock Hydrodynamics (LULESH) [16] proxy application is from the area of shock hydrodynamics. The original version of LULESH uses MPI two-sided nonblocking APIs. The weak scaling results on 8–4096 processes (1–128 nodes) are shown in Fig. 5; the problem size of an MPI task is $30^3 = 27,000$.

On each iteration of LULESH, a process communicates with 26 of its neighbors in a 3D domain for 100 iterations. The "Figure of Merit" (FOM) metric of LULESH is expressed in terms of the elements solved per microsecond.

Fig. 5a and Fig. 5b compares the reported execution time and FOM of the native MPI two-sided and `mpl` implementations of LULESH v2.0. Overall, we observe up to 3% performance difference between `mpl` and MPI over successive runs on greater than 512 processes.

The code snippet in Listing 6 attempts to capture the *receive* operation of a communication round, comparing the original MPI version with the updated `mpl` one. Unlike MPI, `mpl` requires derived types for bulk communication (i.e., `mpl::contiguous_layout`), as discussed in Section III-B. In case of the `mpl` version in Listing 6, `rpool` is a `std::vector` of `mpl::irequest`, and the `wait` function (synonymous with `MPI_Wait`) is later invoked on the individual requests for completion.



(a) Elapsed time (in secs.) (b) Figure of Merit (Elements/us)

Fig. 5 MPI vs `mpl` versions of LULESH proxy application.

```
#if defined(USE_MPL_CXX)
mpl::contiguous_layout<Real_t> lxy(dx*dy*xferFields);
mpl::tag rtag = mpl::tag(msgType);
const mpl::communicator &comm_world = mpl::environment::
    comm_world();
...
rpool.push_back(comm_world.irecv(&domain.commDataRecv[pmsg *
     maxPlaneComm], lxy, fromRank, rtag));
#else
MPI_Irecv(&domain.commDataRecv[pmsg * maxPlaneComm],
    recvCount, baseType, fromRank, msgType, MPI_COMM_WORLD,
     &domain.recvRequest[pmsg]) ;
...
#endif
```

Listing 6 A nonblocking *receive* operation in LULESH

On the sender side, the `mpl` version of LULESH uses `mpl::irequest_pool` (refer to Section III-A) to manage multiple request objects associated with the *send* operations, as shown in Listing 7. A `waitall` (similar to `MPI_Waitall`) is later invoked on the `mpl::irequest_pool` object when all the nonblocking sends have been issued.

```
#if defined(USE_MPL_CXX)
mpl::contiguous_layout<Real_t> lxy(xferFields*dx*dy);
mpl::irequest_pool spool;
mpl::tag stag = mpl::tag(msgType);
const mpl::communicator &comm_world = mpl::environment::
    comm_world();
...
spool.push(comm_world.isend(destAddr, lxy, myRank-domain.tp
    ()*domain.tp(), stag));
#else
MPI_Isend(destAddr, xferFields*sendCount, baseType, myRank-
    domain.tp()*domain.tp(), msgType, MPI_COMM_WORLD, &
    domain.sendRequest[pmsg]) ;
...
#endif
```

Listing 7 A nonblocking *send* operation in LULESH

The `rpool` vector of `mpl::irequest` associated with the *receives* and the `mpl::irequest_pool` for the *sends* are maintained by C++, which safely destroys the objects when they go out of scope (owing to the Resource Acquisition Is Initialization, i.e., RAII idiom).

V. New features of C++20

We are convinced that recent C++20 features such as `std::concepts` and `std::ranges` can further augment a C++ interface such as `mpl` [22]. Concepts are constraints on template parameters that are evaluated at compile time, whereas C++ ranges provide interfaces to abstract a collection of items, generalizing and extending iterators. For instance, a way to specify acceptable C++ containers as the source buffer for bulk communication scenarios can be expressed through a simple concept as shown in Listing 8.

```
template <typename C>
concept Contig = requires(C c) {
  { c.data() };
} && std::ranges::contiguous_range<C>;
```

Listing 8 Sample C++20 concept

This concept mandates that an input C++ container have the `data()` member that returns the base pointer, and the `std::ranges::contiguous_range` guarantees that the underlying elements are stored contiguously in memory. Parameter substitution errors in C++ codes can lead to lengthy error statements, that are difficult to diagnose. For example, the messages in Listing 9 are part of a huge report emitted by the C++ compiler for a datatype mismatch.

```
   note: template argument deduction/substitution failed:
        mismatched types '_Tp [_Nm]' and 'float*'
   error: request for member 'begin' in '__cont', which is
        of non-class type 'float*'
   note: mismatched types 'std::valarray<_Tp>' and 'float*'
   ...
```

Listing 9 Typical C++ error message for a simple datatype mismatch

Concepts can be employed to alleviate such potential issues, providing constructive error messages when constraints are not satisfied. Trying to pass an `std::list` (non-contiguous container) when the above concept is enabled, the error message is concise, as shown in Listing 10.

```
error: 'class std::__cxx11::list<float>' has no member named
     'data'
```

Listing 10 Concise error message when C++20 concept is enabled

Other built-in concepts in C++20 can efficiently establish various object type requirements at compile-time.

Ranges offer more flexibility than iterators on C++ containers, such as allowing convenient options to chain data transformations and adaptors to enable them. Prospective C++ bindings to MPI are expected to improve the overall programmability and composability, and allowing ranges with the appropriate constraints specified through concepts is a step in that direction. Listing 11 demonstrates use of the previously defined `Contig` concept (from Listing 8).

```
template <Contig C>
void send(C&& x, int dest) {
  using T = typename C::value_type;
  MPI_Send(x.data(), size(x), TypeMap<T>(), dest, 101,
       MPI_COMM_WORLD);
}

template <Contig C>
void recv(C&& x, int source) {
  using T = typename C::value_type;
  MPI_Recv(x.data(), size(x), TypeMap<T>(), source,
       MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
std::vector<float> pb(100), pbr(100);
std::iota(begin(pb), end(pb));

if (rank%2 != 0) {
   auto v = pb | std::views::transform([](float i){ return i
        *2; });
   std::copy(begin(v), end(v), begin(pbr));
   send<float>(pbr, rank-1);
}
else
   recv<float>(pbr, rank+1);
```

Listing 11 C++20 ranges and concepts in an MPI program

In Listing 11, the sender applies a transformation to `pb` prior to sending the data; however, the original data in the vector is unchanged (`v` does not store the updated value, accesses to it just apply the transformation). Specifically, `v` is a view that does not satisfy the `Contig` concept. Hence, an explicit copy is made from the range object to a contiguous container, prior to communication. (Though one could imagine overloaded communication operations that could accommodate views via explicit buffering.)

C++17 introduces execution policy parameters (`std::execution::par`, `std::execution::seq`, etc.) that can specify the intended level of parallelism while using algorithms from the standard library. Since prospective MPI C++ language bindings might extensively use STL (e.g., for operations such as reduction, transformation, scan, sort, etc.), an optional argument for execution policy can allow implementations to enable fine-grain concurrency control.

VI. New features of MPI-4 and prospective MPI-5 features

The MPI standard has recently advanced to MPI-4, with the June, 2021 approval of the latest edition of the standard. With this version of the standard, there are features that also need to be supported by a C++ interface, and for which C++ offers potential for greater performance. Key new features include:

- 64-bit API (aka "big MPI" or "big count") replaces integer based *count* parameter with the new `MPI_Count` datatype (as large as `MPI_Aint`/`MPI_Offset` types used to encode address locations). The new APIs in the MPI C bindings are distinguished with an "_c" suffix.
- Sessions, a means for multi-session MPI program allocations that expand on the "world model." There is no notion of Sessions in the original mpl C++ API.
- Persistent collective communication; persistence with collective operations offers a new opportunity for a factory-model mode for mpl-based interfaces. The initiation is blocking in the current API.
- Partitioned point-to-point communication. This offers channels with pipelining or concurrent producers and consumers, representing an MPI+X extension model.

It is possible to use C++ templates to avoid the duplication of 110+ APIs (affected by the MPI-4 "big count" update) in the interface through sensible polymorphism on types, with a few possible exceptions. Also, coping with certain collective operations where some processes in a communicator would have smaller data sets while others are "big count" deserves further careful consideration.

However, of these new APIs, partitioned point-to-point communication appears to offer the greatest potential for innovation with a C++ language interface. For instance, on both the sender and receiver, the concept of transfer has a partitioned buffer object now. That is a new kind of buffer object. There is fine-grain notification of the completion of partitions on the send side, and fine-grain ability to poll on arrival of partitions on the receive side; send and receive-side partitioning can be different, depending on the application use

case. Each partition is a single-producer, single consumer sub-buffer. This type of container suggests a modern C++ object be created with the 4-tuple: address, datatype, count, and partition count. This object, a partitioned-buffer, has specialization for its use as a send-side or receive-side object. Appropriate methods for indicating completion or availability, based on the C API's `MPI_Pready` (plus variants) and `MPI_Parrived` comprise logical member functions. The concurrency of the buffer object differs from STL containers with iterators. These objects are inherently random-access, concurrent structured partitions that comprise the original buffer.

As a further goal, either a contract-based object model for equivalent, replaceable classes, or else set of alternative implementations with classic polymorphism are needed to support onload and offload implementation, with offload notably to GPUs. Capturing the offload concept at the C++ layer could help support efficient implementation at runtime for MPI+CUDA, for example[1]. Further, special memory spaces, analogous to what KoKKoS [7] provides, might be a useful nuance for such partitioned buffers, or they could literally be realized by the user with aid from a library such as KoKKoS and passed to a constructor in MPI for wrapping in an MPI-specific lightweight object. Persistent collective operations offer an interesting opportunity for this C++ language interface too, because there are info-related restrictions planned for MPI-4.1 with regard to total ordering in the group of the communicator, vs.the default mode in which instantiation of a given operation instance can be done outside the total order of collective operation instantiations for a given communicator. Also, given the persistent collectives are designed to do planned transfer and algorithmic selection, compile-time, runtime-based, and profile-guided approaches to instantiating instances of operations could be aided by a modern C++ design.

Further, we expect to realize the addition of partitioned collective operations in MPI-5, with high probability. These introduce both of the aforementioned new features of MPI-4 in combination. Opportunities for building new kinds of data structures and descriptors abound, as do opportunities for describing some assertions at compile-time via the C++ API.

Lastly, the conflation of sessions, hardware topologies, and collective operations direct from groups is a possible addition to MPI-5 (see the recent work in this area by Herschberg et al. [13]). This type of operational object would most closely resemble a C handle for a persistent request in MPI. However, this type of construct would be highly focused on integrating three currently disparate goals of MPI. The single operational method would be unlike regular communicators with many methods representing all the operations of which MPI is capable, and explicitly pairing algorithmic selection with hardware topology. Such objects subtend specific resources, and extrapolating from this feature could be a means for alternating the use of fixed resources between such operations, providing primitive notions of QoS. The complexity of managing such operations and their underlying state transitions between MPI and the application will undoubtedly be helped by reference counting and other first-class language support unavailable in C and baseline Fortran interfaces.

[1]A pair of potential data structure concepts that C++ could help manage in connection with MPI are CUDA streams and graphs. They are sufficiently complex to manage yet encapsulating them in C++, without compromising performance, would potentially lead to better performance portability when other accelerators offer their analogous ways of describing sequences of communication and computation in DAGs. In some sense, a completely new work enqueueing and notification mechanism is really needed between MPI and accelerators, yet the particular implementation will vary from accelerator to accelerator.

It may well be that the streams and graphs are describing such specialized operations in a given accelerator that there is little opportunity for building abstracted, encapsulating performance-portable interfaces, but it appears worth studying further.

VII. Conclusion

Modern data intensive workloads arising from Data Sciences, Machine Learning and Graph Analytics are driving the design of the next generation of extreme-scale architectures. As the use of HPC systems expands to include data-intensive workloads, there is a long-term need for efficient and productive communication interfaces for supporting the data movement needs of the applications.

We discussed our ongoing efforts in devising modern C++ abstractions over MPI, using a subset of the `mpl` C++17 messaging library to drive the discussion. The code is publicly available in: https://github.com/mpi-advance/mpl-subset.git.

Going forward, we will use the `mpl` subset implementation to design and analyze high-level communication abstractions using modern C++. A large number of data-intensive applications often need features to support communication scenarios that would benefit from a one-sided communication model, which separates communication from synchronization. Hence, we plan on expanding the current `mpl` subset with one-sided communication functionality, which is currently missing from `mpl`.

We are convinced that this is an opportune moment to revive the effort in standardizing C++ language bindings to MPI, acknowledging the previous attempt as a learning experience. Along the way, we plan on actively engaging with the application and middleware developers, and encourage community participation.

Acknowledgements

The authors kindly acknowledge the contributions of Heiko Bauke, the lead developer/author of `mpl`, which underpinned this project and paper.

This research was supported by the Data-Model Convergence (DMC) initiative at Pacific Northwest National Laboratory. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE-AC02-05CH11231.

This work was performed with partial support from the National Science Foundation under Grants Nos. CCF-1562306, CCF-1822191, CCF-1821431, OAC-1925603, and

the U.S.Department of Energy's National Nuclear Security Administration (NNSA) under the Predictive Science Academic Alliance Program (PSAAP-III), Award DE-NA0003966.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation, the U.S. Department of Energy's National Nuclear Security Administration, or the Pacific Northwest National Laboratory.

REFERENCES

[1] 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*, pages 1–2, 2020.

[2] Purushotham V. Bangalore, Nathan E. Doss, and Anthony Skjellum. MPI++: Issues and features. In *In Proceedings of Object-Oriented Numerics Conference (OONSCI'94)*, pages 323–338, April 1994.

[3] 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)*, pages 71–81. IEEE, 2019.

[4] Jason Burmark and USDOE National Nuclear Security Administration. Comb, 7 2018.

[5] Bryan Carpenter, Vladimir Getov, Glenn Judd, Anthony Skjellum, and Geoffrey C. Fox. MPJ: MPI-like message passing for java. *Concurr. Pract. Exp.*, 12(11):1019–1038, 2000.

[6] Lisandro D. Dalcin, Rodrigo R. Paz, Pablo A. Kler, and Alejandro Cosimo. Parallel distributed computing using Python. *Advances in Water Resources*, 34(9):1124–1139, 2011. New Computational Methods and Software Tools.

[7] H Carter Edwards, Christian R Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of parallel and distributed computing*, 74(12):3202–3216, 2014.

[8] Bengisu Elis, Dai Yang, Olga Pearce, Kathryn Mohror, and Martin Schulz. QMPI: A next generation MPI profiling interface for modern HPC platforms. *Parallel Computing*, 96:102635, 2020.

[9] Douglas Gregor and Andrew Lumsdaine. Design and Implementation of a High-Performance MPI for C# and the Common Language Infrastructure. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, page 133–142, New York, NY, USA, 2008. Association for Computing Machinery.

[10] Douglas Gregor and Matthias Troyer. Boost. MPI, version 1.76.0, 2017.

[11] Heiko Bauke. MPL - A message passing library. https://github.com/rabauke/mpl.

[12] Michael A Heroux, Rajeev Thakur, Lois McInnes, Jeffrey S Vetter, Xiaoye Sherry Li, James Aherns, Todd Munson, and Kathryn Mohror. ECP software technology capability assessment report. Technical report, Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States), 2020.

[13] Tom Herschberg, Derek Schafer, and Anthony Skjellum. integrating MPI sessions with topological connection building and collective communication.

[14] Steve Huss-Lederman, Bill Gropp, Anthony Skjellum, Andrew Lumsdaine, Bill Saphir, Jeff Squyres, et al. MPI-2: Extensions to the Message Passing Interface. *University of Tennessee, available online at http://www.mpiforum.org/docs/docs. html*, 1997.

[15] Danny Kalev and Bjarne Stroustrup. The biggest changes in c++11 (and why you should care). *Smartbear.[cit. 6. 5. 2016]. Dostupné na:¡ http://blog. smartbear. com/c-plus-plus/the-biggestchanges-in-c11-and-why-you-should-care*, 2011.

[16] 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*, pages 919–932. IEEE, 2013.

[17] Ignacio Laguna, Ryan Marshall, Kathryn Mohror, Martin Ruefenacht, Anthony Skjellum, and Nawrin Sultana. A Large-Scale Study of MPI Usage in Open-Source HPC Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[18] Robert Latham, William Gropp, Robert Ross, and Rajeev Thakur. Extending the MPI-2 generalized request interface. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 223–232. Springer, 2007.

[19] Jiuxing Liu, Balasubramanian Chandrasekaran, Weikuan Yu, Jiesheng Wu, Darius Buntinas, Sushmitha Kini, Dhabaleswar K Panda, and Pete Wyckoff. Microbenchmark performance comparison of high-speed cluster interconnects. *Ieee Micro*, 24(1):42–51, 2004.

[20] M.D. Mazzeo and P.V. Coveney. HemeLB: A high performance parallel lattice-boltzmann code for large scale fluid flow in complex geometries. *Computer Physics Communications*, 178(12):894–914, 2008.

[21] B.C. McCandless, J.M. Squyres, and A. Lumsdaine. Object Oriented MPI (OOMPI): a class library for the message passing interface. In *Proceedings. Second MPI Developer's Conference*, pages 87–94, 1996.

[22] Marius Iulian Mihailescu and Stefania Loredana Nita. New Features in C++ 20. In *Pro Cryptography and Cryptanalysis with C++ 20*, pages 125–134. Springer, 2021.

[23] Bradley Morgan, Daniel J Holmes, Anthony Skjellum, Purushotham Bangalore, and Srinivas Sridharan. Planning for performance: persistent collective operations for MPI. In *Proceedings of the 24th European MPI Users' Group Meeting*, pages 1–11, 2017.

[24] Brandon L. Morris and Anthony Skjellum. MPIgnite: An MPI-like language and prototype implementation for apache spark. *CoRR*, abs/1707.04788, 2017.

[25] Simone Pellegrini, Radu Prodan, and Thomas Fahringer. A Lightweight C++ Interface to MPI. *2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, 0:3–10, 2012.

[26] Martin Ruefenacht, Derek Schafer, Anthony Skjellum, and Purushotham V. Bangalore. MPIs Language Bindings are Holding MPI Back. *CoRR*, abs/2107.10566, 2021.

[27] Martin Schulz and Bronis R De Supinski. PN MPI tools: a whole lot greater than the sum of their parts. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–10, 2007.

[28] Anthony Skjellum, Ziyang Lu, Purushotham V. Bangalore, and Nathan Doss. Explicit Parallel Programming in C++ based on the Message-Passing Interface (MPI). In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming Using C++*, pages 465–506. MIT Press, July 1996.

[29] Anthony Skjellum, Diane G. Wooley, Ziyang Lu, Michael Wolf, Purushotham Bangalore, Andrew Lumsdaine, Jeffrey M. Squyres, and Brian C. McCandless. Object-oriented analysis and design of the Message Passing Interface. *Concurr. Comput. Pract. Exp.*, 13(4):245–292, 2001.