View-aware Message Passing Through the Integration of Kokkos and ExaMPI

Evan Drake Suggs
evan-suggs@utc.edu
University of Tennessee at Chattanooga
Chattanooga, TN, USA

Stephen L. Olivier
slolivi@sandia.gov
Sandia National Laboratories
Albuquerque, NM, USA

Jan Ciesko
jciesko@sandia.gov
Sandia National Laboratories
Albuquerque, NM, USA

Anthony Skjellum
askjellum@tntech.edu
Tennessee Technological University
Cookeville, TN, USA

ABSTRACT

Kokkos provides in-memory advanced data structures, concurrency, and algorithms to support performance portable C++ parallel programming across CPUs and GPUs. The Message Passing Interface (MPI) provides the most widely used message passing model for inter-node communication. Many programmers use both Kokkos and MPI together. In this paper, Kokkos is integrated within an MPI implementation for ease of use in applications that use both Kokkos and MPI, without sacrificing performance. For instance, this model allows passing first-class Kokkos objects directly to extended C++-based MPI APIs.

We prototype this integrated model using ExaMPI, a C++17-based subset implementation of MPI-4. We then demonstrate use of our C++-friendly APIs and Kokkos extensions through benchmarks and a mini-application. We explain why direct use of Kokkos within certain parts of the MPI implementation is crucial to performance and enhanced expressivity. Although the evaluation in this paper focuses on CPU-based examples, we also motivate why making Kokkos memory spaces visible to the MPI implementation generalizes the idea of "CPU memory" and "GPU memory" in ways that enable further optimizations in heterogeneous Exascale architectures. Finally, we describe future goals and show how these mesh both with a possible future C++ API for MPI-5 as well as the potential to accelerate MPI on such architectures.

CCS CONCEPTS

• **Software and its engineering → Parallel programming languages**.

KEYWORDS

Exascale, Kokkos, High-Performance Computing, MPI

ACM Reference Format:
Evan Drake Suggs, Stephen L. Olivier, Jan Ciesko, and Anthony Skjellum. 2023. View-aware Message Passing Through the Integration of Kokkos and ExaMPI. In *Proceedings of EuroMPI2023: the 30th European MPI Users' Group Meeting (EUROMPI '23), September 11–13, 2023, Bristol, United Kingdom.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3615318.3615321

EUROMPI '23, September 11–13, 2023, Bristol, United Kingdom

ACM ISBN 979-8-4007-0913-5/23/09…$15.00
https://doi.org/10.1145/3615318.3615321

1 INTRODUCTION

This paper proposes a series of function bindings for the Message Passing Interface (MPI) [13] that leverage features of Kokkos [15], a widely used on-node currency model, internal to an implementation and enable better integration of Kokkos and MPI at the application level.

This work has strong motivations based on current practice. While a large number of libraries and applications are still C-based, modern C++-based libraries are gaining popularity among high-performance exascale computing projects. However, the mixing of C++ and legacy C libraries provides many compatibility problems, including combining interfaces and object types that were not designed to work together. Like any other programmers, high-performance computing (HPC) professionals aim to reduce redundant code and increase functionality when possible. There are many libraries today that target HPC professionals, but some of the most heavily used are implementations of the MPI standard [13]. The MPI standard is focused on multi-processor communication, typically between nodes on high-performance computers.

ExaMPI is an implementation of MPI written in modern C++ that aims to help researchers adapt to modern challenges, such as exascale computing [14]. The Kokkos library enables programmers to manipulate both large datasets and the execution and memory spaces associated with them [15].

To a significant degree, software development is not about creating new libraries, programs, etc., but rather improving existing ones, maximizing their performance and/or enhancing their productivity. Our work therefore has the following motivations:

- To improve the general programming experience when using MPI with Kokkos.
- To minimize the possibility of bugs from MPI+Kokkos programs
- To enable optimizations for MPI+Kokkos at the language binding level or below.

As the shift towards exascale systems occurs, existing libraries must be open to improvement. The MPI standard does not have the ability to interact well with modern classes, causing friction between the rapidly evolving C++ language and the MPI standard. Our

work leverages the flexibility of ExaMPI to demonstrate how MPI could support new features (in this case, Kokkos) faster. Because of problems with existing implementations, projects like Water et al. [17] seek to replace MPI with other frameworks [16, 17].

Our work aims to address these concerns by showing how MPI could integrate new libraries into its functionality. This paper adopts a philosophy for supporting new features by rewriting small portions of code (i.e., the language bindings) and keeping the underlying structure as generic as possible, similar to the work of Trott, Plimpton, and Thompson [16]. The immediate advantage of the proposed MPI extension is to improve productivity rather than performance, as the underlying model is still the same. However, they could open the possibility to some improved performance if integrated more fully. Therefore, this work has implications beyond Kokkos, such as how true MPI-based C++ bindings will differ from classic C bindings.

There is currently no way to holistically utilize both MPI and Kokkos in modern C++ (as opposed to just C++ style syntax). Current practice is to set up MPI and Kokkos at the same time, then access the raw pointers Kokkos uses rather than its existing datatypes. While this is workable, it creates redundant code that could be optimized at a lower than user level (e.g., the length of data could be automatically inferred from the Kokkos data structures). In addition, this paper discusses the opportunities and the drawbacks of integrating these two libraries.

Our objectives are as follows:

- To create a series of function bindings within ExaMPI whose syntax utilizes Kokkos objects in the same manner as standard MPI buffers
- These function bindings should have at least comparable performance to existing practices for the majority of use cases
- Allow for easier building of MPI applications using Kokkos alongside ExaMPI

These objectives are accompanied by the following questions:

- How useful are these new bindings for users?
- What are the long term opportunities created by these bindings?
- Should these bindings follow the more traditional C-style MPI bindings or experiment with new parameters?
- Do these bindings increase or decrease performance?

Since the primary metric for this effort is not performance gains, but increased functionality, the results should deliver comparable performance. If the new bindings perform generally as well as previous methods, they will be considered acceptable.

The remainder of this paper is organized as follows. Section 2 provides background and related work on MPI, ExaMPI, and Kokkos, as well as other related work. Section 3 provides details on the APIs and strategy used to achieve this paper's objectives, while Section 4 describes specific results obtained for a number of communication APIs and test programs. Lastly, Section 5 provides conclusions and outlines future work.

2 BACKGROUND AND RELATED WORK

First we describe MPI and ExaMPI. Then we discuss Kokkos, as well as other related work.

2.1 MPI and ExaMPI

The Message Passing Interface (MPI) standard specifies a programming library interface for passing messages between peer processes [13]. The standard is separate from its various implementations, such as OpenMPI [6] and ExaMPI. Version 1.0 of MPI was released in May 1994 and focused on communication between just two processors [13]. The basic message model introduced in Version 1.0 uses contiguous data primitives (e.g., int, double, etc) passed via pointer along with a count for the number of elements. Normally, messages must be of the same `MPI_DATATYPE` [13].

Later versions introduced ways to pass non-contiguous data through derived datatypes and packing. Derived datatypes allow a developer to specify a list of datatypes and the memory offsets between them to create a new MPI datatype. This feature is useful for passing structs with defined datatypes. For more complex and variable examples such as matrix subsets, explicit packing of elements into a contiguous buffer is also supported [13].

While some innovations have been introduced, MPI datatypes have remained essentially the same since MPI 1.0[1].

MPI offers a wide range of functions, but probably the most popular are `MPI_Send` and `MPI_Recv`. A MPI program is launched on a number of different processes identified by a rank number that may be on entirely separate nodes (servers), or grouped on a single node as a virtual concurrent computer. These might be divided up into groups identified by a communicator that they share, but by default they share the same communicator "world," `MPI_COMM_WORLD` [13]. A data primitive buffer along with its length and datatype are provided to `MPI_Send` to transport this information to its destination, where `MPI_Recv` reads that information from the underlying transport and writes it to a buffer passed to it. Specifics on these and other functions, along with their new implementations, are covered later.

2.2 The specifics of ExaMPI

The ExaMPI project is designed to be a springboard for new ideas in MPI, including fault tolerant concepts, modern C++ support, and extensions to the standard that require highly effective progress for communication [14]. The complexity of producing significant improvements, modifications, and/or changes in design to existing MPI implementations is a daunting task. Existing open source middleware implementations, such as OpenMPI [6] and MPICH [7], consist of hundreds of thousands of lines of legacy code. Thus, substantially altering MPI, e.g., changing fundamental parts of MPI that manage internal state or concurrency, is prohibitively time-consuming and error-prone for most researchers. Other middleware implementations are closed-source, precluding most researchers from altering them. ExaMPI targets a smaller subset of the MPI standard than those middleware products, allowing it to focus on new functionality. This approach also avoids dead legacy code and technical debt associated with assumptions about node concurrency or progress made in the 1990s.

[1]There has been discussion among the MPI community about getting rid of MPI datatypes all together. Instead, the size of the message buffer in memory would be specified another way. This is thought to be faster, and would streamline the interface. Due to compatibility issues, this proposal has not progressed far as of this time.

ExaMPI is focused on principles-first design, highlighting the principles below:

- Enable rapid new development of new features, identify ways to increase performance, and improve understanding of the MPI standard
- Support the research interests and experiments of developers, such as effective overlap of communication and computation

ExaMPI is able to achieve these goals by limiting its scope and focusing its design on key elements of productivity and performance. ExaMPI's underlying structure is as follows. The library bindings are the top-most layer, and exist separately so they can interface with the C, C++, and Fortran languages with the same functionality. Their primary use is to take in the parameters (buffers, MPI datatype, etc.) and handle them for communication. Then, the message buffers, along with size and datatype information, are wrapped in a Payload class containing the pointers to the underlying buffer. These Payloads are wrapped in a Payload organizer, then sent to the Request class to be processed through the lower-level transports. This might include transferring or receiving data over the desired transport medium (for example, TCP, UCX, LibFabric, etc.) to communicate with other MPI processes locally or on remote nodes [14]. Depending on the function, the request is specified to send, receive, broadcast, or perform any of a number of other functions. For example, when received, the buffer is still wrapped in a Payload, but this Payload is written to rather than read from. Once this request is made, it can be activated and then either waited on within the binding (blocking communication) or handled as asynchronous, non-blocking communication. In the latter case, the function can exit and the `MPI_Wait` function can be used later to wait on the message [13, 14].

Fundamental to most MPI implementations are `MPI_Init` and `MPI_Finalize`, which bracket the portion of the program where MPI code is executed. At compile-time, an MPI compiler front-end is used (normally, `mpicc`, `mpicxx`, etc). At run-time, a program such as `mpirun` is passed MPI parameters and coordinates background information such as threading. In ExaMPI, this `mpirun` consists of Python Dæmons [13, 14].

The main version of ExaMPI targets the same traditional C functions as all other MPI implementations. MPI is a large standard with hundreds of functions (many of which are de-facto if not technically deprecated), so in order to keep itself small, ExaMPI includes 157 C functions at present. Additionally, Fortran bindings are in development for ExaMPI. While they are not C++ per se, ExaMPI's C bindings are compatible with it. There are only 16 C++ bindings in ExaMPI currently (which are not described in any standard), along with the new work featured in this paper.

2.3 Kokkos, its data structures and model

Kokkos is a library ecosystem and programming model for C++ that provides data structures, concurrency features and algorithms to support advanced C++ parallel programming across different memory spaces [15]. Created to support the needs of US Department of Energy (DOE) applications, Kokkos is a newer library than MPI and exists for similar reasons to ExaMPI: to facilitate new exascale architectures and services.

Exascale computers with large heterogeneous architectures composed of a mixture of traditional CPUs, GPUs, and other accelerators are incredibly powerful. However, differences in vendor preferred programming models among various architectures make performance portability between them a challenging problem for application developers. Kokkos addresses this problem by providing a programming model with a modern C++ interface based on template metaprogramming for the expression of data management and parallelism [15]. Its implementation comprises a set of backends that target vendor preferred programming models such Nvidia CUDA, AMD HIP, and Intel DPC++, as well as OpenMP. A major focus of the Kokkos effort is to transition capabilities into future versions of the C++ language itself, e.g., MDSpan, accepted into the C++ standard as Proposal P0009R9 [8].

2.3.1 The View Data Structure. The primary feature of Kokkos relevant to our work is the View data structure, a datatype similar to a tensor, which handles multidimensional arrays (currently up to eight dimensions) [1, 15]. Views may or may not refer directly to a given array, but they are always containers for data that handles the number of dimensions, layout, and element access [1]. Unlike tensor implementations, the View datatype is low-level, more an extension of the C array class than an implementation of the mathematical concept of tensors. At its core, it consists of a template object pointer to an underlying object, normally a C array [1].

As the Kokkos implementation of Views is essentially a smart pointer wrapper with additional functions, a large amount of meta-information can exist in the View [5, 15]. This includes elements traditionally included in vectors (i.e., length, datatype) along with the given memory and execution spaces. Kokkos offloads a large amount of processing to compile-time features such as using typedefs. A key feature is that Kokkos Views can be declared to specific memory and execution spaces, such as GPUs, accelerators, etc [5, 15].

Similar to traditional C pre-processor macros, the View templates let the C++ compiler push run-time processing of data types to compile-time, allowing both speed-up at run-time and complex data type work that would be difficult to analyze at run-time. In the case of Kokkos, these compile-time features sometimes have the side effect of obfuscating the objects themselves (i.e., there is no function or class member that describes the entire dimensional layout of View objects available at run-time). Instead it must be iterated over one dimension at a time with the extent function [5, 15].

```
1  int n = 5;
2  Kokkos::View<int*> B("B View",n);
3  int* buf = {...};
4  Kokkos::View<int*> C(buf, n);
5  // C.data() == buf;
6  // int m = ...;
7  Kokkos::View<int**> D("2D View", n, m);
8  // int o, p, q, r, s, t = ...;
9  Kokkos::View<int******[10][2]> E(
10         "2D View", n, m, o, p, q, r, s, t);
```

Listing 1: Kokkos View Creation

This section covers initialization of four basic Views as shown in Listing 1. Similar to ExaMPI, there are `Kokkos::initialize` and `Kokkos::finalize` functions to outline the Kokkos portion

of the code [5, 15] In the first constructor, View A is created with template parameters within angle brackets alongside parentheses parameters, the label string and the length. It specifies the datatype for the underlying data (int) followed by the length (in this case, 5 elements) of the first dimension. This is analogous to creating an array of 5 integer elements or int[5]. Instead of using a static length as in View A, View B has an asterisk "*" allowing its dimensions to be determined at run-time instead of compile-time. Compile-time dimensions should be passed both as constant template arguments (View<...>) and parameters. However Views are not dynamically sized, so View A and B are basically the same [5, 15].

View C in Listing 1 shows a useful but problematic feature of Kokkos. The line above View C creates a integer array called buf. The constructor for View C takes this existing buffer and the length n as parameters. This makes buf the underlying data structure C points to, creating what is known as an unmanaged View (as opposed to Managed Views) [15]. This is not legal C++ as it deals directly with the raw pointer; however, most compilers will accept it, as it is legal C syntax. These unmanaged Views lack most of the debugging information of their peers, such as the label string [5, 15]. According to discussions with the Kokkos team, a raw pointer constructor that still offers full debugging information is planned, but has not made its way into Kokkos.

Finally, View D shows the creation of a two-dimensional View. The first difference is that in the template parameters, there are 2 asterisks/dimensions instead of 1. Similarly, 2 size parameters (n and m) are passed in parentheses. This can be continued for up to 8 dimensions, with View E showing that these dimensions can be set at both compile and run-time, so long as compile-time dimensions are last. It is also possible to make an illegal raw pointer View as with View C in this manner, by putting the additional dimensions as parameters as with D [5, 15].

2.3.3 Memory Space. Another feature of Kokkos is its ability to declare in which kind of memory and execution spaces a View resides. For example, if a GPU is available, then a View can be declared as follows $View < int*, GPUExecSpace, GPUMemSpace > A$ [15]. This feature is key to some of the ideas discussed later.

```
Kokkos::View<double*> check("check",n);
Kokkos::parallel_for(check.extent(0), KOKKOS_LAMBDA(int i){
                          check(i) = i*i;
                 });
```

Listing 2: Kokkos View parallel for example

Kokkos has several parallel execution patterns similar to those used in OpenMP:
`parallel_for`, `parallel_reduce`, and `parallel_scan`. The primary pattern relevant to our work is the `parallel_for`, which iterates through Views. This for loop takes a label string, a range (number of iterations, analogous to `int i` in traditional for loop), and a lambda or functor object to be executed [5, 15]. This lambda is a more modern language feature available only in newer versions of C++. As seen in Listing 2, The `parallel_for` and lambda are normally formatted to resemble a single for loop, but the lambda contains the actual code to be executed [3, 5, 15].

2.4 Related Work

```
int n = 5;
// int destination_rank, my_rank, tag, comm = ...;
Kokkos::View<int[5]> A("A View",n);
if (my_rank) {
  MPI_Send(A.data(), int(A.size()), MPI_DOUBLE,
      destination_rank,
  tag, comm);
} else if (my_rank]) {
  MPI_Recv(A.data(), int(A.size()), MPI_DOUBLE,
      destination_rank,
  tag, comm);
}
```

Listing 3: Kokkos View Send

We now consider existing work on the use of MPI and other system-wide programming models with Kokkos. Despite the popularity of MPI and Kokkos, no framework currently exists to integrate both MPI and Kokkos in an application. However, Kokkos since its inception has been used alongside MPI in numerous applications of DOE and other HPC users. The Kokkos documentation itself includes a halo exchange that uses both MPI and Kokkos [15], which is a common application use case. A halo exchange is when data (in this case, part or all of a View) is exchanged among nearest neighbors in the Communicator group of MPI processes. The listing 3 shows the traditional way to transfer a Kokkos View by accessing the underlying data pointer (`.data()`) and size (`.size()`) along with passing the MPI datatype [15].

Previous work using both Kokkos and traditional MPI together has yielded interesting results. For example, Khuvis et al. [11] have a shown a speedup of General Matrix Multiplication (GEMM) code and the Graph500 benchmark using their version of MPI+Kokkos. They use the Intel implementation of MPI, MVAPICH and standard Kokkos. The GEMM code leverages Kokkos for parallelism of matrix multiplication alongside MPI to distribute the matrices; this code has noticeable improvement with each additional process for up to 64 processes [11]. Their Graph500 evaluation includes an MPI only baseline and MPI+Kokkos implementations with and without locks. The Graph500 results show a speed-up with the locking implementation over the MPI-only on up to forty processes, and continued speed-up with the non-locking implementation of up to 5x on 64 processes.

Other frameworks have used MPI+Kokkos to improve their strong scaling. For example, the Uintah framework for modeling chemical reactions uses MPI + Kokkos to consolidate the use of both MPI + Pthreads and MPI + Cuda into a single approach that also enables added portability [9, 10]. Another distributed memory framework, the UPC++ framework, has demonstrated its ability to work with Kokkos to replace MPI in simulating heat conduction without radical changes in performance compared to the IBM vendor version of MPI [17]. Con et al. demonstrated use of the distributed many-task MPI alternative Legion with Kokkos to offload "boiler-plate code" away from the user[2]. Daiß et al.[4] focuses on integrating support for Kokkos parallel constructs (e.g., `parallel_for`) into HPX, a parallelism framework based on asynchronous futures [4].

Overall, the literature shows a large demand for further integration of Kokkos with other HPC frameworks both for general use and from specific scientific application domains. Additionally, there are many programs in which Kokkos and MPI coexist but

are not bound together. The primary innovation that distinguishes our work from previous forms of MPI+Kokkos interaction is the ability to process Kokkos Views as native first class objects in the MPI implementation.

3 IMPLEMENTATION

```
// old method
int *recv_buf = (int*) malloc(n * sizeof(int));
MPI_Recv(recv_buf, n, MPI_INT, 1, 0,
    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
Kokkos::View<int*> recv_check(recv_buf, n);
// new method
Kokkos::View<int*> A(''New Method View'', n);
MPI_Kokkos_Recv<Kokkos::View<int*>, int>(A, n,
    MPI_INT, 1, 0, MPI_COMM_WORLD);
```

Listing 4: Techniques for passing Views using MPI and Kokkos.

In this section, we describe the individual bindings created for our API extensions. Consider first the status quo for a programmer using Kokkos Views with MPI: The View's underlying pointer must be accessed by the programmer, opening up the possibility for memory leaks. In contrast, the functions in our MPI extensions accept Kokkos Views.

All that is required to receive a View is to create an existing View of the correct size and copy to it. For example, `MPI_Kokkos_Send` sends the underlying View as a Payload. Its counterpart, `MPI_Kokkos_Recv` receives this Payload, then wraps it back into a View object (either a View pointer or copying to full array with a performance penalty). This model is used for all the other MPI bindings as well. Note that Currently this method is only compatible with contiguous arrays currently.

3.1 MPI Extension API Functions

Each of the functions in the MPI + Kokkos implementation is based on a common MPI equivalent. The proposed MPI extension functions are listed below. The naming of these functions is `MPI_Kokkos_X` for simplicity of implementation (avoiding overloading errors) and to distinguish them from the original versions. However, there is no reason why these functions could not be overloads of their standard counterparts or placed inside a namespace. In future versions of MPI+Kokkos, these functions and others (e.g., non-contigous versions of these functions) will be implementated as overloads of the MPI standard functions. Note that scalar and contiguous one-dimensional sub-Views can be sent using these functions.

(1) MPI_Kokkos_Send
(2) MPI_Kokkos_Recv
(3) MPI_Kokkos_Isend
(4) MPI_Kokkos_Irecv
(5) MPI_Kokkos_Bcast
(6) MPI_Kokkos_Allgather
(7) MPI_Kokkos_Allreduce

Each section shows the function's name, then the template parameters, followed by the normal function parameters. Templates, especially for complex structures such as Views, contain more than just type information, so a more generic template allows compatibility with different Views. Each binding uses roughly the following template to accept any View class, $template < class\ View\_t >$ along with any additional template parameters.

3.1.1 *MPI_Kokkos_Send and MPI_Kokkos_Recv.*

MPI_Kokkos_Send$\langle View_t, Datatype \rangle$
(View_t * buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

TEMPL	View_t	View type
TEMPL	Datatype	underlying datatype
IN	buf	address of View
IN	count	number of elements
IN	datatype	datatype of View's elements
IN	dest	destination rank
IN	tag	message tag
IN	comm	handle to communicator

MPI_Kokkos_Recv$\langle View_t, Datatype \rangle$
(View_t * buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm)

TEMPL	View_t	View type
TEMPL	Datatype	underlying datatype
OUT	buf	address of View
IN	count	number of elements
IN	datatype	datatype of View's elements
IN	source	source rank
IN	tag	message tag
IN	comm	handle to communicator

These are variations of `MPI_Send` and `MPI_Recv` which take Views as first class objects, `MPI_Kokkos_Send` and `MPI_Kokkos_Recv`. `MPI_Kokkos_Send` takes a View as the first parameter in the place of the usual buffer and sends this over as an ExaMPI Payload. `MPI_Kokkos_Send`'s counterpart, `MPI_Kokkos_Recv` receives the Payload sent by `MPI_Kokkos_Send`, then wraps that in a View object and sends that to the pointer passed as a parameter.

All functions have the same template parameter, `View_t` for taking in Views, along with `Datatype`, which allows the user to pass the underlying datatype for the View. These functions assume that the Views are identically shaped on both ranks for MPI transport, similar to how MPI standard functions assume buffers are sized. This places responsibility on the user and avoids having to communicate details about the View, such as dimensions, type, etc. Similarly, Kokkos reductions require the same View on both ends. We considered two ways to add templates to the bindings. The first was to write the bindings in the primary `MPI` header, but this approach would have increased its size substantially. The second, which we chose, was to include a template instantiation for View type and dimensions (e.g., Kokkos::View<int*> and Kokkos::View<int**> each required different instantiations).

The implementation code does not just redirect the View's data to `MPI_Send/Recv`, but to the same underlying Request layer as all other buffers. The first approach would work, but is less interesting and has less flexibility for future work. It also introduces the slight overhead of another function call.

3.1.2 MPI_Kokkos_Isend and MPI_Kokkos_IRecv. MPI_Kokkos_Isend⟨*View_t*, *Datatype*⟩ (View_t * buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)

TEMPL	View_t	View type
TEMPL	Datatype	underlying datatype
IN	buf	address of View
IN	count	number of elements
IN	datatype	datatype of View's elements
IN	dest	destination rank
IN	tag	message tag
IN	comm	handle to communicator
OUT	request	handle to communication request

MPI_Kokkos_Irecv⟨*View_t*, *Datatype*⟩ (View_t * buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)

TEMPL	View_t	View type
TEMPL	Datatype	underlying datatype
INOUT	buf	address of View
IN	count	number of elements
IN	datatype	datatype of View's elements
IN	source	source rank
IN	tag	message tag
IN	comm	handle to communicator
OUT	request	handle to communication request

`MPI_Kokkos_Isend` and `MPI_Kokkos_IRecv` are the non-blocking versions of `MPI_Kokkos_Send/Recv`, where the function call is returned before communication is finished. `MPI_Kokkos_IRecv` is the counterpart of `MPI_Kokkos_Irecv` in the same way as `MPI_Kokkos_Recv` is to `MPI_Kokkos_Send`. Like `MPI_Kokkos_Recv`, an existing View address is received from a send side and its data is written to a existing View.

We used existing query functions, rather than creating new ones, to ease the process of porting applications. For example, if an existing ExaMPI program is converted to our API, any use of `MPI_Wait` will not have to be changed. Otherwise, this function differs from its non-Kokkos counterpart only in its ability to handle a Kokkos View as a first class object. Note that only `MPI_Wait` has been used in the evaluation.

3.2 Collective Functions

3.2.1 MPI_Kokkos_Bcast. MPI_Kokkos_Bcast⟨*View_t*, *Datatype*⟩ (View_t * buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm)

TEMPL	View_t	View type
TEMPL	Datatype	underlying datatype
INOUT	buf	address of View
IN	count	number of elements
IN	datatype	datatype of View's elements
IN	root	root rank
IN	comm	handle to communicator

`MPI_Kokkos_Bcast` is a version of `MPI_Bcast`. The primary difference from the point-to-point bindings is the use of a collective operation and the need to ensure that the Views would be handled correctly whether the process was a root or not.

3.2.2 MPI_Kokkos_Allgather. MPI_Kokkos_Allgather ⟨*View_t*, *Datatype*⟩ (View_t * buf, int count, MPI_Datatype datatype, View_t * recv_buf, int recv_count, MPI_Datatype recv_type, MPI_Comm comm)

TEMPL	View_t	View type
TEMPL	Datatype	underlying datatype
INOUT	buf	address of View to be sent
IN	count	number of elements for buf
IN	datatype	datatype of sender View's elements
INOUT	recv_buf	address of receiving View
IN	recv_count	number of elements for recv_buf
IN	recv_type	datatype of receiver View's elements
IN	comm	handle to communicator

Next is the collective communication function, `MPI_Kokkos_Allgather`. This is a version of `MPI_Allgather`, a function which collects (gathers) an input View from all processes then arranges them into a View with inputs from each View ordered by their sending process's index ranking [13]. Unlike the previous functions, this function requires the creation of two payloads, one for sending and another for receiving.

3.2.3 MPI_Kokkos_Allreduce. MPI_Kokkos_Allreduce ⟨*View_t*, *Datatype*⟩ (View_t * send_buf, View_t * recv_buf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

TEMPL	View_t	View type
TEMPL	Datatype	underlying datatype
INOUT	send_buf	address of View to be sent
INOUT	recv_buf	address of View to be written to
IN	count	number of elements for buf
IN	datatype	datatype of sender View's elements
IN	op	operation to be performed
IN	comm	handle to communicator

`MPI_Kokkos_Allreduce` is a collective function that collects values from several processes, performs an operation on them (`MPI_Op`) and broadcasts the result to all processes involved. The `MPI_Op` can be any of a number of operations such as sum, max, etc. This function required more conceptual work than previous function. The key design choice is whether this extension should return the resulting buffer from the reduce operation as a View or as a data primitive. To be more consistent with the previously covered functions, this function uses Views for both the send and receive buffers.

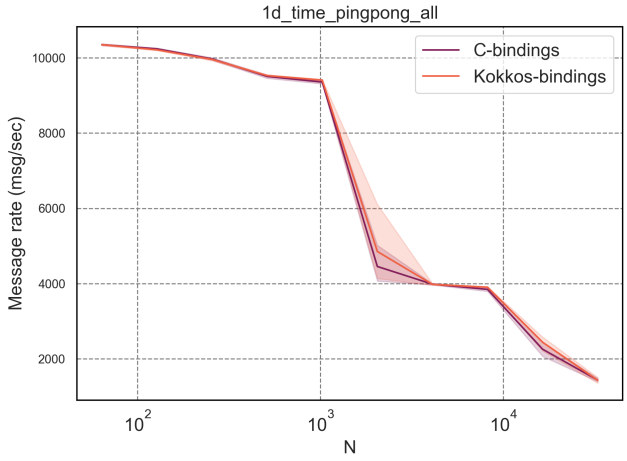

Figure 1: One-dimensional pingpong test using MPI Extension bindings versus traditional method (higher is better)

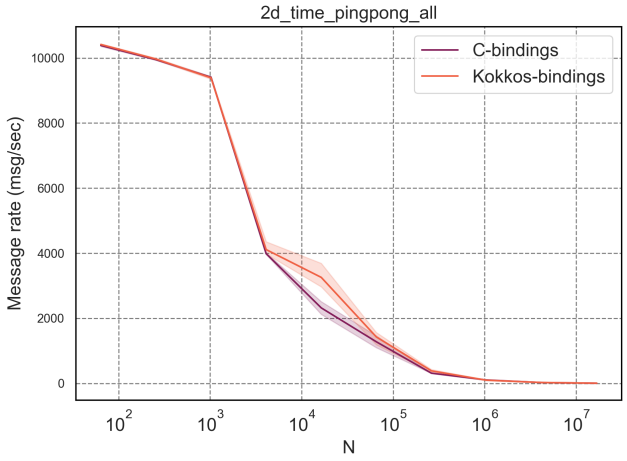

Figure 2: Two-dimensional pingpong test using MPI Extension bindings versus traditional method (higher is better)

4 RESULTS

```
Kokkos::View<int*> check("Example View", n);
// old method
MPI_Recv(check.data(), n, MPI_INT, 1, 0,
    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
// new method
MPI_Kokkos_Recv<Kokkos::View<int*>, int>(&A, n, MPI_INT, 1, 0,
    MPI_COMM_WORLD);
```

Listing 5: Kokkos View + MPI examples

In this section, we evaluate the performance of our extended Kokkos-aware MPI functions compared to using Kokkos with the corresponding existing MPI functions. First we present results for a one-dimensional `MPI_Send` and `MPI_Recv` test, followed by results for two and three dimensional arrays, along with a slightly different test using `MPI_Bcast`. Finally, we demonstrate performance using our MPI extensions in a heat diffusion mini-application.

The following tests were run on the Blake testbed at Sandia National Laboratories. Blake contains 40 24-core Intel Xeon Platinum nodes with the Intel Omnipath Interconnect.

4.1 One-Dimension `MPI_Send` And `MPI_Recv` Tests

This section covers the tests comparing the new Kokkos bindings for sending and receiving using a View (MPI_Kokkos_Send and MPI_Kokkos_Recv) and traditional C bindings. Figure 1 shows the results for MPI ping-pong using one-dimensional data. In a ping-pong test, one process sends a buffer (here a View) and another receives and alters the buffer. Finally, the buffer is sent back to the original sending process. The primary difference between the two methods is shown in Listing 5: The old method for receiving an array must touch the `.data()` method directly, while the new method accepts the Kokkos View.

Aside from the differing methods, the main variable is the size of the View, each consisting of a single dimensional array ranging in size from 64 to 32768 elements (or as mapped on the X-axis, roughly 10 to 10^4 elements) multiplying by 2 at each step. For each length of the array, 500 runs were done with each point in Figure 1 representing the average message rate. The times recorded in the figure are from the original sending side as it will have the longest overall time. This is equivalent to a latency test, as the message rate is the number of messages per second, as determined by the ping-pong's overall time/latency. The standard deviation of the mean is plotted as the shaded areas.

In Figure 1, the bindings have roughly identical performance at most points. The outliers are roughly between 1000 and 8000 elements and after 10,000 elements, where the C-bindings perform better. In the other portions of the graph, the C bindings are slightly below the Kokkos bindings. Since these regimes exhibit an increased standard deviation generally, it it possible that the underlying ExaMPI code tends to be more variable in these cases. Overall, these results seem to be comparable for both methods with both types of bindings. Since the primary goal was roughly equal performance for each method, the bindings performed well by our metrics.

4.2 Two- and Three-dimensional Send And Recv Tests

The next series of tests use two and three dimensional Views, but their code is otherwise identical to the code in the single dimension experiment, and the experimental configurations are likewise the same. The results are shown in Figures 2 and 3, respectively. For each , the x-axis number is the length of one dimension, so the axis label 2 indicates $2 \times 2 = 4$ elements in the two dimensional case and $2 \times 2 \times 2 = 8$ elements in the three dimensional case.

For both graphs, the performance with the new bindings and old bindings are mostly equivalent in execution time and has similar standard error. Again, there are narrow regimes that exhibit increased standard deviation and a stronger performance for one binding or another.

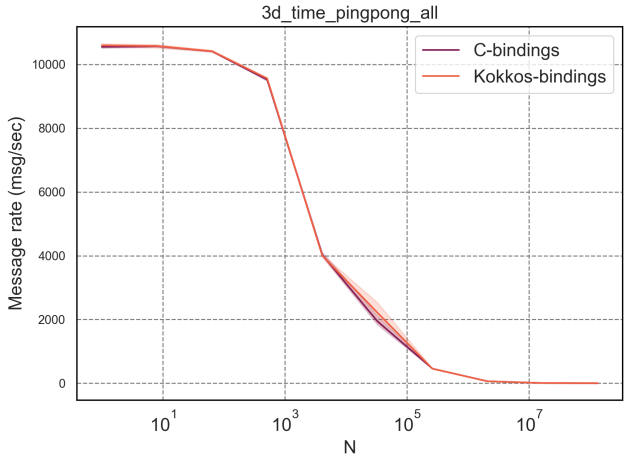

Figure 3: Three-dimensional pingpong test using MPI Extension bindings versus traditional method (higher is better)

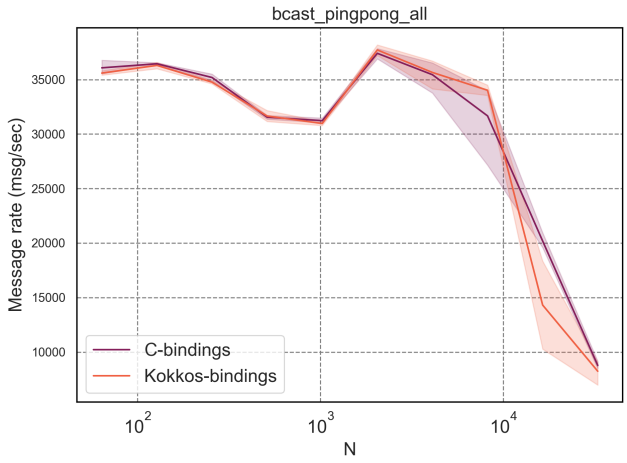

Figure 4: MPI Extension bindings versus traditional method for broadcast (higher is better)

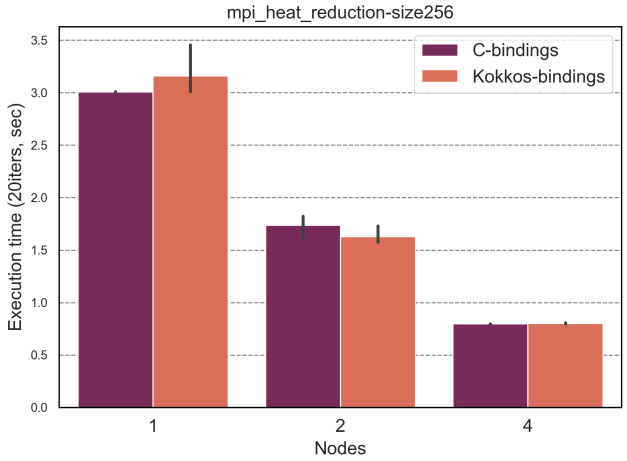

Figure 5: Heat Conduction with 256 elements (lower is better)

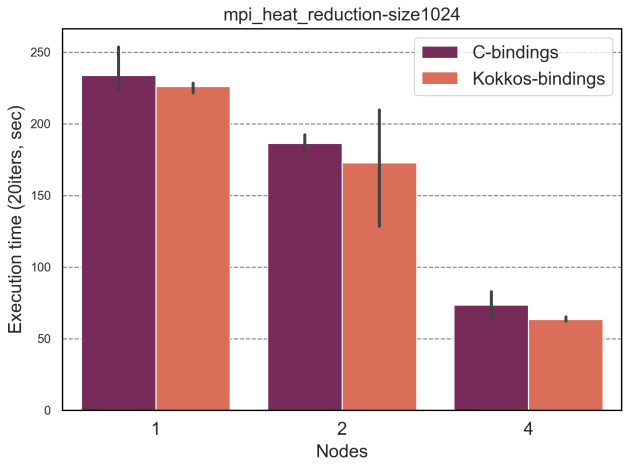

Figure 6: Heat Conduction with 1024 elements (lower is better)

4.3 `MPI_Kokkos_Bcast` Tests

This experiment runs a `MPI_Bcast` test with both the new Kokkos bindings and without. It does not run a pingpong, instead transporting a single one-dimensional View using `MPI_Bcast` with between 100 and 800 elements increasing by increments of 100 with two processes. The time it takes to run this test is recorded for every process, then the maximum is found using the `MPI_Allreduce` function. We ran each of these tests 500 times for a given number of processes then averaged the results.

As shown in Figure 4, the average time and standard error varies more significantly than the previous tests. This behavior is likely due to the fact that the default broadcast algorithm is a series of linear sends, meaning each is dependent on the non-deterministic nature of not just one transport, but every transport. Again, this is mostly pronounced when N is between 10^3 and 10^4. The C bindings here have generally more messages per second, but normally within standard deviation. Within the area of increased standard deviation, the Kokkos bindings reaching their highest rate at just over 40,000 messages/second but for larger sizes the C bindings perform best.

4.4 MPI Heat Conduction

This section covers a series of timing tests using the Kokkos heat conduction mini-application included in the Kokkos tutorials [12]. As with the previous tests, one version uses ExaMPI's C bindings and another uses Kokkos bindings. However, Unlike the previous tests, the MPI functions constitute a relatively small part of the entire program, namely "halo exchange" point-to-point operations and a few collectives. The rest of the program consists of local computations using the View data that are not dependent on MPI.

Figure 5 and Figure 6 each show two different series of runs. The first has the x,y, and z dimensions for the program set to 256

elements, while the second was set to 1024 elements. Each run computed 20 time steps for each combination of nodes (either 1, 2, and 4 nodes) and was launched three times in order to quantify variance.

Figure 5 has roughly equal performance in execution time for both. Kokkos bindings perform slightly better than C bindings on one node with high standard deviation, slightly worse on two nodes with lower standard deviation, and almost exactly equal on four nodes with very low standard deviation. Figure 6 tells a slightly different story for the larger problem size, as the Kokkos bindings perform slightly better than C bindings at each step. There is an increased standard deviation when running 2 nodes, but this seems to be an outlier in the results.

5 CONCLUSIONS & FUTURE WORK

Here, we present conclusions and outline future work, including work already on-going beyond the scope of this communication.

5.1 Conclusions

Our work set about to integrate two programming models, MPI and Kokkos, without negatively impacting performance. Kokkos was chosen due to its prevalence in HPC codes targeting performance portability and its use of modern C++ data structures. Our survey of related work that utilizes both MPI and Kokkos indicate that demand exists for this work, but an interface between its data structures and MPI has not yet been created. Since the ExaMPI implementation of MPI is written in C++, using it to prototype our API enabled the use of templates that would not have been possible in many other implementations.

The primary design decision for our MPI extensions was that users should not have to touch the `.data()` method for Kokkos Views and should instead be able to use Views as buffers directly in MPI. Several MPI bindings were implemented to use Kokkos View objects as their primary buffers alongside template parameters for internal use.

The evaluation of our extensions implemented in ExaMPI found that the new Kokkos bindings performed similarly to the existing C bindings. This was particularly true for the two and three dimensional View tests, which displayed almost identical performance for the majority of View sizes and only small differences in other cases. The broadcast tests were less conclusive in that the performance of both the MPI extension bindings and the traditional methods varied somewhat more for certain message sizes. Finally, we applied our extensions to a Kokkos+MPI heat diffusion mini-application, demonstrating performance commensurate to that of the original code based on C MPI bindings. Unlike previous efforts that used both MPI and Kokkos, we were able to implement several MPI bindings using Kokkos View objects natively in MPI, and did so without significant loss of performance. These bindings also preserve the C++ idiom of the Kokkos View by allowing templates.

5.2 Future Work

Going forward, this work will encompass a wider array of standard MPI functions and more Kokkos-specific functions, with work beginning on additional collective functions such as All-To-All, Scatter and Gather in the near future. The bindings covered in this paper will eventually be function overloads for their standard MPI equivalents and the `MPI_Kokkos_X` scheme will be dropped. Another future goal of this project is more device specific support (i.e., `MPI_Send<View, class, Device>`) for GPUs and other hardware.

Currently, the Kokkos MPI bindings closely follow existing MPI functions from the C Interface with their primary difference being the introduction of View template parameters. One alternative to our current approach would be bindings that return Views directly, rather than `MPI_Success` codes, upon completion. Such a scheme could be desirable and alleviate some issues with View declaration on the user's part. However, it would be more difficult to implement while maintaining our fundamental goal of extending the MPI interface for Kokkos without performance benefit.

A major use case not covered by the MPI extension so far is the support of non-contiguous Views. Generally, non-contiguous arrays can be of several different types: arrays/data structures that are placed into multiple segments or pages of memory or arrays whose elements are out of order within a contiguous memory block (such as a transposed array). While both the MPI standard and Kokkos have methods for dealing with non-contiguous arrays or Views, they are not directly compatible. MPI has derived datatypes where strides are defined for new data structures, allowing disparate memory to be packed into a contiguous space. Kokkos has the ability to adjust the defined strides for non-contiguous memory.

As mentioned previously, the extension shown in this paper assumes some things users (e.g., that Views are similarly shaped.). The goal of this work is to overload `MPI` functions so that when different types of Views are sent, their equivalents functions are substituted using SFINAE rules and `std::enable_if`. For example, when a non-contiguous

The primary path forward for the non-contiguous case is to create a separate transport back-end. Kokkos provides an `is_contiguous` method for this purpose, along with a stride interface to indicate the spaces between memory (strides) [15]. The back-end would use the stride information to send chunks of the View in parallel for non-contiguous Views. This mechanism would have improved performance over the more general case of manually packing the Views. The use of MPI datatypes would be eliminated in this path forward; this direction is promising because the authors and many others have found that direct packing/unpacking (marshaling/unmarshaling) of data generally works faster than using MPI derived datatypes in practice. Even when the datatypes are optimized for certain situations, other use cases often arise.

Further, a new back-end could be created for existing use cases and enabling support for Views within the lower levels of ExaMPI. A related notion is to create an alternative back-end for ExaMPI that only deals with data on a byte level instead of by datatype. This back-end would be more flexible for Views as it could just take in memory size or strides.

ACKNOWLEDGMENTS

Thanks to Riley Shipley and Derek Schafer for reviewing this work, along with the ExaMPI team and the Kokkos team for technical support. Additional thanks to Dr. Joseph Dumas II and Dr. Michael Ward. Funding in part is acknowledged from these NSF Grants 191897, 21501020, and 2201497, as well as 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. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525. This written work is authored by an employee of NTESS. The employee, not NTESS, owns the right, title and interest in and to the written work and is responsible for its contents. Any subjective views or opinions that might be expressed in the written work do not necessarily represent the views of the the U.S. Government. The publisher acknowledges that the U.S. Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this written work or allow others to do so, for U.S. Government purposes. The DOE will provide public access to results of federally sponsored research in accordance with the DOE Public Access Plan.

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 Sandia National Laboratories.

REFERENCES

[1] Björn Andres, Ullrich Köthe, Thorben Kröger, and Fred A. Hamprecht. 2010. Runtime-Flexible Multi-dimensional Arrays and Views for C++98 and C++0x. *CoRR* abs/1008.2909 (2010). arXiv:1008.2909 http://arxiv.org/abs/1008.2909

[2] Ethan T. Coon, Wael R. Elwasif, Himanshu Pillai, Peter E. Thornton, and Scott L. Painter. 2019. Exploring the Use of Novel Programming Models in Land Surface Models. In *2019 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI (PAW-ATM)*. 1–10. https://doi.org/10.1109/PAW-ATM49560.2019.00006

[3] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE* 5, 1 (1998), 46–55.

[4] Gregor Daiß, Mikael Simberg, Auriane Reverdell, John Biddiscombe, Theresa Pollinger, Hartmut Kaiser, and Dirk Pflüger. 2021. Beyond Fork-Join: Integration of Performance Portable Kokkos Kernels with HPX. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 377–386. https://doi.org/10.1109/IPDPSW52791.2021.00066

[5] H. Carter Edwards and Christian R. Trott. 2013. Kokkos: Enabling Performance Portability Across Manycore Architectures. In *2013 Extreme Scaling Workshop (xsw 2013)*. 18–24. https://doi.org/10.1109/XSW.2013.7

[6] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. 2004. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*. Budapest, Hungary, 97–104.

[7] William Gropp, Ewing Lusk, and Anthony Skjellum. 2014. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press.

[8] David S. Hollman, Bryce Adelstein-Lelbach, H. Carter Edwards, Mark Hoemmen, Daniel Sunderland, and Christian R. Trott. 2020. mdspan in C++: A Case Study in the Integration of Performance Portable Features into International Language Standards. *CoRR* abs/2010.06474 (2020). arXiv:2010.06474 https://arxiv.org/abs/2010.06474

[9] John K. Holmen, Alan Humphrey, Daniel Sunderland, and Martin Berzins. 2017. Improving Uintah's Scalability Through the Use of Portable Kokkos-Based Data Parallel Tasks. In *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact* (New Orleans, LA, USA) *(PEARC17)*. Association for Computing Machinery, New York, NY, USA, Article 27, 8 pages. https://doi.org/10.1145/3093338.3093388

[10] John K. Holmen, Brad Peterson, and Martin Berzins. 2019. An Approach for Indirectly Adopting a Performance Portability Layer in Large Legacy Codes. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 36–49. https://doi.org/10.1109/P3HPC49587.2019.00009

[11] Samuel Khuvis, Karen Tomko, Jahanzeb Hashmi, and Dhabaleswar K. Panda. 2020. Exploring Hybrid MPI+Kokkos Tasks Programming Model. In *2020 IEEE/ACM 3rd Annual Parallel Applications Workshop: Alternatives To MPI+X (PAW-ATM)*. 66–73. https://doi.org/10.1109/PAWATM51920.2020.00011

[12] Sandia National Laboratories. 2023. Kokkos Tutorials. https://github.com/kokkos/kokkos-tutorials

[13] Message Passing Interface Forum. 2021. *MPI: A Message-Passing Interface Standard Version 4.0*. https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf

[14] Anthony Skjellum, Martin Rüfenacht, Nawrin Sultana, Derek Schafer, Ignacio Laguna, and Kathryn Mohror. 2020. ExaMPI: A Modern Design and Implementation to Accelerate Message Passing Interface Innovation. In *High Performance Computing*, Juan Luis Crespo-Mariño and Esteban Meneses-Rojas (Eds.). Springer International Publishing, Cham, 153–169.

[15] Christian R. Trott, Damien Lebrun-Grandié, Daniel Arndt, Jan Ciesko, Vinh Dang, Nathan Ellingwood, Rahulkumar Gayatri, Evan Harvey, Daisy S. Hollman, Dan Ibanez, Nevin Liber, Jonathan Madsen, Jeff Miles, David Poliakoff, Amy Powell, Sivasankaran Rajamanickam, Mikael Simberg, Dan Sunderland, Bruno Turcksin, and Jeremiah Wilke. 2022. Kokkos 3: Programming Model Extensions for the Exascale Era. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2022), 805–817. https://doi.org/10.1109/TPDS.2021.3097283

[16] Christian Robert Trott, Steven J. Plimpton, and Aidan P. Thompson. 2017. Solving the performance portability issue with Kokkos. (8 2017). https://www.osti.gov/biblio/1467794

[17] Daniel Waters, Colin A MacLean, Dan Bonachea, and Paul Hargrove. 2021. Demonstrating UPC++/Kokkos Interoperability in a Heat Conduction Simulation (Extended Abstract). https://doi.org/10.25344/S4630V