KOKKOS-ENHANCED EXAMPI

By

Evan Drake Suggs

Anthony Skjellum, Ph.D.
Professor of Computer Science and Engineering
(Chair)

Joseph Dumas, Ph.D.
UC Foundation Professor
(Committee Member)

Michael J. Ward, Ph.D.
Advanced Cyber Infrastructure Facilitator
(Committee Member)

KOKKOS-ENHANCED EXAMPI

By

Evan Drake Suggs

A Thesis Submitted to the Faculty of the
University of Tennessee at Chattanooga
in Partial Fulfillment of the Requirements of the
Master of Science in Computer Science

The University of Tennessee at Chattanooga
Chattanooga, Tennessee

August 2023

ABSTRACT

Kokkos provides in-memory data structures, concurrency, and algorithms for advanced C++ parallel programming. The Message Passing Interface (MPI) provides the ubiquitous model for inter-node communication. Using Kokkos and MPI together is an important use case. Here, Kokkos is integrated within an MPI implementation to obtain performance and functionality benefits both for the MPI itself, and applications that use them. For instance, this enables passing first-class Kokkos objects directly to extended C++-based MPI APIs.

ExaMPI, a C++17-based subset implementation of MPI-4 is integrated with Kokkos. Working with C++-friendly APIs and Kokkos extensions, example benefits of functionality and performance are shown. This thesis explains why use of Kokkos within the certain parts of the MPI implementation are crucial to yielding added performance. This paper motivates why making Kokkos memory spaces visible to the MPI implementation generalizes the idea of "CPU memory' and "GPU" memory,' providing further optimizations in heterogeneous Exascale architectures.

DEDICATION

To S.D.B. and T.L.S.

ACKNOWLEDGMENTS

Thanks to Riley Shipley and Derek Schafer for reviewing this thesis, along with the ExaMPI team and the Kokkos team for technical support. Additional thanks to the thesis committee, Dr. Anthony Skjellum, Dr. Joseph Dumas II, and Dr. Michael Ward.

Funding provided with NSF Grants: #1821431, 1822191, 1918987, 1925603, 2201497.

TABLE OF CONTENTS

ABSTRACT iv

DEDICATION v

ACKNOWLEDGMENTS vi

LIST OF FIGURES x

1. INTRODUCTION 1

1.1 Background 1
1.2 Motivation 2
1.3 Problem Statement & Objectives 3
1.4 Contributions 4
1.5 Outline 4

2. LITERATURE REVIEW 6

2.1 MPI and ExaMPI 6
2.1.1 MPI Standard 6
2.2 The specifics of ExaMPI 7
2.3 Kokkos, its data structures and model 9
2.3.1 The View Data Structure 9
2.3.2 View Syntax 10
2.3.3 Memory Space 11
2.3.4 Dispatch Operations 12
2.4 Previous Kokkos+MPI versions 12
2.5 Summary 14

3. METHODOLOGY 15

3.1 Requirements 15
3.1.1 Layouts and Contiguous Views 16
3.1.2 Binding Syntax and Setup 16
3.2 Summary 17

4. IMPLEMENTATION 19

4.1 MPI Extension API Functions 19
4.1.1 MPI_Kokkos_Get_Dims 20
4.1.2 MPI_Kokkos_Send and MPI_Kokkos_Recv 20
4.1.3 MPI_Kokkos_Isend and MPI_Kokkos_IRecv 22
4.1.4 MPI_Kokkos_Bcast 24
4.1.5 MPI_Kokkos_Allgather 24
4.1.6 MPI_Kokkos_Recv_Dims 25
4.1.7 MPI_Kokkos_Irecv_Dims 26
4.2 Miscellaneous 27
4.2.1 CMake and Binding Creation 27
4.2.2 Templates 28

5. RESULTS 30

5.1 Single-Dimension MPI_Send And MPI_Recv Tests 30
5.2 Multi-Dimension Send And Recv Tests 32
5.3 MPI_Kokkos_Broadcast Tests 35

6. CONCLUSIONS 37

6.1 Future Work 37

BIBLIOGRAPHY 39

APPENDICES

A. KOKKOS/EXAMPI BINDING CODE 41

A Bindings Code 42
A.1 MPI_Kokkos_Send 42
A.2 MPI_Kokkos_Recv 43
A.3 MPI_Kokkos_ISend 45
A.4 MPI_Kokkos_Irecv 46
A.5 MPI_Kokkos_Bcast 47
A.6 MPI_Kokkos_Allgather 49
A.7 MPI_Kokkos_Recv_Dims 51
A.8 MPI_Kokkos_Irecv_Dims 53
B Test Code 55
B.1 New Bindings Kokkos Pingpong Test 55
B.2 Old Method Kokkos Pingpong Test 58
B.3 Two Dimensional New Bindings Kokkos Pingpong Test 60
B.4 Two Dimensional Old Bindings Kokkos Pingpong Test 63

B.5 Three Dimensional New Bindings Kokkos Pingpong Test 66
B.6 Three Dimensional Old Bindings Kokkos Pingpong Test 69
B.7 New Bindings Kokkos Broadcast Test 71
B.8 Old Bindings Kokkos Broadcast Test 73

VITA 76

LIST OF FIGURES

5.1 New MPI Extension bindings vs traditional method averages for pingpong tests 31
5.2 2D MPI Extension bindings vs. traditional method averages pingpong tests 33
5.3 3D MPI Extension bindings vs. traditional method averages pingpong tests 34
5.4 MPI Extension bindings vs. traditional method averages 35

CHAPTER 1

INTRODUCTION

While a large number of libraries and applications are still in C, modern C++-based libraries are gaining popularity among high-performance exascale ($\geq 10^{18}$ floating point instructions per second) computing projects. However, the mixing of legacy C libraries with C++ ones provides many compatibility problems, including combining interfaces and object types that were not designed to work together. Like any other programmer, 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 Message Passing Interface (MPI) standard [8]. The MPI standard is focused on multi-processor communication and is often used for communicating 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 [9]. The Kokkos library allows programmers to manipulate both large datasets and the execution and memory spaces associated with them [11]. This thesis aims to create a series of function bindings within ExaMPI to enable it to process Kokkos features internally. The rest of this chapter discusses what led to this thesis, its objectives, contributions, and offers an outline of the entire thesis.

1.1 Background

The MPI standard provides a widely used message passing model for multicomputer and multiprocessor communication [8]. The first version of the MPI standard was released in May 1994 and is currently on version 4.0 of the standard [8]. The MPI standard details a library for passing message buffers (memory buffers, often data primitives, that will either be sent or received as a message between processors). The first form of MPI communication is blocking communication, in which a message buffer is sent then waited on before doing more operations. There is also

asynchronous, non-blocking communication, where users can begin communication then do other operations before checking on its completion [8].

The MPI implementation utilized for this project, known as ExaMPI, [9] is a C++ MPI implementation that leverages modern C++ features, such as smart pointers [1]. ExaMPI is a research-oriented implementation created by UTC in collaboration with other partners designed for rapid testing of new features, particularly ideas for future versions of the MPI standard [9].

The Kokkos library provides in-memory advanced data structures, concurrency, and algorithms to support advanced C++ parallel programming [3, 11]. Its ability to handle multi-dimensional, typed arrays across memory spaces makes it useful for modern mathematics and data-oriented applications [11]. Since both are widely-used libraries for high-performance computing, work using both Kokkos and MPI together is becoming more common, such as Khuvis et al. [7], where combining MPI and Kokkos increased performance on existing benchmarks. The requirement of using traditional MPI C bindings to pass Kokkos features forces programmers to use workarounds.

1.2 Motivation

A large amount of software development is not creating new libraries, programs, etc., but improving existing ones and maximizing their performance and/or enhancing their productivity. This thesis 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 allow optimizations for MPI+Kokkos at the language binding level or below.

As the shift towards exascale systems occurs, existing libraries will have to be open to improvement. The MPI standard does not have the ability to act with modern classes; this is causing friction between the rapidly evolving C++ language and the MPI standard. This thesis seeks to leverage the flexibility of ExaMPI to demonstrate how MPI could support new features (in

[1] New versions of the C++ standard roll out every 3 years, with the most recent versions of C++ being C++20/23 for the years 2020 and 2023 at the time of this thesis.

this case, Kokkos) faster. Because of problems with existing implementations, projects like Water et al. [14]. seek to replace MPI with other frameworks [12, 14].

This thesis seeks to to address these concerns by showing how MPI could integrate new libraries into its functionality. Similar to Trott, Plimpton, and Thompson [12], this thesis 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. The primary advantage of the proposed MPI extension is to save development time rather than run-time, as the underlying model is still the same. However, this could open the possibility to some improved performance if integrated more fully. Therefore, this work has implications separate from Kokkos, such as how true MPI-based C++ bindings will differ from classic C bindings.

1.3 Problem Statement & Objectives

There does not currently exist a way to 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 a lot of redundant code that could be optimized at a lower than user level (i.e., the length of data could be automatically inferred from the Kokkos data structures). In addition, this thesis discusses the opportunities and the drawbacks of integrating these two libraries.

The thesis' 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 similar performance to existing practices for the majority of use cases
- Allow easier building of MPI applications using Kokkos alongside ExaMPI

These objectives are followed 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 decrease performance?

Since the primary metric for this effort is not performance gains, but increased functionality, the results should deliver comparable performance. This effort was lucky enough to have feedback from the Kokkos team at Sandia National Laboratory and these discussions will feed into how the project was designed. So long as the new bindings perform generally as well as previous methods, they will be considered acceptable.

1.4 Contributions

This thesis creates a new MPI function extension in ExaMPI that allows Kokkos features to be used alongside MPI features to obtain development functionality benefits over having the two running separately. Using this model, it will be possible to pass first-class Kokkos objects directly to a C++-based MPI extension (MPIX) Application Programmer Interface (API). This was done for clarity, and later versions could just overload the existing MPI versions. Alternatively, this work has implications separate from Kokkos, and completely new C++ function definitions could be created. For example, the new functions in the extension can use templates rather than MPI datatype parameters. By internalizing the usage of Kokkos in MPI, it decreases the opportunities for bugs that arise from using Kokkos in unintuitive ways. If new function bindings prove to be comparable in performance to existing methods, then they will save programming time, minimize the possibility of bugs, and allow further optimizations.

1.5 Outline

The structure of the thesis is as follows. First, Chapter 2 is a literature review consisting of an overview of the basics of Kokkos and what differentiates ExaMPI from other MPI implementations. Then, the new MPI extension created for this project is shown, including what was required to connect Kokkos and ExaMPI and some of the new bindings created for it. Next, the test results are presented, which should show comparable performance to existing methods

of transferring View data. Finally, this thesis is summarized and future work is discussed which consists of ideas brought up by existing work and other conclusions.

CHAPTER 2

LITERATURE REVIEW

This chapter covers background information of the ExaMPI implementation and MPI in general, the Kokkos programming library, and discusses existing Kokkos+MPI work while identifying gaps.

2.1 MPI and ExaMPI

Message Passing based on the MPI standard, as well as the ExaMPI implementation, are covered here.

2.1.1 MPI Standard

The Message Passing Interface (MPI) standard specifies a programming library interface for passing messages between peer processes [8]. The standard is separate from its various implementations, such as OpenMPI [4] and ExaMPI. Version 1.0 of MPI was released in May 1994 and focused on communication between just 2 processors [8]. 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 number of elements. Normally, messages are restricted to all being of the same `MPI_DATATYPE` [8].

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 is useful for passing structs with defined datatypes. In the case of trickier and more variable examples such as matrix subsets, explicit packing of elements into a contiguous buffer is also supported [8].

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` [8]. A data primitive buffer along with its length and datatype are sent 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 in this thesis.

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 [9]. The complexity of producing significant improvements, modifications, and/or changes in design to existing MPI implementations is a daunting task. In existing implementations, such as OpenMPI [4] and MPICH [5] middleware, with hundreds of thousands of lines of legacy code, substantially altering MPI is difficult for most researchers. Other middleware implementations are closed-source, precluding any way for most researchers to alter them. For instance, altering fundamental parts of MPI that manage internal state or concurrency would require far too much work. ExaMPI targets a smaller subset of the MPI standard than those middleware products, allowing it to focus on new functionality. This also avoids dead legacy code and technical debt associated with assumptions about node concurrency or progress made in the 1990s.

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

[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 has not progressed far as of this time.

- 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 the above by having a small scope and design. 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 on local or remote nodes [9].

Depending on the function, the request is specified to send, receive, broadcast, or 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, then either waited on within the binding (blocking communication) or asynchronous, non-blocking communication. In the latter, the function can be exited and a function `MPI_Wait` can be used later to wait on the message [9, 8].

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`, `mpicpp`, etc). At run-time, a program `mpirun` is used which is passed MPI parameters and coordinates background information such as threading. In ExaMPI, this `mpirun` consists of Python Dæmons [8, 9].

The main version of ExaMPI targets the same traditional C functions as all other MPI implementations. MPI is a large standard with hundreds of function (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 thesis.

2.3 Kokkos, its data structures and model

Kokkos is a library and programming model for C++ that provides data structures, concurrency features and algorithms to support advanced C++ parallel programming across different memory spaces [11]. Created at Sandia National Laboratories with funds form the DOE exascale computing project, 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 ARM or Intel CPUs, GPUs, and other accelerators are incredibly powerful. Because each can be a different paradigm of programming and different systems may only have some of these, developers can have a difficult time leveraging these resources efficiently [11].

More accurately, Kokkos is a programming model in the shape of a library instead of directives [11]. It is designed to work with other libraries (e.g., OpenMP) and can be used as the basis for sparse kernel features in projects such as Trillinos [11, 10]. Kokkos's sister library, MDSpan, offers similar functionality with a simpler, less template-centric interface [6]. As it is also a project at Sandia National Laboratory and shares team members, it shares much of the same code with Kokkos. It has currently been accepted into the C++ standards as Proposal P0009R9 [6].

2.3.1 The View Data Structure

The primary feature of Kokkos discussed in this thesis is an implementation of the View data structure, a datatype similar to a tensor, which handles multidimensional arrays (up to eight dimensions in Kokkos) [11, 1]. 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 [11, 3]. 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 [11, 3].

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 [3, 11].

2.3.2 View Syntax

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

Listing 2.1: Kokkos View Creation

The following section discusses the initialization of four basic Views as shown in Listing 2.1. Similar to ExaMPI, there are Kokkos::initialize and Kokkos::finalize functions to outline the Kokkos portion of the code[11, 3] In the first constructor, View A is created with template parameters within angle brackets alongside parentheses parameters, the label string and the length. Int is the datatype for the underlying data, 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. Note that 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 [11, 3].

View C in Listing 2.1 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 D points to, creating what is known as an Unmanaged View (as opposed to Managed Views) [11]. 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 [11, 3]. 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 [11, 3].

2.3.3 Memory Space

Another one of the key features of Kokkos is its ability to declare which kind of memory and execution spaces a View is in. For example, if a GPU is available, then a View can be declared

as follows $View < int*, GPUExecSpace, GPUMemSpace > A$ [11]. This is key to some of the ideas discussed later in this thesis.

2.3.4 Dispatch Operations

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

Listing 2.2: Kokkos View parallel for example

Kokkos has several parallel dispatch operations similar to those used in OpenMP: `parallel_for`, `parallel_reduce`, and `parallel_scan`. The primary dispatch operation used in this thesis is the `parallel_for` which is used for iterating 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 [3, 11]. This lambda is a more modern language feature (it is only introduced in newer versions of C++). As seen in Listing 2.2, The `parallel_for` and lambda are normally formatted to resemble a single for loop, but the lambda contains the actual code to be executed [2, 3, 11]. While these lambdas are more limiting than OpenMP equivalents, they are focused on their primary job, which is iterating and manipulating Views.

2.4 Previous Kokkos+MPI versions

Despite their popularity, there is not a framework that programmers can utilize that mixes the two. There does exist quite a bit of work that uses both libraries at the same time.

Previous work combining both Kokkos and traditional versions of MPI has yielded interesting results. For example, Khuvis et al. [7] have a shown a speedup of General Matrix Multiplication (GEMM) code and the Graph500 benchmark using their version of MPI+Kokkos. The tests used in Khuvis et al. use the Intel implementation of MPI, MVAPICH and standard Kokkos. The GEMM code uses 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 [7]. The Graph500 code actually has three implementations: one that

uses MPI only, one with MPI and Kokkos that uses locking operations, and another MPI and Kokkos implementation without locks. The Graph500 results show a speed-up with the locking implementation over the MPI-only one up to forty processes, and continued speed-up with the non-locking implementation of up to 5x on 64 processes.

```
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 2.3: Kokkos View Send

There exists an implementation of the lattice Boltzman method fluid flow solver for distributed CPUs/GPUs that uses both MPI and Kokkos [13]. In the Kokkos documentation itself, there exists a halo exchange which uses both MPI and Kokkos [11]. A halo exchange is when data (in this case, part or all of a View) is exchanged around the Communicator group of MPI processes. The listing 2.3 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 [11].

The primary innovation that distinguishes this project from previous forms of MPI+Kokkos interaction is the ability to take Kokkos Views as native first class objects. While there are plenty of programs that use both Kokkos and MPI, these are only interactions not frameworks that actually bind them together. The work highlighted in this thesis also lends itself to examine the tricky subject of the future of the MPI_Datatype. Specifically, this thesis examines whether and how to replace it with templates or possibly automatic type inference.

2.5 Summary

This chapter has shown some of MPI's history, followed by an overview of how MPI programs run and their basic functions. Then this chapter covers the ExaMPI implementation. This includes ExaMPI's basic philosophy and a technical overview. Then, this chapter presented an overview of Kokkos along with its fundamental data structures, structures across memory spaces, and its parallel dispatch operations. Finally, it showcased previous work that utilized both Kokkos and MPI, showing the programs that utilize both libraries still make concessions to be able to use both.]

CHAPTER 3

METHODOLOGY

This chapter covers the requirements for the MPI extension covered in this thesis. This framework utilizes the MPI and Kokkos library features but seeks to avoid the issues of previous MPI+Kokkos systems. This chapter lays out how the requirements are based in abstractions built-in to Kokkos (e.g., View layouts) and binding specifics.

3.1 Requirements

This project began with the conceptualization of one goal: alter MPI functions so that their function bindings should handle Views directly as parameters. To restate this as a requirement, create a series of functions based on MPI functions where any binding should be able to handle a Kokkos View as input or output in a similar way to buffers in MPI. Here, this means that an object can be passed to a function and used similarly to a traditional data buffer in MPI.

To use a View in this manner before this work, one would either have to use a work-around with the `.data()` method, or use the View as a derived datatype. The `.data()` method of a View allows access to the underlying array, but this is discouraged as it can cause memory leaking and overwriting. Since functions execute the same every time, they can better handle data in a repeatable way. Furthermore, a library extension is likely to have a larger number of eyes on any bugs than individual programs to catch dangling references.

Due to the large amount of possible permutations of dimensions, data primitives, etc. and the desire to show results in a reasonable time frame, the following requirements were laid out. The MPI extension functions for this thesis were restricted to handling up to 3 dimensional Views. The MPI extension functions should be able to handle int and double datatypes. The previous two requirements should not prevent additional dimensions or data primitives from being implemented. These are minimum requirements designed to focus the thesis rather than an end goal.

The derived datatype method allows MPI programmers to add new datatypes for MPI to use [8]. As this is based on the size of an object, it would have to be redone for Views with different datatypes and sizes. Thus, another requirement was created, the bindings should not require the user to use the `.data()` method or repeatedly create new derived datatypes for Views.

3.1.1 Layouts and Contiguous Views

One of the main goals of Kokkos is to handle a number of layouts, namely C-style row-major "right" layout and Fortran-style column-major "left" layout [11]. Additionally, matters are further complicated by Kokkos allowing users to define their own tiled layouts and the fact that either of these layout styles may or may not be contiguous. Kokkos works around this only caring about the distance between elements (or stride), as well as featuring support for these via extent and stride methods [11]. Therefore, a fundamental requirement of the implementation is to handle these without the user having to specify additional information. This streamlines the experience for developers.

In cases where layouts are contiguous, it does not matter how the data is transferred as long as it is transferred in the proper order. So long as the user ensures that both the sender and receiver Views have the same layout, the data will be transferred accurately. The ExaMPI extension will handle data transfer while ignoring layout specifics where possible (e.g., contiguous layouts) and where not possible, act according to the stride and datatype information. Therefore the requirement is as follows: the bindings should be able to handle any contiguous View that meets the previous requirements, regardless of layout. Non-contiguous objects are supported in both MPI and Kokkos, but are less common and more complex. To add them to this project would require packing techniques and complicate the more common use cases, possibly to the detriment of their performance. Therefore, since this project is aiming to be a proof of concept with continuing development, this was pushed to future work.

3.1.2 Binding Syntax and Setup

The next step was deciding how the actual function bindings would look. Expanding on the earlier requirement that a user should not have to use the `.data()` method at all, Views would

act as buffers in functions. For example, in a new version of `MPI_Send`, rather than taking in const void *buf, the new bindings could take in `Kokkos::View<int*> buf` [8].

More radical bindings were discussed during the creation of this thesis. As mentioned in a previous footnote, there is talk of attempting a type-less MPI centered purely around memory buffers. Any attempt at creating a type-less MPI is a far bigger project than this thesis. So, Kokkos-Enhanced ExaMPI still relies on MPI datatypes for memory transfer on its lower levels and Kokkos uses template datatype parameters itself. There was the idea of returning the received buffer, however creating a return datatype would have caused additional problems for template instantiation. Additionally, as it diverges from the MPI standard, it would have been controversial. The desire to closely follow the existing MPI standard for these functions led to another requirement: for compatibility with Kokkos and MPI Datatypes, template parameters should be used where possible.

3.2 Summary

This chapter explains the thought process that went into this project's design. First, it identified the minimum requirements. These were as follows:

- Create an MPI Extension composed of a series of functions based on MPI functions where any binding should be able to handle a Kokkos View as input or output in a similar way to buffers in MPI.
- The interface should not require the user to touch the `.data()` method;
- The interface should not require the user to create new derived datatypes for Views where possible;
- For the purposes of this thesis, any bindings should be able to handle Views with at least 3 dimensions;
- For the purposes of this thesis, any bindings should be able to handle Views with either int or double as their underlying datatype;

- The previous two requirements should not prevent additional dimensions or data primitives from being implemented later;
- The bindings should be able to handle any contiguous View that meets the previous requirements regardless of layout;
- For purposes of compatibility with Kokkos and avoiding reliance on the `MPI_Datatype`, template parameters should be used where possible.

CHAPTER 4

IMPLEMENTATION

This chapter describes the MPI extension with the requirements identified in Chapter 3. The bindings included here focus on point-to-point and collection communication [8]. The `MPI_Kokkos_Bcast` and `MPI_Kokkos_Allgather` functions are the only collective communication functions. All others are point-to-point with the exception of `MPI_Kokkos_Get_Dims` which is a helper function.

4.1 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 includes a Kokkos in the middle, to identify them as variants of the traditional function.

1. MPI_Kokkos_Get_Dims
2. MPI_Kokkos_Send
3. MPI_Kokkos_Recv
4. MPI_Kokkos_Isend
5. MPI_Kokkos_Irecv
6. MPI_Kokkos_Bcast
7. MPI_Kokkos_Allgather
8. MPI_Kokkos_Recv_Dims
9. MPI_Kokkos_Irecv_Dims

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. A complete code listing of the functions is listed in Appendix A.

4.1.1 MPI_Kokkos_Get_Dims

MPI_Kokkos_Get_Dims⟨*View_t*⟩(std::vector⟨*int*⟩ & vec, View_t view)

TEMPL	View_t	View type
INOUT	vec	dimensions as a vector
IN	view	input View

The utility function `MPI_Kokkos_Get_Dims` returns every extent (size of each individual dimension) of the View as a single integer vector, vec. `MPI_Kokkos_Get_Dims` iterates over a View's extents and pushes them to a vector. For example, if a View A has dimensions [5, 6, 7], `MPI_Kokkos_Get_Dims` will ask for each extent, push it to a View until there are no more, then return a vector containing the integers [5, 6, 7] at its tail. The template parameter `View_t` ensures the function does not have to know the underlying datatype or dimensions. This was intended as a proof of concept that MPI and Kokkos could be compiled together in the manner needed for the rest of the project.

4.1.2 MPI_Kokkos_Send and MPI_Kokkos_Recv

MPI_Kokkos_Send⟨*View_t*, *Datatype*⟩(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⟨*View_t*, *Datatype*⟩(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

The next two bindings 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 traditional buffer and sends this over as an ExaMPI Payload.

All functions have the same template parameter, `View_t`, as `MPI_Kokkos_Get_Dims`. These two and all following functions also have, `Datatype`, which allows the user to pass the underlying datatype for the View.

`buf` is a parameter used to pass the address of the View, along with accessing the underlying data with the data() method. Input parameters, `count` and `datatype`, are used to figure how many elements and the size of each element respectively. This information is then used to traverse and figure the size of the underlying allocation.

The `dest` parameter is the rank index of the process that receives the message. The `tag` parameter is an optional id for messages. The `comm` parameter is a handle to the underlying communicator group that handles communication in a set of processes. Commonly, `MPI_COMM_WORLD` is used which oversees all processes in an instance.

`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. The source parameter is the rank of the sending process.

4.1.3 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 both MPI non-blocking calls, where the function call is returned before communication is finished. In MPIs with strong progress, such as ExaMPI, this allows further code execution to be run while the underlying communication happens between processes [8]. The functions are already called as well as exited and Payloads are already set-up; all that needs to happen is the communication. Note that input and output parameters could be written or read during the further code execution [8]. The two previous functions, `MPI_Kokkos_Send` and `MPI_Kokkos_Recv`, are blocking functions; they halt any further code execution until they are finished with communication [8].

In non-blocking functions, the function call is returned before communication is finished. While the communication and writing to the buffer happens in the background, further code can be executed. When a developer wants to tell if the communication is finished, they can query the new parameter, request, with functions, such as `MPI_Wait`. After this, the output parameters will be finalized and ready to use. MPI_Wait will halt further code execution until the request is completed.

Rather than creating new query functions, existing ones are used, easing the process of changing over other programs. For example, if an existing ExaMPI program is converted, any lines with `MPI_Wait` will not have to be changed. Note that only `MPI_Wait` has been used in the Chapter 5. Otherwise, this function differs from its non-Kokkos counterpart only in its ability to handle a Kokkos View as a first class object.

`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.

4.1.4 *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` short for MPI Broadcast. This function resembles the functionality of Send and Recv combined into one. Broadcast functions are classified as a form of collective communication as they handle several processes [8]. A singular root process sends (Broadcasts) its View to all the other process ranks in the given communicator group, enabling a process to send to any number of other processes [9, 8].

4.1.5 *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 `MPI_Kokkos_Allgather`, which takes (gathers) an input View from all processes then compiles them into a View with inputs from each View ordered by their sending process's index ranking. `MPI_Allgather` functions are classified as a form of collective communication as they handle several processes [8].

The first parameter, `buf`, contains the data to be sent. The other buffer parameter, `recv_buf`, contains the contents of the first buffer from each process in the comm group that used `MPI_Kokkos_Allgather`, placed in order by index rank. Unlike the previous functions, this process requires the creation of two payloads, one for sending and another for receiving.

4.1.6 MPI_Kokkos_Recv_Dims

MPI_Kokkos_Recv⟨*View_t*, *Datatype*⟩(View_t * buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, int* dims)

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
IN	dims	integer array containing the View's dimensions

This function receives the buffer of a Kokkos View, then creates a Kokkos View from an allocated View pointer. This is an older, alternate version of `MPI_Kokkos_Recv` that forces the

correct dimensions on the received View. Neither Kokkos nor its sister project, MD-Span, have a way for an existing multi-dimensional array to be passed to a new one with full debugging information [11, 6]. The primary way to replicate a View is to manually copy each element, use a copy function (e.g., `deep_copy()`), or use unmanaged Views; only unmanaged Views allow passing an existing array directly. In an unmanaged View, the underlying data buffer of the View is passed as a parameter along with the dimensional size information. Originally intended to allow a more flexible packing structure, it passes the received buffer to an unmanaged View. This means that `MPI_Kokkos_Recv_Dims` is not legal in C++, though it will compile on most compilers. This was discovered in discussions with the Kokkos dev team and slack channel. This code being illegal is due to the line, `new (&buf[0]) View_t(temp_buf, count);`, having an illegal pointer access. Other forms of constructors have been proposed but are not part of Kokkos currently. The output `buf` is also different; instead of being a pointer to a View where the underlying array will be written, it is an allocated but empty array of Views. `MPI_Kokkos_Recv_Dims` will fill the first index of this array with an unmanaged View.

The dims parameter is an array with the View's dimensions. If no dims are passed, it is assumed that it is one-dimensional and the count is passed. This dims is designed to take an integer array from `MPI_Kokkos_Get_Dims` for the dimensions.

4.1.7 MPI_Kokkos_Irecv_Dims

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

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
IN	dims	integer array containing the View's dimensions

This function is used to receive the View transported by the non-blocking `MPI_Kokkos_Isend`. As with `MPI_Kokkos_Recv_Dims`, `MPI_Kokkos_IRecv_Dim` has an optional dims parameter and creates an unmanaged View on a View pointer array. Like `MPI_Kokkos_IRecv`, it is non-blocking and other code can be executed while communication is completed.

There is no `MPI_Kokkos_Send_Dims`, as the dims parameter is only intended to enforce the correct dimensions on a View being created. It cannot alter the dimensions of existing Views after creation. Additionally, sending them alongside the View data would require additional packing or another send specific to the dims. Both of these would harm performance.

The dims technique was deprecated because this enforcement is not necessary on contiguous Views and in its present state, forces the use of unmanaged Views. See Chapter 6 for more details.

4.2 Miscellaneous

4.2.1 CMake and Binding Creation

This project is designed to integrate Kokkos with ExaMPI, a research-focused implementation of MPI-4 in C++ [9]. This project uses the more experimental C++ bindings, rather than traditional C or Fortran bindings, so the `mpicpp.h` header is used. Here, a templated version of `MPI_Send` can be created that takes the class of the buffer as a template parameter instead of just using `MPI_Datatype`. Without these C++ features, letting MPI handle Views would have been difficult if not impossible without sacrificing much of the utility of Views [9].

The first attempts at setting up a MPI+Kokkos framework required linking Kokkos to MPI during MPI compilation. This failed because ExaMPI must find the Kokkos library to compile the individual bindings correctly. ExaMPI uses the CMake build automation to assemble all of its code, third-party libraries, etc. Therefore, the CMake system must be able to find the Kokkos file either automatically or by being handed a file path.

The second stage required the Kokkos library to not just be compiled at the library level with Kokkos, but when any ExaMPI program that utilized Kokkos has to be compiled with Kokkos. For the first stage, Kokkos is set up as a third-party library. This involves creating FindKokkos file that sets up a few environmental variables (similar files were made for other third party libraries such as LibFabric or Nvidia tools). In the CMakeLists.txt file, a few new lines were added to properly handle the new environmental variables: `option(GET_KOKKOS_PATH ‘‘Obtain the kokkos path’’ OFF)`, an `add_subdirectory(kokkos)` directive, `find_package(Kokkos)`, `include_directories (${Kokkos_INCLUDE_DIRS_RET})`, and `target_link_libraries-(exampi Kokkos::kokkos)`.

After adding the Kokkos source directory to the ExaMPI directory, ExaMPI's CMake will execute Kokko's CMake file as a dependency and send the appropriate Kokkos header files to `/usr/local/include`. Finally, ExaMPI will be able to link the Kokkos header files to files that include them. After compiling ExaMPI, the ExaMPI compiler wrapper will link Kokkos to new programs.

4.2.2 Templates

A primary issue for function creation is that, unlike templates, MPI datatypes contain no direct information relating to the datatypes handled by the compiler. Instead, the MPI datatype is a guide for how MPI should process the given data and is very implementation-dependent. In the case of ExaMPI, MPI Datatype is a constant cast from an underlying Datatype class instance used by the lower levels. While this is a robust system that can handle packing and data conversion, there is no way to directly infer the data's actual datatype form the MPI Datatype. [8, 9]. This means that programmers will declare the datatype somewhere else to send it to the underlying Kokkos code.

Instead of trying to add a datatype to the MPI datatype lookup table, this thesis utilizes template parameters to supplement the existing parameters. There are two different template parameters used in the MPI extension. The first template parameter is View_t, which replaces a Kokkos View and is used in each function. The second is the underlying Datatype, which is used for preparing additional Kokkos and pointer code.

Inside the binding files, function prototypes for each combination of dimensions and underlying datatype are created. Otherwise, the compile will not create template logic for any function and will throw an error when compiling individual programs. As noted previously, to simplify this project, only the cases with up to 3 dimensions, and int and double datatypes were implemented for this thesis. The alternative, defining all bindings inside the header file `mpicpp.h`, made the header file much larger and harder to debug.

Radically different bindings, such as ones that return Views, were pushed to future work (See Ch. 6 for more details). This is due to their possibility for controversy, the lack of an existing interface to work from, and the fact that they would have added development time without a definite increase in functionality. This project, instead, decided to stick with having Views replace buffers in a traditional MPI interface.

CHAPTER 5

RESULTS

This chapter evaluates the performance the functions in the MPI extension, highlighting how they compare to existing methods of using MPI and Kokkos. First, its shows the results for a one-dimensional `MPI_Send` and `MPI_Recv` test that changed length from 64 to 32768 with the length multiplying by 2 at each step. For each length of the array, 101 runs were done. Then two similar tests were run for two and three dimensional arrays, as well as a different test using `MPI_Bcast`. All tests were run on the Epyc cluster at the UTC SimCenter [15]. Parameters such as the number of nodes or tasks vary based on the test being run. The code for each test can be seen in Appendix A.

5.1 Single-Dimension MPI_Send And MPI_Recv Tests

```
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 5.1: Kokkos View parallel for example

This section covers the tests comparing the new bindings for sending and receiving using a View (MPI_Kokkos_Send and MPI_Kokkos_Recv) and traditional methods for this. Figure 5.1 graphs the results for the runs for two MPI ping-pong tests, one with the new MPI extension bindings from Chapter 4 and another with the older traditional bindings. A ping-pong tests is

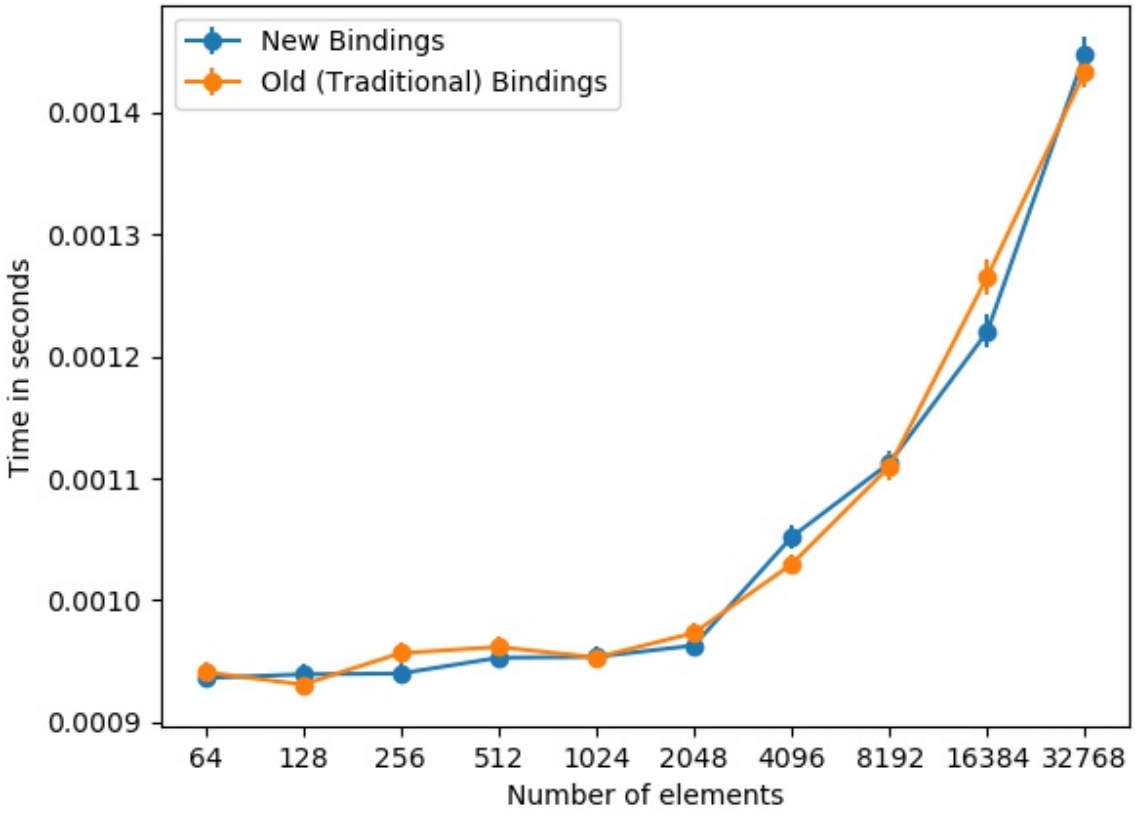

Figure 5.1 New MPI Extension bindings vs traditional method averages for pingpong tests

where one process sends a buffer (here a View), another receives and alters the buffer. Finally, the buffer is sent back to the original sending process.

The primary difference between the two is shown in Listing 5.1, where the old method for receiving an array must touch the `.data()` method directly. These tests were run on a node of the EPYC cluster at the UTC Simcenter with full access to two nodes with two processes each and up to three cores available per process [15].

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 multiplying by 2 at each step. For each length of the array, 101 runs were done with each point in Figure 5.1 being the average time. The times recorded in the figure are from the original sending side as it will have the longest overall time. The standard deviation of the mean is plotted as error bars. The full code listings for all tests are shown in A.

In Figure 5.1, which bindings has better performance varies on almost every point. This is likely due to the run-time environment and access to resources rather than the code itself as there are no specific handling cases in the added code. Overall, these results seem to be comparable for both methods with both types of bindings being better at different times and a very low standard deviation of the mean. Since the primary goal of this thesis was not a significant change in performance but roughly equal performance for each method, this means that the bindings performed well by this thesis's metrics.

5.2 Multi-Dimension Send And Recv Tests

This series of tests was run on both two and three-dimensional Views, but their code is otherwise identical to the single dimensional Views unless otherwise specified. The slurm configuration for both the two and three dimensional tests are the same as the one-dimensional Views. For each of these, the x-axis number is the length of one dimension, so the 2 is actually $2 * 2 = 4$ in the two dimensional case and $2 * 2 * 2 = 8$ in the three dimensional.

For both graphs, the new bindings and old bindings are roughly equal for time and standard error. The primary outliers are for the two dimensional, where n=8, new bindings perform slightly worse, and at n=512 they perform slightly better. For the three dimensional case and unlike

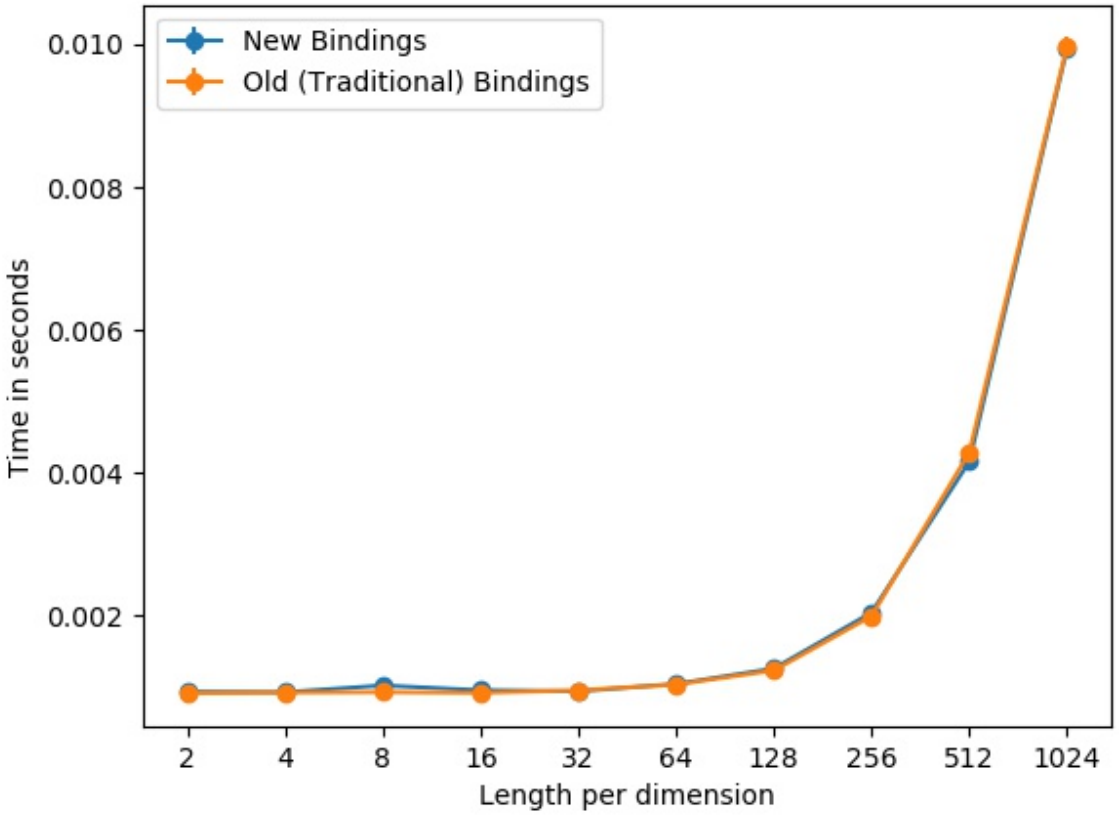

Figure 5.2 2D MPI Extension bindings vs. traditional method averages pingpong tests

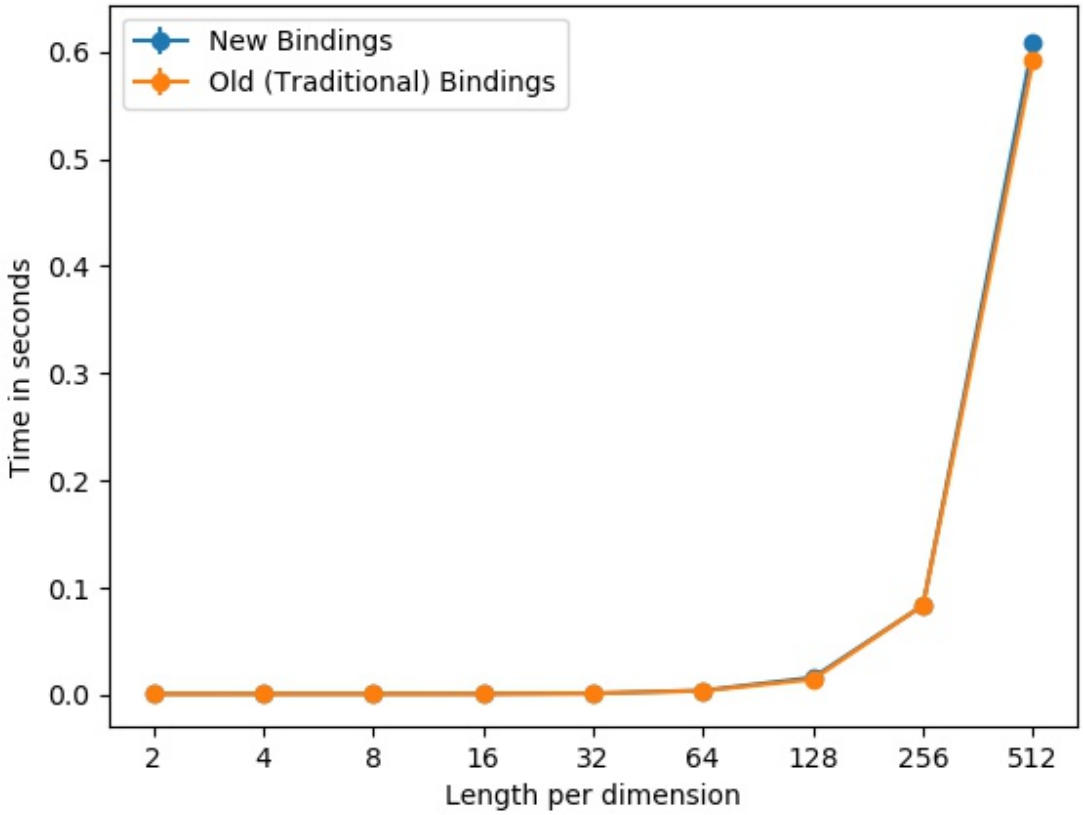

Figure 5.3 3D MPI Extension bindings vs. traditional method averages pingpong tests

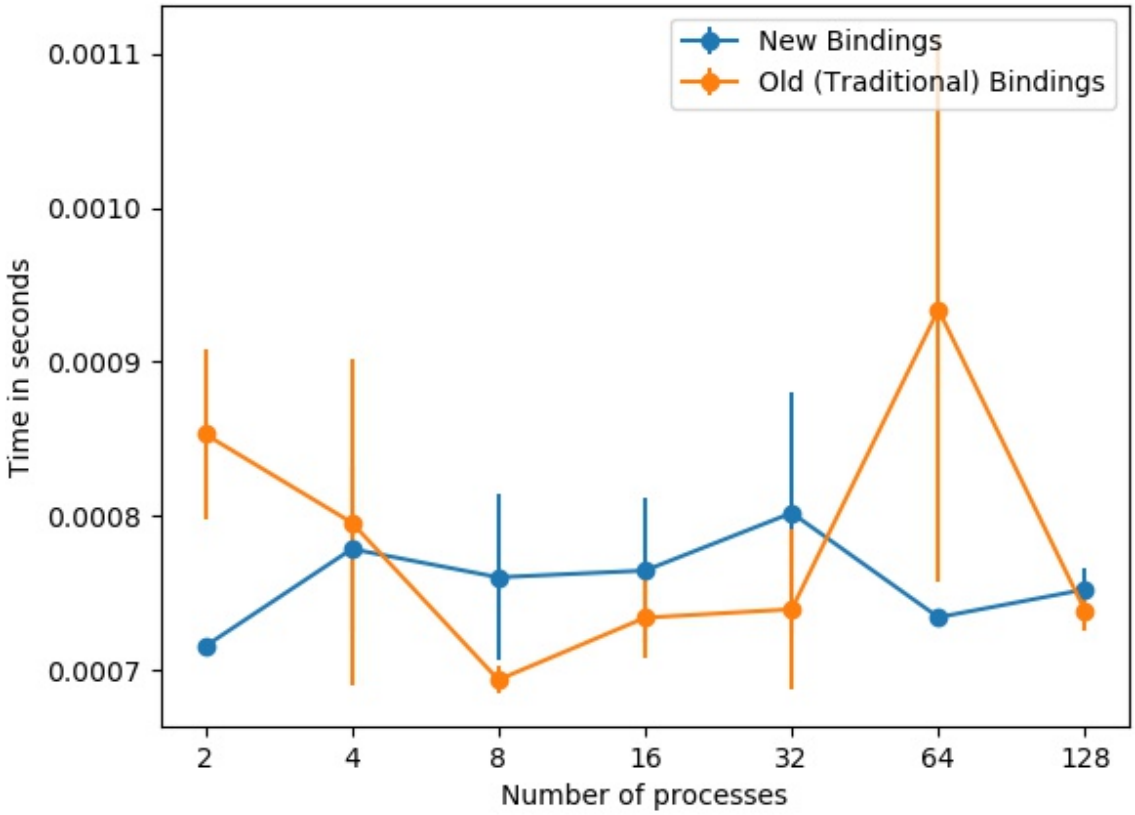

Figure 5.4 MPI Extension bindings vs. traditional method averages

the other tests, new bindings do perform slightly worse for n=512. This might be an outlier or indicative of a larger trend and while 512^3 is a large number, it is no where near the highest size possible for a Kokkos View. Otherwise, both bindings perform identically.

5.3 MPI_Kokkos_Broadcast Tests

This test runs a `MPI_Bcast` test with both the new Kokkos bindings and without. In this series, the Epyc configuration was run with 2 nodes and 64 tasks. It does not run a pingpong, instead transporting a single one dimensional View with 5000 elements using the `MPI_Bcast` command with a varying number of processes (2 to 128). The time it takes to run this is recorded for every process then the maximum is found using the `MPI_Allreduce` function. Each of these

tests is run 100 times for a given number of processes then the first time is discarded for warm-up, leaving 99 run times for the average.

As shown in Figure 5.4, the average time and standard error varies wildly for both the new bindings and traditional bindings. This 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. As shown in Figure 5.4, this does not even necessarily even increase linearly as one might think with the increasing number of processes. Overall, this series of tests is less conclusive than previous ones by having no cohesive trends.

CHAPTER 6

CONCLUSIONS

This thesis set about to integrate two programming models, MPI and Kokkos, better than previous attempts. Kokkos was chosen due to its unique data structures, but no one had yet created an interface for interaction between its data structures and MPI. The ExaMPI implementation of MPI was utilized due to it being written in C++, which enabled the use of templates that would not have been possible in many other implementations. After introducing these models, a few existing works that utilize both MPI and Kokkos are highlighted to show that demand exists for this work.

Then, a few guidelines for this thesis were decided, namely 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. During the course of this thesis, several MPI bindings were implemented to use Kokkos View objects as their primary buffers alongside template parameters for internal use.

After implementing the bindings for this project, this thesis found that the new bindings' performed similarly to the old bindings' with almost no loss of performance. 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 significantly. Unlike previous programs that used both MPI and Kokkos, this thesis was able to implement several MPI bindings using Kokkos View objects without significant loss of performance. These bindings also preserve the C++ nature of the Kokkos View by allowing templates.

6.1 Future Work

Going forward, this work will encompass a wider array of standard MPI functions and more Kokkos-specific functions, with work beginning on functions, such as All-To-All, Scatter

and Gather, in the near future. Another future goal of this project is more device specific support (i.e., `MPI_Send<View, class, Device>`) for GPUs, FPGAs, and other hardware.

Currently, the Kokkos MPI bindings closely follow existing MPI functions with their primary difference being the introduction of View template parameters. Other alternatives, notably bindings that return Views directly, complicated the fundamental goal of extending the MPI interface for Kokkos without performance benefit. Returning the Views directly rather than `MPI_Success` codes upon completion is desirable and allows less issues with View declaration on the user end.

The major use-case not covered by the MPI extension covered in this thesis is non-contiguous Views. Generally, non-contigous 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 of 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.

The primary path forward is to create a separate transport back-end for the non-contiguous case. Kokkos provides a `is_contigous` method for this, along with a stride interface for knowing the spaces between interfaces (strides) [11]. The back-end would use the stride information to send chunks of the View in parallel for non-contiguous View. This would have improved performance over the more general case of manually packing the Views.

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 would be more flexible for Views as it could just take in memory size or strides.

BIBLIOGRAPHY

[1] Andres, B., Köthe, U., Kröger, T., and Hamprecht, F. A. (2010). Runtime-flexible multi-dimensional arrays and views for C++98 and c++0x. *CoRR*, abs/1008.2909. 9

[2] Dagum, L. and Menon, R. (1998). Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55. 12

[3] Edwards, H. C. and Trott, C. R. (2013). Kokkos: Enabling performance portability across manycore architectures. In *2013 Extreme Scaling Workshop (xsw 2013)*, pages 18–24. 2, 10, 11, 12

[4] Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R. H., Daniel, D. J., Graham, R. L., and Woodall, T. S. (2004). Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary. 6, 7

[5] Gropp, W., Lusk, E., and Skjellum, A. (2014). *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press. 7

[6] Hollman, D. S., Adelstein-Lelbach, B., Edwards, H. C., Hoemmen, M., Sunderland, D., and Trott, C. R. (2020). mdspan in C++: A case study in the integration of performance portable features into international language standards. *CoRR*, abs/2010.06474. 9, 26

[7] Khuvis, S., Tomko, K., Hashmi, J., and Panda, D. K. (2020). Exploring hybrid mpi+kokkos tasks programming model. In *2020 IEEE/ACM 3rd Annual Parallel Applications Workshop: Alternatives To MPI+X (PAW-ATM)*, pages 66–73. 2, 12

[8] Message Passing Interface Forum (2021). *MPI: A Message-Passing Interface Standard Version 4.0*. 1, 2, 6, 7, 8, 16, 17, 19, 23, 24, 25, 28

[9] Skjellum, A., Rüfenacht, M., Sultana, N., Schafer, D., Laguna, I., and Mohror, K. (2020). Exampi: A modern design and implementation to accelerate message passing interface innovation. In Crespo-Mariño, J. L. and Meneses-Rojas, E., editors, *High Performance Computing*, pages 153–169, Cham. Springer International Publishing. 1, 2, 7, 8, 24, 27, 28

[10] Trilinos Project Team, T. (2020 (acccessed May 22, 2020)). *The Trilinos Project Website*. 9

[11] Trott, C. R., Lebrun-Grandié, D., Arndt, D., Ciesko, J., Dang, V., Ellingwood, N., Gayatri, R., Harvey, E., Hollman, D. S., Ibanez, D., Liber, N., Madsen, J., Miles, J., Poliakoff, D., Powell, A., Rajamanickam, S., Simberg, M., Sunderland, D., Turcksin, B., and Wilke, J. (2022). Kokkos 3: Programming model extensions for the exascale era. *IEEE Transactions on Parallel and Distributed Systems*, 33(4):805–817. 1, 2, 9, 10, 11, 12, 13, 16, 26, 38

[12] Trott, C. R., Plimpton, S. J., and Thompson, A. P. (2017). Solving the performance portability issue with kokkos. 3

[13] Vasyivy (2023). lbm2d-mpi-kokkos. Accessed: 2022-11-12. 13

[14] Waters, D., MacLean, C. A., Bonachea, D., and Hargrove, P. (2021). Demonstrating upc++/kokkos interoperability in a heat conduction simulation (extended abstract). 3

[15] Wiki, S. (2023). Epyc cluster. Accessed: 2023-2-13. 30, 32

APPENDIX A

KOKKOS/EXAMPI BINDING CODE

A Bindings Code

A.1 MPI_Kokkos_Send

```
#include <vector>
#include "debug.h"
#include "engines/blockingprogress.h"
#include "requests/request.h"
#include "mpicpp.h"
#include <Kokkos_Core.hpp>
template<class View_t, class Datatype>
int MPI_Kokkos_Send(View_t * buf,
                    int count,
                    MPI_Datatype datatype,
                    int dest,
                    int tag,
                    MPI_Comm comm)
{
        using namespace exampi;
        std::shared_ptr<PayloadOrganizer> under_payload
            =
                std::make_shared<PayloadOrganizer>(
                   Payload(buf->data(), count, datatype
                   ));
        std::shared_ptr<Request> temp_req =
            std::make_shared<Request>(under_payload,
                                      USE_MY_RANK,
                                      dest,
                                      tag,
```

```
                                                comm,
                                                Operation::SEND
                                                    ,
                                                nullptr,
                                                false,
                                                false,
                                                Op::OP_NULL);
        temp_req->activate();
        auto pe = comm->get_progress();
        pe->post_request(temp_req);
        pe->wait_for_complete(temp_req);
        return MPI_SUCCESS;
}
```

A.2 MPI_Kokkos_Recv

```
#include <vector>
#include "debug.h"
#include "engines/blockingprogress.h"
#include "requests/request.h"
#include "mpicpp.h"
#include <Kokkos_Core.hpp>
template<class View_t, class Datatype>
int MPI_Kokkos_Recv(View_t *buf,
                    int count,
                    MPI_Datatype datatype,
                    int source,
                    int tag,
                    MPI_Comm comm)
```

```
{
        using namespace exampi;
        // This buffer is what the payload actually
           uses from given View
        Datatype* buf_data = buf->data();
        std::shared_ptr<PayloadOrganizer> under_payload
            =
                std::make_shared<PayloadOrganizer>(
                   Payload(buf_data, count, datatype));
        std::shared_ptr<Request> temp_req =
        std::make_shared<Request>(nullptr,
                                  source,
                                  USE_MY_RANK,
                                  tag,
                                  comm,
                                  Operation::RECEIVE,
                                  under_payload,
                                  false,
                                  false,
                                  Op::OP_NULL);
        temp_req->activate();
        auto pe = comm->get_progress();
        pe->post_request(temp_req);
        pe->wait_for_complete(temp_req);

        return MPI_SUCCESS;
}
```

A.3 MPI_Kokkos_ISend

```
template<class View_t, class Datatype>
int MPI_Kokkos_Isend(View_t * buf,
                     int count,
                     MPI_Datatype datatype,
                     int          dest,
                     int          tag,
                     MPI_Comm comm,
                     MPI_Request *request)
{
        using namespace exampi;
        std::shared_ptr<PayloadOrganizer> under_payload
            =
                std::make_shared<PayloadOrganizer>(
                   Payload(buf->data(), count, datatype
                   ));
        *request = std::make_shared<Request>(
           under_payload,
                                              USE_MY_RANK,
                                              dest,
                                              tag,
                                              comm,
                                              Operation::SEND
                                                  ,
                                              nullptr,
                                              false,
                                              false,
                                              Op::OP_NULL);
```

```
        return MPI_Start(request);
}
```

A.4 MPI_Kokkos_Irecv

```
template<class View_t, class Datatype>
int MPI_Kokkos_Irecv(View_t *        buf,
                     int             count,
                     MPI_Datatype    datatype,
                     int             source,
                     int             tag,
                     MPI_Comm        comm,
                     MPI_Request *request)
{
        using namespace exampi;
        /* NOTE: this is legal c++ unlike above
         */
        Datatype* buf_data = buf->data();
        std::shared_ptr<PayloadOrganizer> under_payload
            =
             std::make_shared<PayloadOrganizer>(Payload(
                buf_data, count, datatype));

        *request = std::make_shared<Request>(nullptr,
                                             source,
                                             USE_MY_RANK,
                                             tag,
                                             comm,
```

```
                                                    Operation::
                                                        RECEIVE,
                                                    under_payload,
                                                    false,
                                                    false,
                                                    Op::OP_NULL);
        return MPI_Start(request);
}
```

A.5 MPI_Kokkos_Bcast

```
template<class View_t, class Datatype>
int MPI_Kokkos_Bcast(View_t * buf,
                     int count,
                     MPI_Datatype datatype,
                     int          root,
                     MPI_Comm comm)
{
        using namespace exampi;
        std::shared_ptr<PayloadOrganizer> under_payload
           ;

        std::shared_ptr<Request> temp_req;

        if(root == comm->get_rank()){
          // send side
          under_payload = std::make_shared<
             PayloadOrganizer>(Payload(buf->data(),
             count, datatype));
```

```
            temp_req = std::make_shared<Request>(
                under_payload,
        root,
        USE_MY_RANK,
        COLLECTIVE_TAG,
        comm,
        Operation::BCAST,
        nullptr,
        false,
        true,
        Op::OP_NULL);
            temp_req->activate();
            auto pe = comm->get_progress();
            pe->post_request(temp_req);
            pe->wait_for_complete(temp_req);

            return MPI_SUCCESS;
}
else{
  // recv side
  under_payload = std::make_shared<
     PayloadOrganizer>(Payload(buf->data(),
     count, datatype));
            temp_req = std::make_shared<Request>(
                nullptr,
    root,
    USE_MY_RANK,
    COLLECTIVE_TAG,
```

```
            comm,
            Operation::BCAST,
            under_payload,
            false,
            true,
            Op::OP_NULL);
                temp_req->activate();
                auto pe = comm->get_progress();
                pe->post_request(temp_req);
                pe->wait_for_complete(temp_req);

                return MPI_SUCCESS;
        }

}
```

A.6 MPI_Kokkos_Allgather

```
template<class View_t, class Datatype>
int MPI_Kokkos_Allgather(View_t * buf,
                         int          count,
                         MPI_Datatype datatype,
                         View_t *       recv_buf,
                         int          recv_count,
                         MPI_Datatype recv_type,
                         MPI_Comm     comm)
{
        using namespace exampi;
        std::shared_ptr<PayloadOrganizer> s_payload =
```

```
        std::make_shared<PayloadOrganizer>(Payload(
            buf->data(), count, datatype));
    Datatype *buf_data = recv_buf->data();
    std::shared_ptr<PayloadOrganizer> r_payload =
        std::make_shared<PayloadOrganizer>(
            buf_data, recv_count, recv_type, comm->
                get_size());

    std::shared_ptr<Request> temp_req =
        std::make_shared<Request>(s_payload,
                                  USE_MY_RANK,
                                  0,
                                  COLLECTIVE_TAG,
                                  comm,
                                  Operation::
                                      ALLGATHER,
                                  r_payload,
                                  false,
                                  true,
                                  Op::OP_NULL);

    temp_req->activate();
    auto pe = comm->get_progress();
    pe->post_request(temp_req);
    pe->wait_for_complete(temp_req);

    return MPI_SUCCESS;
}
```

A.7 MPI_Kokkos_Recv_Dims

```
template<class View_t, class Datatype>
int MPI_Kokkos_Recv_Dims(View_t * buf,
                         int count,
                         MPI_Datatype datatype,
                         int source,
                         int tag,
                         MPI_Comm comm,
                         int* dims)
{
    using namespace exampi;
    // This buffer is what the payload actually uses,
       this is then put in an unmanaged View
    Datatype * temp_buf=(Datatype *)malloc(count*sizeof
       (Datatype));
    std::shared_ptr<PayloadOrganizer> under_payload =
        std::make_shared<PayloadOrganizer>(Payload(
           temp_buf, count, datatype));
    std::shared_ptr<Request> temp_req =
        std::make_shared<Request>(nullptr,
            source,
            USE_MY_RANK,
            tag,
            comm,
            Operation::RECEIVE,
            under_payload,
            false,
            false,
```

```
    Op::OP_NULL);
temp_req->activate();
auto pe = comm->get_progress();
pe->post_request(temp_req);
pe->wait_for_complete(temp_req);
/* NOTE: this is technically not legal C++ (it
   is legal C) thus is allowed on almost any
   compiler
   As per Kokkos slack, the only alternative (
      currently, they are working on it) is to
   manually copy everything which is very much
      not performant.
 */
switch((int) View_t::rank){
case 1:
        new (&buf[0]) View_t(temp_buf, count);
        break;
case 2:
        new (&buf[0]) View_t(temp_buf, dims[0],
           dims[1]);
        break;
case 3:
        new (&buf[0]) View_t(temp_buf, dims[0],
            dims[1], dims[2]);
        break;
default:
        debug("PROBLEM: Too many or not enough
           dimensions in View");
```

```
            //MPI_ABORT(comm, 600);
        }

        return MPI_SUCCESS;
}
```

A.8 MPI_Kokkos_Irecv_Dims

```
template<class View_t, class Datatype>
int MPI_Kokkos_Irecv_Dims(View_t *       buf,
                          int            count,
                          MPI_Datatype   datatype,
                          int            source,
                          int            tag,
                          MPI_Comm       comm,
                          MPI_Request *request,
                          int* dims)
{
        using namespace exampi;
        Datatype * temp_buf = (Datatype *) malloc(count
           *sizeof(Datatype));
        std::shared_ptr<PayloadOrganizer> under_payload
            =
             std::make_shared<PayloadOrganizer>(Payload(
                temp_buf, count, datatype));

        *request = std::make_shared<Request>(nullptr,
                                             source,
                                             USE_MY_RANK,
```

```
                                        tag,
                                        comm,
                                        Operation::
                                            RECEIVE,
                                        under_payload,
                                        false,
                                        false,
                                        Op::OP_NULL);
/* NOTE: this is technically not legal C++ (it
   is legal C) thus is allowed on almost any
   compiler
   As per Kokkos slack, the only alternative (
      currently, they are working on it) is to
   manually copy everything which is very much
      not performant.
 */
switch((int) View_t::rank){
case 1:
        new (&buf[0]) View_t(temp_buf, count);
        break;
case 2:
        new (&buf[0]) View_t(temp_buf, dims[0],
           dims[1]);
        break;
case 3:
        new (&buf[0]) View_t(temp_buf, dims[0],
            dims[1], dims[2]);
        break;
```

```
        default:
                debug("PROBLEM: Too many or not enough
                    dimensions in View");
                //MPI_ABORT(comm, 600);
        }

        return MPI_Start(request);
}
```

B Test Code

B.1 New Bindings Kokkos Pingpong Test

```
// ### processes 2
// ### labels base, short

#include <mpicpp.h>
#include <Kokkos_Core.hpp>
#include <iostream>
#include <chrono>

int main(int argc, char** argv) {
        int error = 0;

        MPI_Init(&argc, &argv);
        Kokkos::initialize( argc, argv );
        {
        int rank = -1;
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        //int n = 8;
```

```
int n = std::stoi(argv[argc-1]);
Kokkos::View<int*> check("check",n);
Kokkos::parallel_for(check.extent(0),
   KOKKOS_LAMBDA(int i){
                check(i) = i*i;
                //check(i,1) = (i*i)+1;
                //std::cout << i << "th entry:
                   " << check[i] << std::endl;
        });

if(rank == 1){
        auto begin = std::chrono::steady_clock
           ::now();
        // MPI Send time
        MPI_Kokkos_Send<Kokkos::View<int*>, int
           >(&check, check.size(), MPI_INT, 0,
           0, MPI_COMM_WORLD);
        Kokkos::View<int*> A("A side", n);
        MPI_Kokkos_Recv<Kokkos::View<int*>, int
           >(&A, check.size(), MPI_INT, 0, 0,
           MPI_COMM_WORLD);

        auto end = std::chrono::steady_clock::
           now();
        std::chrono::duration<double>
           elapsed_seconds = end-begin;
```

```
        std::cout << "Send Time Count: " <<
            elapsed_seconds.count() << std::endl
            ;
}
else{
        auto begin = std::chrono::steady_clock
            ::now();
        // MPI Recv time
        Kokkos::View<int*> B("B side", n);
        MPI_Kokkos_Recv<Kokkos::View<int*>, int
            >(&B, check.size(), MPI_INT, 1, 0,
            MPI_COMM_WORLD);
        B(0)+=200;
        MPI_Kokkos_Send<Kokkos::View<int*>, int
            >(&B, B.size(), MPI_INT, 1, 0,
            MPI_COMM_WORLD);

        auto end = std::chrono::steady_clock::
            now();
        std::chrono::duration<double>
            elapsed_seconds = end-begin;
        std::cout << "Recv Time Count: " <<
            elapsed_seconds.count() << std::endl
            ;
}
}
Kokkos::finalize();
MPI_Finalize();
```

```
        return error;
}
```

B.2 Old Method Kokkos Pingpong Test

```
// ### processes 2
// ### labels base, short

#include <mpicpp.h>
#include <Kokkos_Core.hpp>
#include <iostream>
#include <chrono>

int main(int argc, char** argv) {
        int error = 0;

        MPI_Init(&argc, &argv);
        Kokkos::initialize( argc, argv );
        {
        int rank = -1;
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        //int n = 8;
        int n = std::stoi(argv[argc-1]);
        Kokkos::View<int*> check("check",n);//[4][3]
        // randomize?
        Kokkos::parallel_for(check.extent(0),
           KOKKOS_LAMBDA(int i){
                        check(i) = i*i;
```

```
        });
if(rank == 1){
        auto begin = std::chrono::steady_clock
            ::now();
        // MPI Send time
        MPI_Send(check.data(), check.size(),
            MPI_INT, 0, 0, MPI_COMM_WORLD);
        int * send_buf = (int *) malloc(check.
            size() * sizeof(int));
        MPI_Recv(send_buf, check.size(),
            MPI_INT, 0, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
        Kokkos::View<int *> rank1_check(
            send_buf, check.size());
        // time
        auto end = std::chrono::steady_clock::
            now();
        std::chrono::duration<double>
            elapsed_seconds = end-begin;
        std::cout << "Send Time Count: " <<
            elapsed_seconds.count() << std::endl
            ;
}
else{
        auto begin = std::chrono::steady_clock
            ::now();
        // MPI Recv time
```

```
                int * recv_buf = (int *) malloc(check.
                    size() * sizeof(int));
                MPI_Recv(recv_buf, check.size(),
                    MPI_INT, 1, 0, MPI_COMM_WORLD,
                    MPI_STATUS_IGNORE);
                Kokkos::View<int *> recv_check(recv_buf
                    , check.size());
                recv_check(0) += 200;
                MPI_Send(recv_check.data(), recv_check.
                    size(), MPI_INT, 1, 0,
                    MPI_COMM_WORLD);
                // time
                auto end = std::chrono::steady_clock::
                    now();
                std::chrono::duration<double>
                    elapsed_seconds = end-begin;
                std::cout << "Recv Time Count: " <<
                    elapsed_seconds.count() << std::endl
                    ;
        }
        }
        Kokkos::finalize();
        MPI_Finalize();

        return error;
}
```

B.3 Two Dimensional New Bindings Kokkos Pingpong Test

```
#include <mpicpp.h>
#include <Kokkos_Core.hpp>
#include <iostream>
#include <chrono>

int main(int argc, char** argv) {
        int error = 0;

        MPI_Init(&argc, &argv);
        Kokkos::initialize( argc, argv );
        {
        int rank = -1;
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        //int n = 8;
        int n = std::stoi(argv[argc-1]);
        Kokkos::View<int**> check("check",n, n);
        Kokkos::parallel_for(check.extent(0),
           KOKKOS_LAMBDA(int i){
             for(int j = 0; j<check.extent(1); j++){
               check(i,j) = (i+1)*(j+1);
               //check(i,1) = (i*i)+1;
               //std::cout << i << "th entry: " << check
                  [i] << std::endl;
             }
           });

        if(rank == 1){
```

```
            auto begin = std::chrono::steady_clock
                ::now();
            // MPI Send time
            MPI_Kokkos_Send<Kokkos::View<int**>,
                int>(&check, check.size(), MPI_INT,
                0, 0, MPI_COMM_WORLD);
            //Kokkos::View<int**> A("A side", n, n)
                ;
            MPI_Kokkos_Recv<Kokkos::View<int**>,
                int>(&check, check.size(), MPI_INT,
                0, 0, MPI_COMM_WORLD);

            auto end = std::chrono::steady_clock::
                now();
            std::chrono::duration<double>
                elapsed_seconds = end-begin;
            std::cout << "Send Time Count: " <<
                elapsed_seconds.count() << std::endl
                ;
}
else{
            auto begin = std::chrono::steady_clock
                ::now();
            // MPI Recv time
            //Kokkos::View<int**> B("B side", n, n)
                ;
```

```
                MPI_Kokkos_Recv<Kokkos::View<int**>,
                    int>(&check, check.size(), MPI_INT,
                    1, 0, MPI_COMM_WORLD);
                check(0, 0)+=200;
                MPI_Kokkos_Send<Kokkos::View<int**>,
                    int>(&check, check.size(), MPI_INT,
                    1, 0, MPI_COMM_WORLD);

                auto end = std::chrono::steady_clock::
                    now();
                std::chrono::duration<double>
                    elapsed_seconds = end-begin;
                //std::cout << "Recv Time Count: " <<
                    elapsed_seconds.count() << std::endl
                    ;
            }
            }
            Kokkos::finalize();
            MPI_Finalize();

            return error;
}
```

B.4 Two Dimensional Old Bindings Kokkos Pingpong Test

```
#include <mpicpp.h>
#include <Kokkos_Core.hpp>
#include <iostream>
#include <chrono>
```

```
int main(int argc, char** argv) {
        int error = 0;

        MPI_Init(&argc, &argv);
        Kokkos::initialize( argc, argv );
        {
        int rank = -1;
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        //int n = 8;
        int n = std::stoi(argv[argc-1]);
        Kokkos::View<int**> check("check",n, n);
        Kokkos::parallel_for(check.extent(0),
           KOKKOS_LAMBDA(int i){
             for(int j = 0; j<check.extent(1); j++){
               check(i,j) = (i+1)*(j+1);
               //check(i,1) = (i*i)+1;
               //std::cout << i << "th entry: " << check
                  [i] << std::endl;
             }
           });

        if(rank == 1){
                auto begin = std::chrono::steady_clock
                   ::now();
                // MPI Send time
                MPI_Send(check.data(), check.size(),
                   MPI_INT, 0, 0, MPI_COMM_WORLD);
```

```
        //Kokkos::View<int**> A("A side", n, n)
            ;
        //int * A = (int *) malloc(check.size()
            * sizeof(int));
        MPI_Recv(check.data(), check.size(),
            MPI_INT, 0, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
        //Kokkos::View<int**> rank1_check(A, n,
            n);
        auto end = std::chrono::steady_clock::
            now();
        std::chrono::duration<double>
            elapsed_seconds = end-begin;
        std::cout << "Send Time Count: " <<
            elapsed_seconds.count() << std::endl
            ;
}
else{
        auto begin = std::chrono::steady_clock
            ::now();
        // MPI Recv time
        //Kokkos::View<int**> B("B side", n, n)
            ;
        //int * B = (int *) malloc(check.size()
            * sizeof(int));
        MPI_Recv(check.data(), check.size(),
            MPI_INT, 1, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
```

```
                check(0, 0)+=200;
                MPI_Send(check.data(), check.size(),
                    MPI_INT, 1, 0, MPI_COMM_WORLD);

                auto end = std::chrono::steady_clock::
                    now();
                std::chrono::duration<double>
                    elapsed_seconds = end-begin;
                //std::cout << "Recv Time Count: " <<
                    elapsed_seconds.count() << std::endl
                    ;
        }
        }
        Kokkos::finalize();
        MPI_Finalize();

        return error;
}
```

B.5 Three Dimensional New Bindings Kokkos Pingpong Test

```
#include <mpicpp.h>
#include <Kokkos_Core.hpp>
#include <iostream>
#include <chrono>

int main(int argc, char** argv) {
        int error = 0;
```

```
MPI_Init(&argc, &argv);
Kokkos::initialize( argc, argv );
{
int rank = -1;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
//int n = 8;
int n = std::stoi(argv[argc-1]);
Kokkos::View<int***> check("check",n, n, n);
Kokkos::parallel_for(check.extent(0),
   KOKKOS_LAMBDA(int i){
     for(int j = 0; j<check.extent(1); j++){
       for(int k = 0; k<check.extent(2); k++){
         check(i,j, k) = (i+1)*(j+1)*(k+1);
       //check(i,1) = (i*i)+1;
       //std::cout << i << "th entry: " << check
          [i] << std::endl;
       }}
  });

if(rank == 1){
         auto begin = std::chrono::steady_clock
            ::now();
         // MPI Send time
         MPI_Kokkos_Send<Kokkos::View<int***>,
            int>(&check, check.size(), MPI_INT,
            0, 0, MPI_COMM_WORLD);
```

```
            MPI_Kokkos_Recv<Kokkos::View<int***>,
                int>(&check, check.size(), MPI_INT,
                0, 0, MPI_COMM_WORLD);

            auto end = std::chrono::steady_clock::
                now();
            std::chrono::duration<double>
                elapsed_seconds = end-begin;
            std::cout << "Send Time Count: " <<
                elapsed_seconds.count() << std::endl
                ;
}
else{
            auto begin = std::chrono::steady_clock
                ::now();
            // MPI Recv time
            MPI_Kokkos_Recv<Kokkos::View<int***>,
                int>(&check, check.size(), MPI_INT,
                1, 0, MPI_COMM_WORLD);
            check(0, 0, 0)+=200;
            MPI_Kokkos_Send<Kokkos::View<int***>,
                int>(&check, check.size(), MPI_INT,
                1, 0, MPI_COMM_WORLD);

            auto end = std::chrono::steady_clock::
                now();
            std::chrono::duration<double>
                elapsed_seconds = end-begin;
```

```
                //std::cout << "Recv Time Count: " <<
                    elapsed_seconds.count() << std::endl
                    ;
        }
        }
        Kokkos::finalize();
        MPI_Finalize();

        return error;
}
```

B.6 Three Dimensional Old Bindings Kokkos Pingpong Test

```
#include <mpicpp.h>
#include <Kokkos_Core.hpp>
#include <iostream>
#include <chrono>

int main(int argc, char** argv) {
        int error = 0;

        MPI_Init(&argc, &argv);
        Kokkos::initialize( argc, argv );
        {
        int rank = -1;
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        //int n = 8;
        int n = std::stoi(argv[argc-1]);
        Kokkos::View<int***> check("check", n, n, n);
```

```
Kokkos::parallel_for(check.extent(0),
   KOKKOS_LAMBDA(int i){
     for(int j = 0; j<check.extent(1); j++){
          for(int k = 0; k<check.extent(2); k++){
               check(i,j,k) = (i+1)*(j+1)*(k+1);
          }}
   });

if(rank == 1){
          auto begin = std::chrono::steady_clock
             ::now();
          // MPI Send time
          MPI_Send(check.data(), check.size(),
             MPI_INT, 0, 0, MPI_COMM_WORLD);
          MPI_Recv(check.data(), check.size(),
             MPI_INT, 0, 0, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
          auto end = std::chrono::steady_clock::
             now();
          std::chrono::duration<double>
             elapsed_seconds = end-begin;
          std::cout << "Send Time Count: " <<
             elapsed_seconds.count() << std::endl
             ;
}
else{
          auto begin = std::chrono::steady_clock
             ::now();
```

```
                // MPI Recv time
                MPI_Recv(check.data(), check.size(),
                    MPI_INT, 1, 0, MPI_COMM_WORLD,
                    MPI_STATUS_IGNORE);
                check(0, 0, 0)+=200;
                MPI_Send(check.data(), check.size(),
                    MPI_INT, 1, 0, MPI_COMM_WORLD);
                auto end = std::chrono::steady_clock::
                    now();
                std::chrono::duration<double>
                    elapsed_seconds = end-begin;
                //std::cout << "Recv Time Count: " <<
                    elapsed_seconds.count() << std::endl
                    ;
            }
            }
            Kokkos::finalize();
            MPI_Finalize();

            return error;
}
```

B.7 New Bindings Kokkos Broadcast Test

```
#include <mpicpp.h>
#include <Kokkos_Core.hpp>
#include <iostream>
#include <chrono>
```

```
int main(int argc, char** argv) {
        int error = 0;

        MPI_Init(&argc, &argv);
        Kokkos::initialize( argc, argv );
        {
        int rank = -1;
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        int n = 5000;
        Kokkos::View<int*> check("check",n);
        if(rank == 0){
          Kokkos::parallel_for(check.extent(0),
            KOKKOS_LAMBDA(int i){
                        check(i) = i*i;
                });
          MPI_Request req1;

        }
        for(int i=0; i<100; i++){
        MPI_Barrier(MPI_COMM_WORLD);
        auto begin = std::chrono::steady_clock::now();
        MPI_Kokkos_Bcast<Kokkos::View<int*>, int>(&
            check, check.size(), MPI_INT, 0,
            MPI_COMM_WORLD);
        // time
        auto end = std::chrono::steady_clock::now();
        //MPI_Barrier(MPI_COMM_WORLD);
```

```
        std::chrono::duration<double> elapsed_seconds =
            end-begin;
        double local_max_value = elapsed_seconds.count
           ();//end.count()-begin.count();
        double max_time;
        MPI_Allreduce(&local_max_value, &max_time, 1,
           MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
        if(rank == 0){
            std::cout << "Max Time: " << max_time  <<
               std::endl;
        }
        }
        }
        Kokkos::finalize();
        MPI_Finalize();

        return error;
}
```

B.8 Old Bindings Kokkos Broadcast Test

```
#include <mpicpp.h>
#include <Kokkos_Core.hpp>
#include <iostream>
#include <chrono>

int main(int argc, char** argv) {
        int error = 0;
```

```
MPI_Init(&argc, &argv);
Kokkos::initialize( argc, argv );
{
int rank = -1;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
int n = 5000;
Kokkos::View<int*> check("check",n);
if(rank == 0){
  Kokkos::parallel_for(check.extent(0),
    KOKKOS_LAMBDA(int i){
                check(i) = i*i;
        });
}
for(int i=0; i<100; i++){
    MPI_Barrier(MPI_COMM_WORLD);
auto begin = std::chrono::steady_clock::now();
MPI_Bcast(check.data(), check.size(), MPI_INT,
   0, MPI_COMM_WORLD);
// time
auto end = std::chrono::steady_clock::now();
//MPI_Barrier(MPI_COMM_WORLD);
std::chrono::duration<double> elapsed_seconds =
    end-begin;
double local_max_value = elapsed_seconds.count
   ();
double max_time;
MPI_Allreduce(&local_max_value, &max_time, 1,
   MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
```

```
        if(rank == 0){
            std::cout << "Max Time: " << max_time  <<
                std::endl;
        }
        }
        }
        Kokkos::finalize();
        MPI_Finalize();

        return error;
}
```

VITA

Evan Drake Suggs was born in Chattanooga, TN to parents Tony Lee Suggs and Susan Diane Brown. After graduating Sequatchie County High School, he attended the University of Tennessee at Chattanooga and in May 2022 graduated magna cum laude with a Bachelor of Science in Computer Science. He graduated with a Master's of Science degree in Computer Science Data Science From the University of Tennessee at Chattanooga in August 2023.