Understanding the Usage of MPI in Exascale Proxy Applications

Nawrin Sultana
Auburn University
Auburn, AL
nzs0034@auburn.edu

Anthony Skjellum
University of Tennessee at Chattanooga
Chattanooga, TA
tony-skjellum@utc.edu

Purushotham Bangalore
University of Alabama at Birmingham
Birmingham, AL
puri@uab.edu

Ignacio Laguna
Lawrence Livermore National Laboratory
Livermore, CA
ilaguna@llnl.gov

Kathryn Mohror
Lawrence Livermore National Laboratory
Livermore, CA
mohror1@llnl.gov

***Abstract*—The Exascale Computing Project (ECP) focuses on the development of future exascale-capable applications. Most ECP applications use the Message Passing Interface (MPI) as their parallel programming model and create mini-apps to serve as proxies. This paper explores the explicit usage of MPI in ECP proxy applications. For our study, we empirically analyze fourteen proxy applications from the ECP Proxy Apps Suite. Our result shows that a small subset of features from MPI is commonly used in the proxies of exascale capable applications, even when they reference third-party libraries. Our study contributes to a better understanding of the use of MPI in current exascale applications. This finding can help focus software investments made for exascale systems in the MPI middleware including optimization, fault-tolerance, tuning, and hardware-offload.**

***Index Terms*—Exascale Proxy Applications, MPI, Empirical Analysis**

I. Introduction

In high-performance computing (HPC) systems, MPI is an extensively used API for data communication [1]. A considerable number of large-scale scientific applications have been written based on MPI. These applications are successfully running in current petascale systems with 1,000's to 100,000's of processors [2] and will continue to run in future exascale systems [3]. To increase the applicability of MPI in next-generation exascale systems, the MPI Forum is actively working on updating and incorporating new constructs to MPI standard.

The Exascale Computing Project (ECP) [4] is an effort to accelerate the development and delivery of a capable exascale computing ecosystem. Different ECP applications use different parallel programming models. One of the focuses of ECP is to provide exascale capability and continued relevance to MPI. These ECP applications are typically large and complex with thousands to millions lines of code. As a means to access their performance and capabilities, most of the applications create "mini-apps" to serve as their proxies or exemplars. Proxy application represents the key characteristics of the real application without sharing the actual details. As part of ECP, multiple proxy apps have been developed. The ECP proxy apps suite [5] consists of proxies of ECP projects that represent the key characteristics of these exascale applications.

In this paper, we explore the usage patterns of MPI among ECP proxy applications. We explore fourteen proxy applications in particular from the ECP Proxy Apps Suite and summarize the MPI functionality used in each. An earlier survey [6] on different U.S. DOE applications shows that the HPC community will continue to use MPI in future exascale systems for its flexibility, portability, and efficiency. We study the proxies of those applications to understand how much and what features of the MPI standard they are using. This provides appropriate guidance to developers of full MPI implementations as well as extremely efficient subsets for future systems.

There are multiple open source MPI implementations available (e.g., OpenMPI [7], MPICH [8], and MVAPICH [9]), and many high-end vendors base their commercial versions on these source bases (e.g., Intel, Cray, and IBM). All of the currently available MPI implementations are monoliths. However, sometimes a small subset of MPI is capable of running the proxy of a real application [10]. Large-scale applications often use only a subset of MPI functionality. Our work contributes to a better understanding of MPI usage in mini-apps that represent real exascale applications. It shows that a small subset of features from MPI standard are commonly used in the proxies of future exascale capable applications.

The remainder of this paper is organized as follows: Section II discusses the overview of the proxy apps used in our study. Section III demonstrates the MPI usage patterns among those applications. Finally, we offer conclusions in section IV.

II. Overview of Applications

For our study, we targeted ECP Proxy Applications Suite 2.0 [5], which contains 15 proxy apps. We focused on applications that use MPI. We found 14 applications in the suite that use MPI for communication. Some of these applications use a hybrid model (OpenMP and MPI) of parallel programming where OpenMP is used for parallelism within a node while

TABLE I
APPLICATION OVERVIEW

Application	Language	Description	Third-party Library	Programming Model
AMG	C	Parallel algebric multigrid solver for linear systems arising from problems on unstructured grids	hypre	MPI, OpenMP
Ember	C	Represent highly simplified communication patterns relevant to DOE application workloads	N/A	MPI
ExaMiniMD	C++	Proxy application for Molecular Dynamics with a Modular design	N/A	MPI
Laghos	C++	Solves Euler equation of compressible gas dynamics using unstructured high-order finite elements	hypre, MFEM, Metis	MPI
MACSio	C	Multi-purpose, scalable I/O proxy application that mimics I/O worklods of real applications	HDF5, Silo	MPI
miniAMR	C	Applies a 3D stencil calculation on a unit cube computational domain– divided into blocks	N/A	MPI, OpenMP
miniQMC	C++	Designed to evaluate different programming models for performance portability	N/A	MPI, OpenMP
miniVite	C++	Detects graph community by implementing Louvain method in distributed memory	N/A	MPI, OpenMP
NEKbone	Fortran	Thermal Hydraulics mini-app that solves a standard Poisson equation	N/A	MPI
PICSARlite	Fortran	Portrays the computational loads and dataflow of complex Particle-In-Cell codes	N/A	MPI
SW4lite	C, Fortran	Solves the seismic wave equations in Cartesian coordinates	N/A	MPI, OpenMP
SWFFT	C	Run 3D distributed memory discrete fast Fourier transform	FFTW3 (MPI interface not used)	MPI
thornado-mini	Fortran	Solves radiative trasnfer equation in a multi-group two-moment approximation	HDF5	MPI
XSBench	C	Represents a key computational kernel of the Monte Carlo neutronics application	N/A	MPI, OpenMP

MPI is used for parallelism among nodes. We empirically analyzed these fourteen applications to understand how they use MPI. For the purpose of our study, we only did static code analysis.

Table I provides an overview of the applications. Although some of the applications are written in C++, they all use C or Fortran to call MPI routines. A number of applications (36%) are dependent on third-party software libraries. However, not all of these libraries are MPI-based. Four of the applications—AMG, Laghos, MACSio, and thornado-mini use MPI-based third-party numerical and I/O libraries.

Almost 50% of the applications (6) use both MPI and OpenMP as their parallel programming model. The hypre [11] library also uses OpenMP along with MPI. The data model library, HDF5 [12] uses "pthread" as its parallel execution model.

III. USAGE OF MPI IN ECP PROXY APPS

In this section, we analyze the proxy applications to elucidate their MPI usage patterns.

A. MPI Initialization

All MPI programs must call `MPI_Init` or `MPI_Init_thread` to initialize the MPI execution environment [13]. MPI can be initialized at most once. In addition to initializing MPI, `MPI_Init_thread` also initializes the MPI thread environment. It requests the desired level of thread support using the argument "required".

The majority of applications (93%) of Table I use `MPI_Init` for initialization. Only one application (PICSARlite) uses MPI thread based initialization. In PICSARlite, `MPI_Init_thread` is called with `MPI_THREAD_SINGLE` and `MPI_THREAD_FUNNELED`. In both cases, only one thread makes the MPI calls.

B. MPI Communication

The MPI standard [13], [14] provides different techniques for communication among processes. A set of processes are managed using "communicator" – an object that allows communication isolation for a group of processes.

Point-to-point. Transmit message between a pair of processes where sender and receiver cooperate with each other, which is referred to as "two-sided" communication. To communicate a message, the source process calls a send operation and the target process must call a receive operation. MPI provides both blocking and non-blocking forms of point-to-point communications.

Wildcard. MPI allows the receive and probe operations to specify a wildcard value for source and/or tag. It indicates that a process will accept a message from any source and/or tag. The source wildcard is `MPI_ANY_SOURCE` and the tag wildcard is `MPI_ANY_TAG`. The scope of these wildcards is limited to the processes of the specified communicator.

Collective. A communication that involves participation of all processes of a given communicator. All processes of the given communicator need to make the collective call. Collective communications do not use tag. Collective communica-

TABLE II
MPI CALLS USED FOR COMMUNICATIONS

Application	Point-to-point		Collective	
	Blocking	Non-blocking	Blocking	Non-blocking
`AMG, Laghos`	MPI_Send MPI_Recv	MPI_Isend MPI_Irsend MPI_Irecv	MPI_Allreduce, MPI_Reduce MPI_Allgather{v}, MPI_Gather{v} MPI_Alltoall MPI_Barrier, MPI_Bcast MPI_Scan MPI_Scatter{v}	N/A
`Ember`	MPI_Send MPI_Recv	MPI_Isend MPI_Irecv	MPI_Barrier	N/A
`ExaMiniMD`	MPI_Send	MPI_Irecv	MPI_Allreduce MPI_Barrier MPI_Scan	N/A
`MACSio`	MPI_Send MPI_Ssend MPI_Recv	MPI_Isend MPI_Irecv	MPI_Allreduce, MPI_Reduce MPI_Allgather MPI_Barrier, MPI_Bcast MPI_Scatterv	N/A
`miniAMR`	MPI_Send MPI_Recv	MPI_Isend MPI_Irecv	MPI_Allreduce MPI_Alltoall MPI_Bcast	N/A
`miniQMC`	N/A	N/A	MPI_Reduce	N/A
`miniVite`	MPI_Sendrecv	MPI_Isend MPI_Irecv	MPI_Allreduce, MPI_Reduce MPI_Alltoallv MPI_Barrier, MPI_Bcast MPI_Exscan	MPI_Ialltoall
`NEKbone`	MPI_Send MPI_Recv	MPI_Isend MPI_Irecv	MPI_Allreduce MPI_Barrier, MPI_Bcast	N/A
`PICSARlite`	MPI_Sendrecv	MPI_Isend MPI_Irecv	MPI_Allreduce, MPI_Reduce MPI_Allgather MPI_Barrier, MPI_Bcast	N/A
`SW4lite`	MPI_Send MPI_Recv MPI_Sendrecv	MPI_Isend MPI_Irecv	MPI_Allreduce, MPI_Reduce MPI_Allgather, MPI_Gather MPI_Barrier, MPI_Bcast	N/A
`SWFFT`	MPI_Sendrecv	MPI_Isend MPI_Irecv	MPI_Allreduce MPI_Barrier	N/A
`thornado-mini`	MPI_Send	MPI_Isend MPI_Irecv	MPI_Allreduce MPI_Allgather MPI_Barrier, MPI_Bcast MPI_Scatterv	N/A
`XSBench`	N/A	N/A	MPI_Barrier MPI_Reduce	N/A

tions will not interfere with the point-to-point communications, and vice versa. It has both blocking and non-blocking variants.

One-sided. It is known as Remote Memory Access (RMA) communication, which requires the involvement of either source or target for communication. The source process can access the memory of the target process without the target's involvement. The source is limited to accessing only a specifically declared memory area on the target called a "window."

Persistent. To reduce overhead of repeated calls, MPI introduces persistent communication. If a communication with the same argument list is repeatedly executed within a loop, it may be possible to optimize the communication by binding the argument list to a persistent request and use that request to communicate messages.

We analyzed the proxy applications from Table I to understand the pattern used for MPI communication. Table II presents the significant MPI calls used for communications including the ones used in the third-party libraries.

Most of the applications use point-to-point and collective operations for communications. Applications use both the blocking and non-blocking variants of point-to-point communications. For collective communications, applications mostly use blocking version. We found two applications (AMG, and PICSARlite) that make use of persistent point-to-point communications[1]. For Laghos, the referenced third-party library uses persistent communications. They use `MPI_Send_init` and `MPI_Recv_init` to create persistent request. In Laghos and thorn-mini, most of the MPI communications occur in the third-party libraries. Our result also shows that only one application (miniVite) uses RMA operations (`MPI_Put` and `MPI_Accumulate`).

Table III presents the frequency of MPI calls. It shows that only a small subset of MPI calls is commonly used in these applications.

A small number of applications (four) use wildcard in receive and probe operations. The third-party library, hypre and HDF5 also use the source and tag wildcards. Applications that

[1]Persistent collective communication is new in MPI-4, and yet used in any of these ECP mini-apps.

TABLE III
SUBSET OF MPI CALLS

Application	MPI_Accumulate	MPI_Allgather	MPI_Allgatherv	MPI_Allreduce	MPI_Alltoall	MPI_Alltoallv	MPI_Barrier	MPI_Bcast	MPI_Exscan	MPI_Gather	MPI_Gatherv	MPI_Ialltoall	MPI_Irecv	MPI_Irsend	MPI_Isend	MPI_Put	MPI_Recv	MPI_Recv_init	MPI_Reduce	MPI_Scan	MPI_Scatter	MPI_Scatterv	MPI_Send	MPI_Send_init	MPI_Sendrecv	MPI_Ssend
AMG, Laghos		×	×	×	×	×	×	×		×	×		×	×	×		×	×	×	×	×	×	×	×		
Ember							×						×		×		×						×			
ExaMiniMD				×			×								×					×			×			
MACSio		×		×			×	×					×		×		×		×			×	×			×
miniAMR				×	×			×					×		×		×						×			
miniQMC																			×							
miniVite	×			×		×	×	×	×			×	×		×	×			×						×	
NEKbone				×			×	×					×		×		×						×			
PICSARlite		×		×			×	×					×		×			×	×					×	×	
SW4lite		×		×			×	×		×			×		×		×		×				×		×	
SWFFT				×			×						×		×										×	
thornado-mini		×		×			×	×					×		×							×	×			
XSBench							×												×							

use wildcards are presented in Table IV with the corresponding MPI call.

TABLE IV
USE OF WILDCARDS

Application	Types of Wildcard	MPI Call
AMG	MPI_ANY_SOURCE	MPI_Recv MPI_Iprobe
MACSio	MPI_ANY_SOURCE	MPI_Recv
NEKbone	MPI_ANY_SOURCE	MPI_Irecv
PICSARlite	MPI_ANY_TAG	MPI_Irecv

C. MPI Datatypes

A compliant MPI implementation [15] provides a number of predefined basic datatypes (e.g., `MPI_INT`, `MPI_FLOAT`, `MPI_DOUBLE`), corresponding to the primitive data types of the programming languages. With basic types, we can only send a contiguous buffer with elements of a single type. To send heterogeneous and non-contiguous data, MPI allows application to create its own data types called derived datatypes. It allows transfer of different datatypes in one MPI communication call. Derived types are constructed from existing types (basic and derived). MPI provides several methods for constructing derived types.

Contiguous. Creates a new contiguous datatype by concatenating user defined `count` copies of an existing type.

Vector. Creates a single datatype representing elements separated by a constant distance (stride) in memory.

Indexed. Defines a new datatype consisting of an user defined number of blocks of arbitrary size. Each block can contain different number of elements.

Struct. Creates a structured datatype from a general set of datatypes.

Subarray. Creates a new MPI type describing an n-dimensional subarray of an n-dimensional array. The subarray may be situated anywhere within the array.

Table V presents the derived types used in the applications. Most of the communication calls use predefined basic types. However, six applications from the suite use derived types for a small portion of their communications. The majority of MPI operations that use derived types are point-to-point communication. Only two applications (MACSio and miniVite) use user defined "struct type" in collective calls. Apart from these six applications, the referenced third-party library of Laghos and thornado-mini uses derived data types for point-to-point communication.

TABLE V
MPI DERIVED DATATYPES

Application	Contiguous	Struct	Subarray	Vector
AMG		×		
MACSio	×	×		
miniVite		×		
PICSARlite	×	×	×	×
SW4lite				×
SWFFT	×		×	

D. Communicators and Groups

A *communicator* is a communication channel for MPI processes [16]. A communicator can be thought of a handle to a group of processes. In MPI, *groups* define the participating processes of a communicator.

Intra-Communicator. It is used for communication within a single group of processes. For collective communication, it specifies the set of processes that participate in the collective operation.

Inter-Communicator. It is used for communication between two disjoint groups of processes. The group containing a process that initiates an inter-communication operation is called the "local group". The group containing the target process is called the "remote group".

The default communicator for MPI is `MPI_COMM_WORLD` (intra-communicator) which is available after MPI initialization. It consists of all the processes in a MPI job. The default group is created during MPI initialization with all of the processes and is associated with `MPI_COMM_WORLD` communicator. However, an MPI application can create new communicators to communicate with a subset of the default group of processes.

Each process in a group is assigned a rank between 0 to $n-1$, where n is the total number of processes. Depending on the communication pattern of the application, MPI provides mechanism for logical process arrangement (virtual topology) instead of linear ranking [17]. An MPI application can create new communicators that order the process ranks in a way that may be a better match for the physical topology.

Neighborhood collectives [18], introduced in MPI-3.0, enables communication on a process topology. It allows user to define their own collective communication patterns. All processes in the communicator are required to call the collective. It has both blocking and non-blocking variants.

Most of the ECP proxy applications use the default communicator `MPI_COMM_WORLD` for message passing. Two of the applications, NEKbone and MACSio, use a duplicate of `MPI_COMM_WORLD` for communications. We found six applications that use the communicator and group functionality of the MPI standard to create new communicators. Three of them use virtual process topology. Laghos does not directly create any communicator. It depends on “hypre” that uses MPI communicator and group functionality. The applications showed in Table VI use communicator and group management of MPI standard. None of the applications from the proxy apps suite create inter-communicators. Although three applications use process topology, they do not use the neighborhood collectives operations.

TABLE VI
MPI COMMUNICATOR AND GROUP MANAGEMENT

Application	Communicator and Group	Process Topology
AMG Laghos	MPI_Comm_create MPI_Comm_split MPI_Group_incl	
miniAMR	MPI_Comm_split	
PICSARlite		MPI_Cart_create
SW4lite	MPI_Comm_create MPI_Comm_split MPI_Group_incl	MPI_Cart_create
SWFFT		MPI_Cart_create MPI_Cart_sub

E. Dynamic Process Management

The *dynamic process management* [19] in MPI provides the ability to an already running MPI program to create new MPI processes and communicate with them. The MPI function, `MPI_Comm_spawn` is used to create the new processes. This function returns an intercommunicator. None of the applications of the proxy suite use dynamic process management of MPI.

F. MPI I/O

MPI provides support for parallel I/O using its own file handle `MPI_File` [20]. It provides concurrent read or write access from multiple processes to a common file. In MPI, file open and close operations are collective over a communicator. However, the communicator will be usable for all MPI routines and the use of the communicator will not interfere with I/O behavior.

Only one of the proxy application (miniVite) from Table I use MPI I/O. However, it uses MPI I/O only to read binary data from the file. MACSio, a is a multi-purpose scalable I/O proxy application, uses Silo I/O library. None of the other applications use MPI I/O.

G. Error Handling

The set of errors that can be handled by MPI is implementation dependent. MPI provides several predefined error handler [21]. An error handler can be associated with communicators, windows, or files.

MPI_ERRORS_ARE_FATAL. This is the default error handler. It causes the program to abort on all executing processes.

MPI_ERRORS_RETURN. This handler returns an error code to the user. Almost all MPI calls return a code that indicates successful completion of the operation. If an MPI call is not successful, an application might use this handler test the return code of that call and executes a suitable recovery model.

In the ECP proxy apps suite, only two applications (MACSio and miniAMR) set error handler in `MPI_COMM_WORLD`. MACSio uses `MPI_ERRORS_RETURN` and checks the return code of MPI calls. On the other hand, miniAMR uses `MPI_ERRORS_ARE_FATAL` error handler.

H. MPI Tools and Profiling interface

The MPI standard, MPI 3.0, introduces the MPI Tools interface (`MPI_T`) [22] to access internal MPI library information. Since its inception, the MPI standard [13] also provides MPI profiling interface (`PMPI`) to intercept MPI calls and to wrap the actual execution of MPI routines with profiling code.

None of the proxy applications from Table I use these two interfaces for performance analysis.

I. Overall Usage of MPI

In this section, we presented the current usage of MPI features in the proxy applications. Table VII summarizes the MPI usage patterns of the applications. For each MPI feature, the table shows the percentage of applications that are currently using it. Our result shows that 86% of the applications use point-to-point communications. Two of the applications (miniQMC and XSBench) do not use any point-to-point MPI calls. They are primarily used to investigate “on node parallelism” issues using OpenMP. There is only one point of MPI communication (a reduce) at the end to aggregate result.

IV. CONCLUSIONS

We empirically analyzed 14 ECP proxy applications to understand their MPI usage patterns. Our study shows that these applications each use a small subset of MPI, with significant overlap of those subsets. In the current ECP Proxy Apps Suite, only one application uses one-sided RMA communication routines. The majority of the applications use non-blocking point-to-point routines whereas only one application uses non-blocking collective communications. None of the applications

TABLE VII
OVERALL USAGE OF MPI

Feature	Usage
MPI thread environment	7%
Point-to-point communications	86%
Collective communications	100%
One-sided communications	7%
Persistent communications	14%
Derived Datatypes	42%
Communicator and Group	28%
Process Topology	21%
Neighborhood collectives	0%
Dynamic Process Management	0%
MPI I/O	7%
Error Handling	14%
MPI Tools and Profiling Interface	0%

use dynamic process management, neighborhood collectives, tools, and profiling interface of MPI.

Since these proxy applications use only a small subset of MPI and they represent the key characteristics of real exascale applications, it may be worth focusing on implementing a subset of the MPI functionality in a highly performant MPI library, particularly in view of the large size of MPI's overall function set, and the opportunity to focus optimization, hardware-off-load, and even fault-tolerance on such subsets.

ACKNOWLEDGEMENTS

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National National Security, LLC (LLNL-CONF-XXXX) and partial support from the National Science Foundation under Grants Nos. CCF-1562659, CCF-1562306, CCF-1617690, CCF-1822191, CCF-1821431. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or Lawrence Livermore National Laboratory.

REFERENCES

[1] William D Gropp, William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999.

[2] Rajeev Thakur, Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, Torsten Hoefler, Sameer Kumar, Ewing Lusk, and Jesper Larsson Träff. MPI at Exascale. *Procceedings of SciDAC*, 2:14–35, 2010.

[3] Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, Torsten Hoefler, Sameer Kumar, Ewing Lusk, Rajeev Thakur, and Jesper Larsson Träff. MPI on millions of cores. *Parallel Processing Letters*, 21(01):45–60, 2011.

[4] Exascale computing project. https://www.exascaleproject.org. September, 2017.

[5] Exascale Computing Project Proxy Apps Suite. https://proxyapps.exascaleproject.org/ecp-proxy-apps-suite/. October, 2017.

[6] David E Bernholdt, Swen Boehm, George Bosilca, Manjunath Gorentla Venkata, Ryan E Grant, Thomas Naughton, Howard P Pritchard, Martin Schulz, and Geoffroy R Vallee. A survey of MPI usage in the US exascale computing project. *Concurrency and Computation: Practice and Experience*, page e4851, 2017.

[7] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *European Parallel Virtual Machine/Message Passing Interface Users Group Meeting*, pages 97–104. Springer, 2004.

[8] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.

[9] Dhabaleswar K Panda, Karen Tomko, Karl Schulz, and Amitava Majumdar. The mvapich project: Evolution and sustainability of an open source production quality mpi library for hpc. In *Workshop on Sustainable Software for Science: Practice and Experiences, held in conjunction with Intl Conference on Supercomputing (WSSPE)*, 2013.

[10] Nawrin Sultana, Anthony Skjellum, Ignacio Laguna, Matthew Shane Farmer, Kathryn Mohror, and Murali Emani. MPI Stages: Checkpointing MPI State for Bulk Synchronous Applications. In *Proceedings of the 25th European MPI Users' Group Meeting*, page 13. ACM, 2018.

[11] Robert D Falgout and Ulrike Meier Yang. hypre: A library of high performance preconditioners. In *International Conference on Computational Science*, pages 632–641. Springer, 2002.

[12] HDF5 Support Page. https://portal.hdfgroup.org/display/HDF5.

[13] *MPI: A Message-passing Interface Standard, Version 3.1 ; June 4, 2015*. 2015.

[14] Lyndon Clarke, Ian Glendinning, and Rolf Hempel. The mpi message passing interface standard. In *Programming environments for massively parallel distributed systems*, pages 213–218. Springer, 1994.

[15] Marc Snir, Steve Otto, Steven Huss-Lederman, Jack Dongarra, and David Walker. *"Datatypes" in MPI–the Complete Reference: The MPI core*, chapter 3. 2015.

[16] Marc Snir, Steve Otto, Steven Huss-Lederman, Jack Dongarra, and David Walker. *"Groups, Contexts, Communicators, and Caching" in MPI–the Complete Reference: The MPI core*, chapter 5. 2015.

[17] Marc Snir, Steve Otto, Steven Huss-Lederman, Jack Dongarra, and David Walker. *"Process Topologies" in MPI–the Complete Reference: The MPI core*, chapter 6. 2015.

[18] Torsten Hoefler and Jesper Larsson Traff. Sparse collective operations for mpi. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8. IEEE, 2009.

[19] William Gropp and Ewing Lusk. Dynamic process management in an mpi setting. In *Parallel and Distributed Processing, 1995. Proceedings. Seventh IEEE Symposium on*, pages 530–533. IEEE, 1995.

[20] Peter Corbett, Dror Feitelson, Sam Fineberg, Yarsun Hsu, Bill Nitzberg, Jean-Pierre Prost, Marc Snirt, Bernard Traversat, and Parkson Wong. Overview of the mpi-io parallel i/o interface. In *Input/Output in Parallel and Distributed Computer Systems*, pages 127–146. Springer, 1996.

[21] Marc Snir, Steve Otto, Steven Huss-Lederman, Jack Dongarra, and David Walker. *"Environmental Management" in MPI–the Complete Reference: The MPI core*, chapter 7. 2015.

[22] Marc Snir, Steve Otto, Steven Huss-Lederman, Jack Dongarra, and David Walker. *"MPI Profiling Interface" in MPI–the Complete Reference: The MPI core*, chapter 8. 2015.