A Large-Scale Study of MPI Usage in Open-Source HPC Applications

Ignacio Laguna
Lawrence Livermore National Laboratory
ilaguna@llnl.gov

Ryan Marshall
University of Tennessee, Chattanooga
ryan-marshall@utc.edu

Kathryn Mohror
Lawrence Livermore National Laboratory
mohror1@llnl.gov

Martin Ruefenacht
University of Tennessee, Chattanooga
martin-ruefenacht@utc.edu

Anthony Skjellum
University of Tennessee, Chattanooga
Tony-Skjellum@utc.edu

Nawrin Sultana
Auburn University
nzs0034@auburn.edu

ABSTRACT

Understanding the state-of-the-practice in MPI usage is paramount for many aspects of supercomputing, including optimizing the communication of HPC applications and informing standardization bodies and HPC systems procurements regarding the most important MPI features. Unfortunately, no previous study has characterized the use of MPI on applications at a significant scale; previous surveys focus either on small data samples or on MPI jobs of specific HPC centers. This paper presents the first comprehensive study of MPI usage in applications. We survey more than one hundred distinct MPI programs covering a significantly large space of the population of MPI applications. We focus on understanding the characteristics of MPI usage with respect to the most used features, code complexity, and programming models and languages. Our study corroborates certain findings previously reported on smaller data samples and presents a number of interesting, previously unreported insights.

CCS CONCEPTS

• **General and reference** → **Surveys and overviews**; • **Computing methodologies** → **Parallel programming languages**.

KEYWORDS

MPI, applications survey, program analysis

ACM Reference Format:
Ignacio Laguna, Ryan Marshall, Kathryn Mohror, Martin Ruefenacht, Anthony Skjellum, and Nawrin Sultana. 2019. A Large-Scale Study of MPI Usage in Open-Source HPC Applications. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '19), November 17–22, 2019, Denver, CO, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3295500.3356176

SC '19, November 17–22, 2019, Denver, CO, USA

ACM ISBN 978-1-4503-6229-0/19/11…$15.00
https://doi.org/10.1145/3295500.3356176

1 INTRODUCTION

The Message-Passing Interface (MPI) is the predominant programming model for multi-node communication in scientific computing. Since its conception in 1994, nine versions of the MPI Standard have been released and more than a dozen implementations of MPI exist between open-source and vendor implementations.

While many scientific computing applications have adopted MPI, the HPC community lacks a comprehensive understanding of the use of MPI in applications. Understanding the state-of-the-practice of MPI usage is important to many aspects of supercomputing. First, optimizations of the communication characteristics of applications require a detailed understanding of MPI usage; given detailed MPI usage characterization, implementation developers and HPC centers can allocate resources to optimize the most commonly used MPI features. Second, MPI usage statistics can inform the MPI standardization body about the features of MPI with the greatest and smallest impact on the community, which can help with prioritization and standardization of new features. Third, MPI usage characteristics can be leveraged in the procurement of HPC systems so that that HPC vendors prioritize the most valuable features of the MPI Standard (e.g., focus efforts on performance to high priority APIs). Fourth, application programming support can be enhanced by knowing current trends in programming languages and programming models used along with MPI.

While previous work has provided surveys of MPI usage, these studies have been limited in various ways: either they surveyed a narrow set of MPI programs [13, 14], or they focused on MPI applications in specific HPC facilities [2, 12] or of specific projects [1]. To the best of our knowledge, no one has presented a large-scale study of MPI usage (i.e., one that uses a data sample large enough to adequately represent the population of MPI programs).

In this paper, we present the first such large-scale survey of MPI usage. We analyze more than one hundred distinct MPI programs, which in total contain more than forty million lines of code, the largest data sample ever surveyed thus far.

Obtaining data from a significant sample of applications presents a number of challenges. One of the key challenges is the cost of building and running applications. Previous work [2] has used lightweight profiling tools to automatically profile the MPI usage of jobs running on a given HPC center. This approach, however, is difficult to scale (the code must be built, executed, and cannot experience errors) and it is not portable to all HPC systems. To address the above challenges, in this work we develop a scalable

program analysis method that allows us to analyze applications statically; that is, programs do not need to be compiled and/or executed to obtain their MPI usage characteristics.

Our survey concentrates on open-source MPI programs. This allows us to consider a broad variety of programs, including programs from different scientific domains, programs of different sizes and levels of complexity (from proxy applications to libraries), and programs that receive contributions from various countries.

We analyze four key characteristics of the population of MPI programs: (a) the most important MPI routines and MPI features, (b) program characteristics with respect to the size and complexity of programs, (c) MPI usage characteristics versus code release dates, and (d) use of multi-threaded programming models (i.e., the MPI+X model). To the best of our knowledge, this is the first comprehensive survey of MPI programs that considers all of these characteristics.

The most important findings of our study are the following:

(1) While advanced features of MPI exist to improve performance and code readability, a large proportion of MPI programs surprisingly do not use these features. We find, for example, that point-to-point (blocking and non-blocking) routines are more prominently used than persistent point-to-point or one-sided routines. We also find (surprisingly) that a large portion of applications (67%) use blocking send and received operations.
(2) The majority of MPI programs use only a small set of features from the MPI Standard—a considerable number of applications use only the point-to-point and collective communication features of the standard, leaving other parts of the standard totally unused. This raises questions about the value of the efforts and costs in standardizing minor features (perhaps "syntactic sugar" features) that are ultimately not widely adopted by users.
(3) While three major versions of the MPI Standard have been released (each with at least a minor version), a large portion of programs (42%) rely on the features in MPI version 1.0 only. We find that, for most programs (about 80%), the minimum version they require is MPI version 2.0. We also find that the features provided in subversions of the standard (i.e., 1.3, 2.1, 2.2, and 3.1) are practically of little value to applications since they are rarely used.
(4) We find (not surprisingly) that MPI is widely used in conjunction with several multi-threaded programming models (OpenMP being the most popular). We observe that about 2/3 of the programs use a mixture of such programming models, which can perhaps be explained by an increase in available architectures and in-node parallelism models in the supercomputing arena.
(5) While the conventional wisdom is that Fortran is largely used on scientific computing programs (partly because many legacy codes were originally developed in Fortran), we find that C++ is the dominant language in MPI programs.

2 RELATED WORK

While a number of previous surveys have reported on the use of general-purpose programming languages in scientific and non-scientific programs [9–11] few studies have focused on the use of the MPI programming model. As far as we know, the first survey of MPI usage in scientific codes goes back to 1997 [14]. This work evaluated the use of the MPI programming model and of other models, such as PVM and Shmem in scientific codes. The goal of this study was not to understand the use of MPI functionality but rather to evaluate the impact of software complexity of these models and its implications to runtime performance. This study evaluated only **six** small benchmarks.

More recently, a number of attempts have been made to better understand the use of MPI. A survey of MPI usage in the US Exascale Computing Project (ECP) is presented in [1]. This study focused on applications of the ECP projects only. The goal was to understand the applications needs around MPI usage and surveyed issues related to possible problems with respect to the MPI Standard, whether current codes expect to use MPI at exascale, and questions about the dynamic behavior of codes. This survey analyzed a total of **56** projects, with each project having on average one main application.

Sultana et al. [13] present a survey of usage of MPI in the ECP proxy applications suite. This work primarily analyzed the dynamic characteristics of the programs but also some static characteristics; it presents analysis of MPI call usage as well as other aspects of MPI, such as error handling, tools, dynamic process management, and I/O. This work analyzed **14** MPI programs. Klenk and Fröning [8] study the dynamic MPI characteristics of **18** exascale proxy programs.

Previous works have reported on the dynamic usage of MPI in production programs in specific HPC systems and centers. Rabenseifner [12] reports dynamic MPI usage on Cray T3E and SGI Origin2000 systems. In a more recent study, Chunduri et al. [2] analyzed MPI jobs on a large IBM BG/Q supercomputing system (Mira) and its corresponding development system (Cetus) at the Argonne National Laboratory over a span of two years. The study attempted to obtain dynamic MPI traces of thousands of jobs in these two systems; however only the jobs that terminate normally (i.e., without an error) and hence produce a profile log file are analyzed. A total of **79** profile files (which correspond to 79 unique MPI executables) were analyzed. Note that the analysis uses *unique executables*, where multiple unique executables can correspond to the same MPI program source code. In contrast, in our survey we analyze *unique program source codes*.

In summary, the three most important differences between previous work and our work are the following: (1) we use the largest data sample ever employed thus far (**110** unique MPI programs) to understand MPI usage; this data sample represents the population of MPI programs better than previous samples and yields a more realistic understanding of MPI usage in the HPC community—the statistics we present can only be observed with a large sample such as ours; (2) we analyze the static usage characteristics of MPI whereas most previous studies focus on the dynamic characteristics of MPI usage; (3) we sample the space of programs as randomly as possible, whereas the relevant previous work sample from specific projects or specific HPC centers.

3 BACKGROUND ON MPI

In this section, we provide background on MPI that is useful for the reader to understand our study. We detail a number of historical

MPI Standard Version	Release Date
MPI-1.0	May 05, 1994
MPI-1.1	June 12, 1995
MPI-1.2	July 18, 1997
MPI-1.3	May 30, 2008
MPI-2.0	July 18, 1997
MPI-2.1	June 23, 2008
MPI-2.2	September 4, 2009
MPI-3.0	September 21, 2012
MPI-3.1	June 04, 2015

Table 1: MPI Standard versions with their release dates

aspects of the MPI standardization process, which will help us explain some of our findings later on. We also provide a summary of the key MPI features that we analyze in the sample programs.

3.1 MPI Standard

The MPI standardization effort was formally started in early 1993 (with pre-organizational meetings in 1992) to facilitate the development of parallel applications and libraries. It involved approximately 40 organizations that included major vendors of concurrent computers, researchers from universities, government laboratories, and industry, mainly from the United States and Europe.

The first version of MPI was released in May 1994. Since then the MPI Forum has released several versions (both major and minor) of the MPI Standard. Several new features (e.g., dynamic process management, one-sided communication, MPI I/O) were introduced in version 2.0. Another major update to the MPI Standard was version 3.0. The MPI-3.0 standard contains significant extensions to MPI functionality, including non-blocking collective operations, new one-sided communication operations, and Fortran 2008 bindings. Table 1 presents all the major and minor MPI Standard version with their respective release dates.

3.2 MPI Features Classification

MPI Standard version 3.1 (the most recent version) specifies roughly 443 distinct routines. These routines can be grouped into 13 categories. In our analysis, we refer to these categories as MPI *features*. As a quick reference for the reader, we summarize these categories as follows—for more information, the reader should refer to the MPI Standard [5]:

- **Point-to-Point communication**: This feature specifies how to transmit messages between a pair of processes where both sender and receiver cooperate with each other.
- **Collective communication**: This feature describes synchronization, data movement, or collective computation that involve all processes within the scope of a communicator.
- **Persistent communication**[1]: To reduce the overhead of repeated point-to-point calls with the same argument list, this feature creates a persistent request by binding the communication arguments and uses that request to initiate/complete messages.
- **One-Sided communication**: This feature defines a communication where a process can write data to or read data from another process without involving that other process directly. Various synchronization models are defined, some involving all the processes in the underlying group that formed the one-sided "window."
- **Derived Datatypes**: This feature describes how to create user-defined structures to transfer heterogeneous/non-contiguous data.
- **Communicator and Group management**: A *Group* is an ordered set of processes and a *Communicator* encompasses a group of processes that may communicate with each other. This feature describes the manipulation of groups and communicators (e.g., construction of new communicator) in MPI.
- **Attribute caching**: This feature allows an application to attach arbitrary pieces of information to MPI communicators, windows, and datatypes.
- **Process Topology**: An MPI application with a specific communication pattern can use this feature to specify a mapping/ordering of MPI processes to a geometric shape. Two topologies supported by MPI are Cartesian and Graph.
- **MPI I/O**: This feature provides support of concurrent read/write to a common file (parallel I/O) from multiple processes.
- **Error Handling**: An MPI application can associate an error handler with communicators, windows, or files. If an MPI call is not successful, an application might use this error handling feature to test the return code of that call and execute a suitable recovery model or abort the application.
- **Info object**: This feature allows passing additional information to MPI APIs through an info argument. The info object consists of an unordered set of (key, value) pairs.
- **Process management**: This feature specifies how a running MPI program can create new MPI processes and communicate with them.
- **Tools Interface**: This interface allows tools to profile MPI applications and to access internal MPI library information.

There are also miscellaneous features, such as `MPI_Wtime` and `MPI_Wtick` and generalized requests, that are not specific to any of these thirteen categories. Their omission from a catch-all category does not detract from our study.

4 DATA GATHERING

In this section, we explain our data gathering strategy. We start by defining the research questions that we seek to answer. Next, we explain the static analysis that we develop to gather MPI usage from a large number of applications, the applications we target, and, finally, we identify certain limitations of this approach.

4.1 Research Questions

In this study, we are interested in understanding how the use of MPI is correlated to four key aspects: (a) number of MPI calls (or percentage of the total calls) in programs, (b) size of the programs, (c) time span (longevity) of the programs (e.g., birthdate to present), and (d) programming models and languages. In particular, we are interested in answering the following questions:

- Q1: Do MPI programs use most of the MPI routines that are defined in the MPI Standard, or do they use a small portion of them?

[1]Persistent Collective communication, which is expected to be part of MPI-4, is not considered in this survey.

- Q2: Is there a correlation between the time span of programs and their MPI usage?
- Q3: Is it true that newer programs use modern MPI features while older programs tend to use only old MPI features?
- Q3: Is there any correlation between the size of programs and MPI usage? For example, do complex programs tend to use more MPI features than smaller programs?
- Q4: What is the use of different programming languages (C, C++ and Fortran) in MPI programs?
- Q5: What is the use of multi-threaded programming models (e.g., CUDA and OpenMP) in conjunction with MPI programs?

4.2 Community Benefits of This Study

People in the HPC community can benefit from the answers to these questions, including: (a) people involved in the procurement of HPC systems, who can request that HPC vendors give higher priority to optimizations that affect the most-used functionality; (b) programmers of new MPI applications, who may decide to utilize only the most used MPI functionality since this is the most reliable and optimized; (c) implementors of MPI libraries, who can opt to focus their efforts on the MPI features that have the highest impact to applications; (d) vendors of communication hardware (e.g., interconnects); (e) programmers of closed-source applications who can benefit from how open-source applications are written; (f) tool developers who could guide their development to cover the most used MPI functionality; and (g) the MPI Forum, the body that standardizes new versions of MPI.

4.3 Static Analysis

4.3.1 Static Versus Runtime Analysis. The main advantage of runtime analysis is that it provides a mechanism to analyze the runtime communication patterns and MPI usage with given inputs in specific HPC systems. The disadvantage of this approach, however, is that it is not a practical method to analyze a large number of applications because the applications must be compiled and run, which is demanding and time-consuming. While a static analysis approach does not give insights on specific runtime behavior, it is a more scalable method to cover a larger set of codes since applications do not need to be fully compiled and/or run. Because our goal is to cover a large number of open-source MPI applications, we base our study on static analysis.

4.3.2 Our Approach: Static Analysis. Our analysis framework statically parses MPI programs and detects the use of *MPI routines* or *calls*[2]. Our framework traverses all the directories in the source code of an application and inspects each file to determine the use of MPI calls in each. In addition to reporting the MPI calls that are used in the program, the analysis also reports whether the application uses C, C++, or Fortran[3], the number of lines (not including comment lines) of the application, and whether or not the application uses CUDA, OpenMP, OpenCL, and/or OpenACC as a multi-threaded programming model. The static analysis framework is written in Python and relies on the `cloc` [3] tool for lines-of-code counting and the detection of programming languages.

4.3.3 Scalability of Static Analysis. Our static analysis framework provides a fast and simple method to gather static MPI usage and other static characteristics of MPI programs, which does not require compiling or running programs. We avoid using complex syntax and semantic analysis of the programs (we assume programs are correctly written using their corresponding language, C, C++, or Fortran) and rely on parsing *MPI statements*; that is, finding places in the program that make use of (a) MPI calls or (b) MPI symbols, using a set of predefined regular expressions. The analysis opens each file and matches MPI statements in a per-line, per-file basis.

The analysis is fairly scalable, and has a complexity of $O(n)$, where n is the number of lines of the program. In most applications it took less than a minute to perform the analysis; only in a few cases with large applications we saw analysis times of a few minutes (but on average less than 10 minutes). On the other hand, performing a runtime analysis on all these applications could take a significantly large amount of time.

4.3.4 Date and Age Information. In addition to static features, we gather time span (longevity) information of the programs. In particular, we obtain two timestamps: *date of the latest release* and *date of the first release*. This time-span information, however, is not captured by the static analysis framework and is performed manually. We use the repository website where the program is obtained as the main source to get these two timestamps. If the repository does not specify the latest release, we use the date of the latest commit as the date of the latest release—we use a similar approach for the date of the first release if such release date cannot be found.

4.4 Sample MPI Programs

We statically analyze a total of 110 MPI programs. Appendix A shows the names of all these programs.

Strategy for Online Crawling. In searching for sample programs, our strategy is to sample as many MPI programs as possible without limiting ourselves to specific classes of applications. More specifically, our crawling strategy uses the following criteria:

- **Randomness:** sample MPI programs as randomly as possible;
- **Significantly large sample size:** target a data sample size that is larger than the size of data samples that has been used in previous studies;
- **Heterogeneity:** sampled applications that include different classes and structures of codes, including (1) small and large codes, (2) proxy and non-proxy applications, and (3) libraries and standalone codes.

MPI programs can be developed in different places around the world end can be made available online using using several hosting methods. Our crawling strategy systematically prioritize our search using this method:

(1) Obtain programs from USA HPC centers and universities
(2) Obtain programs from European HPC centers and universities
(3) Obtain programs from Asian HPC centers and universities
(4) Online crawling on github.com
(5) Online crawling on bitbucket.org

[2]While the MPI Standard often uses the term *routines*, in this paper we use the terms 'MPI routines' and 'MPI calls' interchangeably.
[3]We only detect usage of these languages in the programs since these are the languages formally supported by the MPI Standard.

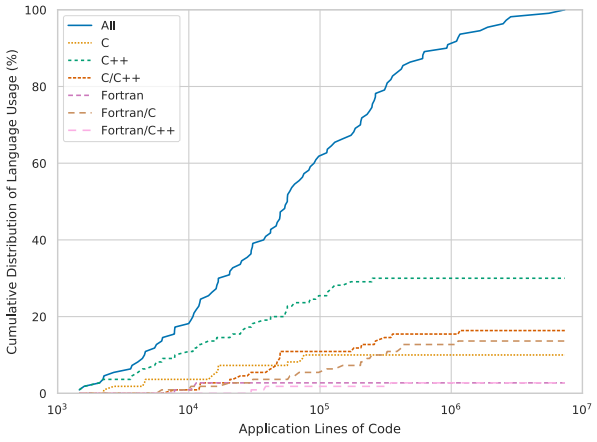


Figure 1: Cumulative distribution of application count with respect to their code size.

4.4.1 Overview of Programs. A distribution of the sample programs versus the number of lines of codes is shown in Figure 1. The application code size stretches from 1,400 lines to seven million lines and are written in all the core high-performance computing languages encountered. We split the codes by languages used (C, C++, Fortran, and some combinations of them) to illustrate the most used language. Other languages and components of an application repository are aggregated under the *All* category.

The cumulative distribution shown in Figure 1 shows the overall probability of application codes by count using particular combinations of languages. We can see *C++* being widely used primarily and in combination with *C*. Using this overview of the data set we can state 30% of the sample programs are written using *C++*. We present more details on the size of programs in Section 7. We also observe that after C++, smaller programs tend to use C initially. This could be explained by the use of C in the vast majority of documentation and introductory material in MPI programming.

4.5 Limitations

While our static analysis considers most of the programming idioms upon which MPI calls are used, the analysis has limitations on identifying a few corner cases of MPI usage, e.g., when MPI is used via Python bindings or C++ bindings, and when MPI calls are made in the program using complex C/C++ macros. Fortunately, we identified only a few of such cases and we filtered these out from the samples set.

The timing information analysis presented several challenges. In particular, in 4% of the codes we were not able to gather the date of first release—sometimes the information was not available from the repository since the code was initially released much earlier than the dates of release in the repository. In these cases, we eliminated the code from any analysis that involves dates of first release.

5 OVERVIEW OF APPLICATIONS' MPI USAGE

In this section, we briefly overview the overall use of MPI by the applications in our study.

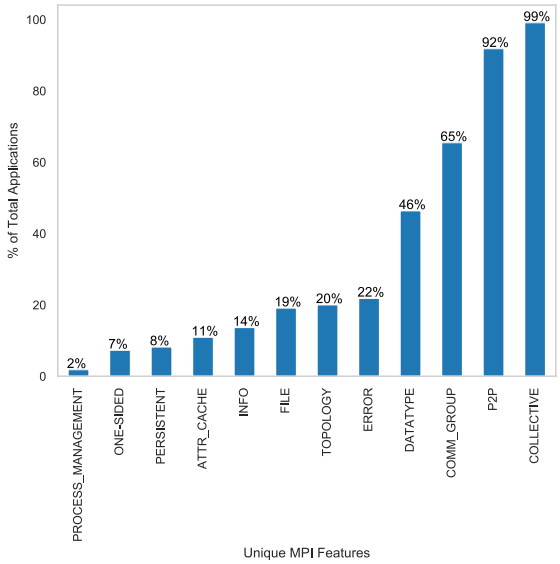


Figure 2: MPI features shown by percentage of applications using them.

5.1 MPI Calls Usage

To understand the usage of MPI in our sample application set, we first create a list of all the MPI calls from the latest MPI Standard 3.1 [5] using the "MPI Function Index". The list excludes any MPI constants or predefined handles. Then, we categorize the MPI calls into MPI features. We group the MPI calls into 13 categories—*P2P*, *Collective*, *Persistent*, *One-Sided*, *Datatype*, *Topology*, *Tool*, *Comm_Group*, *Attr_Cache*, *Error*, *File* (MPI I/O), *Info*, and *Process_Management* based on the feature classification described in Section 3.2. Finally, all the environmental management (e.g., startup, memory allocation) and language bindings routines are considered as *Other* category.

For each application, we collect all the MPI calls using static analysis on the source as mentioned in Section 4.3. Then we categorize the calls to MPI features. An application uses an MPI feature if it has at least one MPI call that falls into that particular category. If an application uses both `MPI_Barrier` and `MPI_Allreduce`, our analysis will consider that the application is using the *Collective* feature of the MPI Standard; thus, we do not count the frequency of the MPI calls.

Findings. Figure 2 presents the overall usage of MPI features. Our figure shows that all of the applications except one use collective communication routines. While the percentage of applications using P2P communication is above 90%, less than 10% use persistent point-to-point and one-sided communication. Out of all applications, only two use the process management feature of the standard.

> Point-to-point calls are more prominently used than either persistent point-to-point or one-sided calls.

Our analysis shows that the process management feature has rarely been adopted in codes even though it was introduced long ago

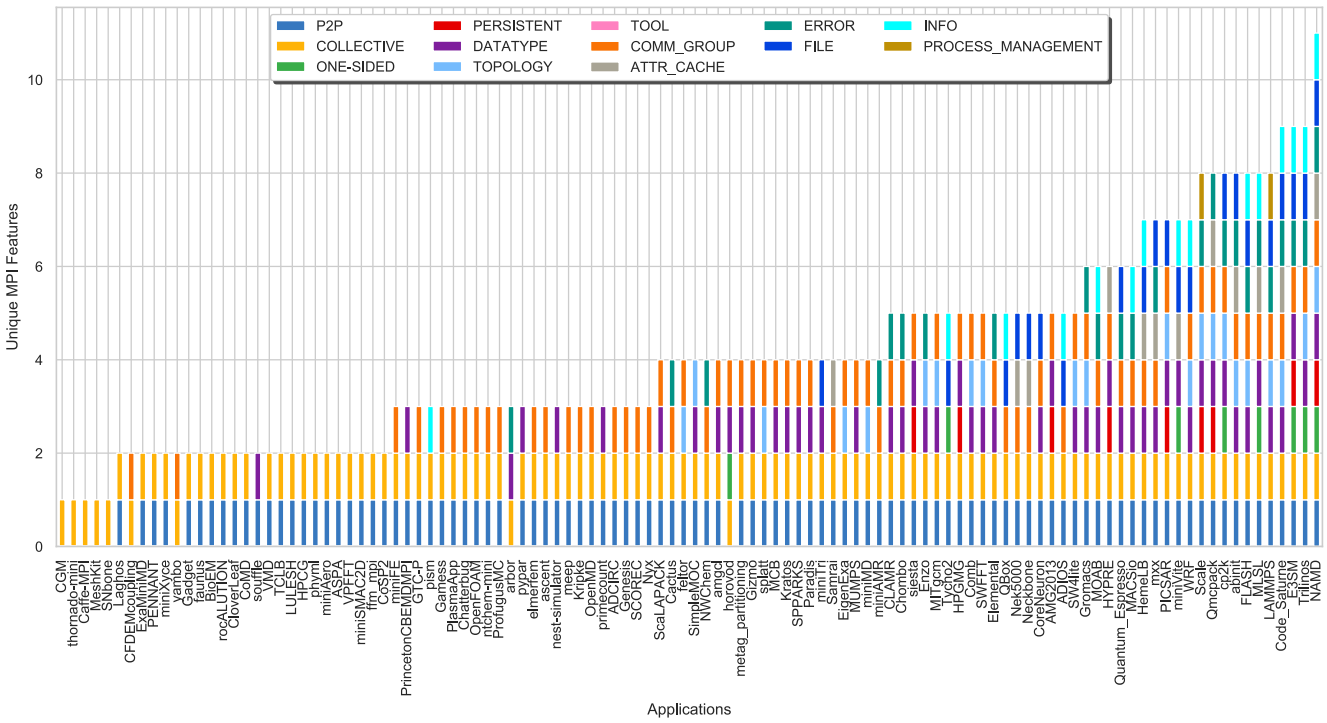
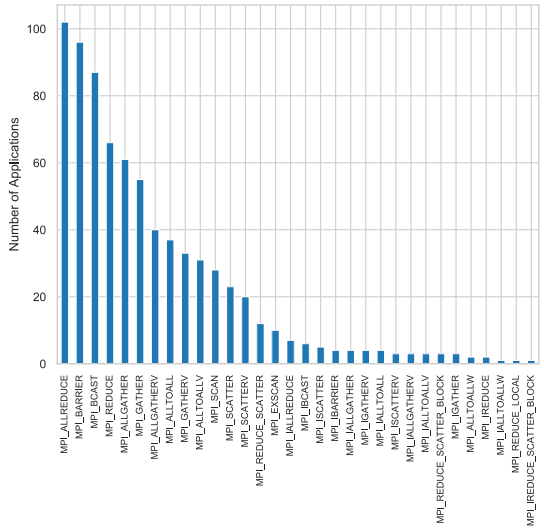
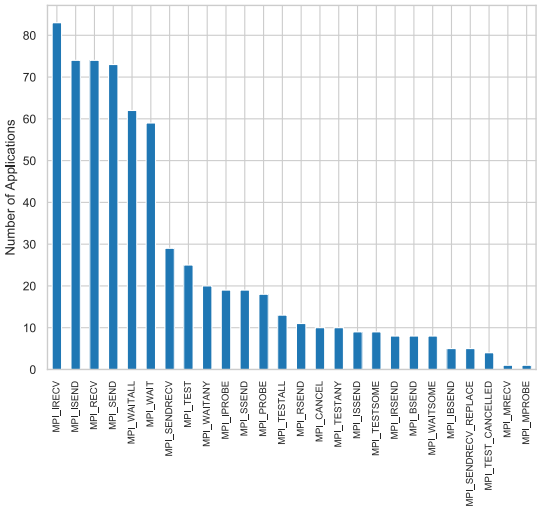


Figure 3: Usage of unique MPI features by applications. We show that the majority of applications use a small fraction of the MPI functionality.



(a) MPI collective function usage. We show AllReduce is the most used collective operation.



(b) MPI point-to-point function usage. We show non-blocking send and recv are the most used functionality.

Figure 4: Overview of communication calls used in the applications.

(in version 2.0, 1997). We hypothesize that the reason is that most production environments use batch schedulers, and such schedulers a) have little or no support for dynamic changes in the runtime size of MPI programs, b) production applications specify their maximum

size of process utilization at startup, and c), most programs operate in terms of the build-down from `MPI_COMM_WORLD`, rather than a build-up mode of process creation and aggregation. The latter is also complex-to-cumbersome syntactically (e.g., merging multiple intercommunicators).

> The process management feature is only rarely used in MPI programs.

Figure 3 shows the per-application usage of MPI features. Different applications use different sets of MPI features. None of our applications use all 13 features. We observe that just one application uses only point-to-point while almost 8% of our sample applications use only collectives for communication. Our analysis also shows that almost 24% of the applications use only point-to-point and collective features from the standard. Only 19% of the applications use six or more MPI features.

In both figures, we do not show the *Other* category as all of the applications use the calls `MPI_Init` and `MPI_Finalize` which are part of the *Other* category.

> The majority of MPI programs surveyed only use a small set of features from the MPI Standard. A considerable number of applications use only point-to-point and collective communication features of the standard.

Figure 4(a)–(b) shows the collective and point-to-point communication calls used in the applications. Figure 4(a) shows the collective routines versus the number of applications that use these routines. Here we observe that, `MPI_Allreduce`, `MPI_Barrier` and `MPI_Bcast` are the most commonly used collective routines. We also observe that few applications use the non-blocking collective routines introduced in the MPI Standard version 3.0.

Figure 4(b) shows the usage of point-to-point communication routines of the data sample. We observe that the majority of the applications use the non-blocking point-to-point calls (`MPI_Irecv` and `MPI_Isend`). However, a large number of applications (over 70) use blocking send and receive calls.

> `MPI_Allreduce` is the most used collective routine. Applications use more non-blocking point-to-point calls than blocking calls.

5.2 MPI Version Usage

In this section, we explore the version of MPI used by applications in the sample set. This is done by matching the minimum function set used by the application with the function sets extracted from the different versions of the MPI Standard.

As expected, many applications only use MPI 1.0 features, whereas higher versions of MPI are more rarely used. Additions to the MPI Standard in minor versions are almost never used. The threading support added by the `MPI_INIT_THREAD` in MPI 2.0 seems to be the only reason why twelve applications use that version. Finally, the usage of MPI 3.0 is largely because of the usage of non-blocking collectives and neighborhood collectives with nine of nineteen applications making use of either of those MPI-3.0 features.

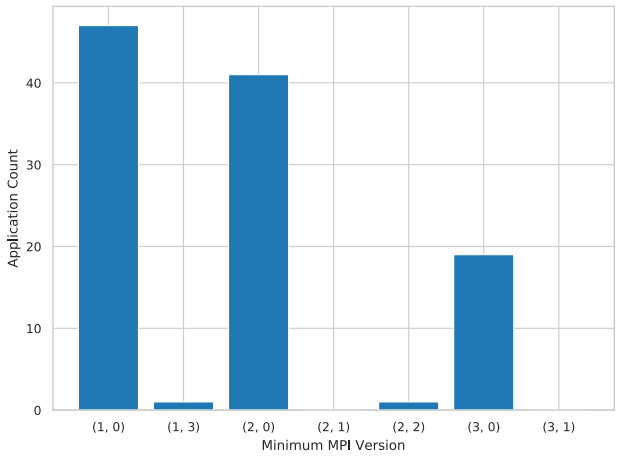


Figure 5: Application count versus the minimum version of MPI which provides the functionality required by each application. Most applications require only early versions of MPI.

6 TIME-BASED USAGE ANALYSIS

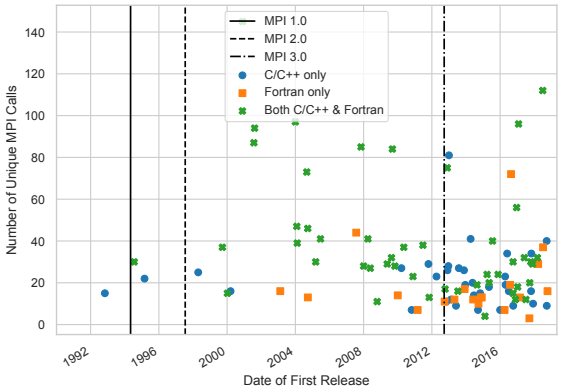
In this section, we analyze MPI usage with respect to the release dates of applications. We gather two date metrics for each code. The first is *date of the first release*. We expect that this helps us understand MPI usage with respect to the maturity of the code—older codes tend to be more mature than newer codes. The second metric is *date of the latest release*. We expect that this helps us understand MPI usage with respect to codes that have been actively improving their functionality—we assume that codes with recent releases are working toward adding new functionality to the code or toward improving the code reliability (e.g., fixing bugs).

When analyzing release dates, each code is classified into three groups, depending on the language that is used: applications that use C/C++ only, applications that use Fortran only, and applications that use both C/C++ and Fortran.

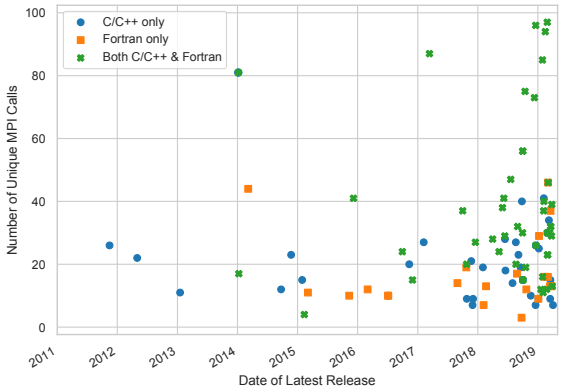
Findings. Figures 6(a)–(b) shows correlation of MPI calls to release dates. Figure 6(a) shows the number of unique MPI calls versus the date of the first code release of the application. The three vertical lines present the release dates of the major MPI Standard version, 1.0, 2.0, and 3.0. We observe that codes that were released before 2000 limit the number of used MPI calls to only 40. This is expected because the MPI Standard version 2.0 was released within 1997–1998 and contained fewer MPI calls than future versions (i.e., versions after 1997–1998); thus, codes that were released after 2000 tend to use a richer set of MPI calls, possibly because they make use of newer MPI functionality.

> The MPI Standard version 2.0 may have significantly allowed developers to explore using new MPI features in scientific codes.

Figure 6(b) shows the number of unique MPI calls versus the date of the latest code release of the application. Here, we observe that, in our data sample, a large proportion of the codes pushed out their latest release in the last two years (2018–2019); these recent



(a) Unique MPI calls versus the first code release date.



(b) Unique MPI function calls versus the latest code release date.

Figure 6: Correlation of MPI function calls with application release dates.

releases (in this case, recent updates since this is the latest release) use a larger set of MPI calls with respect to the codes that were last updated before 2017. Our sample indicates that a significant portion of MPI projects (about 75%) are actively improving their codes with recent releases in the last two years.

7 COMPLEXITY ANALYSIS

In this section, we discuss the MPI usage with respect to the complexity of programs. We want to explore whether the code complexity correlates with different usages of MPI. Several software metrics, such as number of lines of code, cyclomatic complexity, and instruction path length, can be used to measure complexity [6]. We chose to use *lines of code* since it has been proven to be effective [15] and it is easy to calculate in practice.

In Section 7.1, we explore the correlation between line count and the number of unique MPI functions used. Section 7.2 analyzes the usage of different programming languages with which MPI is used.

7.1 MPI Function Usage

Figure 7 shows all applications' unique MPI function usage with respect to the number of lines present in the respective application. We exclude *NAMD* from this analysis because it exhibits counts that are distant from the rest of the applications—it uses 314 of the 444 MPI functions present in MPI 3.1—thus we classify it as an outlier for this analysis.

The trend line shown in Figure 7 indicates a correlation between *lines of code* and the number of unique MPI functions used. From this it can be observed that larger applications have more complex usage of MPI and require a more complex set of communications capability, an intuitive observation. On the other hand, smaller applications below 100,000 lines of code rarely use more than forty functions. The plot shows a general set of MPI functionality required by many applications, where only few large applications exploit even close to a large set of MPI functionality. In addition,

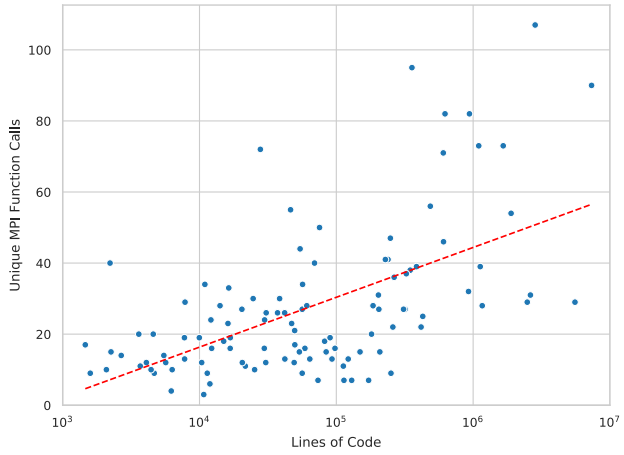


Figure 7: Unique MPI functions used by applications of various sizes. Larger applications tend to use more MPI functionality.

few applications use less than nine MPI functions, which suggests a minimum requirement for a functional usage of MPI.

7.2 Language Usage vs Code Size

In addition to the direct usage of MPI, it is also interesting to note what languages are used most for development with MPI. Figure 8 shows the language fractions of an application ordered in respect to the code base size. The languages of interest here are C, C++, and Fortran since these are the languages that the MPI Standard supports (at least directly). Applications often use other languages, such as scripting languages (e.g., Bash and Python), in their source code. We classify languages that are distinct from C, C++, and

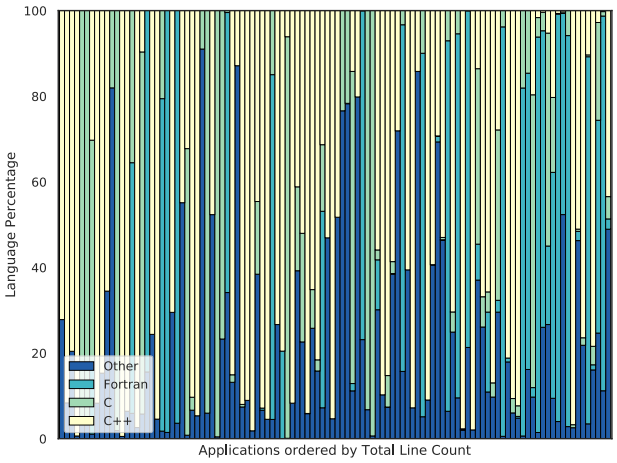


Figure 8: Programming language percentage used versus application code size. Fortran and C++ are the most used programming languages, with C being used surprisingly little.

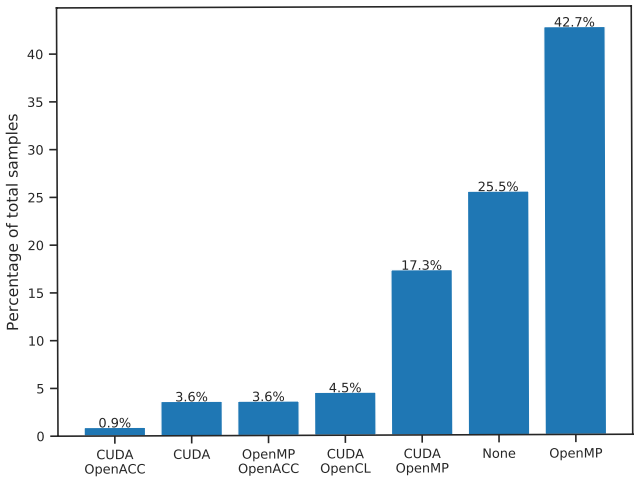


Figure 9: Percentage of total applications using multi-threaded programming models. The most often used model is MPI+OpenMP.

Fortran as *Other* and we calculate it by subtracting the number of C, C++, and Fortran codes lines from the total number of code lines.

As shown, C is used more rarely than we expected, however, it is used mainly for smaller applications. For larger applications C++ or Fortran are the predominant programming languages. The Other category is also interesting since, for some applications, this comprises the majority of the code base.

Surprisingly, C only comprises 16% of the language usage. C++ comprises 44%, with Fortran at 19%, and other languages at 21% of the application code bases. Since no C++ MPI interface exists[4], usage from C++ is most likely through the C interface.

> C++ appears to be the most significant programming language in MPI programs—C appears to be the least significant.

8 PROGRAMMING MODELS ANALYSIS

This section describes our findings for the `MPI+X` programming model, where `X` corresponds to multi-threaded programming models used along with MPI. We also study the adoption of the three general-purpose programming languages that the MPI Standard supports—C, C++, and Fortran—in our sample set.

Our static analysis detects whether applications make use of `OpenMP`, `OpenCL`, `OpenACC`, and/or `CUDA` constructs. We do not consider the frequency of these constructs used in a given application, we consider only whether or not the application uses any of these models if the static analysis detects at least one construct for any model. If none of the models are seen in the code, we record it as `None`. We also checked for all combinations of mixed usage over the four models, such as `CUDA` and `OpenMP`.

Findings. Figure 9 shows the percentage of total programs using multi-threaded programming models. We find that 74.5% of the programs use some form of hybrid code (i.e., MPI+X). Additionally, we manually checked the samples comprising the 25.5% in the `None` category for custom-built solutions using `pthreads`, but none were found. `OpenMP` is by far the most prevalent, found in nearly 3/4 of the sampled applications and 42.7% exclusively. We found that 17.3% of the sample set uses `CUDA` and `OpenMP`. `OpenCL` and `OpenACC` are not found to be used exclusively in any of the samples; however, we found 4.5% of the samples use `CUDA` and `OpenCL`, and 0.9% use `CUDA` and `OpenACC`, while 3.6% use `OpenACC` and `OpenMP`. We found another 3.6% of the applications use native `CUDA` exclusively.

The fact that a significant number of applications (28%) use a mixture of multi-threaded models can be attributed to an increasing availability of various classes of architectures in supercomputing systems, including the increased prevalence of accelerators. This has implications on frameworks to enable performance portability [4, 7], which allow executing programs on multiple platforms and in-node parallelism models; these frameworks may become increasingly important in the future.

> About one third of programs use a mixture of multi-threaded programming models, possibly due to an increase of available distinct architectures in HPC systems.

The MPI execution environment is initialized by using `MPI_Init` or `MPI_Init_Thread`. If `MPI_Init_Thread` is used, the program can use one out of four thread environment options: `MPI_THREAD_SINGLE`, `MPI_THREAD_FUNNELED`, `MPI_THREAD_SERIALIZED`, and `MPI_THREAD_MULTIPLE`. We find that 70% of the programs in our sample use the hybrid (MPI+OpenMP) programming model. However, not all of these programs use `MPI_Init_thread` to initialize the thread environment. We find that less than half of the programs (30 out of 77) with the hybrid model use `MPI_Init_thread` to initialize MPI.

The routine `MPI_Init_thread` takes two parameters: *required* which indicates the level of desired thread support and *provided* indicating the actual thread support while executing the program.

[4]No C++ interface exists after MPI 2.1

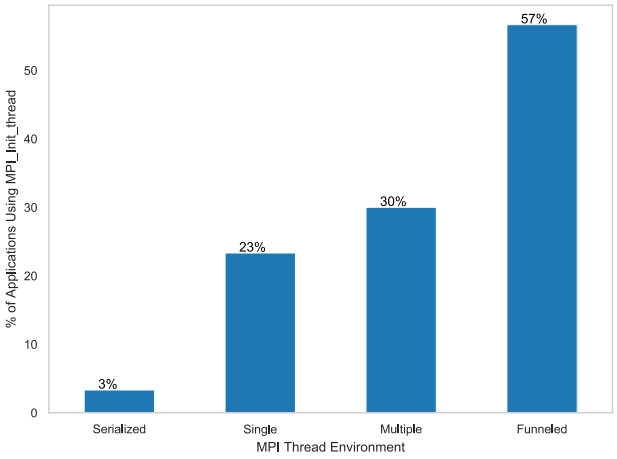


Figure 10: The percentage use by applications of thread modes provided by MPI. The funneled threading model is used most often.

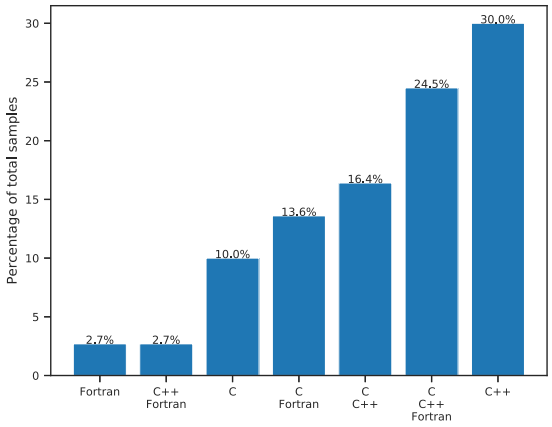


Figure 11: The percentage of applications using specific programming languages. Most applications use a mixture of languages.

In our static analysis, we only capture the *required* thread level specified by the application.

Figure 10 shows the use of thread environment in our sample programs. Here, we show the percentage with respect to the total application that actually use `MPI_Init_thread`. Our analysis shows that, the majority (almost 60%) of them use `FUNNELED` as the required level of thread support. We also observe that, around one-third of these applications use `MULTIPLE`. With respect to the total application set, only 8% of programs use thread `MULTIPLE`. One possible reason of observing more programs with `FUNNELED` than `MULTIPLE` might be that the performance implications of `MULTIPLE` in MPI implementations.

> Programs mostly use `MPI_THREAD_FUNNELED` as the required level of thread support.

Figure 11 shows the percentage of sampled applications whose source codes used C, C++, and Fortran programming languages exclusively, or some combination of the three. Only 2.7% of the applications use Fortran exclusively in their sources, while about 10% use C and 30% use C++. We found that 24.5% of the applications in the sample set use all three languages, while 16.4% use C with C++, 13.6% use C with Fortran, and 2.7% use C++ with Fortran.

To our surprise, few applications in our sample set are written purely in Fortran, though Fortran is found in about 41% of the applications that use more than one language. While early HPC applications were based in C and Fortran, the trend appears to be shifting toward using more modern languages such as C++, which is used exclusively in 30% of our sample set.

> In the past, programs were predominantly written in Fortran and C. We observe that C++ is becoming the dominant programming language for MPI programs.

9 DISCUSSION OF PREVIOUS INSIGHTS

We compare our key findings with the observations that have been reported in previous surveys. Unfortunately, most previous surveys use only a small fraction of MPI programs in comparison to our sample set; thus, a direct comparison would not be fair. The study presented in [2] analyzed 79 MPI executables—we consider this survey as the closest survey to ours, at least in terms of data sample size; (we analyzed 110 unique MPI programs).

It is worth noting that [2] analyzed the **runtime** characteristics of MPI programs (in an HPC center) while we analyze the **static** characteristics of general MPI programs; clearly both studies differ on the target characteristics and scope of the data. It is therefore worthwhile to discuss the findings of both studies.

Table 2 shows a comparison of our findings and the findings in [2]. The table shows the main seven insights we find and whether they were reported before in [2] or not. We find that three out of seven insights were also reported previously. These insights are mostly related to the use of different classes of MPI calls and functionality, which was the goal of [2] since they focus on executed programs. Findings 4–7 are more oriented towards other aspects of MPI programs, such as compatibility with the standard versions, programming languages utilized, and unused features of the standard—since we are the first analyzing these aspects, these insights have not been reported before.

This comparison highlights that our study serves as a confirmation of some previously reported trends in MPI usage (however, at a larger scale), as well as reporting new insights.

10 CONCLUSIONS

While previous works have studied the use of MPI, the vast majority of these works focused either on a small set of programs or on programs executed in specific HPC centers. Our work presents the first study based on data from a large number of MPI programs,

	Insight	First Time Reported?	Comments
1	Hybrid MPI+X applications are more widely used than expected.	No	[2] reports a similar insight
2	`MPI_Allreduce` is the most significantly used collective.	No	[2] reports a similar insight
3	Applications use more non-blocking point-to-point calls than blocking calls.	No	[2] reports a similar insight
4	The majority of the MPI programs use only a small set of features from the MPI Standard.	Yes	
5	A large proportion of programs (42%) require the features in MPI version 1.0 only; most programs (about 80%), require only the features of MPI version 2.0.	Yes	
6	The most significant language in MPI programs is C++.	Yes	
7	Some features of the MPI Standard, such as *process management* are practically unused.	Yes	
8	The MPI Standard version 2.0 may have significantly allowed developers to explore the use of new MPI features.	Yes	
9	Programs mostly use `MPI_THREAD_FUNNELED` as the required level of thread support.	Yes	

Table 2: Comparison of our findings with the previous studies of MPI usage.

analyzing a total of 110 programs. To the best of our knowledge, this is the largest study reported so far on MPI usage.

A significant challenge in examining a large sample of MPI programs is the cost in time and effort of building and running programs—we avoid this by statically analyzing programs using an analysis framework that can quickly infer several metrics of MPI programs without the need of building and/or running the programs. Our static analysis framework allows us to quickly scale to large samples sets of MPI programs and therefore gives us the opportunity to obtain a larger view of MPI usage in the community.

While our study confirms some insights of MPI usage previously reported in studies that used smaller data samples, it also reveals a number of surprising previously unreported insights. We find that the majority of MPI programs use only a small set of features from the MPI Standard; 42% programs need only functionality from MPI 1.0 and 80% need only the functionality from MPI 2.0. This interesting finding has implications into where effort should be concentrated in MPI implementations and in the process of standardizing new features in MPI. We report that blocking and non-blocking point-to-point operations are more widely used than persistent or one-sided routines—a surprising finding given that the latter could potentially improve performance over the former. We also find that hybrid MPI+X applications populate a considerable portion of the space (29%), and that, surprisingly, C++ dominates the programming language use over Fortran and C. We speculate that as C++ evolves, more projects are adopting C++ perhaps to support more advanced language features.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their suggestions and comments on the paper. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-771757). This work also was supported in part by the National Science Foundation under grants 1821431, 1812404, 1925603, and 1918987. 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] David E Bernholdt, Swen Boehm, George Bosilca, Manjunath Gorentla Venkata, Ryan E Grant, Thomas Naughton, Howard P Pritchard, Martin Schulz, and Geoffroy R Vallee. 2017. A survey of MPI usage in the US exascale computing project. *Concurrency and Computation: Practice and Experience* (2017), e4851.

[2] S. Chunduri, S. Parker, P. Balaji, K. Harms, and K. Kumaran. 2018. Characterization of MPI Usage on a Production Supercomputer. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 386–400. https://doi.org/10.1109/SC.2018.00033

[3] Al Danial. 2019. CLOC: Count Lines of Code. https://github.com/AlDanial/cloc. Online; accessed Jan/10/2019.

[4] H Carter Edwards, Christian R Trott, and Daniel Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3202–3216.

[5] Message Passing Interface Forum. 2015. *MPI: A Message-passing Interface Standard, Version 3.1 ; June 4, 2015.* High-Performance Computing Center Stuttgart, University of Stuttgart. https://books.google.com/books?id=Fbv7jwEACAAJ

[6] Robert B Grady. 1992. *Practical software metrics for project management and process improvement.* Prentice-Hall, Inc.

[7] Richard D Hornung and Jeffrey A Keasler. 2014. *The RAJA portability layer: overview and status.* Technical Report. Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States).

[8] Benjamin Klenk and Holger Fröning. 2017. An overview of MPI characteristics of exascale proxy applications. In *International Supercomputing Conference*. Springer, 217–236.

[9] Luke Nguyen-Hoan, Shayne Flint, and Ramesh Sankaranarayana. 2010. A Survey of Scientific Software Development. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '10)*. ACM, New York, NY, USA, Article 12, 10 pages. https://doi.org/10.1145/1852786.1852802

[10] Prakash Prabhu, Hanjun Kim, Taewook Oh, Thomas B Jablin, Nick P Johnson, Matthew Zoufaly, Arun Raman, Feng Liu, David Walker, Yun Zhang, et al. 2011. A survey of the practice of computational science. In *SC'11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.

[11] L. Prechelt. 2000. An empirical comparison of seven programming languages. *Computer* 33, 10 (Oct 2000), 23–29. https://doi.org/10.1109/2.876288

[12] Rolf Rabenseifner. 1999. Automatic MPI counter profiling of all users: First results on a CRAY T3E 900-512. In *Proceedings of the message passing interface developer's and user's conference*, Vol. 1999. 77–85.

[13] Nawrin Sultana, Anthony Skjellum, Purushotham Bangalore, Ignacio Laguna, and Kathryn Mohror. 2018. Understanding the Usage of MPI in Exascale Proxy Applications. Workshop on Exascale MPI (ExaMPI). (2018).

[14] Steven P VanderWiel, Daphna Nathanson, and David J Lilja. 1997. Complexity and performance in parallel programming languages. In *Proceedings Second International Workshop on High-Level Parallel Programming Models and Supportive Environments*. IEEE, 3–12.

[15] H. Zhang. 2009. An investigation of the relationships between lines of code and defects. In *2009 IEEE International Conference on Software Maintenance*. 274–283.

A LIST OF PROGRAMS

Table 3 shows the list of MPI programs that we analyze in our study.

Table 3: The set of MPI application used in our study.

No.	Program Name	Version	Latest Release
1	ADCIRC	53.04	3/18/19
2	ADIOS	1.13.1	4/1/18
3	AMG2013	2.3	1/6/14
4	ASPA		9/23/14
5	BioEM	2	12/18/18
6	CFDEMcoupling	3.8.0	11/30/17
7	CGM	16	1/4/19
8	CLAMR	2.0.7	11/23/14
9	Cactus	4.3.0	3/1/19
10	Caffe-MPI	2	2/5/18
11	Chatterbug	1	8/27/18
12	Chombo	3.2	12/7/15
13	CloverLeaf	1.3	10/28/15
14	CoMD	1.1	10/26/17
15	CoSP2		1/29/15
16	Code_Saturne	5.3.2	1/29/19
17	Comb	0.1.1	1/8/19
18	CoreNeuron	0.14	2/28/19
19	E3SM	v1.0.0	2/28/19
20	EigenExa	2.4b	8/20/18
21	Elemental	v0.87.7	2/6/17
22	Enzo	branch_dev-bd5ddbbcb6ad	2/5/19
23	ExaMiniMD	1	3/14/18
24	FLASH	4.5	3/1/19
25	GTC-P	gtcp_apex-c5bf52e7cfe5	9/18/18
26	Gadget	2.0.7	5/1/05
27	Gamess	R3-Public-Release	9/30/18
28	Genesis	1.3.0	6/18/18
29	Gizmo		3/25/19
30	Gromacs	2019.1	2/15/19
31	HPCG	3	11/11/15
32	HPGMG		2/1/18
33	HYPRE	2.11.2	3/13/17
34	HemeLB	0.2.6	3/7/14
35	Kratos	7	3/20/19
36	Kripke	1.2.3	10/23/18
37	LAMMPS	12-Dec-18	12/11/18
38	LULESH	2.1	8/30/17
39	Laghos	2	11/19/18
40	MACSio	1.1	10/2/18
41	MCB	1	1/6/14
42	MITgcm	cp67g	1/7/19
43	MLSL	v2018.2	10/1/18
44	MOAB	5.1.0	3/27/19
45	MUMPS	5.1.2	10/1/17
46	MeshKit	1.5.0	3/4/19
47	NAMD	2.13	4/3/19
48	NWChem	6.8.1	6/14/18
49	Neckbone	2.3.4.1	1/6/14
50	Nek5000	17	12/17/17
51	Nyx	18.1	10/2/18
52	OpenFOAM	18.12	12/20/18
53	OpenMD		3/26/19
54	PENNANT	0.9	3/2/16
55	PICSAR		2/7/19
56	Paradis	2.5.1.1	11/15/11
57	PlasmaApp		1/9/14
58	PrincetonCBEMDMPI		1/17/13
59	ProfugusMC		8/31/18
60	QBox	1.2	1/6/14
61	Qmcpack	3.6.0	12/19/18
62	Quantum_Espresso	6.4	3/4/19
63	SCOREC	2.2.0	10/19/18
64	SNbone		2/11/15
65	SPPARKS	0.14	12/20/18
66	SW4lite	1	3/14/19
67	SWFFT	1	10/25/17
68	Samrai	v-3-12-0	5/31/18
69	ScaLAPACK	2.0.2	5/1/12
70	Scale	5.33	2/6/19
71	SimpleMOC	4	7/31/18
72	TCLB	6	1/30/19
73	Trilinos	release-12-14-1	2/27/19
74	Tycho2		3/9/19
75	VMD	1.9.3	11/30/16
76	VPFFT		3/5/15
77	WRF	4	6/8/18
78	abinit	8.10.2	10/15/18
79	amgcl	1.2.0	5/10/18
80	arbor	0.2	3/3/19
81	ascent	0.4.0	10/1/18
82	cp2k	6.1	10/2/18
83	elmerfem	release-8.4	3/20/19
84	faunus	2.2.0	2/20/18
85	feltor	v5.0	6/15/18
86	ffm_mpi		12/3/17
87	horovod	0.16.1	3/19/19
88	meep	1.8.0	2/13/19
89	metag_partitioning		9/29/16
90	miniAMR	1.4.3	2/14/19
91	miniAero	1.0.0	7/5/16
92	miniFE	2.2	11/22/17
93	miniMD	1.2	2/28/19
94	miniSMAC2D	2.0.0	7/5/16
95	miniTri	1	10/23/17
96	miniVite	1	9/26/18
97	miniXyce	1.0.0	7/5/16
98	mxx		1/4/19
99	nest-simulator	v2.16.0	8/21/18
100	ntchem-mini	1.2	3/15/19
101	phyml	v3.3.20190321	4/3/19
102	pism	1.1	3/29/19
103	primecount	4.5	2/23/19
104	pypar		11/10/16
105	rocALUTION	1.4.1	3/17/19
106	siesta	4.0.2	7/20/18
107	souffle	1.5.1	1/20/19
108	splatt	1.1.0	9/5/18
109	thornado-mini	1	9/24/18
110	yambo	4.3.2	2/2/19

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

We ran mpiusage.py, which parses the source code of MPI programs. The input of this artifact is a directory containing the source code of a program, and the output is metrics of MPI usage.

The MPI Usage program will by default output statistics in a json format in standard output, for example:

$ ./mpiusage.py /path/lulesh/src/ { "MPI_COMM_SIZE": 3, "MPI_REDUCE": 3, "MPI_WAITALL": 2, "MPI_WAIT": 116, "MPI_COMM_RANK": 15, "MPI_ALLREDUCE": 3, "MPI_INIT": 3, "MPI_FINALIZE": 3, "MPI_ABORT": 12, "MPI_WTIME": 3, "MPI_IRECV": 26, "MPI_ISEND": 52, "MPI_BARRIER": 3, "OPENMP": 0, "OPENACC": 0, "CUDA": 168, "OPENCL": 0, "C_LINES": 0, "CPP_LINES": 297, "C_CPP_H_LINES": 365, "FORTRAN_LINES": 0, "LINES_OF_CODE": 14574 }

The user can specify an output file using the -o.

For help the user can use the -h option:

$ ./mpiusage.py -h usage: mpiusage.py [-h] [-o OUTPUT] [-v] path

positional arguments: path file or directory to analyze

optional arguments: -h, –help show this help message and exit -o OUTPUT, –output OUTPUT name of output file -v, –verbose print what the script does

ARTIFACT AVAILABILITY

Software Artifact Availability: All author-created software artifacts are maintained in a public repository under an OSI-approved license.

Hardware Artifact Availability: There are no author-created hardware artifacts.

Data Artifact Availability: There are no author-created data artifacts.

Proprietary Artifacts: None of the associated artifacts, author-created or otherwise, are proprietary.

List of URLs and/or DOIs where artifacts are available:

`https://doi.org/10.11578/dc.20190410.3`

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: MacBook Pro, 3.3 GHz Intel Core i7, 16 GB 2133 MHz LPDDR3

Operating systems and versions: Mac OS Darwin

Compilers and versions: N/A

Applications and versions: Described in the paper

Libraries and versions: Described in the paper

Key algorithms: Static analysis (as described in the paper)

Input datasets and versions: 110 MPI programs (described in the paper)

Paper Modifications: We didn't make modifications

Output from scripts that gathers execution environment information.

```
$ ./collect_environment.sh
TERM_PROGRAM=Apple_Terminal
TERM=xterm-256color
SHELL=/bin/tcsh
TMPDIR=/var/folders/tj/9109txhn0jg_79c471ksgd880013t⌋
↪ 6/T/
Apple_PubSub_Socket_Render=/private/tmp/com.apple.la⌋
↪ unchd.QOwuxm8ogC/Render
TERM_PROGRAM_VERSION=388.1.3
TERM_SESSION_ID=818CBCAD-B17D-4DD0-9EAC-EB02595337DB
USER=USER
GROUP=unknown
SSH_AUTH_SOCK=/private/tmp/com.apple.launchd.JVndrTd⌋
↪ qCW/Listeners
HOSTTYPE=unknown
__CF_USER_TEXT_ENCODING=0x8F46:0x0:0x0
PWD=.../AD_script/Author-Kit
LANG=en_US.UTF-8
XPC_FLAGS=0x0
XPC_SERVICE_NAME=0
HOME=/Users/USER
SHLVL=3
OSTYPE=darwin
VENDOR=apple
LOGNAME=USER
MACHTYPE=x86_64
DISPLAY=/private/tmp/com.apple.launchd.TKJTqhwxgP/or⌋
↪ g.macosforge.xquartz:0
+ lsb_release -a
./collect_environment.sh: line 10: lsb_release:
↪ command not found
+ uname -a
Darwin unsullied.llnl.gov 16.7.0 Darwin Kernel
↪ Version 16.7.0: Thu Dec 20 21:53:35 PST 2018;
↪ root:xnu-3789.73.31~1/RELEASE_X86_64 x86_64
+ lscpu
./collect_environment.sh: line 12: lscpu: command not
↪ found
+ cat /proc/cpuinfo
cat: /proc/cpuinfo: No such file or directory
+ cat /proc/meminfo
cat: /proc/meminfo: No such file or directory
+ inxi -F -c0
./collect_environment.sh: line 14: inxi: command not
↪ found
+ lsblk -a
./collect_environment.sh: line 15: lsblk: command not
↪ found
```

```
+ lsscsi -s
./collect_environment.sh: line 16: lsscsi: command
↪  not found
+ module list
./collect_environment.sh: line 17: module: command
↪  not found
+ nvidia-smi
./collect_environment.sh: line 18: nvidia-smi:
↪  command not found
+ cat
+ lshw -short -quiet -sanitize
./collect_environment.sh: line 19: lshw: command not
↪  found
+ lspci
./collect_environment.sh: line 19: lspci: command not
↪  found
```