Why Is MPI So Slow? Analyzing the Fundamental Limits in Implementing MPI-3.1

Ken Raffenetti
Argonne National Laboratory
raffenet@mcs.anl.gov

Abdelhalim Amer
Argonne National Laboratory
aamer@anl.gov

Lena Oden
Argonne National Laboratory
loden@anl.gov

Charles Archer
Intel Corporation
charlesarcher@gmail.com

Wesley Bland
Intel Corporation
wesley.bland@intel.com

Hajime Fujita
Intel Corporation
hajime.fujita@intel.com

Yanfei Guo
Argonne National Laboratory
yguo@anl.gov

Tomislav Janjusic
Mellanox Technologies
tomislavj@mellanox.com

Dmitry Durnov
Intel Corporation
dmitry.durnov@intel.com

Michael Blocksome
Intel Corporation
michael.blocksome@intel.com

Min Si
Argonne National Laboratory
msi@anl.gov

Sangmin Seo
Argonne National Laboratory
sseo@anl.gov

Akhil Langer
Intel Corporation
akhil.langer@intel.com

Gengbin Zheng
Intel Corporation
gengbin.zheng@intel.com

Masamichi Takagi
RIKEN Advanced Institute of Computational Science
masamichi.takagi@riken.jp

Paul Coffman
Argonne National Laboratory
pcoffman@anl.gov

Jithin Jose
Intel Corporation
jithinjose@gmail.com

Sayantan Sur
Intel Corporation
sayantan.sur@intel.com

Alexander Sannikov
Intel Corporation
alexander.sannikov@intel.com

Sergey Oblomov
Intel Corporation
sergey.oblomov@intel.com

Michael Chuvelev
Intel Corporation
michael.chuvelev@intel.com

Masayuki Hatanaka
RIKEN Advanced Institute of Computational Science
mhatanaka@riken.jp

Xin Zhao
Mellanox Technologies
xinz@mellanox.com

Paul Fischer
University of Illinois
fischerp@illinois.edu

Thilina Rathnayake
University of Illinois
rbr2@illinois.edu

Matt Otten
Cornell University
mjo98@cornell.edu

Misun Min
Argonne National Laboratory
mmin@mcs.anl.gov

Pavan Balaji
Argonne National Laboratory
balaji@anl.gov

ABSTRACT

This paper provides an in-depth analysis of the software overheads in the MPI performance-critical path and exposes mandatory performance overheads that are unavoidable based on the MPI-3.1 specification. We first present a highly optimized implementation of the MPI-3.1 standard in which the communication stack—all the way from the application to the low-level network communication API—takes only a few tens of instructions. We carefully study these instructions and analyze the root cause of the overheads based on specific requirements from the MPI standard that are unavoidable under the current MPI standard. We recommend potential changes to the MPI standard that can minimize these overheads. Our experimental results on a variety of network architectures and applications demonstrate significant benefits from our proposed changes.

SC17, November 12–17, 2017, Denver, CO, USA

ACM ISBN 978-1-4503-5114-0/17/11…$15.00
https://doi.org/10.1145/3126908.3126963

CCS CONCEPTS

• **Computing methodologies → Concurrent algorithms**; *Massively parallel algorithms*;

ACM Reference format:
Ken Raffenetti, Abdelhalim Amer, Lena Oden, Charles Archer, Wesley Bland, Hajime Fujita, Yanfei Guo, Tomislav Janjusic, Dmitry Durnov, Michael Blocksome, Min Si, Sangmin Seo, Akhil Langer, Gengbin Zheng, Masamichi Takagi, Paul Coffman, Jithin Jose, Sayantan Sur, Alexander Sannikov, Sergey Oblomov, Michael Chuvelev, Masayuki Hatanaka, Xin Zhao, Paul Fischer, Thilina Rathnayake, Matt Otten, Misun Min, and Pavan Balaji. 2017. Why Is MPI So Slow? Analyzing the Fundamental Limits in Implementing MPI-3.1. In *Proceedings of SC17, Denver, CO, USA, November 12–17, 2017,* 12 pages.
https://doi.org/10.1145/3126908.3126963

1 INTRODUCTION

MPI is widely considered to be the de facto standard for communication on distributed-memory systems. Despite its wide success, however, MPI is often criticized as being a heavyweight runtime system that can add significant overhead, particularly for applications that need very fine-grained communication on fast networks, such as those relying on strong scaling on large supercomputing systems. There is some truth in such criticism. For instance, MPI provides a generalized API where multiple kinds of communication are funneled through the same function API. That is, whether the application wants to send a small single-integer contiguous buffer or a large noncontiguous multidimensional data structure, the communication is funneled through `MPI_ISEND`, for example. The MPI implementation internally needs to check for and distinguish these different communication patterns, thus adding overhead to the performance-critical path.

Most publications that draw such conclusions evaluate specific implementations of MPI without carefully distinguishing whether the performance degradation is an artifact of the MPI implementation or the MPI standard. If the MPI implementation is performing poorly, is it because the design or implementation choices made in the MPI library are suboptimal for performance? Or is it because of fundamental limitations in the MPI standard that make a more optimized implementation impossible?

In this paper we try to understand and illustrate what the shortcomings of the current MPI-3.1 standard [26] are for applications that rely on time to solution at the strong-scaling limit of the problem they are trying to solve. We use a three-step approach.

1. We first present a highly optimized implementation of the MPI-3.1 standard, such that the communication stack—all the way from the application to the low-level network communication API—takes only a few tens of instructions. The purpose of this implementation is to demonstrate the "best case" for the shortest path from the application to the network communication API.

2. We then carefully study the instruction counts in each aspect of this implementation. The focus here is on accounting for each instruction and associating it with the requirements in the MPI standard that result in that instruction.

3. We use this analysis to recommend potential changes to the MPI standard to address the overheads. We also evaluate these proposed changes and showcase the performance improvements such changes can achieve.

We analyze a small subset of the communication routines with examples from point-to-point communication and one-sided communication and use them to illustrate the kind of overheads each of these calls has to deal with. In addition to a detailed description and analysis of the highly optimized MPI implementation, this paper presents performance evaluation results from a variety of benchmarks and applications on a number of hardware platforms. We evaluate the applications close to their strong-scaling limit in order to showcase the core capability range that this paper targets.

Recommended reading. This paper is not meant to be a tutorial on MPI. It assumes that the reader is fairly familiar with the MPI standard and has some notion of the workings of MPI implementations. For readers unfamiliar with these aspects, we recommend reading [20, 21, 25] to obtain the relevant background before reading this paper. These references provide an easier-to-read alternative to the MPI standard, although in doing so they sometimes introduce approximations to the true intent of the standard. We caution all readers that the only authoritative reference to correctness in MPI is the latest MPI standard.

What this paper does not handle. The goal of this paper is to provide an overview of the complexities an MPI implementation has to address and the associated overheads these complexities incur. While the MPI standard defines a large number of communication functions, this paper does not focus on all of them and instead picks a small sample set of functions and studies their overheads. The intent here is to give a rough idea of these overheads; we expect the reader to be able to extend these observations to other functions. In particular, this paper makes no mention of some segments of the MPI standard such as MPI I/O, dynamic processes, threads, and Fortran bindings, all of which would impose additional checks and requirements on the MPI implementation.

The rest of the paper is organized as follows. In Section 2 we present a highly optimized implementation of MPI including possible optimizations to minimize the instruction counts in the performance-critical path. In Section 3 we analyze the root cause of the overheads in the MPI standard. Performance results showcasing our implementation and analysis are presented in Section 4. Other work related to our research is described in Section 5, followed by concluding remarks in Section 6.

2 HIGHLY OPTIMIZED MPI IMPLEMENTATION

Before we discuss what the fundamental limits in implementing MPI-3.1 are, we need to first understand what *can* be optimized within the confines of the MPI standard. Therefore, in this section, we present a highly optimized implementation of the MPI-3.1 standard. This implementation is done in the context of MPICH.

MPICH is a multilayered software stack. In the topmost layer, known as the "MPI layer," MPICH implements machine-independent code such as machine-independent collectives, derived datatype management, and group management. Below the MPI layer comes the machine-dependent code, known as the "abstract device" or simply the "device" layer. Since this layer is machine dependent, multiple implementations of it exist, each suited to a

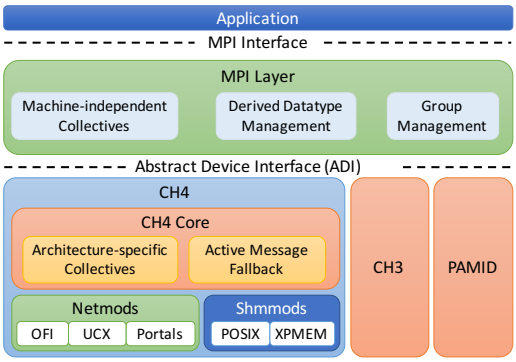

Figure 1: Overview of the MPICH software stack.

different architecture, although some architectures share a common device. MPICH currently provides two devices: `ch3` (for most platforms) and `pamid` (for the IBM Blue Gene/Q). For the optimized implementation that we are describing here, we developed a new device called `ch4`. Each device internally has further abstract layers for managing specific aspects of the code.

We architect `ch4` from the ground up, keeping low instruction and cycle counts as a primary design goal. Artifacts of this goal will become evident in a number of design choices that we describe in this section. The overall architecture of `ch4` can be viewed as a combination of three layers: the `ch4 core`, the `netmods`, and the `shmmods` (see Figure 1). The `ch4` infrastructure is designed to allow flow-through of most MPI-level information all the way down to the `netmods` and `shmmods`. That is, the `netmods` and `shmmods` know what MPI-level call triggered a particular data movement operation, including all its parameters. Thus, each `netmod` and `shmmod` can decide the best way to implement the operation. If it does not have a network or shared-memory-specific method for optimizing that communication operation, it simply falls back to the active-message-based implementation provided by the `ch4 core`. For instance, let us consider the `MPI_PUT` operation and walk through the steps it takes.

MPI Layer: The MPI library, MPICH in this case, provides an entry point for `MPI_PUT` as a function call, which also is aliased to another function call `PMPI_PUT` (the equivalent function in the profiling interface). The `MPI_PUT` function is implemented at the MPI layer inside the MPICH library, where three actions are performed: (1) we check for any errors in the parameters, such as bad arguments or invalid MPI objects being referenced by the call; (2) we look up the actual MPI window object in whose context the communication is taking place: this window object describes the scope of processes and memory that is accessible for that communication; and (3) we check to see whether the communication needs to be thread safe, and accordingly we take either the thread-safe or the thread-unsafe communication path.

CH4 Core: Once these three actions at the MPI layer are done, the control passes down to the appropriate device, which is `ch4` in this case. First, `ch4 core` does a check for locality. When the target is self), `ch4 core` handles the communication; when the target is on the same node, it hands over control to the `shmmod`; and when the target is on a different node, it hands over control to the `netmod`.

netmod/shmmod: Once the `ch4 core` passes down control to the `netmod/shmmod`, we need to analyze the operation and decide whether it can be fully implemented in hardware or whether it needs to be implemented as an active-message fallback. For instance, suppose the network hardware can implement `MPI_PUT` only in hardware (as an RDMA operation) for simple contiguous data but needs to fall back to active messages for more complex data layouts. In this case, a check is performed at the `netmod`, and a branch is added either to implement the `MPI_PUT` operation "natively" in the `netmod` or to fall back to the active-message implementation in the `ch4 core`. If the operation can be supported natively, which we refer to as the fast-path for the communication operations, the `netmod` then performs the necessary translation of the MPI-level parameters to network-level parameters (e.g., MPI target offset to OS virtual address) and performs one or more network operations to correspond with the MPI operation.

To summarize, we highlight two takeaways in the way the different layers of `ch4` are designed.

(1) The communication fast-path, which is the most common performance-critical path that MPI data movement takes, flows as directly as possible to either the netmod or the shmmod using the fewest instructions.
(2) The communication semantics (e.g., which MPI function is being used to perform this data movement) are never lost all the way through the software stack. Thus, any layer of the stack can freely perform communication optimizations with a full view of the communication path.

In the next few subsections, we present a detailed instruction and cache-level analysis of the MPICH/CH4 communication stack. We have two goals: (1) identifying the smallest number of instructions that an MPI implementation needs in order to implement the full MPI standard and (2) determining where the remaining instructions are spent and what aspects of the MPI standard cause these instruction overheads.

2.1 Instruction-Count Analysis of the MPICH/CH4 Stack

Let us first analyze the number of instructions used by the MPICH/CH4 stack in a simple "default" build.[1] This build enables a number of features that make it friendlier for users and administrators, although it is not fully optimized for performance. The configuration includes checking for errors in arguments, ensuring that datatypes are correctly created and committed before being used, and making certain that the target process rank is within the communicator range. In this configuration, the MPICH/CH4 stack takes 221 instructions for `MPI_ISEND` and 215 instructions for `MPI_PUT`: these instructions cover the entirety of instructions contributed by the MPI implementation, all the way from the application to the network communication API. In other words, this is the additional number of instructions that would be used if the application used MPI instead of directly using the low-level network communication API. We omit analysis of `MPI_IRECV`, as the software path is largely identical to `MPI_ISEND` for network APIs that support matching.

[1]The default build enables both shared and static libraries; we used static libraries in our experiments.

The first observation we make is that this is already a significant improvement in instruction count compared with that of traditional MPI implementations. For instance, the MPICH/CH3 device (referred to as MPICH/Original in the rest of the paper), which is used in multiple MPI implementations such as MVAPICH, Intel MPI, and Cray MPI, takes 253 instructions for `MPI_ISEND` and 1,342 instructions for `MPI_PUT`—a reduction of 13% and 84% for `MPI_ISEND` and `MPI_PUT`, respectively. The bulk of these savings is attributed to the low-instruction fast-path that MPICH/CH4 provides compared with most other MPI implementations.

While these savings are useful, the real question is, Where are the remaining instructions being used? We carefully analyzed the instructions being used and mapped them to the source path used for `MPI_ISEND` and `MPI_PUT` in order to identify the source of these instructions. Our analysis results are listed in Table 1.

Table 1: Instruction analysis for MPI calls

Reason	`MPI_ISEND`	`MPI_PUT`
Error checking	74 instructions	72 instructions
Thread-safety check	6 instructions	14 instructions
MPI function call	23 instructions	25 instructions
Redundant runtime checks	59 instructions	62 instructions
MPI mandatory overheads	59 instructions	44 instructions

Some of the overheads described in Table 1 are easy to work around. For instance, although error checking is a useful feature—it looks for incorrect arguments, invalid buffer accesses, and other such user errors—it is, technically, not mandated by the MPI standard. Thus, a valid optimization would be to provide two builds of the MPI library—one with error checking enabled and the other without—so that applications looking for extreme performance would have their pick.

Thread-safety checks are, in some sense, similar to error checks. In theory, two different MPI libraries could be provided: one that is thread safe and another that is not. Combining these into a single library and having them check for thread-safety requirements at runtime is really a software distribution optimization to deal with users who might accidentally link to the wrong library. It has the side effect, however, of costing additional instructions even when thread safety is not needed.

MPI function call overheads are trickier to deal with. Each MPI function call can take around 16–18 instruction just to load the stack and registers before the function can start executing. This also has the side effect that since the compiler generally views the function call as a black box, some unnecessary or redundant runtime checks might need to be performed by the MPI library. For instance, suppose the application calls `MPI_ISEND` for data represented with `MPI_DOUBLE`. Since the MPI function call is generic and can be used for any datatype, the MPI library cannot assume the usage of any specific datatype by the user and must check for what datatype is used in order to calculate the actual size of the data being transmitted at runtime. In theory, since the application provided a constant datatype, the exact size of the data is known at compile/link time, and thus checking for it at runtime is unnecessary—this check is simply an artifact of the MPI library exposing its communication routines as functions rather than as compile-time macros.

2.2 Link-Time Inlining

One can work around the MPI function call and the associated redundant runtime check overheads by using interprocedural and link-time optimizations, such as link-time inlining. Link-time inlining must be used carefully, however, since it increases the total number of instructions used and can potentially result in additional instruction cache misses if used indiscriminately, especially for large applications. Fortunately, most standard compilers (including the Intel compiler suite, GCC, and LLVM) provide sophisticated link-time inlining capabilities that enable specific files to be targeted for inlining while leaving other files behind.

The first set of functions we consider are those in the MPI library. The ability to inline specific files gives an opportunity for the MPI implementation to inline performance-critical functions, such as `MPI_ISEND` and `MPI_PUT`, while leaving noncritical functions, such as `MPI_INIT`, as regular function calls. Doing so significantly reduces the amount of inlined code and consequently the number of redundant instructions generated.

While inlining just the MPI function calls removes the function call overhead, as shown in Table 1, it might not always be able to remove the redundant runtime checks overhead. We therefore surveyed 62 applications in order to understand their usage of datatypes as an example of the redundant runtime checks. This survey comprised the NAS parallel benchmarks [6], CORAL benchmarks [3], DOE codesign applications [1, 2], and other large applications that consume significant compute cycles at large supercomputing centers [17, 33]. Based on our survey, we divided the application usage of datatypes into three classes.

- **Class 1 (derived datatypes):** Most applications primarily use predefined datatypes in their performance-critical path. The use of derived datatypes was observed only in two applications, HACC [23] and MCB [5]; and both applications use such datatypes in the setup phase and not the performance-critical path.
- **Class 2 (predefined datatypes as compile-time constants):** Several applications directly use compile-time constants such as `MPI_DOUBLE` or `MPI_INT` in the MPI communication call to describe the data layout.
- **Class 3 (predefined datatypes as runtime constants):** A few applications use predefined datatypes but not directly as compile-time constants passed in the MPI communication call. Instead, they perform some runtime checks to find the datatype to use but then use it as a runtime constant throughout the application.

Of these three classes of applications, we target the latter two for performance optimization because of their widespread use in the performance-critical path of most applications.

For the second class of datatype usage, simply inlining the MPI function calls is sufficient. If the compiler can see that a constant is being used for the datatype, it can calculate the datatype properties, including its size, at compile time, thus avoiding the additional instructions at runtime. In such cases, for all practical purposes, the code would function in a way that is similar to forcing MPI to assume the usage of a single datatype by the application.

The third class of datatype usage is the trickiest. Simply inlining the MPI function call is not sufficient to remove the datatype checks

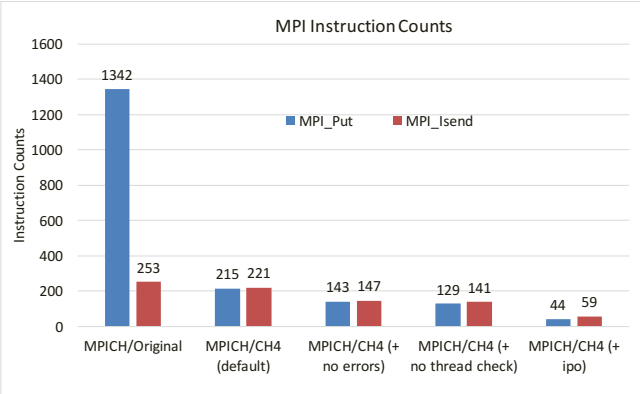

Figure 2: Instruction counts.

since the compiler will not be able to tell that the datatype variable is indeed constant for the remaining lifetime of the application. This behavior is used in applications primarily as an interlibrary type conversion mechanism to provide type compatibility between multiple libraries and languages. This is usually done by creating a user-defined `MPI_Datatype` and conditional mapping it to a predefined datatype. For example, LULESH [4] maps the user-defined `baseType` to `MPI_FLOAT` or `MPI_DOUBLE` depending on the size of `Real_t`. While the size of `Real_t` is available at compile time, the assignment of the datatype is often hidden inside wrapper functions, making it hard for the compiler to know that the variable `baseType` is indeed immutable once it has been assigned. Consequently, when the MPI communication routine is called, it sees only the `baseType` variable and must do a runtime check in order to understand what predefined type that variable corresponds to. Nekbone [7] performs a check similar to that of LULESH, although it uses a switch statement within an internal function to perform this assignment and returns the type as a function parameter, thus making it hard for the compiler to understand. Other applications such as QMCPACK [8], LSMS [9], and miniFE [24] use a C++ template to implement this type mapping, although conceptually the mapping is equivalent to that in LULESH.

For such applications, the link-time inlining needs to be expanded to include more files and functions potentially subsuming all files in the application and its libraries into the link-time inlined executable. We tested each applications by expanding the scope of link-time inlining to subsume the entire application. As expected, the datatype checks eventually turn into compile-time checks without any redundant runtime checks when enough of the application is subsumed into the inlined code, although at the cost of a largely increased instruction count for the application executable itself.

2.3 Summary of Instruction Analysis

Figure 2 shows a summary of the instruction counts in the MPICH/CH4 library with various capabilities disabled. In particular, we notice that after including all the performance optimizations described above, the MPICH/CH4 stack uses as few as 59 instructions for `MPI_ISEND` and 44 instructions for `MPI_PUT`—an overall reduction of 77% and 97% compared with the default build of MPICH/Original, respectively.

So, what's remaining? Despite the instruction count reductions described, the MPI library still takes 44–59 instructions in the performance-critical path, encompassed in the large black box of "mandatory MPI overheads" in Table 1. We have identified six such overheads in the MPI implementation that cannot be removed without modifying the MPI standard beyond MPI-3.1. In Section 3 we present details of these overheads, including potential changes to the MPI standard to address these overheads.

3 SHORTCOMINGS IN MPI

In Section 2 we presented a highly optimized implementation of MPI-3.1. Using this, we demonstrated that with the right set of implementation optimizations we can build a lightweight implementation of MPI: one that takes only a few tens of instructions all the way from the application to the network communication API. While we may congratulate ourselves on this achievement, a keen reader might observe that MPI's role is really to expose as much of the network performance as possible. Thus, the real question is why the instruction count from the application to the network communication API is not *zero*. That is, why is MPI taking any instructions at all?

To answer that question, we present an analysis of what we *cannot* optimize in an implementation of MPI, at least not without improving the MPI standard beyond MPI-3.1.

We acknowledge that performance is not the only metric used in real application development. Productivity—with respect to simplicity in writing code, debugging, and maintaining newer software releases—is important, too. However, the focus of this paper is on performance: where we are losing performance and what can be done to regain that performance. Whether such changes hurt productivity is an orthogonal question that is not addressed in this paper.

3.1 Network Address Virtualization with MPI Communicators

MPI processes are logically represented as integer ranks within opaque objects known as communicators. A process provides a communicator and a rank within that communicator to specify whom to send data to (or receive data from). This communicator/rank tuple is then internally converted by the MPI implementation to a physical network address that is used for the communication. The communicator, in some sense, provides a level of virtualization between the physical network addresses and the application. An application can create multiple communicators, each with a different mapping of integer ranks to physical network addresses.

Converting the communicator rank to the network address is not trivial and can cost several instructions. The simplest address translation model uses an array lookup, where a separate array is maintained for each communicator that provides a mapping from the rank in that communicator to the network address. This model provides a lookup mechanism in two instructions; but at least one of those instructions is a memory dereference, making it expensive in practice. More important, such simple lookup mechanisms are extremely expensive with respect to the memory used, since they maintain an $O(P)$ table for each communicator, where P is the number of processes in the communicator. More sophisticated

mechanisms, such as those described in [22], optimize the memory usage of such network address lookups but can further increase the instruction counts to around 11 instructions even for simple communicators.

Proposal for the MPI Standard: We propose a mechanism that would allow the application to bypass the communicator virtualization layer for network address translation by querying for the physical network address index. The execution of this idea would involve two steps. In the first step, if the application wanted to communicate with rank X in its communicator, it would translate X to the equivalent rank in `MPI_COMM_WORLD`. This could be done by using the function `MPI_GROUP_TRANSLATE_RANKS` that already exists in the current MPI standard. In the second step, the application would communicate with this translated rank by using new specialized functions that would assume `MPI_COMM_WORLD` ranks, for example, `MPI_ISEND_GLOBAL`.

Such a model would be useful for applications that communicate with a small set of processes. An example would be a five-point stencil computation on a Cartesian grid where the application could simply store the `MPI_COMM_WORLD` ranks of its north, south, east, and west neighbors in four separate variables and use those for the appropriate communication.

We note that the remaining parameters passed to `MPI_ISEND_GLOBAL` would still be the same as `MPI_ISEND`, including the user communicator that provides the context isolation for the communication. The only difference compared with `MPI_ISEND` is that the rank provided would be that in `MPI_COMM_WORLD` rather than that in the actual communicator being used. We also note that this function would not be "intercommunicator-safe." That is, one could not use this function for communicating across processes that belong to different `MPI_COMM_WORLD` communicators.

Instruction Savings: We implemented this proposal in MPICH/CH4 and noticed a reduction of around 10 instructions in the performance-critical path.

3.2 Virtual Memory Addressing

In MPI one-sided communication operations, each process can declare a part of its memory as remotely accessible. The collection of such remotely accessible memory across all processes is referred to as a *window*. Once a window has been declared, any process within that window's communicator can access any of the remotely accessible memory. Such communication, however, treats the target process's window as an array, and all communication operations refer to the memory to be accessed in terms of an offset from the base address of that array. Networks, on the other hand, perform communication in terms of virtual and physical addresses rather than as offsets from a base address, thus requiring a translation from the offset to the actual virtual address.

In some situations, a simpler approach is for the application to keep track of the remote virtual address. For example, when the application has allocated a symmetric virtual address across all processes, the local address and the remote address of a given variable are equivalent. Similarly, if an application is communicating with only a small number of processes, it can easily store the base addresses of those processes and perform the appropriate translation, rather than relying on MPI to do so. Without such application-level algorithmic knowledge, however, the MPI implementation must always perform this translation in the performance-critical path, resulting in at least one dereference (to look up the base address in the window) and at least two additional arithmetic instructions to calculate the virtual address based on the base address and displacement unit.

We note that MPI-3.1 also defines dynamic windows where communication calls directly use virtual addresses rather than offsets. This model, however, has some disadvantages. Most important, dynamic windows are harder to optimize for the MPI implementation since they give more flexibility to the user compared with statically created windows. Directly passing virtual addresses by the user requires the MPI implementation to potentially disable some optimizations, thus losing performance. A second disadvantage is that since the same communication calls are used for all types of windows, the MPI implementation still needs to check for what the type the MPI window is. It can assume virtual-address-based communication only if the window is a dynamic window. This check, however, costs nearly the same number of instructions as does the translation from an offset to virtual address, thus washing out any potential benefit.

Proposal for the MPI Standard: We propose to add new functions to the MPI standard, for example, `MPI_PUT_VIRTUAL_ADDR` that would allow the user to directly communicate based on virtual addresses. These routines would be usable on all types of windows, thus removing any disadvantages of dynamic windows. Moreover, since these functions would guarantee that the application was using virtual addresses, no additional check would need to be done by the MPI implementation.

Instruction Savings: This proposal eliminates 3–4 instructions in our implementation, including an expensive memory-access instruction.

3.3 Communication Isolation with Communicators, Windows, and Files

All MPI communication is isolated into communication objects. Any point-to-point or collective operation must happen in the context of a specific communicator and is isolated from communication on other communicators. Similarly, all one-sided communication must happen in the context of a window, and all I/O operations must happen in the context of a file handle. This model enables users to easily reason about their communication pattern without worrying about interference from communicators being used by other libraries. Such isolation also allows users to set properties on specific communicators, windows, or files (such as info hints or window base address for one-sided communication) without affecting other objects. In the rest of this section, we focus on communicators, although the same concepts are valid for windows and file objects as well.

While such separation based on communicators is valuable, since the MPI implementation does not know how many such communicators the user needs, they are created dynamically by using functions such as `MPI_COMM_DUP` or `MPI_COMM_SPLIT`. Consequently, when a communication operation references a communicator, the

corresponding properties of that communicator must be looked up at runtime through at least one expensive pointer dereference and potentially some additional arithmetic instructions.

Proposal for the MPI Standard: Obviously, we cannot remove communicators, windows, and file handles in MPI. These objects give MPI the flexibility and ability to express a variety of complex algorithms. However, we can reduce the instruction complexity of referencing the communication object by making two subtle changes to the MPI standard.

(1) Communicator handles would no longer be dynamically created. Instead, a set of communicator handles would be precreated by the MPI library. The user application would then be responsible for managing them. For example, `mpi.h` would include a compile-time defined number of predefined communicator handles. Let us assume four such predefined handles are provided: `MPI_COMM_1`, `MPI_COMM_2`, `MPI_COMM_3`, and `MPI_COMM_4`.
(2) Communicator properties, such as which processes it covers and info hints, would still be dynamically assigned to each communicator handle. The only difference would be that the communicator handle would now be an input parameter to the communicator creation function, rather than an output parameter. For instance, the new function `MPI_COMM_DUP_PREDEFINED` would take the predefined handle, for example, `MPI_COMM_1`, as an input argument but would internally associate the corresponding set of processes with this handle.

How would the MPI implementation use this model? During compile time, if the MPI library knew how many communicators would be used by the application, it could preallocate the space for it as global variables: `PREDEFINED_COMMS[MAX_COMMS]`. Next, when the communicator `MPI_COMM_1` was actually created, the corresponding preallocated space for that communicator would be populated with the appropriate information, for example, `PREDEFINED_COMMS[MPI_COMM_1].var = value`. When a communication function, say `MPI_ISEND`, was called with the predefined communicator `MPI_COMM_1`, the MPI library would look up the information associated with this communicator as a global variable: `PREDEFINED_COMMS[MPI_COMM_1].var`. We note that even though we technically still would be dereferencing into the `PREDEFINED_COMMS[]` array, the array index would be a compile-time constant, making it easy for the compiler to convert that dereference to a simple global-variable lookup instead of an expensive dereference into the dynamically allocated communicator object.

Instruction Savings: This proposal eliminates 8 instructions in our implementation, including the expensive memory-reference instructions associated with dereferencing to the dynamically allocated communicator object.

3.4 Handling `MPI_PROC_NULL`

The MPI standard defines `MPI_PROC_NULL` as a possible destination for all communication operations: any communication to `MPI_PROC_NULL` is discarded by the MPI implementation. `MPI_PROC_NULL` is not a true communication target but serves as a convenient mechanism, especially for applications that need to deal with boundary conditions, for example, in an application that does neighborhood communication where some subset of the processes does not have neighbors in all directions.

While `MPI_PROC_NULL` is convenient and helps reduce application code complexity, however, it potentially does so at the expense of additional overhead to MPI in the performance-critical path. In particular, every MPI communication call needs to add comparison and branch instructions to check for `MPI_PROC_NULL` and perform the associated processing. More important, these checks cost overhead even for applications that do not use `MPI_PROC_NULL`, since the MPI implementation must assume that the application *might* use this functionality.

Proposal for the MPI Standard: We propose new routines within MPI that would explicitly disallow the usage of `MPI_PROC_NULL`. These routines, for example, `MPI_ISEND_NPN`, could be used only with non-`MPI_PROC_NULL` destinations. For applications that do not use `MPI_PROC_NULL`, the choice is simple: they would simply replace `MPI_ISEND` with `MPI_ISEND_NPN`. Applications that do use `MPI_PROC_NULL` could use `MPI_ISEND`; or they could add a branch themselves to check for `MPI_PROC_NULL` and, when the target was not `MPI_PROC_NULL`, use the routine `MPI_ISEND_NPN`.

Instruction Savings: This proposal can save 3 instructions in our implementation, including a branch instruction that can be expensive on some platforms.

3.5 Per-Operation Completion Semantics

Point-to-point communication in MPI allows the user to have very fine-grained per-operation completion semantics. For applications that issue many communication operations, a request object is returned for each operation and needs to be managed and completed (or freed) explicitly. This model is flexible, although in some applications such fine-grained completion is unnecessary. Often, these applications issue several communication operations and cannot move to the next iteration until they have all completed. Stencil computations are a simple example, although this communication model is also common in several fluid dynamics and quantum chemistry applications.

In order to meet this requirement in the MPI standard, MPI implementations need to maintain request objects for each operation and must assume that the application *might* ask for separate completion of each individual communication operation. This is expensive for the MPI implementation in terms of both memory usage and instruction counts.

Proposal for the MPI Standard: We propose a completion model that is closer to MPI one-sided communication, where operations would be completed in bulk, rather than individually. Specifically, the application would use new routines, such as `MPI_ISEND_NOREQ`, that would perform data transfer just like `MPI_ISEND` but would not return an explicit user request. At a later point, the application could call another new function, `MPI_COMM_WAITALL`, that would complete all requestless operations on that communicator.

How would the MPI implementation use this model? For communication operations that can complete immediately, the working

model would be trivial: no request would be allocated. For operations that are not immediately completed, the MPI implementation could simply maintain a count of the operations issued, rather than request objects for each operation. This would completely remove all the instructions associated with allocating and dereferencing into the request and would add approximately three instructions to increment a counter instead.

Instruction Savings: This proposal saves approximately 10 instructions in our implementation, including dereference instructions into the request structure.

3.6 MPI Matching Bits

MPI point-to-point communication provides the ability for the receiver to match messages to buffers in a different order from what the messages were sent in. This semantics, known as MPI matching semantics, is based on the triplet of communicator, source, and tag. While convenient in some cases, such semantics are unnecessary in many cases where the application can, algorithmically, guarantee the order of communication, thus making these extra matching semantics redundant.

Proposal for the MPI Standard: We propose new routines in the MPI standard, such as `MPI_ISEND_NOMATCH`, that would completely disable match bits for the source and tag but would retain isolation based on the communicator. Thus, different messages coming from different sources or different tags would simply be matched to buffers in arrival order at the destination process. We still would retain the communicator match bits to allow per-communicator isolation. We believe that disabling the communicator isolation would be too disruptive for applications, causing interference with communication being performed by other libraries.

One can envision an alternative proposal using an MPI info hint on the communicator that would guarantee that the application would always use `MPI_ANY_SOURCE` and `MPI_ANY_TAG` for all communication. Such a proposal would be semantically equivalent to our proposal, but it would add an additional dereference into the communicator object to look up the info hint and an additional branch to add a special case for path with no match bits. When combined with the proposal in Section 3.3, however, the additional dereference would no longer exist, but the extra branch would still add two additional instructions.

Instruction Savings: This proposal eliminates 5 instructions in our implementation. When combined with the proposal in Section 3.3, this would allow us to set the communicator match bits as a single load instruction.

3.7 Putting It All Together

Although the proposals presented in this section are somewhat orthogonal concepts, they also are designed to work together. Specifically, we designed new functions, such as `MPI_ISEND_ALL_OPTS`, that encompass all of the proposed optimizations under a common roof. With all the proposed changes, the final instruction count we were able to achieve is 16 instructions for `MPI_ISEND_ALL_OPTS`. That is a 94% reduction in instruction count compared with MPICH/Original. Compared with the optimized MPICH/CH4 implementation that stays within the MPI-3.1 standard (showcased in Section 2), we still get a 73% reduction in instruction count, highlighting the fact that the current MPI standard still leaves considerable performance on the table.

4 EVALUATION

In this section we evaluate the performance of the MPICH/CH4 stack and compare it with MPICH/Original on various benchmarks and applications on multiple platforms. The goal of MPICH/CH4 is to optimize the communication overhead down to the least possible number of instructions. That is, it focuses on optimizing applications that rely on the strong-scaling limit of computation on large supercomputing systems. Accordingly, our evaluation involves on benchmarks and applications in that realm.

4.1 Experimental Testbeds

We used four different platforms for our experimentation evaluation. The first two platforms were IBM Blue Gene/Q supercomputers: Cetus and Mira, both at Argonne National Laboratory. Cetus is a 4,096-node system, and Mira is a 49,152-node system; apart from that, both systems are identical with respect to their software and hardware infrastructure. Each has 16 cores and 16 GB of memory. Our application experiments were performed on Cetus and Mira because of their larger scale. However, the IBM BG/Q environment does not provide some special tools, such as the Intel SDE, that are available only on Intel processors. Therefore, for experiments such as our instruction count analysis that needed these capabilities, we ran small-scale experiments on the "IT" and "Gomez" clusters at the Argonne Joint Laboratory for System Evaluation. IT is equipped with two Intel E5-2699 v4 processors each with 22 cores running at 2.20 GHz and relies on an Intel Omni-Path network fabric. Gomez runs four 16-core Intel E7-8867 v3 processors at 2.5 GHz with a Mellanox EDR-based network fabric. Both machines have dynamic frequency scaling disabled for stable and reproducible results.

4.2 Microbenchmark Evaluations

In this section, we first measure the message-issue rate of the MPICH/CH4 library and compare it with that of the MPICH/Original library, for `MPI_ISEND` and `MPI_PUT`. The benchmark is designed to demonstrate the maximum rate at which a single core can inject data into the network. All performance numbers are shown for a single byte of data transfer.

We performed three experiments. The first experiment was on on the IT cluster for MPICH/CH4/OFI evaluations using the Intel OPA network with PSM2 (Figure 3). The second experiment was on the Gomez cluster for MPICH/CH4/UCX using the Mellanox EDR network (Figure 4). The third experiment (Figure 5) emulated an infinitely fast network: for this purpose, we modified the MPI library to perform all the relevant operations except the actual network communication. Thus, the communication went through the entire MPI stack, but no data was transmitted over the network.

In all three experiments we compared five cases: (1) the baseline MPICH/Original implementation (legend: "mpich/original"); (2) the default MPICH/CH4 implementation (legend: "mpich/ch4 (default)"); (3) the MPICH/CH4 implementation with error checking disabled (legend: "mpich/ch4 (no-err)"); (4) the MPICH/CH4 implementation with no error checking and no checks for thread safety

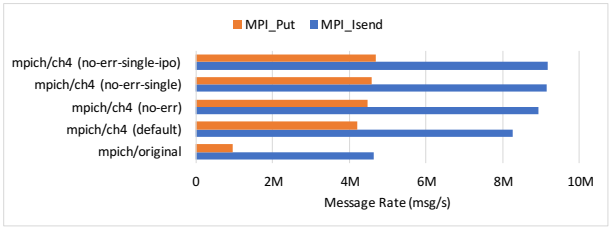

Figure 3: Message rates with OFI/PSM2.

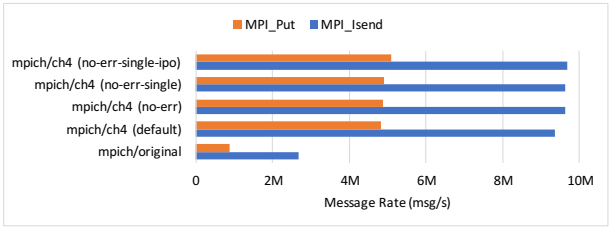

Figure 4: Message rates with UCX.

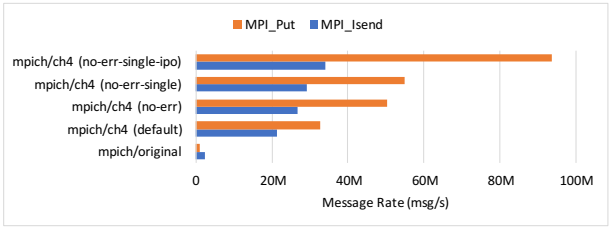

Figure 5: Message rates with infinitely fast network.

(legend: "mpich/ch4 (no-err-single)"); and (5) the MPICH/CH4 implementation with no error checking, with no checks for thread safety, and with link-time inlining enabled (legend: "mpich/ch4 (no-err-single-ipo)"). We note that in all three experiments the performance dramatically improves as we reduce the instruction counts in each step. For the experiments with real networks, we see nearly a 50% increase in the message rate for `MPI_ISEND` and close to a fourfold increase in the message rate for `MPI_PUT`. Despite the large increase in performance, however, we note that the improvement here is still limited by the fact that the networks themselves add a significant number of cycles in transmitting the actual data. On future networks with hardware capable of injecting hundreds of millions of messages per second, we expect the lightweight stack of MPICH/CH4 to make an even larger difference in performance. This situation can be viewed in the third experiment results shown in Figure 5), where the message rate increase is several orders of magnitude.

Based on the best-case performance demonstrated above, we next evaluated the performance improvements achievable in MPICH/CH4 with the improvements to the MPI standard that we proposed in Section 3. The evaluation is done with the same "infinitely fast" network which the low-level network transmission completes instantaneously. The performance numbers are shown in Figure 6. We note that each of the proposed changes provides a large improvement to the MPI performance, peaking at around 132.8 million messages per second for a single communication core.

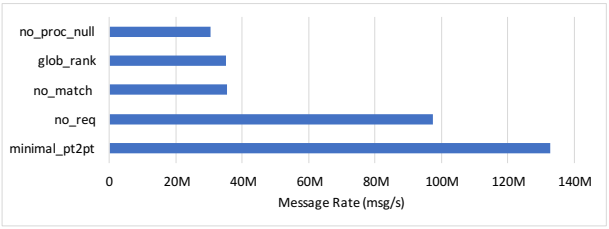

Figure 6: MPI standard improvements for `MPI_ISEND` on infinitely fast network.

4.3 Nek5000

Nek5000 is a high-order spectral element code used for the simulation of turbulence and heat transfer. Nek5000 is a Gordon Bell winner [31] that strong-scales to over a million ranks and has been analyzed extensively in [16, 18, 27]. It is one of the codes that is forming the basis of a new set of libraries being developed by the DOE Center for Efficient Exascale Discretizations (CEED), which is targeting several large scale applications in DOE's Exascale Computing Project (ECP).

Typical Nek5000 simulation campaigns require 10^5–10^7 core hours and $n = 10^7$–10^{10} gridpoints to simulate turbulent flows containing a broad range of space and time scales. The long time-integrations required for direct numerical simulation and large eddy simulation of turbulence put an imperative on speed, that is, on *strong scaling*. Virtually every simulation of this type on high-performance computing platforms is run at the strong-scale limit. A natural question therefore is what factors are setting the strong-scale limit and what can be done to push this limit further, that is, to increase the number of processors P used to solve a problem of fixed size, n.

The *per-timestep* computational complexity for most domain-decomposition-based partial differential equation (PDE) solvers is linear in n. (The number of steps, however, is often dependent on n, e.g., $O(n^{\frac{1}{3}})$, such that the overall complexity is superlinear in n.) If the substeps are explicit, then time advancement simply involves nearest-neighbor data (e.g., halo) exchanges and, to leading order, the solution time per step depends only on the ratio n/P and not on P explicitly [16]. Large values of n/P yield order-unity efficiency because the run time is dominated by computation and the communication overhead is relatively small. Conversely, in the strong-scale limit (n/P small), communication overhead is relatively high and can account for a significant fraction of run time. It is in this important (fast) regime where message sizes are small and the impact of lightweight MPI is important. As a first step towards quantifying this impact, we have established a model problem that solves the linear system $B\underline{u} = \underline{f}$ using conjugate gradient iteration, where B is the mass matrix associated with a spectral element (SE) discretization comprising E elements of order N covering the unit cube, $\Omega := [0,1]^3$, for a problem size of $n \approx EN^3$ grid points. This problem is relevant because it is a central substep in time-dependent solution of many transport problems.

Figure 7 shows performance results on the ALCF BG/Q *Cetus* at Argonne National Laboratory. All tests were performed with 512 nodes in -c32 mode (16384 ranks) in an all-MPI mode. The underlying mesh is a tensor product array of brick elements, each of order N, and the problem is perfectly load balanced, with $E = 2^k$,

for $k = 14, \ldots, 21$. Here, we consider N=3, 5, and 7, such that $n/P \in [27, 43904]$. Earlier analysis [16] suggested that BG/Q should yield order-unity efficiency for this problem with $n/P > 1000$–2000, a transition point well within the range covered here. In Figure 7 (left), we plot the performance as the number of gridpoint-iterations that can be realized per CPU-second (larger is better). For large n/P the problem becomes work dominated; for small n/P it is communication dominated. The solid lines show the performance for MPICH/Original, while the dashed lines show results for MPICH/CH4. Each pair of curves reflects a different value of N. We see that the lower value of N does not perform well, in part because of caching and vectorization strategies in Nek5000, but also because of the $O(M^3N)$ interpolation overhead, which is large when N is small. In all cases, MPICH/CH4 outperforms MPICH/Original except for the largest values of n/P, where the two models are equal. We reiterate that in this application space large values of n/P are relevant only when the application is using all available resources.

The performance ratio is plotted in Figure 7 (center). In the range $n/P \approx 100$–1000, there is a 1.2 to 1.25 performance gain for the three values of N considered. For all the cases, there is a reduction in the ratio moving from $E/P = 2$ to E/P=1 (one element per rank being the finest realizable granularity in Nek5000). In the left panel we see that the downturn in the ratio is due to a slight uptick in the performance for MPICH/Original. Although this is a curiosity, we note that the parallel efficiency is so low at this extreme granularity that the anomaly is not relevant to practical computation.

From a performance standpoint, the most interesting points in Figure 7 are near n/P =1000 for N =5 and 7 (the N =3 case being too slow to be of interest). This is the area where performance-oriented simulations are executed and it is precisely at this point where gains offered by MPICH/CH4 are most important. While a 20% performance gain is desirable, there is a more critical issue at hand: the reduction in communication overhead boosts the parallel efficiency and, under *fixed costs* (e.g., power), can allow significant reductions in runtime. To illustrate the point, consider for example a standard (Amdahl) parallel complexity estimate with runtime on P processors modeled as $T_P = O + W/P$, where O represents overhead and W is the parallel work. In the strong-scale limit where local problem sizes are small, O consists primarily of overhead from short, latency-dominated, messages (e.g., [16]). Energy costs scale as the number of cores times the occupancy time, $E_P = cPT_P$, where c is a scaling constant, which yields $E_P = c(PO + W)$. A reduction in O (say $O' = \frac{1}{2}O$) allows an increase in P for the same simulation cost, $E'_{2P} = c2P[O' + W/(2P)] = c(PO + W)$. At this (fixed) cost, the solution time reduces two-fold $T'_{2P} = O' + W/(2P) = \frac{1}{2}(O + W/P)$. Away from the strong-scale limit ($W/P \gg O$), reductions in overhead have little benefit, but this is also not the regime where most turbulence simulations are executed in HPC environments.

4.4 LAMMPS

LAMMPS (Large-scale Atomic/Molecular Massively Parallel Simulator) is an open-source molecular dynamics code maintained by Sandia National Laboratories. Like many molecular dynamics applications the current algorithm does a 3-D spatial decomposition over the real space coordinates of a given atomic system and assigns the volume with atomic contents of each grid box to an MPI rank. It uses point-to-point communication extensively throughout most simulations for each rank to communicate atomic information for atoms residing in its box to its nearest neighbors on timestep intervals normally on the order of femtoseconds in simulation duration.

Strong scaling for faster timesteps is of utmost importance because several areas of research require simulations to run for relatively long periods of time in order to observe various phenomena at the atomic level, requiring the simulation to run on larger hardware resources, making the 3-D mesh finer and the boxes smaller, containing fewer atoms. The timestep processing time therefore diminishes because less force computation is required for each individual rank, since it holds fewer atoms within its box. As a result, the neighbor exchange communication bottleneck is magnified. Because there are fewer neighbor atoms with which to communicate, the size of the MPI messages also decreases, making the latency of MPI much more apparent. A lower-latency MPI implementation therefore will have a direct effect on strong scaling, as exhibited by the following benchmark.

This simulation was performed on a 3-million=atom face-centered cubic crystal structure for 10,000 timesteps using a simple Leonard-Jones potential. We used an MPI/OpenMP hybrid mode with 1 MPI rank per core (16 per node) and 4 OpenMP threads. We compared the MPICH/Original library with the MPICH/CH4 library, and we showcase the results in Figure 8. As the atoms per core decreases with the larger block sizes, the message sizes also decrease, making the point-to-point communication much more sensitive to MPI latency. As the results indicate, the simulation is sped up overall, with more speedup at higher scale as the scaling limit is approached. We note, however, that the MPICH/Original library completely stops scaling at 8,192 nodes.

5 RELATED WORK

Various improvements have been proposed to address performance and scalability challenges in MPI implementations. Several optimizations are generic improvements to MPI performance, such as improvements to datatype handling [14, 29], performance of collective operations [30], communication progress [28], and interoperability with threads [11, 12, 15]. While these are significant improvements, however, they do not directly address the issue of the communication overheads in MPI at the strong-scaling limit, in other words, reducing MPI message latencies to close to that provided by the network hardware.

Some research has been done to improve message matching [19], which, broadly speaking, can be considered to have the same goal as we have: very lightweight MPI communication. However, the authors do not address the overheads of message matching; instead, they focus on the ability to parallelize such matching across multiple network endpoints for different messages.

Little research directly focuses on the implications of the MPI implementation on lightweight communication. Even less work correlates these overheads from the MPI implementation with the specific requirements of the MPI standard. Recent investigations of MPI operations highlighted their expensive nature on throughput-oriented cores, such as Intel Xeon Phi processors [13]. Some works have investigated lightweight MPI implementations on custom hardware [32] or in non-HPC environments, such as embedded

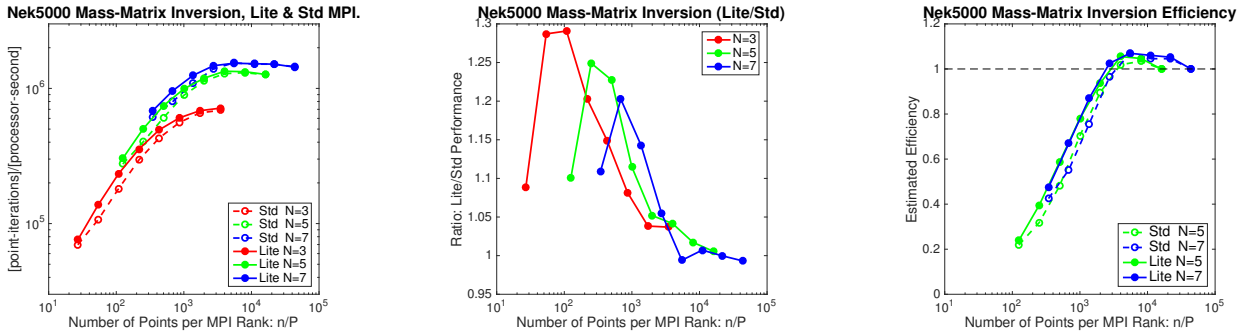

Figure 7: Mass-matrix inversion performance for Nek5000 on Cetus: (left) performance vs. n/P for MPICH/Original (dashed lines) and MPICH/CH4 (solid lines); (center) performance ratio; and (right) parallel-efficiency model for $N = 5$ and 7.

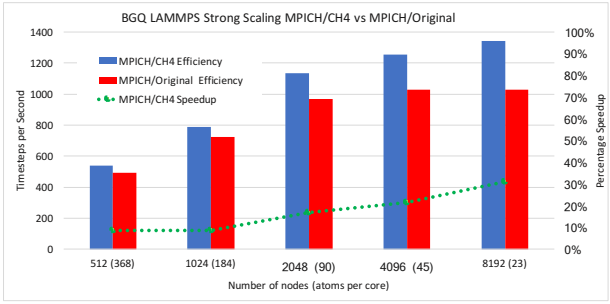

Figure 8: LAMMPS strong scaling.

systems [10], thus requiring adapting MPI to lighter hardware and system software. None of these works, however, presents the fundamental limits of implementing MPI or suggests improvements to the standard in order to run as close to native hardware performance as possible.

6 CONCLUDING REMARKS

As applications look to scale to the largest systems in the world, their ability to perform efficient fine-grained communication determines how close to their strong-scaling limit they can get. This paper focused on understanding the limits in MPI in pushing applications toward this limit. We first presented a highly optimized implementation of MPI that reduces the number of instructions used by the MPI library—all the way from the application to the low-level network communication API—to only a few tens of instructions. The idea of this implementation is to demonstrate the "best case" for the shortest path from the application to the network communication API. We then associated each instruction with the various requirements in the MPI standard, and we proposed changes to the MPI standard to eliminate unnecessary or redundant instructions. All the work was prototyped as a new device in MPICH, called CH4, and evaluated with several microbenchmarks and real applications.

ACKNOWLEDGMENTS

This material was based upon work supported by the U.S. Department of Energy, Office of Science, under Contract DE-AC02-06CH11357.

REFERENCES

[1] 2017. Center for Exascale Simulation of Advanced Reactors. https://cesar.mcs.anl.gov. (2017).
[2] 2017. Center for Exascale Simulation of Combustion in Turbulence. https://science.energy.gov/ascr/research/scidac/co-design/. (2017).
[3] 2017,. CORAL Benchmarks. https://asc.llnl.gov/CORAL-benchmarks. (2017,).
[4] 2017. Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH). https://codesign.llnl.gov/lulesh.php. (2017).
[5] 2017. Monte Carlo Benchmark (MCB). https://codesign.llnl.gov/mcb.php. (2017).
[6] 2017. NAS Parallel Benchmarks. http://www.nas.nasa.gov/publications/npb.html. (2017).
[7] 2017. Nekbone. https://cesar.mcs.anl.gov/content/software/thermal_hydraulics. (2017).
[8] 2017. QMCPack. http://qmcpack.org. (2017).
[9] 2017. The Local-Self-Consistent Mutliple-Scattering (LSMSL) Code. https://www.ccs.ornl.gov/mri/repository/LSMS/index.html. (2017).
[10] Adnan Agbaria, Dong-In Kang, and Karandeep Singh. 2006. LMPI: MPI for heterogeneous embedded distributed systems. In *Parallel and Distributed Systems, 2006. ICPADS 2006. 12th International Conference on*, Vol. 1. IEEE, 8–pp.
[11] Abdelhalim Amer, Pavan Balaji, Wesley Bland, William Gropp, Rob Latham, Huiwei Lu, Lena Oden, Antonio Pena, Ken Raffenetti, Sangmin Seo, et al. 2015. MPICH User's Guide. (2015).
[12] Pavan Balaji, Darius Buntinas, D. Goodell, W. D. Gropp, and Rajeev Thakur. 2010. Fine-Grained Multithreading Support for Hybrid Threaded MPI Programming.

International Journal of High Performance Computing Applications (IJHPCA) 24 (2010), 49–57.
[13] Brian W Barrett, Ron Brightwell, Ryan Grant, Simon D Hammond, and K Scott Hemmert. 2014. An evaluation of MPI message rate on hybrid-core processors. (2014).
[14] Surendra Byna, Xian-He Sun, Rajeev Thakur, and William Gropp. 2006. Automatic memory optimizations for improving MPI derived datatype performance. In *European Parallel Virtual Machine/Message Passing Interface UsersâĂŹ Group Meeting*. Springer, 238–246.
[15] James Dinan, Pavan Balaji, Dave Goodell, Doug Miller, Marc Snir, and Rajeev Thakur. 2013. Enabling MPI Interoperability through Flexible Communication Endpoints. In *Proceedings of the 17th European MPI Users' Group Meeting Conference on Recent Advances in the Message Passing Interface (EuroMPI'13)*. Madrid, Spain, 13–18.
[16] P. Fischer, K. Heisey, and M. Min. 2015. Scaling Limits for PDE-Based Simulation (Invited). In *22nd AIAA Computational Fluid Dynamics Conference, AIAA Aviation*. AIAA 2015-3049.
[17] P. Fischer, J. Lottes, and S. Kerkemeier. 2008. Nek5000: Open source spectral element CFD solver. http://nek5000.mcs.anl.gov and https://github.com/Nek5000/nek5000. (2008).
[18] P. F. Fischer and A. T. Patera. 1991. Parallel Spectral Element Solution of the Stokes Problem. *J. Comput. Phys.* 92 (1991), 380–421.
[19] Mario Flajslik, James Dinan, and Keith D Underwood. 2016. Mitigating MPI message matching misery. In *International Conference on High Performance Computing*. Springer, 281–299.
[20] William Gropp, Torsten Hoefler, Rajeev Thakur, and Ewing Lusk. 2004. *Using Advanced MPI: Modern Features of the Message-Passing Interface*. MIT Press.
[21] William Gropp, Ewing Lusk, and Rajeev Thakur. 1999. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press.
[22] Yanfei Guo, Charles Archer, Michael Blocksome, Scott Parker, Wesley Bland, Kenneth J. Raffenetti, and Pavan Balaji. 2017. Memory Compression Techniques for Network Address Management in MPI. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Orlando, Florida.
[23] Salman Habib, Vitali Morozov, Nicholas Frontiere, Hal Finkel, Adrian Pope, and Katrin Heitmann. 2013. HACC: Extreme Scaling and Performance Across Diverse Architectures. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*. ACM, New York, NY, USA, Article 6, 10 pages. https://doi.org/10.1145/2503210.2504566
[24] Michael A Heroux, Douglas W Doerfler, Paul S Crozier, James M Willenbring, H Carter Edwards, Alan Williams, Mahesh Rajan, Eric R Keiter, Heidi K Thornquist, and Robert W Numrich. 2009. *Improving Performance via Mini-applications*. Technical Report SAND2009-5574. Sandia National Laboratories.
[25] Torsten Hoefler, James Dinan, Rajeev Thakur, Brian Barrett, Pavan Balaji, William Gropp, and Keith Underwood. 2015. Remote Memory Access Programming in MPI-3. *TOPC'15* (2015).
[26] MPI Forum. 2015. MPI: A Message Passing Interface Standard. (2015). http://www.mpi-forum.org/docs/docs.html.
[27] M. Otten, J. Gong, A. Mametjanov, A. Vose, J. Levesque, P. Fischer, and M. Min. 2016. An MPI/OpenACC Implementation of a High Order Electromagnetics Solver with GPUDirect Communication. *Int. J. High Perf. Comput. Appl.* (2016).
[28] Mohammad J Rashti and Ahmad Afsahi. 2008. Improving communication progress and overlap in mpi rendezvous protocol over rdma-enabled interconnects. In *High Performance Computing Systems and Applications, 2008. HPCS 2008. 22nd International Symposium on*. IEEE, 95–101.
[29] Xian-He Sun et al. 2003. Improving the performance of MPI derived datatypes by optimizing memory-access cost. In *Cluster Computing, 2003. Proceedings. 2003 IEEE International Conference on*. IEEE, 412–419.
[30] Rajeev Thakur and William D Gropp. 2003. Improving the performance of collective operations in MPICH. In *European Parallel Virtual Machine/Message Passing Interface UsersâĂŹ Group Meeting*. Springer, 257–267.
[31] H. M. Tufo and P. F. Fischer. 1999. Terascale Spectral Element Algorithms and Implementations. In *Proc. of the ACM/IEEE SC99 Conf. on High Performance Networking and Computing, Gordon Bell Prize*. IEEE Computer Soc., CDROM.
[32] Isaías A Comprés Ureña, Michael Riepen, and Michael Konow. 2011. RCKMPI–lightweight MPI implementation for IntelâĂŹs Single-chip Cloud Computer (SCC). In *European MPI Users' Group Meeting*. Springer, 208–217.
[33] M. Valiev, E. J. Bylaska, N. Govind, K. Kowalski, T. P. Straatsma, H. J. J. Van Dam, D. Wang, J. Nieplocha, E. Apra, T. L. Windus, and W. A. de Jong. 2010. NWChem: A Comprehensive and Scalable Open-Source Solution for Large Scale Molecular Simulations. *Computer Physics Communications* 181, 9 (2010), 1477–1489.

A ARTIFACT DESCRIPTION

This appendix presents the software environment used in our experiments.

A.1 Obtaining Our Software

A.1.1 MPICH. Our MPI implementation is based on the open source MPICH software available on GitHub (https://github.com/pmodels/mpich).

A.1.2 Benchmarks. Source files for the benchmarks used in this paper can be found in the supplemental materials.

A.1.3 Analysis Traces. SDE traces used in this paper can be found in the supplemental materials. These serve for inspection as well as for comparison with traces generated by the reader.

A.2 Target Platforms

Because our derived MPI implementation is based on MPICH. It is portable to various hardware and environments. It was tested on Linux and OS/X operating systems, on various fabrics (Intel Omni-Path, Mellanox FDR/EDR, Cray Aries, and Blue Gene/Q), and on x86 and Power architectures. Our microbenchmarks are also portable across similar environments. The Intel SDE tool does not depend on any compiler but requires running on an x86 architecture.

A.3 Building and Running

Building our MPICH derivatives follows the same model as the original MPICH, that is, the common `configure`, `make`, `make install` method. Scripts to generate the various MPICH builds and convenience scripts to construct the benchmarks and performance and analysis traces are found in the supplemental materials.