Debugging MPI Implementations via Reduction-to-Primitives

Gene Cooperman
Khoury College of Computer Sciences
Northeastern University
Boston, USA
gene@ccs.neu.edu

Dahong Li
MemVerge, Inc.
Milpitas, CA, USA
dahong.li@memverge.com

Zhengji Zhao
NERSC
Lawrence Berkeley National Laboratory
Berkeley, USA
zzhao@lbl.gov

***Abstract*—Testing correctness of either a new MPI implementation or a transparent checkpointing package for MPI is inherently difficult. A bug is often observed when running a correctly written MPI application, and it produces an error. Tracing the bug to a particular subsystem of the MPI package is difficult due to issues of complex parallelism, race conditions, etc. This work provides tools to decide if the bug is: in the subsystem implementing of collective communication; or in the subsystem implementing point-to-point communication; or in some other subsystem. The tools were produced in the context of testing a new system, MANA. MANA is not a standalone MPI implementation, but rather a package for transparent checkpointing of MPI applications. In addition, a short survey of other debugging tools for MPI is presented. The strategy of transforming the execution for purposes of diagnosing a bug appears to be distinct from most existing debugging approaches.**

***Index Terms*—MPI, supercomputing, debugging, MANA, transparent checkpointing**

I. Introduction

This work provides two debugging tools that have been extensively used in debugging MANA-2.0 [1]. MANA-2.0 (more generally referred to as MANA here) is a production-ready package for transparent checkpointing of MPI applications. The tools are equally useful in debugging both MANA and arbitrary MPI implementations. For this reason, the rest of the paper will frequently refer to "a new MPI implementation".

The original and primary context for this work is MANA: a checkpointing package that operates by *transforming the execution of an MPI application* through wrappers around MPI functions. The combination of MANA on top of a standard MPI implementation, acts as a "new MPI implementation".

MANA's Requirements to Transform MPI calls:
Since the tools were motivated by debugging MANA, we pause to describe the architecture of MANA. The architecture of MANA is based on the idea of *split processes*. In split processes, two programs are loaded into the address space of a single process. One program (upper half) is the MPI application (linked to a MANA library), and the second program (lower half) is a small MPI application that includes an actual MPI library. The MANA library includes stub functions for each MPI function, and each stub function calls one or more actual MPI functions in the small MPI application. On checkpoint, only the memory of the MPI application is saved. On restart, a new MPI application is run, and that application restores the MPI application to its original location in memory. See Section II and especially Figure 1, for further details of the MANA architecture.

The split process technique was developed in order to save the memory of an MPI application, while at the same time avoiding the need to save the memory of the MPI library. This is important since the MPI library intensively uses hardware associated with the network and possibly a high-performance network switch. Saving the state of the a network switch, and possible some kernel modules, would be a difficult proposition.

The architecture of MANA requires that at the time of checkpoint, no process must be in the middle of a call to MPI. (The memory of the small MPI application (lower half) is not saved during a checkpoint. It will be replaced by a fresh process on restart.) This forces the MPI wrappers of the MANA library to replace the original MPI call by one or more alternate MPI calls to the actual MPI library. For example, the MANA library wrapper function for MPI_Wait calls the MPI_Test of the actual MPI library in a small loop. A pending checkpoint is permitted to proceed only after MPI_Test returns, so that no call frame on the stack refers to the lower half program.

Hence, one of the most difficult intellectual challenges of MANA is to re-organize the large family of MPI calls around a restricted set of calls to be passed to the actual MPI library: while maintaining efficiency; and while guaranteeing that no MPI process is blocked in a call to the lower half at the time of checkpoint. *Thus, MANA is implicitly translating the MPI calls of the original application to an alternate sequence of MPI calls to the actual MPI library.*

Motivation for Tools for Reduction-to-Primitives:
The difficulty of developing a robust checkpointing algorithm in MANA can be understood by analogy to developing a new optimizing compiler. Both endeavors are concerned with translating high-level code representations into lower-level executions. Both endeavors require subtle algorithms to preserve the high-level semantics when translated to a lower level. checkpoint-restart boundary. As with optimizing compilers, this implies myriad corner cases, whose bugs can only be

uncovered by testing on diverse real-world software.

There are two tools presented here:

Collective-to-P2P: This tool translates each MPI collective communication call to a sequence of point-to-point calls. This can be done for all collective calls or a subset of the collective calls.

Deterministic-P2P: This tool has two parts. First, in each process, it keeps a log of the metadata of all messages received . Second, when applied in MANA, the log is "turned on" after resuming from a checkpoint. Then, after restart, any ambiguities due to `MPI_ANY_SOURCE` and `MPI_ANY_TAG` are resolved by choosing the same tag and source as displayed in the log.

The second tool, Deterministic-P2P, is closely related to *communication determinism* [2]. Communication determinism is well-defined only when all collective calls are reduced to point-to-point calls. In this case, the execution is communication-deterministic if the result is equivalent to a deterministic ordering of MPI messages being received. In the case of non-blocking receives, the message is received by a call to MPI_Wait or MPI_Test.

Note that in defining communication determinism, Cappello et al. state that "We assume in this paper that collective communications are implemented atop point-to-point communications and the implementation is deterministic." [2, Section 2]. The current work provides an easy way to enforce that a target application implements collective communication atop point-to-point communication.

Deterministic-P2P is especially useful in the checkpointing context. It compares execution during resume (of the same process after a checkpoint) and restart (a new process from a checkpoint image), to see if the checkpointing package preserves communication parallelism.

With respect to Deterministic-P2P, note that this can be used in debugging two distinct cases:

a) It can be used to test if a target MPI application running under a new MPI or checkpointing implementation produces the same result *aside from floating point error*.
b) It can be used to test if a target MPI application running under a new MPI or checkpointing implementation produces exactly the same result *(including the low bits of floating point values)*.

The distinction arises because floating point computer arithmetic is not commutative. As an example, if an MPI process receives floating point values from four neighbors using `MPI_ANY_SOURCE` and adds those values, the resulting sum will be different, depending on the order in which the four values were received.

Novelty:

The novelty of this work is as follows.

1) The first debugging tool, Collective-to-P2P, allows a target MPI application to be executed in a special mode, in which all or a selected subset of MPI collective communication calls are replaced by point-to-point calls. If a bug disappears when the target MPI application runs with this tool, then this shows that the bug is associated with one or more of the selected MPI collective communication calls. This allows one to trace the bug to one of: a transparent checkpointing package; an MPI implementation; or the target MPI application itself.
2) The second debugging tool, Deterministic-P2P, allows a target MPI application to be executed in a special mode, in which a form of communication determinism is enforced. It can force a second execution of a target application to bind `MPI_ANY_SOURCE` and `MPI_ANY_TAG` to the same source and tag as the first execution. This enforces communication determinism between the two executions (the second execution produces the same result as the first execution). This is useful to determine if a new MPI implementation or checkpointing implementation preserves the deterministic results seen in an earlier execution.
3) A target MPI application can be run using the two tools in combination. In this mode, the Collective-to-P2P tool is used as a prerequisite to the Deterministic-P2P tool, in order to isolate questions of communication determinism.

Organization of This Work:

This work is organized into the following sections. Section II briefly describes the underlying split-process design of the original MANA, and then provides further details of how wrapper functions are used to translate MPI calls. Section III describes in detail the two debugging tools and how they are used. Section IV presents an experimental evaluation of the two tools for debugging. Section V presents related work. Finally, Section VI is the conclusion.

II. Background

MANA (MPI-Agnostic Network-Agnostic transparent checkpointing tool) is a previously developed package for checkpointing MPI applications [3]. A newer version, MANA-2.0 [1], has been developed for production use in supercomputing. MANA uses the idea of split processes.

A. Split-processes

In brief, the key idea of a *split-process* approach is to load two independent programs into the virtual memory of a single process. Because they are contained within the same virtual memory, a function from one program (typically the "upper-half" program) may call a function of the other program (typically the "lower-half" program) — so long as the address of the lower-half function is known to the upper-half function.

The upper-half program is the MPI application program, dynamically linked with a "stub" MPI library. The "stub" MPI library consists of wrapper functions around each MPI call, as described in the introduction. The wrapper may call one or more lower-half MPI functions in the actual MPI library. Finally, the lower-half program consists of a small MPI application linked to the actual MPI library, which also links

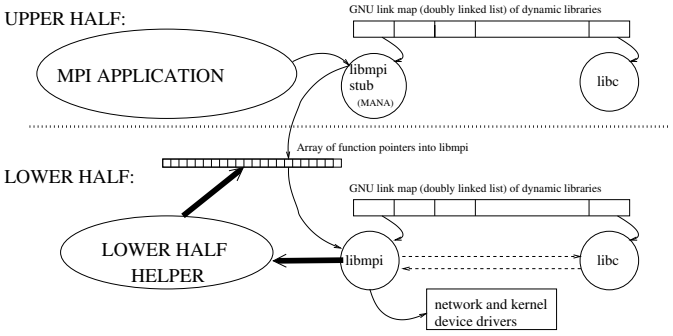

Fig. 1. MANA implementation: split processes

to the necessary libraries. See Figure 1 for a visual description. For simplicity of exposition, the figure shows the special case where each MPI wrapper function calls only a single MPI function in the lower half.

For a deeper description of split processes, see the original paper of Garg et al. [3].

B. Communication determinism and send determinism

We assume in this work that communication consists solely of MPI collective communication and point-to-point communication. One of the novelties of this work is the ability to transform the execution of an MPI application from one that uses both collective and point-to-point communication to one that uses *only* point-to-point communication. This allows us to consider communication determinism within a modern MPI implementation.

Note that Cappello et al. [2] had argued for studying communication determinism directly within a point-to-point context. They argued for this, because the MPI implementations that they were familiar with in 2010 had implemented collective communication atop point-to-point communication, as seen in this quote.

> We assume ... that collective communications are implemented atop point-to-point communications and the implementation is deterministic. The last condition is satisfied by the MPI implementations we are familiar with. Therefore, w.l.o.g., we can restrict our attention to point-to-point communication. [2, Section II]

However, there is no guarantee that collective communications are implemented atop point-to-point communications in a modern MPI implementation. For example, the MPI library may use a network switch to accelerate performance in hardware. One can easily imagine such a case for MPI_Reduce, or for MPI_Alltoall.

It is for this reason that the current work requires a first tool, Collective-to-P2P, which translates collective operations into multiple point-to-point operations through a very simple algorithm (e.g., no broadcast trees, no scan algorithms for MPI_Reduce). The simple Collective-to-P2P tool of this work has better guarantees of correctness, because each collective operation is implemented by a routine of at most 34 lines, which can be informally verified "by eye".

Related to communication determinism is the concept of *send determinism* [4]. In send-determinism, if the same code is run twice with the same input, then each MPI process is guaranteed to execute the same sequence of MPI_Send calls (with the same parameters) over the two runs.

Cappello et al. studied how common it is that MPI applications are already structured to support determinism [5]. They analyzed 27 application and benchmark codes used at NERSC. The study showed that 11 were communication-deterministic, one was completely deterministic, and 13 were send-deterministic.

The intuition for communication determinism versus send determinism can be understood by reviewing a relaxation equation in partial differential equations modeling an elastic membrane. The membrane is spread above an x-y plane, and the height of the membrane at an (x, y) coordinate is the z coordinate.

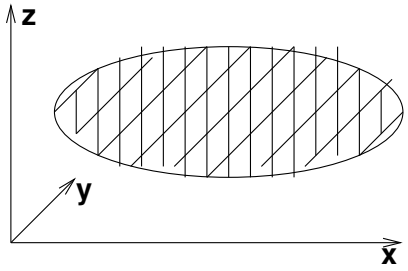

Fig. 2. A physical membrane is simulated by an x-y grid. At each intersection point, the z coordinate represents the height of the membrane. In a relaxation equation, at each iteration, the z coordinate is changed to be the average of the four nearest neighbors.

Figure 2 shows such a membrane. If each grid point is stored in a separate MPI process, then at each iteration, an MPI process must receive the z value from its four nearest neighbors, and set the new local z value to the average of the four neighbors. This can be done by:

```
for (int i = 0; I < 3; i++) {
  MPI_Recv(..., rank[i], ...);
}
```

or by:

```
for (int i = 0; I < 3; i++) {
    MPI_Recv(..., MPI_ANY_SOURCE, ...);
}
```

The former code guarantees communication determinism, since the z values from the neighboring ranks (given by `rank[i]`) are received in a prescribed manner. The latter code guarantees only send determinism. If the four z values are averaged by adding them in the order in which they are received, then the latter code may produce slightly different averages of the z values, since the ordering will be different, and computer addition is not commutative.

In Section III-B, the tool Deterministic-P2P will be described. If the MPI target application (or the result of transforming the MPI code) causes two runs to vary in the small bits of the floating point numbers, then one suspects that this is due to send determinism (the later MPI code of the example).

This can be tested by applying the Deterministic-P2P tool. If this causes the variation in the low bits of floating point numbers to disappear, then this shows that the variation is due to an instance of send determinism. The tool enforces communication determinism in the resulting run. But if the low bits of floating point numbers continue to vary, then this shows that the variation is due to some race condition (a bug) in the MPI implementation itself.

III. Two Debugging Tools

This section is divided into three parts. A description is provided for the Collective-to-P2P tool, for the Deterministic-P2P tool, and finally a motivation for the combined use of Collective-to-P2P and Deterministic-P2P.

A. Collective-to-P2P: Reduction of Collective Communication to Point-to-Point

When seeing a bug in an MPI execution, the first question that a developer usually asks him or herself is whether the bug is related to the use of MPI collective communication calls. This could be related to bugs in the MPI target application, the MPI implementation itself, or a transparent checkpointing package, such as MANA. The solution presented here is to insert an additional library in the library search path that redefines each of the collective calls to themselves make calls to point-to-point calls.

In the case of MANA, the file
https://github.com/mpickpt/mana/blob/main/mpi-proxy-split/mpi-wrappers/mpi_collective_p2p.c
redefines the functions MPI_Barrier, MPI_Bcast, MPI_Gather, MPI_Gatherv, MPI_Scatter, MPI_Scatterv, MPI_Allgather, MPI_Alltoall, MPI_Alltoallv, MPI_Alltoallw, MPI_Reduce, MPI_Allreduce, MPI_Reduce_scatter, MPI_Ibcast, MPI_Igather, MPI_Iallgather, MPI_Ialltoall, MPI_Ialltoallw, MPI_Ireduce, MPI_Iallreduce, MPI_Ireduce_scatter, and MPI_Iexscan. This subset of all collective functions was chosen according to which calls occur commonly in MPI applications.

The non-blocking (asynchronous) collective calls are implemented by calling the non-blocking function

```
MPI_Ibarrier(MPI_COMM_SELF, request)
```

after which the blocking version of the MPI collective call is executed. The call to MPI_Ibarrier forces the non-blocking collective call to block until all MPI processes have "arrived". At that point, the corresponding request is rapidly completed by MPI_Ibarrier, and the remaining blocking version of the collective call is then valid. See the next figure as an example.

```
int MPI_Ireduce(const void* sendbuf, void* recvbuf,
    int count, MPI_Datatype datatype, MPI_Op op,
    int root, MPI_Comm comm, MPI_Request *request) {
        MPI_Ibarrier(MPI_COMM_SELF, request);
        return MPI_Reduce(sendbuf, recvbuf, count,
                datatype, op, root, comm);
}
```

In addition to obvious implementations using MPI_Send and MPI_Recv, additional code is added to handle the parameter MPI_IN_PLACE.

Finally, since each MPI function is defined individually, and since these definitions shadow the definition of the MPI library, it is easy to selectively re-define some collective communication functions, but not others. In this way, one can pin down a particular MPI collective communication function that is causing a bug.

B. Deterministic-P2P: Reduction of Parallel Point-to-point Communication to Deterministic Point-to-point

The Collective-to-P2P tool can be used either to convert all MPI collective calls to point-to-point, or else to selectively choose just some MPI collective calls and convert them to point-to-point. If one of these transformations causes a bug to disappear, then this shows that there is a bug in one of the implementation of the MPI collective call that was transformed.

Thus, the Collective-to-P2P allows us to rule out a bug in the implementation of the MPI collective calls. It remains to trace a bug found in the output of an MPI execution back to the MPI implementation. This section provides a tool, Deterministic-P2P, to help find a possible bug in the implementation of the point-to-point layer.

In employing this tool, we usually use it in combination with the Collective-to-P2P tool. This simplifies the debugging task. It effectively transforms the target MPI application into an application that employs *only* the point-to-point calls for communication, and does not use any MPI collective calls. (We also assume that, like most MPI applications today, the MPI one-sided operations are not being used.)

The Deterministic-P2P tool is intended for a particular class of bugs. It is assumed that either the entire application is supposed to be *communication-deterministic*, or else some portion of the execution. For simplicity, we assume that the entire MPI application is intended to be communication-deterministic. In particular, VASP [6], [7] is an example of a large, sophisticated application that obeys communication-determinism.

See Section II-B for background on communication determinism. In particular, note the difference between send determinism and communication determinism, as described in that section. By employing the Deterministic-P2P, one can distinguish between: a natural send determinism in the MPI code (or in the internals of the MPI implementation); and a bug in the MPI implementation itself, due to a race condition.

The methodology in implementing Deterministic-P2P is to use a log-and-replay strategy. During logging, an MPI receive

call binds the MPI wild cards to the actual tag and rank from which the receive occurred during logging. Later, during replay, the MPI receive call is re-executed, but this time with the original MPI tag and rank, rather than the MPI wild card. This guarantees determinism in comparing the original execution, which logs the actual tag and rank in the execution, and the second execution (using replay).

In detail, the workflow proceeds as follows:

```
MANA_P2P_LOG=1 mana_launch -i SECONDS ...
                        mpi_executable
mpirun mana_p2p_update_logs
MANA_P2P_REPLAY=1 mana_restart ...
                    --restartdir ./DIR
```

There is one log created per MPI process. That log is modified by `mana_p2p_update_logs`, so as to prepare it in the correct format for `mpi_restart`.

As described in the introduction, the investigation was based on MANA, with checkpoint/restart. However, there is nothing in this architecture that would prevent this software from being adapted to use with a standard MPI library (but without MANA). In that case, we would replace each of `mana_launch` and `mana_restart` by a simple call to `mpirun`. This would allow us to execute a second run in a deterministic mode in which MPI wildcards (MPI_ANY_TAG and MPI_ANY_SOURCE) were replaced by the actual tag and source, as determined in the log.

An additional feature of the log is that during logging, it records an exclusive-or of the bytes of MPI message that is received. Then, during replay, it verifies that the message that is received has the same exclusive-or as the value recorded in the log. This provides further guarantees that the MPI (or MANA) implementation is not mistakenly accepting a message on replay that has the correct metadata, but the wrong message body.

C. Combining Collective-to-P2p with Deterministic-P2P: Motivation

The primary goal of combining the Deterministic-P2P tool with Collective-to-P2P is to analyze a more unusual type of bug. Suppose target MPI application is written so as to be deterministic. If it is run twice, then it will report the same output in both runs. This is a feature of some scientific applications. Suppose a new version of the code is a performance optimization over the old version, but it is intended to produce the same output. This constraint can be verified most easily, if it is guaranteed that two runs of the same version of the code will produce *exactly* the same output.

Next, suppose that this is satisfied on a reference MPI implementation, but not on a new MPI implementation. One then wishes to determine the cause of the lack of determinism. A first step is to enforce *communication determinism*, as described earlier. Hence, in the first run, logging is employed, and in the second run, replay from the log is employed; as described in the previous subsection. If determinism can be enforced by this tool, and yet there is some variation in the output from one run to the next, then this proves that there is a flaw in the MPI implementation that is inserting non-determinism.

Many MPI implementations essentially implement collective MPI calls in a routine consisting solely of MPI point-to-point calls. However, the logic of that internal transformation may be complex, due to the use of broadcast trees, scan operations (e.g, for MPI_Reduce), and other complex data structures. Such implementations may be error-prone. Even worse, such implementations are likely to use MPI *wild cards* (e.g., MPI_ANY_TAG, MPI_ANY_SOURCE) for greater parallel efficiency.

Unfortunately, such use of wild cards in internal implementations of MPI collective calls would force us to abandon any absolute claims to determinism. The internal use of MPI_ANY_SOURCE, for example, implies that there is a race that will invalidate any claim to communication determinism, and may possibly even invalidate any claim to send determinism. Such an MPI implementation "poisons" the environment in using the second tool, Deterministic-P2P. The goal of Deterministic-P2P is to determine whether the MPI target application maintains its guarantee of deterministic execution. But if a flawed MPI implementation adds some non-determinism through its internal use of MPI wild cards in its implementation of collective calls, then it is impossible to distinguish whether a flawed non-deterministic MPI implementation is adding non-determinism in the collective layer or in the point-to-point layer.

```
int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype,
              int root, MPI_Comm comm) {
  PROLOG_Comm_rank_size;
  if (rank == root) {
    int i;
    for (i = 0; i < size; i++) {
      if (i != root) {
        MPI_Send(buffer, count, datatype, i, 0, comm);
      }
    }
  } else {
    MPI_Recv(buffer, count, datatype, root, 0, comm,
                                    MPI_STATUS_IGNORE);
  }
  return MPI_SUCCESS;
}
```

Fig. 3.

The benefit of the current approach, using Collective-to-P2P, is that the wrapper function employs a communication-deterministic implementation of a collective call that also fits on a single screen. Rather than trust a possibly flawed MPI implementation, one can instead trust a small wrapper function that directly translates a collective call to a point-to-point call in a deterministic manner. The correctness of this wrapper function can be informally debugged "by eye", by inspecting that single screen of code for the small wrapper. This produces

confidence that the Collective-to-P2P tool is not the cause of the non-determinism. Hence, the search for an incorrect insertion of non-determinism can be restricted to the point-to-point layer of the MPI implementation.

The lines of code needed to implement an *easy-to-check* collective call using point-to-point calls varies from 15 lines for MPI_Bcast[1] (see Figure 3) to 34 lines of code in the worst case (MPI_Alltoall).

IV. Experimental Evaluation

All experiments were run on Cori, a Cray XC40 system at NERSC. Cori contains two types of compute nodes, dual-socket Intel Haswell and single-socket KNL nodes, interconnected with Cray Aries network. Each Haswell node has 32 cores (64 hardware threads) running at 2.3 GHz, and 128 GB DDR4 2133 MHz memory; each KNL node has 68 cores (272 hardware threads) running at 1.4 GHz, and 96 GB DDR4 2400 MHz memory. Cori runs Cray Linux environment version 7.0.UP03 with Linux kernel version 5.3. All experiments were run on Cori Haswell nodes, and used NERSC's community file system based on IBM's Spectrum Scale for I/O.

The version of MANA that was used is commit 507187 (Sept. 6, 2022), found as open source on the main branch, at https://github.com/mpickpt/mana. The version of MANA used at the time of writing was based on a "hybrid two-phase commit" algorithm for collective communication. The documentation of how to use the two tools is found at https://github.com/mpickpt/mana/blob/main/manpages/mana.1.md.

Experiments were run on two MPI micro-benchmarks (Figure 4), and also on two real-world MPI programs (Figure 5). The two real-world MPI programs are VASP [7] (VASP version 5) and CP2K [8] (version 9.1).

The version of MANA employed at the time of writing incurred a typical runtime overhead of 20% for the real-world applications and more than 50% for the micro-benchmarks, as compared to running the MPI benchmarks natively. This overhead is expected to proportionately affect all times reported in experiments. A more recent version of MANA employs a newer "sequence number" algorithm for collective communication [9], with much lower runtime overheads. That version was not available until shortly before the camera-ready deadline of the current work.

Figure 4 shows the use of two MPI micro-benchmarks: Reduce and Alltoall. The program Reduce iterates in a loop over MPI_Reduce. The program Alltoall iterates in a loop over MPI_Alltoall followed by a single call to MPI_Allreduce. The two programs are run under: original MANA; MANA with the Collective-to-P2P tool; and MANA with the combination of Collective-to-P2P and Deterministic-P2P.

In all cases, the applications in Figure 4 are run with 128 MPI processes. The ratio of the time with the tool to the base time (no tool) varies between: a moderate value of 1.47 or 2.16 (Collective-to-P2P tool alone); and a ratio of approximately 13.8 (using both tools together). The slowdown of 13.8 is considered acceptable, since the computation is used only for debugging, and would normally be executed only once. (See Section III.)

[1]The observant reader will note the "`PROLOG_Comm_rank_size`" macro, collapsing a standard MPI idiom to one line, in order to stay at 15 lines.

Application (input)	Ranks	Run	Time (s)	Run/Base (ratio)
Reduce (100)	128 (4 nodes)	MANA	94	base
Reduce (100)	128 (4 nodes)	Collective to P2P	138	1.47
Reduce (100)	128 (4 nodes)	Collect.+ Determ.	1294	13.77
Alltoall (1000)	128 (4 nodes)	MANA	43	base
Alltoall (1000)	128 (4 nodes)	Collective to P2P	93	2.16
Alltoall (1000)	128 (4 nodes)	Collect.+ Determ.	593	13.79

Fig. 4. The "Time" column represents: the time to run under MANA (typically within 5% of the time to run the target application without MANA); the time using the Deterministic-P2P tool alone, the time with the Collective-to-P2P tool, and the time for using the combined tool. These small test programs show that even in the worst case (combination of Collective-to-P2P and Deterministic-P2P tools), the slowdown is about a factor of 14. The ratio for Collective-to-P2P is larger for the Alltoall test, since the implementation of MPI_Alltoall uses n^2 messages, as opposed to n messages for MPI_Reduce. However, the larger performance penalty is incurred when using Deterministic-P2P, and this penalty dominates for the combined tools. In this case, the two tests have similar penalties.

Finally, Figure 5 shows the use of MANA with two real-world applications: VASP [6], [7] and CP2K [8]. In that figure, the ratio for Deterministic-P2P is 3.1, which is larger than what would be seen in the micro-benchmarks. This is because the micro-benchmarks emphasize collective calls rather than point-to-point calls. VASP uses point-to-point calls more extensively. This creates the higher overhead of saving metadata (source/destination/tag) for each MPI message. Saving the metadata is required for the log-and-replay strategy outlined in Section III-B.

In that figure, Collective-to-P2P tool is particularly expensive for 128 MPI processes (ranks) for VASP5. We hypothesize that this is due to the broad usage of MPI_Alltoall within VASP5. Hence the ratio of 4.6 for VASP5 using Collective-to-P2P is similar to the ratio of 2.16 for Alltoall using Collective-to-P2P in Figure 4.

The experiment with CP2K in Figure 5 presents a different usage to the two tools. In this case, we take advantage of the ability of Collective-to-P2P to selectively expand only the calls to MPI_Bcast into point-to-point calls. Other MPI collective calls are not expanded. In this more selective usage of the Collective-to-P2P tool, the ratio of the time for the combined tools versus the base case is a moderate 4.1. This is much smaller than the ratio for VASP5, or the ratio for the two micro-benchmarks seen in Figure 4.

Application (input)	Ranks	Run	Time (s)	Run/Base (ratio)
VASP5 (PdO4)	128 (4 nodes)	MANA	2191	base
VASP5 (PdO4)	128 (4 nodes)	Determ. P2P	6771	3.1
VASP5 (PdO4)	128 (4 nodes)	Collective to P2P	9980	4.6
VASP5 (PdO4)	128 (4 nodes)	Collect.+ Determ.	$>$ 14,400	$>$ 6.6
CP2K (hco3)	64 (4 nodes)	MANA	181	base
CP2K (hco3)	64 (4 nodes)	Bcast to P2P + Determ.	740	4.1

Fig. 5. The Time column represents: the time to run under MANA (typically within 5% of the time to run the target application without MANA); the time using the Deterministic-P2P tool alone, the time with the Collective-to-P2P tool, and the time for using the combined tool. In the case of VASP5 (with an input for PdO4), the time for the combined tool was more than the 4 hours (14,400 seconds) allowed for a job in the interactive queue, and the ratio was more than 6.6. For CP2K, the ratio for the combined time is only 4.1. The smaller ratio is accounted for by only reducing MPI_Bcast to point-to-point operations in this case, while retaining MANA's original collective communication calls for the remaining collective calls.

V. Related Work

The two debugging tools described here appear to be unique in radically transforming the execution of an MPI application. By doing so, they provide information that aids in the diagnosis of bugs in an implementation for transparent checkpointing, such as MANA, or potentially in a new MPI implementation itself. Nevertheless, it is useful to review some other MPI debugging tools that are widely used.

A short survey of MPI debugging tools: The debugging tool most closely related to the ideas in this paper (transforming an MPI application and comparing the difference) is GDB4HPC. The GDB4HPC package is a GDB-based command-line-based parallel debugger, developed by Cray. GDB4HPC is unique in its comparative debugging feature, which enables programmers to run two versions of an application side by side and compare data structures between them to identify the location where the two codes start to deviate from each other. CCDB (Comparative Debugging, from Cray) is a GUI tool for GDB4HPC's comparative debugging.

Another tool, related to Deterministic-P2P, is Sreplay [10]. This tool provides for deterministic group replay in one-sided communication. Similarly, PRUNERS [11] employs reproducibility to uncover non-deterministic errors. And Chapp et al. [12] describe a three-phase workflow to express non-determinism in HPC applications.

Other tools have been developed to automatically detect a variety of error conditions in MPI programs. DeFreez et al. [13] combine static analysis with program repair to detect incorrect propagation of MPI error codes in MPICH and its derivative software. They also employ fault injection techniques to reproduce bugs due to incorrect error-code propagation. This work adapts earlier work on Linux [14], [15] to the special requirements of MPI.

One also notes Intel® Message Checker [16], which works to detect incorrect calls to MPI functions. Errors that are caught include: types of mismatches, race conditions, deadlocks and potential deadlocks, and resource misuse. Likewise, Sato et al. [17] use noise injection to discover unintended race conditions for MPI messages.

There are, of course, many commercial debuggers to debug MPI applications. These include Arm DDT [18], [19] (where DDT is Distributed Debugging Tool) and TotalView (originally part of the PALLAS programming environment [20]) are most commonly used, and many HPC computing centers provide access to these debuggers. Both TotalView and DDT provide intuitive graphic user interfaces as well as command-line interfaces, and can be used to debug MPI (and OpenMP) programs written in C/C++ and Fortran run at scale.

In addition, other parallel debuggers are available either from open source projects or vendor extensions to detect a wider spectrum of bugs. STAT (the Stack Trace Analysis Tool) [21] is a highly scalable lightweight tool that gathers and merges stack traces from all of the processes of a parallel application into a prefix tree to identify which processes are executing similar code. Cray's ATP (Abnormal Termination Processing) automatically runs STAT when the code crashes. These tools are specifically useful to detect program hangs or crashes at large scale.

However, these general-purpose debugging tools are for debugging MPI applications, but not the MPI implementations themselves. These tools include DDT's Message Queues feature, which can show the status of the message buffers of MPI. (Note that the message queues feature does not work with Cray MPICH [22].) Further, runtime MPI checkers, such as Umpire [23], Marmot [24], and MUST [25] can detect the MPI semantics at runtime, but they cover only bugs in the application, and not in the MPI library [26].

There are other tools, which can debug an MPI library, such as FlowChecker [27], which extracts program intentions of message passing, and checks whether these intentions are fulfilled correctly by the underlying MPI libraries. If the intentions are not fulfilled correctly, then it reports the bugs and provides diagnostic information. FlowChecker, however, is not intended for debugging MPI's non-deterministic communication bugs.

There are a few record-and-replay tools that can detect non-deterministic errors. For example, ReMPI [28] is a highly scalable record-and-replay tool for MPI applications. It records the order of MPI message matching in one run, and can deterministically replay it during subsequent runs. However, it cannot be used in combination with MANA to compare the original run (which resumes after a checkpoint), and a second run which restarts a new process from the checkpoint image files on disk. Deterministic-P2P (presented here) was created for this particular purpose.

Review of transparent checkpointing for MPI:

Many early MPI checkpointing approaches were tied too closely to a specific underlying network, whether it was TCP/IP (DMTCP [29]; MPICH-V [30]) or InfiniBand an InfiniBand plugin for DMTCP [31]; (the Open MPI checkpoint-restart service [32], [33]; and the MVAPICH2 checkpoint-restart service [34]). The approach of all except the two examples based on DMTCP was to disconnect from the network prior to checkpoint, and then to re-connect to the network when resuming the computation (or when restarting from a checkpoint image). In the case of Open MPI and MVAPICH2, they delegated to BLCR [35] to do the actual checkpoint. BLCR could checkpoint a tree of processes on a *single node*.

An approach to support mobile MPI applications exists, albeit while partially abandoning application transparency and requiring re-compilation of the MPI application source code [36]. And CIFTS provides a fault-tolerant BLCR-based "backplane" [37].

MANA is unusual in modifying the execution behavior of a target MPI application even prior to checkpointing. MANA introduced the *split-process model* for checkpointing of MPI [3]. Details are in Section II-A. The first work [3] demonstrated transparent checkpointing of GROMACS [38] and an additional three applications at 2048 MPI processes over 64 Haswell nodes. Efforts to deploy MANA at NERSC are described in [39], and efforts to deploy independently of the NERSC environment are described in [40]. The second work [40] demonstrated transparent checkpointing for 64 processes with GROMACS and 512 processes with the HPCG benchmark [41].

Note that transparent checkpointing of MPI had previously been demonstrated to the level of 16,368 processes for NAMD and 32,368 processes for HPCG (using 1/3 of the supercomputer) on Stampede at TACC in 2016 [42]. That early work was based on DMTCP's transparent support for InfiniBand [31], and that code likely would not run on today's machines using either Cray GNI or extensions to the original InfiniBand. Further, no effort was made in the current work to test the limits of scalability of MANA-2.0.

VI. Conclusion

Two debugging tools were introduced: Collective-to-P2P and Deterministic-P2P. The tools may be used individually or in combination. The tools are particularly valuable in debugging MPI implementations or checkpointing tools that fail to preserve the communication-determinism that would be present when the MPI application is run with a simpler, reference implementation of MPI. The tools presented here increase the runtime of an MPI execution by several times (and even by more than a factor of ten, when used in combination). However, since the tools need only be used once, as part of a larger debugging strategy, this is considered an acceptable trade-off. As described in the section on related work, the ability to transform the execution of an MPI application for purposes of diagnosing a bug seems to be unique at this time.

Acknowledgments

This work used the resources of the National Energy Scientific Computing Center (NERSC) at the Lawrence Berkeley National Laboratory. The work of the first author was partially supported by NSF Grant OAC-1740218.

References

[1] Y. Xu, Z. Zhao, R. Garg, H. Khetawat, R. Hartman-Baker, and G. Cooperman, "MANA-2.0: A future-proof design for transparent checkpointing of MPI at scale," in *Int. Symp. on Checkpointing for Supercomputing (SuperCheck'SC-21), 2021 SC Workshops Supplementary Proceedings*. IEEE, Nov. 2021, pp. 68–78.

[2] F. Cappello, A. Guermouche, and M. Snir, "On communication determinism in parallel HPC applications," in *2010 Proceedings of 19th International Conference on Computer Communications and Networks*. IEEE, 2010, pp. 1–8.

[3] R. Garg, G. Price, and G. Cooperman, "MANA for MPI: MPI-agnostic network-agnostic transparent checkpointing," in *Proc. of the 28th Int. Symp. on High-Performance Parallel and Distributed Computing*. ACM, 2019, pp. 49–60.

[4] A. Guermouche, T. Ropars, E. Brunet, M. Snir, and F. Cappello, "Uncoordinated checkpointing without domino effect for send-deterministic mpi applications," in *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE, 2011, pp. 989–1000.

[5] F. Cappello, A. Guermouch, T. Herault, and M. Snir, "Revisiting fault tolerant protocols for parallel HPC applications," in *Proc. of the 2nd Workshop of the Joint Laboratory for Petascale Computing*, 2009.

[6] J. Hafner, "Ab-initio simulations of materials using VASP: Density-functional theory and beyond," *Journal of computational chemistry*, vol. 29, no. 13, pp. 2044–2078, 2008.

[7] "VASP: Vienna Ab Initio Simulation Package," https://www.vasp.at, [Online; accessed 21-Nov-2018].

[8] J. Hutter, M. Iannuzzi, F. Schiffmann, and J. VandeVondele, "CP2K: atomistic simulations of condensed matter systems," *Wiley Interdisciplinary Reviews: Computational Molecular Science*, vol. 4, no. 1, pp. 15–25, 2014.

[9] Y. Xu and G. Cooperman, "Low runtime overhead for transparent checkpointing of MPI," September 2022, (in preparation).

[10] X. Qian, K. Sen, P. Hargrove, and C. Iancu, "Sreplay: Deterministic group replay for one-sided communication," in *30th International Conference on Supercomputing (ICS'16)*, 2016.

[11] K. Sato, I. Laguna, G. L. Lee, M. Schulz, C. M. Chambreau, S. Atzeni, M. Bentley, G. Gopalakrishnan, Z. Rakamaric, G. Sawaya *et al.*, "PRUNERS: Providing reproducibility for uncovering non-deterministic errors in runs on supercomputers," *The International Journal of High Performance Computing Applications*, vol. 33, no. 5, pp. 777–783, 2019.

[12] D. Chapp, D. Rorabaugh, K. Sato, D. H. Ahn, and M. Taufer, "A three-phase workflow for general and expressive representations of nondeterminism in HPC applications," *The International Journal of High Performance Computing Applications*, vol. 33, no. 6, pp. 1175–1184, 2019.

[13] D. DeFreez, A. Bhowmick, I. Laguna, and C. Rubio-González, "Detecting and reproducing error-code propagation bugs in MPI implementations," in *Proc. of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2020, pp. 187–201.

[14] H. S. Gunawi, C. Rubio-González, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and B. Liblit, "EIO: Error handling is occasionally correct." in *FAST*, vol. 8, 2008, pp. 1–16.

[15] C. Rubio-González, H. S. Gunawi, B. Liblit, R. H. Arpaci-Dusseau, and A. C. Arpaci-Dusseau, "Error propagation analysis for file systems," in *Proc. of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009, pp. 270–280.

[16] J. DeSouza, B. Kuhn, B. R. De Supinski, V. Samofalov, S. Zheltov, and S. Bratanov, "Automated, scalable debugging of MPI programs with Intel® Message Checker," in *Proc. of Second International Workshop on Software Engineering for High Performance Computing System Applications*, 2005, pp. 78–82.

[17] K. Sato, D. H. Ahn, I. Laguna, G. L. Lee, M. Schulz, and C. M. Chambreau, "Noise injection techniques to expose subtle and unintended message races," in *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2017, pp. 89–101.

[18] K. Antypas, "Allinea DDT as a parallel debugging alternative to totalview," Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), Tech. Rep., 2007.

[19] B. Krammer, V. Himmler, D. Lecomber *et al.*, "Coupling DDT and Marmot for debugging of MPI applications." in *Parallel Computing Conference (ParCo'07)*, vol. 7, 2007, pp. 653–660.

[20] W. Krotz-Vogel and H.-C. Hoppe, "The pallas portable parallel programming environment," in *European Conference on Parallel Processing*. Springer, 1996, pp. 897–903.

[21] D. C. Arnold, D. H. Ahn, B. R. De Supinski, G. L. Lee, B. P. Miller, and M. Schulz, "Stack trace analysis for large scale debugging," in *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2007, pp. 1–10.

[22] "Arm forge user guide (version 22.04)," 2022. [Online]. Available: https://developer.arm.com/documentation/101136/22-0-4/DDT/Message-queues/View-message-queues

[23] J. S. Vetter and B. R. De Supinski, "Dynamic software testing of MPI applications with Umpire," in *SC'00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*. IEEE, 2000, pp. 51–51.

[24] B. Krammer, M. S. Müller, and M. M. Resch, "MPI application development using the analysis tool MARMOT," in *International Conference on Computational Science*. Springer, 2004, pp. 464–471.

[25] T. Hilbrich, M. Schulz, B. R. d. Supinski, and M. S. Müller, "MUST: A scalable approach to runtime error detection in MPI programs," in *Tools for high performance computing 2009*. Springer, 2010, pp. 53–66.

[26] I. Laguna, D. H. Ahn, B. R. de Supinski, T. Gamblin, G. L. Lee, M. Schulz, S. Bagchi, M. Kulkarni, B. Zhou, Z. Chen, and F. Qin, "Debugging high-performance computing applications at massive scales," in *Communications of the ACM*, 2015, pp. 72–81.

[27] Z. Chen, Q. Gao, W. Zhang, , and F. Qin, "Flowchecker: Detecting bugs in mpi libraries via message flow checking," in *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010, pp. 1–11.

[28] K. Sato, D. H. Ahn, I. Laguna, G. L. Lee, and M. Schulz, "Clock delta compression for scalable order-replay of non-deterministic parallel applications," in *SC'15: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2015.

[29] J. Ansel, K. Arya, and G. Cooperman, "DMTCP: Transparent checkpointing for cluster computations and the desktop," in *2009 IEEE International Symposium on Parallel & Distributed Processing (IPDPS'09)*. Rome, Italy: IEEE, 2009, pp. 1–12.

[30] A. Bouteiller, T. Herault, G. Krawezik, P. Lemarinier, and F. Cappello, "MPICH-V project: A multiprotocol automatic fault-tolerant MPI," *The International Journal of High Performance Computing Applications*, vol. 20, no. 3, pp. 319–333, 2006.

[31] J. Cao, G. Kerr, K. Arya, and G. Cooperman, "Transparent checkpoint-restart over InfiniBand," in *ACM Symposium on High Performance Parallel and and Distributed Computing (HPDC'14)*. ACM Press, 2014.

[32] J. Hursey, J. M. Squyres, T. I. Mattox, and A. Lumsdaine, "The design and implementation of checkpoint/restart process fault tolerance for Open MPI," in *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2007, pp. 1–8.

[33] J. Hursey, T. I. Mattox, and A. Lumsdaine, "Interconnect agnostic checkpoint/restart in Open MPI," in *Proc. of 18th ACM Int. Symp. on High Performance Distributed Computing*, 2009, pp. 49–58.

[34] Q. Gao, W. Yu, W. Huang, and D. K. Panda, "Application-transparent checkpoint/restart for MPI programs over InfiniBand," in *Int. Conf. on Parallel Processing (ICPP'06)*, 2006, pp. 471–478.

[35] P. H. Hargrove and J. C. Duell, "Berkeley Lab Checkpoint/Restart (BLCR) for Linux clusters," *Journal of Physics: Conference Series*, vol. 46, no. 1, p. 494, 2006.

[36] R. Fernandes, K. Pingali, and P. Stodghill, "Mobile MPI programs in computational grids," in *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, 2006, pp. 22–31.

[37] R. Gupta, P. Beckman, B.-H. Park, E. Lusk, P. Hargrove, A. Geist, D. Panda, A. Lumsdaine, and J. Dongarra, "CIFTS: A coordinated infrastructure for fault-tolerant systems," in *Int. Conf. on Parallel Processing (ICPP'09)*, September 2009, pp. 237–245.

[38] H. Berendsen, D. van der Spoel, and R. van Drunen, "GROMACS: A message-passing parallel molecular dynamics implementation," *Computer Physics Communications*, vol. 91, no. 1, pp. 43 – 56, 1995.

[39] Z. Zhao, R. Hartman–Baker, and G. Cooperman, "Deploying Checkpoint/Restart for Production Workloads at NERSC," in *International Conference for High Performance Computing Networking Storage and Analysis*, 2020, pp. 1–3.

[40] P. S. Chouhan, H. Khetawat, N. Resnik, T. Jain, R. Garg, G. Cooperman, R. Hartman–Baker, and Z. Zhao, "Improving scalability and reliability of MPI-agnostic transparent checkpointing for production workloads at NERSC (extended abstract)," in *First International Symposium on Checkpointing for Supercomputing (SuperCheck'21)*, Berkeley, CA, 2021, pp. 1–3, https://arxiv.org/abs/2103.08546; from https://supercheck.lbl.gov/resources.

[41] J. Dongarra, M. A. Heroux, and P. Luszczek, "A new metric for ranking high-performance computing systems," *National Science Review*, 2016, (benchmark at https://www.hpcg-benchmark.org/).

[42] J. Cao, K. Arya, R. Garg, S. Matott, D. K. Panda, H. Subramoni, J. Vienne, and G. Cooperman, "System-level scalable checkpoint-restart for petascale computing," in *22nd IEEE Int. Conf. on Parallel and Distributed Systems (ICPADS'16)*. IEEE Press, 2016, pp. 932–941.