MANA for MPI: MPI-Agnostic Network-Agnostic Transparent Checkpointing

Rohan Garg*
Northeastern University
Boston, MA
rohgarg@ccs.neu.edu

Gregory Price
Raytheon Company
Annapolis Junction, MD
gregory.m.price@raytheon.com

Gene Cooperman*
Northeastern University
Boston, MA
gene@ccs.neu.edu

ABSTRACT

Transparently checkpointing MPI for fault tolerance and load balancing is a long-standing problem in HPC. The problem has been complicated by the need to provide checkpoint-restart services for all combinations of an MPI implementation over all network interconnects. This work presents MANA (MPI-Agnostic Network-Agnostic transparent checkpointing), a single code base which supports all MPI implementation and interconnect combinations. The agnostic properties imply that one can checkpoint an MPI application under one MPI implementation and perhaps over TCP, and then restart under a second MPI implementation over InfiniBand on a cluster with a different number of CPU cores per node. This technique is based on a novel *split-process* approach, which enables two separate programs to co-exist within a single process with a single address space. This work overcomes the limitations of the two most widely adopted transparent checkpointing solutions, BLCR and DMTCP/InfiniBand, which require separate modifications to each MPI implementation and/or underlying network API. The runtime overhead is found to be insignificant both for checkpoint-restart within a single host, and when comparing a local MPI computation that was migrated to a remote cluster against an ordinary MPI computation running natively on that same remote cluster.

CCS CONCEPTS

• **Computer systems organization → Reliability**; *Dependable and fault-tolerant systems and networks*; • **Software and its engineering → Checkpoint / restart**.

ACM Reference Format:
Rohan Garg*, Gregory Price, and Gene Cooperman*. 2019. MANA for MPI: MPI-Agnostic Network-Agnostic Transparent Checkpointing. In *The 28th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '19), June 22–29, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3307681.3325962

1 INTRODUCTION

The use of transparent or system-level checkpointing for MPI is facing a crisis today. The most common transparent checkpointing packages for MPI in recent history are either declining in usage, or abandoned entirely. These checkpointing packages include: the Open MPI [23] checkpoint-restart service, the MVAPICH2 [15] checkpoint-restart service, DMTCP for MPI [1], MPICH-V [7], and a fault-tolerant BLCR-based "backplane", CIFTS [18]. We argue existing transparent or system-level checkpoint approaches share common issues that makes long-term maintenance impractical. In particular, the HPC community requires checkpoint-restart support for any of m popular MPI implementations over n different network interconnects.

We propose MPI-Agnostic Network-Agnostic transparent checkpointing (MANA), a single code base that can support all combinations of the many MPI implementations and network libraries that are in wide use. In particular, it supports all $m \times n$ combinations, where m is the number of MPI implementations and n is the number of underlying network libraries. The new approach, based on a *split-process*, is fully transparent to the underlying MPI, network, libc library, and underlying Linux kernel. MANA is free and open-source software [28]. (Transparent checkpointing supports standard system-level checkpointing, but it can alternatively be customized in an application-specific manner.)

We begin by distinguishing this work from that of Hursey et al. [22], which demonstrated a network-agnostic implementation of checkpointing for a single MPI implementation (for Open MPI). Hursey's work adds network-agnostic checkpointing by "taking down" the network during checkpoint and "building it up" upon resuming — but only within the single Open MPI implementation. Further, it requires maintenance of n code bases, where n is the number of network libraries that Open MPI supports.

In contrast, The new approach of MANA employs a single code base that exists *external to any particular MPI implementation.* At checkpoint time, MANA "disconnects" the application from the MPI and network libraries, and at the time of resuming, it "reconnects" to the MPI and network libraries. Further, at the time of restart, MANA starts a new and independent MPI session (even possibly using a newer version of the original MPI implementation (perhaps due to a system upgrade), or it may even switch MPI implementations.

Next, we present three case studies to demonstrate the declining usage of transparent checkpointing, and how maintenance costs have factored into supressing adoption of even semi-agnostic systems.

First, we consider Hursey et al. [22] and Open-MPI. Open MPI developers created a novel and elegant checkpoint-restart service that was *network-agnostic*, with the ability to checkpoint under network A and restart under network B. The mechanism presented by Hursey et al warrants careful analysis of how it provides *network-agnostic* checkpointing (a primary goal of MANA). Their implementation lifts checkpoint code out of interconnect drivers, applying

*This work was partially supported by NSF Grant OAC-1740218.

HPDC '19, June 22–29, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6670-0/19/06…$15.00
https://doi.org/10.1145/3307681.3325962

a Chandy/Lamport [11] checkpoint algorithm to an abstraction of "MPI Messages".

However, their choice to implement within the MPI library may have ultimately suppressed widespread adoption. Hursey et al. can support multiple MPI implementations only by requiring each MPI implementation to individually integrate the Hursey approach and maintain support for taking down and restoring network interconnects for each supported network library. This imposes a significant maintenance penalty on packages maintainers. Even Open-MPI support remains in question. As of this writing in 2019, the Open-MPI FAQ says: "Note: The checkpoint/restart support was last released as part of the v1.6 series. . . . This feature is looking for a maintainer." [30].

The second case study concerns BLCR. MPICH and several other MPI implementations adopted BLCR for checkpointing. BLCR is based on a kernel module that checkpoints the local MPI rank. BLCR lacks System V shared memory support (widely used for intra-node communication), which severely limits its use in practice. As of this writing, BLCR 0.8.5 (appearing in 2013) was the last officially supported version [5], and formal testing of the BLCR kernel module stopped with Linux 3.7 (Dec., 2012) [6]. Here, again, we argue that BLCR declined not due to any fault with BLCR, but due to the difficulty of supporting or maintaining common interconnects.

The third case study concerns DMTCP. As discussed above, DMTCP/InfiniBand is *MPI-agnostic*, but not *network-agnostic*, requiring a plugin for each network interconnect. While it supports InfiniBand [10], and partially supports Intel Omni-Path [8, Chapter 6], DMTCP does not support Cray GNI Aries network, the Mellanox extensions to InfiniBand (UCX and FCA), the libfabric API [26], and many others.

Separate *MPI-agnostic* or *network-agnostic* checkpoint systems have not been widely adopted, it seems, in-part due to the maintenance costs. MANA attempts to resolve this.

The *split-process* approach eliminates this maintenance penalty, supporting checkpoint/restart on all combinations of MPI library and interconnect with a single codebase. In a *split-process*, a single system process contains two programs in its memory address space. The two programs are an MPI proxy application (denoted the lower-half) and the original MPI application code (denoted the upper-half). MANA tracks which memory regions belong to the upper and lower halves. At checkpoint time, only upper-half memory regions are saved. MANA is also fully transparent to the specific MPI implementation, network, libc library and Linux kernel.

At restart time, MANA initializes a new MPI library and underlying interconnect network in the lower half of a process. The checkpointed MPI application code and data is then copied in and restored into the upper half from the checkpoint image file. By initializing a new MPI library at the time of restart, MANA provides excellent load-balancing support without the need for additional logic. The fresh initialization inherently detects the correct number of CPU cores per node, optimizes the topology as MPI ranks from the same node may now be split among distinct nodes (or vice verse), re-optimizes the rank-to-host bindings for any MPI topology declarations in the MPI application, and so on. (See Section 4 for further discussion.)

MANA maintains low runtime overhead, avoiding RPC by taking advantage of a split-process to make library calls within the same process. Other Proxy-based approaches had been previously used for checkpointing in general applications [39] and CUDA (GPU-accelerated) applications [16, 34, 35]. However, such approaches incur significant overhead due to context switching and copying buffers between the MPI application and the MPI proxy.

Scalability is a key criterion for MANA. This criterion motivates the use of version 3.0 of DMTCP [1] as the underlying package for transparent checkpointing. DMTCP was previously used to demonstrate petascale-level transparent checkpointing using Lustre. In particular this was applied to MPI-based HPCG over 32,752 CPU cores (checkpointing an aggregate 38 TB in 11 minutes), and to MPI-based NAMD over 16,368 cores (checkpointing an aggregate 10 TB in 2.6 minutes) [9]. DMTCP employs a stateless centralized checkpoint coordinator for its results, and MANA re-uses for its own purposes this same coordinator. (The single coordinator is not a barrier to scalability, since it can use a broadcast tree for communication with its peers.)

Next in this work, Section 2 describes the design issues in fitting the split-process concept to checkpoint MPI. In particular, the issue of checkpointing during MPI collective communications is discussed. Section 3 presents an experimental evaluation. Section 4 discusses current and future inquiries opened up by these new ideas. Section 5 discusses related work. Finally, Section 6 presents the conclusion.

2 MANA: DESIGN AND IMPLEMENTATION

Multiple aspects of the design of MANA are covered in this section. Section 2.1 discusses the design for supporting a split-process. Section 2.2 discusses the need to save and restore persistent MPI opaque objects, such as communicators, groups and topologies. Section 2.3 briefly discusses the commonly used algorithm to drain point-to-point MPI messages in transit prior to intiaiting a checkpoint. Sections 2.4 and 2.5 present a new two-phase algorithm (Algorithm 2), which enables checkpointing in-progress MPI collective communication calls in a fully agnostic environment. Finally, Sections 2.6 and 2.7 present details of the overall implementation of MANA.

2.1 Upper and Lower Half: Checkpointing with an Ephemeral MPI Library

In this section, we define the *lower half* of a split-process as the memory associated with the MPI library and dependencies, including network libraries. The *upper half* is the remaining Linux process memory associated with the MPI application's code, data, stack, and other regions (e.g., environment variables). The terms *lower half* and *upper half* are in analogy with the upper and lower half of a device driver in an operating system kernel. This separation into lower and upper half does not involve additional threads or processes. Instead, it serves primarily to tag memory so that only upper half memory will be saved or restored during checkpoint and restart. Section 2.6 describes an additional "helper thread", but that thread is active only during checkpoint and restart.

Libc and other system libraries may appear in both the lower half as a dependency of the MPI libraries, and the upper half as an independent dependency of the MPI application.

This split-process approach allows MANA to balance two conflicting objectives: a shared address space; and isolation of upper and lower halves. The isolation allows MANA to omit the lower half memory (an "ephemeral" MPI library) when it creates a checkpoint image file. The shared address space allows the flow of control to pass efficiently from the upper-half MPI application to the lower-half MPI library through standard C/Fortran calling conventions, including call by reference. As previously noted, Remote Produce Calls (RPC) are not employed.

Isolation is needed so that at checkpoint time, the lower half can be omitted from the checkpoint image, and at the time of restart, replaced with a small "bootstrap" MPI program with new MPI libraries. The bootstrap program calls `MPI_Init()` and each MPI process discovers its MPI rank via a call to `MPI_Rank()`. The memory present at this time becomes the lower half. The MPI process then restores the upper-half memory from a checkpoint image file corresponding to the MPI rank id. Control is then transferred back to the upper-half MPI application, and the stack in the lower half is never used again.

Shared address space is needed for efficiency. A dual-process proxy approach was explored in [16, Section IV.B] and in [35, Section IV.A]. The former work reported a 6% runtime overhead for real-world CUDA applications, and the latter work reported runtime overheads in excess of 20% for some OpenCL examples from the NVIDIA SDK 3.0. In contrast, Section 3 reports runtime overheads less than 2% for MANA under older Linux kernels, and less than 1% runtime overhead for recent Linux kernels.

Discarding the lower half greatly simplifies the task of checkpointing. By discarding the lower half, the MPI application in the upper half appears as an isolated process with no inter-process communication. Therefore, a single-process checkpointing package can create a checkpoint image.

A minor inconvenience of this split-process approach is that calls to `sbrk()` will cause the kernel to extend the process heap in the data segment. Calls to `sbrk()` can be caused by invocations of `malloc()`. Since the kernel has no concept of a split-process, the kernel may choose, for example, to extend the lower half data segment after restart since that corresponds to the original program seen by the kernel before the upper-half memory is restored. MANA resolves this by interposing on calls to `sbrk()` in the upper-half libc, and then inserts calls to `mmap()` to extend the heap of the upper-half.

Finally, MANA employs coordinated checkpointing, and a checkpoint coordinator sends messages to each MPI rank at the time of checkpoint (see Sections 2.3, 2.4 and 2.5). MPI opaque objects (communicators, groups, topologies) are detected on creation and restored on restart (see Section 2.2). This is part of a broader strategy by which MPI calls with persistent effects (such as creation of these opaque objects) are recorded during runtime and replayed on restart.

2.2 Checkpointing MPI Communicators, Groups, and Topologies

An MPI application can create communication subgroups and topologies to group processes for ease of programmability and efficient communication. MPI implementations provide opaque handles to the application as a reference to a communicator object or group.

MANA interposes on all calls that refer to these opaque identifiers, and virtualizes the identifiers. At runtime, MANA records any MPI calls that can modify the MPI communication state, such as `MPI_Comm_create`, `MPI_Group_incl`, etc. On restart, MANA recreates the MPI communicator state by replaying the MPI calls using a new MPI library. The runtime virtualization of identifiers allows the application to continue running with consistent handles across checkpoint-restart.

A similar checkpointing strategy also works for other opaque identifiers, such as, MPI derived datatypes, etc.

2.3 Checkpointing MPI Point-to-Point Communication

Capturing the state of MPI processes requires quiescing the process threads, and preserving the process memory to a file on the disk. However, this alone is not sufficient to capture a consistent state of the computation. Any MPI messages sent but not yet received at the time of quiescing processes must also be saved as part of the checkpoint image.

MANA employs a variation of an all-to-all bookmark exchange algorithm to reach this consistent state. LAM/MPI [31] demonstrated the efficacy of a such a Chandy/Lamport [11] algorithm for checkpointing MPI applications. Hursey et al. [22] lifted this mechanism out of interconnect drivers and into the MPI library. MANA further lifts this mechanism outside the MPI library, and into a virtualized MPI API.

An early prototype of MANA demonstrated a naïve application of this bookmark exchange algorithm was sufficient for handling pre-checkpoint draining for point-to-point communication; however, collective-communication calls may have MPI implementation effects that can determine when it is "safe" to begin a checkpoint. For this reason, a naïve application to the entire API was insufficient to ensure correctness. This is discussed in Section 2.4.

2.4 Checkpointing MPI Collectives: Overview

The MPI collective communications primitive involves communication amongst all or a program-defined subset of MPI ranks (as specified by the MPI communicator argument to the function). The internal behavior of collectives are specific to each MPI implementation, and so it is not possible to make guarantees about their behavior, such as when and how messages are exchanged when ranks are waiting for one or more ranks to enter the collective.

In prior work [22, 31], internal knowledge of the MPI library state was required to ensure that checkpointing would occur at a "safe" state. In particular, Hursey et al. [22] required interconnect drivers be classified as "checkpoint-friendly" or "checkpoint-unfriendly", changing behavior based on this classification. As MANA lives outside the MPI library, a naive application of the Hursey et al. algorithm can have effects that cross the upper and lower half boundaries of an MPI rank (for example, when shared memory is being used for MPI communication).

This problem occurs because of the truly *network-agnostic* trait of MANA. As MANA has no concept of transport level constructs, it cannot determine what "safe" means in context of collectives. To

correct this, MANA's support for collective communication requires it to maintain the following invariant:

> *No checkpoint must take place while a rank is inside a collective communication call.*

There exists one exception to this rule: a *trivial barrier*. A *trivial barrier* is a simple call to MPI_Barrier(). This call produces no side effects on an MPI rank, and so it can be safely interrupted during checkpoint, and then re-issued when restarting the MPI application. This is possible due to the split-process architecture of MANA, as *trivial barrier* calls occur exclusively in the lower half, which is discarded and replaced across checkpoint and restart. MANA leverages this exception to build a solution for all other collective calls.

As we discuss MANA's algorithm for checkpointing collective calls, we take into consideration three subtle, but important, concerns.

Challenge I (consistency): In the case of a single MPI collective communication call, there is a danger that rank A will see a request to checkpoint before entering the collective call, while rank B will see the same request after entering the collective call, in violation of MANA's invariant. Both ranks might report that they are ready to checkpoint, and the resulting inconsistent snapshot will create problems during restart. This situation could arise, for example, if the message from the checkpoint coordinator to rank B is excessively delayed in the network. To resolve this, MANA introduces a two-pass protocol in which the coordinator makes a request (sends an intend-to-checkpoint message), each MPI rank acknowledges with its current state, and finally the coordinator posts a checkpoint request (possibly preceded by extra iterations).

Challenge II (progress and latency): Given the aforementioned solution for consistency, long delays may occur before a checkpoint request can be serviced. It may be that rank A has entered the barrier, and rank B will require several hours to finish a task before entering the barrier. Hence, the two-pass protocol may create unacceptable delays before a checkpoint can be taken. Algorithm 2 addresses this by introducing a *trivial barrier* prior to the collective communication call. We refer to this as a *two-phase algorithm* since each collective call is now replaced by a wrapper function that invokes a trivial barrier call (phase 1) followed by the original collective call (phase 2).

Challenge III (multiple collective calls): Until now, it was assumed that at most one MPI collective communication call was in progress at the time of checkpoint. It may happen that there are multiple ongoing collective calls. During the time that some MPI ranks exit from a collective call, it may happen that MPI ranks associated with an independent collective call have left the MPI *trivial barrier* (phase 1) and have now entered the real collective call (phase 2). As a result, servicing a checkpoint may be excessive delayed. To solve this, we introduce an intend-to-checkpoint message, such that no ranks will be allowed to enter phase 2, and extra iterations will be inserted into the request-acknowledge protocol between coordinator and MPI rank.

2.5 Checkpointing MPI Collectives: Detailed Algorithm

Here we present a single algorithm (Algorithm 2) for checkpointing MPI collectives which contains the elements described in Section 2.4: a multi-iteration protocol; and a two-phase algorithm incorporating a *trivial barrier* before any collective communication call.

From the viewpoint of an MPI application, any call to an MPI collective communication function is interposed on by a wrapper function, as shown in Algorithm 1.

Algorithm 1 Two-Phase collective communication wrapper. (This wrapper function interposes on all MPI collective communication functions invoked by an MPI application)

```
1: function Collective Communication Wrapper
2:     # Begin Phase 1
3:     Call MPI_Barrier() # trivial barrier
4:     # Begin Phase 2
5:     Call original MPI collective communication function
6: end function
```

Recall that a *trivial barrier* is an extra call to `MPI_Barrier()` prior to a collective call. A collective MPI call can intuitively be divided into two parts: the participating MPI ranks "register" themselves as ready for the collective communication; and then the "work" of communication is carried out. Where the time for the collective communication calls of an MPI program is significant, it is typically due to significant "work" in the second part of the calls. Adding a trivial barrier requires the MPI ranks to register themselves once for the trvial barrier (but no work is involved), and then register themselves again for the actual MPI collective communication. The overhead due to registering twice is tiny in practice. Evidence for this can be seen in the experiments in Section 3.2.3, which show small overhead.

The purpose of Algorithm 1 is to enforce the following extension of the invariant presented in Section 2.4:

> *No checkpoint must take place while a rank is inside the collective communication call (Phase 2) of a wrapper function for collective communication (Algorithm 1).*

We formalize this with the following theorem, which guarantees Algorithm 2 satisfies this invariant.

Theorem 1. *Under Algorithm 2, an MPI rank is never inside a collective communication call when a checkpoint message is received from the checkpoint coordinator.*

The proof of this theorem is deferred until the end of this subsection. We begin the path to this proof by stating an axiom that serves to define the concept of a barrier.

Axiom 1. *For a given invocation of an MPI barrier, it never happens that a rank A exits from the barrier before another rank B enters the barrier under the "happens-before" relation.*

Next, we present the following two lemmas.

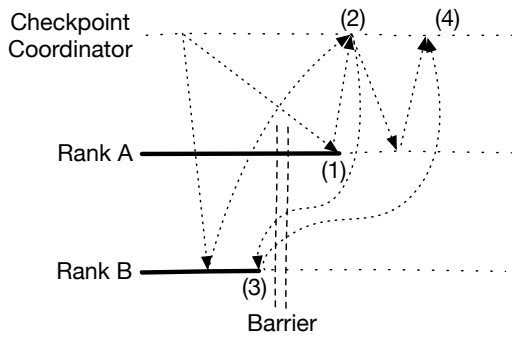


Figure 1: Fundamental "happens-before" relation in communication between the checkpoint coordinator and the MPI ranks involved in an MPI barrier.

Lemma 1. *For a given MPI barrier, if the checkpoint coordinator sends a message to each MPI rank participating in the barrier, and if at least one of the reply messages from the participating ranks reports that its rank has exited the barrier, then the MPI coordinator can send a second message to each participating rank, and each MPI rank will reply that it has entered the barrier (and perhaps also exited the barrier).*

Proof. We prove the lemma by contradiction. Suppose that the lemma does not hold. Figure 1 shows the general case in which this happens. At event 4, the checkpoint coordinator will conclude that event 1 (rank A has exited the MPI barrier) happened before event 2 (the first reply by each rank), which happened before event 3 (in which rank B has not yet entered the barrier). But this contradicts Axiom 1. Therefore, our assumption is false, and the lemma does indeed hold. □

Lemma 2. *Recall that an MPI collective communication wrapper makes a call to a trivial barrier and then makes an MPI collective communication call. For a given invocation of an MPI collective communication wrapper, we know that one of four cases must hold:*

(a) *an MPI rank is in the collective communication call, and all other ranks are either in the call, or have exited;*
(b) *an MPI rank is in the collective communication call, and no rank has exited, and every other rank has at least entered the trivial barrier (and possibly proceeded further);*
(c) *an MPI rank is in the trivial barrier and no other rank has exited (but some may not yet have entered the trivial barrier);*
(d) *either no MPI rank has entered the trivial barrier, or all MPI ranks have exited the MPI collective communication call.*

Proof. The proof is by repeated application of Lemma 1. For case a, if an MPI rank is in the collective communication call and another rank has exited the collective call, then Lemma 1 says that there cannot be any rank that has not yet entered the collective call. For case b, note that if an MPI rank is in the collective communication call, then that rank has exited the trivial barrier. Therefore, by Lemma 1, all other ranks have at least entered the trivial barrier. Further, we can assume that no ranks that have exited the collective call, since we would otherwise be in case a, which is already accounted for. For case c, note that if an MPI rank is in the trivial barrier and no rank has exited the trivial barrier, then Lemma 1 says that there cannot be any rank that has not yet entered the trivial barrier. Finally, if we are not in case a, b, or c, then the only remaining possibility is case d: all ranks have not yet entered the trivial barrier or all ranks have exited the collective call. □

Algorithm 2 Two-Phase algorithm for deadlock-free checkpointing of MPI collectives

```
1:  Messages: {intend-to-checkpoint, extra-iteration, do-ckpt}
2:  MPI states: {ready, in-phase-1, exit-phase-2}
3:  Process Checkpoint Coordinator do
4:      function Begin Checkpoint
5:          send intend-to-ckpt msg to all ranks
6:          receive responses from each rank
7:          while some rank in state exit-phase-2 do
8:              send extra-iteration msg to all ranks
9:              receive responses from each rank
10:         end while
11:         send do-ckpt msg to all ranks
12:     end function
13: Process MPI Rank do
14:     upon event intend-to-ckpt msg or extra-iteration msg do
15:         if not inCollectiveWrapper then
16:             reply to ckpt coord: state ← ready
17:         end if
18:         if inCollectiveWrapper and in Phase 1 then
19:             reply to ckpt coord: state ← in-phase-1
20:         end if
21:         if inCollectiveWrapper and in Phase 2 then
22:             # guaranteed ckpt coord won't request ckpt here
23:             finish executing coll. comm. call
24:             reply to ckpt coord: state ← exit-phase-2
25:             # ckpt coord can request ckpt after this
26:             set state ← ready
27:         end if
28:         continue, but wait before next coll. comm. call
29:     upon event do-ckpt msg do
30:         # guaranteed now that no rank is in phase 2 during ckpt
31:         do local checkpoint for this rank
32:         # all ranks may now continue executing
33:         if this rank is waiting before coll. comm. call then
34:             unblock this rank and continue executing
35:         end if
```

We now continue with the proof of the main theorem (Theorem 1), which was deferred earlier.

Proof. *(Proof of Theorem 1 for Algorithm 2.)* Lemma 2 states that one of four cases must hold in a call by MANA to an MPI collective communication wrapper. We wish to exclude the possibility that an MPI rank is in the collective communication call (case a or b of the lemma) when the checkpoint coordinator invokes a checkpoint.

In Algorithm 2, assume that the checkpoint coordinator has sent an *intend-to-ckpt* message, and has not yet sent a *do-ckpt* message.

An MPI rank will either reply with state *ready* or *in-phase-1* (showing that it is not in the collective communication call and that it will stop before entering the collective communication call), or else it must be in Phase 2 of the wrapper (potentially within the collective communication call), and it will not reply to the coordinator until exiting the collective call. □

Theorem 2. *Under Algorithm 2, deadlock will never occur. Further, the delay between the time when all ranks have received the* intend-to-checkpoint *message and the time when the* do-ckpt *message has been sent is bounded by the maximum time for any individual MPI rank to enter and exit the collective communication call, plus network message latency.*

Proof. The algorithm will never deadlock, since each rank must either make progress based on the normal MPI operation or else it stops *before* the collective communication call. If any rank replies with the state *exit-phase-2*, then the checkpoint coordinator will send an additional *extra-iteration* message. So, at the time of checkpoint, all ranks will have state *ready* or *in-phase-1*.

Next, the delay between the time when all ranks have received the *intend-to-checkpoint* message and the time when the *do-ckpt* message has been sent is clearly bounded by the maximum time for an individual MPI rank to enter and exit the collective communication call, plus the usual network message latency. This is the case since once the *intend-to-checkpoint* message is received, no MPI rank may enter the collective communication call. So, upon receiving the *intend-to-checkpoint* message, either the rank is already in Phase 2 or else it will remain in Phase 1. □

Implementation of Algorithm 2: At the time of process launch for an MPI rank, a separate checkpoint helper thread is also injected into each rank. This thread is responsible for listening to checkpoint-related messages from a separate coordinator process and then responding. This allows the MPI rank to asynchronously process events based on messages received from the checkpoint coordinator. Furthermore at the time of checkpoint, the existing threads of the MPI rank process are quiesced (paused) by the helper thread, and the helper thread carries out the checkpointing requirements, such as copying the upper-half memory regions to stable storage. The coordinator process does not participate in the checkpointing directly. In the implementation, a DMTCP coordinator and DMTCP checkpoint thread [1] are modified to serve as checkpoint coordinator and helper thread, respectively.

2.6 Verification with TLA+/PlusCal

To gain further confidence in our implementation for handling collective communication (Section 2.5), we developed a model for the protocol in TLA+ [25] and then used the PlusCal model checker of TLA+ based on TLC [38] to verify Algorithm 2. Specifically, PlusCal was used to verify the algorithm invariants of deadlock-free execution and consistent state when multiple concurrent MPI processes are executing. The PlusCal model checker did not report any deadlocks or broken invariants for our implementation.

2.7 Checkpoint/Restart Package

Any single-process checkpointing package could be utilized for the basis of implementing MANA. This work presents a prototype implemented by extending DMTCP [1] and by developing a DMTCP plugin [2]. Cao et al. [9] demonstrated that DMTCP can checkpoint MPI-based HPCG over 32,752 CPU cores (38 TB) in 11 minutes, and MPI-based NAMD over 16,368 cores (10 TB) in 2.6 minutes.

DMTCP uses a helper thread inside each application process, and a coordinated checkpointing protocol by using a centralized coordinator daemon. Since this was close to the design requirements of MANA, we leveraged this infrastructure and extended the DMTCP coordinator to implement the two-phase algorithm.

The same approach could be extended to base MANA on top of a different underlying transparent checkpointing package. For example, one could equally well have modified an existing MPI coordinator process to communicate with a custom helper thread in each MPI rank that then invokes the BLCR checkpointing package when it is required to execute the checkpoint. In particular, all sockets and other network communication objects are inside the lower half, and so even a single-process or single-host checkpointing package such as BLCR would suffice for this work.

3 EXPERIMENTAL EVALUATION

This section seeks to answer the following questions:
Q1: What is the runtime overhead of running MPI applications under MANA?
Q2: What are the checkpoint and restart overheads of transparent checkpointing of MPI applications under MANA?
Q3: Can MANA allow transparent switching of MPI implementations across checkpoint-restart for the purpose of load balancing?

3.1 Setup

We first describe the hardware and software setup for MANA's evaluation.

3.1.1 Hardware. The experiments were run on the Cori supercomputer [13] at the National Energy Research Scientific Computing Center (NERSC). As of this writing, Cori is the #12 supercomputer in the Top-500 list [36]. All experiments used the Intel Haswell nodes (dual socket with a 16-core Xeon E5-2698 v3 each) connected via Cray's Aries interconnect network. Checkpoints were saved to the backend Lustre filesystem.

3.1.2 Software. Cori provides modules for two implementations of MPI: Intel MPI and Cray MPICH. The Cray compiler (based on an Intel compiler) and Cray MPICH are the recommended way to use MPI, presumably for reasons of performance. Cray MPICH version 3.0 was used for the experiments.

3.1.3 Application Benchmarks. MANA was tested with five real-world HPC applications from different computational science domains:

(1) GROMACS [4]: Versatile package for molecular dynamics, often used for biochemical molecules.
(2) CLAMR [12, 29]: Mini-application for CelL-based Adaptive Mesh Refinement.

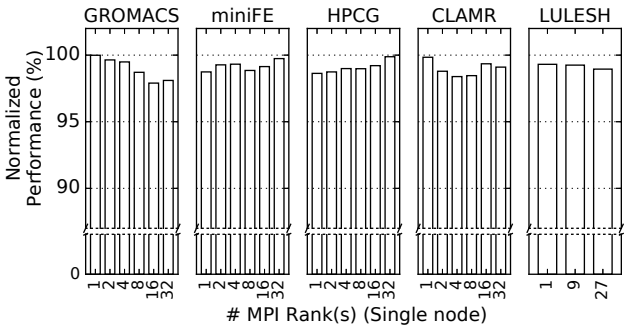


Figure 2: Single Node: Runtime overhead under MANA for different real-world HPC benchmarks with an unpatched Linux kernel. (Higher is better.)

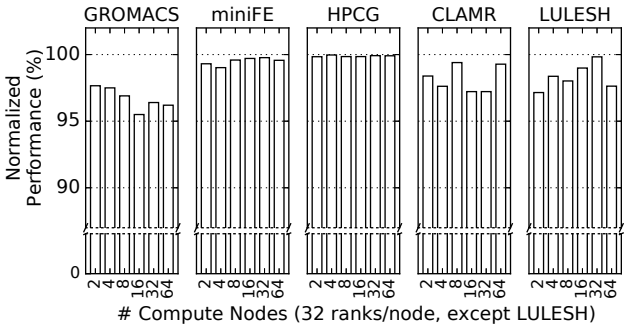


Figure 3: Multiple Nodes: Runtime overhead under MANA for different real-world HPC benchmarks with an unpatched Linux kernel. In all cases, except LULESH, 32 MPI ranks were executed on each compute node. (Higher is better.)

(3) miniFE [20]: Proxy application for unstructured implicit finite element codes.
(4) LULESH [24]: Unstructured Lagrangian Explicit Shock Hydrodynamics
(5) HPCG [14] (High Performance Conjugate Gradient): Uses a variety of linear algebra operations to match a broad set of important HPC applications, and used for ranking HPC systems.

3.2 Runtime Overhead

3.2.1 Real-world HPC Applications. Next, we evaluate the performance of MANA for real-world HPC applications. It will be shown that the runtime overhead is close to 0 % for miniFE and HPCG, and as much as 2 % for the other three applications. The higher overhead has been tracked down to an inefficiency in the Linux kernel [27] in the case of many point-to-point MPI calls (send/receive) with messages of small size. This worst case is analyzed further in Section 3.3, where tests with an optimized Linux kernel show a worst case runtime overhead of 0.6 %. The optimized Linux kernel is based on a patch under review for a future Linux version.

Single Node: Since the tests were performed within a larger cluster where the network use of other jobs could create congestion, we first eliminate any network-related overhead by running the benchmarks on a single node with multiple MPI ranks, both under

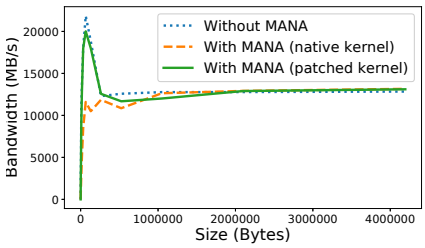


Figure 4: Point-to-Point Bandwidth under MANA with patched and unpatched Linux kernel. (Higher is better.)

MANA and natively (without MANA). This experiment isolates the single-node runtime overhead of MANA by ensuring that all communication among ranks is intra-node.

Figure 2 shows the results for the five different real-world HPC applications running on a single node under MANA. Each run was repeated 5 times (including the native runs), and the figure shows the mean of the 5 runs. The absolute runtimes varied from 4.5 min to 15 min, depending on the configuration. The worst case overhead incurred by MANA is 2.1 % in the case of GROMACS (with 16 MPI ranks). In most cases, the mean overhead is less than 2 %.

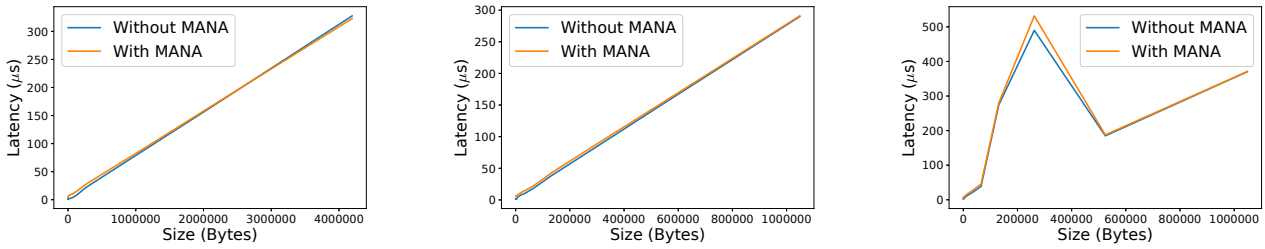
Multiple Nodes: Next, the scaling of MANA across the network is examined for up to 64 compute nodes and with 32 ranks per node (except for LULESH, whose configuration restricts the number of ranks/node based on the number of nodes). Hence, the number of MPI ranks ranges from 64 to 2048.

Figure 3 shows the results of five different real-world HPC applications running on multiple nodes under MANA. Each run was repeated 5 times, and the mean of 5 runs is reported. We observe a trend similar to the single node case. MANA imposes an overhead of typically less than 2 %. The highest overhead observed is 4.5 % in the case of GROMACS (512 ranks running over 16 nodes). However, see Section 3.3 where we demonstrate a reduced overhead of 0.6 % with GROMACS.

3.2.2 Memory Overhead. The upper-half libraries were built with `mpicc`, and hence include additional copies of the MPI library that are not used. However, the upper-half MPI library is never initialized, and so no network library is ever loaded into the upper half.

Since a significant portion of the lower half is comprised only of the MPI library and its dependencies, the additional copy of the libraries (with one copy residing in the upper half) imposes a constant memory overhead. This text segment (code region) was 26 MB in all of our experiments on Cori with the Cray MPI library.

In addition to the code, the libraries (for example, the networking driver library) in the lower half also allocate additional memory regions (shared memory regions, pinned memory regions, memory-mapped driver regions). We observed that the shared memory regions mapped by the network driver library grow in proportion with the number of nodes (up to 64 nodes): from 2 MB (for 2 nodes) to 40 MB for (64 nodes). We expect MANA to have a reduced checkpoint time compared to DMTCP/InfiniBand [10], as MANA discards these regions during checkpointing, reducing the amount of data that's written out to the disk.



(a) Point-to-Point Latency **(b) Collective MPI_Gather** **(c) Collective MPI_Allreduce**

Figure 5: OSU Micro-benchmarks under MANA. (Results are for two MPI ranks on a single node.)

3.2.3 Microbenchmarks. To dig deeper into the sources for the runtime overhead, we tested MANA with the OSU micro-benchmarks. The benchmarks stress and evaluate the bandwidth and latency of different specific MPI subsystems. Our choice of the specific micro-benchmarks was motivated by the MPI calls commonly used by our real-world MPI applications.

Figure 5 shows the results with three benchmarks from the OSU micro-benchmark suite. These benchmarks correspond with the most frequently used MPI subsystems in the set of real-world HPC applications. The benchmarks were run with 2 MPI ranks running on a single compute node.

The results show that latency does not suffer under MANA, for both point-to-point and collective communication. (The latency curves for application running under MANA closely follow the curves when the application is run natively.)

3.3 Source of Overhead and Improved Overhead for Patched Linux Kernel

All experiments in this section were performed on a single node of our local cluster, where it was possible to directly install a patched Linux kernel in the bare machine.

Further investigation revealed two sources of runtime overhead. The larger source of overhead is due to the use of the "FS" register during transfer of flow of control between the upper and lower half and back during a call to the MPI library in the lower half. The "FS" register of the x86-64 CPU is used by most compilers to refer to the thread-local variables declared in the source code. The upper and lower half programs each have their own thread-local storage region. Hence, when switching between the upper and lower half programs, the value of the "FS" register must be changed to point to the correct thread-local region. Most Linux kernels today require a kernel call to invoke a privileged assembly instruction to get or set the "FS" register. In 2011, Intel Ivy Bridge CPUs introduced a new, unprivileged FSGSBASE assembly instruction for modifying the "FS" register, and a patch to the Linux kernel [27] is under review to allow other Linux programs to use this more efficient mechanism for managing the "FS" register. (Other architectures, such as ARM, use unprivileged addressing modes for thread-local variables that do not depend on special constructs, such as the x86 segments.)

A second (albeit smaller) source of overhead is the virtualization of MPI communicators and datatypes, and recording of metadata for MPI sends and receives. Virtualization requires a hash table lookup and locks for thread safety.

The first and larger source of overhead is then eliminated by using the patched Linux kernel, as discussed above. Point-to-point bandwidth benchmarks were run both with and without the patched Linux kernel (Figure 4). A degradation in runtime performance is seen for MANA for small message sizes (less than 1 MB) in the case of a native kernel. However, the figure shows that the patched kernel yields much reduced runtime overhead for MANA. Note that the Linux kernel community is actively reviewing this patch (currently in its third version), and it is likely to be incorporated in future Linux releases.

Finally, we return to GROMACS, since it exhibited a higher runtime overhead (e.g., 2.1 % in the case of 16 ranks) in many cases. We did a similar experiment, running GROMACS with 16 MPI ranks on a single node with the patched kernel. With the patched kernel, the performance degradation was reduced to 0.6 %.

3.4 Checkpoint-restart Overhead

In this section, we evaluate MANA's performance when checkpointing and restarting HPC applications. Figure 6 shows the checkpointing overhead for five different real-world HPC applications running on multiple nodes under MANA. Each run was repeated 5 times, and the mean of five runs is reported. For each run, we use the `fsync` system call to ensure the data is flushed to the Lustre backend storage.

The total checkpointing data written at each checkpoint varies from 5.9 GB (in the case of 64 ranks of GROMACS running over 2 nodes) to 4 TB (in the case of 2048 ranks of HPCG running over 64 nodes). Note that the checkpointing overhead is proportional to the total amount of memory used by the benchmark. This is also reflected in the size of the checkpoint image per MPI rank. While Figure 6 reports the overall checkpoint time, note that there is significant variation in the write times for each MPI rank during a given run. (The time for one rank to write its checkpoint data can be up to 4 times more than that for 90 % of the other ranks.) This phenomenon of stragglers during a parallel write has also been noted by other researchers [2, 37]. Thus, the overall checkpoint time is bottlenecked by the checkpoint time of the slowest rank.

Next, we ask what are the sources of the checkpointing overhead? Does the draining of MPI messages and the two-phase algorithm impose a significant overhead at checkpoint time?

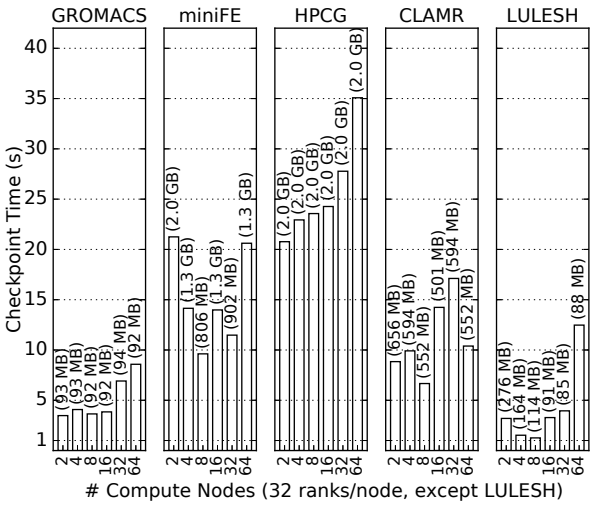


Figure 6: Checkpointing overhead and checkpoint image sizes under MANA for different real-world HPC benchmarks running on multiple nodes. In all cases, except LULESH, 32 MPI ranks were executed on each compute node. For LULESH, the total number of ranks was either 64 (for 2, 4, and 8 nodes), or 512 (for 16, 32, and 64 nodes). Hence, the maximum number of ranks (for 64 nodes) was 2048. The numbers above the bars (in parentheses) indicate the checkpoint image size for each MPI rank.

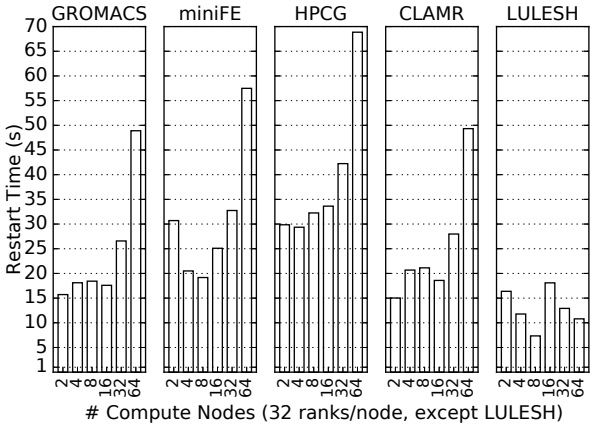


Figure 7: Restart overhead under MANA for different real-world HPC benchmarks running on multiple nodes. In all cases, except LULESH, 32 MPI ranks were executed on each compute node. Ranks/node is as in Figure 6.

Figure 8 shows the contribution of different components to the checkpointing overhead for the case of 64 nodes for the five different benchmarks. In all cases, the communication overhead for handling MPI collectives in the two-phase algorithm of Section 2.5 is found to be less than 1.6 s.

In all cases, the time to drain in-flight MPI messages was less than 0.7 s. The total checkpoint time was dominated by the time to write to the storage system. The next big source of checkpointing

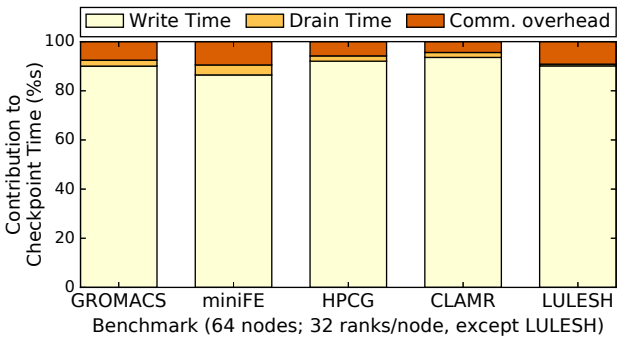


Figure 8: Contribution of different factors to the checkpointing overhead under MANA for different real-world HPC benchmarks running on 64 nodes. Ranks/node is as in Figure 6. The "drain time" is the delay in starting a checkpoint while MPI message in transit are completed. The communication overhead is the time required in the protocol for network communication between the checkpoint coordinator and each rank.

overhead was the communication overhead. The current implementation of the checkpointing protocol in DMTCP uses TCP/IP sockets for communication between the MPI ranks and the centralized DMTCP coordinator. The communication overhead associated with the TCP layer is found to increase with the number of ranks, especially due to metadata in the case of small messages that are exchanged between MPI ranks and the coordinator.

Finally, Figure 7 shows the restart overhead under MANA for the different MPI benchmarks. The restart time varies from less than 10 s to 68 s (for 2048 ranks of HPCG running over 64 nodes). The restart times increase in proportion to the total amount of checkpointing data that is read from the storage. In all the cases, the restart overhead is dominated by the time to read the data from the disk. The time to recreate the MPI opaque identifiers (see Section 2.2) is less than 10 % of the total restart time.

3.5 Transparent Switching of MPI libraries across Checkpoint-restart

This section demonstrates that MANA can transparently switch between different MPI implementations across checkpoint-restart. This is useful for debugging programs (even the MPI library) as it allows a program to switch from a production version of an MPI library to a debug version of the MPI library.

The GROMACS application is launched using the production version of CRAY MPI, and a checkpoint is taken 55 s into the run. The computation is then restarted on top of a custom-compiled debug version of MPICH (for MPICH version 3.3). MPICH was chosen because it is a reference implementation whose simplicity makes it easy to instrument for debugging.

3.6 Transparent Migration across Clusters

Next, we consider cross-cluster migration for purposes of wide-area load balancing either among clusters at a single HPC site or even among multiple HPC sites. This is rarely done, since the two common vehicles for transparent checkpoint (BLCR as the base of

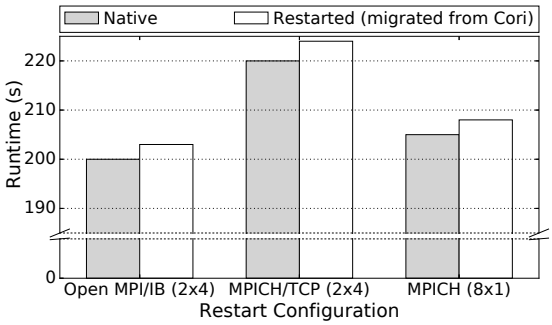


Figure 9: Performance degradation of GROMACS after cross-cluster migration under three different restart configurations. The application was restarted after being checkpointed at the half-way mark on Cori. (Lower is better.)

an MPI-specific checkpoint-restart service; or DMTCP/InfiniBand) both save the MPI library within the checkpoint image and continue to use that same MPI library on the remote cluster after migration. At each site and for each cluster, administrators typically configure and tune a locally recommended MPI implementation for performance. Migrating an MPI application *along with its underlying MPI library* destroys the benefit of this local performance tuning.

This experiment showcases the benefits of MPI-agnostic, network-agnostic support for transparent checkpointing. GROMACS is run under MANA, initially running on Cori with a statically linked Cray MPI library running over the Cray Aries network. GROMACS on Cori is configured to run with 8 ranks over 4 nodes (2 ranks per node). Each GROMACS rank is single-threaded. A checkpoint was then taken exactly half way into the run. The checkpoints were then migrated to a local cluster that uses Open MPI over the InfiniBand network.

The restarted GROMACS under MANA was compared with three other configurations: GROMACS using the local Open MPI, configured to use the local InfiniBand network (8 ranks over 2 nodes); GROMACS/MPICH, configured to use TCP (8 ranks over 2 nodes); and GROMACS/MPICH, running on a single node (8 ranks over 1 node). The network-agnostic nature of MANA allowed the Cori version of GROMACS to be restarted on the local cluster with any of three network options.

We wished to isolate the effects due to MANA from the effects due to different compilers on Cori and the local cluster. In order to accomplish this, the native GROMACS on the local cluster was compiled specially. The Cray compiler of Cori (using Intel's C compiler) was used to generate object files (.o files) on Cori. Those object files were copied to the local cluster. The native GROMACS was then built using the local `mpicc`, but with the (.o files) as input instead of the (.c files). The local `mpicc` linked these files with the local MPI implementation, and the native application was then launched in the traditional way.

Figure 9 shows that GROMACS's performance degrades by less than 1.8% post restart on the local cluster for the three different restart configurations (compared to the corresponding native runs). Also, note that the performance of GROMACS under MANA post restart closely tracks the performance of the native configuration.

4 DISCUSSION AND FUTURE WORK

Next, we discuss both the limitations and some future implications of this work concerning dynamic load balancing.

4.1 Limitations

While the split-process approach for checkpointing and process migration is quite flexible, it does include some limitations inherited by any approach based on transparent checkpointing. Naturally, when restarting on a different architecture, the CPU instruction set must be compatible. In particular, on the x86 architecture, the MPI application code must be compiled to the oldest x86 sub-architecture among those remote clusters where one might consider restarting a checkpoint image. (However, the MPI libraries themselves may be fully optimized for the local architecture, since restarting on a remote cluster implies using a new lower half.)

Similarly, while MPI implies a standard API, any local extensions to MPI must be avoided. The application *binary* interface (ABI) used by the compiled MPI application must either be compatible or else a "shim" layer of code must be inserted in the wrapper functions for calling from the upper half to the lower half.

And of course, the use of a checkpoint coordinator implies coordinated checkpointing. If a single MPI rank crashes, MANA must restore the entire MPI computation from an earlier checkpoint.

4.2 Future Work

MPI version 3 has added nonblocking collective communication calls (e.g., MPI_Igather). In future work, we propose to extend the two-phase algorithm for collective communication of Section 2.5 to the nonblocking case. The approach to be explored would be to employ a first phase that uses a nonblocking trivial barrier (MPI_Ibarrier), and to then convert the actual asynchronous collective call to a synchronous collective call (e.g., MPI_Gather to MPI_Igather) for the second phase. Nonblocking variations of collective communication calls are typically used as performance optimizations in an MPI application. If an MPI rank reaches the collective communication early, then instead of blocking, it can continue with an alternate compute task while occasionally testing (via MPI_Test/MPI_Wait) to see if the other ranks have all reached the barrier. In the two-phase analog, a wrapper around the nonblocking collective communication causes MPI_Ibarrier to be invoked. When the ranks have all reached the nonblocking trivial barrier and the MPI_Test/MPI_Wait calls of the original MPI application reports completion of the MPI_Ibarrier call of phase 1, then this implies that the ranks are all ready to enter the actual collective call of phase 2. A wrapper around MPI_Test/MPI_Wait can then invoke the actual collective call of phase 2.

The split-process approach of MANA opens up some important new features in managing long-running MPI applications. An immediately obvious feature is the possibility of switching *in the middle of a long run* to a customized MPI implementation. Hence, one can dynamically substitute a customized MPI for performance analysis (e.g., using PMPI for profiling or tracing); or using a specially compiled "debug" version of MPI to analyze a particular but occurring in the MPI library in the middle of a long run.

This work also helps support many tools and proposals for optimizing MPI applications. For example, a trace analyzer is sometimes

used to discover communication hotspots and opportunities for better load balancing. Such results are then fed back by re-configuring the binding of MPI ranks to specific hosts in order to better fit the underlying interconnect topology.

MANA can enable new approaches to dynamically load balance across clusters and also to re-bind MPI ranks in the middle of a long run to create new configurations of rank-to-host bindings (new topology mappings). Currently, such bindings are chosen statically and used for the entire lifetime of the MPI application run. This added flexibility allows system managers to burst current long-running applications into the Cloud during periods of heavy usage or when the the MPI application enters a new phase for which a different rank-to-host binding is optimal.

Finally, MANA can enable a new class of very long-running MPI applications — ones which may outlive the lifespan of the original MPI Implementation, cluster, or even the network interconnect. Such temporally complex computations might be discarded as infeasible today without the ability to migrate MPI implementations or clusters.

5 RELATED WORK

Hursey et al. [22] developed a semi-network-agnostic checkpoint service for Open-MPI. It applied an "MPI Message" abstraction to a Chandy/Lamport algorithm [11], greatly reducing the complexity to support checkpoint/restart for many multiple network interconnects. However, it also highlighted the weakness of implementing transparent checkpointing within the MPI library, since porting to an additional MPI implementation would likely require as much software development as for the first MPI implementation. Additionally, its dependence on BLCR imposed a large overhead cost, as it lacks support for SysV shared memory.

Separate proxy processes for high- and low-level operations have been proposed both by CRUM (for CUDA) and McKernel (for the Linux kernel). CRUM [16] showed that by running a non-reentrant library in a separate process, one can work around the problem of a library "polluting" the address space of the application process — i.e., creating and leaving side-effects in the application process's address space. This decomposition of a single application process into two processes, however, forces the transfer of data between two processes via RPC, which can cause a large overhead.

McKernel [17] runs a "lightweight" kernel along with a full-fledged Linux kernel. The HPC application runs on the lightweight kernel, which implements time-critical system calls. The rest of the functionality is offloaded to a proxy process running on the Linux kernel. The proxy process is mapped in the address space of the main application, similar to MANA's concept of a lower half, to minimize the overhead of "call forwarding" (argument marshalling/unmarshalling).

In general, a proxy process approach is problematic for MPI, since it can lead to additional jitter as the operating system tries to schedule the extra proxy process alongside the application process. The jitter harms performance since the MPI computation is constrained to complete no faster than its slowest MPI rank.

Process-in-process [21] has in common with MANA that both approaches load multiple programs into a single address space. However, the goal of process-in-process was intra-node communication optimization, and not checkpoint-restart. Process-in-process loads *all* MPI ranks co-located on the same node as separate threads within a single process, but in different logical "namespaces", in the sense of the `dlmopen` namespaces in Linux. It would be difficult to adapt process-in-process for use in checkpoint-restart since that approach implies a single "ld.so" run-time linker library that managed all of the MPI ranks. In particular, difficulties occur when restarting with fresh MPI libraries while "ld.so" retains pointers to destructor functions in the pre-checkpoint MPI libraries.

In the special regime of application-specific checkpointing for bulk synchronous MPI applications, Sultana et al. [33] supported checkpointing by separately saving and restoring MPI state (MPI identifiers such as communicators, and so on). This is combined with application-specific code to save the application state. Thus, when a live process fails, it is restored using these two components, without the need restart the overall MPI job.

SCR [32], and FTI [3] are other application-specific checkpointing techniques. An application developer declares memory regions they'd like to checkpoint and checkpointing can only be done at specific points in the program determined by the application developer. Combining these techniques with transparent checkpointing is outside the scope of this work, though it is an interesting avenue for further inquiry.

In general, application-specific and transparent checkpointing each have their merits. Both application-specific and transparent checkpointing are used in practice.

At the high end of HPC, application-specific checkpointing is preferred since the labor for supporting this is small compared to the labor already invested in supporting an extreme HPC application.

At the low and medium end of HPC, developers prefer transparent checkpointing because the development effort for the software is more moderate, and the labor overhead of a specialized application-specific checkpointing solution would then be significant. System architectures based on burst buffers (e.g., Cray's DataWarp [19]) can be used to reduce the checkpointing overhead for both application-specific and transparent checkpointing.

6 CONCLUSION

This work presents an MPI-Agnostic, Network-Agnostic transparent checkpointing methodology for MPI (MANA), based on a *split-process* mechanism. The runtime overhead is typically less than 2%, even in spite of the overhead incurred by the current Linux kernel when the "FS" register is modified each time control passes between upper and lower half. Further, Section 3.3 shows that a commit (patch) to fix this by the Linux kernel developers is under review and that this commit reduces the runtime overhead of GROMACS from 2.1% to 0.6% using the patched kernel. Similar reductions to about 0.6% runtime overhead are expected in the general case.

An additional major novelty is the demonstration of practical, efficient migration between clusters at different sites using different networks and different configurations of CPU cores per node. This was considered impractical in the past because a checkpoint image from one cluster will not be tuned for optimal performance on the second cluster. Section 3.6 demonstrates that this is now

feasible, and that the migration of a GROMACS job with 8 MPI ranks experiences an average runtime overhead of less than 1.8% as compared to the native GROMACS application (without MANA) on the remote cluster. As before, even this overhead of 1.8% is likely to be reduced to about 0.6% in the future, based on the results of Section 3.3 with a patched Linux kernel.

ACKNOWLEDGMENT

We thank Zhengji Zhao and Rebecca-Hartman Baker from NERSC for the resources and feedback on an earlier version of the software. We also thank Twinkle Jain for discussions and insights into an earlier version of this work. We also benefited from valuable comments and feedback from reviewers. We are also grateful for constructive feedback from Jay Lofstead during the shepherding process.

REFERENCES

[1] Jason Ansel, Kapil Arya, and Gene Cooperman. 2009. DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop. In *23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS'09)*. IEEE, 1–12.

[2] Kapil Arya, Rohan Garg, Artem Y. Polyakov, and Gene Cooperman. 2016. Design and Implementation for Checkpointing of Distributed Resources using Process-level Virtualization. In *IEEE Int. Conf. on Cluster Computing (CLUSTER'16)*. IEEE Press, 402–412.

[3] Leonardo Bautista-Gomez, Seiji Tsuboi, Dimitri Komatitsch, Franck Cappello, Naoya Maruyama, and Satoshi Matsuoka. 2011. FTI: High Performance Fault Tolerance Interface for Hybrid Systems. In *SC'11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.

[4] H.J.C. Berendsen, D. van der Spoel, and R. van Drunen. 1995. GROMACS: A Message-passing Parallel Molecular Dynamics Implementation. *Computer Physics Communications* 91, 1 (1995), 43 – 56. https://doi.org/10.1016/0010-4655(95)00042-E

[5] BLCR site 2019. Berkeley Lab Checkpoint/Restart for Linux (BLCR) Downloads. http://crd.lbl.gov/departments/computer-science/CLaSS/research/BLCR/berkeley-lab-checkpoint-restart-for-linux-blcr-downloads/. (2019). [Online; accessed Jan., 2019].

[6] BLCR site 2019. BLCR Admin Guide — version 0.8.5. https://upc-bugs.lbl.gov/blcr/doc/html/BLCR_Admin_Guide.html. (2019). [Online; accessed Jan., 2019].

[7] A. Bouteiller, T. Herault, G. Krawezik, P. Lemarinier, and F. Cappello. 2006. MPICH-V Project: A Multiprotocol Automatic Fault Tolerant MPI. *International Journal of High Performance Computing Applications* 20 (2006), 319–333. (web site at http://mpich-v.lri.fr/, accessed Jan., 2019).

[8] Jiajun Cao. 2017. *Transparent Checkpointing over RDMA-based Networks*. Ph.D. Dissertation. Northeastern University. https://doi.org/10.17760/D20290419

[9] Jiajun Cao, Kapil Arya, Rohan Garg, Shawn Matott, Dhabaleswar K. Panda, Hari Subramoni, Jéôme Vienne, and Gene Cooperman. 2016. System-level Scalable Checkpoint-Restart for Petascale Computing. In *22nd IEEE Int. Conf. on Parallel and Distributed Systems (ICPADS'16)*. IEEE Press, 932–941. also, technical report available as: arXiv preprint arXiv:1607.07995.

[10] Jiajun Cao, Gregory Kerr, Kapil Arya, and Gene Cooperman. 2014. Transparent Checkpoint-Restart over InfiniBand. In *ACM Symposium on High Performance Parallel and and Distributed Computing (HPDC'14)*. ACM Press, 12.

[11] K. Mani Chandy and Leslie Lamport. 1985. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. Comput. Syst.* 3, 1 (Feb. 1985), 63–75. https://doi.org/10.1145/214451.214456

[12] CLAMR Source 2019. CLAMR Source Code. https://github.com/lanl/CLAMR. (2019). [Online; accessed Jan., 2019].

[13] Cori 2019. Cori Supercomputer at NERSC. http://www.nersc.gov/users/computational-systems/cori/. (2019). [Online; accessed Jan., 2019].

[14] Jack Dongarra, Michael A Heroux, and Piotr Luszczek. 2016. A New Metric for Ranking High-performance Computing Systems. *National Science Review* 3, 1 (2016), 30–35.

[15] Qi Gao, Weikuan Yu, Wei Huang, and Dhabaleswar K. Panda. 2006. Application-Transparent Checkpoint/Restart for MPI Programs over InfiniBand. In *PP '06: Proceedings of the 2006 Int. Conf. on Parallel Processing (ICPP'06)*. IEEE Computer Society, Washington, DC, USA, 471–478. https://doi.org/10.1109/ICPP.2006.26

[16] Rohan Garg, Apoorve Mohan, Michael Sullivan, and Gene Cooperman. 2018. CRUM: Checkpoint-Restart Support for CUDA's Unified Memory. In *Proceedings of the 20th Int. Conf. on Cluster Computing (CLUSTER'18)*. IEEE.

[17] Balazs Gerofi, Masamichi Takagi, Atsushi Hori, Gou Nakamura, Tomoki Shirasawa, and Yutaka Ishikawa. 2016. On the Scalability, Performance Isolation and Device Driver Transparency of the IHK/McKernel Hybrid Lightweight Kernel. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*. IEEE, 1041–1050.

[18] R. Gupta, P. Beckman, B.H. Park, E. Lusk, P. Hargrove, A. Geist, D. K. Panda, A. Lumsdaine, and J. Dongarra. 2009. CIFTS: A Coordinated Infrastructure for Fault-Tolerant Systems. In *38th Int. Conf. on Parallel Processing (ICPP'09)*. (web site at https://wiki.mcs.anl.gov/cifts/index.php/CIFTS, accessed Jan., 2019).

[19] Dave Henseler, Benjamin Landsteiner, Doug Petesch, Cornell Wright, and Nicholas J Wright. 2016. Architecture and Design of Cray DataWarp. *Cray User Group (CUG)* (2016).

[20] M Heroux and S Hammond. 2019. MiniFE: Finite Element Solver. https://tinyurl.com/y7hslf65. (2019). [Online; accessed Jan 2019].

[21] Atsushi Hori, Min Si, Balazs Gerofi, Masamichi Takagi, Jai Dayal, Pavan Balaji, and Yutaka Ishikawa. 2018. Process-in-process: techniques for practical address-space sharing. In *Proc. of the 27th Int. Symposium on High-Performance Parallel and Distributed Computing (HPDC'18)*. ACM, 131–143.

[22] Joshua Hursey, Timothy I. Mattox, and Andrew Lumsdaine. 2009. Interconnect Agnostic Checkpoint/Restart in OpenMPI. In *Proc. of the 18th ACM Int. Symp. on High performance Distributed Computing (HPDC'09)*. ACM, 49–58.

[23] Joshua Hursey, Jeffrey M. Squyres, Timothy I. Mattox, and Andrew Lumsdaine. 2007. The Design and Implementation of Checkpoint/Restart Process Fault Tolerance for Open MPI. In *Proc. of 21^{st} IEEE International Parallel and Distributed Processing Symposium (IPDPS'07) / 12^{th} IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems*. IEEE Computer Society.

[24] Ian Karlin, Jeff Keasler, and Rob Neely. 2013. *LULESH 2.0 Updates and Changes*. Technical Report LLNL-TR-641973. 1–9 pages.

[25] Leslie Lamport. 1999. Specifying Concurrent Systems with TLA$^+$. *NATO ASI SERIES F COMPUTER AND SYSTEMS SCIENCES* 173 (1999), 183–250.

[26] Libfabric site 2019. Libfabric. https://ofiwg.github.io/libfabric/. (2019). [Online; accessed Jan., 2019].

[27] Linux Weekly News (LWN) 2018. x86: Enable FSGSBASE instructions. https://lwn.net/Articles/769355/. (2018). [Online; accessed Jan., 2019].

[28] MANA for MPI: Github source code 2019. MANA for MPI: Github source code. https://github.com/mpickpt/mana. (2019). [Online; accessed Apr., 2019].

[29] D. Nicholaeff, N. Davis, D. Trujillo, and R. W. Robey. 2012. Cell-Based Adaptive Mesh Refinement Implemented with General Purpose Graphics Processing Units. (2012).

[30] Open MPI site 2019. FAQ: Fault tolerance for parallel MPI jobs. https://www.open-mpi.org/faq/?category=ft#cr-support. (2019). [Online; accessed Jan., 2019].

[31] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, Vishal Sahay, Andrew Lumsdaine, Jason Duell, Paul Hargrove, and Eric Roman. 2005. The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing. *Int. Journal of High Performance Computing Applications* 19, 4 (2005), 479–493.

[32] SCR: Scalable Checkpoint/Restart for MPI 2019. SCR: Scalable Checkpoint/Restart for MPI. https://computation.llnl.gov/projects/scalable-checkpoint-restart-for-mpi. (2019). [Online; accessed Apr., 2019].

[33] Nawrin Sultana, Anthony Skjellum, Ignacio Laguna, Matthew Shane Farmer, Kathryn Mohror, and Murali Emani. 2018. MPI Stages: Checkpointing MPI State for Bulk Synchronous Applications. In *Proceedings of the 25th European MPI Users' Group Meeting (EuroMPI'18)*. ACM, New York, NY, USA, Article 13, 11 pages. https://doi.org/10.1145/3236367.3236385

[34] Taichiro Suzuki, Akira Nukada, and Satoshi Matsuoka. 2016. Transparent Checkpoint and Restart Technology for CUDA Applications. GPU Technology Conference (GTC'16). (2016). http://on-demand.gputechconf.com/gtc/2016/presentation/s6429-akira-nukada-transparen-checkpoint-restart-technology-cuda-applications.pdf accessed Jan., 2019.

[35] Hiroyuki Takizawa, Kentaro Koyama, Katsuto Sato, Kazuhiko Komatsu, and Hiroaki Kobayashi. 2011. CheCL: Transparent Checkpointing and Process Migration of OpenCL Applications. In *2011 IEEE International Parallel and Distributed Processing Symposium (IPDPS'11)*. IEEE, 864–876.

[36] Top500 2018. Top500 Supercomputers. https://www.top500.org/list/2018/11/?page=1. (2018). [Online; accessed Jan., 2019].

[37] Bing Xie, Jeffrey Chase, David Dillow, Oleg Drokin, Scott Klasky, Sarp Oral, and Norbert Podhorszki. 2012. Characterizing Output Bottlenecks in a Supercomputer. In *Proc. of the Int. Conf. on High Performance Computing, Networking, Storage and Analysis (SC'12)*. IEEE Computer Society Press, 11.

[38] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. 1999. Model checking TLA+ specifications. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*. Springer, 54–66.

[39] Victor C. Zandy, Barton P. Miller, and Miron Livny. 1999. Process Hijacking. In *Proc. of Int. Symp. on High-Performance Parallel and Distributed Computing (HPDC'99)*. IEEE Press, 177–184.