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

Anonymous Author(s)

ABSTRACT

Transparent 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 a single solution based on a single code base, MANA (MPI-Agnostic Network-Agnostic transparent checkpointing), which works for all such 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 approach is based on split processes, which enable two separate programs to co-exist within a single process with a single address space. In MANA, the two programs are the MPI application code and an MPI proxy server, denoted the upper-half program and the lower-half program, respectively. This work also overcomes the limitations of the two previous most widely used approaches to transparent checkpointing, BLCR and DMTCP/InfiniBand, which require separate code bases for 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 naively on that same remote cluster.

ACM Reference Format:

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 now used less and less. These well-known checkpointing packages that are now either in declining usage or abandoned include: the checkpoint-restart service of Open MPI [22], the checkpoint-restart service of MVAPICH2 [14], DMTCP for MPI [1], and MPICH-V [7], as well as a fault-tolerant BLCR-based "backplane", CIFTS [17]. We argue that this crisis is caused by the difficulty of maintaining existing approaches to transparent or system-level checkpointing. In particular, the need for the HPC community to support checkpoint-restart services for any of $m$ popular MPI implementations over $n$ different networks creates a burden to maintain $m \times n$ different code bases.

We propose MPI-Agnostic Network-Agnostic transparent checkpointing (MANA), a single code base to support all of the $m \times n$ combinations of MPI implementation and underlying network. The new approach, based on split processes, is fully transparent to the underlying MPI, network, and even the particular libc library or underlying Linux kernel. MANA is free and open-source software. (Transparent checkpointing supports standard system-level checkpointing, but it can alternatively be customized in an application-specific manner.)

Currently, the two most widely used approaches to transparent checkpointing today are BLCR [18] (used by several MPI implementations) and DMTCP/InfiniBand [18]. Both packages support an underlying checkpoint for single processes in isolation, and so both packages can be used for the foundation of MANA. However, in the case of BLCR, there is still an $m \times n$ explosion: each of $m$ MPI implementations must implement a custom checkpoint-restart service for each of $n$ interconnect network that are to be supported. In the case of DMTCP/InfiniBand, it is also MPI-agnostic in that it transparently saves and restores the underlying MPI libraries along with many other constructs. However, DMTCP also requires separate plugins [2] to save and restore sockets and InfiniBand/OFED [8, 10]. Thus, DMTCP is not network-agnostic, and potentially would require $n$ plugins for each of $n$ network interconnects.

Next, we review three case histories to demonstrate the declining usage of transparent checkpointing. We first consider Open MPI. The developers had created a novel and elegant checkpoint-restart service that made MPI applications appear to be network-agnostic [21] (checkpointing under network A and restarting under network B). But the $m \times n$ penalty is still present since the maintainers of this single MPI must maintain code for each of the $n$ possible networks supported by Open MPI. This burden caused the maintainers to drop official checkpointing support as of Open MPI version 1.7 (introduced in 2013). As of this writing in 2019, the Open MPI FAQ continues to say: "Note: The checkpoint/restart support was last released as part of the v1.6 series. This feature is looking for a maintainer. Interested parties should inquire on the developers mailing list." [28].

The second case history concerns BLCR. The MPICH reference implementation adopted BLCR for checkpointing, along with several other MPI implementations. BLCR is based on a kernel module that checkpoints the local MPI rank. In practice, the use of BLCR is severely limited because it does not support the System V shared memory construct that is widely used for intra-node communication among MPI ranks. As of this writing, BLCR 0.8.5 (appearing in 2013) was the last officially supported version [4], and formal
testing of the BLCR kernel module appear to have stopped with Linux 3.7 (Dec., 2012) [5]. Here again, we argue that BLCR declined not due to any fault with BLCR, but due to the difficulty of maintaining the \( m \times n \) checkpoint-restart services based on top of BLCR.

The third case history concerns DMTCP. As discussed above, DMTCP/InfiniBand is \textit{MPI-agnostic}, but is not \textit{network-agnostic} and so it requires \( n \) plugins for each of \( n \) networks. While it supports InfiniBand [10], it has only partial support for Intel Omni-Path [8, Chapter 6], and currently has no support for Cray GNI Aries network, the Mellanox extensions to InfiniBand (UCX and FCA), the libfabric API [25], and so on.

The new \textit{split processes} approach provides a single code base in contrast to the \( m \times n \) code bases discussed earlier. With split processes, a single process contains two independent programs in its memory address space. The two programs are an MPI proxy application (denoted the lower-half program) and the the original MPI application code (denoted the upper-half program). MANA tracks which memory regions belong to the upper half and which belong to the lower half. Only the upper-half memory regions are saved at checkpoint time. As stated earlier, MANA is also fully transparent to the MPI, network, libc library and Linux kernel.

At restart time, MANA initializes a new MPI proxy along with a new underlying interconnect network in the lower half of a process. The checkpointed MPI application code and data is then copied in an restored into the upper half from the checkpoint image file. Because a new MPI library is initialized at the time of restart, MANA provides excellent load-balancing support in which the new MPI library will be initialized on restart for the correct number of CPU cores per node, the optimal topology as MPI ranks from the same node may be split among distinct nodes, or vice versa, re-optimization of the rank-to-host bindings as optimized by any MPI topology declarations in the MPI application, and so on. (See Section 4 for further discussion.)

MANA maintains low overhead at runtime by taking advantage of split processes to directly make library calls from the upper half to an MPI routine in the lower half. Prior to this work, a related, proxy-based approach had been used for checkpointing in the framework of checkpointing general applications [36] and in the framework of CUDA applications [15, 31, 32]. However, such approaches incur a significant overhead due to both context switching and the need to copy buffers (e.g., the MPI\_send buffers) from an MPI application process to an MPI proxy.

The implementation of MANA employs a centralized checkpoint coordinator that creates a consistent snapshot in which each MPI rank locally creates a checkpoint image. At very large scale, for the sake of efficiency, one can consider replacing the abstraction of a centralized coordinator by a set of coordinator processes in the form of a broadcast tree.

Finally, there is a latency between when the coordinator invokes a checkpoint and when the MPI ranks actually being executing the checkpoint. However, this latency is bounded by a time proportional to the time to send an MPI message and the time to execute a collective communication call in the underlying MPI implementation. In particular, see Sections 2.4 and 2.5 for the algorithm and a formal proof of the latency in the case of collective communication.

\textbf{FINAL VERSION:} MANA uses version 3.0 of DMTCP [1] for checkpointing. DMTCP has been shown to checkpoint MPI-based HPCG over 32,752 CPU cores (38 TB) in 11 minutes, and MPI-based NAMD over 16,368 CPU cores in 2.6 minutes [9].

Next in this work, Section 2 describes the subtle design issues in fitting the split process concept to an MPI implementation. In particular, the question of how to checkpoint when there are ongoing MPI point-to-point or collective communications is discussed there. Section 3 then presents the experimental evaluation. Section 4 discusses current and future work that is opened up by these new ideas. Section 5 discusses the related work. Finally, Section 6 presents the conclusion.

\section{MANA: DESIGN AND IMPLEMENTATION}

Multiple aspects of the design of MANA are covered in this section. Section 2.1 discusses the design for supporting split processes themselves. Section 2.2 discusses the need to save and restore persistent MPI opaque objects, such as communicators, groups and topologies. Section 2.3 presents Algorithm 1 for draining any point-to-point MPI messages in transit prior to initiating a checkpoint. Sections 2.4 and 2.5 present a two-phase algorithm (Algorithm 3) for completing any MPI collective communication calls in progress prior to initiating a checkpoint. And finally, Section 2.6 presents details of the overall implementation of MANA.

\subsection{Upper and Lower Half: Checkpointing with an Ephemeral MPI Library}

In this section, we define the \textit{lower half} of a split process as the memory associated with the MPI libraries and all of their dependencies, including any network libraries. The \textit{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 definitions are by analogy with the upper and lower half of a device driver in an operating system kernel. Nevertheless, the process continues to have only a single thread, a single stack (in the upper half), and a single flow of control. (More precisely, the process continues to include each thread of the original MPI application, each with its own stack. Section 2.6 describes an additional “helper thread”, but that thread is active only during a checkpoint or a restart from a checkpoint image file.)

Note that libc and other system libraries are a special case here. There will be one copy of libc appearing in the lower half as a dependency of the MPI libraries, and there will be a second copy of libc appearing in the upper half as an independent dependency of the MPI application code.

This split-process approach allows MANA to balance two conflicting objectives: a shared address space; and isolation of upper and lower half. The isolation allows MANA to omit the lower half memory (and omit 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 the standard C/Fortran calling conventions, including call by reference: passing pointer parameters from the upper half to the lower half, where those pointers continue to refer to data in the upper half.

The \textit{isolation} is needed so that at restart time, the old lower half can be omitted from the checkpoint image, and at the time of restart,
a small “bootstrap” MPI program with new MPI libraries. The boot-
strap 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 correct MPI rank. Control is then transferred back to the
upper half MPI application, and the original stack in the lower half
is never used again.

The shared address space is needed for efficiency. Earlier work
in the checkpointing literature had used a separate proxy process
instead of a lower half within the same thread. This approach was
explored in [15, Section IV.B] and in [32, Section IV.A], where the
former work reported a typical runtime 6% overhead for real-world
CUDA applications, and the latter reported runtime overheads of
20% or higher in in some of the OpenCL examples from the
NVIDIA SDK 3.0. Section 3 reports typical overheads under 2%
for the current work under older Linux kernels.

TODO: Check that the overheads in our experiments are re-
ported correctly in the above paragraph, and then delete this TODO
comment.

Finally, note that the ability to discard the lower half when cre-
ating a checkpoint image file greatly simplifies the task of check-
pointing. When the lower half is discarded, the MPI application in
the upper half appears as an isolated process with no inter-process
communication. Hence, the upper half does not involve network
communication or shared memory between processes. So, a single-
process checkpointing package suffices to create this checkpoint
image.

TODO: This next paragraph can be commented out if we’re short of space.

A minor inconvenience of the current approach is that a call
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 the upper and lower
half of a process, the kernel will 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. In the upper half, we interpose on calls to sbrk() in
the upper half libc, and then insert calls to mmap() to extend the
heap of the upper half.

TODO: I PROPOSE THAT WE PLACE THIS NEXT PARAGRAPH
IN COMMENTS. IT COULD ONLY RUNTIME AS A RED FLAG TO A
MALICIOUS REFEREE. IF DESIRED, WE CAN BRING IT BACK IN
AN ACCEPTED VERSION OF THE PAPER.

An alternative approach of using dlopen/dlclose was considered
earlier and then discarded. See Section 5 for a related discussion of
why a process-in-process approach using dlclose is not feasible in
this context.

Finally, MANA employs coordinated checkpointing, and a check-
point 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 record-
replay strategy by which MPI calls with persistent effects (such as
creation of these opaque objects) are recorded during runtime and
replayed on restart.

TODO: By removing some of the detailed text, I’ve also removed
the Figure ?? (“Split process approach used by MANA”). I propose
that we continue to remove the figure, since we’re short of space.
— Gene

2.2 Checkpointing MPI Communicators,
Groups, and Topologies

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

MANA interposes on all the calls that refer to these opaque iden-
tifiers, 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 recre-
ates the MPI communicator state by replaying the MPI calls us-
ing a new MPI library. So, while the new MPI library may pro-
vide different opaque identifiers, the runtime virtualization allows
the application to continue to run 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

The next challenge in checkpointing MPI computations is the prob-
lem of saving the network state.

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 that were already sent
but not yet received at the time of quiescing all the processes must
also be saved as part of the checkpoint.

Figure 1 shows the state of a system with two MPI ranks, $R_1$
and $R_2$. Rank $R_1$ sends a message, $m_1$, to rank $R_2$ and then,
receives a checkpoint request from the user. On the other hand, rank $R_2$
receives the checkpoint request before it can receive the message,
$m_1$. If the state of the network channel, with the message, $m_1$, is not
saved, this can lead to a deadlock on restart. This is because when
rank $R_1$ is restored from the same point on restart, it remembers
that it had sent the message in the “past” and it is not going to send
the message again; and since rank $R_2$ had not received the message
at checkpoint time, it is going to get stuck when it resumes to do
the receive on restart.

Thus, MANA uses an MPI message draining algorithm to save
all in-flight MPI messages to capture the state of the network. The
main idea is to publish local information (send/receive counts, and
pending sends) to a centralized key-value database, and iterate until
all unreceived messages have been received and buffered. A similar
approach was used in [8, Chapter 5.4.2]. Algorithm 1 describes the
algorithm in more detail.

At runtime, MANA keeps track of message send and receive
requests to MPI. In particular, it tracks the blocking, synchronous,
and asynchronous send and receive requests. Note that the only
information that is tracked is the number of such requests and
Figure 1: Checkpointing MPI point-to-point communication. (see Section 2.3)

Algorithm 1: Algorithm for MPI message draining.

```
upon event Checkpoint request do
  for all R ∈ MPI Ranks do
    pendingSends ← GetLocalPendingSends()
    PublishLocalSendAndRecvCounts(pendingSends)
    GlobalBarrier()
    <totalSends, totalRecvs> ← GetGlobalCounts()
    while totalSends > totalRecvs do
      DrainMessages(pendingSends)
      GlobalBarrier()
      pendingSends ← GetLocalPendingSends()
      PublishLocalSendAndRecvCounts(pendingSends)
      <totalSends, totalRecvs> ← GetGlobalCounts()
  end while
end for

function GetLocalPendingSends
  pendingSends ← ∅
  for all s ∈ MPI Asynchronous Send Requests do
    if MPI_Test(s) = Success then
      pendingSends ← pendingSends ∪ {s}
    end if
  end for
  return pendingSends
end function

procedure DrainMessages(pendingSends)
  for all s ∈ pendingSends do
    if MPI_Iprobe(s) = Success then
      MPI_Recv(s, localBuffer)
      UpdateLocalRecvCount()
    end if
  end for
end procedure
```

Some additional metadata (like the communicator and datatype) of the message contents are not tracked. This allows MANA to compute the global set of pending sends at each rank at any given point in time.

At checkpoint time, first, each rank publishes its local counts and set of pending (unreceived) sends to a centralized key-value database. Second, each rank receives the information for all the ranks through the centralized database. Third, each rank locally probes and tries to receive the unreceived sent messages. If the receive is successful at a rank (meaning that some rank had sent it a message prior to the checkpointing request arrived), the rank updates its local receive count. Finally, each rank loops through the same set of these three steps, until the global send count becomes less than or equal to the global receive count and there are no more unreceived sent messages. (Note that the receive count can be higher than the send count, since a rank might post receive requests even before any message has been sent.)

After resuming from a checkpoint, any receive requests by the application are matched against the receive requests that were completed during the execution of Algorithm 1 (at checkpoint time) and returned locally.

**Theorem 1.** Algorithm 1 finishes without deadlocks, assuming a network with reliable messaging.

**Proof.** If the total receive count is higher than the total send count, and there are no pending sends, then there are no in-flight messages to drain.

None of the MPI calls used in draining of messages can block indefinitely (assuming a network with reliable messaging). Also, the call to MPI_RECV() in the DrainMessages() function must also not block, since the only reason it was executed was because the earlier MPI_Iprobe() call was successful. The MPI standard guarantees that the receive will not block if the probe was successful.

Since at each iteration of the main loop, the total receive count can only increase, eventually, the control must break out of the loop.

### 2.4 Checkpointing MPI Collectives: Overview

The next challenge in checkpointing of MPI applications is about handling the MPI barriers and collective communication.

MPI collective communication primitives involve communication amongst all or a program-defined subset of MPI ranks (as specified by the MPI communicator argument to the collective communication call). 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.

Three subtle challenges exist in taking a consistent snapshot during a collective communication. Recall that MANA employs a centralized checkpoint coordinator process (for synchronous checkpointing). The checkpoint coordinator communicates with the MPI ranks through a protocol that will guarantee that no rank is inside an MPI collective communication call at the time when the coordinator requests a checkpoint.
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, and rank B will see the corresponding request to checkpoint 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 would 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. In order 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 checkpoint latency): Given the previous solution for consistency, there can still be long delays before a checkpoint request can be sent. It may happen that rank A has already entered the barrier, and rank B will require several more hours to finish its task before entering the barrier. Hence, the two-pass protocol may create unacceptable delays before a checkpoint can be taken. Algorithm 3 addresses this by introducing an additional, trivial barrier: a call to MPI_Barrier() prior to the original 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). The “trivial” barrier call produces no side effects on the MPI rank, and so it can be safely interrupted during checkpoint and the call can even be restarted at the time of restarting the MPI rank. This works because the split process architecture of MANA means that only the upper half of an MPI rank (process) is saved during checkpoint, and so there will be no inconsistent state associated with the trivial barrier call in the lower half.

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 are exiting from one of the collective calls, it may happen that there are MPI ranks associated with an independent collective call that were formerly in the MPI trivial barrier (phase 1) and have now entered the actual collective call (phase 2). To solve this, as will be seen in Algorithm 3, after an intend-to-checkpoint message, no ranks will be allowed to enter phase 2, the actual collective call, and extra iterations will be inserted into the request-acknowledge protocol between coordinator and MPI rank.

2.5 Checkpointing MPI Collectives: Detailed Algorithm

Next, we present a single algorithm for checkpointing MPI collectives that contains the elements described in Section 2.4: a multiiteration protocol; and a two-phase algorithm incorporating an additional call to a trivial barrier before the main collective communication call. From the viewpoint of the MPI application, any call to an MPI collective communication function is interposed on by a wrapper function, as shown in Algorithm 2.

**Algorithm 2 Two-Phase collective communication wrapper.** (This wrapper function interposes on all calls of MPI application to the corresponding MPI collective communication function.)

```
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 the trivial barrier is an extra call to MPI_Barrier() prior to the collective call. The cost of the extra or trivial barrier is dominated by the collective communication call, and hence it is usually negligible.

The key to this algorithm is to ensure the following extended statement of the invariant in the previous section:

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

We formalize this with the following theorem, which guarantees that the protocol of Algorithm 3 satisfies this invariant.

**Theorem 2.** Under Algorithm 3, 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.

**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 2 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 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, note that 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 there are 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. □

We now continue with the proof of the main theorem (Theo-
rem 2), which was deferred earlier.

Proof. (Proof of Theorem 2 for Algorithm 3.) 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 3, 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 (show-
ing 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 3. Under Algorithm 3, 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 the usual
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 check-
point, 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-cp` 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 begin to 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 and will not enter the call.

**Implementation of Algorithm 3:** 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 incoming 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 to stable storage the upper-half memory regions. 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.

2.6 Implementation and Verification with TLA+/PlusCal

The MANA prototype was implemented by extending DMTCP [1] and by developing a DMTCP plugin [2]. We used DMTCP version 3.0 for developing the prototype. 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.

However, 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 BLCR when it is required to execute the checkpoint. In particular, all sockets and other network communication objects are inside the lower half, and so any single-process checkpointing package suffices for this work.

**TODO:** OPTIONAL: We could mention number of lines of code. About 6,000 lines of code in plugin, of which about 1,000 are for wrappers. We would exclude any test suites. But probably we don’t need this.

**TODO:** OPTIONAL: Discuss use of kernel-loader, fs register. But probably we don’t need this.

To gain further confidence in our implementation for handling collective communication (Section 2.5), we developed a model for the protocol in TLA+ [24] and then used the PlusCal model checker of TLA+ based on TLC [35] to verify Algorithm 3. 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.

**TODO:** We need to state if we’re using dynamic or static linking in these MPI cases. If we’re worried about raising a red flag for the referee (e.g., statically linked MPI library and dynamically linked MPI application or something), then let’s add the information in a “finalVersion” macro in Latex.

3 EXPERIMENTAL EVALUATION

MANA’s evaluation is driven by a prototype implementation, which was used to run and checkpoint real-world HPC applications.

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 for transparent checkpointing of MPI applications under MANA?

**Q3:** Can MANA allow transparent switching between MPI implementation across checkpoint-restart for MPI applications for purposes 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 [12] at the National Energy Research Scientific Computing Center (NERSC). As of this writing, Cori is the #12 supercomputer in the Top-500 list [33]. All experiments used the Intel Haswell nodes (Xeon E5-2698 v3) 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. Cray compilers and Cray MPICH is 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 [3]: Versatile package for molecular dynamics, often used biochemical molecules.
3. miniFE [19]: Proxy application for unstructured implicit finite element codes.
4. LULESH [23]: Unstructured Lagrangian Explicit Shock Hydrodynamics
5. HPCG [13] (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 operating on real-world HPC applications. It will be shown that the runtime overhead is almost 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 [26] 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 less than 1%. The optimized Linux kernel is based on a patch under review for a future Linux version.

TODO: Rohan, please review the previous paragraph for accuracy. Thanks. — Gene

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

Figure 3 shows the results for the five different real-world HPC applications running on a single node under MANA. Each run was repeated 5 times, and the figure shows the mean of the 5 runs with error bars for the smallest and largest of the 5 run times. The worst case overhead incurred by MANA is 3.15% in the case of miniFE (with 8 MPI ranks). In some cases, MANA exhibits negative overhead (improved performance of MANA over native operation). In most cases, the mean overhead (positive or negative) is less than 1%, and there is a maximum or minimum overhead within 0.1%. (miniFE shows a 0.5% negative overhead.)

TODO: I have a question. Figure 3 relates MANA to native. MANA was run 5 times. But how many times was native run? If it was run only once, is that because native runs have extremely reproducible times for runs? If not, don’t we need to do something like five native runs and take an average?

Figure 4: 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.)

TODO: If time permits or if we need the space, then Figures 3 and 4 can be created with reduced vertical height. Also, let’s increase the height of the white space between 0% and 90%, and maybe add some different dotted or dashed line above and below the white space to show the gap between 0% and 90%.

**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 4 shows the results for the 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 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 less than 1% with GROMACS.

**TODO:** “an overhead of less than 2%”; CLAMR seems to have a somewhat larger overhead. Should we say “an overhead that is typically less than 2%”?

### 3.2.2 Memory Overhead.

TODO: We need some clarifying introduction like these sentences: 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 note that while these regions are ignored during checkpointing by MANA, DMTCP/InfiniBand [10] would have been forced to save these memory regions as well, adding to the checkpointing overhead.

**TODO:** If we run out of space, we can comment out this section on “memory overhead”.

### 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. We chose these benchmarks to 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 follows 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. **TODO:** Please confirm that both the Figure 6 OSU benchmark and GROMACS were run in the single-node case. **FINAL VERSION:** Fixme: We can write Engaging-1 at MGHPCC site in the final version.

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 memory 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 note that while these regions are ignored during checkpointing by MANA, DMTCP/InfiniBand [10] would have been forced to save these memory regions as well, adding to the checkpointing overhead.

**TODO:** If we run out of space, we can comment out this section on “memory overhead”.

### 3.4 Checkpoint-restart Overhead

Next, we evaluate MANA’s ability to transparently checkpoint-restart different real-world HPC applications.

Figure 7 shows the checkpointing overhead for the 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 7 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, 34]. Thus, the overall checkpoint time is reflective of the checkpoint time taken by the slowest rank.

Our next set of questions were: what are the sources of the checkpointing overhead? Does the draining of MPI messages and 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 costs of these two minor sources of overhead can have a significant effect when the application issues many short MPI calls in a short time span. **TODO:** I don’t understand the last sentence. If there are many sends and receives, will the FS register or this part dominate? Or does this part dominate for calls other than send/receive? Are these effects always much less than 1% in our measurements here? — Gene

The first and larger source of overhead is then eliminated by using the patched Linux kernel, as discussed above. Point-to-point bandwidth benchmarks (Figure 6 from the OSU micro-benchmarks) were run both with and without the patched Linux kernel. 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 that application exhibited a higher runtime overhead (e.g., 1.5% in the case of 16 ranks). We did a similar experiment, running GROMACS with 16 MPI ranks on a single node with the patched kernel. **TODO:** Please review the original statement, because I’m not seeing a 3% overhead in the single-node figure.

**TODO:** You wrote: “We observed that the performance degradation was reduced to less than 1% from about 3%.” I don’t understand this statement. I see 1.5% when I look at the single-node figure.
Figure 5: OSU Micro-benchmarks under MANA. (Results are for two MPI ranks on a single node.)

Figure 6: Point-to-Point Bandwidth under MANA with patched and unpatched Linux kernel. (Higher is better.)

Figure 7: 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 9 shows the contribution of different components to the checkpointing overhead. We only show results for the case of 64 nodes for the five different benchmarks. In all the cases, the time to execute the two-phase algorithm (see Section 2.5) to ensure that the checkpointing does not occur in the middle of an MPI collective call was less than 1.6 s.

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

Finally, Figure 8 shows the restart overhead under MANA is presented 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). We observe that the restart times increase with 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 and even MPI library as this

---

(b) After optimizations

(a) Before optimizations

(a) Point-to-Point Latency

(b) Collective MPI_Gather

(c) Collective MPI_Allreduce
work communication between the checkpoint coordinator 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.

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 today, since both current vehicles for transparent checkpoint (a checkpoint-restart service for a particular MPI implementation or DMTCP/InfiniBand) 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 will typically configure and tune a locally recommended MPI implementation for performance. Migrating an MPI application along with its underlying MPI library eliminates the benefits 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 55 s into the run. The checkpoints were then copied (migrated) to a local cluster that uses Open MPI over the InfiniBand network.

Since the local cluster only provided dynamic libraries, GROMACS was restarted using a dynamically linked Open MPI library. At the time of restart, Open MPI was configured to run with 2 nodes (4 ranks per node), thus maintaining the 8 MPI ranks for GROMACS, but moving some formerly shared ranks (ranks on the same node) onto separate nodes.

The restart took 2.3 s. No performance degradation in GROMACS was observed after restart, when compared to a run in which MANA/GROMACS directly ran with an Open MPI-based lower half) on the local cluster. **TODO:** ADD SOMETHING LIKE THIS WHEN WE HAVE THE REVISED EXPERIMENT READY: The restarted GROMACS under MANA was compared with two other configurations: GROMACS using the local Open MPI configured to use the local InfiniBand network; and GROMACS/Open MPI configured to use TCP. The network-agnostic nature of MANA then allowed the Cori version of GROMACS also to be restarted on the local cluster with either network.

**TODO:** ADD: 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...
files with the local MPI implementation, and the native application was then launched in the traditional way.

TODO: As we discussed on the phone, this experiment will be much more impressive if we can present a bar graph with actual times for the comparison. This can be done by running first on Cori to determine the total runtime. Then checkpoint exactly at the halfway mark. (If it takes 10 minutes on Cori, checkpoint after 5 minutes.) Then on the local cluster, run the second half of GROMACS versus a full run of GROMACS running natively, i.e., without MANA. If the second half of GROMACS/MANA takes exactly half the time that native GROMACS takes on the local cluster, then we’ve proven that there is no degradation due to migration.

As you suggested, it will be great if we can have two network configurations on the local cluster: Open MPI with InfiniBand; and Open MPI with TCP. We can then show that the restarted MANA/GROMACS tracks perfectly with the local GROMACS: both are faster under InfiniBand and both are slower under TCP.

AND A FINAL WISH LIST: Can we add a third configuration: 8 ranks on a single node for the local cluster? So, the figure would show three pairs of bars: Open MPI/IB/2 ranks-per-node, Open MPI/TCP/2 ranks-per-node, Open MPI/XXX/8 ranks-per-node. In the last case, the network doesn’t matter since all 8 ranks are on one node. Each pair would show both a restarted GROMACS from Cori and half of the time for a native GROMACS. I hope that MPI launch time is tiny, so that it makes sense to report it this way, but an alternative is to add a timing statement in the main function of GROMACS so that we can get times after MPI_Init() and before MPI_Finalize().

4 DISCUSSION AND FUTURE WORK

TODO: If we’re short of space, we can remove a lot of this section on Discussion and Future Work. It takes up at least a full column.

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, the MPI application must be limited to the oldest MPI version on which one might wish to restart. But on the brighter side, a very long-running application can use MANA to survive a systems upgrade that installs a newer MPI library, a newer Linux kernel, or upgrades the x86 CPU.

Similarly, while MPI implies a standard API, any local extensions to MPI must be avoided. And either 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. Similarly, the constant values of the MPI constants must be the same on all MPI implementations being used, or else MANA must add process virtualization code to virtualize each of the MPI constants.

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

Nevertheless, 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 custom MPI for performance analysis (e.g., using PMPI for profiling or tracing; or using a specially compiled "debug" version of MPI to help developers understand an unusual bug in the MPI library that occurs only in the middle of a long run.

This work also helps support the 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 reconfiguring the binding of MPI ranks to specific hosts in order to better fit the underlying interconnect topology.

Currently, such bindings of MPI ranks are chosen statically and used for the life of the MPI application run. But MANA allows one to dynamically re-bind MPI ranks in the middle of a long run to create new configurations of rank-to-host bindings (new topology mappings). This is useful either when the MPI application enters a new phase for which a different rank-to-host binding is optimal, or else when other codes that run on the same cluster begin to create contention or interference through communication hotspots. This will enable researchers to leverage tools[6, 29] for online dynamic monitoring and dynamic performance engineering by creating new topology mappings for rank-to-host bindings.

TODO: Review last paragraph to discuss MPI Tools (MPI_T) interface of MPI 3.0. https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report/node372.htm. Maybe it already does a lot of what we’re saying here. Also, is the MPI_T interface compatible with MANA in the future, since it involves a number and type of variables that are discovered by the tools via the query interface? The tool writer is independent of the MPI implementation (but not the MPI application; probably this can’t be virtualized). Maybe a MANA-aware application must re-tune with the tools at restart time. See [29] for a good example of a tool (for MVAPICH2 in this case).

For dynamic performance engineering, MANA can co-locate arbitrary MPI ranks onto the same host, where they will benefit from MPI library optimizations such as shared memory for improved communication. Under older approaches to transparent checkpoint-restart, this was impossible, since the older approaches were saving all of process memory, including the shared memory regions created by the MPI library.

Finally, MANA can enable new approaches to dynamic load balancing by checkpointing on one cluster and restarting on a different cluster. This added flexibility allows system managers to burst current long-running applications into the Cloud during periods of heavy usage. At the same time, the restarted application
benefits from the locally configured MPI on the new cluster, which has been optimized for that cluster topology (e.g., through topology mappings).

TODO: Do we currently replay the MPI topology calls at restart time to declare certain topologies? If so, this is an important remark to add to the last paragraph. If not, then we should add this to the "Limitations" subsection with a comment that this will be remedied in a future version.

5 RELATED WORK

Process-in-process [20] loads the different ranks (the binary executable and its dependencies) in different logical “namespaces” (using dlmopen). So, there are multiple copies of the same library, including that of libc.

This approach does not support different libc’s trying to use a shared heap segment (which the kernel keeps track of for the process). A more fundamental issue is that of a shared runtime linker/loader (“ld.so”). The upper and the lower half will depend on the dlmopen functionality of a single “ld.so”. At restart time, we must load a second, newer lower half. This makes the approach non-transparent because the “ld.so” logic will call on the destructors of the first MPI library, which is now invalid but the second MPI library has been initialized after restart.

Garg et al., [15] showed that by running a non-reentrant library in a separate process (which is executable across checkpoint-restart), 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 one to do RPC across the two processes, and to transfer data between the two processes, which can be a large overhead.

Note that one of the assumptions of the split-process approach used by MANA is that the two halves must communicate with each other using a strict API. However, this is not a major limitation, since even the two-process based proxy approach has the same requirement.

McKernel [16] runs a “lightweight” kernel along with a full-fledged Linux kernel. The HPC application runs on the lightweight kernel, which provides implements time-critical system calls. 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, which is similar to MANA’s conception of a lower half, to minimize the call forwarding (argument marshalling/un-marshalling) overhead.

Sultana et al. [30] extend MPI checkpoint-restart support to also provide ability to save and restore MPI state (MPI identifiers such as communicators, and so on). For recovery of application state, it relies on application-specific code.

Hussey et al. [21] developed a network-agnostic checkpointing tool for OpenMPI. It relies on BLCR for checkpointing a single isolated process and relies on the MPI implementation to handle the network connections. This requires disconnecting the network services prior to checkpointing and then reconnecting the network services at resume time after completing the checkpoint. This not only imposes a large checkpointing overhead, but can also be difficult to do for certain communication mechanisms such as SysV shared memory. MANA does not require this disconnect and reconnect procedure at checkpoint time and is agnostic to the specific MPI implementations.

6 CONCLUSION

TODO: Still need to write a conclusion. In addition to the normal high points from the abstract, this should emphasize the actual and future performance: near-zero runtime overhead in a future Linux kernel; checkpoint/restart time proportional to memory footprint size; tested on real-world programs; etc.

Let’s also emphasize Section 3.6, since the cross-cluster migration showcases the MPI-agnostic, network-agnostic feature, and even the possibility of breaking the co-location of ranks on the same node, or creating new co-locations.

FINAL VERSION:

ACKNOWLEDGMENT

We would like to thank the reviewers for their constructive feedback. 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.

REFERENCES

[14] Qi Gao, Weikuan Yu, Wei Huang, and Dhabaleswar K. Panda. 2006. Application-Transparent Checkpoint/Restart for MPI Programs over InfiniBand. In PP ’06: