MANA-2.0: A Future-Proof Design for Transparent Checkpointing of MPI at Scale

author(s) omitted

***Abstract*—MANA-2.0 is a scalable, future-proof design for transparent checkpointing of MPI-based computations. Its network transparency ("network-agnostic") ensures that MANA will provide a viable, efficient mechanism for transparently checkpointing computational workloads on future supercomputers. MANA is being tested on today on the Cori supercomputer at NERSC with its proprietary Cray GNI network, but it is designed to work over any standard MPI running over an arbitrary network. MANA-2.0 is a work in progress that already supports transparent checkpoint-restart over thousands of ranks. Experimental results include GROMACS's point-to-point-intensive MPI implementation, and VASP's intensive use of collective communication. Perhaps the most important lesson to be learned from MANA-2.0 is a series of algorithms and data structures for library-based transformations that enable MPI computations over MANA-2.0 to reliably survive the checkpoint-restart transition needed for long-running HPC computations. The transformation of each MPI call into calls to a restricted set of underlying "checkpoint-safe" MPI calls is analogous to the central problem of compilers.**

***Index Terms*—transparent checkpointing, MANA, DMTCP, split-process model, production workloads, supercomputing**

I. Introduction

MANA (MPI-Agnostic Network-Agnostic checkpoint-restart) is a previously developed package for checkpointing MPI [1]. Among its unique features, MANA supports checkpoint-restart for MPI applications, while being transparent to (a) the MPI application; (b) the MPI library itself; and (c) the network libraries underlying the MPI library. These features are based on a novel split-process architecture (see II-A). Unlike previous approaches, MANA directly and transparently interposes on the MPI calls themselves, taking advantage of the standardized API for MPI.

MANA-2.0 is intended as a scalable, future-proof design for transparent checkpointing of MPI-based computations. While MANA is being tested on today's Cori supercomputer using the Cray GNI network, its network transparency ("network-agnostic") ensures that MANA will provide a viable, efficient mechanism for transparently checkpointing computational workloads on future supercomputers. This is important not only for fault tolerance for long-running HPC jobs, but also as a mechanism to chain together resource allocations (typically limited to 48 hours by HPC center policies), so as to run very long jobs while retaining the resource-allocation flexibility required by sysadmins.

MANA's unique feature of MPI and network transparency is a good fit for the National Energy Research Scientific Computing (NERSC) center[1], which employs supercomputing resources with a proprietary Cray GNI network. The development of MANA has been the basis for a long-term collaboration between NERSC and the MANA team. Note that previous approaches to checkpointing MPI [2]–[6] supported TCP and InfiniBand, but did not support the Cray GNI network.

This initial promise led to a goal of supporting production-level deployment of MANA on NERSC's Cori supercomputer [7]; and following that, to support Perlmutter (the #5 supercomputer in the world as of this writing [8]). But history has shown achieving the goal of production-level transparent checkpointing to be more difficult than anticipated. In personal communications, the authors both of the original MANA [1] and of an updated version [9] have both expressed concerns about the fragility of the software architecture and its key components.

In general, the difficulty of developing robust checkpointing algorithms for MPI can be understood by analogy to developing a new optimizing compiler. Both endeavors are concerned with translating high-level code representations into lower-level executions. Both endeavors require subtle algorithms to preserve the high-level semantics when translated to a lower level (compilers), or across the checkpoint-restart boundary (MANA).

This work demonstrates a robust version of MANA (MANA-2.0) that can scale to a high level. This work provides two novel elements:

1) While still a work in progress, MANA-2.0 is already shown to scale well on two very different types of computations (see Section IV). GROMACS [10], [11] highlights intensive MPI point-to-point computation. VASP [12] highlights intensive MPI collective computation. VASP is responsible for more than 20% of the machine time on Cori [7].
2) Perhaps even more important, MANA-2.0 is the fruit of a series of enhancements that serve as lessons learned for related research projects. MANA-2.0 applies a wrapper-function based strategy to *efficiently* translate each MPI call of the original MPI application into one or more direct MPI calls. The algorithms of MANA-2.0 include data structures that enable MANA-based computations to survive a checkpoint and full restart. In this respect, MANA-2.0 bears as much resemblance to a compiler

[1]NERSC is the primary computing facility for the US Department of Energy's Office of Science (https://www.nersc.gov/).

(translating into lower level MPI calls), as it does to a simple library utility.

The two novel elements mentioned above represent a qualitative difference of the current work over the original MANA. For the first time, MANA-2.0 demonstrates the ability to *reliably* checkpoint GROMACS, even at 2048 ranks. In comparison, the original MANA work [1] was intended as a proof-of-concept, only. The previous authors had encountered a race condition in their GROMACS experiments, and this race condition forced them to run the GROMACS experiments of their experimental section *multiple times*, before they could succeed in doing a checkpoint-restart — and this was the case for *each experiment performed for 256 ranks and above* [13].

The In a personal communication, the first author from the original MANA work (from [1]) noted that they had to run the experiments multiple times to get checkpoint-restart to go through for Gromacs, for 256 ranks and up. Since the initial implementation was primarily aimed to demonstrate a proof-of-principle, there remained multiple corner cases that would cause the program to crash at random during checkpoint-restart.

MANA-2.0 is an example of a larger class of projects that rely on source-level transformations of MPI calls while maintaining correctness and performance. See the beginning of Section III for a list of issues in MPI source transformations that were found to be important for correctness and performance. A short list of the relevant issues (expanded on in Section III) includes: (i) decomposition of blocking MPI calls into asynchronous calls (e.g., `MPI_Send` to `MPI_Isend/MPI_Test`); (ii) insertion of blocking MPI calls while avoiding deadlock (e.g., is inserting `MPI_Barrier` before `MPI_Bcast` valid?); (iii) determining if an MPI call can be satisfied locally (e.g., `MPI_Translate_group_ranks`); and (iv) when can it be proved that a (virtualized) MPI request object can no longer be accessed by the MPI application in the future.

This work is organized into the following sections. Section II briefly describes the underlying split-process design of the original MANA, and then gives further details of how checkpointing is supported for key components of MPI. Section III describes the algorithmic innovations of this work in fixing many of the deficiencies in key components of MANA. Section IV presents an experimental evaluation of the modified version of MANA. Section V presents the related work. Finally, Section VI is the conclusion.

II. Background

A. Split processes

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

```
1  USER_DEFINED_WRAPPER(int, Barrier, (MPI_Comm) comm)
2  {
3    int retval;
4    // Call the two-phase-commit algorithm before the real communication
5    commit_begin(comm);
6    // Disable checkpointing when the program is in the lower-half
7    DMTCP_PLUGIN_DISABLE_CKPT();
8    // Translate virtual resource ids if needed
9    MPI_Comm realComm = VIRTUAL_TO_REAL_COMM(comm);
10   // Context switch to the lower-half with this macro
11   JUMP_TO_LOWER_HALF(lh_info.fsaddr);
12   // NEXT_FUNC is a macro looking for the address of the real MPI functions
13   retval = NEXT_FUNC(Barrier)(realComm);
14   // Context switch back to the upper-half
15   RETURN_TO_UPPER_HALF();
16   // Re-enable checkpointing since finished the work
17   DMTCP_PLUGIN_ENABLE_CKPT();
18   // Clean up for the two-phase-commit algorithm
19   commit_finish();
20   return retval;
21 }
```

Fig. 1. Code snippet of the MPI_Barrier wrapper.

In practice, the upper-half program will be the MPI application program, dynamically linked with a "stub" MPI library. The "stub" MPI library consists of wrapper functions around each MPI call. The wrapper calls a lower-half function in the actual MPI library. Finally, the lower-half program consists of a small MPI application linked to the actual MPI library, which links to the necessary libraries. Figure 1 shows an example of the `MPI_Barrier` wrapper. The network library is based on the proprietary Cray GNI interconnect for Cori, and will be based on the proprietary HPE Cray Slingshot interconnect for the new Perlmutter supercomputer.

The advantage of this scheme is that only the upper-half program is checkpointed. (Only its memory is saved in a checkpoint image file.) This sidesteps the key problem of other checkpointing approaches: There is no need to disconnect and re-connect the network (the proprietary Cray GNI network in our case). At the time of restart, the lower-half program is started, and it loads into memory at the original address the upper-half program, from the checkpoint image file. For a deeper description of split processes, see the original paper of Garg et al. [1].

B. Overview of Semantic Components of MANA

We standardize here on MPI-3.1 [14]. There are primarily four categories for which MANA must save state at the time of checkpoint:

1) the state of all memory in the upper half;
2) a consistent snapshot associated with any MPI collective communication calls in progress (e.g., `MPI_Barrier` or `MPI_Bcast`;
3) a consistent snapshot associated with any MPI point-to-point calls in progress (e.g., `MPI_Send` and `MPI_Recv`;
4) any MPI one-sided communication calls (the `MPI_Win_XXX` family of calls).

MPI's one-sided communication calls are not yet supported in MANA. The support for the `MPI_Win_` family is in the roadmap of MANA. The details of the remaining three categories are described in the next section.

C. Virtualized MPI objects

MPI calls may create new objects of types such as `MPI_Comm` and `MPI_Request`. In the MANA wrapper functions around these calls, a new virtual object (virtualized communicator or virtualized request in our example) is created and returned to the user's MPI application. An internal mapping from the virtual object to the "real" object returned by the lower-half MPI library is maintained. Thus, when the user's MPI application makes a later call, using one of these virtualized objects, the MANA wrapper function automatically replaces the virtualized object by the real object stored in its mapping.

This is important, since the MPI application may make copies of its objects, to be stored at arbitrary addresses. So, at restart time, MANA simply updates its virtual-to-real mapping with new, real objects, instead of trying to directly patch the memory of the MPI application with updated real objects.

III. Novel algorithms for key components of MANA

The challenges in supporting MANA robustly can be attributed to several factors.

1) MANA is unusual in interposing directly at the level of the MPI API. A checkpoint can be taken *only* if no MPI rank is in the middle of the MPI library. Hence, some MPI calls, such as `MPI_Send` and `MPI_Recv`, interposed on by wrappers that convert the calls to asynchronous calls: `MPI_Isend` and `MPI_Irecv`, along with a loop around `MPI_Test`.
2) The conversion to semantically equivalent MPI calls can result in higher runtime overhead.
3) A conversion to semantically equivalent MPI calls, while valid for most MPI implementations, cannot be guaranteed for all MPI implementations. In particular, see [15], the MPI-4.0 addendum for semantics. This helps resolve questions such as: when it is valid to add an `MPI_Barrier` in front of a *non-blocking* MPI collective communication (e.g., the root in `MPI_Bcast`); whether the insertion of an `MPI_Barrier` will slow down or accelerate an MPI application (see [16, page 41: MPICH_COLL_SYNC]; and which MPI calls may be resolved solely using local information. `MPI_Barrier` before `MPI_Bcast` also can create deadlock. Details are discussed in following subsections.
4) While MANA can use its centralized coordinator as a side channel to communicate among the ranks, this is inefficient. Hence, MANA-2.0 instead makes direct use of MPI calls as being more maintainable and more efficient, while making sure semantically to be non-intrusive.
5) MANA-2.0 takes care to internally use MPI calls that solely access local information, such as `MPI_Translate_group_ranks`. However, the overhead can still be sensitive to particular MPI implementations.
6) MANA-2.0 virtualizes several objects such as MPI communicators and MPI requests. Thus, if a checkpoint-restart occurs between the creation of the object and a second use, then the virtualized object can be bound to a newly created object on restart. However, unless the growing list of virtualized objects is garbage-collected, the size of that list continues to grow — resulting both in a growing memory footprint, and in higher overhead to access an object.

The next subsection discusses these and other issues that arose.

A. Virtualized requests

Resources allocated by MPI libraries like `MPI_Comm` and `MPI_Groups` are virtualized to survive the ckpt-restart barrier, but `MPI_Request` was not. This issue did not arise in the earlier implementation [1] because non-blocking collective communications were not supported. Recall that new virtual variables of type `MPI_Request`, such as `MPI_Isend` and `MPI_IBcast` (broadcast) are created in the original non-blocking MPI function wrappers. The requests are retired when the application calls `MPI_Test` or `MPI_Wait`. The action of the MPI call must be completed by the time of an `MPI_Wait` call or an `MPI_Test` whose flag parameter returns "true" (success).

Unlike resources like `MPI_Comm`, which are never actually removed from the virtual ID table, MPI requests are generated so frequently that one needs to aggressively prune completed MPI requests to avoid large performance and large memory overhead. New virtual MPI requests are created in non-blocking MPI function wrappers, and retired in `MPI_Test` and `MPI_Wait` wrappers. One exception is handling requests generated by non-blocking point-to-point communications. In this case, virtual requests are saved in a special log-and-replay data structure. So we can't update the user's memory to update the request to `MPI_REQUEST_NULL`.

A two-step retirement algorithm is developed to safely delete completed requests without requiring knowledge of the addresses where the user's application may have stored the request. When a request is complete, we update the virtual ID table so that the completed virtual request points to a special value `MPI_REQUEST_NULL`. The next time `MPI_Test` and `MPI_Wait` are called, we know the virtual request is ready to be removed, since the real request is `MPI_REQUEST_NULL`. Then we can safely remove the virtual request from the table and set user's request variable to `MPI_REQUEST_NULL`.

B. Drain send-receive for point-to-point communication

In the previous work [1], MANA translated blocking point-to-point communications to their non-blocking versions, and used a variation of an all-to-all bookmark exchange algorithm to drain point-to-point messages in the network during checkpoint. Point-to-point communication wrappers accumulated the count of the number of messages at runtime. When checkpointing, each rank sent the count number of messages

to the coordinator, and the coordinator sent the total number of messages to each rank. If the total send and receive counts did not match, MANA used `MPI_Iprobe` to detect messages still in the network and tried to receive them with `MPI_Recv`. Finally, MANA updated the new send and receive counts to the coordinator and repeated the process.

This design has some drawbacks, however. Frequent communication with the coordinator can be expensive when running at large scale. And sharing only the total number of sends and receives makes it impossible to identify which rank the missing messages belong to.

Therefore, we improved the algorithm to use a smaller-grain message counter for each pair of ranks, and share only essential information with `MPI_Alltoall`. After calling `MPI_Alltoall` at checkpoint time, all ranks know, without further communication, how many bytes they were expected to receive and how many they actually received. Locally, each rank is able to use `MPI_Recv` to drain missing bytes from peers.

Another lesson we learned is that if `MPI_Recv` or `MPI_Irecv` has already been called, then `MPI_Iprobe` can no longer detect the message in the network. Therefore, if some rank found no messages in the network by using `MPI_Iprobe`, and if the send-receive count is still unbalanced, then there must be an unfinished `MPI_Irecv` waiting to be completed. In this case, instead of using an extra `MPI_Recv` to drain the message, we call `MPI_Test` on existing `MPI_Irecv` records to discover pending MPI requests associated with `MPI_Irecv` records, and to then drain the missing messages.

C. Keep a list of only the active communicators for the sake of restart

In the original design, when restoring MPI communicators during restart, all functions used to create communicators were recorded and replayed. Therefore, many communicators that were no longer used would be recreated during restart. In addition, we couldn't retire any MPI communicators, in case they were used to create other communicators. As a result, time was wasted on replaying unnecessary functions. The virtual ID table (mapping) for communicators also occupied more memory and slowed down the lookup performance.

Our new design instead keeps a list of active communicators and groups, and reconstructs only communicators and groups in the active list during restart. A knowledge of the underlying MPI group and its members suffices to recreate a semantically identical communicator. So, it is no longer necessary to replay MPI calls that build new communicators from old communicators.

D. Adding a barrier before collective communication

In the MPI standard [14], there is no requirement in a collective function that all participating ranks must enter the function before any rank can return. (For example, the "root" in `MPI_Bcast` can broadcast its message and return before other ranks receive the message.) However, because of the two-phase-commit algorithm (see [1]), we add a barrier before each collective communication call. We do this so that we will not be required to checkpoint in the middle of an arbitrary collective call, such as `MPI_Bcast`. This added semantic can change the behavior of applications.

A major impact is the performance of collective communications. For example, adding a barrier before a `MPI_Bcast` forces the "root" rank to wait until all other ranks arrives. Generally the barrier makes the `MPI_Bcast` running two to three times slower. However, in the case of `MPI_Allreduce` where all ranks need to send and receive data from other ranks, the barrier slightly improved the performance. Hence, see the recommendation of Cray for trying both and testing with CRAYPAT [16, page 41].

E. Deadlock between `MPI_Bcast` and `MPI_Send/Recv`

In addition to the impact on performance, in some rare cases the added barrier can lead to deadlocks that do not exist in the native MPI application. Assuming two ranks, rank 0 and rank 1 communicate with each other. Rank 0 calls `MPI_Bcast` as the "root" rank then calls `MPI_Send`. Rank 1 calls `MPI_Recv` and then calls `MPI_Bcast` as the receiver. There is no deadlock when running natively. However, if we add an barrier before the `MPI_Bcast`, rank 0 will wait rank 1 to join the barrier, but rank 1 can only join the barrier when rank 0 return from the `MPI_Bcast` and `MPI_Send`. Therefore, we have a deadlock.

To avoid the deadlock, We provided an alternative wrapper implementations for functions like `MPI_Bcast` that uses point-to-point communications instead of the real `MPI_Bcast` function in the lower half. However, this is just a naive implementation and lacks of optimization compared to real MPI implementations. A better algorithm that doesn't require barriers before collective communiation is on MANA's roadmap.

F. Handling MPI named constants in Fortran

Because of the nature of Fortran, some MPI named constants, such as `MPI_IN_PLACE` and `MPI_STATUS_IGNORE`, are set at link time instead of compile time [17]. This is because Fortran uses common blocks, instead of true global constants. So, named constants in Fortran are addresses to special values in the underlying MPI library. Therefore, when using MANA with Fortran MPI applications, the named constants passed into MANA's Fortran wrappers are addresses, instead of the actual constant values as in the C interface. See [17] for details.

To identify these link-time named constants correctly in MANA's wrappers, we linked a small Fortran program into MANA that discovers the value/address of the Fortran named constants dynamically. If a parameter passed in from a user's application matches a Fortran named constants, then MANA-2.0 replaces the value with the equivalent C constant when calling the real MPI function in the lower half.

G. The FS register

A major source of the runtime overhead is due to the use of the "FS" register. For systems that cannot use the FSGSBASE Linux kernel patch, we designed a workaround to reduce the cost of using the "FS" register. For details, see [18] in this workshop.

H. C++ lambda functions

C++ lambda functions were used in many MPI function wrappers in MANA to increase the readability of codes, but they come at the cost of performance. A C++ lambda function was found to be compiled into three or four additional call frames at runtime. For frequent MPI calls, this can add significant runtime overhead. To remove lambda functions in MANA, functions that take lambda functions as callbacks are decomposed into dedicated "prepare" and "finish" functions.

For example, in collective communication wrappers, codes that call the real collective functions in the MPI library are wrapped in lambda functions, and passed into the `commit()` function as part of the two-phase-commit algorithm describes. The `commit()` function will do the preparation work, call the lambda function, and finally clean up the resources and states. In this new design, the commit function is separated into two functions: `commit_begin()` for the preparation and `commit_end()` for the cleanup. The wrapper is responsible for calling: (i) `commit_begin()`; (ii) the codes that used to be in the lambda function; and (iii) `commit_end()` in sequence to perform the two-phase-commit algorithm.

I. Other sources of runtime overhead

Other sources of runtime overhead in MANA-2.0 are reported in [19]. These smaller factors also contribute to MANA's runtime overhead. A brief list of these factors follows.

1) Translating virtual ID to real ID depends on map operations of C++ std::map. Typically C++ std::map requires $O(\log n)$ to lookup an entry in the map. In some cases, we also need linear search in the map.
2) Disable and enable DMTCP checkpoint are used widely in MPI function wrappers. The cost of lock operations are too expensive because of the high frequency usage.
3) An internal helper method that translates local ranks of a communicator to global ranks makes several calls to the lower half.
4) We currently replay all non-blocking collective communications, like `MPI_Ibarrier`, `MPI_Ireduce` and `MPI_Ibcast`, to re-create virtualized requests. Not only is time wasted to create completed requests, but this also increases the size of virtual request table and slows down the translation time.

J. Stragglers

A *straggler* is an MPI rank that participates in a collective communication, but because it is finishing a CPU-intensive operation, it can take minutes to hours before it can join the collective communication. As a result, the completion of the collective communication is delayed. Even worse, from the viewpoint of checkpointing, no checkpoint can take place while some ranks are still in the middle of a collective call in the lower-half MPI library.

K. Globally unique IDs: `MPI_Translate_group_ranks`

A challenge in the previous implementation of the two-phase-commit algorithm is that the MANA centralized coordinator does not know which ranks participate in the same active communicator. This limits the ability of the MANA centralized coordinator to determine which MPI ranks must stop and wait for the final checkpoint command, and which MPI ranks must continue to execute in order to "unblock" later collective communication calls.

In order to get around this issue in MANA-2.0, each MPI rank reports to the centralized coordinator whether it is currently executing within a collective communication call — and if so, provides a globally unique id for that collective communicator. For performance reasons, the rank must compute this globally unique id *without further communication with its peers*. This is done using `MPI_Translate_group_ranks`. This allows it to translate the ranks of the current communicator to the corresponding rank in `MPI_COMM_WORLD`. The ranks in the current communicator are all known as 0, 1, ..., `MPI_Comm_size()`-1, where we have taken liberties with the syntax of `MPI_Comm_size()`. `MPI_Translate_group_ranks` then produces the set of corresponding ranks in `MPI_COMM_WORLD`, and a hash function is used to produce an integer that is globally unique with high probability.

L. Hybrid Phase 2 (out of scope for this article)

To further improve the performance of the two-phase-commit algorithm, we designed a hybrid version of the algorithm, which removes the barrier before each collective communication. The details of this improved two-phase-commit algorithm are extensive, and so are out of scope for this brief survey.

M. Lessons learned

There are some lessons learned from algorithms discussed above. First, additional communications used by MANA should be minimized. Where possible, use MPI calls for MANA's internal requirements for sharing information among ranks, instead of relying on MANA's centralized coordinator. Also where possible, MPI calls that complete based on local information is preferred over MPI calls requiring peer-to-peer communication. Minimizing the communication inside MANA not only improves the runtime performance, but also reduces race conditions and helps debugging.

Another lesson learned in MANA-2.0 is that some MPI calls can be emulated with other MPI calls. For example, `MPI_Wait` is implemented as a loop that repeatedly calls `MPI_Test`. With this emulation, tools like MANA can interrupt a time-consuming or blocking MPI call without breaking the execution. However, emulating MPI calls with other

MPI calls requires deep understanding of the MPI semantics. Otherwise, it's easy to introduce bugs without noticing. For example, adding a barrier before `MPI_Bcast` may result in deadlock.

Last but not least, one should instrument MANA to provide additional information that can be used in its algorithms: a globally unique id for each communicator (see `MPI_Translate_group_ranks`); recording the number of bytes sent and received for each possible sender-receiver pair (using `MPI_Alltoall`).

IV. Experimental Evaluation

All experiments were run on Cori, a Cray XC40 system at NERSC. Cori has two types of compute nodes, dual-socket Intel Haswell and single-socket KNL nodes, interconnected with Cray Aries network. Each Haswell node has 32 cores (64 hardware threads) running at 2.3 GHz , and 128 GB DDR4 2133 MHz memory; each KNL node has 68 cores (272 hardware threads) running at 1.4 GHz, and 96 GB DDR4 2400 MHz memory. Cori runs Cray Linux environment version 7.0.Up01 with Linux kernel version 4.12. All of the computations were done using Cori's burst buffer [20], the most suitable file system for writing checkpoint images on Cori.

We evaluated MANA-2.0 using two commonly used applications at NERSC: VASP, a materials science code and Gromacs, a molecular dynamics code. Two versions of VASP were tested: VASP 5 (5.4.4), a pure MPI code and VASP 6 (6.2.1), a hybrid OpenMP + MPI code. Both VASP versions were compiled with an Intel compiler (v2019.3.199) and were linked to Cray MPICH (7.7.10), MKL (2019.3.199) and FFTW (3.3.4) libraries. Gromacs (2021.02) was also compiled with the Intel compiler 2019.3.199, and was linked to Cray MPICH 7.7.10, and FFTW 3.3.8.

Two branches of MANA were used in the experiments. A relatively stable branch (interface7) with a higher runtime overhead was used for checkpoint/restart experiments; another branch (interface8) containing the runtime overhead fixes was used in the runtime overhead tests. MANA is free and open-source software [21]. Documentation of the internals of MANA can be found at [19].

A. Running GROMACS at Scale

Gromacs was chosen to evaluate the scalability improvement of MANA after the code enhancements described earlier. Gromacs was run with MANA-2.0 on a AuCoo monolayer system containing 407,156 atoms (nano particles in water). This was a system studied in [22] by a NERSC user.

First, we evaluated the runtime overhead of MANA by running the benchmark using 1 to 64 Haswell and KNL nodes (strong scaling) with and without MANA. We used the interface8 branch of MANA which contains a few overhead fixes. We measured the Gromacs run time of 10,000 MD steps. Figure 2 shows the results. The blue and red bars show the run time of Gromacs when running natively and under MANA, respectively; the yellow line shows the run-time ratio between

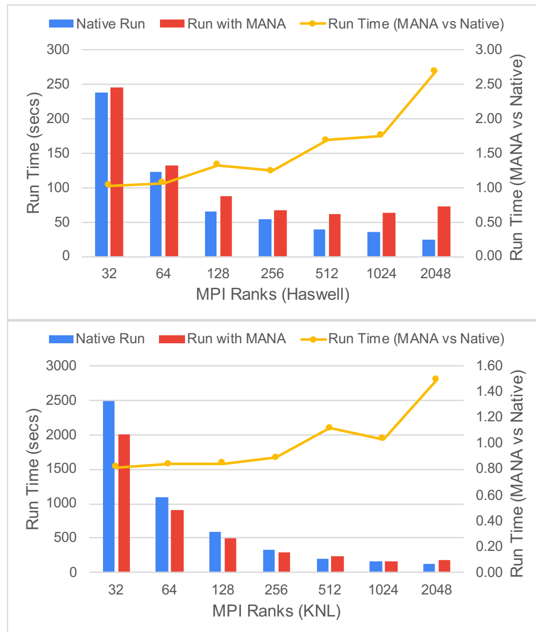

Fig. 2. Run time comparison between running Gromacs natively (blue bars) and under MANA (red bars) on Haswell (upper panel) and KNL (lower panel) nodes. The yellow line represents the run-time ratio between the MANA-enabled and native runs. Experiments were run on Cori Haswell and KNL nodes using 32 ranks per node. For the KNL runs, each task was run with two OpenMP threads.

the MANA-enabled and native runs. One can see that for KNL runs, the runtime overhead is not significant except at 2048 ranks, but for Haswell runs, except 1 and 2 node runs ($< 4\%$), the runtime overhead is still excessively high and increases rapidly when the number of processes increases. We continue our efforts to further reduce MANA's runtime overhead by addressing the remaining causes of the overhead.

Next, we tested MANA's checkpoint/restart capability at scale. We ran Gromacs with MANA using 2048 ranks using 64 Haswell and KNL nodes (32 ranks/node), respectively. For the KNL runs we used two OpenMP threads per rank. We checkpointed the job every 5 minutes, terminated it every 8 minutes, and then restarted it. MANA was able to checkpoint/restart the job many times reliably running at 2048 ranks. Figure 3 shows the checkpoint/restart overheads of the first 10 checkpoint/restart for both Haswell and KNL runs. The yellow line shows the total checkpoint image sizes. While the figure shows the scalability improvement of MANA, it also exposes multiple issues when running at this scale. First, the checkpoint file size increases over time both on Haswell and KNL runs. For example, the total file size grows from 1.4 TB at the first checkpointing to about 5 TB at the 10th checkpointing with Haswell runs. Second, when running under MANA the job uses significantly more memory than the native run and its memory usage increases over time, while the native run has a consistent memory footprint throughout the run. Third, we observed that occasionally checkpointing does not occur at the specified checkpoint interval. These issues are

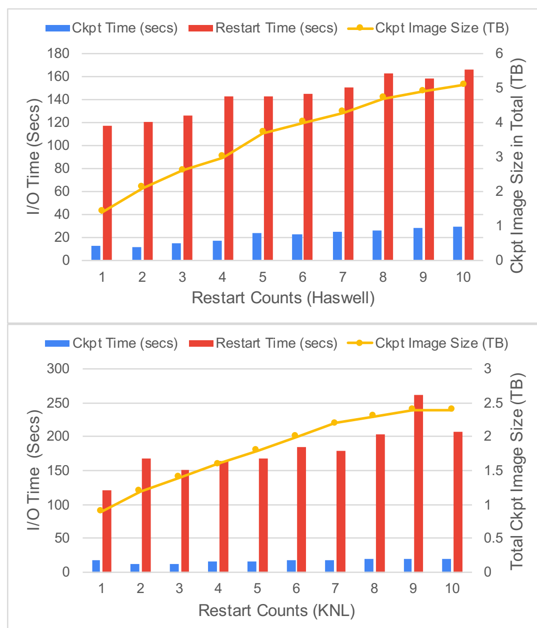

Fig. 3. Checkpoint/Restart overhead of MANA when running Gromacs with 2048 ranks on Haswell (upper panel) and KNL (lower panel) nodes on Cori's Burst Buffer. The blue and red bars show the checkpoint and restart time, respectively; the yellow line indicates the total size of the checkpoint files.

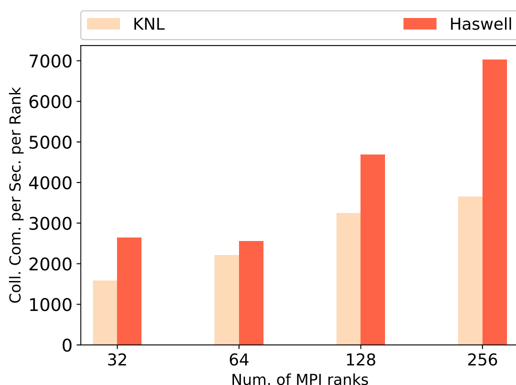

Fig. 4. Number of collective communications per second per rank for VASP-5 on Haswell and KNL nodes. When doubling the number of ranks, the growth in the number of collective calls is roughly logarithmic in the number of nodes. This figure was cited from [18].

under investigation now. We identified a memory leak in the code, and implemented a fix in the latest development branch of MANA. However, the latest development branch can not restart at 2048 ranks for some unknown reason. So we still use the result of the stable branch.

Note that while Gromacs does not scale well when using more processes with this benchmark (partially because we did not carefully tune the run for optimal load balance), this does not impact our goal of demonstrating the scalability of MANA.

B. VASP: a resource for robustness testing

We tested MANA-2.0 with VASP, a materials science code that consumes the most computing cycles at NERSC. VASP has been extensively tested with MANA using the representative workloads summarized in Table I. These benchmark cases were chosen to cover the representative VASP workloads and to exercise different code paths. For example, the first test case, denoted as PdO4 in the table, is a PdO slab containing 348 atoms. It was chosen to test the most commonly used code path, the DFT functional calculations using the RMM-DIIS iteration scheme. Some of the test cases were actual NERSC users' production jobs.

Many of the MANA code enhancements described earlier arose from fixing bugs and issues exposed by these VASP jobs when running them in production settings. As of this writing MANA (interface7) can successfully checkpoint and restart all the benchmark cases listed in Table I with both VASP 5 (MPI) and VASP 6 (OpenMP + OpenMP). For VASP 6 we needed to disable the use of `MPI_Win_` family APIs at compilation time to use MANA, because they are not yet supported in MANA. There are still other issues to resolve, e.g., for example, some of the VASP jobs run into segmentation faults after many rounds of checkpoint/restart.

Note that VASP is highlighted for its intensive use of MPI collective communication. While users typically run VASP across a small number of nodes due to its low-latency communication requirements, VASP invokes an unusually high frequency of MPI collective calls per second, as shown in Figure 4. This presents an additional challenge: runtime overhead.

C. MANA "interface8": initial successes in reducing runtime overhead

VASP was chosen to evaluate improvement in runtime overhead. We tested the CaPOH workload with 128 ranks on both Haswell nodes and KNL nodes. Table II shows the performances of the native VASP program, and VASP run under the MANA interface7 branch and MANA interface8 branch.

The MANA interface7 branch focuses on scalability and stability; the interface8 branch is our first attempt to reduce the runtime overhead. Currently, interface8 includes the hybrid two-phase-commit algorithm and removes lambda functions in the code base. From the table, we can see that on Haswell nodes, the runtime overhead reduced from 64% to 40%. On KNL nodes, the runtime overhead reduced from 99% to 46%. As discussed in the algorithm section, there are additional known sources of runtime overhead. We continue our efforts to solve each problem and reduce MANA's runtime overhead.

V. Related Work

The history of checkpointing of MPI is littered with approaches that tried too closely to tie the checkpointing process to a specific underlying network.

There have also been a series of checkpointing approaches for particular implementations of MPI. These include: the Open MPI checkpoint-restart service [5], [23], the MVAPICH2 checkpoint-restart service [4] — both of which temporarily disconnect the network and then delegate to BLCR [24] for checkpointing an individual process. Similarly, MPICH-V [2]

TABLE I
VASP TEST CASES FOR MANA-2.0. THESE CASES WERE CHOSEN TO COVER REPRESENTATIVE WORKLOADS AND TO EXERCISE DIFFERENT CODE PATHS.

	PdO4	GaAsBi-64	CuC_vdw	Si256_hse	B.hR105_hse	PdO2	CaPOH	WOSiH	GaAs-GW0
Electrons (Ions)	3288 (348)	266 (64)	1064 (98)	1020 (255)	315 (105)	1644 (174)	288 (44)	80 (18)	8(2)
Functional	DFT	DFT	VDW	HSE	HSE	DFT	DFT	HSE	GW0
Algo	RMM (VeryFast)	BD+RMM (Fast)	RMM (VeryFast)	CG (Damped)	CG (Damped)	RMM (VeryFast)	BD (Normal)	BD+RMM (Fast)	BD (Normal)
KPOINTS	1 1 1	4 4 4	3 3 1	1 1 1	1 1 1	1 1 1	2 1 1	3 3 3	3 3 3

TABLE II
PERFORMANCE COMPARISON OF THE VASP CAPOH WORKLOAD WITH 128 RANKS.

	Native	MANA interface7	MANA interface8
Haswell	25s	41s	35s
KNL	69s	137s	101s

disconnects a transport layer channel of MPICH (primarily based on TCP). It then delegates to the Condor package for checkpointing single-threaded individual processes [25].

Each of the above packages is implemented within a particular implementation of MPI. In contrast, the original DMTCP [3] (for TCP), and an InfiniBand plugin for DMTCP [6] are independent of the MPI implementation, and do not disconnect the network during checkpoint. But both are otherwise tightly bound to the underlying network.

An approach to support mobile MPI applications exists, albeit while partially abandoning application transparency and requiring re-compilation of the MPI application source code [26]. And CIFTS provides a fault-tolerant BLCR-based "backplane" [27].

MANA then introduced the *split-process model* for checkpointing of MPI [1]. Details are in Section II-A. Efforts to deploy MANA at NERSC are described in [7], while MANA was previously updated for compatibility with the latest NERSC environment and to remove code specific to one environment, as described in [9]. The first work [1] demonstrated transparent checkpointing of GROMACS [10] over MPI for 512 ranks over 64. The second work [9] demonstrated transparent checkpointing for 64 ranks with GROMACS and 512 ranks with the HPCG benchmark [28].

Note that transparent checkpointing of MPI was already demonstrated to the level of 16,368 ranks for NAMD and 32,368 ranks for HPCG (using 1/3 of the supercomputer) on Stampede at TACC in 2016 [29]. That early work was based on DMTCP's transparent support for InfiniBand, and that code likely would not on today's machines using either Cray GNI or extensions to the original InfiniBand. Further, no effort was made in the current work to test the limits of scalability of MANA-2.0.

VI. CONCLUSION

This report on MANA-2.0 represents encouraging progress toward a robust, reliable package for transparent checkpointing that will be future-proof. There are two important lessons from this work. First, each individual subsystem for MANA-2.0 must be carefully designed with appropriate data structures and algorithms to enable an MPI computation to survive over the checkpoint-restart barrier. The subsystems requiring particular support are: point-to-point (translating MPI_Send to MPI_Isend, etc.); MPI collective communication (allowing each MPI rank to proceed until all MPI ranks have reached a safe point with no MPI rank currently in an MPI call); decisions whether to wait for an MPI call to complete, or to virtualize and replay at restart time; MPI requests (virtualizing those requests and deciding when the memory of old requests can be reclaimed); and in the case of asynchronous MPI calls, deciding which ranks must replay point-to-point and collective calls in order to re-instantiate vitual MPI requests for completion after restart.

Second, while MANA-2.0 now runs reliably over Cray's GNI network interconnect and library, its design is expected to also run equally well on the future Perlmutter's HPE Cray Slingshot network interconnect on Perlmutter (currently the #5 supercomputer in the world, still being tested). MANA is still a work in progress. However, it runs reliably with GROMACS (emphasizing MPI point-to-point communication) and with VASP (emphasizing MPI collective communication). For running at larger scales, in order to write the checkpoint images of the many ranks in parallel, it has been found necessary to use Cori's high-performance burst buffer with access to a faster file system.

In the case of GROMACS, the currently chosen GROMACS input resulted in an imbalanced configuration at 2048 ranks — even for running GROMACS natively. Nevertheless, MANA-2.0 scales reasonably to 2048 ranks. In future work, we will design a custom GROMACS input suitable for testing at very large scale.

REFERENCES

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

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

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

[4] Q. Gao, W. Yu, W. Huang, and D. K. Panda, "Application-transparent checkpoint/restart for MPI programs over InfiniBand," in *Int. Conf. on Parallel Processing (ICPP'06)*, 2006, pp. 471–478.
[5] J. Hursey, J. M. Squyres, T. I. Mattox, and A. Lumsdaine, "The design and implementation of checkpoint/restart process fault tolerance for Open MPI," in *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2007, pp. 1–8.
[6] J. Cao, G. Kerr, K. Arya, and G. Cooperman, "Transparent checkpoint-restart over InfiniBand," in *ACM Symposium on High Performance Parallel and and Distributed Computing (HPDC'14)*. ACM Press, 2014.
[7] Z. Zhao, R. Hartman-Baker, and G. Cooperman, "Deploying checkpoint/restart for production workloads at NERSC," in *International Conference for High Performance Computing Networking Storage and Analysis*, 2020.
[8] "Top500 supercomputers (June, 2021)," https://www.top500.org/lists/top500/2021/06/, 2018, [Online; accessed Aug., 2021].
[9] P. S. Chouhan, H. Khetawat, N. Resnik, T. Jain, R. Garg, G. Cooperman, R. Hartman-Baker, and Z. Zhao, "Improving scalability and reliability of MPI-agnostic transparent checkpointing for production workloads at NERSC (extended abstract)," in *First International Symposium on Checkpointing for Supercomputing (SuperCheck'21)*, 2021, https://arxiv.org/abs/2103.08546; from https://supercheck.lbl.gov/resources.
[10] "Gromacs," http://www.gromacs.org/.
[11] "Gromacs ADH benchmark," ftp://ftp.gromacs.org/pub/benchmarks/ADH_bench_systems.tar.gz.
[12] "VASP," https://www.vasp.at/.
[13] R. Garg, "Personal Communication (from author of original MANA paper)," Sept., 2021.
[14] Message Passing Interface Forum, "MPI: A message-passing interface standard (version 3.1)," Jun. 4, 2015. [Online]. Available: https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf
[15] ——, "Summary of the semantics of all operation-related MPI procedures," Feb. 24, 2021. [Online]. Available: https://www.mpi-forum.org/docs//mpi-4.0/addendum-Semantics.pdf
[16] Cray, "Understanding communication and mpi on cray xc40," 2014. [Online]. Available: https://www.hpc.kaust.edu.sa/sites/default/files/files/public//KSL/150607-Cray_training/3.05_cray_mpi.pdf
[17] J. Zhang, B. Long, K. Raffenetti, and P. Balaji, "Implementing the mpi-3.0 fortran 2008 binding," in *Proceedings of the 21st European MPI Users' Group Meeting*, 2014, pp. 1–6.
[18] **AUTHORS OMITTED DURING DOUBLE-BLIND REVIEW**, "Removing kernel overhead in MANA's split-process model for checkpointing (tentative title)," in *Proceedings of SuperCheck Workshop at SC'21 (submitted to this workshop)*, 2021.
[19] MANA team, "MANA plugin documentation," 2021. [Online]. Available: https://docs.google.com/document/d/1pT25gvMNeT1Vz4SU6Gp4Hx9MGLfkK8ZK-sSOwtu5R50
[20] "Cori's burst buffer," https://docs.nersc.gov/filesystems/cori-burst-buffer/.
[21] MANA team, "MANA software (refactoring branch)," 2021. [Online]. Available: https://github.com/mpickpt/mana/tree/refactoring
[22] G. Brancolini *et al.*, "The manuscript is under review," *JOURNAL OMITTED*, 2021.
[23] J. Hursey, T. I. Mattox, and A. Lumsdaine, "Interconnect agnostic checkpoint/restart in Open MPI," in *Proc. of 18th ACM Int. Symp. on High Performance Distributed Computing*, 2009, pp. 49–58.
[24] P. H. Hargrove and J. C. Duell, "Berkeley Lab Checkpoint/Restart (BLCR) for Linux clusters," *Journal of Physics: Conference Series*, vol. 46, no. 1, p. 494, 2006.
[25] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny, "Checkpoint and migration of UNIX processes in the Condor distributed processing system," https://research.cs.wisc.edu/htcondor/doc/ckpt97.ps, University of Wisconsin, Madison, Wisconsin, Technical Report 1346, April 1997.
[26] R. Fernandes, K. Pingali, and P. Stodghill, "Mobile MPI programs in computational grids," in *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, 2006, pp. 22–31.
[27] R. Gupta, P. Beckman, B.-H. Park, E. Lusk, P. Hargrove, A. Geist, D. Panda, A. Lumsdaine, and J. Dongarra, "CIFTS: A coordinated infrastructure for fault-tolerant systems," in *Int. Conf. on Parallel Processing (ICPP'09)*, September 2009, pp. 237–245.
[28] "HPCG," https://www.hpcg-benchmark.org/.
[29] J. Cao, K. Arya, R. Garg, S. Matott, D. K. Panda, H. Subramoni, J. Vienne, and G. Cooperman, "System-level scalable checkpoint-restart for petascale computing," in *22nd IEEE Int. Conf. on Parallel and Distributed Systems (ICPADS'16)*. IEEE Press, 2016, pp. 932–941.