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

Yao Xu†
Khoury College of Computer Sciences
Northeastern University
Boston, USA
xu.yao1@northeastern.edu

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

Rohan Garg
Nutanix, Inc.
Seattle, USA
rohan.garg@nutanix.com

Harsh Khetawat
Department of Computer Science
North Carolina State University
Raleigh, USA
hkhetaw@ncsu.edu

Rebecca Hartman–Baker*
NERSC
Lawrence Berkeley National Laboratory
Berkeley, USA
rjhartmanbaker@lbl.gov

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

2021 SC Workshops Supplementary Proceedings (SCWS) | 978-1-6654-8388-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/SCWS55283.2021.00019

***Abstract*—MANA-2.0 is a scalable, future-proof design for transparent checkpointing of MPI-based computations. Its network transparency ("network-agnostic") feature ensures that MANA-2.0 will provide a viable, efficient mechanism for transparently checkpointing MPI applications on current and future supercomputers. MANA-2.0 is an enhancement of previous work, the original MANA, which interposes MPI calls, and is a work in progress intended for production deployment. MANA-2.0 implements a series of new algorithms and features that improve MANA's scalability and reliability, enabling transparent checkpoint-restart over thousands of MPI processes. MANA-2.0 is being tested on today's Cori supercomputer at NERSC using Cray MPICH library over the Cray GNI network, but it is designed to work over any standard MPI running over an arbitrary network. Two widely-used HPC applications were selected to demonstrate the enhanced features of MANA-2.0: GROMACS, a molecular dynamics simulation code with frequent point-to-point communication, and VASP, a materials science code with frequent MPI 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-based computations over MANA-2.0 to reliably survive the checkpoint-restart transition.**

***Index Terms*—transparent checkpointing, MANA-2.0, split-process, MPI, supercomputing**

I. Introduction

MANA (MPI-Agnostic Network-Agnostic transparent checkpointing tool) is a previously developed package for checkpointing MPI applications [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 the MPI calls themselves, taking advantage of the standardized API for MPI.

Transparent checkpointing is a prerequisite for system-level checkpointing, a vital tool for the operational needs of computing centers. System-level checkpointing is used to manage jobs in a real-world environment, which includes outage and maintenance events, and workloads with differing priorities and turnaround-time expectations. For example, experimental and observational facilities supported by the U.S. Department of Energy's Office of Science (DOE-SC) often require high-priority, real-time access to computing resources to generate immediate feedback for follow-up experiments.

While many scientific applications employ some level of internal checkpoint-restart (C/R) support, they lack a standard API for checkpoint and restart. And they usually require waiting for a particular computation phase (e.g., after an iteration completes). For example, when chaining together allocation slots for a long-running execution, the inability to guarantee a checkpoint within the last half hour of an allocation makes its use inflexible [2].

In addition, application-level checkpoint-restart may support some application features and not others, especially for applications with a large code base. As an example, VASP [3] has internal C/R support for atomic relaxation and MD simulations, but not for Random Phase Approximations. VASP has tens of thousands of users, and it consumes about 20% of computing cycles at the NERSC supercomputing center[1]. Transparent checkpointing helps VASP users to chain long-running jobs, while allowing NERSC to shift this 20% of its computing resources upon arrival of a large, real-time workloads as described above.

*This work was supported by the Office of Advanced Scientific Computing Research in the Department of Energy Office of Science under contract number DE-AC02- 05CH11231.

†This work was partially supported by National Science Foundation Grant OAC-1740218 and a grant from Intel Corporation.

[1]NERSC (National Energy Research Scientific Computing) is the primary computing facility for the U.S. Department of Energy's Office of Science (https://www.nersc.gov/).

978-1-6654-8388-9/21/$31.00 ©2021 IEEE
DOI 10.1109/SCWS55283.2021.00019

MANA-2.0 is intended as a scalable, future-proof design for transparent checkpointing of MPI-based computations. While MANA is being tested on NERSC'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 both current and future supercomputers.

MANA's unique feature of MPI and network transparency is a good fit for NERSC, which employs supercomputing resources with a proprietary Cray GNI network. The development of MANA is the basis for a long-term collaboration between NERSC and the MANA team. Note that previous approaches to checkpointing MPI [4]–[8] supported TCP and InfiniBand, but did not support the Cray GNI network. The original MANA was the first package that could support Cray GNI, through a new split-process architecture [1].

This initial promise led to a goal of supporting production-level deployment of MANA on NERSC's Cori supercomputer [9]; and following that, to support Perlmutter (the #5 supercomputer in the world as of this writing [10]). But history has shown achieving the goal of production-level transparent checkpointing to be more difficult than anticipated. In personal communications, the authors of the original MANA [1] and of an updated version [11] have both expressed concerns about the fragility of the software architecture and its key components.

In general, the challenge 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 [12] highlights intensive MPI point-to-point communication. VASP [3] highlights intensive MPI collective communication. VASP is responsible for more than 20% of the machine time on Cori [9].
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 (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 MPI processes. In comparison, the original MANA work [1] was intended as a proof-of-concept, thus it was not able to reliably checkpoint-restart at this scale.

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 whether 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, then provides 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 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 contained within the same virtual memory, a function from one program (typically the "upper-half" program) may call a function of the other program (typically the "lower-half" program) — so long as the address of the lower-half function is known to the upper-half function.

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 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 issue of other checkpointing approaches: There is no need to disconnect and re-connect the network (the Cray GNI network in our case). At the time of restart, the lower-half program is started, and it loads the upper-half program into memory at the

```
1  USER_DEFINED_WRAPPER(int, Barrier, (MPI_Comm) comm)
2  {
3    int retval;
4    // Call the two-phase-commit algorithm before the real
5    // communication
6    commit_begin(comm);
7    // Disable checkpointing when the program is in
8    // the lower-half
9    DMTCP_PLUGIN_DISABLE_CKPT();
10   // Translate virtual resource ids if needed
11   MPI_Comm realComm = VIRTUAL_TO_REAL_CCMM(comm);
12   // Context swtich to the lower-half with this macro
13   JUMP_TO_LOWER_HALF(lh_info.fsaddr);
14   // NEXT_FUNC is a macro looking for the address
15     // of the real MPI function
16   retval = NEXT_FUNC(Barrier)(realComm);
17   // Contex switch back to the upper-half
18   RETURN_TO_UPPER_HALF();
19   // Re-enable checkponting since the work is finished
20   DMTCP_PLUGIN_ENABLE_CKPT();
21   // Clean up for the two-phase-commit algorithm
22   commit_finish();
23   return retval;
24 }
```

Fig. 1. Code snippet of the MPI_Barrier wrapper.

original address, 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 [13]. 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, but support for the `MPI_Win_` family is on the roadmap of MANA. Details on 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 an MPI application may make copies of its objects, to be stored at arbitrary addresses. 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

We begin this section by enumerating the challenges faced in bringing the original MANA proof-of-principle closer to a production standard, capable of supporting C/R in production. The subsections following this discussion describe individual, novel algorithms that were introduced with MANA-2.0 to improve its reliability and performance.

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 process is in the middle of the MPI library. Hence, MANA uses wrapper functions to interpose between some MPI calls to `MPI_Send` and `MPI_Recv`, and then redirects to asynchronous calls like `MPI_Isend` and `MPI_Irecv`. As another example, MANA interposes calls to `MPI_Wait` and redirects to a loop around `MPI_Test`. These asynchronous calls are non-blocking, and so MANA can guarantee not to checkpoint a send/receive in the middle of a checkpoint. Extensions to `MPI_THREAD_MULTIPLE` are not considered here. Subsections III-A and III-B discuss how to handle the conversion of MPI point-to-point communications correctly.
2) The conversion to semantically equivalent MPI calls can result in higher runtime overhead. Subsection III-D shows an example of this type of runtime overhead. Nevertheless, semantic conversions are required for multiple reasons. The previous item gave an example, transforming `MPI_Send`/`MPI_Recv` to `MPI_Isend`/`MPI_Irecv`. In a similar example, `MPI_Wait` is converted to multiple calls to `MPI_Test`. And in some cases, an MPI call is converted to a POSIX system call, as in the conversion of `MPI_Alloc_mem`/`MPI_Free_mem` to malloc/free.
3) A conversion to semantically equivalent MPI calls, while valid for most MPI implementations, cannot be guaranteed for all MPI implementations. In particular, see [14], 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 [15, 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 Section III-E of how `MPI_Bcast` is supported, while avoiding potential deadlock.

4) While MANA can use its centralized coordinator as a side channel to communicate among the processes, 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. Subsections III-B and III-K are examples of this practice.
   For example, the original MANA needed to "drain" any point-to-point messages (`MPI_Send` and `MPI_Recv`). Previously, the DMTCP coordinator tracked the total number of bytes sent and received. In MANA-2.0, at the time of checkpoint, `MPI_Alltoall` is used to directly communicate among the MPI processes to determine if the number of bytes sent equals the number of bytes received (implying that there are no pending bytes in the network). While Subsection III-B discusses draining messages, this last point is covered at the end of Subsection III-M.
5) As part of the draining of point-to-point messages discussed in the last item, all MPI_Processes need a way to unambiguously identify a unique MPI process. This can be done by using `MPI_Translate_group_ranks` to translate the MPI rank within the current communicator to the MPI rank within the world communicator. MANA-2.0 takes care to internally use MPI calls that solely access local information, where possible, for the sake of efficiency. However, the overhead can be still sensitive to the particular MPI implementation. An implementation may choose to implement these MPI calls as "local" calls, or else it may choose a simpler implementation that invokes communication with a central MPI resource manager. Subsections III-B and III-K show how MPI calls are used internally in MANA-2.0.
6) MANA-2.0 improves on the original implementation that virtualizes MPI objects. For example, the original MANA virtualizes MPI communicators, and MANA-2.0 additionally virtualizes MPI requests. Virtualization of objects or IDs is a technique from *process virtualization* [16]. A given communicator, request or other object is associated both with a real ID (known within the MPI library) and with a virtual ID (stored within the MPI application memory). MANA does the translation when it interposes on any calls from the MPI application to the MPI library. Thus, if a checkpoint-restart occurs between the creation of the object and a second use, then the virtualized object can later be bound to a newly created real object on restart.
   However, when a real object is no longer referenced within the MPI library, this may be unknown to MANA, and MANA can be left with a growing list of stale virtualized objects. The size of that list continues to grow if it cannot be garbage-collected — resulting both in a growing memory footprint, and in higher overhead to access an object. Subsection III-A discusses details of virtualizing MPI requests.
7) The original MANA has a large runtime overhead, especially when used with applications with intensive collective communications. Subsections III-G, III-H, III-I, III-J, III-K and III-L focus on addressing the performance issues of the original MANA.
8) The original MANA supports Fortran MPI programs with a series of wrappers that translate MPI's Fortran bindings to MPI's C bindings. However, some corner cases are not handled correctly due to differences between the Fortran and C languages. III-F shows one example of such differences.

The following subsections discuss these and other issues that arose.

A. Virtualized requests

Resources like `MPI_Comm` and `MPI_Group` are allocated by MPI libraries. The resources must be virtualized to survive the checkpoint-restart barrier. But the `MPI_Request` resource was not virtualized in the original MANA. This issue was not seen in the original implementation [1] because non-blocking collective communications were not supported.

Compared to other virtualized resources, virtual MPI requests are generated so frequently that one must aggressively prune completed virtual MPI requests to avoid large performance and memory overhead. New virtual MPI requests are created in non-blocking MPI function wrappers, and retired in `MPI_Test` and `MPI_Wait` wrappers. Recall that the MPI library sets a request to `MPI_REQUEST_NULL` when the request has been satisfied. Similarly in MANA-2.0, when MANA wishes to "retire" or garbage-collect the virtualized form of an MPI request, it deallocates its request from its internal table, and sets the MPI request value in application memory to `MPI_REQUEST_NULL`.

An alternative implementation was suggested by one reviewer and is to be investigated. MANA-2.0 could internally use `MPI_Request_get_status` to non-destructively interrogate the MPI library. As a result, the application's MPI request status will be in a known (non-null) state, and so it will be safe to later internally call `MPI_Test` instead of directly setting an MPI request value to `MPI_REQUEST_NULL` in application memory.

MPI requests are used in two different cases: non-blocking collective communication and point-to-point communication. Because these two cases require different algorithm and data structures to support checkpoint/restart, virtualized MPI request retirements need to be supported differently for each case.

In the case of non-blocking collective communication, a log-and-replay algorithm is used to support checkpoint/restart. Upon a successful `MPI_Test` or `MPI_Wait`, virtualized MPI requests can be removed immediately from MANA's internal table without interfering with replaying on restart. Since the addresses of the tested MPI request is known in wrapper functions of `MPI_Test` and `MPI_Wait`, the MPI requests values in application's memory can simply be set to `MPI_REQUEST_NULL` directly.

The second use of MPI requests is to handle non-blocking point-to-point communication. In this case, values of virtual requests are saved in a list of active non-blocking calls, and addresses of MPI requests in the application's memory are unknown. Therefore, we cannot update the user application's memory to update the request to `MPI_REQUEST_NULL` directly.

Since we cannot directly update the user's memory in this case, a two-step retirement algorithm was developed to safely delete completed requests without requiring knowledge of the addresses where the user 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`. We can then safely remove the virtual request from the table and set the user application'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 comparable non-blocking versions, and used a variation of an all-to-all exchange algorithm to perform bookkeeping when draining 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 process sent the counted number of messages to the coordinator, and the coordinator sent the total number of messages to each process. 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 do any debugging to determine to which MPI rank a missing message might belong.

Therefore, MANA-2.0 improves the algorithm by using a smaller-grain message counter for each pair of process and sharing only essential information with `MPI_Alltoall`. After calling `MPI_Alltoall` at checkpoint time, all processes know, without further communication, how many bytes they were expected to receive and how many bytes they actually received. Locally, each process is able to use `MPI_Recv` to drain missing bytes from peers.

Another issue addressed in MANA-2.0 is that if `MPI_Recv` or `MPI_Irecv` has already been called, then the message may have already been received, in which case `MPI_Iprobe` can no longer detect the message in the network. Therefore, if some process 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` for which the corresponding request has not yet been satisfied. 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 no longer being used would be recreated during restart. In addition, MPI communicators could not be retired, in case they had been 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.

The new design in MANA-2.0 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. Not all collective communications are barriers: performance issues

In the MPI standard [13], there is no requirement in a collective function that all participating MPI processes must enter the function before any process can return. (For example, the "root" in `MPI_Bcast` can broadcast its message and return before other processes receive the message.) However, because of the two-phase-commit algorithm of the original MANA (see [1]), a barrier was added before each collective communication call. This was done to avoid checkpointing in the middle of collective calls, such as `MPI_Bcast`. However, this changes the semantics by making `MPI_Bcast` a blocking call. (See the next subsection for how MANA-2.0 restores the semantics of `MPI_Bcast` as a non-blocking call.)

The modified semantics incurs a major performance impact for collective communications. For example, adding a barrier before an `MPI_Bcast` forces the "root" process to wait until all other processes arrives. Generally the barrier makes the `MPI_Bcast` run two to three times slower. (See the next subsection for the solution in MANA-2.0.) Nevertheless, in the case of `MPI_Allreduce`, where all processes need to send and receive data from other processes, the barrier slightly improved the performance in our tests. Hence, note the recommendation of Cray for optionally adding a barrier to end-user code, and then testing with CRAYPAT to see if the performance improves [15, page 41].

E. Not all collective communications are barriers: correctness issues

In addition to the impact on performance, in some rare cases the added barrier can lead to a deadlock that did not

exist in the native MPI application. Assume a scenario where two MPI processes (rank 0 and rank 1) communicate with each other. Let rank 0 call `MPI_Bcast` as the "root" rank. In this scenario, rank 0 first calls `MPI_Send` and then calls `MPI_Bcast`. And rank 1 calls `MPI_Recv` and then calls `MPI_Bcast`. There is no deadlock when running natively, since rank 0 (the root rank) will not block on `MPI_Bcast`. However, if MANA adds a barrier before the `MPI_Bcast`, then rank 0 will wait for rank 1 to join the barrier, and rank 1 can only join the barrier after receiving in `MPI_Recv`. Yet, rank 0 has not yet called `MPI_Send`. Therefore, we have a deadlock.

This deadlock occurred in the original MANA because it converted the semantics of `MPI_Bcast` to a blocking call. To avoid the deadlock, MANA-2.0 must restore the non-blocking (but synchronizing) semantics of the MPI standard (see MPI-4, semantics appendix [14]). So, MANA-2.0 provides alternative wrapper implementations for `MPI_Bcast` and similar MPI calls, which use point-to-point communication (`MPI_Send` and `MPI_Recv`) instead of the real `MPI_Bcast` function in the lower half. (Note that the new persistent collectives that are part of the MPI-4 standard are outside the scope of this discussion.)

Normally, this will cause a performance degradation. Typically, the use of point-to-point communication compared to the standard `MPI_Bcast` implementation. However, a new hybrid two-phase-commit algorithm is currently being developed. The new algorithm only inserts the trivial barrier at checkpoint time. Similarly, the MANA wrapper function for `MPI_Bcast` in the new algorithm will use the standard `MPI_Bcast` before checkpoint. At checkpoint time, all calls to `MPI_Bcast` will use the alternative point-to-point implementation.

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 of unique storage locations in the underlying MPI library. Therefore, when using MANA with Fortran-based 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. (Note that Fortran 2018 has recently introduced a more direct way for Fortran and the C language to communicate Fortran constants.)

To identify these link-time named constants correctly in MANA's wrappers, we linked a small Fortran routine 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 constant, 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 runtime overhead is due to the use of the "FS" register. The split-process model of MANA [1] requires an upper-half program to do a context switch to a lower-half program. This requires MANA to modify the FS register. Unfortunately, that operation has been inordinately expensive (microseconds or more) due to the need to make a kernel call to the Linux kernel. Only recently, the unprivileged use of the FS register was enabled in Linux 5.9 [18]. Most HPC sites conservatively use older Linux kernels. For systems that cannot use the FSGSBASE Linux kernel patch, MANA-2.0 added a workaround to reduce the cost of using the "FS" register. For details, see [19].

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. The C++ compiler had compiled the lambda function of the source code into three or four additional call frames in the binary. 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.

I. Other sources of runtime overhead

Additional sources of runtime overhead are described in the public documentation of internals provided by MANA-2.0 [20]. 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 look up an entry in the map. In some cases, the original MANA also uses a linear search in the map. This can be reduced by employing a C++ map based on hash arrays.
2) Calls to disable and enable DMTCP checkpoint are used widely in MPI function wrappers. The cost of lock operations are too expensive because of their high frequency of use.
3) An internal helper method that translates the local rank of a communicator to a global rank makes multiple calls to the lower half. Calls to the lower half adjust the FS register, which is expensive (see Subsection III-G). This can be rewritten to make fewer calls.
4) We currently replay all non-blocking collective communications, like `MPI_Ibarrier`, `MPI_Ireduce` and `MPI_Ibcast`, in order to re-create virtualized requests. Not only is time wasted by creating completed requests, but this also increases the size of the virtual request table and slows the translation between real requests and virtual requests.

J. Stragglers

A *straggler* is an MPI process participating in a collective communication that may take minutes to hours to

join the collective communication, because it is finishing a CPU-intensive operation. 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 processes are still in the middle of a collective call in the lower-half MPI library.

The original algorithm used a two-phase commit algorithm for collective communication, in order to be able to transparently checkpoint without significant waits (see [1]). That algorithm inserted a barrier operation before every collective communication call, and was found to result in runtime slowdowns. A modified algorithm was introduced in a revised implementation. The modified algorithm assumed that there were no stragglers, but that modified algorithm was found to have some flaws. The current MANA-2.0 now includes a hybrid algorithm that adds a barrier operation before collective communication calls only after the DMTCP coordinator has requested a checkpoint. Additional details outside the scope of this paper are included in the MANA-2.0 documentation [20].

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 MPI processes participate in a given active communicator. This limits the ability of the MANA centralized coordinator to determine which MPI processes must stop and await the final checkpoint command, and which MPI processes 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 process 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 process must compute this globally unique ID *without the overhead of additional communication with its peers*. This is done using `MPI_Translate_group_ranks`. This function enables the MPI process 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 two-phase-commit algorithm (out of scope for this article)

To further improve the performance of the two-phase-commit algorithm, we have designed a hybrid version of the algorithm that removes the barrier before each collective communication. Additional details of this improved two-phase-commit algorithm are outside the scope of this paper; they are available in the MANA-2.0 documentation [20].

M. Lessons learned

There are some lessons learned from building the algorithms discussed above. First, additional communication by MANA should be minimized to optimize performance and correctness. When possible, MANA-2.0 uses MPI calls for internally sharing information among processes, instead of relying on MANA's centralized coordinator. Also when possible, MPI calls that complete based on local information are preferred over MPI calls requiring peer-to-peer communication. Minimizing the internal communication of MANA not only improves runtime performance, but also reduces the possibility of race conditions in the code, and helps to simplify debugging.

Another lesson learned in MANA-2.0 is that some MPI calls can be emulated with other MPI calls. This was described in greater detail in item 1 at the beginning of Section III.

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`); and 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 contains two types of compute nodes, dual-socket Intel Haswell and single-socket KNL nodes, interconnected with Cray Aries network. Each Haswell node has 32 cores (64 hardware threads) running at 2.3 GHz, and 128 GB DDR4 2133 MHz memory; each KNL node has 68 cores (272 hardware threads) running at 1.4 GHz, and 96 GB DDR4 2400 MHz memory. Cori runs Cray Linux environment version 7.0.UP01 with Linux kernel version 4.12. All experiments used Cori's burst buffer [21] for I/O, the most suitable file system for writing checkpoint images on Cori.

We evaluated MANA-2.0 using two commonly used applications at NERSC: GROMACS, a molecular dynamics code, and VASP, a materials science code. GROMACS (2021.02) was compiled with the Intel compiler (2019.3.199), and linked with Cray MPICH 7.7.10 and FFTW 3.3.8. 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 the Intel compiler (2019.3.199), and linked with Cray MPICH (7.7.10), MKL (2019.3.199) and FFTW (3.3.4) libraries.

Two branches of MANA-2.0 were used in the experiments: a relatively stable branch, the master branch, and a development branch, "feature/2pc". The "feature/2pc" branch has been extensively tested (and is hence more reliable), but has a high runtime overhead. We used the master branch in the checkpoint/restart experiments. The "feature/2pc" branch contains the latest improvements to runtime overhead, thus we used it in the runtime overhead tests. There are some issues to resolve in interface8 before merging it into the stable master branch, e.g., it fails to restart GROMACS using 2048 MPI processes. MANA is free and open-source software [22]. Documentation of the internals of MANA can be found at [20].

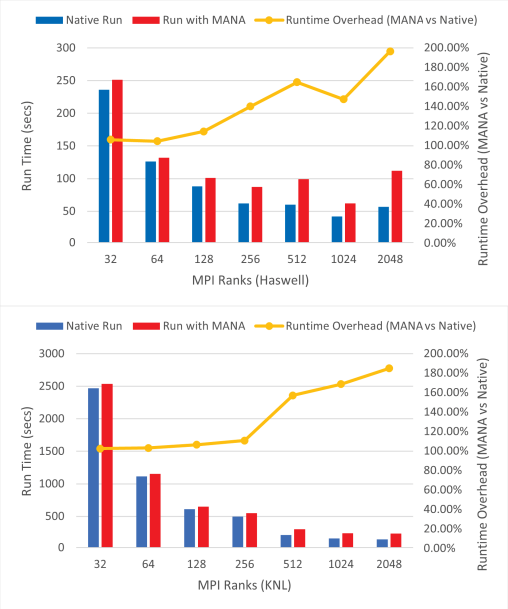

Fig. 2. Run time comparison between running GROMACS natively (blue bars) and under MANA (red bars) on Haswell (upper panel) and KNL (lower panel) nodes. The yellow line represents the run-time ratio between the MANA-enabled and native runs. Experiments were run on Cori Haswell and KNL nodes using 32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

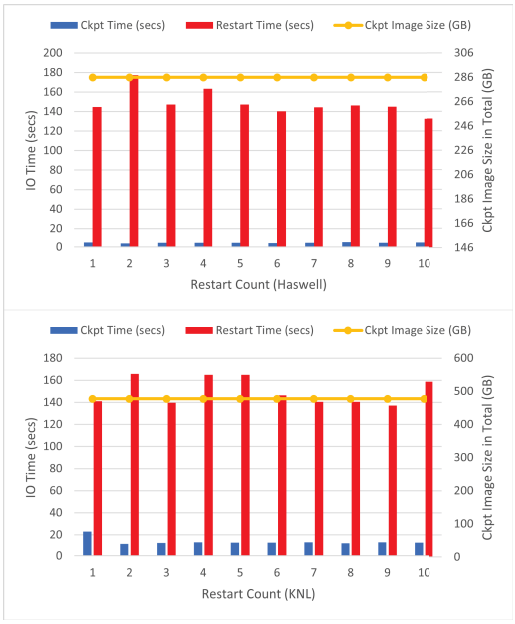

Fig. 3. Checkpoint/Restart overhead of MANA when running GROMACS with 2048 processes on Haswell (upper panel) and KNL (lower panel) nodes on Cori's Burst Buffer. The blue and red bars show the checkpoint and restart time, respectively; the yellow line indicates the total size of the checkpoint files.

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 system was studied in [23] by a NERSC user.

First, we evaluated the runtime overhead of MANA by running the benchmark using from 1 to 64 Haswell and KNL nodes (strong scaling) with and without MANA. We used the interface8 branch of MANA with commits present at the time of this writing. The interface8 branch includes mainly runtime overhead performance improvements.

The GROMACS run time was measured for 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 the MANA-enabled and native runs. One can see that for KNL runs, the runtime overhead is negligible except at 2048 processes. But for the Haswell architecture, the runtime overhead is still excessively high when running on more than two nodes, 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, MANA's checkpoint/restart capability was tested at scale. GROMACS was run with MANA using 2048 processes using 64 Haswell and KNL nodes (32 processes per node), respectively. The KNL runs were configured to use two OpenMP threads per process. The jobs were checkpointed at the 5-minute mark and terminated after 8 minutes (to assure sufficient time to write the checkpoint file), and then restarted. MANA was able to successfully checkpoint and restart GROMACS 10 times on each of Haswell and KNL.

Note that GROMACS does not scale well to an increasing number of MPI processes (see Figure 2). So using 2048 MPI processes to run the selected benchmark is not optimal. In fact, a high load imbalance was observed when running with 2048 MPI processes. This, however, does not affect our goal of demonstrating the scalability of MANA.

B. *VASP: a resource for robustness testing*

MANA-2.0 was tested with VASP, a materials science code. VASP is largest consumer of computer time on Cori 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 (Density Functional Theory) functional calculations using the RMM-DIIS [24] iteration scheme. Some test cases are real-world experiments, taken from the work of NERSC users.

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-2.0 can successfully checkpoint and restart all

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 master branch	MANA feature/2pc branch
Haswell	25s	41s	35s
KNL	69s	137s	101s

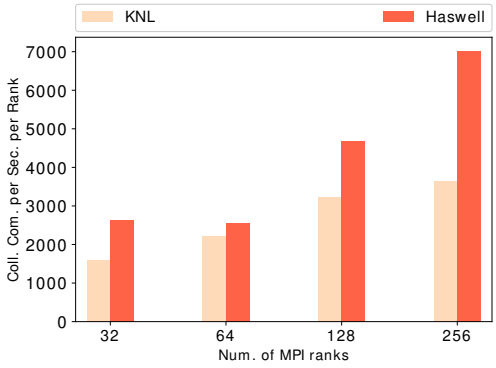

Fig. 4. Number of collective communications per second per process for VASP-5 on Haswell and KNL nodes. When doubling the number of ranks, the growth in the number of collective calls is roughly logarithmic in the number of nodes. This figure was cited from [19].

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., 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 the nature of its algorithms, VASP invokes an excessively high number of MPI collectives per second, as shown in Figure 4. This presents an additional challenge: runtime overhead.

C. MANA "Hybrid-2PC": initial successes in reducing runtime overhead

VASP was chosen to evaluate improvement in runtime overhead. We tested the CaPOH workload with 128 processes on both Haswell and KNL nodes. Table II shows the performances of the native VASP program, and VASP run under the MANA master branch and MANA experimental "feature/2pc" branch.

The MANA master branch focuses on scalability and stability; the "feature/2pc" branch is our first attempt to reduce runtime overhead. Currently, "feature/2pc" 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 has been reduced from 64% to 40%. On KNL nodes, the runtime overhead has been reduced from 99% to 46%. As discussed in Section III, there are additional known sources of runtime overhead. MANA-2.0 remains a work in progress, and efforts to solve each problem and reduce MANA's runtime overhead are continuing.

V. RELATED WORK

The history of MPI checkpointing 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 [7], [25], the MVAPICH2 checkpoint-restart service [6] — both of which temporarily disconnect the network and then delegate to BLCR [26] for checkpointing an individual process. Similarly, MPICH-V [4] disconnects a transport layer channel of MPICH (primarily based on TCP). It then delegates to the Condor package for checkpointing single-threaded individual processes [27].

Each of the above packages is implemented within a particular implementation of MPI. In contrast, the original DMTCP [5] (for TCP), and an InfiniBand plugin for DMTCP [8] 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 [28]. And CIFTS provides a fault-tolerant BLCR-based "backplane" [29].

MANA then introduced the *split-process model* for checkpointing of MPI [1]. Details are in Section II-A. The first work [1] demonstrated transparent checkpointing of GROMACS [12] and additional three applications at 2048 MPI

processes over 64 Haswell nodes. Efforts to deploy MANA at NERSC are described in [9], while MANA was previously updated for compatibility with the latest NERSC environment and to remove code specific to one environment, as described in [11]. The second work [11] demonstrated transparent checkpointing for 64 processes with GROMACS and 512 processes with the HPCG benchmark [30].

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

VI. Conclusion

This report on MANA-2.0 represents encouraging progress toward a robust, reliable package for transparent checkpointing that will be future-proof. There are important lessons from this work. 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 communication (translating `MPI_Send` to `MPI_Isend`, etc.); MPI collective communication (allowing each MPI process to proceed until all MPI processes have reached a safe point with no MPI process 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 processes must replay point-to-point and collective calls in order to re-instantiate vitual MPI requests for completion after restart.

MANA-2.0 is a significant improvement over the previous MANA in both scalability and reliability. It can checkpoint and restart GROMACS reliably, which uses MPI point-to-point communication intensively, using up to 2048 MPI processes (with the selected benchmark system). It can also checkpoint and restart the representative production workloads of VASP, which uses MPI collective communications intensively.

MANA-2.0 is still a work in progress. There are multiple issues to resolve before it can be used in production. Currently applications that invoke MPI collectives frequently, or applications that run at large scale, incur high runtime and memory overheads. Many causes have been identified, and fixes have been implemented or are under implementation.

In particular, MANA-2.0 need not be restricted to a standalone environment. It would be simple to extend MANA-2.0 to support the MPI-3.1 tools interfaces. This would give MANA-2.0 the ability to introspect into an MPI implementation. Hence, MANA-2.0 could play a supportive role within other fault-tolerant libraries. And the use of the tools interface could lighten MANA's current burden of indirect discovery through wrapper functions around MPI calls. The tools interface also represents an opportunity to provide a deadlock detector, as one more component in a general fault-tolerant ecosphere that includes such well-known packages as CRAFT [32] (Checkpoint/Restart and Automatic Fault Tolerance), SCR [33] (Scalable Checkpoint/Restart library), ULFM [34] (User-Level Failure Mitigation), and VeloC [35] (Very Low Overhead Checkpoint-Restart).

Acknowledgments

The authors thank two NERSC users, Giorgia Brancolini (Institute of Nanoscience CNR–Modena) and Daniel Pert (University of Michigan), for providing the benchmark cases for GROMACS and VASP (WOSiH), respectively, and for providing detailed descriptions of the systems. The authors also thank Chris Samuel at NERSC for valuable discussions and help, and Prashant Chouhan for his insights into the issues of allowing MANA to run at production scale. The authors wish to thank the reviewers for their insightful comments. They would especially like to thank the particular reviewer who provided many detailed suggestions for the future improvement of MANA-2.0. This work used the resources of the National Energy Scientific Computing Center (NERSC) at the Lawrence Berkeley National Laboratory.

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] Z. Zhao, and R. Hartman–Baker, "Checkpoint/Restart Vision and Strategies for NERSC's Production Workloads," Aug. 18, 2021. [Online]. Available: https://escholarship.org/uc/item/48v5r5rj

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

[10] "Top500 supercomputers (June, 2021)," https://www.top500.org/lists/top500/2021/06/, 2021, [Online; accessed Aug., 2021].

[11] P. S. Chouhan, H. Khetawat, N. Resnik, T. Jain, R. Garg, G. Cooperman, R. Hartman–Baker, and Z. Zhao, "Improving scalability and reliability of MPI-agnostic transparent checkpointing for production workloads at NERSC (extended abstract)," in *First International Symposium on Checkpointing for Supercomputing (SuperCheck'21)*, Berkeley, CA,

2021, pp. 1–3, https://arxiv.org/abs/2103.08546; from https://supercheck.lbl.gov/resources.

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

[13] 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

[14] Message Passing Interface Forum, "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

[15] 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

[16] K. Arya, R. Garg, A. Y. Polyakov, and G. Cooperman, "Design and implementation for checkpointing of distributed resources using process-level virtualization," in *IEEE Int. Conf. on Cluster Computing (CLUSTER'16)*. IEEE Press, 2016, pp. 402–412.

[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] Thom Holwerda, "Osnews: Linux 5.9 released," Oct. 12, 2020. [Online]. Available: https://www.osnews.com/story/132444/linux-5-9-released/

[19] Y. Xu, D. Hou, Z. Zhao, and G. Cooperman, "Removing kernel overhead in MANA's split-process model for checkpointing (document in preparation)."

[20] MANA team, "MANA plugin documentation," 2021. [Online]. Available: https://docs.google.com/document/d/1pT25gvMNeT1Vz4SU6Gp4Hx9MGLfkK8ZK-sSOwtu5R50

[21] "Cori's Burst Buffer," https://docs.nersc.gov/filesystems/cori-burst-buffer/.

[22] MANA team, "MANA software (refactoring branch)," 2021. [Online]. Available: https://github.com/mpickpt/mana/tree/refactoring

[23] G. Brancolini *et al.*, "The manuscript is under review," *JOURNAL OMITTED*, 2021.

[24] "RMM-DIIS," https://www.vasp.at/wiki/index.php/RMM-DIIS.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

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

[31] 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.

[32] F. Shahzad, J. Thies, M. Kreutzer, T. Zeiser, G. Hager, and G. Wellein, "CRAFT: A library for easier application-level Checkpoint/Restart and Automatic Fault Tolerance," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 3, pp. 501–514, 2018.

[33] A. Moody, G. Bronevetsky, K. Mohror, and B. R. De Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2010, pp. 1–11.

[34] I. Laguna, D. F. Richards, T. Gamblin, M. Schulz, B. R. de Supinski, K. Mohror, and H. Pritchard, "Evaluating and extending User-Level Fault Tolerance in MPI applications," *The International Journal of High Performance Computing Applications*, vol. 30, no. 3, pp. 305–319, 2016.

[35] B. Nicolae, A. Moody, E. Gonsiorowski, K. Mohror, and F. Cappello, "VeloC: Towards high performance adaptive asynchronous checkpointing at large scale," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019, pp. 911–920.