Implementation-Oblivious Transparent Checkpoint-Restart for MPI

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

Leonid Belyaev
Khoury College of Computer Sciences
Northeastern University
Boston, MA, USA
belyaev.l@northeastern.edu

Twinkle Jain
Khoury College of Computer Sciences
Northeastern University
Boston, MA, USA
jain.t@northeastern.edu

Derek Schafer
University of New Mexico
Albuquerque, NM, USA
dschafer1@unm.edu

Anthony Skjellum
College of Engineering
Tennessee Tech University
Cookville, TN, USA
askjellum@tntech.edu

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

ABSTRACT

This work presents experience with traditional use cases of checkpointing on a novel platform. A single codebase (MANA) transparently checkpoints production workloads for major available MPI implementations: "develop once, run everywhere". The new platform enables application developers to compile their application against any of the available standards-compliant MPI implementations, and test each MPI implementation according to performance or other features.

Since its original academic prototype, MANA has been under development for three of the past four years, and is planned to enter full production at NERSC in early Fall of 2023. To the best of the authors' knowledge, MANA is currently the only production-capable, system-level checkpointing package running on a large supercomputer (Perlmutter at NERSC) using a major MPI implementation (HPE Cray MPI). Experiments are presented on large production workloads, demonstrating low runtime overhead with one codebase supporting four MPI implementations: HPE Cray MPI, MPICH, Open MPI, and ExaMPI.

CCS CONCEPTS

• **Computer systems organization** → **Reliability**.

KEYWORDS

MPI, MPICH, Open MPI, ExaMPI, MANA, transparent checkpointing

SC-W 2023, November 12–17, 2023, Denver, CO, USA

ACM ISBN 979-8-4007-0785-8/23/11…$15.00
https://doi.org/10.1145/3624062.3624255

ACM Reference Format:
Yao Xu, Leonid Belyaev, Twinkle Jain, Derek Schafer, Anthony Skjellum, and Gene Cooperman. 2023. Implementation-Oblivious Transparent Checkpoint-Restart for MPI. In *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3624062.3624255

1 INTRODUCTION

A checkpointing library for MPI-agnostic checkpointing of MPI applications has been available for application-level checkpointing since at least 2003 [10, 43]. This allows an MPI developer to choose the best MPI implementation for that application. Similarly, some large application codes maintain their own code for saving and restoring data across a checkpoint. As with all application-level approaches, the developer has the burden of declaring and maintaining those data structures that must be preserved and restored across a checkpoint-restart.

However, many large, complex codes do not have their own application-level checkpointing. One large application that is typical of this constraint is VASP [24]. VASP accounted for approximately 20% of CPU time at the NERSC supercomputing center [37] as of 2020 [17, Figure 4]. VASP supports multiple algorithms and data structures that are continually evolving. As the VASP code evolves, it would be a large burden to continually update any application-specific module to reflect VASP's latest algorithms and data structures.

MANA [19] (MPI-Agnostic Network-Agnostic checkpointing) is open-source, and freely available at https://github.com/mpickpt/mana. MANA [19] has achieved production quality in recent testing, and is planned for production use at NERSC [37] in early Fall of 2023. To the best of the authors' knowledge, MANA is currently the only production-capable, *system-level* checkpointing package for MPI [47].

Yet, just as it would be a burden for each application developer to implement their own application-specific checkpointing for MPI, it is also a burden for each MPI implementation to support its own transparent checkpointing feature. This was attempted in the decade of the 2010s for MVAPICH and Open MPI (see details in Section 7). However, those modules required specific code to support each type of network being used by MPI. Eventually, the

need to support checkpointing of a proliferation of networks [5, 15, 26, 33, 34] caused the MPI developers to drop checkpoint support.

MANA's split-process approach (Section 2.2) created, for the first time, a transparent checkpointing package that was Network-Agnostic in the strong sense of not requiring any knowledge or code to support particular networks. Earlier MPI-specific checkpointing implementations relied on knowledge of each supported network, in order to shut down MPI's network connections prior to a checkpoint and then restart the network connections during restart. Hence, although Open MPI promised "Interconnect-Agnostic" checkpointing [27], it still required sufficient network code for stopping and starting each supported network.

In addition to transparent and application-level checkpointing, one should note the intermediate choice of library-based, application-specific checkpointing. For examples, see VeloC, SCR, FTI, ULFM, and Reinit in Section 7. These libraries are often used in conjunction with applications that support a particular execution model: a main loop, with each iteration globally synchronized by a call to an MPI barrier. At the synchronization point, there are no active MPI objects, and hence no MPI context needs to be saved.

The library-based and transparent varieties of checkpointing each make important contributions. First, transparent checkpointing will usually be preferred for applications that do not follow the execution model of a globally synchronized main loop. As stated, VASP accounts for about 20% of execution time at NERSC, and yet its multi-algorithm execution model conflicts with the model of a single main-loop often assumed by library-based packages.

Second, an MPI application may encounter bugs and crash at an arbitrary place in the code. The ability to take frequent checkpoints at arbitrary times aids in isolating the environment for analysis and replay, just prior to the crash.

Third, and most important for the future, the United States DOE (Department of Energy) has a goal of supporting large real-time computations, such as data processing due to an astronomical event, a high-energy particle accelerator, or responding to natural disasters. This requires preemptible jobs using system-level checkpointing on short notice (even minutes). An example of large real-time computations with a particularly extensive literature is X-ray scattering experiments using Free Electron Lasers (XFELs) [7, 8, 20]. Still other examples can be found in the workshop on *Interactive and Urgent HPC* (https://www.urgenthpc.com/). Library-based checkpointing using the main-loop execution model may not be able to reach the next globally synchronized iteration within the short notice. Quoting from [2, Section3.6] (Real-time scheduling):

> "In the next five years, we expect an increasing amount of our workload to come from short-notice, rapid-turnaround compute jobs from experiment facilities. Our system of reservations plus preemptible jobs is working well at relatively small scales - it is unclear whether this will scale to jobs that require a significant fraction of the machine. Work will be needed to appropriately incentivise a preemptible workload to fill the gaps in reservations. This will include increased use of (and support for) checkpointable applications. . . . A large pool of general workload that is preemptible (e.g., via user or perhaps system checkpointing) may be an option, but significant work is still needed to identify a technical solution."

1.1 MANA's New Implementation-Oblivious Virtual Ids: Supporting diverse MPI implementations

MANA's new production platform is now capable of supporting transparent checkpointing for any standards-compliant implementation of MPI, while maintaining low runtime overhead. Low overhead is essential for production MPI jobs. This work is based on a flexible design for virtual ids that is no longer targeted primarily toward HPE Cray MPI (based on MPICH [23]). This is demonstrated on the highly diverse MPI implementations by MPICH [23], Open MPI [22], and ExaMPI [44]. For details, see Section 2.2 for MANA's need for virtual ids, and Section 4 for a novel virtual-id design to implement a platform supporting arbitrary standards-compliant implementations of MPI.

As transparent checkpointing for MPI reaches ever wider use, it is important to accommodate multiple MPI implementations. Developers should not need to port MANA to work with each new major release of a MPI implementation. The existing MANA code is unfortunately directly tied to implementation decisions within HPE Cray MPI (hereafter called Cray MPI). This work concentrates on the novel implementation-oblivious production platform needed to bring that vision to fruition.

The ability of MANA developers to "develop once, run everywhere" impacts strongly on user support. As an example, MVAPICH [18] and Open MPI [27, 28] both used to include separate support for transparent checkpointing. Both MPI implementations were later forced to drop support for transparent checkpointing, due to the large developer burden of supporting an ever-increasing variety of network interconnects. By adopting a philosophy of "develop once, run everywhere", MANA hopes to avoid that trap.

MANA [19] demonstrated "MPI-Agnostic Network-Agnostic" checkpointing in an academic prototype in 2019. Since then, the academic prototype has been made fully robust, and is planned to enter production use at NERSC in early Fall of 2023. MANA's novel split-process design originally promised the "MPI-Agnostic" feature: the promise to *develop MANA once, and use MANA with every MPI implementation*. However, this feature was only tested in the original academic prototype, which concentrated on Cray MPI. An analysis of the original MANA internals shows that much of the original design was hardwired for Cray MPI.

This work describes a new platform for MANA that is MPI "implementation-oblivious", to distinguish from the more limited MPI-agnostic demonstration of the original academic prototype. The prototype was tested only on Cray MPI, with the exception of one experiment checkpointing under Cray MPI, and then restarting under Open MPI [19, Section 3.6]. That experiment ran a simple version of GROMACS [4], which was restricted to the MPI primitives (such as `MPI_COMM_World`), and did not create any new MPI objects — not even a new MPI communicator. Finally, when the academic

prototype was replaced by a production-quality version, even that limited version of MPI agnosticism was lost. (To understand better why the original GROMACS experiment [19, Section 3.6] cannot be extended to larger MPI codes, see Section 9.)

This work demonstrates a modified MANA that is truly "MPI-Agnostic". It is tested against Cray MPI, MPICH, Open MPI, and ExaMPI. The previous MANA work replaced the communicator, group, request, operation and datatype integer ids of Cray MPI and MPICH with virtual ids. The use of MANA-internal virtual ids allows MANA to re-bind the virtual ids to new ids in the MPI library during restart. However, the 32-bit virtual integer ids are not compatible with the 64-bit pointers and dynamically allocated MPI global constants used by Open MPI and ExaMPI.

In this work, a new "virtual-id" subsystem was designed to eliminate the multiple places where the original MANA design was hardwired to favor Cray MPI. Further, the new design is also intended to support subsets of the full MPI implementation (e.g., ExaMPI). For this purpose, a modified version of MANA was developed to (i) use the new virtual-id subsystem; and (ii) remove MANA's reliance on some MPI calls outside of a core subset. This work also identifies the core subset of MPI required for MANA support.

1.2 Points of Novelty

The novelty of this work is as follows.

(1) No currently used MPI implementation provides direct support for transparent checkpointing. A single, implementation-oblivious codebase now supports transparent checkpointing for Open MPI and ExaMPI, as well as continuing to support the MPICH family.
(2) This design makes MANA more flexible in supporting future MPI implementations. A virtual id is now a pointer to a MANA-internal struct that can flexibly point to MPI physical ids based on int, pointer, pointer to struct, or other datatypes for MPI ids. The MANA virtual id occupies the first 32 bits of any MPI id type declared in the `mpi.h` include file of a particular MPI implementation.
(3) A single virtual id that encodes any of the five MPI id types: communicator, group, request, operation, and datatype.
(4) The MANA virtual id stores additional MANA-internal information to adapt to future evolutions of the MANA architecture (e.g., the choice between record-replay of MPI objects during restart; or use of MPI functions to serialize a representation of the MPI object; or hybrids of the two strategies).

1.3 Organization of This Work

This work is organized into the following sections. Section 2 briefly describes the underlying split-process design of the original MANA, and then provides further details of how wrapper functions are used to translate MPI calls. Section 3 reviews the range of design choices made by different MPI implementations. Section 4 describes the new design of virtual ids, and how it supports multiple MPI implementations. Section 5 specifies the MPI subset support required of a particular MPI implementation, to be supported by MANA. The chosen MPI must support both MANA and the targeted application. (MANA itself supports most of the full range of MPI-3.0, except for MPI's one-sided communication.) Section 6 presents an experimental evaluation of the novel virtual-id design for MANA, while maintaining low runtime overhead. Section 7 presents related work. Section 8 is the conclusion, and Section 9 describes future work.

2 BACKGROUND

2.1 Design of MANA

MANA (MPI-Agnostic Network-Agnostic transparent checkpointing tool) is a previously developed package for checkpointing MPI applications [19]. A newer version, MANA-2.0 [47], has been developed for production use in supercomputing. MANA uses the idea of split processes.

Hence, the large family of MPI calls are passed to the actual MPI library: while maintaining efficiency; and while guaranteeing that no MPI process is blocked in a call to the lower half at the time of checkpoint.

2.2 Split processes

The most difficult challenge of MANA is to be able to checkpoint an MPI application, while not having to checkpoint the network libraries or operating system kernel modules for RDMA-based shared memory across computer nodes. The solution is a split-process strategy in which the MPI-based code is split into two parts:

upper half:	the MPI application itself; and
lower half:	the MPI library, along with the network library, kernel drivers, etc.

This approach is formalized using a split process architecture.

The *split-processes* architecture is summarized in Figure 1. In split processes, two programs are loaded into the address space of a single process. One program (upper half) is the MPI application (linked to a MANA library), and the second program (lower half) is a small MPI application that includes the actual MPI library. A MANA-internal library includes a stub function (wrapper function) for each MPI function, and each wrapper function calls one or more actual MPI functions in a small (lower-half) MANA-internal MPI program. On checkpoint, only the memory of the upper-half MPI application is saved. On restart, a new lower-half MPI program is launched, and that program restores the upper-half MPI application to its original location in memory.

The split process technique saves the memory of an MPI application (the upper half), while at the same time avoiding the need to save the memory of the MPI library. This is critical, since the MPI library intensively uses hardware associated with the network and possibly a high-performance network switch. Saving and restoring the state of the network switch, and possibly some kernel modules, is impractical, due to the difficulty of restoring the state of the associated hardware. The term "Network-Agnostic" in the MANA acronym (MPI-Agnostic Network-Agnostic checkpointing) is based

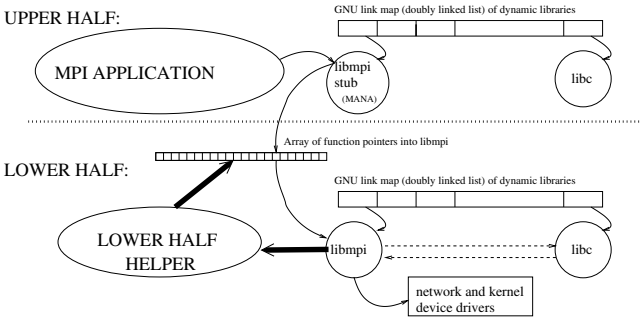

Figure 1: MANA implementation: split processes

on using split processes to make the design of MANA independent of the particular network hardware or software drivers.

2.3 Virtual Ids for Split Processes

A final piece of this puzzle is the problem of restoring the ids for MPI communicators, groups, and datatypes at the time of restart. In the split-process design, the MPI application is linked to a stub library, which calls the actual MPI library function in the lower half (see Figure 1). Upon creation of a communicator, group, request, operation, or datatype, a corresponding global MPI id is created by the lower-half MPI library and passed to the end-user code (the upper-half MPI application).

Upon restart, before returning control to the user's MPI application, MANA makes calls to the lower-half MPI library to create new copies of each of the communicators, groups, and datatypes that were present at the time of checkpointing. The difficulty here is that on restart, the MPI library will create new MPI ids that do not correspond to the original MPI ids at the time of checkpoint. Hence, any MPI ids saved and later restored inside the memory of the upper-half MPI application will no longer be valid.

The solution is to maintain a MANA-internal table of virtual MPI ids that are passed to the upper half and physical MPI ids that are maintained in the lower-half MPI library. Thus, in Figure 1, the stub function shown in the upper half refers to the MANA-internal table and translates from virtual id to physical id when making an MPI function call; and translates from physical id to virtual id when returning from the MPI function call.

The core issue in this work is to design a more general scheme for virtual-to-real translation of ids. The ids include MPI ids for communicator, group, request, operation, and datatype. During runtime, each MPI application can create new instances of ids.

3 DESIGN CHOICES MADE BY DIFFERENT MPI IMPLEMENTATIONS

This section reviews the range of design choices made by different MPI implementations.

Communicators/datatypes in Open MPI are 64 bits, for sake of a pointer. The pointer directly points to an internal struct with information about the communicator, group, request, operation or other.

The MPICH family of MPI implementations (including MPICH, MVAPICH, Intel MPI, and Cray MPI) uses a special 32-bit id to support a 2-layer table (similar to what is found in 2-layer page tables in operating systems): The MPICH id consists of: (i) bits representing whether it is a communicator, group, or other; (ii) first-level index into an internal table that then points to a second table; and (iii) the second-level index that accesses the actual struct representing the communicator, group, request, operation, or other MPI objects.

ExaMPI makes an unusual design choice. Primitive datatypes are defined in an enum class. Other types are defined as pointers to internal structs. The enum conflicts with the class template used by MANA. The new virtual-id subsystem is compliant with the ExaMPI design choices.

4 A NEW ARCHITECTURE FOR VIRTUAL IDS OF MPI OBJECTS

First, we present the motivation for why a new architecture is needed for MANA's virtual ids. In the following subsection, we present the new architecture that now supports MANA's MPI-oblivious feature.

4.1 Motivation

MANA's current design of virtual id's for MPI types like MPI_Comm is based on a series of C++ `std::map`. Each map represents one MPI type. This design has several drawbacks:

(1) All virtual ids are defined as `int`. It's okay when running with MPICH. However, other MPI implementations define MPI types as pointers to their internal structure representations. Using int as virtual id's will conflict with user applications linked with other MPI implementations.
(2) MANA's existing virtual ids use C++ associative arrays based on a string index. Repeated string comparisons add some small overhead.
(3) The virtual id table only translates between real and virtual ids without any other information. As a result, all data associated with a virtual id have to be stored in separate maps. When accessing multiple data related to the same virtual id, MANA needs to look up the same virtual id multiple times.
(4) MANA needs to replay functions that create communicators, groups, operations, requests, and datatypes on restart to update the real id's in the virtual id table.
(5) The virtual-to-real translation's performance depends on the map implementation of C++. The `std::map` class is O(logn) and the `std::unordered_map` is faster, at O(1). However, the real-to-virtual translation is always O(n) because MANA needs to iterate over all values of the map. (Real-to-virtual translation is used rarely — in just one MANA wrapper function.)

4.2 The New Virtual Id Architecture

To solve the problems mentioned above, a new data structure has been designed, based on a two-level table. Prior to this work, MANA maintained a simple integer for the virtual id, with a separate virtual id for each type of MPI object id. In the academic prototype of MANA [19], this was sufficient to support Cray MPI. But as MANA grew to production quality, independent data structures

had to be added, motivated solely by support for Cray MPI. Hence, MANA's initial promise of being MPI-agnostic was lost in the production development. A new virtual id architecture presented here now recovers that MPI-agnostic feature, and is here designated implementation-oblivious, to recognize the new architecture.

Each virtual id in the new design is represented by a structure that corresponds to an MPI communicator, group, request, operation, or datatype. This structure contains additional MANA-specific information associated with that MPI object. The MANA-specific information can be updated during normal execution. It is used to correctly save the state of MPI objects created by the lower-half MPI library.

The MANA virtual id consists of a 32-bit integer index into a table of the structures described earlier. The 32-bit index internally includes a *ggid* or "global group id" for communicators and groups, and a related index for request, operation, and datatype. The MANA stub functions of the upper half pass back to the MPI application the MPI object id (MPI_Comm, MPI_Group, etc.) as declared by the include file for the specific MPI implementation. MANA embeds its virtual id (the 32-bit integer) into the first 4 bytes of the MPI object type declared by the MPI include file. The integer is an index into a table of pointers to the structures. When the MPI application (compiled against the existing MPI include file) passes an MPI object as an argument, the MANA stub function recovers the 32-bit integer and uses this to call the lower-half MPI library with the corresponding "physical" MPI object id.

The new virtual-id structures are important to MANA at the time of checkpoint and restart. At the time of checkpoint, the structures may be further updated based on the current state of MPI objects. The structures are then saved as part of the checkpoint image of the upper half. MANA does not require a special data structure in the checkpoint image to identify these MANA-internal structures.

At the time of restart, MANA must create MPI objects that are semantically equivalent to the objects that existed prior to checkpoint. The MPI objects are re-created during restart. After restoring the upper-half memory, control is passed back to MANA in the upper half. For each MANA-internal structure representing an MPI object, MANA calls the MPI library in the lower half, in order to create a semantically equivalent MPI object. MANA then updates the internal structures to represent the new "physical " object id (of types MPI_Comm, MPI_Group, etc.) returned by the call to the MPI library of the lower half.

Note that a consequence of the design of MANA (regardless of virtual ids) is that MANA must be recompiled for each MPI implementation that it supports. This is required because the MANA stub functions (see Figure 1) make calls to the MPI library, and MANA itself may make internal calls to the MPI library. Hence, MANA must be recompiled against the MPI include file for the targeted MPI implementation. However, under the new virtual-id system, none of the code of MANA needs to be modified when a different MPI include file ("mpi.h") is substituted. Hence, MANA remains truly "MPI implementation-oblivious".

4.3 MPI Global Constants as functions

In the development of virtual ids to support MPI implementations outside the MPICH family, it was discovered that Open MPI implements certain global constants, such as MPI_COMM_WORLD, as macros that expand to functions. In the MPICH family, MPI_COMM_WORLD and other MPI constants expand to unique integers that are the same in the upper and lower half, and the same before checkpoint and after restart.

In Open MPI, it expands to a function call that returns a pointer, which may vary when it occurs in the upper-half MPI application (linked dynamically against MPI), or in the lower-half trivial application (linked statically against MPI). (See Figure 1 to review the MANA architecture.) Further, the value of MPI_COMM_WORLD for Open MPI can vary between its use prior to checkpoint and its use after restart.

ExaMPI adds a small additional requirement to MANA. In Open MPI, global MPI constants are determined at library startup. But ExaMPI defines the global constants using smart, shared pointers with reinterpret casts. This is done so that, for example, MPI_INT8_T and MPI_CHAR can share a pointer. But a consequence of this decision is that the address of a constant is known relatively late at runtime, on a lazy basis. MANA now accounts for this "lazy" design, when translating MPI global constants in the upper half to the correct lower-half form.

The situation is analogous to the case of MPICH's Fortran named constants (e.g., MPI_STATUS_IGNORE, MPI_MPI_IN_PLACE), which can vary from one session to the next [48, Section 3.3]. The solution in the case of MANA is to re-define the MPI_COMM_WORLD and other macros as a pointer to a lower-half array that contains the results of calling the corresponding functions.

5 MPI SUBSET REQUIREMENTS FOR "MPI-AGNOSTIC" MANA SUPPORT

MANA uses MPI functions internally to operate the network and to gather and sync MPI's runtime status among ranks. MANA is designed to be MPI implementation agnostic and network agnostic. Therefore, MANA cannot use lower-level network libraries for operations like draining messages in the network.

MANA requires three categories of MPI functions from MPI implementation to work properly:

(1) Functions that send, detect, and receive messages in the network. At checkpoint time, some point-to-point messages may still be pending in the network. MANA requires MPI_Iprobe to detect pending messages in the network, MPI_Recv, and MPI_Test to complete pending point-to-point communications.
(2) Functions that decode MPI objects that need to be reconstructed at restart time. MPI objects like communicator, operation, and datatype need to be reconstructed at restart time. Therefore, MANA needs the functions that decode such objects to obtain essential information for the reconstruction. Currently, MANA requires: MPI_Comm_group, MPI_Group_translate_ranks, MPI_Type_get_envelope, and MPI_Type_get_contents.

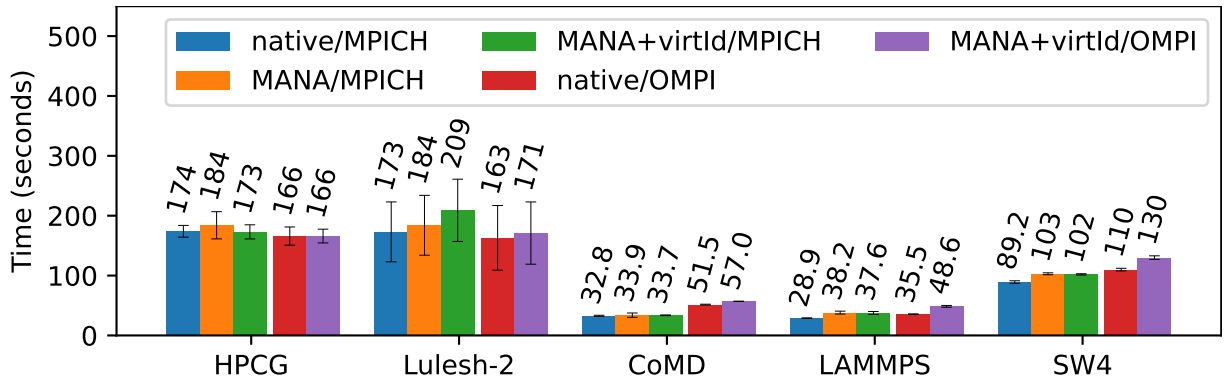

Figure 2: Application runtimes of MPI for MPICH versus Open MPI. A median of ten trials for each applicatoin was taken, and the standard deviation is shown. The five cases are: native, MANA, and MANA with virtual ids (virtId) for MPICH, and then native and MANA with virtual ids, for Open MPI.

(3) A small set of MPI communication functions used by MANA to share messages among processes, which includes: MPI_Send, MPI_Recv, and MPI_Alltoall.

6 EXPERIMENTAL EVALUATION

The experiments for MPICH, Open MPI and ExaMPI were run on a local HPC cluster called "Discovery", at Northeastern University, using Linux kernel 3.10. OpenMPI and ExaMPI both used intra-node TCP, while MPICH used intra-node shared-memory. The cluster was configured with CentOS 7.9 and Slurm 21.08.8-2. Jobs were run on Cascade Lake processors: Intel Xeon Platinum 8276 rated at 2.20GHz. Each node had 2 sockets, with a total of 56 cores per node (no hyper-threading). The MPI implementations were MPICH-3.3.2 (provided on the cluster); Open MPI 4.1.5 (built locally, since both dynamically and statically linked MPI are required to build MANA); and ExaMPI (git developer branch for August, 2023, requiring C++-20). The "mpicc" commands were based on gcc-10.1. Due to the experimental nature of ExaMPI, it was tested with a subset of applications known to be compatible. In all instances, MANA was built with gcc optimizations enabled. Timings were obtained through the use of SBATCH scripts and the Linux `date` utility – internal application timings were NOT used.

Section 6.4 contains runs on the Perlmutter supercomputer, using Cray MPI and dual-socket AMD EPYC 7763 CPUs. Cray mpicc is based on gcc-11.2. The Linux operating system is SUSE Linux Enterprise Server 15 SP4 (Release 15.4), with Linux kernel 5.14. The Perlmutter supercomputer is the #8 supercomputer, as of the TOP500 list of June, 2023 [46].

Note that due to the much older Linux-3.10 kernel (introduced in 2013) at the local site, the kernel does not support the FSGSBASE kernel feature (introduced in 2020). Hence, the MANA split processes required using a call to the kernel ("prctl(ARCH_SET_FS, ...)") to switch to a new "fs" register during any MPI function calls. The FSGSBASE feature allows recent kernels to support switching "fs" registers in user space with a single assembly instruction. The penalty of using the older "prctl" approach has been found to range anywhere from 3% to 30% or higher, depending on the frequency of MPI calls (calls to the lower-half program).

Laguna et al. [30] present a long list of MPI-based real-world applications. We selected five real-world applications with diverse MPI features, including ExaMPI compatibility. The selected applications include CoMD [35, 41], HPCG [16], LAMMPS [45], Lulesh-2.0 [29], and SW4 [42].

In the rest of this section, subsection 6.1 shows the use of MANA+virtId (the new virtual id design) on Open MPI. Subsection 6.2 shows the use of MANA+virtId on ExaMPI. Subsection 6.4 shows the use of MANA+virtId on Cray MPI on the Perlmutter supercomputer. We report runtime overhead for these cases. Subsection 6.5 shows trends for checkpoint time versus checkpoint image size.

Table 1 shows the inputs used for each of the applications of Figure 2.

App.	Ranks	Input
CoMD	$27 = 3^3$	`-N 10000`
HPCG	56	`--nx=104 --ny=104 --nz=104 --it=50`
LAMMPS	56	`-in bench/in.lj (run=50000)`
LULESH	$27 = 3^3$	`-p -i 100 -s 100`
SW4	56	`tests/curvimr/energy-1.in`

Table 1: Input for each application on a single node

6.1 Analysis of Open MPI versus MPICH

Figure 2 shows runtimes for each application in five cases:

(1) native/MPICH (no MANA)
(2) MANA/MPICH (the previous production version of MANA, using MPICH)
(3) MANA/MPICH with the new virtual-ids feature, described in 4
(4) native/Open-MPI (no MANA)
(5) MANA/Open-MPI with the new virtual-ids feature, described in 4

The MPICH implementation of MPI was highlighted since HPE Cray MPI is part of the MPICH family. HPE Cray MPI and MPICH

share much of their code. This makes possible a rough comparison of trends, by using MPICH as the "standard" for comparison on the local site, and HPE Cray MPI as the standard for production jobs on Perlmutter.

As seen in Figure 2, HPCG and Lulesh-2 have substantially more timing variation than the other applications even in native execution, which appeared to fall into clusters. As such, reasoning about the possible runtime overhead is more difficult – it is unlikely that MANA+virtId can improve HPCG performance while running with MPICH beyond the native execution. This work still illustrates the new capability to run these applications under non-MPICH MPI. Runtime overhead analysis later in this section is restricted to CoMD, LAMMPS, and SW4, although there exists some evidence to believe that the true runtime overhead with HPCG and Lulesh-2 is low (See Section 6.3).

In the cases of Lulesh-2 and SW4, a particular observation is pertinent. These programs are not built in the default manner (using MPI plus OpenMP). When running *natively* under MPICH, Lulesh-2 and SW4 consumed over 50% more time than native Open MPI. This was tracked down to the MPICH on Slurm 21.08.8-2 and CentOS 7.9 thrashing with too many OpenMP threads when run with these codes. As a workaround, these programs are built without OpenMP support. All Lulesh-2 and SW4 tests are reported in this mode.

We observe that while running with MPICH, the new virtId feature can improve performance by up to 1.6% (in the case of LAMMPS). We speculate that this is the result of an improved differentiation and access mechanism for virtual ids, which embeds binary tags in the 32-bit virtual ids to determine virtual id type inside a single map, instead of maintaining separate singleton maps for each type chosen between via macro-encoded string comparison. Especially when very many MPI calls are made, the time needed to look up a virtual id can become a significant factor.

We observe that while runtime overhead differs for each application (see Section 6.3), runtime overhead between Open MPI and MPICH in each application is comparable. However, the overhead under Open MPI tends to be greater than under MPICH. For instance, LAMMPS has a 37% runtime overhead under Open MPI, and 32% under MPICH. SW4 has a 18% runtime overhead under Open MPI, and 15% under MPICH. This may be the result of a difference in the speed of network calls causing more context switches to occur, e.g., while MANA internally calls `MPI_Test` while wrapping non-blocking communication.

6.2 Analysis of ExaMPI versus MPICH

Figure 3 demonstrates the ability of MANA to run with ExaMPI by using the new virtId feature. Recall that ExaMPI is intended as an experimental implementation that allows developers to *easily* experiment with new algorithms. Hence, rather than carry out research on implementing multiple, experimental new algorithms inside a production MPI, one can do the same experimentation more easily inside the smaller, C++-based ExaMPI codebase, and later port the best of the algorithms to a production implementation.

We build Lulesh-2 without OpenMP in order to be consistent with the methodology used for MPICH and Open MPI. Recall that native MPICH had unaccountably high overhead when built with OpenMP in the experiments of Section 6.1.

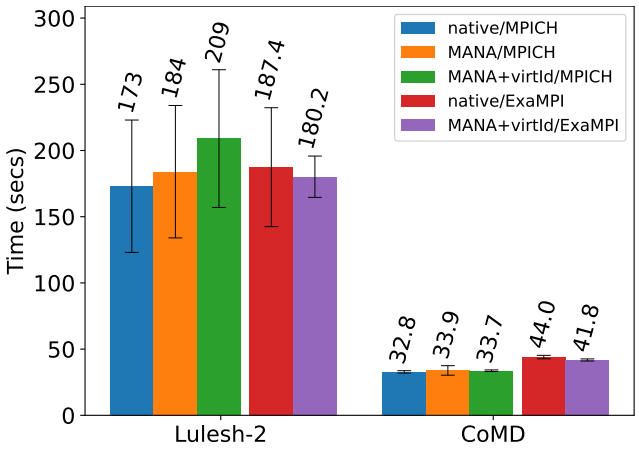

Figure 3: Runtimes for ExaMPI on Discovery. A median of ten trials for each application was taken, and the standard deviation is shown. The five cases of the legend are similar to that indicated in Figure 2.

These experiments show that MANA+virtId under ExaMPI can improve the runtime of CoMD by approximately 5% as compared to its native runtime under ExaMPI. It is speculated that the new virtId design may improve cache performance by producing greater code locality, or by caching some information that is otherwise re-computed in ExaMPI. We are continuing to investigate the exact cause of the improved performance with MANA+virtId.

6.3 Analysis of context switches per application

We observe that the runtime overhead measured at the local site for the five applications varies per application. There exists evidence to support the view that the body of this overhead variation is accounted for by the lack of the userspace FSGSBASE feature at the local site. Recall that this feature is a factor whenever MANA switches into the lower half (and back out of it) per the split-process architecture for MPI API wrappers.

We examined the amount of context switches performed by each application in a batch run of the graphed class, i.e., the same input and the same number of ranks as shown in Figure 2. Although the precise quantity of context switches can vary (as would be expected under network variability), it differs by as much as an order of magnitude between applications. In one experiment, CoMD made 3.7M CS/s under 27 ranks. Under 56 ranks, LAMMPS made the most switches: 22.9M CS/s. SW4 made 12.5M CS/s. This observation agrees with the relative differences in runtime overhead observed when running these applications under MANA at the local site. Recall that, per the split-process architecture, an application making more MPI calls results in MANA performing more context switches. Therefore, with more MPI calls, the lack of userspace FSGSBASE at the local site becomes a more significant factor.

HPCG and Lulesh were also tested for context switch rate: 4.7M CS/s and 1.3M CS/s respectively. This observation implies that the true runtime overhead for these applications is comparable to that of CoMD, or is yet lower.

6.4 Analysis of FSGSBASE runtime overhead for Cray MPI on Perlmutter

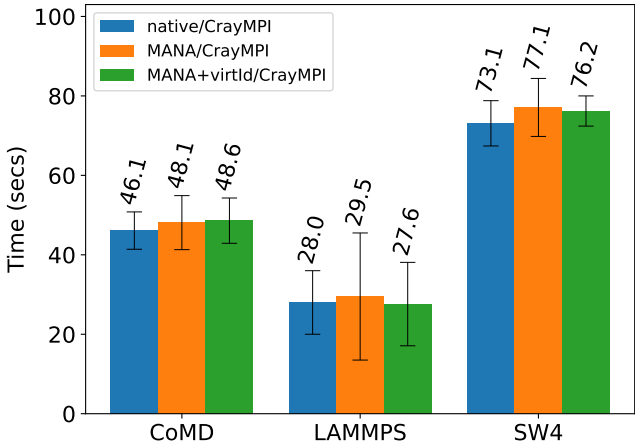

Figure 4: Runtimes for Cray MPI on Perlmutter. A median of twenty-five runs for each timing was used. The purpose is to show that on a production system supporting the FSGSBASE feature, both MANA and MANA+virtId do indeed perform comparably to the native execution. Error bars represent the standard deviation among 25 trials.

App.	Ranks	Input
CoMD	$64 = 4^3$	`-N 30000`
LAMMPS	64	`-in bench/in.lj (run=50000)`
SW4	64	`tests/curvimr/energy-1.in`

Table 2: Input for each application on Perlmutter

Figure 4 shows the results of testing CoMD, LAMMPS, and SW4 on Perlmutter, where the userspace FSGSBASE feature is present. Recall that Figure 2 showed LAMMPS and SW4 to scale poorly with MANA, especially with Open MPI, at the local site.

Figure 4 confirms that the high runtime overhead for LAMMPS and SW4 disappears when the userspace FSGSBASE feature is available. It's likely that the phenomenon was emphasized for SW4 and LAMMPS due to very frequent MPI calls for those particular applications.

Recall that, in our context switch measurements, LAMMPS had the highest context switch rate out of any application. Yet, the median runtime overhead across twenty-five trials is found to be 5.4% in the case of standard MANA, dramatically lower than the 32.2% observed on the local site. SW4 is measured to have an overhead of 5.5% with standard MANA, but this is improved to 4.2% with virtId. Improvement in median runtime was not observed under virtId with CoMD, as it was at the local site. These results show that virtId can potentially improve performance even with userspace FSGSBASE available, as in a production context.

6.5 Analysis of checkpoint times per application

On Discovery, the filesystem is based on NFSv3. Table 3 shows that checkpoint image sizes and checkpoint times follow similar trends. While it is dangerous to draw strong conclusions from the low-performance NFSv3 filesystem, it is clear that checkpoint times will continue to be modest on the high-performance filesystems of larger HPC sites.

Applic.	Ckpt size/rank	Ckpt time	MB/s/rank
CoMD	32MB	8.9	3.6
LAMMPS	42MB	12.8	3.3
SW4	49MB	12.3	4.0
Lulesh-2	207MB	16.3	12.7
HPCG	934MB	72.9	12.8

Table 3: Checkpoint times on Discovery

7 RELATED WORK

MANA [47] appears to be the only current production-ready package supporting transparent checkpointing of MPI.

The history of checkpointing of MPI is one that moved toward greater ease of use. Some milestones along the way include:

- application-level library-based checkpointing [10, 43] (2003, 2004);
- MPICH-V [9] (2006), based on a software framework at the lowest level of the MPICH [23] software stack;
- BLCR [25] and DMTCP [1] (2006, 2009) as lower-level tools for transparently saving and restoring process images;
- the MVAPICH2 checkpoint-restart service [18]) (2006), based on BLCR;
- the Open MPI checkpoint-restart service [27, 28] (2007, 2009), based on BLCR;
- an early attempt at MPI-agnostic checkpointing over InfiniBand [12] (2014), based on DMTCP (including the first petascale transparent checkpoints: 16,368 processes for NAMD and 32,368 processes for HPCG [11] (2016));
- VeloC [38] (2019) with its support for asynchronous, adaptive I/O in library-based checkpointing, as well as the earlier library-based tools: SCR [36] (2010), FTI [3] (2011), ULFM [6, 32] (2014), and Reinit [31] (2016, a simpler interface inspired by ULFM); and
- MANA [19] (2019) for network-agnostic, transparent checkpointing (but primarily focused on Cray MPI and the MPICH family of MPI implementations);

In the quest for a production-ready checkpointing package for MPI, VeloC [38] and MANA [47] are the two packages in greatest recent use (although SCR and ULFM/Reinit continue to see active use). The MVAPICH and Open MPI packages that were dependent on BLCR were sometimes used after they were developed. But BLCR does not support SysV shared memory, and so the MVAPICH and Open MPI checkpointing packages are no longer supported.

As noted in the introduction, the original and later work on MANA [13, 19, 47] concentrated mainly on Cray MPI, on the Cori

supercomputer at NERSC [37]. An exception to the use of Cray MPI was an example checkpointing under Cray MPI on Cori and restarting under Open MPI at a university cluster [19, Section 3.6]. That example was performed solely on GROMACS [4], and did not create any additional MPI ids beyond those that are built into MPI (such as MPI_COMM_WORLD).

Cray MPI [14, 21] and MVAPICH [39, 40] derive from the MPICH [23] family of MPI implementations. Open MPI [22] is an independently developed full production version of MPI. ExaMPI [44] is an experimental implementation of MPI designed to quickly test new algorithms, implementations, and features for MPI.

8 CONCLUSION

An implementation-oblivious feature was added to the production-quality MANA package for transparent checkpointing of MPI. This feature enables a philosophy of "developing once, and running everywhere": i.e., re-compiling MANA against the MPI include file of each new MPI implementation. This was demonstrated for three widely diverse MPI implementations: MPICH, Open MPI, and the experimental ExaMPI implementation. This was demonstrated with five real-world applications. A runtime overhead of about 5% or less was demonstrated — except where Linux lacked the userspace FSGSBASE feature and frequent MPI calls were made. A subset of the MPI standard was identified that is sufficient to support MANA on each of the three MPI implementations.

9 FUTURE WORK

Note that the MANA-internal structure can store a copy of the MPI object id directly in the MANA-internal structure. The MPI object is declared in an MPI-specific include file (i.e., `mpi.h`), which is different for each MPI implementation. Currently, the MANA structure saves the "physical" MPI object id directly on the MPI types found in the MPI include file, and MANA is re-compiled for the MPI include file of each separate MPI implementation. In future work, it may be possible to define a MANA version of the MPI include file, in which MANA's MPI types are defined to have sufficiently large size to contain any implementation-specific information.

This would recover interoperability between MPI implementations even at the level of checkpointing. One could checkpoint arbitrary MPI applications under one MPI implementation and restart under a different MPI implementation. This was previously accomplished only for an MPI application (a version of GROMACS) that used only the MPI primitives, but did not create MPI user-defined objects.

On a more modest level, we currently use an eager policy for computing the ggid (see Section 4.2 for new communicators. Because some codes repeatedly create and free communicators in a loop, we are considering the use of a lazy or hybrid policy.

ACKNOWLEDGMENTS

This work used the resources of the National Energy Scientific Computing Center (NERSC) at the Lawrence Berkeley National Laboratory. The work of the first author was partially supported by NSF Grant OAC-1740218. We are grateful for the collaboration of Zhengji Zhao, Rebecca Hartman and William Arndt of NERSC, along with the collaboration of MemVerge, Inc., for earlier bringing MANA to the production quality upon which the current work is based. We would especially like to highlight the generous donations of time by Zhengji Zhao that helped bring MANA into its production-ready phase at NERSC. We would also like to thank Kapil Arya for suggestions on how to work around a misfeature in MANA when taking timings for checkpoint.

REFERENCES

[1] Jason Ansel, Kapil Arya, and Gene Cooperman. 2009. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *2009 IEEE International Symposium on Parallel & Distributed Processing (IPDPS'09)*. IEEE, Rome, Italy, 1–12.

[2] Deborah Bard, Cory Snavely, Lisa Gerhardt, Jason Lee, Becci Totzke, Katie Antypas, William Arndt, Johannes Blaschke, Suren Byna, Ravi Cheema, et al. 2022. *The LBNL superfacility project report.* Technical Report. U.S. Department of Energy Office of Scientific and Technical Information (OSTI); and Lawrence Bekeley National Laboratory (LBNL).

[3] Leonardo Bautista-Gomez, Seiji Tsuboi, Dimitri Komatitsch, Franck Cappello, Naoya Maruyama, and Satoshi Matsuoka. 2011. FTI: High performance fault tolerance interface for hybrid systems. In *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis*. 1–32.

[4] H.J.C. Berendsen, D. van der Spoel, and R. van Drunen. 1995. GROMACS: A Message-passing Parallel Molecular Dynamics Implementation. *Computer Physics Communications* 91, 1 (1995), 43 – 56.

[5] Mark S Birrittella, Mark Debbage, Ram Huggahalli, James Kunz, Tom Lovett, Todd Rimmer, Keith D Underwood, and Robert C Zak. 2015. Intel® Omni-Path architecture: Enabling scalable, high performance fabrics. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. IEEE, 1–9.

[6] Wesley Bland, Aurelien Bouteiller, Thomas Herault, George Bosilca, and Jack Dongarra. 2013. Post-failure recovery of MPI communication capability: Design and rationale. *The International Journal of High Performance Computing Applications* 27, 3 (2013), 244–254.

[7] Johannes P Blaschke, Aaron S Brewster, Daniel W Paley, Derek Mendez, Asmit Bhowmick, Nicholas K Sauter, Wilko Kröger, Murali Shankar, Bjoern Enders, and Deborah Bard. 2021. *Real-time XFEL data analysis at SLAC and NERSC: a trial run of nascent exascale experimental data analysis.* Technical Report.

[8] Johannes P Blaschke, Felix Wittwer, Bjoern Enders, and Debbie Bard. 2023. How a Lightsource Uses a supercomputer for live interactive analysis of large data sets: Perspectives on the NERSC-LCLS superfacility. *Synchrotron Radiation News* (Sept. 2023), 1–7.

[9] Aurelien Bouteiller, Thomas Herault, Géraud Krawezik, Pierre Lemarinier, and Franck Cappello. 2006. MPICH-V project: A multiprotocol automatic fault-tolerant MPI. *The International Journal of High Performance Computing Applications* 20, 3 (2006), 319–333.

[10] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. 2003. Automated application-level checkpointing of MPI programs. In *Proc. of the Ninth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*. 84–94.

[11] Jiajun Cao, Kapil Arya, Rohan Garg, Shawn Matott, Dhabaleswar K. Panda, Hari Subramoni, Jérôme Vienne, and Gene Cooperman. 2016. System-level Scalable Checkpoint-Restart for Petascale Computing. In *22nd IEEE Int. Conf. on Parallel and Distributed Systems (ICPADS'16)*. IEEE Press, 932–941.

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

[13] Prashant Singh Chouhan, Harsh Khetawat, Neil Resnik, Twinkle Jain, Rohan Garg, Gene Cooperman, Rebecca Hartman–Baker, and Zhengji Zhao. 2021. 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, 1–3. https://arxiv.org/abs/2103.08546; from https://supercheck.lbl.gov/resources.

[14] Cray. 2014. Understanding Communication and MPI on Cray XC40. https://www.hpc.kaust.edu.sa/sites/default/files/files/public//KSL/150607-Cray_training/3.05_cray_mpi.pdf

[15] Daniele De Sensi, Salvatore Di Girolamo, Kim H McMahon, Duncan Roweth, and Torsten Hoefler. 2020. An in-depth analysis of the Slingshot interconnect. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–14.

[16] Jack Dongarra, Michael A Heroux, and Piotr Luszczek. 2016. A New Metric for Ranking High-performance Computing Systems. *National Science Review* (2016), 30–35. (benchmark at https://www.hpcg-benchmark.org/).

[17] Benjamin Driscoll and Zhengji Zhao. 2020. Automation of NERSC Application Usage Report. In *2020 IEEE/ACM International Workshop on HPC User Support Tools (HUST) and Workshop on Programming and Performance Visualization Tools (ProTools)*. IEEE, 10–18.

[18] Qi Gao, Weikuan Yu, Wei Huang, and Dhabaleswar K. Panda. 2006. Application-Transparent Checkpoint/Restart for MPI Programs over InfiniBand. In *Int. Conf. on Parallel Processing (ICPP'06)*. 471–478.

[19] Rohan Garg, Gregory Price, and Gene Cooperman. 2019. MANA for MPI: MPI-Agnostic Network-Agnostic Transparent Checkpointing. In *Proc. of the 28th Int. Symp. on High-Performance Parallel and Distributed Computing*. ACM, 49–60.

[20] Anna Giannakou, Johannes P Blaschke, Deborah Bard, and Lavanya Ramakrishnan. 2021. Experiences with cross-facility real-time light source data analysis workflows. In *2021 IEEE/ACM HPC for Urgent Decision Making (UrgentHPC)*. IEEE, 45–53.

[21] Richard L Graham, George Bosilca, and Jelena Pješivac-Grbovic. 2007. A Comparison of Application Performance Using Open MPI and Cray MPI. *Cray Users Group (CUG'07)* (2007), 10 pages.

[22] Richard L Graham, Timothy S Woodall, and Jeffrey M Squyres. 2006. Open MPI: A flexible high performance MPI. In *Parallel Processing and Applied Mathematics: 6th International Conference, PPAM 2005, Poznań, Poland, September 11-14, 2005, Revised Selected Papers 6*. Springer, 228–239.

[23] William Gropp and Ewing Lusk. 1996. User's guide for MPICH, a portable implementation of MPI.

[24] Jürgen Hafner. 2008. Ab-initio simulations of materials using VASP: Density-functional theory and beyond. *Journal of computational chemistry* 29, 13 (2008), 2044–2078.

[25] Paul H Hargrove and Jason C Duell. 2006. Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters. *Journal of Physics: Conference Series* 46, 1 (2006), 494.

[26] Hewlett Packard Enterprise. 2017. Aries High-Speed Network. https://pubs.cray.com/bundle/Urika-GX_Hardware_Guide_H-6142_Rev_C_Urika-GX_HW_Guide_DITAval/page/Aries_High_Speed_Network_Urika-GX.html

[27] Joshua Hursey, Timothy I Mattox, and Andrew Lumsdaine. 2009. Interconnect agnostic checkpoint/restart in Open MPI. In *Proc. of 18th ACM Int. Symp. on High Performance Distributed Computing*. 49–58.

[28] Joshua Hursey, Jeffrey M Squyres, Timothy I Mattox, and Andrew Lumsdaine. 2007. The design and implementation of checkpoint/restart process fault tolerance for Open MPI. In *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 1–8.

[29] Ian Karlin, Jeff Keasler, and J Robert Neely. 2013. *Lulesh 2.0 updates and changes*. Technical Report. Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States).

[30] Ignacio Laguna, Ryan Marshall, Kathryn Mohror, Martin Ruefenacht, Anthony Skjellum, and Nawrin Sultana. 2019. A large-scale study of MPI usage in open-source HPC applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[31] Ignacio Laguna, David F Richards, Todd Gamblin, Martin Schulz, Bronis R de Supinski, Kathryn Mohror, and Howard Pritchard. 2016. Evaluating and extending User-Level Fault Tolerance in MPI applications. *The International Journal of High Performance Computing Applications* 30, 3 (2016), 305–319.

[32] Nuria Losada, Patricia González, María J Martín, George Bosilca, Aurélien Bouteiller, and Keita Teranishi. 2020. Fault tolerance of MPI applications in exascale systems: The ULFM solution. *Future Generation Computer Systems* 106 (2020), 467–481.

[33] Ping-Jing Lu, Ming-Che Lai, and Jun-Sheng Chang. 2022. A survey of high-performance interconnection networks in high-performance computer systems. *Electronics* 11, 9 (2022), 1369.

[34] Mellanox Technologies. 2015. RDMA Aware Networks Programming User Manual (Rev 1.7). https://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf

[35] Jamaludin Mohd-Yusof, Sriram Swaminarayan, and Timothy C Germann. 2013. Co-design for molecular dynamics: An exascale proxy application. *LA-UR 13-20839* (2013), 88–89.

[36] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R De Supinski. 2010. 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, 1–11.

[37] NERSC [n. d.]. NERSC, the primary scientific computing facility for the Office of Science in the U.S. Department of Energy. https://nersc.gov/.

[38] Bogdan Nicolae, Adam Moody, Elsa Gonsiorowski, Kathryn Mohror, and Franck Cappello. 2019. VeloC: Towards high performance adaptive asynchronous checkpointing at large scale. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 911–920.

[39] Dhabaleswar Kumar Panda, Hari Subramoni, Ching-Hsiang Chu, and Mohammadreza Bayatpour. 2021. The MVAPICH project: Transforming research into high-performance MPI library for HPC community. *Journal of Computational Science* 52 (2021), 101208.

[40] Dhabaleswar K Panda, Karen Tomko, Karl Schulz, and Amitava Majumdar. 2013. The MVAPICH project: Evolution and sustainability of an open source production quality MPI library for HPC. In *Workshop on Sustainable Software for Science: Practice and Experiences, held in conjunction with Int'l Conference on Supercomputing (WSSPE)*. 5 pages.

[41] Massimo Papa, Toshiki Maruyama, and Aldo Bonasera. 2001. Constrained molecular dynamics approach to fermionic systems. *Physical Review C* 64, 2 (2001), 024612.

[42] N Anders Petersson and Björn Sjögreen. 2015. Wave propagation in anisotropic elastic materials and curvilinear coordinates using a summation-by-parts finite difference method. *J. Comput. Phys.* 299 (2015), 820–841.

[43] Martin Schulz, Greg Bronevetsky, Rohit Fernandes, Daniel Marques, Keshav Pingali, and Paul Stodghill. 2004. Implementation and evaluation of a scalable application-level checkpoint-recovery scheme for MPI programs. In *SC'04: Proc. of the 2004 ACM/IEEE Conf. on Supercomputing*. IEEE, 38–38.

[44] Anthony Skjellum, Martin Rüfenacht, Nawrin Sultana, Derek Schafer, Ignacio Laguna, and Kathryn Mohror. 2020. ExaMPI: A modern design and implementation to accelerate Message Passing Interface innovation. In *High Performance Computing: 6th Latin American Conference, CARLA 2019, Turrialba, Costa Rica, September 25–27, 2019, Revised Selected Papers 6*. Springer, 153–169.

[45] Aidan P Thompson, H Metin Aktulga, Richard Berger, Dan S Bolintineanu, W Michael Brown, Paul S Crozier, Pieter J in't Veld, Axel Kohlmeyer, Stan G Moore, Trung Dac Nguyen, et al. 2022. LAMMPS-a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales. *Computer Physics Communications* 271 (2022), 108171.

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

[47] Yao Xu, Zhengji Zhao, Rohan Garg, Harsh Khetawat, Rebecca Hartman-Baker, and Gene Cooperman. 2021. MANA-2.0: A future-Proof design for transparent checkpointing of MPI at scale. https://ieeexplore.ieee.org/document/9721343; technical report at https://arxiv.org/abs/2112.05858. In *Int. Symp. on Checkpointing for Supercomputing (SuperCheck'SC-21), 2021 SC Workshops Supplementary Proceedings* (St. Louis, MO). IEEE, 68–78.

[48] Junchao Zhang, Bill Long, Kenneth Raffenetti, and Pavan Balaji. 2014. Implementing the MPI-3.0 Fortran 2008 binding. In *Proceedings of the 21st European MPI Users' Group Meeting*. 1–6.