Transparent High-Speed Network Checkpoint/Restart in MPI

Julien Adam
ParaTools SAS
Bruyères-le-Châtel, France

Jean-Baptiste Besnard
ParaTools SAS
Bruyères-le-Châtel, France

Allen D. Malony
ParaTools Inc.
Eugene, USA

Sameer Shende
ParaTools Inc.
Eugene, USA

Marc Pérache
CEA, DAM, DIF
F-91297 Arpajon, France

Patrick Carribault
CEA, DAM, DIF
F-91297 Arpajon, France

Julien Jaeger
CEA, DAM, DIF
F-91297 Arpajon, France

ABSTRACT

Fault-tolerance has always been an important topic when it comes to running massively parallel programs at scale. Statistically, hardware and software failures are expected to occur more often on systems gathering millions of computing units. Moreover, the larger jobs are, the more computing hours would be wasted by a crash. In this paper, we describe the work done in our MPI runtime to enable transparent checkpointing mechanism. Unlike the MPI 4.0 User-Level Failure Mitigation (ULFM) interface, our work targets solely Checkpoint/Restart (C/R) and ignores wider features such as resiliency. We show how existing transparent checkpointing methods can be practically applied to MPI implementations given a sufficient collaboration from the MPI runtime. Our C/R technique is then measured on MPI benchmarks such as IMB and Lulesh relying on Infiniband high-speed network, demonstrating that the chosen approach is sufficiently general and that performance is mostly preserved. We argue that enabling fault-tolerance without any modification inside target MPI applications is possible, and show how it could be the first step for more integrated resiliency combined with failure mitigation like ULFM.

KEYWORDS

Checkpoint-Restart, Fault-Tolerance, DMTCP, Infiniband

ACM Reference format:
Julien Adam, Jean-Baptiste Besnard, Allen D. Malony, Sameer Shende, Marc Pérache, Patrick Carribault, and Julien Jaeger. 2018. Transparent High-Speed Network Checkpoint/Restart in MPI. In *Proceedings of 25th European MPI Users' Group Meeting, Barcelona, Spain, September 23–26, 2018 (EuroMPI '18),* 11 pages.
DOI: 10.1145/3236367.3236383

EuroMPI '18, Barcelona, Spain

DOI: 10.1145/3236367.3236383

1 INTRODUCTION

The trend towards parallel high-performance computing (HPC) systems with extreme numbers of cores, deep memory hierarchies, and multi-dimensional topological networks is pushing application developers towards programming models that must take advantage of nodes executing a large number of threads, while also maintaining efficient inter-node communication. The evolution to *hybrid* programming models consequently results in parallel applications that are effectively operating multiple runtime systems simultaneously to carry out its computation. The common MPI+OpenMP approach is an example. In such a context, it is reasonable to allow programming models to collaborate when performing some runtime actions.

Consider the objective of *checkpoint/restart* (C/R) as a fault-tolerance mechanism aimed at saving the current state of a given parallel program's execution (i.e., *checkpoint*) and then restoring the program's status at that point (i.e., *restart*). There are multiple methods to realize the results, both *explicit* (requiring direct modifications in the code), and others *transparent* (in the sense that they are able to checkpoint indifferently from the code itself). However, in the case of a hybrid application, especially one that is executing in a HPC environment, we want to exploit potential C/R optimizations through leveraging hybrid runtime support.

Our paper introduces a general method for transparent checkpointing of hybrid applications that can take advantage of high-speed hardware, especially high-speed networks such as Infiniband. The method is based on the *distributed multi-threaded checkpointing* (DMTCP)[1] concept integrated in a thread-based MPI runtime. In contrast to related research work, our contributions allow checkpoint/restart on demand with minimum effort and reduced performance overhead, including when compared to other transparent models such as IB Verbs wrapping. Moreover, we demonstrate that performance advantage is transitive (unlike alternate methods) in that it generalizes to any network. The paper concludes with a summary of our results and prospects for future work.

2 RELATED WORK

Fault tolerance in the context of HPC applications is a very active field. The increasing complexities and constraints on parallel systems, combined with falling *mean time between failure (MTBF)* on systems with millions of components, motivates the development

of technology to mitigate the consequence of failures during parallel execution. Such failures directly map to lose simulation results, but also the financial cost of a highly priced resource (including power). Beyond fault tolerance, these technologies can also benefit other purposes, such as steering of a parallel application to improve solutions or remapping system resources to address allocation constraints on a given machine. With respect to MPI applications in general, we can identify three main approaches for fault tolerance: (1) explicit and (2) transparent approaches, followed by (3) failure mitigation. Although these are not mutually exclusive, we describe each in turn.

2.1 Explicit Methods

The checkpoint/restart methodology is about both saving and restoring the state of a program. When it comes to parallel applications, this supposes that a program (e.g., a simulation) is able to restore its state (data) and current time-step (control) to take over the computation from where it was checkpointed. The most basic way to achieve this behavior is to manually save data associated with a given time-step and reload it again to restart it, this being done by the program itself. In this manner, results from multiple intermediate time-steps can be saved and reloaded. This is a portable method which has the advantage of not requiring any external tool. The application describes which data has to be saved and the resultant checkpoint file contains exactly what is needed for restart while the program interruption time remains low, keeping a small overhead for the overall application execution time. One step further is to consider checkpoint file storage in a redundant manner. An easy way is to store files on a shared mount point. However, this approach exposes issues when scaling to thousands of nodes/processes. SCR[24] and ACR[25] answer this by storing checkpoint files over faster, local mount points and replicate them to ensure redundancy.

Unfortunately, the basic approach has further limitations. First, it requires that the full dataset remain easily serializable, and supposes that all the artifacts linked to a given computation state are preserved and restorable. This can be a cumbersome task when dealing with highly modular frameworks hosting several data structures. Second, it supposes that each simulation implements its own checkpoint format and dedicate development efforts to provide a similar feature. While it is possible to leverage external libraries that optimize certain support (e.g., FTI[3] and MPI-IO[17, 29]), application-level checkpointing still requires representative data to be manually described using a dedicated API. As a consequence, they cannot be seen as transparent, as the target code still has to insert calls to the checkpointing API. For these reasons, methodologies which do not involve such annotations, have also been explored.

2.2 Transparent Methods

Transparent checkpointing tries to save the state of a running program, without having any previous application knowledge. Several tools have been developed for this purpose, leveraging multiple approaches. A general "external" method utilizes a virtual machine (VM) running inside an emulator, which can be frozen and then saved (both from memory and disk point of view)[4, 16]. While effective, it requires the whole operating system to be saved, and has severe performance overhead.

Tools for checkpoint/restart that are more appropriate for the HPC field include the *Berkeley Lab Checkpoint Restart (BLCR)* [18] tool. BLCR relies on a kernel-level approach to both suspend and checkpoint. This has the advantage of avoiding a complete wrapping of every syscall, and thus avoids the associated overhead. Being part of the Linux kernel also give the advantage to restart applications in the exact same UNIX environment (same process ID) and open pipes between them. However, the kernel approach first requires an administrator to load the corresponding module. Without considering resources outside of the current OS, it is not possible to save/restore network communication like sockets, and the application will have to handle these limitations to provide a complete C/R support. As multiple patched kernels are not able to communicate through a whole cluster, BLCR, on its own, cannot be used by itself for MPI purposes. MPI applications have to integrate explicit BLCR support to enable its distributed usage. In particular, an approach using BLCR similar to what we present in this paper has been developed with the idea of closing network ressources before checkpointing [9]. However, it was limited to TCP protocol but considered emulation on high-speed networks.

Another approach consists in providing checkpointing in user-space by wrapping any needed system calls, in order to constantly track application states. Because tools cannot be sure about the application behavior, all potential calls involving resources outside the process, need to be captured, such as network or storage. The Distributed Multi-threaded CheckPointing (DMTCP) [1] tool can checkpoint applications at user-space level, injecting a preloaded shared library upon application start in order to wrap system calls. Such a tool has the advantage of not requiring recent kernel features or administrative privileges for installation or recompiling the application to enable, disable or update the support. From this viewpoint, it becomes easier to make multiple nodes collaborate, and checkpointing distributed applications does not necessarily need MPI-aware implementations. However, catching system calls and associated bookkeeping creates a measurable performance overhead. Moreover, a log of on-the-wire messages has to be preserved in order to replay them in case of a failure. Such a model introduces a non-negligible cost for the application.

The last method allows transparent checkpointing without wrapping system calls, as done by tools such as CRIU [13]. However, it relies on more recent kernels to be able to fully extract information from the operating system. CRIU has the advantage of supporting name-spaces and is, therefore, the solution of choice when dealing with containers.

As far as MPI support is concerned, only DMTCP and BLCR currently integrate a mechanism to enable a distributed checkpoint involving multiple UNIX processes. For this reason, and due to test environment constraints (i.e., kernel), the solution we will develop in the rest of this paper relies on DMTCP, but CRIU is recognized as a promising future alternative particularly as it does not create additional overhead due to wrapping.

2.3 Failure Mitigation

The failure mitigation approach is more focused on the way to identify and put up with a failure than actually on how to recover from it. For example, if some nodes suffer from a hardware failure during a MPI job, it would be faster for the application to recover from remaining MPI processes than restarting the whole program (reallocating resources)[14, 20]. If the workload can be adjusted dynamically, such approaches are bound to be more efficient than pure C/R. In this field, we can cite the User-Level Failure Mitigation[6, 7] (ULFM), a solution implemented on top of OpenMPI, providing new MPI semantics that help the application to recover process failures. This model defines a *state* at communicator level. If at least one MPI process becomes unreachable – for any reason defined by the implementation – the MPI call returns an error. In addition, ULFM provides routines to revoke and shrink communicators in order to recover from failures. This approach can be made straightforward by attaching an "error-handling" routine to the MPI interface, somehow analogous to signal handlers on UNIX systems, they allow a given program to react appropriately to a failure. ULFM is therefore an MPI toolbox for resiliency in MPI context, and should be seen as complementary approach to C/R. One drawback of such interface is that it is still up to the application to implement the part of the code dedicated to failure mitigation[8, 15, 28].

2.4 Discussion

From a general point of view, transparent methodologies have the drawback of saving more than needed for a given execution. Indeed, it is not compulsory to save internal runtime states to restore a given simulation. Nonetheless, in some cases, it may not be sufficient to solely rely on data restore. For example, a given computation may use data types which are solely created during program startups. As a consequence, a program based on application-level checkpointing also has to go through an initialization phase of some form, to restore pertinent resources. One advantage of transparent restart methods is that it abstracts these dependencies due to its exhaustive nature and spans to all the libraries.

3 CONTRIBUTION

In this paper, we leverage the DMTCP checkpointing tool to transparently save the state of an MPI program. In particular, we show how the MPI runtime can collaborate with the checkpointing tool to enable support for high-speed networks. Indeed, when using specialized networking hardware, such as Infiniband (IB), care must be taken with respect to initialization and handling of dedicated objects like queue pairs. Moreover, even if part of this context is saved in a transparent checkpoint, restarting must avoid errors that could occur by launching the program on an un-initialized Host-Channel Adapter (HCA). We propose to leverage a dedicated modular network management infrastructure developed in the MPI runtime to both reset and initialize networks on the fly to enable such checkpoints. The paper makes the following contributions:

- The definition of a collective checkpoint interface enabling transparent checkpointing in MPI runtimes (Section 4);
- The concept of an in-band signaling network with the associated routing, and the use of multi-rail logic to enable partial checkpointing (Section 5);
- A general MPI implementation of transparent checkpointing including high-speed networks (Section 6).

Our work has been implemented in the MPC thread-based MPI runtime, although it is applicable to any MPI implementation, as we will describe. What makes MPC particularly challenging is that we needed to manage transparently the checkpointing of multiple *runtime stacking* configurations that MPC supports. Indeed, because MPC is built on user-level threading system, not only does our approach track process-based MPI, but it can accommodate any type of thread-based MPI, including user-level threads in MPC. Thus, this demonstration in MPC gives us confidence that the methodology will translate well to future evolutions of MPI, including those supporting the concepts of endpoints [12] and sessions [19], which involve intra-process parallelism.

Given the MPC infrastructure, we present a general methodology enabling checkpoint/restart for programs relying on high-speed networks. More precisely, we detail how the MPC runtime is able to dynamically open and close communication rails through a two-level checkpoint infrastructure. Such an approach provides MPI runtime with the ability to be checkpointed, and transitively applications to benefit from this feature. Moreover, we show that this approach incurs a reduced performance overhead.

4 CHECKPOINTING INTERFACE

From an end-user's point of view, this paper defines a new MPI collective function call, whose role is to realize a transparent checkpoint. Furthermore, we define a set of constants linked to the state of the parallel program:

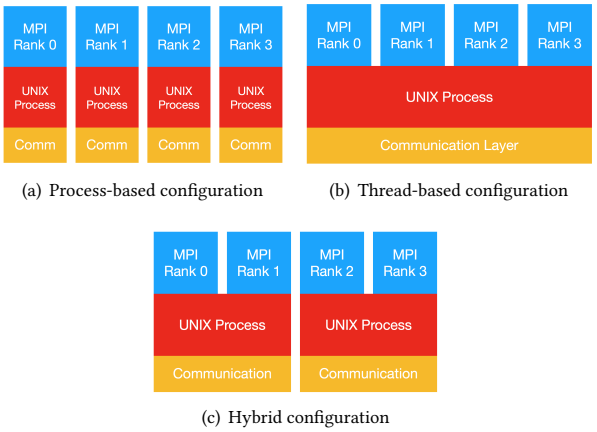
```
int MPIX_Checkpoint(MPIX_CR_state_t* state);                    1
```

Figure 1: Proposed transparent checkpoint interface.

CR Constant	Definition
MPIX_CR_STATE_ERROR	An error has occurred
MPIX_CR_STATE_CHECKPOINT	The program has checkpointed
MPIX_CR_STATE_RESTART	The program has restarted
MPIX_CR_STATE_IGNORE	Command ignored (not supported)

Table 1: `MPIX_Checkpoint` constants definitions.

As presented in Figure 1, the `MPIX_Checkpoint` call is a collective with respect to `MPI_COMM_WORLD`. It will return a state defined in Table 1, depending on its effect. When calling this function, there should be no unmatched MPI messages. One point to note is that this call can be invoked in different scenarios. First, a program may checkpoint on a regular basis and therefore proceed to call `MPIX_Checkpoint`, each time returning `STATE_CHECKPOINT`. In the second case, a given program can be restarted from a previous checkpoint. Here its flow-control will immediately come from `MPIX_Checkpoint`, except that this time the return value will be `STATE_RESTART`, allowing the application to account for such case. If it is not possible to checkpoint (e.g., due to lack of support), a runtime can return `STATE_IGNORE` to inform the application that nothing was saved.



(a) Process-based configuration (b) Thread-based configuration

(c) Hybrid configuration

Figure 2: Outline of the various execution configurations supported by the MPC thread-based MPI runtime.

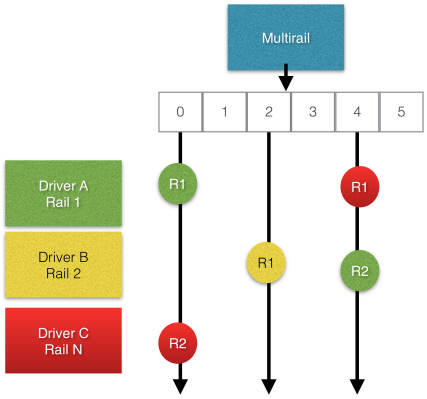


Figure 3: Overview of the multi-rail infrastructure in MPC.

The collective nature of the call also ensures that it is correctly invoked in the case of a hybrid program. For instance, if this function is called in an OpenMP parallel region, it will require the application to implement a critical region so as not to violate the collective nature of the call. By clearly stating how the checkpoint function is to be called globally, it abstracts the integration of such a call, while simplifying the implementation requirements.

With the checkpoint interface definition, we turn our attention to how an implemenation can leverage high-performance networking infrastructure. We start by introducing the networking support in the MPC[26] runtime. Then we discuss the underlying mechanisms behind the `MPIX_Checkpoint` function, particularly in the context of a thread-based MPI.

5 NETWORK MODULARITY IN MPC

MPC's network architecture is based on *communication rails* which are associated with a given network driver. MPC can combine at runtime multiple communication drivers, which are used together to provide communication capabilities at the MPI interface level. In this section, we present an overview of "multi-rail" support in MPC. In particular, we discuss how MPC is able to bootstrap its networking support using network *control messages* ("signalling" messages) routed on a base topology. With this mechanism in place, we show how it enables modular, transparent checkpointing while preserving high-speed network capabilities.

5.1 Multi-Rail in MPC

MPI is dedicated to enabling high-performance messaging between distributed processes. To do so, it can rely on multiple network technologies. For example, one system could use Infiniband EDR between nodes and a shared-memory segment (SHM) inside a given node at the same time. More generally, an MPI runtime usually supports at least two network types: 1) for optimized intra-node communications (latencies lower than the μsec), and 2) for inter-node communications, where remote-direct memory access (RDMA) support could be used to optimize MPI's performance (in the μsec range). The switch between intra- and inter-node policies is then defined as the position of the target MPI process relative to the source, that is, whether they are located on the same node. Multi-rail is naturally present in any state-of-the-art MPI runtime. The following describes how MPC handles multi-rail, but the overall principle is applicable to any MPI runtime, thread-based or not.

As shown in Figure 2, MPC is a thread-based MPI implementation which makes it possible to have multiple MPI "tasks" within a MPI "process" that is bound and running in a UNIX process. MPI tasks are equivalent to traditional MPI processes in that they can communicate via MPI with each other. For instance, messages could be exchanged inside a given MPI "process" if both tasks are running in shared memory as in Figure 2(b). Or, the messages could be routed to the multi-rail network layer, if the tasks are remote from each other. In this case, the multi-rail support must identify the most efficient rail to reach a given remote UNIX process (Figure 2(a)). Moreover, these means of exchange are not mutually exclusive and hybrid configurations such as the one of Figure 2(c) involve both messaging layers depending on peers.

In the rest of this paper, we will consider solely the communication between UNIX processes, as it is the only part of MPC involving inter-node communications and requiring interaction with network cards. This put us in the general case of a process-based MPI where MPI processes are UNIX processes and allow us to reason in a more general context applicable to any MPI implementation. However, it should be noted that the methodology we develop in this paper has been validated in all configurations of Figure 2.

Communications in MPC are based on endpoints belonging to a communication rail (see Figure 3). Endpoints are sorted by priority inside ordered lists corresponding to a given remote process. When MPC tries to communicate with a remote endpoint, it walks on the list for a given endpoint and tries to *elect* a candidate. Election includes the concept of *gate*, setting conditions (in terms of message type or size) related to the use of a given rail. If no endpoint is found in the list, a second election process is done walking rails in order of priority to create a new endpoint. If one rail supports this *on-demand* feature, the connection handler is called in order to create the low-level route. Section 5.3 will detail how this *on-demand* connection process is implemented in MPC. If this succeeds, MPC

```
<config>                                                    1
  <name>tcp_config_mpi</name>                               2
  <driver><tcp/></driver>                                   3
</config>                                                   4
                                                            5
<rail>                                                      6
  <name>tcp_mpi</name>                                      7
  <priority>1</priority>                                    8
  <topology>ring</topology>                                 9
  <config>tcp_config_mpi</config>                           10
</rail>                                                     11
                                                            12
<rail>                                                      13
  <name>tcp_large</name>                                    14
  <priority>10</priority>                                   15
  <topology>none</topology>                                 16
  <config>tcp_config_mpi</config>                           17
  <gates>                                                   18
    <gate>                                                  19
      <minsize>                                             20
        <value>32KB</value>                                 21
      </minsize>                                            22
    </gate>                                                 23
  </gates>                                                  24
</rail>                                                     25
                                                            26
<cli_option>                                                27
  <name>multirail_tcp</name>                                28
  <rails>                                                   29
    <rail>tcp_large</rail>                                  30
    <rail>tcp_mpi</rail>                                    31
  </rails>                                                  32
</cli_option>                                               33
```

Figure 4: Example of XML configuration file for MPC's multi-rail engine.

proceeds to use the new endpoint. Otherwise, it crashes with a *no route to process* error, meaning that no valid network path exists or could be created to reach the targeted process.

As presented in Figure 4, MPC's multi-rail support relies on an XML configuration file that we now describe bottom to top. First, we define a Command Line option (CLI) named *multirail_tcp* and attach two rail definitions to it: *tcp_large* and *tcp_mpi*. As a consequence, when launching the parallel execution with *mpirun*, the `-net=multirail_tcp` option will create the two aforementioned rails. If we now look closer at the rail definitions, each of them is named and is attached to a priority. Observe how the *tcp_large* has a higher priority than the *tcp_mpi* one, it is because we want each message to first try it. Indeed, the "large" rail has a gate function defined and requires a message to be larger than 32Kb to be able to transit through it. If this test fails, the message then checks the *tcp_mpi* rail which matches any message (as it has no gate function). One last part involved at the beginning of the configuration file is the network-level parameters in the *config* markup. In this case, they are shared between the two rails and we simply use the default TCP configuration – the end user is free to create configurations for his own rails.

One point that we overlooked in the previous configuration is rail *topology*. It plays an important role in the checkpoint-restart mechanism because it defines the initial connection state of rails (defined as *static* routes). Such initial routes are used to convey *control-messages*, allowing on-demand connection mechanisms to

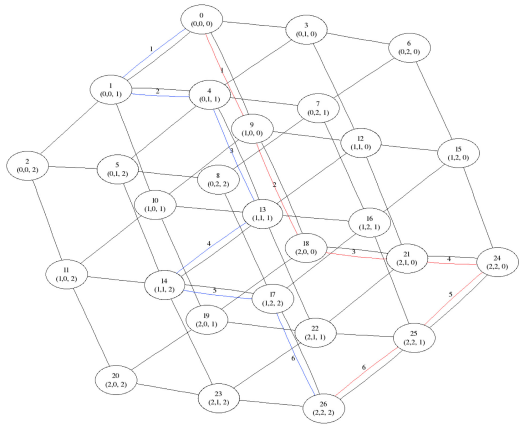


Figure 5: Routing comparisons on a 3D topology with 1D in red, and 3D distances in blue.

establish additional networking configuration. We refer to this as the *signalling network* for MPC.

5.2 Signalization Network

MPC's network layer can also be used to provide a signalling network whose role it is to allow remote processes to be reachable in a one-sided fashion. (This could also be described as remote procedure calls (RPCs) or as active messages (AM) with MPI semantics). Indeed, some MPI functionalities already depend on this being possible, for example, when establishing *on-demand* connection, emulating one-sided when no RDMA capable network is available, and even within the rendezvous protocol, where target notification is required.

One of the main proprieties of the signalling network is its capability to route messages according to a simple 1D distance metric, defined as the absolute value between the source and target ranks. The reason for retaining such a simple metric is because we wanted it to be portable on any topology, indifferently from its complexity. To do so, we imposed the simple constraint of embedding at minimum a ring in the topology. This ring is what we call "static routes" in MPC's bootstrap and its main role is to ensure that, for given a source process, there will always be a path to minimize the 1D distance to its destination. It is this property that incited us to rely on a minimal ring, since dealing otherwise with sparse and/or arbitrary topologies, there may be cases where a distance metric is not sufficient to escape from a local minimum.

Despite managing to limit the theoretical network diameter, the 1D distance metric we retained to ensure routing robustness for arbitrary topologies, nevertheless fails in identifying the shortest path in higher dimensions. Nonetheless, the availability of *shortcuts* still allows 1D routing to take advantage of the higher dimensions – see for example the difference of behavior in Figure 5.

5.3 Network Bootstrap in MPC

MPC implements a multi-rail engine and a signalling network. However, in order to be able to create endpoints, there are some

cases when a process would like to query information from another one without knowing it explicitly. The case which is of particular interest in this paper is the on-demand connection when, for example, two processes are exchanging and building their Queue-Pair information. This can be illustrated with a TCP analogy, where the IP address and remote port has to be exchanged prior to establishing a connection. When MPI starts, processes are usually disconnected and connected on demand. To do so, MPI runtimes rely on the *process management interface* (PMI) provided by the launcher. PMI provides a Key-Value Storage (KVS) which is relied upon to bootstrap connections, prior to MPI processes creation. MPC naturally relies on the PMI, but it also implements its own bootstrap system in order to limit the amount of information to be exchanged with the PMI. Indeed, if there are 10^6 processes it can be costly to exchange the whole information relative to all the ranks in an all-to-all manner, particularly prior to having any high-performance communication substrate. To circumvent this, MPC defines the notion of rail topology. In all cases, there must be a rail accepting all messages with a ring topology (see *tcp_mpi* in Figure 4). This rail is initialized using solely the PMI KVS, exchanging, in this case, *rank:host:port* tuples.

Later *on-demand* connections, however, will not rely on the PMI, but on control messages which can be routed through the network to the destination. Such messages use a distance metric and take advantage of any route and any rail. Consequently, even if only a TCP ring is present during startup, it is highly probable that "shortcuts" will appear as MPI processes start communicating. This property is at the core of MPC's ability to checkpoint-restart. Indeed, existing checkpointing tools are not able to save the network state for high-speed networks, unless by wrapping all existing API calls. This is a high overhead, for example, in the case of Infiniband. Instead, such tools are limited to solely restoring TCP sockets between processes. This capability in MPC allows restored MPI programs to operate immediately after the restart, instead of relying on a complete network re-initialization through a PMI key exchange.

The main points to remember are MPC's multi-rail engine and its ability to manage endpoints of multiple types to enable communication. These endpoints are stored in an ordered list and go through an election mechanism. MPC relies on the PMI only to bootstrap an initial ring which is relied upon to convey later on-demand connection requests. Thanks to its modular definition, MPC is capable of closing a given rail removing all references to the associated network. It is this mechanism, combined with signalization, that enables MPC's checkpoint functionalities.

6 CHECKPOINTING MPI

Given the description of MPC's multi-rail and signalling capabilities, we are ready to describe how to transparently checkpoint a thread-based MPI implementation using DMTCP. Our strategy is to adapt DMTCP's approach to the specific problem related to hosting multiple MPI Processes in a given UNIX process. By leveraging how MPC takes advantage of its network infrastructure to enable high-speed network support in its checkpoints, it is possible to overcome some of DMTCP's limitations. In particular, we will contrast our model with the Infiniband wrapping approach.

6.1 DMTCP Overview

DMTCP [1] is the distributed implementation of MTCP [27], a user-level checkpoint implementation compatible with POSIX threads. Its goal is to transparently save and restore distributed applications. To do so, it relies on a coordinator process (`dmtcp_coordinator`), steering applications under C/R for the current user. It can be reached through an IP address/port tuple. Users can then interact with the coordinator through running applications or the CLI. Each application to track is wrapped with the `dmtcp_launch` command, preloading the MTCP wrapping library, on each process to start. By wrapping most of the libc, DMTCP is able to closely track the relationship between execution streams. Moreover, a signal handler is defined in each thread (by default `SIGUSR2`), to trigger a checkpoint, stopping each thread (using `tkill`), saving its own data, including local context (register) and stack.

At the network level, DMTCP is able to save live sockets and pipes (after converting them to socket pairs). For this purpose, it goes through a comprehensive process including the election of an owner of the respective file descriptors (when shared between forks) and accounting for "on wire" data inside the socket in order to restore them in case of a restart. As a consequence, DMTCP can reliably save TCP connection between distributed processes in a transparent manner. It is this aspect we rely upon for MPC. Also, DMTCP is able to save shared-memory segments, making it compatible with processes running on the same node with SHM.

As far as the restart model is concerned, the first step is to recreate the same topology, relaunching each checkpointed process. DMTCP proposes a dedicated script `dmtcp_restart_script.sh`, only compatible with Hydra and Slurm, ensuring the new configuration (from the restarting environment) is compliant with the initial one, before restarting the processes. The first step deals with restoring network connections (and pipes) as they might be shared between processes. Then, execution streams are restored and eventually the program image is reinjected from the checkpoint data and file descriptors are reopened and offsets restored. At this point, execution streams wait in a semaphore and are able to restart once all threads are ready. DMTCP reproduces the same process and thread hierarchy (by tracking fork/clone) to make the system topology consistent (parent/child relation).

It is this process, fully accounted for by DMTCP, that we leverage in MPC to provide the checkpoint-restart feature with the subtlety of hosting several MPI processes in a single UNIX process. In this case, a dedicated synchronization mechanism is required.

6.2 Thread-Based MPI Checkpoint

DMTCP and its coordinator are designed so that a single request for the checkpoint is automatically broadcasted to all the processes. However, in MPC we have to handle the fact that there are multiple MPI processes in a given UNIX process – checkpointing taking place at this latter level.

As depicted in Figure 6, MPC solves this by proceeding to a first intra-node barrier between MPI tasks located in the same process. Once a master task has been elected, a second barrier occurs between processes such as only a single rank invokes the internal checkpointing routines of DMTCP.

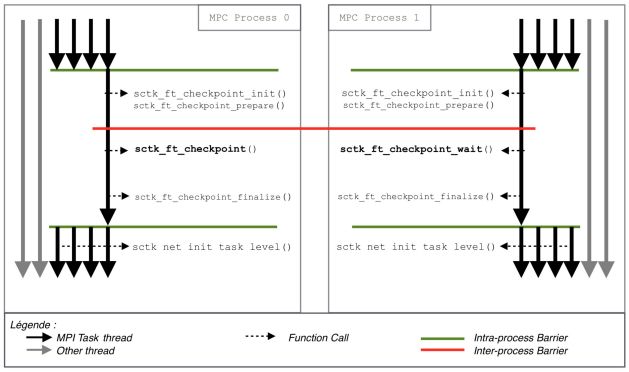


Figure 6: Two-level synchronization scheme enabling checkpointing in MPC.

6.3 Limitations in DMTCP

During our developments around this integration of DMTCP in MPC, we discovered limitations in the tool. The developers have been very active to address some of them and some others are still pending. We will now provide a quick outline for each of them.

Pinning Preservation. When we began our developments, pinning was preserved at the checkpoint, but not at the restart. The consequence was that threads were not bound to a particular core. This may remain unnoticed in the case of a process-based MPI. However, as MPC launches a single process per node, this led to performance loss. This has been reported and fixed in the Git repository, in the branch tracking the version 2.5.

Memory Locality. When a process is restarted with DMTCP, pages are generally not located on the correct numa-node. This leads to a loss of locality for the restarted process. To date, this has not been addressed in DMTCP. A possible workaround to this would be to rely on external tools such as *autonuma*.

GS Register Handling. In its current version, DMTCP does not save the GS register. This is generally harmless as this register is mostly unused on x86_64. However, MPC uses this register to infer its own level of TLS indirection[5], similar to how FS register is used the NPTL implementation for the same reason. As a consequence, an unmodified version of DMTCP is not able to correctly checkpoint a privatized program (i.e. multiple MPI tasks inside a single UNIX process). This has been discussed with the developers[1] and we proposed a fix.

Runtime defining pthread_create. As MPC provides its own user-level thread scheduler, it provides its own pthread implementation. When being wrapped with DMTCP we encounter an issue as it is preloaded and implements `dlsym`, yielding the following call stack:

```
#0 pthread_create (from libdmtcp.so) //<-------          1
#1 dlsym() (from batch_queue.so)                         2
#2 dlsym() (from your mpc_framework.so)                  3
#3 pthread_create() (from mpc_framework.so)              4
#4 pthread_create() (from dmtcp.so)  //<-------          5
#5 pthread_create() (from a.out)                         6
```

[1]https://github.com/dmtcp/dmtcp/issues/607

This leads to a stack overflow by creating a loop. This code seems to be present solely for IntelMPI which resolves `PMI_Init` with `dlsym(RTLD_NEXT, "PMI_Init")`. The call is currently not compiled conditionally. We reported it to the developers[2], but our current workaround is simply to comment it out. This is done in the version of DMTCP which is bundled in MPC.

6.4 High-Speed Network Support

One of the most difficult parts of the checkpoint is the high-speed network. Indeed, as it relies on dedicated hardware, it represents the possibility of shared state located outside processes' memory. As a consequence, saving process state is not sufficient to restore connections over HPC networks such as Infiniband or Portals. For example, memory pinning register segments in the device (to allow address translation and to retrieve authentication tokens) is not checkpointable. In order to circumvent this issue, DMTCP provided a plugin completely wrapping the libverbs (low-level Infiniband programming interface) in order to track and preserve a shadow state of all the operations taking place on the card[10]. This approach enabled transparent checkpointing of Infiniband networks, but not without some drawbacks.

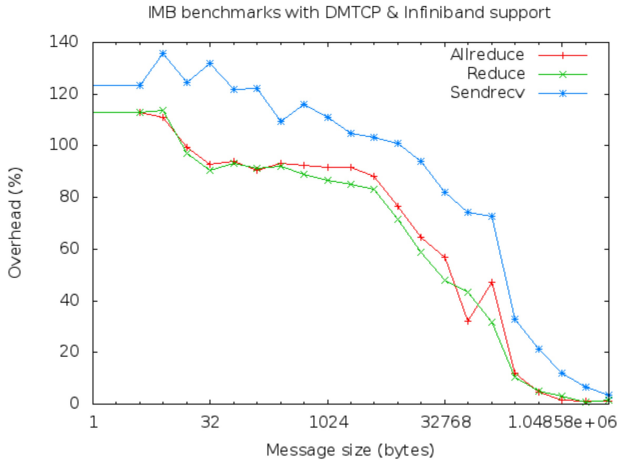


Figure 7: Overhead for Infiniband wrapping in DMTCP.

As presented in Figure 7, wrapping Infiniband has a direct impact on common MPI implementations. Indeed, as libverbs calls are by definition on the critical path of any IB communication, this extra wrapping leads to a performance overhead. We observed up to 140% overhead for small messages, where the extra latency is most visible. The main drawback of the approach is that it imposes this performance loss outside of checkpointing sections, leading to a permanent slowdown. It is this problem that encouraged us to look for other alternatives mitigating the cost.

MPC's network has been built as a modular set of driver instances (called rails) stacked on top of each other (Section 5). Moreover, on-demand connections are managed through in-band messages which can be routed through a dedicated signalization network (Section 5.3). Then, without any action, routes existing prior to

[2]https://github.com/dmtcp/dmtcp/issues/604

```
  (...)
<rail>
  <name>tcp</name>
  <priority>1</priority>
  <topology>ring</topology>
  <config>tcp_config_mpi</config>
</rail>

<rail>
  <name>ib_mpi</name>
  <priority>2</priority>
  <topology>none</topology>
  <config>ib_config_mpi</config>
</rail>

<cli_option>
  <name>ckpt_net_no_pmi</name>
  <rails>
    <rail>tcp</rail>
    <rail>ib_mpi</rail>
  </rails>
</cli_option>
```

Figure 8: Sample configuration file for PMI-less restart.

the checkpoint will be included in the checkpoint, as present in internal data-structures. However, some of these routes will be invalid at restart, because part of information they relied on, are now undefined. While TCP network is fully handled by DMTCP with a minimum cost, this is not the case for Infiniband. It is not possible to purge the multi-rail undefined endpoints efficiently after each restart because we cannot ensure in which state the network layer has been stopped at checkpoint time, potentially leading us to deadlocks.

Thus, we consider removing these routes before checkpointing the application. Rails which are not checkpointable had to be fully closed each time a checkpoint is performed. This means that MPC frees all the resources linked to a given driver and proceeds to remove the routes from the multi-rail lists (see Figure 3). Some drivers are exempted from this closing as they are compatible with DMTCP (e.g., TCP and SHM). In this case, static routes from the original topology are preserved. For these last two drivers, DMTCP will be able to restore a state matching one of existing routes known to the process. Dealing with drivers which had to be closed, there will be no route associated to these rails in the restarted process image, a new rail will be allocated from scratch.

In order to illustrate more practically how this is achieved, Figure 8 presents a network configuration capable of restarting without PMI support. It consists of an Infiniband rail with a `None` topology meaning that it expects no route to be created at initialization time. The second rail is a TCP one with an initial ring built over PMI, as described in Section 5.3. When restarting the application from a checkpoint, the network is reinitialized and therefore, is each rail. However, there will be routes present in the TCP rail – as we did not remove them – and they will not be recreated from the PMI. As a consequence, the signalization network will be immediately available upon restart and so will be the on-demand feature.

If we consider the excerpt of configuration presented in Figure 9, there is now a single rail having a *torus* topology. As an Infiniband rail, it will be closed when checkpointing. However, unlike in the case where there was a signalling network, routes will have to be

```
<rail>
  <name>ib_mpi</name>
  <priority>2</priority>
  <topology>torus</topology>
  <config>ib_config_mpi</config>
</rail>

<cli_option>
  <name>ckpt_net_pmi</name>
  <rails>
    <rail>ib_mpi</rail>
  </rails>
</cli_option>
```

Figure 9: Sample configuration file for PMI restart.

recreated using the PMI in MPC. This process is always limited to a ring before switching to control messages through this bootstrap topology in order to promote the network to a torus. The PMI is not checkpointed by DMTCP. It means that a process, if restarted in a new allocation, will find no keys inside the key-value data-store. For this reason, any runtime willing to implement checkpoint-restart has to make sure that it does not make expectation on PMI values as they may have disappeared. Moreover, saving the PMI values is totally practicable, but would be of little use as such values would be bound to a previous allocation.

6.5 Current Issues

It is important to observe that a consequence of this approach is that it closes dynamic routes at each checkpoint. Indeed, in order to create a valid process image, we alter the state of the application even if it does not goes through a restart. This is currently a limitation of our model as later communications will immediately recreate routes previously closed for the sole purpose of a checkpoint. Initially, we envisioned to simply remove uncheckpointable endpoints from the multi-rail list (see Figure 3) without freeing any memory. This, however, led to various issues, first obviously a memory leak with the added complexity that it was not possible to free this dangling memory at the restart. Second, leaving an open device, for example, the IB HCA, means that there is an open file descriptor upon checkpoint and that DMTCP will try to drain it, eventually leading to a deadlock. Consequently, dynamic route closing and its associated performance impact appeared to be the least worst method in the case of Infiniband networks. Other network types, in particular, connection-less networks such as Portals 4 or Omnipath, may circumvent this limitation. We are currently studying this possibility.

Figure 7 compares the performance of our high-speed network checkpointing methodology to the Infiniband wrapping one. A direct execution shows no measurable overhead on communications when starting from a checkpoint on the IMB benchmarks. Comparing the relative overhead for the IMB with the `MPI_Allreduce` collective (see Figure 10), it can be seen that the restarted program has similar performance to the initial process image. The reason is because the only penalty taken by the restarted program is route creation which is a punctual process mitigated by the repetitive communication pattern in communications. The checkpoint by itself, however, has a performance cost as it closes connections, nonetheless, we believe that this is an acceptable point as the user

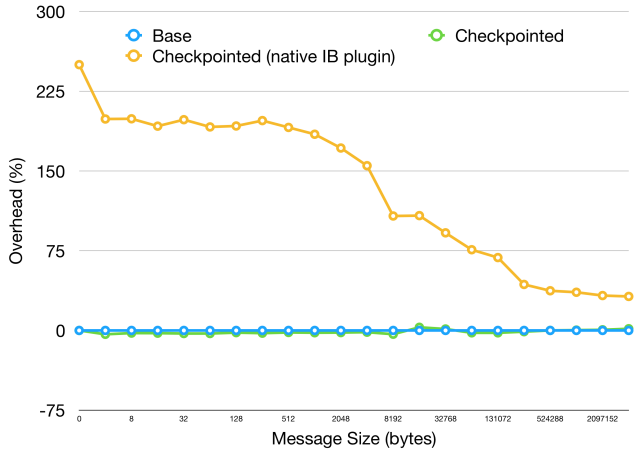


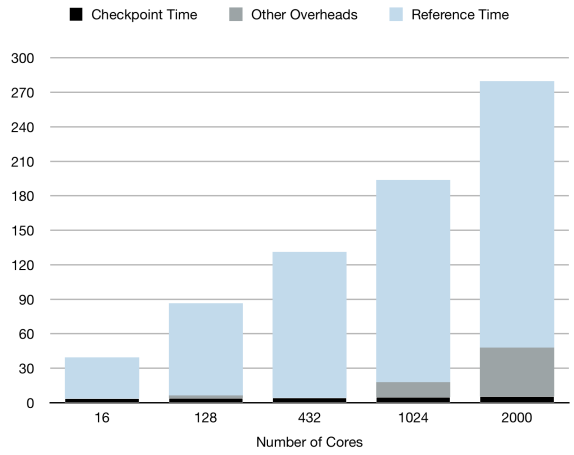
Figure 10: IMB Allreduce performance overhead between DMTCP Infiniband support and MPC's support.

is free to set its frequency. In summary, our method creates a transient overhead to prevent a permanent one.

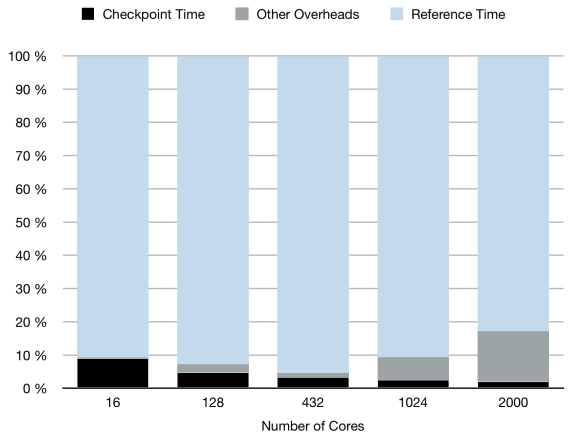
The duration of a given checkpoint is highly dependent on both the scale of the MPI job and the amount of memory it uses. Moreover, the wall-time overhead it incurs is correlated with the number of checkpoints performed during a given execution. Consequently, as for transparent checkpointing, we are willing to leave the application untouched, the only parameter available to limit the overhead is checkpoint frequency. If we consider a computation lasting T_s seconds and a checkpointing time of T_c every τ seconds, we have the following total duration D for the checkpointed program: $D = T_s + \frac{T_s}{\tau} T_c$. Denoting $f = \frac{1}{\tau}$ as the checkpointing frequency, we immediately have $D = T_s(1 + fT_c)$. Now reasoning in terms of overhead, we have $O_{vh} = \frac{D}{T_s} = 1 + fT_c$, this shows that the overhead is necessarily positive and easily computable from both checkpoint frequencies and duration. More importantly, it can easily be budgeted. For example, considering a one-minute checkpoint time and a maximum overhead of 1% we have $f = \frac{1\%}{T_c}$ and therefore a checkpointing period $\tau = 6000$ seconds or 1 hour and 40 minutes. This small formula shows that it is relatively easy to amortize the checkpointing time through the frequency parameter in a reasonable time. When measuring the Intel Messaging Benchmark (IMB), we encountered checkpoints around three seconds for 32 MPI processes – $T_c = 60$ is then already a pessimistic value.

6.6 Application to a Representative Benchmark

In order to assess the performance or our checkpointing methodology we chose to run our checkpointing methodology at scale on the Lulesh[23] benchmark. In particular, we focused ourselves on two aspects. First, the checkpoint time that can be directly connected to a global overhead given a checkpointing frequency – as outlined earlier. Second, we want to measure the cost associated with closing connections in terms of execution time outside of checkpoints. Measurements were carried over on a test systems with Sandy-Bridge processors and 16 cores per node using a problem size of 30. Interconnect consists in mlx4 Infiniband Host-Channel Adapters. In



(a) Walltime breakdown in seconds



(b) Walltime breakdown in percentage

Figure 11: Checkpoint and reference times for Lulesh (size 30) in function of the number of cores.

order to characterize the cost of our methodology we proceeded to measure a single checkpoint in the middle of the parallel execution at various scales. Lulesh was launched with a single process per node in MPI plus OpenMP configuration.

In Figure 11(a), we see the breakdown of the walltimes in terms of reference time, checkpoint and other overheads. The checkpoint is the time spent generating the data in the collective call, other overheads accounts for other difference with respect to reference time, including on-demand connections. In Figure 11(b), we present the same results in percentage to highlight the relative cost of each times. What can be seen that that the checkpointing time by itself remains relatively steady as the number of cores increases. However, we observe an increase in terms of indirect costs from 0.3 % up to 18 % at larger scale. This can be explained by the increasing number of on-demand connections as the number of nodes increases.

In order to expose these results in a more practical manner, we used the formula presented in previous section to compute

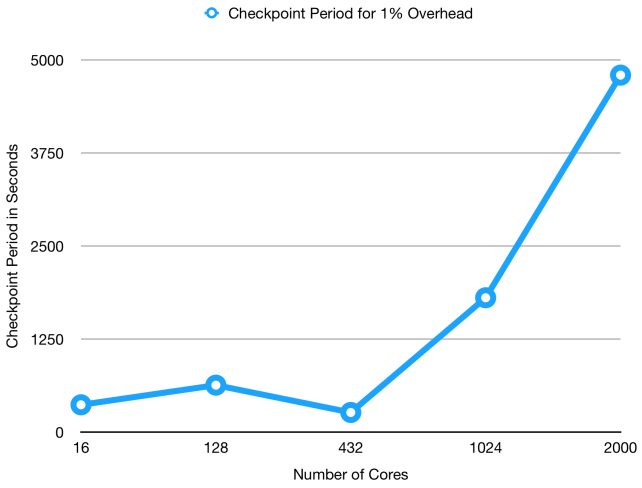


Figure 12: Checkpoint period for 1% overhead computed from results presented in Figure 11.

the checkpointing period such as the overhead is 1% in the light of previous measurement. To do so, we added overheads direct and indirect and considered them as the checkpoint cost. This yielded the values presented in Figure 12. One can see that despite potentially expensive, the checkpoint cost on the walltime can be mitigated for long-running programs. In our case, for Lulesh, we see checkpointing periods ranging from five minutes at one process to one hour and twenty minutes at larger sizes. The overall checkpoint cost therefore increases requiring checkpoints to be further spaced to mitigate their apparent cost. However, we think that our methodology still yields checkpoint periods compatible with the use at scale as checkpointing periods of a few hours are not unrealistic. Eventually it is important to note that the cost is directly linked not only to the scale but also to the size of the dataset manipulated by the application – the transparent approach dumping full process images.

6.7 Generalizing to other MPI Implementations

Results presented in this paper were obtained with the MPC runtime which provides support for transparent checkpointing. In particular, we presented a dedicated collective call `MPIX_Checkpoint` and relied on high-speed network disconnection prior to checkpointing. In addition, MPC has a signalization network which can be restored by DMTCP (being in TCP) and latter used to relay on demand connection demands to reconnect routes. This second aspect is not compulsory to enable transparent checkpointing and therefore allows our methodology to be adaptable to other runtimes. Compulsory requirements are (1) the ability to close high-speed connections and to restore them later on and the (2) capacity of restarting either from the PMI or using a support network (one may consider launcher processes). This methodology can then be adapted to other runtimes and is not dependent from MPC, it simply requires state management capacities in MPI for connections and startup – DMTCP handling most of the checkpoint transparently.

7 CONCLUSION

In this paper, we have presented our implementation transparent checkpointing in the MPC MPI runtime. This is the first illustration of transparent checkpoint restart – agnostic from the application – with network support in a thread-based MPI. Checkpointing has already been illustrated in runtimes involving user-level threads in the past including Charm++[21] and its combination with AMPI[30]. Our approach is more general as it does not rely on serialization assumptions in terms of application's programming model, aspect directly inherited from DMTCP's versatility. However, as we put no constraints on the application, some scenarios possible with Charm++ are out of reach, they include in-memory checkpointing[31] and restarting the program on a different number of processes[22]. In our case, we solely presented a synchronous checkpointing interface which is only a subset of what is possible in terms of fault tolerance. Indeed, new interfaces such as ULFM in MPI should allow applications to react to failures at runtime – limiting the need for post-mortem restart as provided by DMTCP. Moreover, our approach does not support partial checkpoint restart, it is nonetheless a point that we would like to explore in the future.

After presenting related work, we presented the synchronous checkpointing interface that we implemented in MPC. MPC's network modularity is a key component in high-speed network checkpointing capabilities. Indeed, it allows the closing of a given driver thanks to the unified routing tables and later on at restart the restoration of these routes through an in-band signalization network which is used to implement on-demand connections. To implement checkpointing, we adapted DMTCP for MPC. We provide support for the high-speed network through a disconnect-reconnect scheme preventing constant overhead, otherwise present with methods involving Infiniband wrapping. Our model has the advantage of avoiding any overhead on MPI calls but requires to re-establish connections after checkpoints, a cost which can be mitigated by limiting their frequency. This method was designed for applications performing punctual checkpoints (at an hour scale for example) and was intended to be fully integrated into MPC's workflow thanks to dedicated options, allowing C/R through a few simple command-line flags as presented in Appendix **??**.

8 FUTURE WORK

Current implementation in MPC has been designed to provide an initial support for transparent checkpoint/restart, saving our users from the development of their own solution. Thanks to DMTCP, we were able to provide such feature through the addition of a single collective call in the code. However, checkpointing and more generally fault tolerance, for example through ULFM, allows a much wider range of scenarios. Indeed, currently, our runtime has to fully restart in order to recover from a single node failure. We would like to explore partial checkpointing with spare nodes leveraging our signalization network. Another aspect that seems promising is the exploration of connection-less networks and how they might be checkpointed more efficiently than by disconnecting-reconnecting peers. In particular, we are working on the Bull Exascale Interconnect (BXI)[11] Portals 4 network[2] to develop such support.

REFERENCES

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

[2] Brian W Barrett, Ronald Brightwell, Scott Hemmert, Kevin Pedretti, Kyle Wheeler, Keith Underwood, Rolf Riesen, Arthur B Maccabe, and Trammell Hudson. 2012. The Portals 4.0 network programming interface. *Sandia National Laboratories, November 2012, Technical Report SAND2012-10087* (2012).

[3] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka. 2011. FTI: High performance Fault Tolerance Interface for hybrid systems. In *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 1–12. DOI: https://doi.org/10.1145/2063384.2063427

[4] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator.. In *USENIX Annual Technical Conference, FREENIX Track*, Vol. 41. 46.

[5] Jean-Baptiste Besnard, Julien Adam, Sameer Shende, Marc Pérache, Patrick Carribault, Julien Jaeger, and Allen D Maloney. 2016. Introducing Task-Containers as an Alternative to Runtime-Stacking. In *Proceedings of the 23rd European MPI Users' Group Meeting*. ACM, 51–63.

[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. DOI: https://doi.org/10.1177/1094342013488238 arXiv:https://doi.org/10.1177/1094342013488238

[7] Wesley Bland, Aurelien Bouteiller, Thomas Herault, Joshua Hursey, George Bosilca, and Jack J Dongarra. 2012. An evaluation of user-level failure mitigation support in MPI. In *European MPI Users' Group Meeting*. Springer, 193–203.

[8] Aurelien Bouteiller, George Bosilca, and Jack J. Dongarra. 2015. Plan B: Interruption of Ongoing MPI Operations to Support Failure Recovery. In *Proceedings of the 22Nd European MPI Users' Group Meeting (EuroMPI '15)*. ACM, New York, NY, USA, Article 11, 9 pages. DOI: https://doi.org/10.1145/2802658.2802668

[9] Darius Buntinas, Camille Coti, Thomas Herault, Pierre Lemarinier, Laurence Pilard, Ala Rezmerita, Eric Rodriguez, and Franck Cappello. 2008. Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI Protocols. *Future Generation Computer Systems* 24, 1 (2008), 73 – 84. DOI: https://doi.org/10.1016/j.future.2007.02.002

[10] Jiajun Cao, Gregory Kerr, Kapil Arya, and Gene Cooperman. 2014. Transparent Checkpoint-restart over Infiniband. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing (HPDC '14)*. ACM, New York, NY, USA, 13–24. DOI: https://doi.org/10.1145/2600212.2600219

[11] S. Derradji, T. Palfer-Sollier, J. P. Panziera, A. Poudes, and F. W. Atos. 2015. The BXI Interconnect Architecture. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. 18–25. DOI: https://doi.org/10.1109/HOTI.2015.15

[12] James Dinan, Ryan E Grant, Pavan Balaji, David Goodell, Douglas Miller, Marc Snir, and Rajeev Thakur. 2014. Enabling communication concurrency through flexible MPI endpoints. *The International Journal of High Performance Computing Applications* 28, 4 (2014), 390–405.

[13] P EMELYANOV. 2011. CRIU: Checkpoint/Restore In Userspace, July 2011. (2011). https://criu.org/

[14] Graham E. Fagg and Jack J. Dongarra. 2000. FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Jack Dongarra, Peter Kacsuk, and Norbert Podhorszki (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 346–353.

[15] Marc Gamell, Daniel S. Katz, Hemanth Kolla, Jacqueline Chen, Scott Klasky, and Manish Parashar. 2014. Exploring Automatic, Online Failure Recovery for Scientific Applications at Extreme Scales. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*. IEEE Press, Piscataway, NJ, USA, 895–906. DOI: https://doi.org/10.1109/SC.2014.78

[16] R. Garg, K. Sodha, Z. Jin, and G. Cooperman. 2013. Checkpoint-restart for a network of virtual machines. In *2013 IEEE International Conference on Cluster Computing (CLUSTER)*. 1–8. DOI: https://doi.org/10.1109/CLUSTER.2013.6702626

[17] William Gropp, Rajeev Thakur, and Ewing Lusk. 1999. *Using MPI-2: Advanced Features of the Message Passing Interface* (2nd ed.). MIT Press, Cambridge, MA, USA.

[18] 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. http://stacks.iop.org/1742-6596/46/i=1/a=067

[19] Daniel Holmes, Kathryn Mohror, Ryan E Grant, Anthony Skjellum, Martin Schulz, Wesley Bland, and Jeffrey M Squyres. 2016. MPI Sessions: Leveraging Runtime Infrastructure to Increase Scalability of Applications at Exascale. In *Proceedings of the 23rd European MPI Users' Group Meeting*. ACM, 121–129.

[20] Joshua Hursey, Richard L. Graham, Greg Bronevetsky, Darius Buntinas, Howard Pritchard, and David G. Solt. 2011. Run-Through Stabilization: An MPI Proposal for Process Fault Tolerance. In *Recent Advances in the Message Passing Interface*, Yiannis Cotronis, Anthony Danalis, Dimitrios S. Nikolopoulos, and Jack Dongarra (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 329–332.

[21] Laxmikant V. Kale and Sanjeev Krishnan. 1993. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '93)*. ACM, New York, NY, USA, 91–108. DOI: https://doi.org/10.1145/165854.165874

[22] Laxmikant V Kale and Gengbin Zheng. 2009. Charm++ and AMPI: Adaptive runtime strategies via migratable objects. *Advanced Computational Infrastructures for Parallel and Distributed Applications* (2009), 265–282.

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

[24] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R De Supinski. 2010. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*. IEEE, 1–11.

[25] Xiang Ni, Esteban Meneses, Nikhil Jain, and Laxmikant V Kalé. 2013. ACR: Automatic checkpoint/restart for soft and hard error protection. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 7.

[26] Marc Pérache, Patrick Carribault, and Hervé Jourdren. 2009. MPC-MPI: An MPI implementation reducing the overall memory consumption. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 94–103.

[27] Michael Rieker, Jason Ansel, and Gene Cooperman. 2006. Transparent User-Level Checkpointing for the Native Posix Thread Library for Linux.. In *PDPTA*, Vol. 6. 492–498.

[28] Keita Teranishi and Michael A. Heroux. 2014. Toward Local Failure Local Recovery Resilience Model Using MPI-ULFM. In *Proceedings of the 21st European MPI Users' Group Meeting (EuroMPI/ASIA '14)*. ACM, New York, NY, USA, Article 51, 6 pages. DOI: https://doi.org/10.1145/2642769.2642774

[29] Rajeev Thakur, William Gropp, and Ewing Lusk. 1999. On Implementing MPI-IO Portably and with High Performance. In *Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems (IOPADS '99)*. ACM, New York, NY, USA, 23–32. DOI: https://doi.org/10.1145/301816.301826

[30] Gengbin Zheng, Chao Huang, and Laxmikant V. Kalé. 2006. Performance Evaluation of Automatic Checkpoint-based Fault Tolerance for AMPI and Charm++. *SIGOPS Oper. Syst. Rev.* 40, 2 (April 2006), 90–99. DOI: https://doi.org/10.1145/1131322.1131340

[31] Gengbin Zheng, Lixia Shi, and L. V. Kale. 2004. FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI. In *2004 IEEE International Conference on Cluster Computing (IEEE Cat. No.04EX935)*. 93–103. DOI: https://doi.org/10.1109/CLUSTR.2004.1392606