

Checkpoint/Restart Approaches for a Thread-Based MPI Runtime

Julien Adam^a, Maxime Kermarquer^b, Jean-Baptiste Besnard^a, Leonardo Bautista-Gomez^c, Marc Pérache^b, Patrick Carribault^b,
Julien Jaeger^b, Allen D. Malony^d, Sameer Shende^d

^aParaTools SAS, Bruyères-le-Châtel, France

^bCEA, DAM, DIF, F-91297 Arpajon, France

^cBarcelona Supercomputing Center, Barcelona, Spain

^dParaTools Inc., Eugene, United States

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 both transparent and application-level checkpointing mechanisms. Unlike the MPI 4.0 User-Level Failure Mitigation (ULFM) interface, our work targets solely Checkpoint/Restart and ignores other features such as resiliency. We show how existing checkpointing methods can be practically applied to a thread-based MPI implementation given sufficient runtime collaboration. The two main contributions are the preservation of high-speed network performance during transparent C/R and the over-subscription of checkpoint data replication thanks to a dedicated user-level scheduler support. These techniques are measured on MPI benchmarks such as IMB, Lulesh and Heatdis, and associated overhead and trade-offs are discussed.

Keywords: Checkpoint-Restart, Fault-Tolerance, DMTCP, Infiniband, Multilevel Checkpointing, MPI Oversubscribing

1. Introduction

The trend towards parallel high-performance computing systems with extreme numbers of cores, deep memory hierarchies, and multidimensional 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 internode communication. The evolution of *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 achieve this purpose:

- *explicit* requiring direct modifications in the code;
- *transparent* in the sense that they are able to checkpoint indifferently from the code itself.

One of the major stakes for end users is to select the C/R method fitting with application needs. This paper presents C/R optimizations leveraging runtime support for both these cases. Transparently checkpointing complex applications may lead to challenges when involving for example high-speed networks.

The application-level approach defers the C/R support to developers, using their knowledge to checkpoint only relevant data, whereas the runtime is, in most cases, more suited to deal with low-level notions like C/R data replication. In these two examples, we show how the runtime may collaborate to enable more efficient checkpointing.

In the rest of this paper we consider application-level and transparent checkpointing methodologies. We describe their respective implementation with the *Fault Tolerance library*[4] (FTI) and *distributed multithreaded check-pointing* (DMTCP)[2], contrasting their use and purpose. Section 2 starts by describing related work and Section 2.4 discusses various levels for checkpoint-restart and tradeoffs. Then, Section 3 presents the specificities of our MPI runtime executing MPI processes in user-level threads. The rest of the paper eventually describes and validates the integration of two fault-tolerance tools, FTI and DMTCP, with a focus on runtime oriented optimizations. More generally, we make the following contributions:

- We show how application-level checkpointing could rely on dedicated progress threads, positively taking advantage of oversubscribing;
- We demonstrate high-speed network checkpointing thanks to collaboration from MPI runtime;
- We introduce a compact collective checkpointing call for transparent C/R.

This work is an extended version of a paper originally focused on the sole integration of DMTCP in a thread-based MPI

runtime[1]. This new version features extended descriptions, additional contrast with respect to application-level checkpointing and more generally C/R trade-offs. In addition, we describe how we integrated FTI application-level checkpointing library to take advantage of user-level threads.

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. 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[28] and ACR[30] answer this by storing checkpoint files over faster, local mount points and replicate them to ensure redundancy. The *Fault Tolerance library (FTI)*[4] that we describe in more detail in Section 6.1 is also aimed at solving these issues.

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 difficult 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.

As far as application-level checkpointing implementation mechanism is concerned, incremental checkpointing [29] was proposed to reduce the amount of data to write in consecutive checkpoints, but the benefits of this technique are not always important. Thus, disk-less checkpointing [34] was proposed to alleviate this issue. With the arrival of new storage devices, multilevel checkpointing was proposed [4, 19], including a certain number of features, such as asynchronous transfers to the parallel file system. Semi-blocking algorithms have been proposed to save the checkpoint data without stopping the application execution[31], however this work does not leverage threading mechanisms as the one presented in this paper to safely and efficiently oversubscribe compute nodes and allow fault tolerance tasks to take place in an opportunistic fashion.

Oversubscribing, an approach we retained in this paper, has been scarcely studied. A complete survey of oversubscribing with the use of several parallel programming languages [23] shows that oversubscribing MPICH-2 MPI processes induces an overhead of 10% (equivalent to the one we observed with OpenMPI), while oversubscribing threads may improve overlap and recovering waiting periods. It has also been studied how bad placement of processes for checkpoint/restart may hurt performance [38]. Another work describes how, even if possibly harmful inside one application, oversubscribing can be used to efficiently execute multiple applications sharing one node [37]. To circumvent this drawback when applying MPI oversubscribing in a unique application, some work focused on enabling multiple MPI process in one OS process [26], verifying the positive impact of such implementation.

While it is possible to leverage external libraries that optimize certain support, 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)[5, 18]. 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[20] (BLCR) tool. BLCR relies on a kernel-level approach to both suspend and checkpoint. This has the advantage of avoiding a complete wrapping of every system call, 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, restoring UNIX pipes). 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 in MPI context. The parallel application has 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 resources before checkpointing [10]. However, it was limited to TCP protocol and 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. Indeed, as tools cannot be sure about the application behavior, all potential calls involving resources outside the process are to be captured, such as network or storage. The *Distributed Multi-threaded CheckPointing* (DMTCP)[2] 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 [15]. 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 transparent 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. MPI Failure Mitigation

The failure mitigation approach is more focused on how 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)[16, 22]. 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(ULFM)[8, 7], a solution implemented on top of OpenMPI, providing new MPI semantics that helps the application to recover process failures. This model defines a *state* at the 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[9, 36, 17].

2.4. Summarizing Checkpointing Approaches

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 startup. 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. There is then a clear tradeoff between these approaches.

As presented in Table 1, we compared various levels for checkpoint restart and describe the tradeoff they incur. The levels we considered, can be described as follows:

- **Application-level:** adding code or using dedicated libraries to implement checkpoint-restart inside the target application, for example using FTI;
- **User-level:** implementing (transparent) checkpoint restart through either state capturing or system call wrapping in user-space, for example with DMTCP;
- **OS-level:** (transparent) checkpoint/restart thanks to Operating System (OS) support possibly through a dedicated kernel module, for example with BLCR;
- **Hypervisor:** using capabilities of the virtualization environment to suspend, save and restart (transparently) a running virtual machine, for example with QEMU[5].

It can be seen that no solution is ideal, indeed being transparent comes at the cost of more overhead due to system-call wrapping and bookkeeping. Moreover, transparent methodologies are not able to extract the minimal dataset linked with application state and control, and generally save the complete memory image (either application or sometimes the full OS). This leads to larger checkpoints offset by benefits in development time as no modifications to the code are needed. The right checkpointing method is then probably a mix of those presented in this table and is yet to be defined. In this , we decide to not focus on one specific method but instead proposing multiple solutions on top of your MPI implementation, offering the user the ability to choose the right one for his own scenario. In the next section we are going to describe the thread-based MPI

Checkpoint Level	Selectivity	Size	Administrative Rights	Implementation. Cost	Transparent	Overhead
Application	High	Small	No	High	No	Variable
User-level	Low	Large	No	Small	Mostly	Medium
OS-level	Lower	Larger	Yes (install)	Null	Yes	Low
Hypervisor	Lower	Larger	Run virtualized	Null	Yes	High

Table 1: Comparison of the various checkpoint/restart levels

runtime in which we propose to integrate transparent and application level checkpointing, providing an initial context to the later developments.

3. MPC Overview

MPC [33] is a framework dedicated to the smooth integration of shared-memory parallel programming models in MPI applications. To this end, MPC provides different implementations such as MPI, OpenMP, and Pthread, all unified on top of the same user-level thread scheduler. By having its own MPI implementation and its own thread scheduler, MPC is then able to execute MPI processes in different configurations, as discussed below. One can consider that all MPI implementations fit into one of two categories: process-based and thread-based.

Process-based implementations are based on MPI Processes being regular UNIX processes, with separate address spaces. Most MPI implementations fit in this category, such as MPICH and OpenMPI. An indirect consequence of this state of things is that applications may feature global variables duplicated for each MPI process running as a UNIX process.

Thread-based implementations are using threads for running MPI processes. In this second configuration, multiple MPI Processes run in a single UNIX process and share memory. A hybridization is also possible by running multiple UNIX processes, themselves gathering several MPI processes. In this context, global variables are not fully isolated anymore, by construction.

To address this second configuration, MPC relies on a privatizing compiler to transparently separate global variable by creating multiple copies of it for each MPI process, thanks to a hierarchical TLS storage approach[6]. It is then possible to port C, C++ and Fortran applications with little effort to this thread-based configuration. The advantage of running in such configuration is that the scheduling of MPI processes does not rely necessarily on the OS scheduler, but on a user-level thread scheduler. In addition, intranode communication is simplified, in that regular shared memory copies can be used, instead of relying on SHMEM or CMA to achieve the same effect. MPC encompasses these two “flavors” of MPI processes[32]. These configurations are displayed in Figure 1. On the left, the usual process-based MPI model completely separates components for each MPI process. With the process-based flavor, each MPI process has its own thread scheduler instance, its own allocator instance, working on independent address spaces. On the right, thanks to the thread-based approach, more components are shared between MPI processes. This induces a reduction of the global memory footprint since some internal structures

(such as network buffers) are not duplicated. Besides, multiple MPI processes on the same node rely on the same MPC scheduler, potentially bypassing the OS scheduler

4. Contribution

Here we explore more extensively two use cases of checkpoint restart in the context of a thread-based MPI runtime. First, we consider the integration of DMTCP in the MPC runtime to provide transparent checkpoint-restart capabilities. Second, we focus ourselves on the FTI library and how it was integrated in MPC’s unified scheduler. For each of these examples we will follow the same plan. We will introduce the approach in general, contrasting it with other methodologies and explaining how it is implemented. Then, we will detail how it was integrated in our thread-based runtime. Eventually, we present for each model a commented performance measurement, outlining respective advantages and limitations.

As far as transparent checkpointing is concerned, we present a simple collective call enabling checkpointing at coherency points with respect to communications. In addition, we detail how we developed a signaling network enabling transparent restart through DMTCP. Our performance results on Lulesh at scale, show that transparent checkpointing is possible without sacrificing network performance. However, there is a trade-off with respect to checkpoints as our methodology supposes that the high-speed network is closed during each checkpoint, requiring a transitive overhead which can still be mitigated as we further elaborate.

Dealing with application-level checkpointing, we considered the FTI checkpointing library and explored alternative approaches with respect to the mapping of helper processes. Indeed, by default, FTI relies on MPMD to map a process in charge of data-replication on each node. In our model, we moved this process inside a thread running in an oversubscribed fashion over an user-level scheduler. In particular, we show slight improvements thanks to this collocation in threads, instead of processes. But still, as MPC does not integrate scheduling points inside I/Os the overall gain remains limited in terms of I/O recovering.

Overall the rest of this paper is an assessment of how checkpointing could be integrated inside a thread-based MPI. We show not only that it is possible with some advantages. But the most important conclusion is that doing so is not different from what would be done with a “regular” process-based MPI. In fact, for the transparent C/R examples concerns were focused on how to save and restore the network state aspect which is directly translatable to other MPIs.

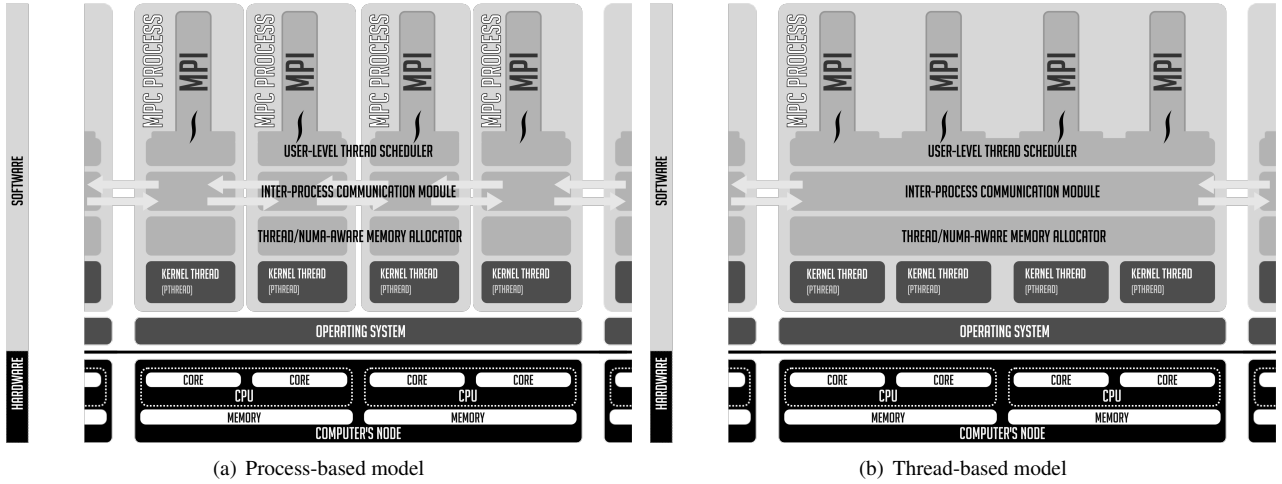


Figure 1: MPI implementation flavors

5. Transparent Checkpoint-Restart with DMTCP

In this Section we focus ourselves on transparent checkpointing inside the MPC thread-based MPI runtime. The goal of such approach is to enable C/R with limited developments (hence the transparent adjective). However, to achieve such thing, a careful handling of application’s state is compulsory. Indeed, while checkpointing a serial program is straightforward, the distributed nature of an MPI application requires a restoration of the network connectivity. The rest of this section aims at describing how we made such restart possible in MPC MPI. To do so, we first introduce DMTCP, then after recalling MPC’s network structure (bootstrapping, multi-rail) we cover the integration of DMTCP itself. To do so, we present a simple collective checkpointing interface that dodges the complex issue of in-flight messages. Eventually, we conclude this section with a performance study done with the Lulesh benchmark.

5.1. DMTCP Overview

We consider checkpointing through the user-level transparent approach using DMTCP [2] which is the distributed implementation of MTCP [35], 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 alive 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 connections 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 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 injected 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 and as we will further describe, a dedicated synchronization mechanism is required.

5.2. Network Modularity in MPC

As our support for transparent checkpointing is based on the ability to close a network rail and then restoring it, this section is dedicated to describing how we managed to expose sufficient modularity in our communication layer to enable such support. MPC’s low-level 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 network using *control messages* (“signaling” messages) routed on a base topology. With this mechanism in place, we outline how it contributes to transparent checkpointing by preserving high-speed network capabilities.

5.2.1. Multi-Rail

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 (latency lower than the μsec), and (2) for internode communications, where remote-direct memory access (RDMA) support could be used to optimize MPI’s performance (in the μsec range). The switch between intranode and internode 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 then 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 1, 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. Thus, message headers in MPC carry both MPI process id (internal to MPC) and task id. The process id is used to determine which UNIX process hosts a given MPI process. As a consequence, there is no direct correlation between a communication endpoint and a given MPI process. Indeed, a given network is only initialized once per UNIX process and therefore multiple MPI processes will share the same network layer. Several situations can result. For instance, messages could be exchanged inside a given MPI “process” if both tasks are running in shared memory. 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. Moreover, these means of exchange are not mutually exclusive and hybrid configurations involve both messaging layers depending on peers.

In the rest of this paper, as far as transparent checkpointing is concerned, we will consider solely the communication between UNIX processes, as it is the only part of MPC involving internode communications interacting with network cards. This put us in the process-based case 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 1.

Communications in MPC are based on endpoints belonging to a communication rail (see Figure 2). Endpoints are sorted by priority inside ordered lists corresponding to a given remote

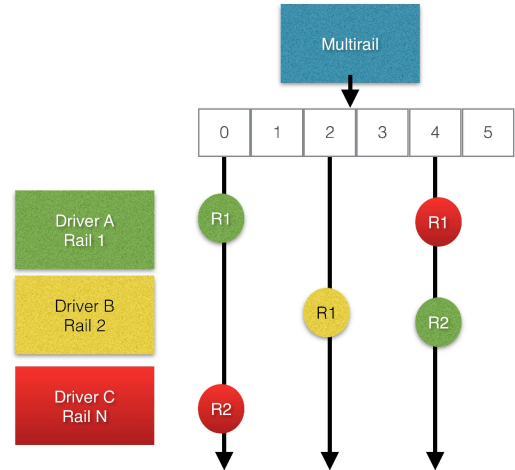


Figure 2: Overview of the multi-rail infrastructure in MPC.

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 5.2.3 will detail how this *on-demand* connection process is implemented in MPC. If this succeeds, MPC 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 3, MPC’s multi-rail support relies on an XML configuration file that we now describe bottom to top. First, we define a Command Line Interface (CLI) option 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 establish additional networking configuration.


```

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

```

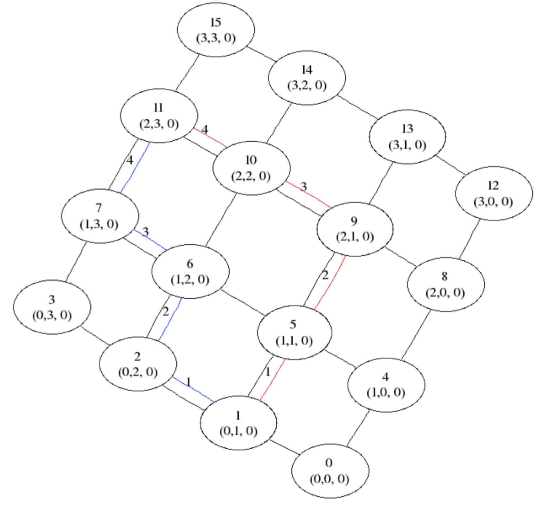
Figure 3: Example of XML configuration file for MPC’s multi-rail engine.

We refer to this as the *signaling network* for MPC.

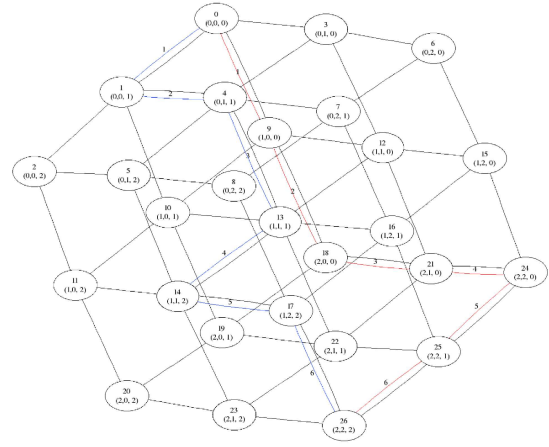
5.2.2. Signaling Network

MPC’s network layer can also be used to provide a signaling 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) in 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.

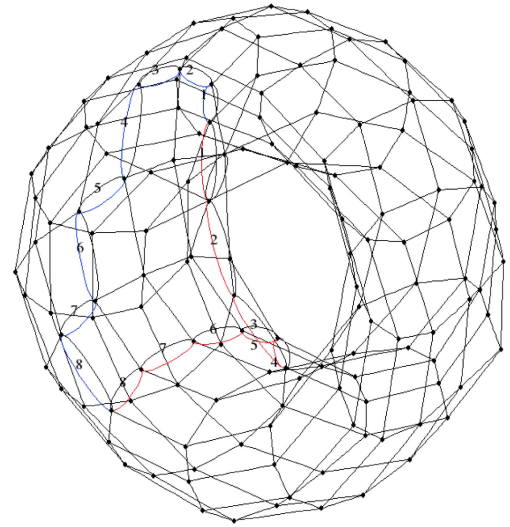
One of the main proprieties of the signaling network is its capability to route messages according to a simple 1D distance metric, defined as the absolute value between the source and target ranks. The reason for retaining such a simple metric is because we wanted it to be portable on any topology, indifferently 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 the 1D distance metric is not sufficient to escape from a local minimum.



(a) 2D-Mesh



(b) 3D-Mesh



(c) 2D-Torus

Figure 4: Routing comparisons on various topologies with 1D in red, and 3D distances in blue.

Despite managing to limit the theoretical network diameter with denser topologies, the 1D distance metric we retained to ensure routing robustness for arbitrary topologies 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 between 1D and nD distances in Figure 4.

5.2.3. Network Bootstrap

As presented in previous Sections, MPC implements a multi-rail engine and a signaling 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 have 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 thousands or even millions of 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 3). 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 until their 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 leads to important overheads, for example, in the case of Infiniband. Instead, such tools are limited to solely restoring TCP sockets between processes. As we will further discuss, 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 remov-

ing all references to the associated network. It is this mechanism, combined with signaling, that enables MPC’s transparent checkpointing capabilities without wrapping network calls.

5.3. DMTCP Support

We leverage the DMTCP checkpointing tool to transparently save the state of an MPI program. In particular, we show how the MPI runtime can work with a transparent checkpointing tool to enable support for high-speed networks. 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 without setting up a connection to the Host-Channel Adapter (HCA) within the process first. If we omit the question of high-speed networking, checkpointing with DMTCP is transparent and relies on submitting requests to a daemon in charge of the process without synchronization from the application or the runtime. However, to deal with high-speed networks in a more efficient manner, a contribution from the runtime is necessary to avoid large overheads. 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. As far as transparent checkpointing is concerned this paper makes the following contributions:

- The definition of a collective checkpoint interface enabling transparent checkpointing in MPI runtimes (Section 5.3.4);
- The concept of an in-band signaling network with the associated routing, and the use of multi-rail logic to enable partial checkpointing (Section 5.2);
- General MPI implementation of transparent checkpointing including high-speed networks.

This 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 [14] and sessions [21], which involve intra-process parallelism.

Given the MPC infrastructure, the following presents a general methodology enabling transparent checkpoint/restart for programs using 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.

5.3.1. Thread-Based MPI Checkpoint

DMTCP and its coordinator are designed so that a single request for the checkpoint is automatically broadcast 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 5, MPC solves this by proceeding to a first intra-node barrier between MPI processes 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.

5.3.2. 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 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*[12].

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 (Thread-Local Storage) indirection[6], similar to how FS register is used in common Pthread implementations 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 processes inside a single UNIX process). This has been discussed with the developers¹ 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 by 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

This leads to a stack overflow by creating a loop. This code seems to be present solely for IntelMPI resolving PMI_Init with dlsym(RTLD_NEXT, "PMI_Init"). The call is currently not compiled conditionally. We reported it to the developers², but our current workaround is simply to comment it out. This is done in the version of DMTCP which is bundled in MPC.

5.3.3. 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[11]. This approach enabled transparent checkpointing of Infiniband networks, but not without some drawbacks.

As presented in Figure 6, 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 5.2). Moreover, on-demand connections are managed with in-band messages, which can be routed through a dedicated signaling network (Section 5.2.3). Then, without any action, routes existing prior to 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 the state of the network layer when it has been stopped at checkpoint time, potentially leading us to deadlocks.

Thus, we consider removing these routes before checkpointing the application. Rails not checkpointable had to be fully

¹<https://github.com/dmtcp/dmtcp/issues/607>

²<https://github.com/dmtcp/dmtcp/issues/604>

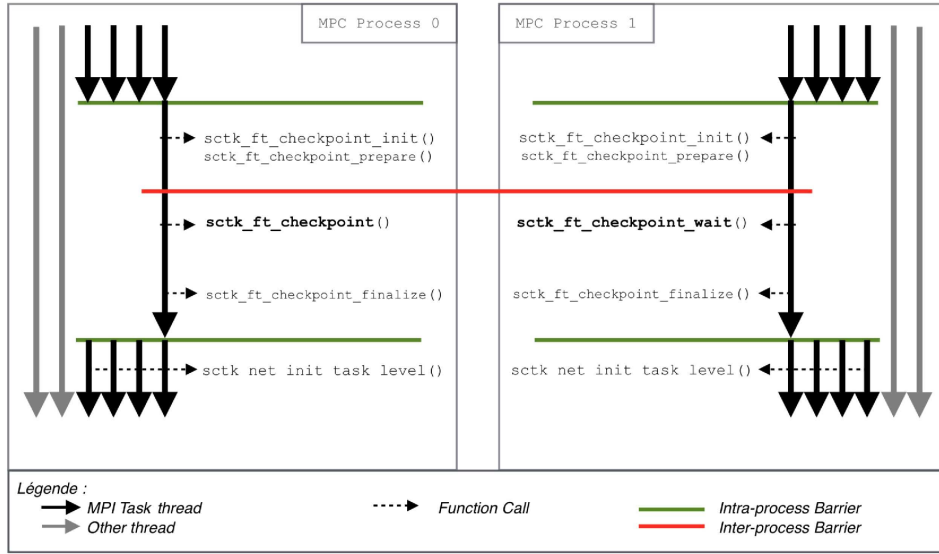


Figure 5: Two-level synchronization scheme enabling checkpointing in MPC.

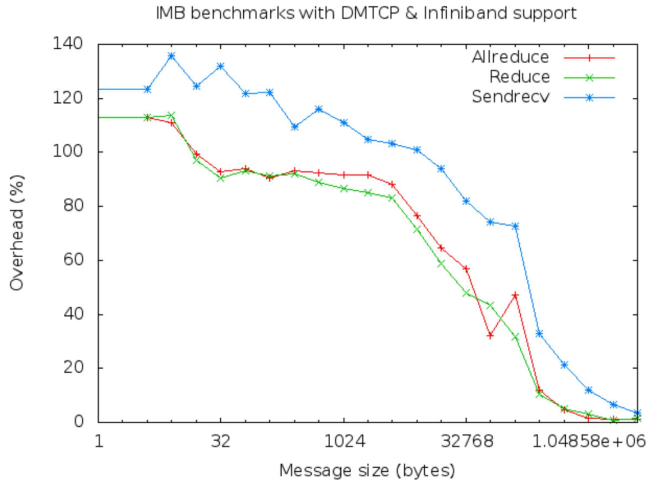


Figure 6: Overhead for Infiniband wrapping in DMTCP.

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 2). 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 requiring to be closed, there will be no route associated with these rails in the restarted process image, a new rail will be allocated from scratch.

5.3.4. 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 7: 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 2: MPIX.Checkpoint constants definitions.

As presented in Figure 7, the MPIX_Checkpoint call is a collective with respect to MPI_COMM_WORLD. It will return to a state defined in Table 2. Entering this function means the application is requiring the MPI implementation to create a new checkpoint, there should be no unmatched MPI messages to prevent message losses. One point to note is that this call can return in different scenarios. First, a program returning from a regular checkpoint proceeds to call MPIX_Checkpoint. In this scenario, the function call will return each time CHECKPOINT when the step completes. When the application program restarts, the work-flow will immediately come from MPIX_Checkpoint and the return value will be RESTART, allowing the application to be notified of the current state (post-checkpoint or restart). If it is not possible to checkpoint (e.g., due to lack of support), a runtime can return IGNORE to inform the application that nothing was saved.

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.

5.4. Evaluating our DMTCP Integration

It is important to observe that a direct consequence our transparent checkpointing 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 go 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 2) 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 that DMTCP will try to drain, eventually leading to a deadlock. Consequently, dynamic route closing and its associated performance impact appeared to be a good tradeoff 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.

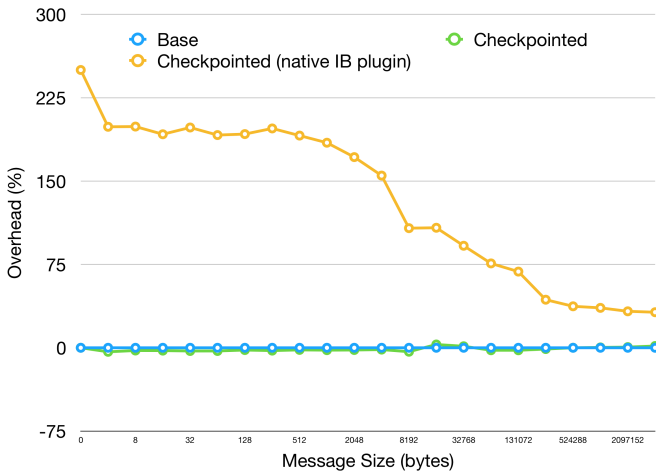


Figure 8: IMB Allreduce performance overhead between DMTCP Infiniband support and MPC’s support.

Figure 8 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 and 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 is free to set its

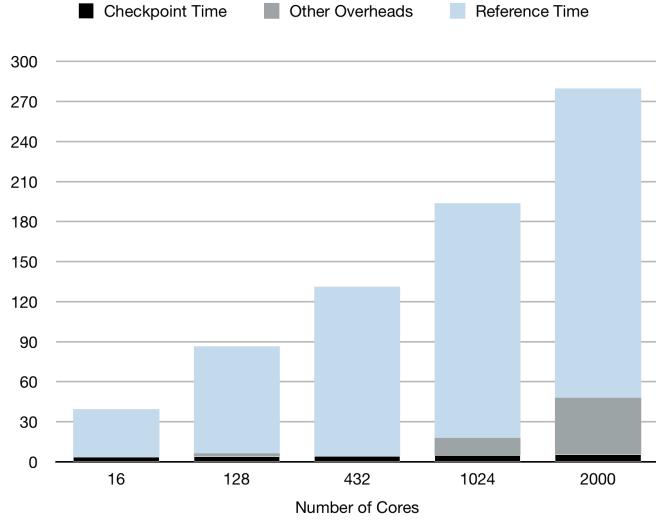
frequency. In summary, our method creates a transient overhead to prevent a permanent one.

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 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 second, we have the following total duration D of 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.

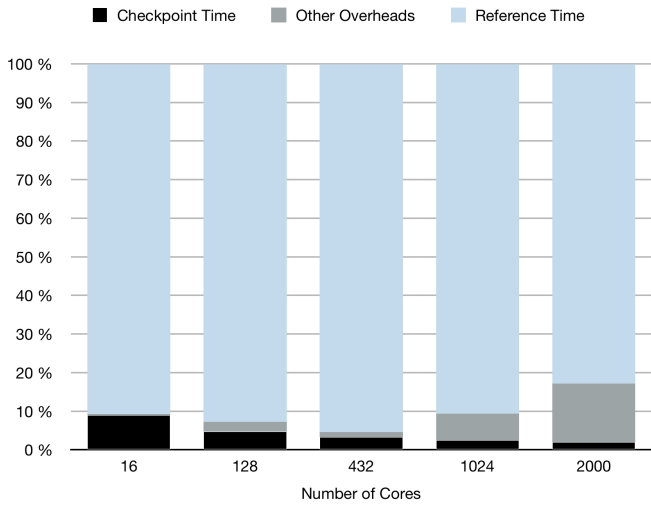
5.4.1. DMTCP Performance Evaluation on Lulesh

In order to assess the performance of our checkpointing methodology, we ran it at scale on the Lulesh[27] 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 small test system at CEA featuring Sandy-Bridge processors and 16 cores per node using a problem size of 30. Interconnect consists in mlx4 Infiniband Host-Channel Adapters. In 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+OpenMP configuration. Runs were done with a fixed size of 30 (`-s` flag), given Lulesh’s design this size is given as the size per process and therefore our measurements are all done in a weak scaling fashion – problem size is 30^3 per MPI process.

In Figure 9(a), we see the breakdown of the walltimes in terms of reference time, checkpoint and other overheads. The checkpoint overhead is the time spent generating the data in the collective call, other overhead accounts for other differences with respect to reference time, including on-demand connections. In Figure 9(b), we present the same results as a percentage, to highlight the relative cost of each time. What can be seen that that the checkpointing time by itself remains relatively steady as the number of cores increases. However, we observe a rise in indirect costs from 0.3 % up to 18 % when considering large scale. This can be explained by the high number of on-demand connections, as the number of nodes gets larger. This



(a) Walltime breakdown in seconds



(b) Walltime breakdown in percentage

Figure 9: Checkpoint and reference times for Lulesh (size 30) in function of the number of cores.

overhead, mitigating the I/O saturation effect linked with an increasing number of MPI processes saving their state in parallel is also correlated to the transitive cost of disconnecting and then reconnecting MPI processes. This cost is then clearly not negligible. However, and as we are now going to discuss, the nature of the checkpoint and in particular its punctual nature can be used to slightly reduce performance impact.

5.4.2. Checkpointing Lulesh with Constant Overhead

In order to expose these results in a more practical manner, we used the formula presented in the previous section to compute the checkpointing period such as the overhead is 1% in the light of previous measurements. To do so, we added direct and indirect overheads, considered as the total checkpoint cost. This yielded the values presented in Figure 10. One can see that despite potentially expensive, the checkpoint cost on the walltime can be mitigated for long-running programs. In our case, for

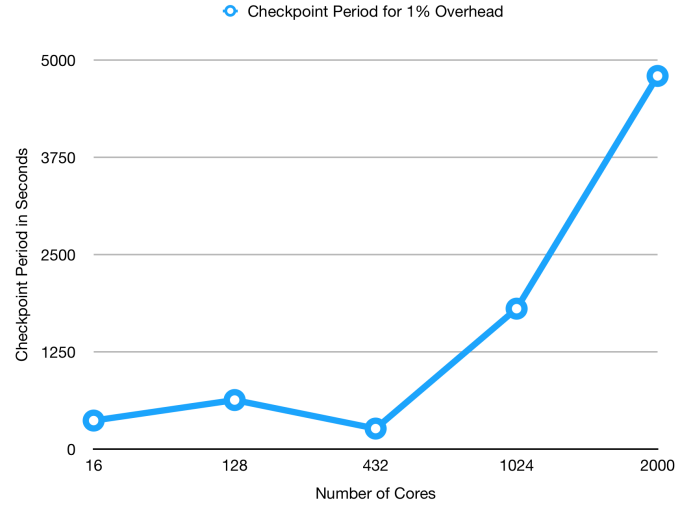


Figure 10: Checkpoint period for 1% overhead computed from results presented in Figure 9.

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 rapidly increases requiring checkpoints to be further spaced to mitigate their apparent cost. This can find its explanation in several factors that we described as *other overheads* in Figure 9. Indeed, the number of connections to be restored is dependent from communication topology which is polynomial in the case of Lulesh’s 3D mesh. In addition, the saturation effect on I/O caches and more generally the file-system is not to be neglected as the number of processes increases. These two factors are possible explanations to the important overhead as the number of cores increases. However, we think that our methodology is still usable 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 scale but also the data-set size manipulated by the application – transparent approach dumping full process images. Dealing with this later constraint, application-level checkpointing has a lot of advantages as it can benefit from application developer’s input at the impediment of the associated programming cost.

5.4.3. Generalizing to other MPI Implementations

Results presented in this paper were obtained with the MPC runtime which provides support for checkpointing. In particular, we presented a dedicated collective call `MPiX_Checkpoint` and relied on high-speed network disconnection prior to checkpointing. In addition, MPC include the signaling 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 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 (2) the 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 depen-

dent on MPC, it simply requires state management capacities in MPI for connections and startup – DMTCP handling most of the checkpoint.

6. Application-level Checkpointing

In this second part we focus ourselves on how checkpointing could be achieved at application-level. In previous Section we have seen how it was possible to save the distributed state of an MPI application without requiring substantial modifications, just a single line of code at time-step level. However, and as we also commented this method despite practical is far from optimal as it supposes the saving of more than actually needed by the application. This leads to inefficient checkpoints in terms of memory and overhead linked to the associated I/Os. This second approach requiring application developer input is then more efficient in terms of storage space, although it requires the program state to be fully serializable. This supposes that the state of each library can be correctly intercepted which is sometimes not practical, considering for example third-party software. In this Section, following what we have already done for DMTCP, we will first introduce the FTI library which aims at exposing convenient mechanism for application-level checkpointing. Then, we present our integration inside MPC. We show some performance results in a heat-dissipation benchmark which was easily ported to FTI. We also ported Lulesh on top of FTI to enable further comparisons with the transparent method. This comparison, however, has to be mitigated as measurements were made on a different machine due to organizational constraints.

6.1. FTI Overview

FTI is a multilevel checkpointing library with a wide set of features. The purpose of this library is to provide an interface to address the various storage levels in high-performance computing environments for checkpointing purposes.

High-performance computing is an always evolving field in which new hardware devices are continuously being developed and integrated; not only for computing but also for storage. As a consequence there exists a discrepancy between types of storage in terms of performance, availability and reliability. For instance, mechanical hard-disk drives usually offer higher capacity but lower performance than solid state drives. Such trade-offs are at the core of the concept of multilevel checkpointing, with the goal of finding the sweet spot in the reliability versus performance trade-offs. To further illustrate this, FTI offers the following four checkpointing levels:

- Level 1. Checkpoint in local storage.
- Level 2. Local checkpoint and copy on a partner node.
- Level 3. Local checkpoint with erasure coding.
- Level 4. Checkpoint in the PFS.

Level 1 checkpoint is the least reliable level but also the fastest, while Level 4 is the most reliable but also the slowest of all levels. Given that most failures in supercomputers do not

affect all nodes simultaneously, there exist possibilities to combine specificity from each level to yield improved performance, this is the goal of the FTI interface that we describe in the next Section.

Writing checkpoints in local storage is sufficient to put up with soft errors but cannot withstand node failures, data stored in the local storage being inaccessible until the node is repaired. Therefore, local checkpointing has to be combined with some sort of data redundancy in order to tolerate one or multiple node crashes. FTI implements several approaches for this purpose, such as data replication on a partner node, data redundancy through Reed-Solomon encoding or data persistence into the parallel file system. Application-level checkpointing thanks to its improved data selectivity is an efficient method in terms of minimizing data to be saved. In addition, addressing different types of storage (e.g. node local) further enhance performance. Nonetheless, checkpointing remains an expensive operation. Therefore, it can be optimized by dividing it into two stages: (1) write the checkpoint in local storage and (2) apply data redundancy in the background while the application continues its execution.

Data redundancy techniques systematically require some sort of processing, either by transferring data through the network or by performing extra computations. This additional work can be done locally, impeding extra overhead to the application or using dedicated hardware (like a RAID array). It is that capacity we want to provide more efficiently thanks to the runtime support.

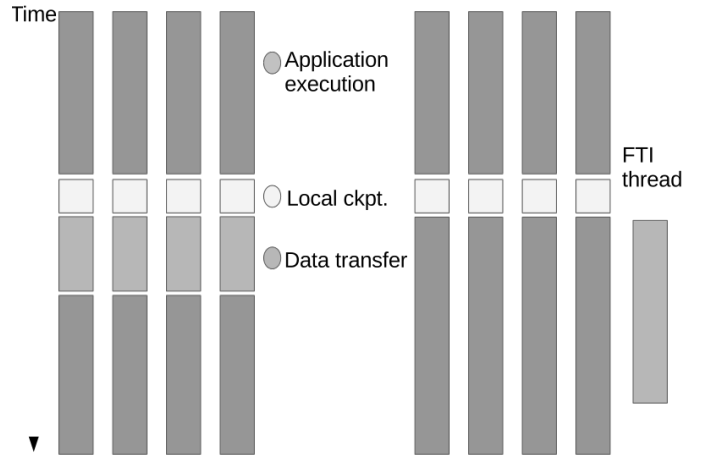


Figure 11: FTI synchronous vs asynchronous transfer

As far as this post processing is concerned, FTI offers the option to “steal” one process per node from the application in order to perform these tasks. In this case, the dedicated processes are isolated from the application processes by splitting the global communicator and providing a new one to the application. The FTI helper processes then run in their own communicator to perform their resiliency-related tasks.

Using this technique, application processes can pursue their execution as soon as the local checkpoint has been made. Data replication is offloaded to these helper processes in parallel to

the regular execution as shown in Figure 11. This has been proven to be quite efficient, because it virtually transforms all checkpoints into local checkpoints. Unfortunately, this supposes the use of dedicated resources for this purpose. Moreover, most large scale supercomputers do not allow applications to run more processes than there are on a given node due to batch-manager constraints. For this reason we explored the possibility of relying on oversubscription in MPC’s user-level scheduler to perform such tasks in the background without dedicating specific resources.

6.2. MPC’s Unified user-level thread scheduler

Before explaining how we integrated FTI in MPC, this Section recalls some aspects of MPC’s scheduler. First, it handles user-level threads, bypassing the OS scheduler, often ill-suited for HPC parallel applications. An MxN user-level scheduler as in MPC bypasses the OS scheduler, one OS thread is created per computing unit on a node, and pinned to a given computing unit. On top of this thread, the user-level scheduler handles the selection of user threads to be executed on this OS thread. This way, scheduling decisions, previously delegated to the OS scheduler, are now handled by the user-level threads in-place, as there are as many OS threads than cores. Scheduling policies are then completely handled by the user-level thread scheduler.

By collocating multiple programming models in its user-level scheduler, MPC is able to coordinate the execution of threads from various origins. With its global view of the whole node, all available computing units and threads to be executed, the scheduler can then make the best decision possible according to the scheduling policies. This feature could be illustrated with oversubscribing. Indeed, since everything is a thread the scheduler knows when a thread is idle and may replace it with another active one. In the case of oversubscription, it means that as soon as a thread is idle, an extra thread can be scheduled to use these idle resources. This thread may originate from any of the active models. This helps maximizing the usage of available resources, and reduces performance loss, due to waiting and idle threads. It is this last propriety that motivated the integration of FTI inside MPC, exploring the possibility of collocating the *helper* process in an oversubscribed thread instead of a POSIX process.

6.3. Supporting FTI in MPC

Now that we presented our integration of DMTCP inside MPC, this Section focuses on the integration of application-level checkpointing in the context of the FTI library. As explained in previous sections, MPC provides its own implementation of the MPI standard. Since FTI is relying on MPI to implement its checkpoint/restart method, we simply used MPC as the MPI implementation for FTI. The rest of this section, we first detail the port of FTI atop MPC and then describes how checkpoint data post-processing took advantage of MPC’s user-level scheduler and oversubscription.

6.3.1. Port of FTI atop MPC

This step was simple to achieve. As MPC is an MPI implementation, the only necessary action was to compile FTI with MPC compiler wrappers (`mpc-cc`) relying on “automatic privatization”.

After checking that FTI and MPC were correctly collaborating both in process-based and thread-based configurations, we sought to benefit from MPC’s specificities. In particular, we moved the dedicated MPI process inside a user-level thread to benefit of oversubscription.

In Section 6.1, we explained how having a dedicated MPI process for post-checkpoints per node improves performance. However, this is mainly true when the MPI process runs on its own resources. If no core is available, this additional MPI process will be oversubscribed. It means that this additional MPI process shares resources with the original application. As two processes, or even threads, cannot run at the same time on the same core, their respective code will be executed, turn by turn, after context switches.

In order to mitigate oversubscription overhead, we targeted MPC’s thread-based MPI capabilities. The interest is twofold with (1) lighter context switches and (2) the ability to use MPI waiting time (in the application) to progress checkpointing. We saw in Section 6.2 that the MPC scheduler uses its own user-level threads. Hence, in a full thread-based mode, each MPI process on a node is a user-level thread managed by a unified scheduler. Switching from one MPI process to another is “just” a user-level thread context switch, which is lighter than between two UNIX processes. This then makes the approach involving an oversubscribed MPI process more attractive than in a regular process-based MPI setup.

6.4. Evaluating the FTI Integration

Now that we presented the results with user-level transparent checkpointing, this Section now studies the impact of our MPC integration on FTI for application-level checkpointing. In particular, we compare performance between additional MPI processes and our oversubscribed model, taking advantage of user-level threads.

In a first approach, we ported Lulesh on FTI. To do so, the first step was to change all `MPI_COMM_WORLD` references to `FTI_COMM_WORLD`. If such replacement can be tedious on large production code, the LULESH code infrastructure simplified this change as all MPI calls are located in only two files. The next phase to port an application to FTI relies on the data election to be saved for checkpoint. Here again, the code structure was conveniently outlining important data in a single C++ class. However, one highlighted issue by porting LULESH to FTI was this C++ class. Indeed, FTI can handle C structures natively with `FTI_InitType()`. However, this checkpointed dataset includes a tree-like structure relying on pointers. As such pointer cannot be saved due to memory remap upon restart, we had to serialize the structure. As a consequence, this C++ class object is serialized into a buffer, handled to FTI. Thus, before checkpointing, this serialization had to be performed. Symmetrically, this buffer is de-serialized into

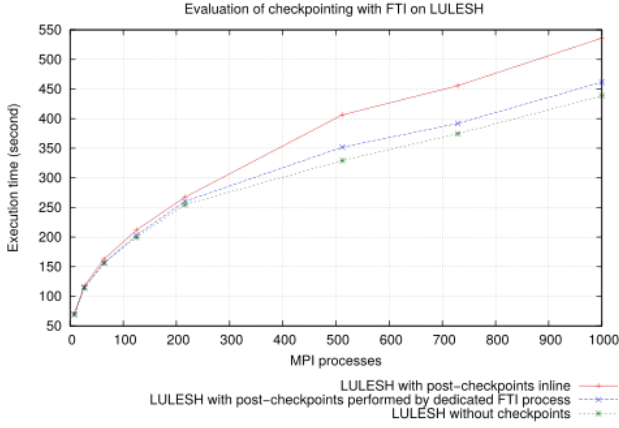


Figure 12: Performances without and with FTI checkpointing methods, no oversubscribe

the C++ object upon recovery. The BOOST library was used handle the serialization. The port of LULESH to FTI, and the serialization process were validated by arbitrarily killing the job at different times before recovering the job and resuming execution. All program outputs remained valid whatever the number of MPI processes (up to 1728).

As presented in Figure 12 which does not rely on oversubscription, using a dedicated checkpointing process is advantaging when compared to the inline approach which does not provide any overlap. This shows that there is an interest in integrating such support though an user-level scheduler. Since Lulesh works with numbers of MPI processes which are power of 3, it was not possible to produce a sufficient number of configuration where all cores are loaded with computation, and thus to realize oversubscribing when using a dedicated FTI process for the checkpoints. We tested our oversubscribed application-level checkpoint restart approach on a heat distribution benchmark (heatdis). Heatdis is a 2-dimensional stencil code that distributes a 2D grid among MPI processes. Processes only communicate with neighbor processes for exchanging ghost cells. As this benchmark does not impose restriction on the number of MPI processes (unlike Lulesh), we were able to validate multiple configurations. Performance measurements were realized on the MareNostrum 3 supercomputer at the Barcelona Supercomputing Center (BSC). MareNostrum 3 is a 1.1 petaflop peak performance supercomputer with Intel SandyBridge processors. The machine features 3056 nodes connected through an Infiniband FDR network. As presented in Figure 13, we ran this benchmark in different configurations:

- without FTI to provide a base time;
- with FTI and inline post-processing;
- with FTI and a dedicated oversubscribed MPI process (running as thread).

It can be seen that when relying on a thread to perform post-checkpointing operations the overhead is slightly lower than if it was done inline, directly impacting the code. This shows

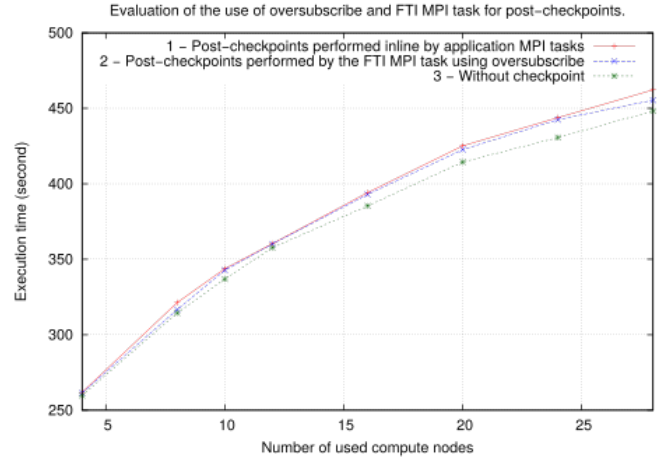


Figure 13: Oversubscribing with an MPC MPI thread

that such a model can lead to some benefits when being used in threads. However, gains are still relatively limited. We think that the non-preemptive nature of the scheduler and the fact that our integration of file I/Os in MPC is not fully taking advantage of the scheduler, preventing MPC from inserting yielding points inside I/O operations (when being blocked in a write, for example). Indeed, integrating a model inside user-level threads requires generally a complete wrapping of every call to avoid cases potentially blocking the OS thread carrying the execution. For this reason, mutexes, semaphores, Pthread operations, and so on, are captured by MPC to be managed accordingly in the unified scheduler. We think that converting I/Os to non-blocking operations and accounting for it in the scheduler as yield points should bring improved performance compared to what is presented in Figure 13.

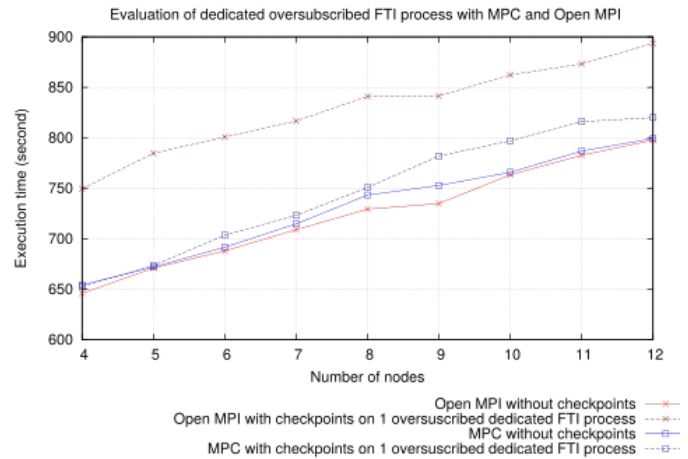


Figure 14: Comparing oversubscribe with MPC and OpenMPI

However, for users willing to fully use all cores by oversubscribing the FTI processing instead of inlining it, performance gains can be obtained with user-level threads. Indeed, as depicted in Figure 13, the same benchmark has been run with MPC (user-level threads) and OpenMPI (UNIX processes)

in order to compare oversubscription costs. Since the previous node configuration was not available during these tests, the comparison has been realized on a Sandy Bridge partition, with 16 cores per node. The Heatdis benchmark has been initially run on both MPC and OpenMPI, with disabled checkpointing, yielding to similar results. However, as far as oversubscription is concerned, it can be seen that user-level threads yield lower overhead than processes – OpenMPI showing an additional overhead between 10% and 15%, depending on the number of nodes. One of the main reasons for this is that MPI is designed to efficiently poll the resources of its core to progress communications with minimum latency whereas in MPC, this additional process directly benefits from the existing shared-memory communication layer. This allows the helper MPI processes to yield to a computation process in a fairer manner, limiting the overhead.

To summarize, the use of FTI inside user-level threads shown that oversubscription was more efficient than with OS processes. Reasons for this are the unified communication layer mitigating potentially aggressive polling, leading to improved fairness between threads and more efficient context switches. However, we observed reduced gains when it comes to I/O integration in the scheduler as we did not integrate yield points inside POSIX I/Os in MPC’s non-preemptive scheduler, mitigating the potential overlap of an I/O intensive thread such as the one exposed by FTI. We are considering to address this issue as future work.

7. Conclusion

The paper presents our implementation of transparent checkpointing in the MPC MPI runtime. Based on our knowledge, it 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, like Charm++[24] and its combination with AMPI[39]. 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[40] and restarting the program on a different number of processes[25]. 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 restarts from scratch, 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.

In addition, we focused ourselves on application-level checkpointing with the help of the FTI library which targets multi-level checkpointing by providing application developers with a dedicated checkpointing API. The approach has the advantage of benefiting from developers’ knowledge to limit the checkpoint size when compared to transparent approaches. However, it has the drawback of requiring modifications in the tar-

get application. Consequently, despite yielding the same checkpoint/restart result, the application-level model has different implications and can then be seen as complementary to transparent methods. Our integration in the MPC runtime relied on user-level threads to perform post-checkpointing processing (data replication) and demonstrated that in terms of oversubscription, where MPI processes running in threads were more efficient than those using regular UNIX processes. Although, MPC’s scheduler is lacking when it comes to handling blocking I/Os in a non-preemptive manner, we would like to address in the future.

In this paper we presented two approaches for checkpoint-restart in the context of a thread-based MPI called MPC. In particular, we considered two different kinds of approaches at application- and user-level and discussed how they collaborated with our runtime either at communication level or with a unified scheduler. This showed that runtimes can provide mechanisms to improve checkpointing efficiency and that such mechanisms were applicable in a relatively straightforward manner to the specificity of a thread-based MPI runtime.

8. Future Work

We see several tracks of enhancements following this work. As far as the FTI model is concerned, thread migration would allow oversubscribed MPI processes to take advantage of idle time more efficiently as they are currently stuck in the scheduling list of a single OS-level thread. In addition, the integration of I/Os in the scheduler using non-blocking file descriptors and a wrapping of the POSIX I/O interface should improve performance when it comes to oversubscribed I/O intensive payloads.

Regarding transparent C/R, current implementation in MPC has been designed to provide an initial support saving our users from the development of their own solution. However, checkpointing and more generally fault tolerance, for example through ULFM, allows a much wider range of scenarios. Indeed, our runtime has to fully restart in order to recover from a single node failure. The overhead currently impacts both checkpoint and restart phases. Closing the network could be considered a waste of time if no failure occurs between two checkpoints. Another idea would be to save the full network structure – updating DMTCP accordingly to disregard such network – and paying the price of cleaning it only at restart.

We would like to explore partial checkpointing with spare nodes leveraging our signaling network. Another aspect that seems promising is the exploration of connection-less networks and how they might be checkpointed more efficiently than by actively disconnecting-reconnecting peers. In particular, we are considering the Bull Exascale Interconnect (BXI)[13] Portals 4 network[3] to develop such support.

Acknowledgements

This research has been partially sponsored by the European Unions Horizon 2020 Programme under the LEGaTO Project (www.legato-project.eu), grant agreement 780681 and the Mont-Blanc2020 project, grant agreement n. 779877.

Appendix A. Usage Example

In this Appendix, we give a quick overview of how to launch MPC with transparent checkpoint-restart support. First, you need to install the last release of MPC with `--enable-mpc-ft` option. This should install DMTCP and enable its support in the code. When you proceed to launch the code with `mpcrun` you may pass the `--checkpoint` option in order to enable DMTCP's preloading and provide either checkpointing capabilities through the coordinator or via the `MPICH_Checkpoint` call. Eventually, to restart a checkpointed program you may simply use the `mpcrun` command with the `--restart` option, taking as an optional argument the path to the restart script generated during the checkpoint (current directory by default). Eventually, we recommend relying on the Slurm launcher as it is the most widely supported by DMTCP.

References

- [1] Adam, J., Besnard, J.B., Malony, A.D., Shende, S., Pérache, M., Carribault, P., Jaeger, J., 2018. Transparent high-speed network checkpoint/restart in mpi, in: Proceedings of the 25th European MPI Users' Group Meeting, ACM, New York, NY, USA. pp. 12:1–12:11. URL: <http://doi.acm.org/10.1145/3236367.3236383>, doi:10.1145/3236367.3236383.
- [2] Ansel, J., Arya, K., Cooperman, G., 2009. Dmtcp: Transparent checkpointing for cluster computations and the desktop, in: Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on, IEEE. pp. 1–12.
- [3] Barrett, B.W., Brightwell, R., Hemmert, S., Pedretti, K., Wheeler, K., Underwood, K., Riesen, R., Maccabe, A.B., Hudson, T., 2018. The portals 4.2 network programming interface. Sandia National Laboratories, November 2012, Technical Report SAND2018-12790.
- [4] Bautista-Gomez, L., Tsuboi, S., Komatitsch, D., Cappello, F., Maruyama, N., Matsuoka, S., 2011. Fti: High performance fault tolerance interface for hybrid systems, in: 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC), pp. 1–12. doi:10.1145/2063384.2063427.
- [5] Bellard, F., 2005. Qemu, a fast and portable dynamic translator, in: USENIX Annual Technical Conference, FREENIX Track, p. 46.
- [6] Besnard, J.B., Adam, J., Shende, S., Pérache, M., Carribault, P., Jaeger, J., Malony, A.D., 2016. Introducing task-containers as an alternative to runtime-stacking, in: Proceedings of the 23rd European MPI Users' Group Meeting, ACM. pp. 51–63.
- [7] Bland, W., Bouteiller, A., Herault, T., Bosilca, G., Dongarra, J., 2013. Post-failure recovery of mpi communication capability: Design and rationale. The International Journal of High Performance Computing Applications 27, 244–254. URL: <https://doi.org/10.1177/1094342013488238>, doi:10.1177/1094342013488238, arXiv:<https://doi.org/10.1177/1094342013488238>.
- [8] Bland, W., Bouteiller, A., Herault, T., Hursey, J., Bosilca, G., Dongarra, J.J., 2012. An evaluation of user-level failure mitigation support in mpi, in: European MPI Users' Group Meeting, Springer. pp. 193–203.
- [9] Bouteiller, A., Bosilca, G., Dongarra, J.J., 2015. Plan b: Interruption of ongoing mpi operations to support failure recovery, in: Proceedings of the 22nd European MPI Users' Group Meeting, ACM, New York, NY, USA. pp. 11:1–11:9. URL: <http://doi.acm.org/10.1145/2802658.2802668>, doi:10.1145/2802658.2802668.
- [10] Buntinas, D., Coti, C., Herault, T., Lemarini, P., Pilard, L., Rezmerita, A., Rodriguez, E., Cappello, F., 2008. Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant mpi protocols. Future Generation Computer Systems 24, 73 – 84. URL: <http://www.sciencedirect.com/science/article/pii/S0167739X07000258>, doi:<https://doi.org/10.1016/j.future.2007.02.002>.
- [11] Cao, J., Kerr, G., Arya, K., Cooperman, G., 2014. Transparent checkpoint-restart over infiniband, in: Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing, ACM, New York, NY, USA. pp. 13–24. URL: <http://doi.acm.org/10.1145/2600212.2600219>, doi:10.1145/2600212.2600219.
- [12] Corbet, J., 2012. Autonuma: the other approach to numa scheduling. LWN.net.
- [13] Derradji, S., Palfer-Sollier, T., Panziera, J.P., Poudes, A., Atos, F.W., 2015. The bxi interconnect architecture, in: 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects, pp. 18–25. doi:10.1109/HOTI.2015.15.
- [14] Dinan, J., Grant, R.E., Balaji, P., Goodell, D., Miller, D., Snir, M., Thakur, R., 2014. Enabling communication concurrency through flexible mpi endpoints. The International Journal of High Performance Computing Applications 28, 390–405.
- [15] EMELYANOV, P., 2011. Criu: Checkpoint/restore in userspace, july 2011. URL: <https://criu.org/>.
- [16] Fagg, G.E., Dongarra, J.J., 2000. Ft-mpi: Fault tolerant mpi, supporting dynamic applications in a dynamic world, in: Dongarra, J., Kacsuk, P., Podhorski, N. (Eds.), Recent Advances in Parallel Virtual Machine and Message Passing Interface, Springer Berlin Heidelberg, Berlin, Heidelberg. pp. 346–353.
- [17] Gamell, M., Katz, D.S., Kolla, H., Chen, J., Klasky, S., Parashar, M., 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, IEEE Press, Piscataway, NJ, USA. pp. 895–906. URL: <https://doi.org/10.1109/SC.2014.78>, doi:10.1109/SC.2014.78.
- [18] Garg, R., Sodha, K., Jin, Z., Cooperman, G., 2013. Checkpoint-restart for a network of virtual machines, in: 2013 IEEE International Conference on Cluster Computing (CLUSTER), pp. 1–8. doi:10.1109/CLUSTER.2013.6702626.
- [19] Hakkarinen, D., Chen, Z., 2013. Multilevel diskless checkpointing. IEEE Transactions on Computers 62, 772–783.
- [20] Hargrove, P.H., Duell, J.C., 2006. Berkeley lab checkpoint/restart (blcr) for linux clusters. Journal of Physics: Conference Series 46, 494. URL: <http://stacks.iop.org/1742-6596/46/i=1/a=067>.
- [21] Holmes, D., Mohror, K., Grant, R.E., Skjellum, A., Schulz, M., Bland, W., Squyres, J.M., 2016. Mpi sessions: Leveraging runtime infrastructure to increase scalability of applications at exascale, in: Proceedings of the 23rd European MPI Users' Group Meeting, ACM. pp. 121–129.
- [22] Hursey, J., Graham, R.L., Bronevetsky, G., Buntinas, D., Pritchard, H., Solt, D.G., 2011. Run-through stabilization: An mpi proposal for process fault tolerance, in: Cotronis, Y., Danalis, A., Nikolopoulos, D.S., Dongarra, J. (Eds.), Recent Advances in the Message Passing Interface, Springer Berlin Heidelberg, Berlin, Heidelberg. pp. 329–332.
- [23] Iancu, C., Hofmeyr, S., Blagojevic, F., Zheng, Y., 2010. Oversubscription on multicore processors, in: 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS), pp. 1–11.
- [24] Kale, L.V., Krishnan, S., 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, ACM, New York, NY, USA. pp. 91–108. URL: <http://doi.acm.org/10.1145/165854.165874>, doi:10.1145/165854.165874.
- [25] Kale, L.V., Zheng, G., 2009. Charm++ and ampi: Adaptive runtime strategies via migratable objects. Advanced Computational Infrastructures for Parallel and Distributed Applications, 265–282.
- [26] Kamal, H., Wagner, A., 2012. Added concurrency to improve mpi performance on multicore, in: 2012 41st International Conference on Parallel Processing, pp. 229–238. doi:10.1109/ICPP.2012.15.
- [27] Karlin, I., Keasler, J., Neely, J., 2013. Lulesh 2.0 updates and changes. Technical Report. Lawrence Livermore National Laboratory (LLNL), Livermore, CA.
- [28] Moody, A., Bronevetsky, G., Mohror, K., De Supinski, B.R., 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. pp. 1–11.
- [29] Naksinehaboon, N., Liu, Y., Leangsukun, C., Nassar, R., Paun, M., Scott, S.L., 2008. Reliability-aware approach: An incremental checkpoint/restart model in hpc environments, in: Cluster Computing and the Grid, 2008. CCGRID'08. 8th IEEE International Symposium on, IEEE.

- pp. 783–788.
- [30] Ni, X., Meneses, E., Jain, N., Kalé, L.V., 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. p. 7.
 - [31] Ni, X., Meneses, E., Kalé, L.V., 2012. Hiding checkpoint overhead in hpc applications with a semi-blocking algorithm, in: *Cluster Computing (CLUSTER)*, 2012 IEEE International Conference on, IEEE. pp. 364–372.
 - [32] Pérache, M., Carribault, P., Jourden, H., 2009. Mpc-mpi: An mpi implementation reducing the overall memory consumption, in: Ropo, M., Westerholm, J., Dongarra, J. (Eds.), *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, *Proceedings of the 16th European PVM/MPI Users' Group Meeting (EuroPVM/MPI 2009)*. Springer Berlin Heidelberg. volume 5759 of *Lecture Notes in Computer Science*, pp. 94–103. URL: http://dx.doi.org/10.1007/978-3-642-03770-2_16, doi:10.1007/978-3-642-03770-2_16.
 - [33] Pérache, M., Jourden, H., Namyst, R., 2008. Mpc: A unified parallel runtime for clusters of numa machines, in: *Proceedings of the 14th International Euro-Par Conference on Parallel Processing*, Springer-Verlag, Berlin, Heidelberg. pp. 78–88. URL: http://dx.doi.org/10.1007/978-3-540-85451-7_9, doi:10.1007/978-3-540-85451-7_9.
 - [34] Plank, J.S., Li, K., Puening, M.A., 1998. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems* 9, 972–986.
 - [35] Rieker, M., Ansel, J., Cooperman, G., 2006. Transparent user-level checkpointing for the native posix thread library for linux., in: *PDPTA*, pp. 492–498.
 - [36] Teranishi, K., Heroux, M.A., 2014. Toward local failure local recovery resilience model using mpi-ulfm, in: *Proceedings of the 21st European MPI Users' Group Meeting*, ACM, New York, NY, USA. pp. 51:51–51:56. URL: <http://doi.acm.org/10.1145/2642769.2642774>, doi:10.1145/2642769.2642774.
 - [37] Utrera, G., Corbalan, J., Labarta, J., 2014. Scheduling parallel jobs on multicore clusters using cpu oversubscription. *The Journal of Supercomputing* 68, 1113–1140. URL: <https://doi.org/10.1007/s11227-014-1142-9>, doi:10.1007/s11227-014-1142-9.
 - [38] Wende, F., Steinke, T., Reinefeld, A., 2015. The impact of process placement and oversubscription on application performance: A case study for exascale computing, in: Gray, A., Smith, L., Weiland, M. (Eds.), *Proceedings of the 3rd International Conference on Exascale Applications and Software*, EASC 2015, pp. 13 – 18.
 - [39] Zheng, G., Huang, C., Kalé, L.V., 2006. Performance evaluation of automatic checkpoint-based fault tolerance for ampi and charm++. *SIGOPS Oper. Syst. Rev.* 40, 90–99. URL: <http://doi.acm.org/10.1145/1131322.1131340>, doi:10.1145/1131322.1131340.
 - [40] Zheng, G., Shi, L., Kale, L.V., 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)*, pp. 93–103. doi:10.1109/CLUSTER.2004.1392606.