POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Executive Summary of the Thesis

A journey towards transparent fault tolerance in embarrassingly parallel MPI applications

Laurea Magistrale in Computer Science and Engineering - Ingegneria Informatica

Author: Luca Repetti

Advisor: Prof. Gianluca Palermo

Co-advisor: Roberto Rocco

Academic year: 2021-2022

1. Introduction

High-Performance Computing (HPC) has been leading innovation in various fields, ranging from weather forecasting to drug discovery. The computing capabilities of such systems are quickly reaching exascale performances, but the programming paradigm is falling behind, limiting the effective usage of such systems. Due to the increasing number of nodes in these clusters, and the related complexity, the likelihood of faults increases exponentially. At the same time, the standard intercommunication paradigm in HPC, the Message Passing Interface (MPI), does not handle fault tolerance: after a fault occurs in the application, the behavior is undefined - requiring a full restart.
The MPI Forum is working to improve the MPI Standard to support new functionalities, including fault tolerance, with the User-Level Fault Mitigation (ULFM) framework [3]. ULFM is a low-level solution to handle MPI faults, enabling applications using its API to continue without undefined behaviors after a fault. The library exposes low-level APIs to deal with faults. To simplify the adoption, various works have provided all-in-one solutions built on top of ULFM. An example of such libraries is Legio [5], which enables resiliency in embarrassingly parallel applications. It hides failures from the user and allows the application to continue the execution without the failed processes. Legio can empower the application to reach a result in case of error if the failed process was not critical to the completion of the application. In those cases, fault resiliency is not enough to achieve complete fault tolerance in MPI applications, and the recovery of the failed process is needed.
In this work, we explore the status of fault recovery through checkpoint/restart in MPI with the primary goal of maximum transparency and minimum overhead. We identify as a goal the minimization of latency and disk usage overhead, which is achieved by applying fault recovery only to the critical processes.
Different works have achieved similar outcomes with a stop-and-restart approach: the entire MPI job is killed and restarted from the most recent consistent checkpoint. The main drawback is the latency overhead in restarting the entire application after a failure. To solve this problem, other approaches do not stop the execution upon a failure but restart only the failed processes, thanks to the usage of ULFM.
The field still lacks a solution to give control

to application developers by allowing them to specify which processes are critical (and must be restarted). Such an approach would minimize latency overhead by paying the restart cost only when needed and minimize disk usage since checkpoints should be performed only on critical processes. At the same time, the framework should be transparent concerning MPI faults, simplifying its adoption in existing applications. In particular, we explore transparent and application checkpoint to achieve the solution described above:

Transparent checkpoint: we research the main strategies and available frameworks to reach fault recovery with user-level frameworks, which do not require any change to user code. After exploring the existing alternatives, we try to integrate them with Legio and ULFM, attempting to achieve a local transparent restart of the failed nodes.

Application checkpoint: we apply a transparent checkpoint/restart paradigm for MPI-related structures, shielding the library user from any details about the failure in the MPI domain. The user can leverage existing tools already used in the domain to checkpoint applications data reliably.

In this thesis, we explore the above alternatives, reporting problems and proposing solutions to the problem stated above.

This work is structured as follows: Section 2 gives a background on the main concepts needed to grasp the proposals; Section 3 explores the attempt to reach transparent checkpointing; while Section 4 proposes the application checkpointing solution. Finally, Section 5 examines the results of the experimental campaign, and section 6 wraps up the work with the conclusions.

2. Background

This chapter outlines the principal concept needed to grasp the journey defined in the rest of the work. Section 2.1 describes the Message Passing Interface (MPI), the communication method used in most HPC applications. Section 2.2 outlines the novelty introduced by User-Level Fault Mitigation (ULFM), a fault tolerance framework for MPI applications. Section 2.3 gives an overview of checkpoint/restart strategies to achieve fault tolerance. Finally, Section 2.4 recaps frameworks that achieve C/R through ULFM and checkpoint/restart.

2.1. Message Passing Interface

MPI is the de-facto standard interface to coordinate multiple processes in a parallel application in a distributed environment [1]. The MPI Forum is the committee that defines the specifications, which are then followed in different implementations, such as OpenMPI and MPICH.

To enable parallel applications to distribute work, communicators are used. They encapsulate the context of communication between a group of processes identified uniquely inside the communicator through their rank, starting from 0. The main APIs exposed by MPI to achieve distributed computing are the following:

Point-to-point operations: they focus on the simplest model of communications: exchanging messages between single processes. Given a communicator, processes are identified with their respective rank in the communicator and can send and receive messages.

Collective operations: they enable complex communication models by involving all processes within the used communicator and must be issued by all the processes in the given communicator.

MPI also exposes other primitives to achieve more complex goals, such as dynamically spawning new processes in the application after startup.

2.2. ULFM

User-Level Fault Mitigation (ULFM) is one of the recent efforts to introduce fault tolerance in MPI applications. Without any fault tolerance extension, the behavior of MPI after the occurrence of faults is undefined. With ULFM, MPI operations that involve failed processes must either succeed or raise an MPI error, which can be propagated to the user. Additionally, it guarantees that operations performed after a failure on non-failed processes will succeed, respecting MPI semantics.

The specification aims to define an interface to enable applications to resume communications after a failure. Still, it does not explore problems related to the recovery from the failure inside the application. Instead, it exposes the primitives needed to implement either shrinking solutions (where the application can continue

with fewer processes in case of failure) or non-shrinking ones (where failed processes continue the execution elsewhere).

2.3. Checkpoint/restart

Reaction to failures can be executed either with fault resiliency (continuing the execution with fewer processes) [5] or fault recovery (replacing the faulty processes with new ones) [2]. Although the first one minimizes the overhead, it has a few drawbacks: the expected result will likely be an approximate one, and the resiliency may not be guaranteed to finish execution in case a process needed for execution completion fails.
To solve these issues, it is possible to react to failures by restarting the entire application at a previous checkpoint (global restart) or only the node that experienced failure (local restart). Together with the different restart procedures, there also exists a taxonomy for the checkpointing strategy, which is differentiated based on the transparency of the solution. We can distinguish application-level and system-level checkpoints. The first requires a dedicated integration inside the user code to save and restore the most important application data. The second is transparent since it does not require user code changes but usually incurs additional overhead and can be system-specific.

2.4. Fault Recovery Techniques

Various efforts aim at fault recovery of an application after a failure by joining checkpoint/restart and ULFM [3]. They achieve non-shrinking recovery by restarting the execution of failed nodes from the latest consistent checkpoint. To implement this, they either initialize the application with a pool of spare nodes (incurring additional resource usage) or use dynamic process management MPI functions to spawn new processes.

3. Transparent Checkpoint

The first explored path proposes a technique to checkpoint and restore processes in an MPI application transparently. No change to the user code should be required, making our work available for already existing HPC applications.
Subsection 3.1 attempts to reach the desired solution with Distributed MultiThreaded Checkpointing (DMTCP) and MANA, while subsection 3.2 explores the usage of Checkpoint/Restore in Userspace (CRIU) to achieve transparent fault recovery.

3.1. DMTCP and MANA

Distributed MultiThreaded Checkpointing (DMTCP)[2] is a library supporting user-level checkpoint/restart without requiring kernel or OS changes.MANA enhances DMTCP with a special flow that separates MPI memory parts and application level, simplifying the restore process. The libraries follow a stop-and-restart approach by killing the application once a fault has been detected. The novelty to introduce concerning the current situation is resiliency: the application should continue in case of non-critical faults and respawn only the failed processes. To minimize latency and disk usage overhead, we try to implement local backward restart, which rollbacks the failed processes at the latest checkpoint and leaves the others to continue the execution.
Unfortunately, upon investigation, it was clear that DMTCP presents architectural challenges in how the MPI process management is structured to achieve this result. In particular, it does not support the respawn of a process attached to the same sockets as the ongoing application. On the other hand, MANA is not yet mature for our needs - we found issues in running it with OpenMPI.

3.2. CRIU

We make a second attempt to achieve the desired goal using Checkpoint-Restart in Userspace (CRIU) [4], which is a robust solution to the checkpoint with support in the Linux kernel. We envisioned the possible execution flow:

1. We checkpoint the current process, in case it is critical, before each MPI call. This ensures always a consistent state of checkpoints between the different processes and avoids deadlocks.
2. In case of failure of a non-critical process, we integrate Legio to ensure the application can continue in the presence of faults.
3. In case of failure of a non-critical process, Legio will handle the failure by restarting the failed processes and re-joining them with the already running MPI application,

by repairing the communicators removing the failed process, and including the new one.

The first two parts used features already part of Legio and CRIU, while the third was less trivial. First, we restarted the failed process using CRIU's already existing features. After, we leveraged the MPI routines `MPI_Comm_connect` and `MPI_Comm_accept` to join the survivor processes with the restarted ones, mimicking a client-server approach. Finally, we attempted to repair the existing communicators in the MPI application by removing the failed process and adding the new one.

Initially, the operation was unsuccessful because shared memory communication should be disabled to let CRIU checkpoints the processes. We fixed the issue by using TCP as a communication method, which is supported by CRIU.

The second issue encountered is the weak support of the dynamic process management in the MPI versions which integrate ULFM. For example, in OpenMPI 4.0.1, there is an issue that makes the dynamic process management operations described above unusable. On the other hand, OpenMPI version 5.0.0 raised an internal error that aborted the application after the `MPI_Comm_connect` and `MPI_Comm_accept` functions were called. The new OpenMPI runtime, introduced in the 5.0.0 version, sets the restarted process communicator as malfunctioning internally, making subsequent MPI operations fail.

3.3. Limitations

Due to the issues found in both C/R state-of-of-the-art frameworks, we decided to re-evaluate the initial transparency requirement. In particular, we chose to aim at restoring failed critical processes and the related MPI objects with local backward recovery. At the same time, we leave control to the user to checkpoint application-level data.

4. Application Checkpoint

The section explores the journey in application checkpointing. In particular, subsection 4.1 defines the motivation for the change from transparent to application checkpoint, while subsection 4.2 propose and details the project work to achieve the desired goal.

4.1. Motivation

Continuing the journey, we use application checkpointing to reach resiliency and recovery. Compared to system-level checkpointing, the main drawbacks are the lack of transparency and the need to implement user code support. On the other hand, different advantages arise:

- Only the critical processes should be checkpointed, minimizing the latency and disk usage. Moreover, the application awareness given from the application checkpoint can help minimize the overall overhead in terms of latency and disk usage.
- The application developer chooses each checkpoint's granularity, frequency, and criticality. It is then possible to easily integrate multi-level checkpointing where the likelihood of failure influences the storage system of the checkpoint.
- The only requirement is a storage layer that allows to dump and restore checkpoints: it is therefore highly portable and does not need to be adapted for particular environments.

4.2. Proposal

To reach this goal, we propose a framework, Legio++, that uses dynamic process management to restore failed processes. At the high level, the main parts of the application are three:

1. Initialization - the application code should use specific routines to ensure our framework can keep track of the MPI structures.
2. Failure detection - during process failure, the framework notices a fault and acts accordingly based on the criticality of the failed process. Failure resiliency without recovery is mainly based on Legio, but it needs a revamp in the context of possible restarted processes.
3. Restart phase - the routine that enables a failed process to be restarted and re-join the existing group of MPI processes is at the core of the thesis work.

During initialization, we wrap MPI communicators, ensuring that the application developer cannot see errors directly. We also expose `initialize_comm`, a function able to create a resilient communicator which must be called during startup. The operation allows us to keep track of communicators, which won't change for

the entire lifetime of the application.
Once a failure is detected, the first process noticing it communicates it to the rest of the alive processes. Other processes will be able to receive and check for failure notification messages without interrupting their work, thanks to a thread spawned during initialization. It periodically checks for incoming messages that report a failure. It is key that, once a failure is detected, all the processes are notified and participate actively in the reparation process. Moreover, it achieves consistency by avoiding that disjoint group of processes trying to recover from a critical process' failure by spawning it twice. The last part of the flow is the restoration, where our work manages to re-create failed processes through the `MPI_Comm_spawn_multiple` function, which spawns several processes equal to the number of failures. After, it repairs the internal `MPI_COMM_WORLD`, not exposed to the user, restoring order in the ranks concerning the newly introduced process. The survivor processes will pass the restarted ones all the information needed to reconstruct the state of the MPI structures, such as the already failed processes and the rank of the processes to respawn. Finally, it uses the new `MPI_COMM_WORLD` to repair the rest of the registered communicators. Contrary to Legio, this approach pays the overhead for restoration only once, ensuring that all communicators are repaired after a failure. This guarantees that, with multiple communicators, failures will be detected and dealt with only once.

5. Experimental Evaluation

We tested and confirmed that the proposed approach is functionally correct, checking that the exposed feature works as expected. To evaluate whether it brings novelty and its usability in real-world conditions, we perform an experimental campaign by analyzing the latency overhead for the proposed Legio++.
To evaluate the performance of the proposed work, we conduct two experiments: we focus on the overhead of the restart operation, and then we analyze a complete application that is embarrassingly parallel (a Montecarlo simulation to compute the value of π). We conducted these experiments on the Antarex node at Politecnico di Milano, featuring 2 x Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz processors and 128 GB of RAM.
To simulate the restart and its overhead, the `SIGINT` signal is injected in one of the processes, making it impossible to continue and abort immediately. Therefore, the `MPI_Barrier` right after will not succeed immediately - making it necessary for the framework to act with either resiliency or recovery.

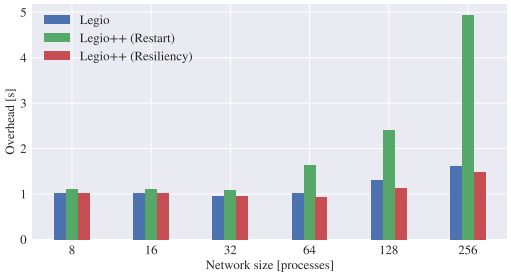

Figure 1: Failure tolerance overhead with different frameworks.

.

In the figure 1, we see the comparison between three different fault tolerance libraries:

- Legio [5], which implements fault resiliency without any recovery of failed processes.
- Legio++ (Restart), acting following the recovery flow. The rank where the `SIGINT` is injected is considered critical. Legio++ performs the recovery procedure respawning a new process and performing the previously missing `MPI_Barrier` operation.
- Legio++ (Resiliency), acting following the resiliency flow. The rank where the `SIGINT` is injected is considered a non-critical rank. Therefore, no recovery operation is needed.

It's clear from the picture that only recovery creates a sizable overhead between 64 and 256 nodes, which can be explained by fixing dynamic process management operations.
To further analyze the motivations behind the higher overhead of Legio++ with a restart approach, we perform a granular benchmark to understand the impact of the various parts of the operation. The following times are examined: failure propagation (communicate the failure to other processes), failure acknowledgment, shrink of `MPI_COMM_WORLD` to remove failed processes, respawn of a new process, construction of the new `MPI_COMM_WORLD`, and the reparation of the communicators during restart.

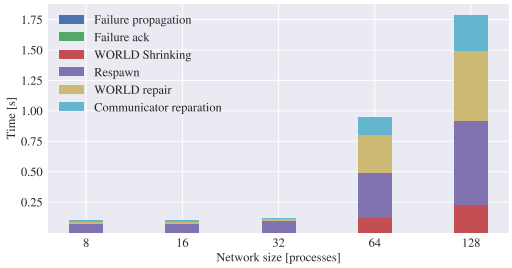

Figure 2: Overhead of each operation in the restart procedure, varying by the number of processes involved.

.

Analyzing figure 2, it is clear that the contribution of dynamic process management operations grows with the number of processes involved. Similarly, also the reparations of each communicator have increasing costs.
Finally, to conclude the experimental campaign, we test Legio++ with a Montecarlo simulation computing the value of π. In such a case, where a `MPI_Reduce` operation is performed to group results from different ranks, failure recovery is needed for the critical process which collects the results.
The results can be seen in Figure 3, confirming the minimal overhead of restart concerning resiliency considering the major impact of a critical node.

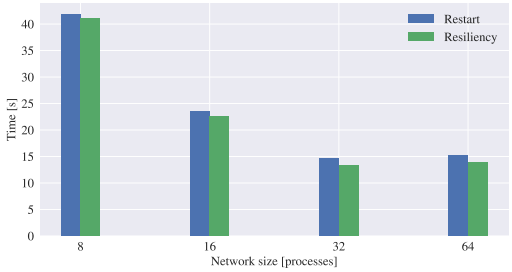

Figure 3: Average execution time of a Montecarlo simulation by varying the number of processes involved and the failure tolerance method.

The testing campaign proved that the proposed work contained latency overhead for fault recovery and resiliency, and recognized that the overhead for reparation grows with the number of processes. Given these results, the developers are in the best position to evaluate whether a process should be marked critical, considering fault recovery's more significant performance overhead than fault resiliency.

6. Conclusions

This thesis presents a journey through fault tolerance in MPI applications, starting with the ambitious goal of full transparent checkpointing and finishing with automatic failure recovery in all process ranks when a failure is detected. After evaluating the difficulties in reaching the initial goal of transparency, we propose Legio++, a prototype that allows application developers to tolerate failures without significant changes to the code for what regards MPI usage; we shield the behavior of MPI under failure by making the application developer unaware of the underlying failures of processes. The experimental evaluations demonstrate that, although the overhead of fault recovery is greater concerning fault resiliency, the trade-off is acceptable regarding critical processes.

References

[1] Mpi: A message-passing interface standard, version 4.0, 2021.

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

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

[4] Adrian Reber. Criu: Checkpoint/restore in userspace, 2012.

[5] Roberto Rocco, Davide Gadioli, and Gianluca Palermo. Legio: fault resiliency for embarrassingly parallel mpi applications. *The Journal of Supercomputing*, 78(2):2175–2195, 2022.