UNIVERSIDADE FEDERAL DO ABC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Pedro Henrique Di Francia Rosso

OCFTL:

an MPI implementation-independent fault tolerance library for task-based applications

Santo André, SP
2021

Pedro Henrique Di Francia Rosso

OCFTL:
an MPI implementation-independent fault tolerance library for task-based applications

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação (área de concentração: Sistemas de Computação) da Universidade Federal do ABC, como parte dos requisitos necessários para a obtenção do Título de Mestre em Ciência da Computação.

Orientador: Emilio de Camargo Francesquini

Santo André, SP
2021

Sistema de Bibliotecas da Universidade Federal do ABC
Elaborada pelo Sistema de Geração de Ficha Catalográfica da UFABC
com os dados fornecidos pelo(a) autor(a).

Di Francia Rosso, Pedro Henrique
OCFTL : an MPI implementation-independent fault tolerance library for task-based applications / Pedro Henrique Di Francia Rosso. — 2021.

110 fls. : il.

Orientador: Emilio de Camargo Francesquini

Dissertação (Mestrado) — Universidade Federal do ABC, Programa de Pós-Graduação em Ciência da Computação, Santo André, 2021.

1. Fault Tolerance. 2. MPI. 3. HPC. 4. OmpCluster. I. de Camargo Francesquini, Emilio. II. Programa de Pós-Graduação em Ciência da Computação, 2021. III. Título.

Este exemplar foi revisado e alterado em relação à versão original, de acordo com as observações levantadas pela banca examinadora no dia da defesa, sob responsabilidade única do(a) autor(a) e com a anuência do(a) (co)orientador(a).

Santo André, 10 de agosto de 2021 .

Pedro Henrique Rosso

Pedro Henrique Di Francia Rosso

Nome completo e Assinatura do(a) autor(a)

Emilio de C. Francesquini

Emilio de Camargo Francesquini

Nome completo e Assinatura do(a) (co)orientador(a)

MINISTÉRIO DA EDUCAÇÃO
Fundação Universidade Federal do ABC
Avenida dos Estados, 5001 – Bairro Santa Terezinha – Santo André – SP
CEP 09210-580 · Fone: (11) 4996-0017

FOLHA DE ASSINATURAS

Assinaturas dos membros da Banca Examinadora que avaliou e aprovou a Defesa de Dissertação de Mestrado do candidato, PEDRO HENRIQUE DI FRANCIA ROSSO realizada em 16 de Julho de 2021:

Prof.(a) DANIEL DE ANGELIS CORDEIRO
UNIVERSIDADE DE SÃO PAULO

Prof.(a) RAPHAEL YOKOINGAWA DE CAMARGO
UNIVERSIDADE FEDERAL DO ABC

Prof.(a) ALFREDO GOLDMAN VEL LEJBMAN
UNIVERSIDADE DE SÃO PAULO

Prof.(a) VLADIMIR EMILIANO MOREIRA ROCHA
UNIVERSIDADE FEDERAL DO ABC

Prof.(a) EMILIO DE CAMARGO FRANCESQUINI
UNIVERSIDADE FEDERAL DO ABC - Presidente

* Por ausência do membro titular, foi substituído pelo membro suplente descrito acima: nome completo, instituição e assinatura

CAPES

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001

Agradecimentos

À UFABC, pela oportunidade de realizar essa pós-graduação. Ao professor e orientador Emilio de Camargo Francesquini, por aceitar, incentivar e participar do desenvolvimento deste trabalho, sempre mostrando os passos para realizar o melhor trabalho possível, compartilhando todos seus valiosos conhecimentos no decorrer do mestrado.

Ao professor Guido Araujo, coordenador do projeto OmpCluster, pela oportunidade de ser um colaborador externo projeto e pela oportunidade de seguir como doutorando sob sua orientação na sequência do mestrado. E aos colegas de projeto que me ajudaram de forma direta e indireta na realização dessa dissertação.

À minha família, meus pais Pedro Rosso e Telma Regina França Rosso, pelo imensurável o apoio e incentivo, e todas discussões acerca dos mais variados assuntos relacionados ou não à pós-graduação, a minha irmã Talita Di Francia Rosso, que sempre foi minha companheira, sempre me incentivando e me apoiando ao máximo, ao meu irmão Pedro Augusto Di Francia Rosso, por ser meu companheiro sempre, por todas as discussões sobre os mais variados assuntos relacionados ou não a computação. À Beatriz Feltrin Canever, pela companhia, ajuda, incentivo e apoio durante todo o período do mestrado. Por fim, a todos meus amigos que sempre me ouviram, apoiaram e torceram por mim.

Ao Centro de Estudo de Petróleo (CEPETRO-Unicamp/Brazil) e a PETROBRAS S/A pelo suporte a este trabalho como parte do projeto BRCloud.

Resumo

Tolerância a falhas (TF) é uma preocupação comum em ambientes de Computação de Alta Desempenho (CAD). Seria de se esperar que, quando se trata de Message Passing Interface (MPI) (uma ferramenta para CAD de suma importância), TF seria um problema resolvido. Contudo, o cenário para TF e MPI é complexo. Embora TF seja efetivamente uma realidade nesses ambientes, geralmente é "feita à mão". As poucas exceções disponíveis vinculam os usuários MPI a implementações MPI específicas. Este trabalho propõe OCFTL, uma Biblioteca de TF que não é dependente de nenhuma implementação MPI específica para ser usada no OmpCluster, para aplicações paralelas baseadas em tarefas. O OCFTL é capaz de detectar falhas em menos de um segundo. Também fornece detecção de falha em caso de falso positivo, reparo do comunicador MPI e pode isolar os usuários do comportamento não especificado de operações MPI na presença de falhas. Os resultados dentro do OmpCluster mostram que o OCFTL não implica sobrecarga significativa e permite que os programas OmpCluster sobrevivam de falhas e sejam executados com êxito depois da ocorrência delas. Além disso, os resultados do Intel MPI Benchmarks mostram que o OCFTL pode superar as técnicas de ponta com a portabilidade de ser independente de implementação, permitindo a execução dos programas em distribuições MPI mais rápidas para diferentes casos.

Palavras-Chaves: Tolerância a falhas, Message Passing Interface (MPI), High-Performance Computing, OmpCluster.

Abstract

Fault tolerance (FT) is a common concern in HPC environments. One would expect that, when Message Passing Interface (MPI) is concerned (an HPC tool of paramount importance), FT would be a solved problem. It turns out that the scenario for FT and MPI is intricate. While FT is effectively a reality in these environments, it is usually done by hand. The few exceptions available tie MPI users to specific MPI implementations. This work proposes OCFTL, an Implementation Independent FT Library for MPI, to be used in OmpCluster, for task-based parallel applications. OCFTL is capable of detecting failures in less than a second. It also provides false-positive failure detection, MPI communicator repair, and it can isolate users from unspecified behavior of MPI operations in the presence of failures. Results within the OmpCluster show that OCFTL does not imply significant overhead and permits OmpCluster programs to survive from failures and execute successfully after. Moreover, results leveraging Intel MPI Benchmarks show that OCFTL can overcome state-of-the-art techniques with the portability of being implementation-independent, permitting the execution of the programs in faster MPI distributions for different cases.

Keywords: Fault Tolerance; Message Passing Interface (MPI), High-Performance Computing, OmpCluster.

List of Figures

List of Tables

Contents

1 Introduction

Many applications have been developed by scientific research in many areas, such as the advances of sequencing technologies in molecular biology (MIKAILOV et al., 2017) or geophysical data processing from spacecraft radars (NAEIMI et al., 2016). These applications often require high computational resources, which make the use of regular computing systems (PCs and small clusters) inviable, often requiring the use of High-Performance Computing Systems (HPC). These HPC systems are composed of multiple computing systems that allow the computation of applications to be divided across those nodes, speeding up the process.

A variety of tools are used to develop and parallelize applications in HPC systems, for example, the Message Passing Interface (MPI). Typically, the developers of those applications and users of HPC environments are not computer scientists, but a specialists in their domains. Domain-specific knowledge and the usage of specialized tools for HPC make the development of a complete, correct, and efficient system a daunting task.

With the distribution of the program across multiple computing nodes, one common concern is fault tolerance (FT), since a large number of computing nodes ultimately leads to an increased failure rate (ELLIOTT et al., 2012). Normally, the main cause of failures in HPC systems is from hardware or software, human factors, malicious attacks, server overloads, and network congestion (EGWUTUOHA et al., 2013). Table 1 shows the mean time between failures (MTBF)[1] for some HPC systems (DI et al., 2019). As seen, the MTBF can be less than one day, which justifies the concern about fault tolerance since HPC workloads have makespans that vary from days to even months.

Table 1 – Mean Time Between Failure for HPC systems

HPC System	**MTBF**
IBM Blue Gene/Q Mira (DI et al., 2019)	3.5 days
Titan Supercomputer (at OLCF) (ROJAS et al., 2019)	approx. 7 hours
Los Alamos National Lab System 5 (EGWUTUOHA et al., 2013)	approx. 55 hours
Los Alamos National Lab System 20 (EGWUTUOHA et al., 2013)	approx. 14 hours

Fault tolerance is an essential property of systems that need to run applications in the presence of failures. FT approaches are commonly divided into two main groups: Reactive and Proactive (HASAN; GORAYA, 2018). When it comes to MPI (the focus of this work), even today, the development of fault-tolerant applications is done manually, with

[1] The MTBF represents the average time between failures. It takes into account the MTTF (mean time to failure, expected time for the failure) and the MTTR (mean time to repair, the time that the service will stay interrupted)(PATTERSON; HENNESSY, 2016).

some exceptions like ULFM and MPICH. User Level Failure Migration (ULFM) (BLAND et al., 2013) is an extension to Open MPI, whereas MPICH has its own FT routines (BOSILCA et al., 2002). However, these approaches are not portable nor easy to use and are far from complete FT solutions.

To tackle those problems, OmpCluster[2] is a project that aims at easing the development applications in HPC environments. When using OmpCluster, most tools needed for parallel and distributed HPC application development are transparent to the user. To use it, the user (generally from non-computing areas) only needs to know one tool: OpenMP, which is a set of annotations used in the program code that automatically parallelizes the execution requiring less effort from the user. OmpCluster also provides container images that can be used to abstract any other requirements, such as installing necessary libraries. OmpCluster is a tool to be used in HPC systems. It uses MPI to automatically distribute the tasks created using OpenMP across the computing nodes. All this is done by leveraging LLVM's compiler infrastructure.

OmpCluster uses the `omptarget` library of OpenMP. This library focuses on the offloading of the computation to specific targets. These targets are devices that execute computation, like GPUs, or in OmpCluster's case, another process in a distributed system, in which, OmpCluster uses MPI to do communication between processes. This procedure characterizes a task-based workflow for OmpCluster, where the tasks generated by OpenMP are distributed via MPI to be executed by the devices.

Since MPI is at the core of the OmpCluster, and that MPI has significant limitations related to FT as mentioned above, this research aims at improving the resilience of the OmpCluster by providing fault tolerance through MPI. One of the main objectives of the OmpCluster is to be portable (having compatibility with different MPI implementations and HPC systems). Thus, this research aims to provide a portable and implementation-independent (MPI-wise) fault tolerance library, which will be referred to as OCFTL (OmpCluster Fault Tolerance Library). This library also needs to follow the restrictions implied by OmpCluster, being compatible with the most recent MPI standard, easy to update, and transparent to the final user of OmpCluster. Although this work focuses on OCFTL in OmpCluster, the library could be used stand-alone with other applications.

OCFTL provides mechanisms to detect and notify all active processing nodes about failures. It also provides extra functionalities such as MPI communicator repairs and recovering from failures with low overhead and latency. The results of this research show that OCFTL can detect and survive failures and recover from them using checkpointing. Even if OCFTL's overhead is higher when compared to other distribution-specific MPI FT libraries (*e.g.,* ULFM), it has the advantage of being portable and therefore can be used with faster MPI distributions. Additionally it supports some functionalities (such as

[2] Site project: <https://ompcluster.gitlab.io/>

FT broadcast with reduced number of messages and false positive failure detection) which are not present in all distribution-specific implementations.

Besides this document, this research resulted in two papers published in the ERAD-SP 2020 and ERAD-SP 2021. The first discussed FT reactive approaches integration with scheduling (ROSSO; FRANCESQUINI, 2020), while the second discussed the failure detection and propagation mechanisms (ROSSO; FRANCESQUINI, 2021). Both papers earned the award of Best Paper in the Graduate Category of each edition.

Following, it is described the objectives of this research and the Organization of this document.

1.1 Objectives

Main

- Improve the resilience of MPI in OmpCluster, providing a fault tolerance solution for MPI, which is capable of detecting failures, surviving them, and completing the program execution correctly even if in the presence of failures.

Specific Objectives

- Provide a transparent, easy to maintain and update, and implementation-independent fault tolerance library for MPI to be used by OmpCluster. It is meant that the library should not use functions or tools that are not well documented or discussed by the community. These would make the process of fixing a bug or updating deprecated functions hard.
- Employ different fault tolerance approaches to improve resilience.
- Provide multi-platform compatibility, enabling fault tolerance in HPC clusters and cloud.

1.2 Organization

This work is divided into seven chapters. The first one is this introduction, and the remaining chapters are organized as follows:

- Chapter 2 presents the background and motivation. This Chapter initially explores some concepts of fault tolerance necessary to the understanding of the proposed work. It also explores two of the fault tolerance challenges that this research will face as a motivation, and finally, it presents the related works.

- Chapter 3 presents an introduction to OmpCluster. In this Chapter, OmpCluster is described in detail. Since this research is inserted in the OmpCluster, the Chapter brings all necessary concepts to understand how the proposed fault tolerance library will work.

- To organize the description of the proposal, this document divides it into two parts. Chapter 4 presents the description of the fault tolerance tools implementation, describing the design and rationale for each one. Moreover, Chapter 5 describes the implementation of the integration and use of the FT tools by OmpCluster.

- Chapter 6 presents an experimental evaluation of the proposed FT. This Chapter presents some MPI implementation-dependent behaviors, the benchmark of OCFTL, and a comparison with state-of-the-art. It also presents the current limitations of OCFTL.

- Chapter 7 outlines the conclusion of this work, pointing out the key contributions and limitations of this research, also presenting future works for this research.

2 Background and Motivation

To motivate this research, a review on the topics related to failures in HPC is necessary. FT is a common concern in such systems since clusters with high node-count lead to increased failures (ELLIOTT et al., 2012). To avoid misunderstandings, from the outset will define some terminology. *Failures* relate to actual hardware or software failures, and *errors* represent the manifestation of those failures (KOREN; KRISHNA, 2020). For example, a *failure* in the hardware causes an operation to be wrong. An *error* will possibly be generated if that operation is called and the computed result is wrong.

Another classification for the failures is based on their occurrence. Failures can be *permanent*, *transient*, or *intermittent* (KOREN; KRISHNA, 2020). Permanent failures lead the components to be unusable, needing restoration (*e.g.,* migrating to another component or component changing). Transient failures occur for some time, causing malfunction of a component, and after a period, it is restored (EGWUTUOHA et al., 2013). Intermittent failures, differently from transient failures, they occur and after some time the component works again, and after a period of time the failure occurs again (KOREN; KRISHNA, 2020). Finally, a final classification of failures is typical of hardware failures, which can be benign when the component stops working or *byzantine* when the failure leads the component to state a wrong result (KOREN; KRISHNA, 2020; HASAN; GORAYA, 2018). This work focuses on *permanent benign failures* (although some attention is paid to transient failures).

One crucial part of fault tolerance is failure measuring, which explains failure occurrence in different HPC systems. Traditional metrics are *reliability* and *availability*. The first denotes the probability that the system will be up given a time interval, usually relates to the MTBF, MTTF, and MTTR, which were discussed in Section n 1 (KOREN; KRISHNA, 2020). Availability is the average of time, given an interval, that the system is up, and can be calculated in terms of reliability metrics, $A = \frac{MTTF}{MTBF}$ (KOREN; KRISHNA, 2020).

Section 1 presented some HPC system measures, showing how low is the MTBF of some systems nowadays. Many works study logs of failures in HPC systems with different objectives. Some to give research directions to future studies on FT (OLINER; STEARLEY, 2007). Others to match failure occurrence into specific probability distributions such as log-normal and Weibull (ROJAS et al., 2019; SCHROEDER; GIBSON, 2009; YUAN et al., 2012), which is essential, for example, to generate failure injectors with the purpose of testing. Other works try to characterize the systems through the classification of failure causes and characteristics (DI et al., 2019). Finally, other works make analyses to create models to predict failures (LIANG et al., 2006).

Now that the basic concepts of failure in HPC are given and the importance of fault tolerance in such systems was shown as motivation to this research, the rest of this chapter brings the remaining concepts necessary to understand the design and use of FT in OmpCluster. Section 2.1 discusses the main tools used in this research, also describing some of the challenges related to FT in these tools. Section 2.2 discusses some concepts of fault tolerance approaches. Finally, Section 2.3 revises the related work.

2.1 Tools employed in OmpCluster

The frameworks, specifications, and pieces of software used by OmpCluster are described in this section. In this text, we employ the word *tool* to describe each one of these components. Section 2.1.1 reviews some of the concepts of OpenMPI, while Section 2.1.2 the concepts of MPI.

2.1.1 OpenMP

OpenMP is an effort to ease the implementation of shared-memory parallel applications by providing a common and straightforward interface. It is important to note that OpenMP is not a new programming language. Instead, it works by annotating existing code written in C, C++, and Fortran (CHAPMAN; JOST; PAS, 2008). This characteristic also makes OpenMP a good tool for the parallelization of existing sequential code with reduced effort.

It provides many features that range from *parallel* control, which controls the flow of the program (*e.g.,* omp parallel), *work sharing*, to distribute the work among the threads (*e.g.,* omp section, omp single, omp for), *data control*, to control the scope and dependency of data (*e.g.,* shared, private variables), synchronization, controlling thread execution (*e.g.,* barriers and atomic operations) and *runtime* features, to control the environment (*e.g.,* number of threads) (CHAPMAN; JOST; PAS, 2008).

OpenMP strives to make its use simple, which is achieved by using code annotations using compiler pragma directives (`#pragma`, in C/C++), followed by some OpenMP clauses. For example, to construct a parallel region, one would use `#pragma omp parallel`, making the following code block run in parallel in as many threads as available. Additionally, suppose the user includes `#pragma omp for` before a loop definition (`for` in C or C++) inside a parallel region. In that case, the loop will be broken into chunks, and each chunk will transparently be distributed for parallel processing across the available OpenMP threads. Several other OpenMP clauses define additional execution attributes for these features.

Another feature of OpenMP is the `omptarget library`. This library defines the

concept of `targets`, which are devices in which the computation can be offloaded[1]. Devices can be, for example, a GPU card, or in the OmpCluster's case, a process running in another machine in a cluster.

Algorithm 1 shows a program parallelized with OpenMP with some `#pragma` definitions. This code first creates a region of parallel code with the `#pragma omp parallel` construct, after, `#pragma omp single` defines that only one of the OpenMP threads will execute the code inside the parallel region, in this case, the outer `for` loop. In the sequence, each time the code inspects the `#pragma omp target` directive, a task to be executed on a target is defined, the `depend(in:..., inout:...)` defines that the associated target task will depend on any previous task that modified `BlockA`, `BlockB` and `BlockC` and will be a dependency of any task that will modify `BlockC` after. The `map(to:..., tofrom:...)` construct means that the variables `BlockA`, `BlockB` and `BlockC` will be transferred to the target and `BlockC` will be retrieved from the target. Finally, the `nowait` clause means that no synchronism of OpenMP threads is needed after the target task statement.

Algorithm 1 – OpenMP parallelization example.

```
1  #define BS 512
2  #define N 2048
3  int BlockMatMul(BlockMatrix &A, BlockMatrix &B, BlockMatrix &C) {
4    #pragma omp parallel
5    #pragma omp single
6    for (int i = 0; i < N / BS; ++i)
7      for (int j = 0; j < N / BS; ++j) {
8        float *BlockC = C.GetBlock(i, j);
9        for (int k = 0; k < N / BS; ++k) {
10         float *BlockA = A.GetBlock(i, k);
11         float *BlockB = B.GetBlock(k, j);
12         #pragma omp target depend(in: BlockA[0], BlockB[0]) \
13                            depend(inout: BlockC[0]) \
14                            map(to: BlockA[:BS*BS], BlockB[:BS*BS]) \
15                            map(tofrom: BlockC[:BS*BS]) nowait
16         #pragma omp parallel for
17         for(int ii = 0; ii < BS; ii++)
18           for(int jj = 0; jj < BS; jj++)
19             for(int kk = 0; kk < BS; ++kk)
20               BlockC[ii + jj * BS] += BlockA[ii + kk * BS] * BlockB[kk + jj * BS];
21       }
22     }
23   return 0;
24 }
```

OpenMP is available in compiler infrastructures, like GCC and LLVM/Clang compiler (used in the OmpCluster project). The last published standard of OpenMP is 5.1, and it is available at the OpenMP site (<https://www.openmp.org>).

[1] We call by offloading the process of transferring part of the program execution to another device.

2.1.2 Message-Passing Interface - MPI

The Message-Passing Interface (MPI) is an effort to standardize the message-passing protocol libraries, mainly focusing on the parallel programming model. MPI itself is not an implementation or a language; instead, a standard. This way, all MPI procedures are made via functions, methods, and binds to another language, such as C or Fortran (FORUM, 2015). MPI tries to establish a standard to be practical, portable, and efficient. MPI provides many features, including point-to-point (*e.g.,* send and receive between two processes) and collective operations (*e.g.,* broadcasts, reduce and gather operations), datatypes, communication contexts, etc. (FORUM, 2015).

There are several MPI implementations, for example, MPICH (BOSILCA et al., 2002), Open MPI (GABRIEL et al., 2004), MVAPICH (PANDA et al., 2021), IntelMPI[2]. Each implementation has ample freedom to implement the standard and make decisions about the behavior regarding non-specified behavior. Which allows MPI implementations to make the most of the hardware architecture they are aiming for. In this work, some MPI implementations were evaluated to check if they provide all the necessary features. We found that Open MPI does not employ fault tolerance. Instead, their creators suggest the use of an extension: ULFM (which was also evaluated). MPICH employs limited fault tolerance since its failure detection mechanism is only available when using TCP/IP communications. Similarly, MVAPICH, which relies on MPICH's implementation, has the same behavior. Although there are many MPI implementations, this research will focus in the MPICH and OpenMPI distributions since they are very commonly used and are the base for other MPI implementations, like MVAPICH and IntelMPI.

Section 2.1.2.1 brings the discussion about the challenges this research may face in its development.

2.1.2.1 Fault Tolerance Challenges

Building a portable and implementation-independent[3] library to provide FT to MPI requires facing many challenges. For instance, depending on the MPI implementation, the behavior in the presence of failures may change and the available means to detect failures and recover from them may differ. These topics are elaborated below, and additional tests and discussions are made in Section 6.

Failure impact: To provide FT to MPI applications, one needs to determine the default behavior of each implementation when failure happens. The default behavior of the evaluated MPI implementations is to abort the execution of the whole MPI program (all processes) when any of the processes finishes prematurely. However, some implementations

[2] Available at:<https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/mpi-library.html#gs.3lcs0t>

[3] In this text *implementation* refers to the MPI implementations and not to OCFTL itself.

provide specific runtime flags that disable this default behavior. These flags[4] are essential when building a user-level library.

Error Handling: MPI implementations also offer the option to associate error handling objects with MPI communicators. By default, there are two options, `MPI_ERRORS_ARE_FATAL` and `MPI_ERRORS_RETURN`. The former aborts the program whenever an error occurs, while the latter returns the error (FORUM, 2015). The specification also allows users to create their callback functions and bind them to error handling objects. Thus, whenever an error occurs, a callback is made, allowing users to capture and handle it. Indeed, OCFTL employs error handling objects and recovery flags to provide an implementation-independent fault-tolerance environment for MPI applications.

Failure Detection: This consists of how the system will detect failures, which is usually accomplished by employing some node monitoring. For example, MPICH looks at TCP/IP sockets to detected failures (BOSILCA et al., 2002) while ULFM employs a heartbeat ring (BLAND et al., 2013) or OS signal monitoring (ZHONG et al., 2019).

Robust failure detection mechanisms are also needed to face recurrent problems. One of these problems is falsely detected failures (or simply false-positive failures). False positives might happen when a process temporarily stops sending alive notifications (*e.g.,* due to a temporary overload) and, after some time, restarts behaving as expected. Other common problems include system noise, failure-detection overheads, detection time (time taken the detect a failure after it occurs), failure propagation (how failures notifications will be propagated to other processes, so every process knows about every failure), and locality (when a process monitors another process that is also located on the same node).

Impact Mitigation: This refers to the ability to reduce the impact of the failures in the system. There are two main categories of approaches to FT, *reactive*, and *proactive*. In the first case, failures are handled after they occur, in the second case, the system will try to avoid possible failures (HASAN; GORAYA, 2018).

There are many approaches to each category. For proactive fault tolerance, for example, there are *Self-Healing*, when the system can periodically supervise and recover from erroneous states; *Pre-emptive migration*, when the computation is moved from suspicious nodes; and, *System Rejuvenation*, when the system restarts periodically (from backups, which can be partial or totally) to make execution fresh again (HASAN; GORAYA, 2018). Some examples of reactive fault tolerance are *Checkpoint restart*, which consists of periodically saving execution states, and in case of failures, the execution is restarted from the saved point; *Migration*, which consists in moving the computation to another node (after the failure occurrence, in contrast to the preemptive migration); and *Replication*, which consists in running multiple instances of the same tasks, in case of one fails, the

[4] Examples of flags are "–enable-recovery" for Open MPI and "–disable-auto-cleanup" for MPICH and MVAPICH.

other can continue the execution (HASAN; GORAYA, 2018).

2.2 Fault Tolerance

This section discusses the concepts of FT used in this research. For failure detection, OCFTL uses the heartbeat algorithm. Section 2.2.1 explains how it works and how the internal FT broadcast is used. For impact mitigation, this research aims at reactive approaches. Focusing on *checkpoint restart*, discussed in Section 2.2.2. This approach composes the current failure mitigation system for OCFTL, which is planned to be extended with the use of `Replication`, which is briefly discussed in Section 2.2.3.

2.2.1 Heartbeat and Internal Broadcast

Heartbeat is one of the ways to detect failures. When using the heartbeat, processes exchange `beat` messages with other processes. Normally the processes are organized in a ring topology, so every processes will be an `emitter` (sending `beat` messages to the next process in the ring) and will be also an `observer` (listening for `beat` messages from the previous process in the ring) (BOSILCA et al., 2018). This is present in ULFM (BOSILCA et al., 2018) and also in another work that uses a heartbeat to control failures on daemon processes[5] (ZHONG et al., 2019).

The heartbeat has two main properties: the `period`, which is the interval time between `beat` messages, and the `timeout`, which is the minimum amount of time a process will wait since the last `beat` was received before considering the `emitter` as failed, this is reseted when a beat is received. The heartbeat algorithm is composed of few tasks (BOSILCA et al., 2018):

Task 1 Occurs every `period`. The `emitter` sends a beat to the `observer`

Task 2 Occurs when a beat is received. The `timeout` is reseted.

Task 3 Occurs when the `timeout` is reached. The `emitter` is considered dead, a new `emitter` is set, a `NewObserver` notification is sent to the new `emitter`, and the information is propagated to the other processes through a FT broadcast algorithm.

Task 4 Occurs when a `NewObserver` notification is received. The `observer` is redefined to the new observer, and a beat is sent immediately.

Task 5 Occurs when a broadcast is received. If is the first time that broadcast is received, replicates the broadcast.

[5] In this case, daemon processes are special processes that monitor some of the application processes. Application failures will be detected by the daemon process that is watching the failed process, while the heartbeat will detect failures in the daemon processes.

These properties and these events compose a complete heartbeat. Another essential topic to discuss is the FT broadcast. The broadcast is essential since the failure needs to be propagated across all processes. As a fault tolerance library, this broadcast needs to be fault-tolerant, first, because there will be failed processes in the set of processes (so, for MPI, the broadcast can not be used). Second, there is the possibility of some broadcast messages to be lost so that the broadcast will not be completed.

There are few options of FT broadcast algorithms. First, the hyper-cube broadcast algorithm (HBA), in which the starter process sends the broadcast for k processes forward and k processes backward in the ring with $k = \lfloor log_2(N) \rfloor$, and N the number of processes) (BOSILCA et al., 2018). Circulant graphs, like binomial graphs (BMG), are a good option to achieve a balance between propagation delay (time for the propagation to achieve all processes) and scalability (number of exchanged messages). In BMG, the broadcast is sent to processes following powers of two (*i.e.*, a process with rank k sends messages to processes with ranks $k+2^0, k+2^1, k+2^2, ...$), forward and backward in the ring. This approach creates redundant messages since a process can be selected more than once (one time forward and backward in the ring) (ZHONG et al., 2019). This research aims to improve the FT broadcast algorithm proposing a new way to do it. Further discussion is made in Section 4.2.

2.2.2 Checkpointing

Checkpointing (or Checkpoint/restart) is a powerful fault tolerance technique that saves a snapshot of the program to stable and reliable storages (KOREN; KRISHNA, 2020).

Traditionally, checkpoints are divided into certain levels according to their management. At the *kernel-level* (or Operating System), checkpoints are taken at the operating system level, where the application does not take part in the checkpointing process (KOREN; KRISHNA, 2020; PLANK, 1997). Some examples are BLCR and DMTCP (HARGROVE; DUELL, 2006; ANSEL; ARYA; COOPERMAN, 2009). At the *user-level* (transparent to the user), checkpointing is done by the application itself, normally achieved by linking a dynamic library to the application (KOREN; KRISHNA, 2020; PLANK, 1997), for example, CPPC (RODRÍGUEZ et al., 2010). Finally, the *application-level* (not transparent to the user), is when the checkpoint explicitly needs application interaction (*i.e:* the application needs to provide functions to save/load the checkpoints) (KOREN; KRISHNA, 2020; PLANK, 1997). SCR, Veloc and FTI (MOODY et al., 2010; NICOLAE et al., 2019; BAUTISTA-GOMEZ et al., 2011) are examples in this category. Each type has it own pros and cons. In one hand, as near the approach is to the *kernel-level*, more transparent it is to the application, so it is not needed for the application to interact with the checkpointing library, at the same time, it has less control of checkpointing and what is

being checkpointed. In the other hand, as near as the approach is to the *application-level*, the application has more control of what is being checkpointed and how checkpoints are taken, but, more interaction is needed, which means that the application developer needs to know how and what to do to provide checkpointing.

There is also a variety of classifications for checkpoint/restart techniques. For example, some libraries are called multi-level, like the SCR, Veloc, and FTI, which allows the checkpoint to be saved in different ways (*e.g.,* only local, at the same machine as the application process; or on distributed file-systems; or replicating it on a neighbor node) (NICOLAE et al., 2019). Further definitions of checkpointing are still possible, a checkpoint can be: *coordinated*, where checkpointing is made with all processes synchronized, ensuring global consistency at the cost of scalability; *uncoordinated*, when the synchronism is not needed, improving the scalability; and *hierarchical*, when processes are divided into groups, in which for each group scope, there is the coordinated approach, and for the inter-groups procedures, the uncoordinated approach. Finally, there is also *message logging*, which consists of saving the messages to reproduce non-deterministic events (DONGARRA; HERAULT; ROBERT, 2015).

Another essential characteristic of checkpointing relies on the saving process. Some checkpoint libraries checkpoint the whole program memory (*full*) while others only save the changed parts of the memory (*incremental*) (NICOLAE et al., 2019).

Moreover, the overhead (which is the extra time applications spend when creating a checkpoint) and the latency (which represents the time necessary to save a checkpoint) are also two crucial concepts that come with checkpointing (KOREN; KRISHNA, 2020). When employing a checkpoint, a checkpointing interval needs to be defined. Since checkpointing can incur significant overhead and latency, it is crucial to make checkpoints at certain times to avoid unnecessary overhead to the program (*e.g.,* small checkpointing interval in systems with higher MTBF means that many checkpoints will be taken before a failure occurs, ultimately leading to increased overheads). Young made a first-order approximation to calculate the optimal checkpoint interval (θ) that takes into account the cost (δ) of the checkpoint (time to write it) and the MTTI (M) (Mean time to interruption)($\theta = \sqrt{2\delta M}$) (YOUNG, 1974). More recent works tried to achieve even more accurate approximations, for example, a conditional calculation based on the comparison between the cost and MTTI (DALY, 2006), or by considering extra inputs to the calculation, such as a value that represents the fraction of lost work, such as done by Lazy checkpointing (TIWARI; GUPTA; VAZHKUDAI, 2014).

Section 4.3.1 discusses which type of checkpoint will be used in this project.

2.2.3 Replication

Replication (sometimes called redundancy) can be divided into two main categories, *space* and *time* replication. The first occurs when the component is replicated, be the hardware, the software, or information. The second occurs when the computation is repeated, and the result is compared with the result of previous executions (DUBROVA, 2013). This work aims at the *space* type of replication, more specifically, in the hardware and software replication.

There are two significant classifications to replication: it can be *active* when every instance of the computation starts together, and the execution will succeed if at least one of them completes successfully, or *passive*, when the other instances of the job do not start at the same time as the primary instance. In this case, the replica will start as soon as the main instance was declared failed. In this research, it is planned to employ space active replication. Further discussions are made in Section 4.3.2.

2.3 Related Work

Fault tolerance for MPI is a frequent topic of research interest with various works, some relying on existing MPI implementations while others provide new MPI implementations with FT. Also, some works employ new FT features over existing FT features (usually uses other MPI FT implementations as a base), while others try to employ initial FT (not complete solutions).

One of the commonly used MPI implementations, Open MPI, for example, refers users that need FT to another implementation, ULFM (BLAND et al., 2013), as its main FT approach[6]. On the other hand, another commonly used implementation, MPICH, includes limited support to FT in the vanilla implementation for TCP-based communications[7], in which the failure detection occurs when MPICH detects that a TCP socket between two processes broke (BUNTINAS et al., 2008). However, it is unclear what behavior can be expected on other configurations such as Infiniband or shared memory.

This Section divides the related work into two subsections. Section 2.3.1 visits the works that use an MPI base implementation or an MPI implementation with some FT employed. While Section 2.3.2 reviews the works that propose new MPI implementation with FT addition.

[6] <https://www.open-mpi.org/faq/?category=ft#ft-future>
[7] <https://wiki.mpich.org/mpich/index.php/Fault_Tolerance>

2.3.1 FT Proposals Based on Existing MPI Implementations

This subsection revises related works that propose additions to existing MPI implementations.

ULFM (BLAND et al., 2013) is proposed as an extension to Open MPI. Their authors propose functions that help the user to perform process failure control (`Revoke, Shrink, Agree, Failure_Ack, Failure_Get_Acked`). Those functions allow users to invalidate and shrink (alter the processes in a communicator, so it only contains alive processes) communicators, agree to something (*e.g.,* a flag), and acknowledge that a failure happened. The heartbeat algorithm uses a ring topology in which processes are sequentially distributed. Each process is an `observer` of the previous process on the ring, and an `emitter` to the following process. The failure propagation algorithm, on the other hand, uses a hypercube-like topology (BOSILCA et al., 2018). A more recent proposal for failure detection (ZHONG et al., 2019) also uses a heartbeat. However, it employs a Binomial Graph (BMG) topology for failure propagation. Similar to these two approaches, OCFTL also uses a heartbeat mechanism, although it does not employ any other detection system such as discussed by Zhong (*e.g.,* OS Signals). This lean OCTFL's mechanism makes our algorithm simpler and allows us to use a proven and scalable peer-to-peer topology for failure propagation based on Chord (STOICA et al., 2001). Chord is a distributed lookup protocol, which maps keys to nodes, and retrieve then, keeping the operations at $O(logN)$ (STOICA et al., 2001) (further information of chord-like broadcasts are found in Section 4.2). ULFM is one of the most significant efforts to fault tolerance for MPI nowadays, although it does not accomplish one of the objectives of the OmpCluster that is being portable.

MPI/FT is a fault tolerance approach that uses MPI/Pro 1.51 (older distribution of MPI standard 1.2). MPI/FT provides three types of middleware to MPI, managing redundancy and checkpointing across the middlewares. It employs different kinds of heartbeats to provides failure detection: internal, at the implementation level, where the heartbeat is done depending on the network backend employed; or external, at the application level, where a coordinator listens to heartbeats of the other ranks. It also employs recovering models (a process will replace the failed in two ways, with or without synchronization with the other processes, it also employs coordinated checkpointing) (BATCHU et al., 2004).

Reinit++ (GEORGAKOUDIS; GUO; LAGUNA, 2020) is also an extension to Open MPI. Relying on simplicity, Reinit++ employs only a few additional data structures and a single new function (`MPI_Reinit`) which allows recovery and restart after failures. Reinit++ authors claim improvements over ULFM and discuss a global recovery system (restarting the entire application). Reinit++'s failure detection mechanism is based on a root node that monitors some daemon processes that, in turn, monitor application MPI processes. In addition to the fact that OCFTL does not rely on a specific MPI implementation, OCFTL

focuses on local recovery and provides other functions like communicator shrinkage.

Leveraging an older LA-MPI (later migrated to Open MPI), there is a similar proposal for a fault tolerance library. It features a ring-based failure detector, automatic recovery, and DLCKPT (disk-less checkpointing) system. Different from the purpose of this research, it relies on the LA-MPI implementation, implemented as an extension to it (LU, 2005).

Some other works rely on the existing FT approaches. LFLR, for example, uses ULFM to allow recovery after failures, and in stark contrast to Reinit++, LFLR deals only with local recovery; in other words, it only deals with the restart of the failed component (TERANISHI; HEROUX, 2014). In this sense, LFLR is closer to OCFTL since its main proposal is to provide local recovery.

On top of LAM/MPI (also a predecessor of Open MPI) (LOUCA et al., 2000) MPI-FT employs failure detection by an O.S. script that monitors the MPI processes, and an MPI process to observe the script. It employs recovery in two ways: by spawning new processes; or by creating different processes at the start of the program, which will replace failed nodes.

Also, on top of ULFM, Fenix is a proposal that uses ULFM functions to support failure detection and process recovery (GAMELL et al., 2014). Additionally, Fenix supports different types of checkpoints (implicit, asynchronous, coordinated, and selective). Contrary to Fenix, OCFTL does not use additional MPI FT libraries for detection and proposes extra features like notification and gathering state functions and MPI wrappers.

A hybrid fault tolerance approach was proposed to take into account failures in the EasyGrid middleware (a framework that provides process management for MPI, easing the creation of grid-type applications) using checkpointing and message logs, as well as self-healing properties (SILVA; REBELLO, 2011). Built on top of LAM/MPI (predecessor of Open MPI) and top of the EasyGrid framework (since the EasyGrid provides process management), the authors use those tools to detect and recover from failures. In this sense, OCFTL tends to be more generic than EasyGrid, providing an FT library with more compatibility (any MPI-compliant implementation) as well as not relying on another tool.

2.3.2 FT Proposals Based on New MPI Implementations

Instead of proposing modifications or wrappers on existing MPI implementations, some works try to provide new MPI implementations with FT.

In the past, FT usually was employed providing new MPI implementations. CoCheck is a facility for checkpointing/restart for MPI. It is also transparent to the application. CoCheck was also integrated to the tuMPI (Technische Universitat Munchen MPI) (STELL-

NER, 1996), a new MPI FT implementation that was based on the first version of MPICH (BRIDGES et al., 1995).

FT-MPI was another try of providing an MPI fault-tolerant implementation, which implements the MPI-2 standard partially. It extended the MPI standard with some FT definitions, like communicator states. FT-MPI improves the communicator management in case of failures with new functions, like `SHRINK` or `ABORT`, although it is not entirely explicit, the failure detection is achieved with OS support (FAGG; DONGARRA, 2000).

Starfish MPI was an MPI implementation, MPI-2 compliant, which employs checkpointing and automatic recovery for MPI applications (AGBARIA; FRIEDMAN, 2003).

LA-MPI was an MPI implementation developed by the Los Alamos National Laboratory. It implemented some of the MPI standard 1.2 and 2 functions and dealt with fault tolerance. The focus of the work on network fault tolerance and is designed to improve the messaging interface while taking into account possible faults related to network communication, for example, message losses (AULWES et al., 2004).

MPICH-V is an MPI implementation that provides a multi-protocol approach, which proposes different approaches using message logging and checkpointing to deal with fault tolerance. MPICH detects when a node disconnects from the network, this way detecting a failure. MPICH is still very a used and maintained MPI implementation nowadays and is one of the implementations explored in this work (BOUTEILLER et al., 2006).

MPI Stages (SULTANA et al., 2018) is a brand-new implementation (also known as ExaMPI). MPI stages, however, offers only basic MPI functionality as well as checkpoints. The paper does not elaborate on how its failure detection and propagation system works. Further research by direct inspection of its source code would be needed. OCFTL, on the other hand, is based on the MPI specification and can (and should) therefore be used with any MPI-compliant implementation.

Compared to this work, creating a new implementation can be very complex. MPI's specification is large (and growing) so many works, such as MPI stages, are not fully specification-compliant. Moreover, many such endeavors are outdated or abandoned, such as tuMPI (CoCheck), or the creator's group migrated to newer projects. For instance, LA-MPI's group directed their efforts to Open MPI, and FT-MPI's group migrated their efforts to ULFM.

In sequence, Chapter 3 outlines the details of OmpCluster that are necessary to understand this research context, which is discussed in Chapter 4 and Chapter 5.

3 OmpCluster

This Chapter discusses OmpCluster's[1] architecture. Section 3.1 explains how OmpCluster uses OpenMP and MPI to provide a distributed way to execute OpenMP programs. Section 3.2 brings the initial fault tolerance discussion related to OmpCluster.

OmpCluster is a project that aims at easing scientific programming for HPC clusters by leveraging OpenMP in the LLVM compiler infrastructure. The basic idea of OmpCluster is to define a new device to offload computation in the `omptarget` library. The tasks generated by the OpenMP will be distributed to the OmpCluster's devices. This devices are another processes based on cluster or cloud computing. The computation will be distributed across various computing nodes, which is achieved by employing MPI in the OmpCluster's device. This procedure will be further explained in Section 3.1.

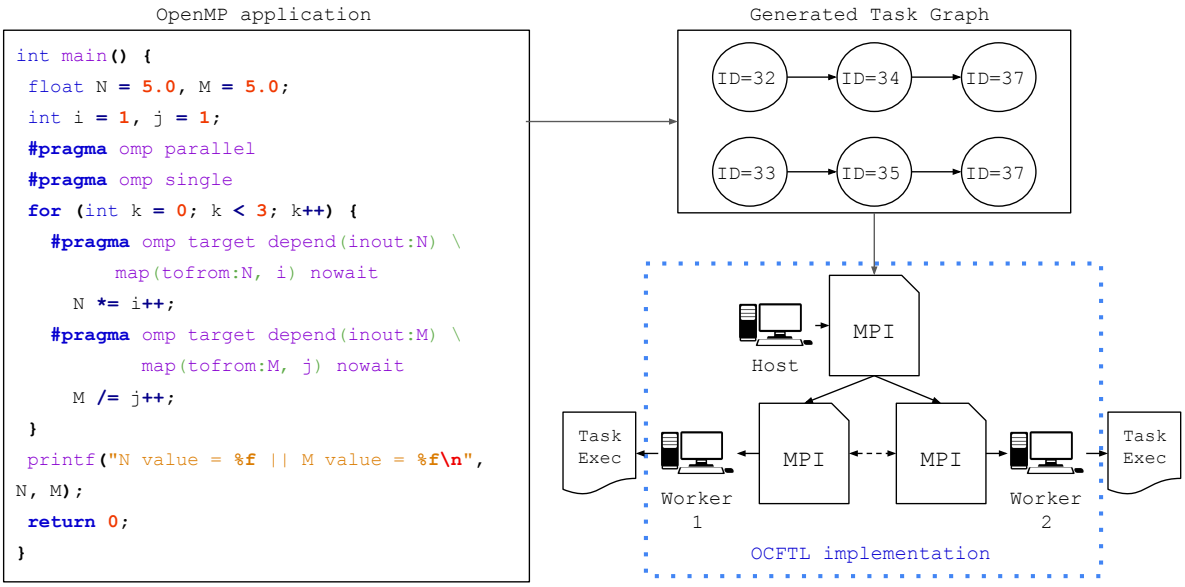
In OmpCluster, there are several research areas, there are studies of OpenMP target applications tracing, benchmarking, scheduling algorithms among other areas. The objective of this work is to provide resilience to OmpCluster. To understand the context of research inside the OmpCluster, this chapter focuses on OmpCluster's architecture. Later in Chapter 4, the proposed fault tolerance library will be described, and lastly, in Chapter 5, the integration between the proposed library and the OmpCluster will be discussed.

Figure 1 shows an example for using OmpCluster. The application execution flow is as follows: from an OpenMP program, the OpenMP runtime system creates the tasks previously described by the programmer using OpenMP directives. OpenMP will use these tasks to parallelize the program. This procedure creates a Directed Acyclic Graph (DAG) that models the dependencies between the tasks. After, it schedules and distributes the tasks across the computing nodes (CPU or CPU-GPU nodes) using MPI. As soon as the execution of these tasks completes, the final results are sent back to the main program. The task execution is controlled by an event system, in which, all the communication is done using MPI. This process only requires from the final user of OmpCluster (*i.e.,* scientific researchers from other areas) the knowledge of parallelization using OpenMP, which is much easier than other approaches.

Concerning this fault tolerance work, as seen in Figure 1, OCFTL is based on the part of the OmpCluster that involves MPI. OCFTL is based on MPI to detect and propagate failures. This representation is a simplified version of the OmpCluster workflow. So, based on the Figure, the library will provide resilience to the distributed part of the OmpCluster. Further discussions about the OCFTL's scope are made in Section 3.2.

[1] Site project: <https://ompcluster.gitlab.io/>

Figure 1 – Flow of the parallelization process.



OCFTL aims to guarantee that the program can complete with success in the presence of transient or permanent failures. In particular, OmpCluster aims to be compatible with multiple types of systems and MPI distributions, so the fault tolerance library should follow this requisite.

3.1 Architecture

This Section discusses OmpCluster's architecture and all the concepts that will be necessary to follow the remaining chapters.

OpenMP: Section 2.1.1 discussed the basics of OpenMP. Here, it will be described how OmpCluster uses OpenMP to offload tasks to distributed computing nodes. To do that, OmpCluster is employs the **`target`** part of OpenMP, this feature is used to offload the execution to other **`devices`**, which can be, for example, a GPU. In OmpCluster's case, a device is another process that can run in another node of a cluster or cloud.

OmpTarget: In the LLVM OpenMP infrastructure, the `omptarget` library is focused on the `targets` (also called `devices`), which are specific devices to which the computation can be offloaded. Normally, OpenMP will distribute its parallelization through the CPU threads in a regular parallelization. When using `targets`, OpenMP can distribute the parallelization through other devices, like GPUs (Graphics Processing Units). In this library, the execution process of OpenMP is divided into three parts. The first part is the `source`. In this part, the application source code is translated to OpenMP code, and this code will be executing the program through the OpenMP implementation (the OpenMP tasks will be created here). The second part is the `interface implementation`, in this

part, an interface is implemented between the `source` and `target implementation` part. This interface is not aware of specific devices that will be implemented, and it is responsible for translating the execution to the devices and features that are common to the devices (*e.g.,* scheduling). Finally, the third part is called `target implementation`, at this point, the implementation is directed to a specific device that is responsible for translating the offloading process to the target platform and executing the offloaded code.

Figure 2 – Representation of OmpCluster architecture inside OpenMP.

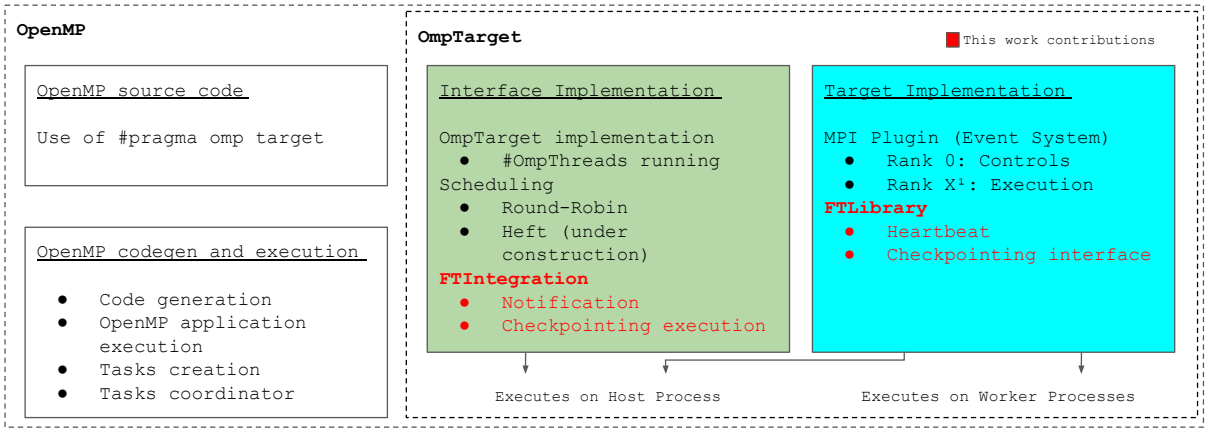


Figure 2 shows how that architecture works in the OmpCluster. Two domains can be observed, the `OpenMP`, which includes everything, and the `OmpTarget`, which includes the `Interface implementation` and `Target implementation`. In the first domain, there is the application source code which is oriented by `#pragma omp` directives (the `target` clause makes OpenMP offloads the computation) and the code generation and execution, which includes all OpenMP tasks. In the second domain, there are two subdivisions. The first is the `interface implementation` (in green), which will be executed by the number of threads defined by the `omp thread number` configuration[2] at the host process. Furthermore, the `Target Implementation` (in blue), which will be the only thing the worker processes will execute but is also present on the host.

As seen in Figure 2, a major part of OmpCluster implementation is on the `OmpTarget` (few changes have been made to the `source` part to provide `OmpTarget` some information, mainly about the OpenMP task graph). In the `interface implementation` part, will reside code related to scheduling and other non-MPI codes (*e.g.,* FT handling and notifications), while the related MPI implementations (*e.g.,* the FT functions implementation, event system codes) will be implemented in the `target implementation` part. OmpCluster proposes a new type of device, called `MPI Plugin`. The idea is to use

[2] It is important to differentiate the `omp thread number` from the MPI number of processes. In the OmpCluster, there will be `omp thread number` threads running on the head process. Moreover, there will be n processes running the program, which the MPI configuration defines n.

MPI to distribute the target tasks created by OpenMP to different computing nodes (*e.g.,* distributing execution to several cluster nodes). This `MPI Plugin` has two jobs; the first MPI process (rank 0, the `head process`) controls the core functions to distribute the tasks generated by OpenMP, while the remaining processes control the execution of the tasks in the computing nodes. In the same figure, highlighted in red, the inclusions of this work are shown. The FT tools are implemented at the `target implementation` component (described in Chapter 4), and the integration between FT tools and OmpCluster resides at the `Interface implementation` component (described in Chapter 5).

Interface implementation: The interface is the core of `OmpTarget`, it communicates with the OpenMP source and the target implementation. During the program execution, when a target task is dispatched to be executed (by the OpenMP runtime), a function will be called on the `OmpTarget` library. This task can be related to computation or data management. Each function will be redirected as a set of events that compose the OpenMP task. Since OmpCluster is based on distributed computing, if the process that will execute the task does not have the necessary data, the interface will forward it. Usually, if the task needs data forwarding, a set of allocation, data submission (forwarding), and retrieving will be done.

The interface is also responsible for the scheduling. During the program execution, the OpenMP runtime will manage the tasks (including the target tasks) based on their dependencies, dispatching the ready target tasks (with no more dependencies) to the OmpTarget library. However, inside the OmpTarget library, all target tasks are scheduled before the execution (currently, a Round-Robin schedule is used, but the HEFT scheduling is being implemented). When a full target task graph is complete (after OpenMP finishes processing a parallel region), the scheduler acquires a copy of the graph and pre-schedule all tasks. As OpenMP's runtime dispatches the target task to the OmpTarget library, the library will check the schedule and direct the execution of the task to the scheduled device.

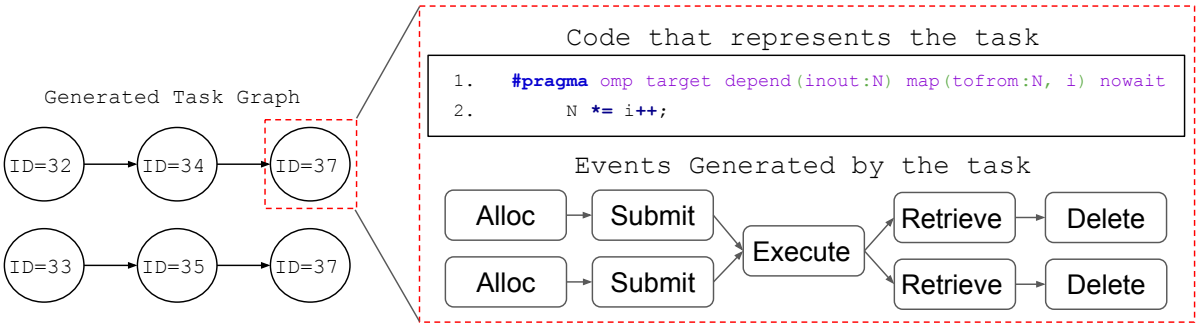
Another job done in the interface is data management, where all the data is mapped in the interface. This map is used to control the data forwarding, and to manage checkpoints.

Finally, most of the fault tolerance handling is done in the interface. The fault tolerance notifications will be received in the interface, the checkpoint and task re-execution are also managed in the interface. Further discussions about this integration between OmpTarget and fault tolerance are present in Chapter 5.

Target implementation: Target implementation: For the target implementation, MPI is used, so the OmpCluster device is called `MPI Plugin`. An event system implements the device. The event system defines events for every necessary operation to execute the target tasks. There are events related to data (alloc, delete, submit, retrieve, forward), related to

execution (execute), related to control (exit), and related to fault tolerance (checkpoint, save pointers, recovery). Figure 3 exemplifies how a target task from the application of Figure 1 is translated to events in the event system. The block of code represents the actual OpenMP code that will generate tasks with `ID` equal to 32, 34, and 37. Each one of those tasks will generate the events presented in the Figure. Since the code maps two variables on the target (`map(to:...)`), it will generate two `alloc` events (allocating one region of memory for each variable), followed by the two `submit` events (send the two variables values to the target). When the target has all the mapped data, the computation is done in the `execute` event. After the execution, since the code maps the two variables from the target (`map(from:...)`), two `retrieve` events will be generated, retrieving updated values. Finally, two `delete` events will be generated, freeing the previously allocated memory on the target.

Figure 3 – Example of events generated by an OmpTarget target task.



Every communication operation on the event system is made using MPI. In OmpCluster, the MPI process with rank 0 is defined as the head process, while the other ones are worker nodes. When a program is compiled with the OmpCluster, the resulting binary will contain both the `interface implementation` and `target implementation`. The host process will execute both parts while the worker processes will execute only the `target implementation` part.

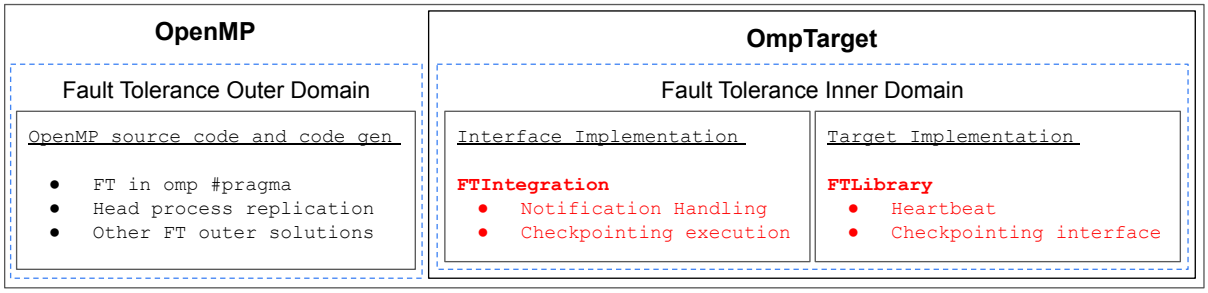
In the `target implementation` there is also OCFTL. In which, there will be one instance of the fault tolerance object for each MPI process. The complete description of the OCFTL is present in Chapter 4.

3.2 Fault Tolerance

Before diving into the fault tolerance library, it is important to define the FT domains. A domain is a context inside the architecture the defines the scope of fault tolerance. Concerning OpenMP and OmpCluster, there are two domains. The first domain is called `inner domain`, which represents the domain of this work. In the `inner domain`

the fault tolerance is provided at the same level as the `target implementation` and `interface implementation`, which means that the library is limited to that scope. At this level, the library should detect, survive and re-execute target tasks (this means the application can survive from failures in the worker processes but not on the head process since it is reponsible for the OpenMP program execution). The second domain is called `outer domain`, which represents the domain closer to the OpenMP code generation and task control. The `outer domain` is not the focus of this work, but it is future work. Within the `outer domain` the fault tolerance methods should be able to survive failures on the head processes, as well as re-executing implicit tasks (the ones that are not redirected to the `OmpTarget`). Figure 4 shows the location of two domains in the OmpCluster architecture. The red highlighted structures in the `inner domain` are the subjects of this research, while the remaining structures in the `outer domain` are objects of future studies, in which, for example, the use of OpenMP `#pragma` directives to control fault tolerance (specifying which fault tolerance approach to use, or defining a fault tolerance property), and outer FT solutions like the replication of the host process could be employed.

Figure 4 – Division of Fault Tolerance domains in OmpCluster.



4 OCFTL — OmpCluster Fault Tolerance Library

This chapter discusses the implementation of OCFTL. Section 4.1 describes the basics of OCFTL. Section 4.2 outlines the design and rationale behind the failure detection mechanisms of the library. Section 4.3 describes the failure mitigation mechanisms. Lastly, Section 4.4 brings the extra features of the fault tolerance library to the discussion.

4.1 Fault Tolerance Library

As one of the goals of the project described in Chapter 3, the runtime system can not rely on a specific MPI implementation. So, OCFTL has to provide FT mechanisms compatible with any specification-compliant MPI implementation. However, in this work, the discussions will be limited to Open MPI and MPICH since they are very commonly used and are the base for other MPI implementations, like MVAPICH and IntelMPI.

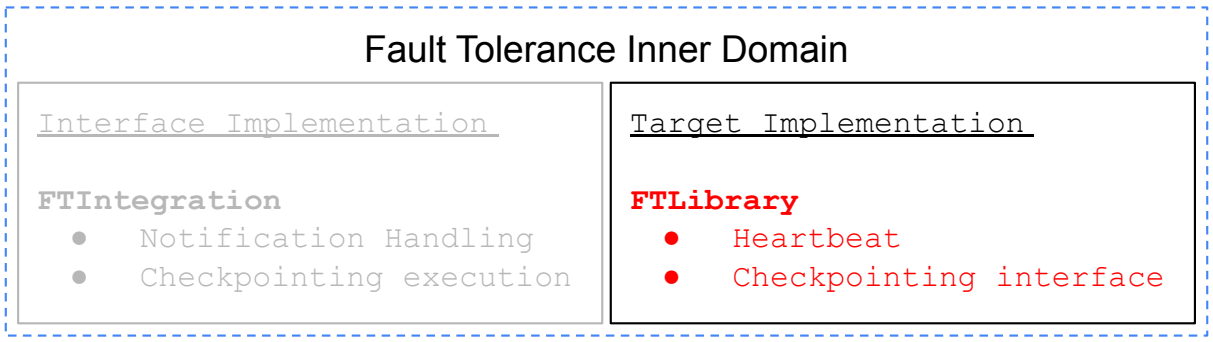
There are two main approaches to fault tolerance in MPI: either providing fault tolerance at the MPI implementation level, such as done by ULFM (BLAND et al., 2013) for example, or at the MPI user-level. In the first case, there is more control of the MPI runtime to deal with failures as one can access internal states of MPI. At the same time, the major drawback is the necessity of making the fault tolerance compatible with multiple different MPI distributions since every distribution could be implemented differently. The second one restricts the library to the use of MPI functions presented in the specification (FORUM, 2015). On the one hand, it makes compatibility easier since every MPI distribution should follow the standard. On the other hand, it has the drawback of necessarily employing whatever communication mechanism the MPI implementation uses as its backend.

In this work, the second approach was chosen. Using user-level MPI functions makes the library easier to maintain and update as new MPI standards are released, also, the documentation about them is extensive, as well as very discussed by the MPI community, and not less important, the compatibility is guaranteed across any MPI-compliant implementation. The only mandatory MPI configurations OCFTL needs are the `MPI_THREAD_MULTIPLE` support enabled (since the OCFTL runs over extra threads) and the runtime recovery flags (*e.g.,* `-enable-recovery` for MPICH and `-disable-auto-cleanup` for Open MPI), since by default the MPI implementations aborts the application if a process terminates prematurely. OCFTL is implemented as a C++ class (so it focuses

only on C++). In the OmpCluster architecture, the implementation of OCFTL tools is present at the `target implementation` part of OmpCluster, in the `inner domain` of FT, as seen highlighted in Figure 5. Since OCFTL is a `C++` class, one instance of the OCFTL object will be created for each MPI process (the head and worker processes of OmpCluster). Other threads will be launched for any extra tool OCFTL needs, like the heartbeat of the checkpoint library. These extra threads share the same MPI initialization done by the main MPI process, which means that have access to the same communicators and communicate as if were the main MPI process. There is also a C++ class in the `OmpTarget interface implementation` part of OmpCluster, which handles the notification of FT, controlling the failure handling for scheduling and task restarting, which is further discussed in Chapter 5.

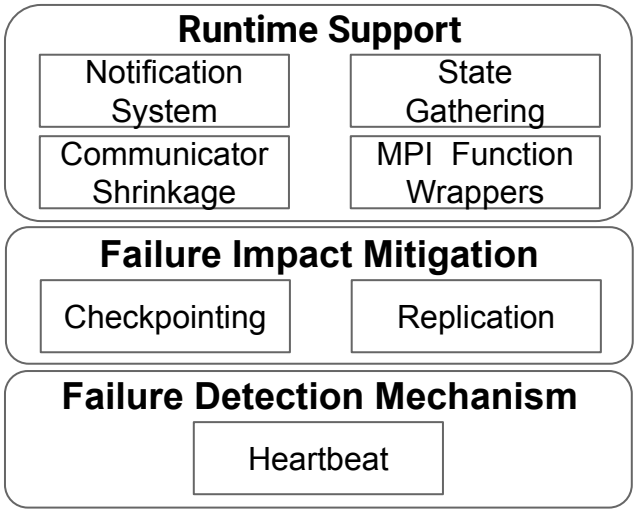
Although this work focuses on OCFTL in OmpCluster, the library can be used as a stand-alone library in other applications. To do so, one just need to compile OCFTL as a static library and link the library to the application. This is the procedure done for the experimental evaluation of OCFTL with MPI discussed in Sections 6.3.3, 6.3.4 and 6.3.2.

Figure 5 – Location of fault tolerance tools inside OmpTarget.



In OmpCluster, OCFTL aims to provide FT by aggregating different tools, which are divided into three groups, as shown in Figure 6. From the bottom to the top, the first group represents the failure identification tool (a heartbeat in OCFTL), the base for FT, which will be discussed in Section 4.2. The second group consists of the failure impact mitigation, which aims to mitigate the impact of failures (using checkpointing and replication) after they are detected (since restarting the entire program from the beginning after failures is not intended). This is discussed in Section 4.3. The top group is composed of runtime support features. These are functionalities that will help OCFTL execution and integration with other parts of OmpCluster, like the event system (using the notification and state gathering systems), as well as providing extra FT tools (communicator shrinkage and MPI wrappers), these features are discussed in Section 4.4. Appendix D describes in detail how the configuration of OCFTL can be achieved.

Figure 6 – Group division of fault tolerance tools in OmpCluster.



4.2 Failure Detection Mechanism

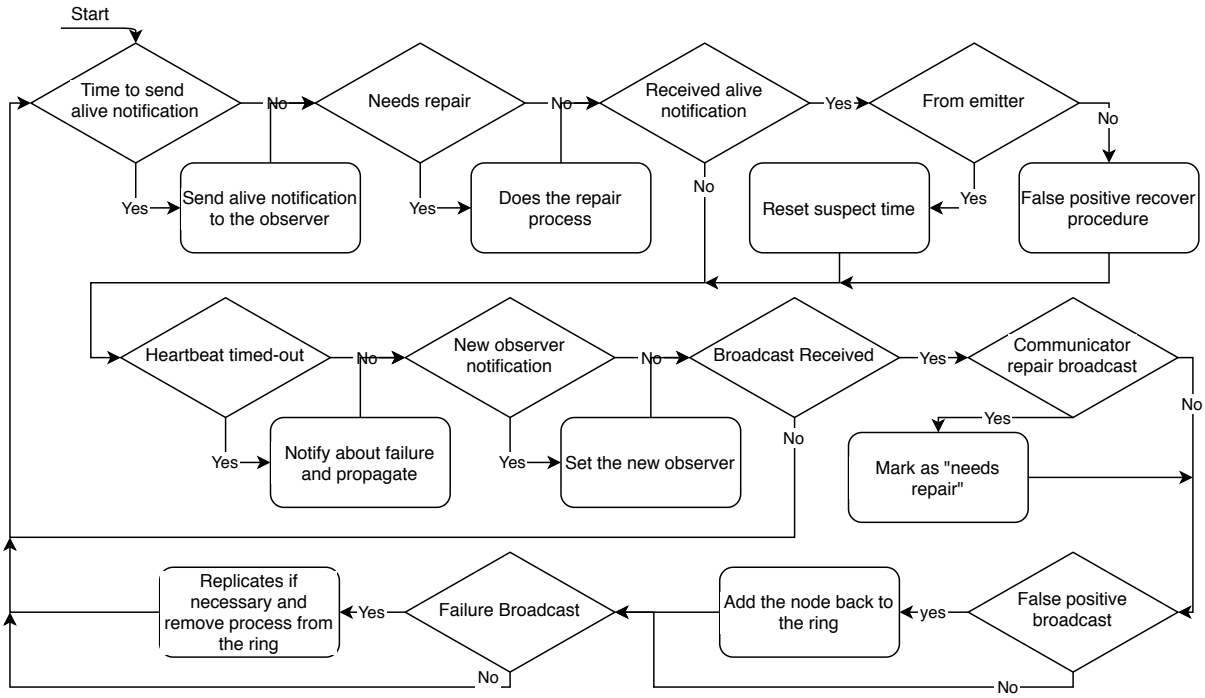
Failure detection is one of the essential features of fault tolerance. OCFTL employs a mechanism inspired by ULFM's (BOSILCA et al., 2018). The algorithm was modified to be compatible with OCFTL, working with user-level functions of MPI provided by all specification compliant implementations. The detector features a ring-based heartbeat, in which one process observes the previous neighbor and, when a failure occurs, propagates the failure to the remaining processes, following the schemes presented in Section 2.2.1. The features of the detection mechanism are described below.

The heartbeat: All the events are checked in a loop that occurs every pre-defined configurable time step in a specific thread. Figure 7 shows the actions made in the main loop: initially, the library checks if it is time to send a heartbeat; in the sequence, it checks if a repair communicator process was started; following, it checks if an alive message was received (if so, resetting the heartbeat timeout), also checking if the sender is the current observed process (indicating a failure) or another process (indicating a false positive failure); after, it checks if a new observer notification was received, which means the previous observer failed and the process now haves a new observer; finally, the library checks for broadcasts[1].

Internal broadcast: An important part of the detection mechanism is failure propagation. When a failure happens, every process has to know about that failure. Typically, a broadcast algorithm is used to propagate the failures. A hypercube-based algorithm (HBA) (BOSILCA

[1] Currently, the broadcast can be either an MPI communicator repair process (indicating that a procedure of communicator repair was initialized); a false positive notification (adding a process that was previously set as failed back to the ring); a failure notification (indicating that a process that failed, where every other process will remove the failed one from the ring); or a checkpoint completion notification (indicating that a checkpoint that was in progress finished).

Figure 7 – Flowchart of the main loop decisions in the heartbeat thread.



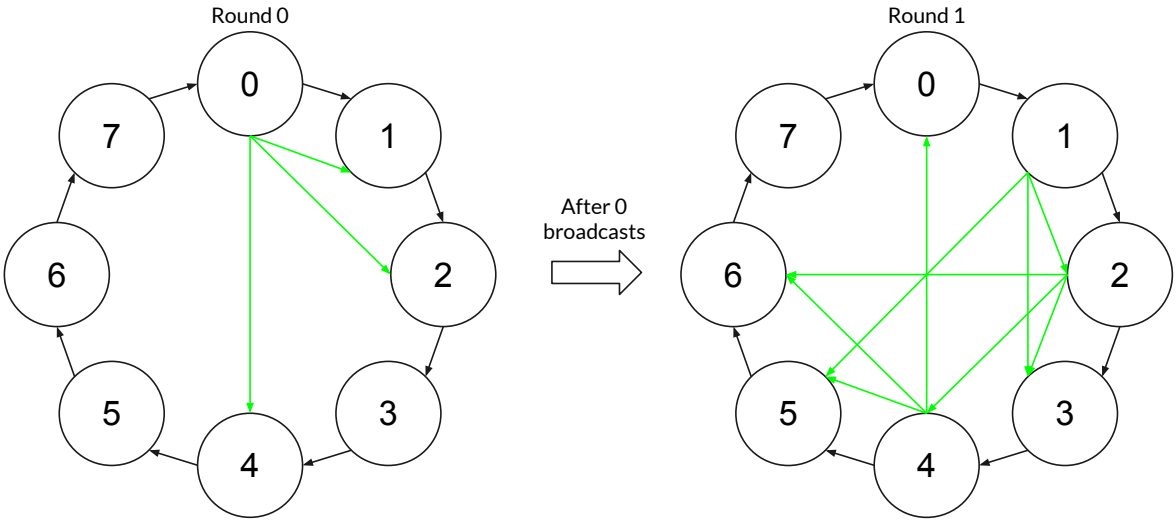
et al., 2018) and a binomial graph algorithm (BMG) (ZHONG et al., 2019) were previously used. Those algorithms present important aspects, like the rapid propagation (in $\log_2(N)$ rounds, where N is the number of processes) and redundant messages, which are essential for FT. OCFTL proposes the use of a Chord-like broadcast, which uses the idea of the request key from the Chord algorithm (STOICA et al., 2001), also proposed in (EL-ANSARY et al., 2003), but different of the original chord-broadcast algorithm, here we keep the message redundancy, which is very important for FT. In this Chord-like broadcast, each process s sends messages to every process r such that $pos_r = (pos_s + 2^i)\ mod\ N$, where i goes from 0 to $\log_2(N)$, and N is the size of the heartbeat ring, following the example of Figure 8. Once per process, this procedure is done upon receiving the first broadcast from another process, replicating the broadcast $\log_2(N) + 1$ times in total.

Additionally, the expected time to achieve a consistent state[2] is $\log_2(N)$ multiplied by the main loop iteration time. This different broadcast aims to maintain efficiency (in terms of FT, with redundant messages and rounds to complete the broadcast) and reduce the number of needed messages to complete the broadcast compared to the HBA or BMG algorithms. Section 6.3 brings a more thorough discussion and experimental evaluation of the internal broadcast.

False-positive failures: Another difference from ULFM's heartbeat is the *false positive* failure detection addition. Meaning that when a process failure is detected but after a while,

[2] The consistent state is defined as the time needed for every process to be notified about a failure.

Figure 8 – Chord like OCFTL broadcast. In this case, in round 0, 0 starts a broadcast sending to 1, 2, and 4. Upon receiving the broadcast, in round 1, 1 sends to 2, 3, and 5; 2 sends to 3, 4, and 6; and 4 sends to 5, 6, and 0. Notice that when 0 receives the broadcast from 4 it will not send the message again. The process stops when every process receives and replicates the message once.



the process recovers (*e.g.,* due to an intermittent network failure), the former observer will capture the message and start a procedure to add it back to the ring. This is achieved using the broadcast procedure discussed before. At an initial stage, to identify a false positive occurrence, the process will be identified as a failed process, and all procedures related to the failure will be made. Only so, the process heartbeat messages will be captured and the process added to the ring again.

Initial position shuffling: Lastly, an initial procedure that shuffles the distribution of the ranks between processes throughout the ring is made. This shuffling minimizes the time needed to detect multiple failures on sequentially-ranked processes that, on a non-randomized rank assignment mechanism, could be running on the same machine, rack, or network. The random distribution of processes is a simple way to solve the problem for most cases. The probability of consecutive ranks assigned to processes on the same nodes decreases as the number of nodes increases. Considering a program is composed of T processes distributing n processes over m machines. If this distribution is in blocks (ranks 0 to 4 on machine 1, rank 5 to 9 on machine 2, and so on), and considering the heartbeat timeout (hb_{to}). If a machine fails, the system will take one hb_{to} to detect the first failure, and another $2 \times hb_{to}$ for each of the $n-1$ remaining failed processes, totaling a timeout of $(2(n-1)+1) \times hb_{to}$ for detecting all failures. This timeout could be reduced to just one hb_{to} if the shuffle makes every MPI process on a machine or rack to be observed by a process on another machine or rack. Since the ranks are uniformly distributed throughout the ring, although the lower bound stated before cannot be guaranteed, the probability

of a worst-case scenario decreases as the number of machines increases, as described by Equation 4.1,

$$P_n = m \prod_{i=0}^{n-1} \frac{1}{T-i}, \tag{4.1}$$

where the probability (P_n) of having n consecutive processes in the ring is m machines multiplied by the probability of the event, which follows $\frac{1}{T} \times \cdots \times \frac{1}{T-n+1}$, with a total of n terms in the product, where T is the number of total MPI processes and n represents the number of processes of each machine. For example, if there is 10 processes for 5 machines ($n = 2$), $P_2 = 0.11$, while having 50 processes for 5 machines ($n = 10$) results in a $P_{10} = 4 \times 10^{-8}$. For further discussion about the shuffling process see Section 6.3.4.

4.3 Failure Mitigation Mechanisms

Once OCFTl can identify and propagate failures, the next step is to reduce the impact of failures in the application's execution. Often this is achieved by using checkpoint/restart and replications, which was explained in Section 2.2. In sequence, it is described how OCFTL uses these approaches to mitigate failure in OmpCluster.

4.3.1 Checkpoint/Restart

Between the various checkpoint libraries options discussed in Section 2.2.2, choosing a checkpointing library that offers the characteristics needed by OCFTL is necessary. Working at the application level, allowing processes to have independent behaviors, which for checkpointing means saving/loading without synchronization of the processes, commonly referred to as uncoordinated checkpointing, are the major requirements. SCR, Veloc, and FTI checkpointing libraries fulfill those requirements. We also considered developing our own checkpointing library, which was discarded because of the time it would take and some libraries already offer the features needed) were evaluated. To enable checkpointing in OCFTL, Veloc (Very-Low Overhead Checkpointing) (NICOLAE et al., 2019) was chosen since it best fits the requirements discussed, since the other restricts the checkpoint to be coordinated only.

OCFTL provides an interface between OmpCluster's runtime and Veloc, which facilitates maintenance and, if needed, the complete exchange of the checkpointing library (as the only location where Veloc is used is inside the fault tolerance library). This interface also provides extra features, like checkpoint interval calculation (YOUNG, 1974) (based on configurable MTBF and Write Speed values) and application notification. The integration of the checkpointing and OmpCluster is made through the event system and in the `OmpTarget interface implementation` part. The calls to procedures that execute Veloc

operations (*e.g.,* saving and loading buffers to/from checkpoints) are made by the event system. The notification about checkpoints and decisions are handled in the `OmpTarget interface implementation` part of OmpCluster since it is needed to restart tasks and interact with the scheduler.

Checkpointing consists of two basic operations. The operations of **saving checkpoints** and **loading checkpoints**. The following paragraphs explain how these operations were designed to be used in OmpCluster.

Saving Checkpoints: When the moment of the next checkpoint is up, OCFTL notifies the system to save a checkpoint. Upon receiving it, the execution of the tasks should be stopped and the desired buffers should be marked to be saved. Veloc will save the buffers locally, and after, send the checkpoint from the local to a persistent directory(*e.g.,* in a distributed file system). After the procedure is complete, the information about the saved buffers is stored to be used in the loading procedure.

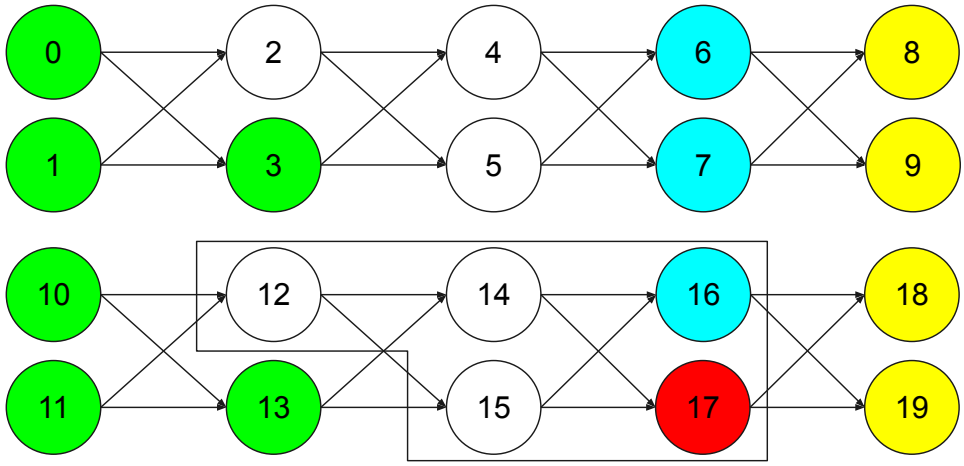
Loading Checkpoints: The load procedure for a checkpoint in OmpCluster starts when a failure is detected. The OCFTL notifies the system of the failure, and the system looks for failed tasks. Any task related to a failed task (be a dependency or dependent of the failed task) which was already executed but not saved will be restarted. The tasks are re-executed before the remaining tasks continue the program. This process avoids the unnecessary overhead that would appear if restarting the entire graph from the last saved checkpoint since many tasks will not depend on the failed tasks. Section 5.2.2.4 discusses the details of how this procedure was implemented.

Figure 9 shows one example of the procedure explained above, the circles represent tasks, and the edges represent the dependency between tasks. Green tasks (0, 1, 3, 10, 11, 13) already have their output saved in a checkpoint, white tasks (2, 4, 5, 12, 14, 15) are tasks that were executed but not saved (will be saved in the next checkpoint), blue tasks (6, 7, 16) are the currently running tasks, yellow tasks (8, 9, 18, 19) are not yet scheduled tasks, and the red task (17) is a failed task. The procedure will look from task 17 for tasks with some dependency relation, stopping at the saved tasks and not scheduled tasks (green and yellow circles). After finishing the search, it is found that tasks 12, 14, and 15 were already finished, and task 16 has an indirect relationship with the failed task, and task 17 that was the failed task, needs a restart.

4.3.2 Replication

Another common procedure to mitigate failure impact is the use of replication. Like checkpointing, there are many ways to use replication. In OCFTL, the location of each process (what node has the process) can be verified, and processes in different machines can be selected for replicating. The replication is not synchronized to avoid overheads

Figure 9 – Procedure of restarting an failed task using checkpointing. Green tasks mean saved tasks, white means executed but not saved tasks, blue means currently running tasks, yellow means not yet executed tasks and the red represents the failed task. This graph shows a possible graph of execution in OmpCluster, where tasks does not communicate with each other during the task execution, only before or after.



from synchronization, which could significantly impact the makespan of the application. The replication is not implemented yet in the OmpCluster, but it is in the scope of future works.

When selecting processes to execute tasks (in the scheduling process), the procedure is as follows: the runtime chooses by pairs, and each process starts executing the tasks. If one of the processes fails, the other can continue. If both of them fail, the task must be re-executed (in the same way as checkpointing procedures). If one of the processes finishes the execution, the replica can be canceled, and both processes are sent back to the process poll. If only one process is available, there are two options, the first is to execute the tasks without replication, and the second one is to mark the task, so next time a process is freed, it will start to execute the marked task as a replication.

As said at the beginning of this Section, replication is not available yet in OmpCluster. It is intended to be added in future OCFTL feature additions. It is explained here since this research also visited the subject of using more than one approach to mitigate failures.

4.4 OCFTL Runtime Support

Once OCFTL can detect and mitigate failures, a few features are implemented to support the OCFTL usage and functionalities. Following, each one of the extra features is described.

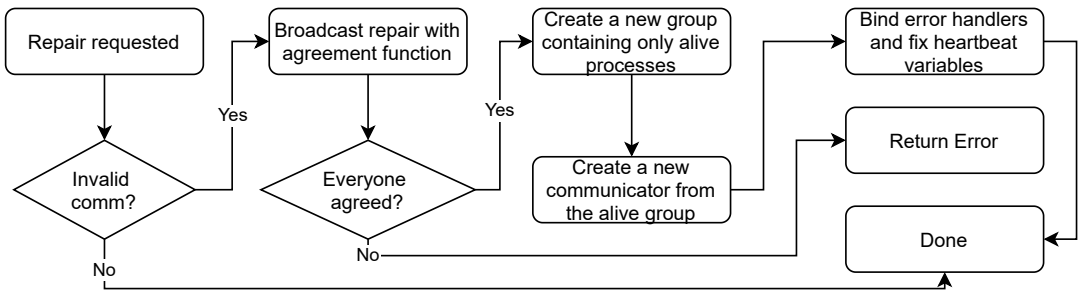
4.4.1 Communicator Shrinkage

OCFTL works over MPI, and MPI provides optimized collective communications between processes. Another consequence of failures is the invalidation of communicators. Even in the presence of failures, MPI can still be used for peer-to-peer operations. However, it will fail if collective operations are utilized since they need every communication process to participate.

To tackle this problem, a repair function similar to `MPI_Comm_Shrink` (available in ULFM (BLAND et al., 2013)) is implemented. When called, this function shrinks the set of processes to contain only alive processes (as determined by the failure detection mechanism described in Section 4.2). The shrinkage only completes successfully if all MPI processes are synchronized, *i.e.,* they have the same view of the whole system with regards to which processes are alive or dead. If their view is not synchronized, the system is said to be in an inconsistent state, which might be reached when a failure happens during the propagation process performed by the library after the detection of a previous failure. In that case, the consistent state will be reached after the end of the ongoing propagation process.

The algorithm to shrink an invalid communicator is described in Figure 10. When the library starts the procedure, it will check if the communicator is invalid (if it has a failed process). If it is invalid, the library will check if every process agrees on the current alive group of processes (if any process disagrees, the procedure will be canceled — this is also discussed as a limitation in Section 6.4). If everyone agrees, a new communicator containing only the group of alive processes will be created.

Figure 10 – Flowchart of communicator repair process.



4.4.2 Notification Callbacks

A notification callback system was built to provide communication between OCFTL and OmpCluster. These callback function notifies the OmpCluster about the FT events, which currently are: `Failure`, `False Positive`, `Checkpoint` and `Checkpoint Done`. To use the callback system, a function need to be bound to it, so, when a FT event list before occurs, that function will be called. The bound function should have a specific procedure

for each one of the possible notifications. In OmpCluster, this function is implemented in the `OmpTarget interface implementation` part. In a stand-alone use of OCFTL, the user should bind their functions to define what to do upon each notification. Each FT notification type is explained in the following statements.

Failure: This notification comes with a parameter representing the failed process. When this is received, the notification handler will add the process to a vector of failed processes and will start the procedures of recovery as described in checkpointing/restart (Section 4.3.1). The vector will be used to define which processes are available for the scheduling of tasks.

False Positive: This notification also comes with a parameter representing the process that came back to the working processes. Here, the process will be removed from the failed processes vector and can be used to schedule tasks again.

Checkpoint: This notification warns the `OmpTarget interface implementation` part that it is time to save a checkpoint. Upon receiving this notification, the handler will do the procedure described in Section 4.3.1, in which it will look for the buffers that are in or will be in use and will save them.

Checkpointing Done: This represents that the checkpoint is done completely. When this notification is received, the handler will do the procedure described in Section 4.3.1, in which the list of buffers used by the tasks in a restart procedure will be loaded.

4.4.3 Process and Communicator States

OCFTL employs gathering state functions that return the constantly updated states of processes and MPI communicator. The state of a process can be `ALIVE` if it is operating normally or `DEAD` if the process is not in operation. In contrast, the state of an MPI communicator can be `VALID` if all processes in the communicator context are `ALIVE`, or `INVALID` otherwise. Those functions are used in OmpCluster to check states in MPI wrappers (discussed in Section 4.4.4). In a stand-alone use of OCFTL, the user could use these functions to, for example, control scheduling.

The process state is used in some OmpCluster runtime procedures to confirm that the other side of a point-to-point communication is `ALIVE`. Before MPI collective calls, the communicator state is used to prevent the runtime from starting an MPI collective operation without all processes `ALIVE` (which could lead to a deadlock). Currently, the library does not employ a solution to failures between the communicator checking (returning a valid communicator) and the collective communication. This is discussed in Section 6.4.

4.4.4 MPI Wrappers

In addition to those features, another possibility to improve FT is the creation of functions to wrap MPI operations. The main reason for wrapping MPI functions is the possibility of deadlock if a side of the communication fails. Those wrappers would give OCFTL better control of the flow of MPI operations since it could make pre and post-processing for each operation. An example of pre-processing is to check the destination or the communicator state, returning an error in case of an invalid communicator or a dead process. As for post-processing, OCFTL could check if the error handling functions reported any abnormalities.

To create wrappers in FT, OCFTL leverages the `"-Wl,-wrap=FUNCTION"` linker option available in GCC and Clang compilers. This option permits it to create two variations of `FUNCTION` symbol, one will be the `"__real_FUNCTION(...)"` and other will be `"__wrap_FUNCTION(...)"`. The first will be redirected to the original implementation of `FUNCTION`, in this case, original MPI functions. The second will be redirected to a custom implementation of the `FUNCTION` that OCFTL provides. When `FUNCTION` is called (without the prefixes), the called function will be the one with the `wrap` prefix, permitting to perform procedures before executing MPI calls without interfering with the regular use of MPI functions.

For the OmpCluster's case, the implementation was evaluated, looking for operations that could lead to deadlock (if the other side of communication failed) so wrappers can be implemented for them. Seven operations were identified: `MPI_Wait`, `MPI_Test` (used as a loop condition), `MPI_Barrier`, `MPI_Comm_free`, `MPI_Mprobe`, `MPI_Send`, `MPI_Recv`. Wrappers implementation and rationale are described below, and each implementation algorithm is presented in Appendix A.

`MPI_Wait`: The problem with `MPI_Wait` is that the function only has a `MPI_Request` and a `MPI_Status` objects as argument, in which it is not possible to identify the other side of the communication. Knowing the other side of communication is essential to verify both sides of the communication are alive. To avoid this problem OCFTL can not wrap directly the `MPI_Wait` function, instead, when using `MPI_Wait` the program will finish and warn the programmer to use a custom version of `MPI_Wait`. `MPI_Iwait` is a version of the real function with an extra argument, the `proc` parameter. This parameter represents the other side of an MPI communication, it can be the rank of another MPI process, or `ft::FT_MPI_COLLECTIVE` if the operation is collective. If the `proc` parameter is equal to `ft::FT_WAIT_LEGACY` the function will fall back to the legacy `MPI_Wait`. Algorithm 2 shows how `MPI_Iwait` works. First, OCFTL checks if FT is disabled or if there is no FT object available, if so, it calls the original `MPI_Wait`. Then, the function implements a loop that keeps checking the flag returned by `MPI_Test` function. Inside this loop, the status of the `MPI_Request` and the state of the communicator or the process (represented by `proc`

parameter) are checked. If the request is completed, it returns success, or and `FT_ERROR` instead.

Algorithm 2 shows the custom MPI function that acts as an alternative to `MPI_Wait`. It is not a wrapper itself but acts like one. This function is typical, it represents any other wrapper construction, which follows the same idea: If FT is disabled, it will switch to the real implementation of the function, acting as normally would do without wrappers; otherwise, it executes the wrappers procedures, which normally employs the check for processes and communicator states. For this reason, the other wrappers implementation are presented in the Appendix A.

Algorithm 2 – OCFTL's custom function alternative to `MPI_Iwait`.

```
1  int MPI_Iwait(MPI_Request *request, MPI_Status *status, int proc) {
2    if (disable_ft || ft_handler == nullptr || proc == ft::FT_WAIT_LEGACY) {
3      return __real_MPI_Wait(request, status);
4    }
5    assertm((proc >= ft::FT_MPI_COLLECTIVE) && (proc < ft_handler->getSize()),
6            "Waiting for request with invalid rank participating.");
7    int test_flag = 0;
8    while (!test_flag) {
9      MPI_Test(request, &test_flag, status);
10     if (proc == ft::FT_MPI_COLLECTIVE) {
11       // If it is a collective call
12       if (ft_handler->getCommState() != ft::CommState::VALID) {
13         MPI_Request_free(request);
14         return ft::FT_ERROR;
15       }
16     } else {
17       // If it is a point-to-point call
18       if (ft_handler->getProcessState(proc) != ft::ProcessState::ALIVE) {
19         MPI_Request_free(request);
20         return ft::FT_ERROR;
21       }
22     }
23   }
24   return ft::FT_SUCCESS;
25 }
```

`MPI_Test`: This MPI function provides a flag determining if the associated `MPI_Request` is finished or not. The flag is used as a condition in the event system to wait for all event requests to be complete. In this case, if the `MPI_Request` can not be completed because of a process failure, the loop would never be broken, and the program would deadlock. As well as `MPI_Wait`, the `MPI_Request` and a `MPI_Status` does not provide the information about the other side of the communication. Because of that, OCFTL wraps the `MPI_Test` function warning about the use of the function in such case, redirecting to use of the custom `MPI_Itest` function provided by OCFTL, this function is presented by the Algorithm 5. As well as the `MPI_Iwait`, this new function uses the same arguments of

the original `MPI_Test` with the addition of another argument (`proc`), which defines the other side of the communication. This function first checks if FT is disabled or if there is no FT object existent. If so, calls the original `MPI_Test` (one can also use `proc` equals to `ft::FT_TEST_LEGACY` to fall back to the original implementation, if it is used in a different way than the discussed). After, the function checks the MPI communicator is valid (if `proc` equals to `ft::FT_MPI_COLLECTIVE`) or if the other process of the communication is alive (if `proc` is a rank of a MPI process). If there is a problem with the communicator or the other process, the function will return `ft::FT_ERROR` (which can be used to break the loop). Otherwise, the library will call the original `MPI_Test` function the fill the flag argument and at the end returns `ft::FT_SUCCESS`.

`MPI_Barrier`: Barrier is also another very common function with can cause deadlock. Algorithm 7 shows the wrapper for `MPI_Barrier`. Same as for `wait`, an initial checking is done, and then it is checked if the status of the communicator is valid. In sequence, it calls the `non-blocking` variation of the `MPI_Barrier` and waits for its request. If the request was not completed (or if the `non-blocking` version failed), OCFTL will try to repair the communicator. If the new communicator is valid, OCFTL will try to execute the `non-blocking` version (now in the new communicator) again. It is important to note that this wrapper also uses the `MPI_Iwait` function described before.

`MPI_Free_comm`: This function is usually employed at the end of programs. It is a collective function that would deadlock if running with failed processes. The wrapper for this function is straightforward. If the communicator is `VALID`, it falls back to the real `MPI_Free_comm` function. If it is not, it simply skips this function. Since this is a cleanup function, it would not affect the application's final result, although it would case some memory leak.

`MPI_Mprobe`: The `Mprobe` function is a point-to-point function that tests for a message, and if there is a corresponding message, it will store the message in a specific object to be later captured by an `MPI_Mrecv` function. The problem is that the `Mprobe` is a blocking operation[3]. If it is testing a message from a dead process, it would be stuck waiting. To solve the problem, OCFTL changes the `Mprobe` to a loop containing the non-blocking version of it, the `MPI_Improbe`. Different from the other non-blocking functions in MPI, this version does not attach a request. Instead, it simply checks if there is or not a corresponding message. In the wrapper version, it is also checked if the corresponding source rank of the message is `DEAD`. If it is, the wrapper will return an error.

`MPI_Send`: This `Send` operation is a blocking send. It is known that for blocking `Send` operation, it needs to be completed on the source rank, not necessarily meaning that the receiving part has received it, especially in cases where the buffer sent is small (FORUM,

[3] For instance, a blocking operation means that the MPI operation only needs to complete locally (on the MPI rank that is executing it), a synchronous operation means that the MPI operation needs to be completed on every MPI rank participating.

2015). However, there are cases where the sender can not complete the operation if the buffer is large and the destination part is `DEAD`. The wrapper for the `Send` operation changes the use of the blocking operation to the corresponding non-blocking and uses the corresponding request object with the custom `MPI_Iwait` function.

`MPI_Recv`: The blocking `Recv` operation stalls the program until it receives the message from the source. Suppose the source of the message is `DEAD`. In that case, this operation could deadlock if the MPI distribution does "know" how to discover this kind of failure (for instance, the MPICH distribution can detect the failure of processes through the TCP sockets, so this operation would not deadlock, but lead to an error). The wrapper changes the blocking `Recv` to the `MPI_Mprobe` to solve the problem. When it returns, if there is an error, it is propagated to the `Recv` return value. Otherwise, if the probe function returns a message, the `Recv` wrapper uses the `MPI_Mrecv` to capture the probed message.

As a general rule, the blocking and synchronous MPI operations (which needs to employ some kind of synchronism) are the main problem related to program deadlocks. The MPI standard proposes a non-blocking version with a request object associated for almost every operation (few exceptions like the `MPI_Mprobe`). With that request and specifying the other side of communication, one can use the `MPI_Iwait` version of `MPI_Wait` to wait for the completion of that request. This procedure makes a generic alternative which would not deadlock and could be used with the most part of MPI operation with the same behavior, especially when using OCFTL as a stand-alone library.

5 OCFTL usage on OmpCluster

This Chapter discusses the integration of OCFTL, proposed in Chapter 4 with OmpCluster, discussed in Chapter 3. Section 5.1 explains the integration of OCFTL at the `OmpTarget target implementation` level of OmpCluster and Section 5.2 describes the integration of OCFTL at the `OmpTarget interface implementation` level of OmpCluster.

5.1 Integration at the Target Level

As explained in Section 3.1, in the `OmpTarget target implementation` level, is where the fault tolerance and event system implementations reside. The integration at this level is straightforward. The wrappers already do the most significant integration since they are used the same way the original functions (besides the particular case of the `MPI_Wait` and `MPI_Test`. The return value of the wrappers are used to invalidate the event in case of `Failures`. At this level, there is also the fault tolerance library initialization. The location of this integration is at the same level as OCFTL implementation, like shown in Figure 4, presented in Chapter 4.

The remaining integration point for OCFTL is the OmpCluster's event system.. The event system is responsible for executing procedures, so it is also used to call the procedures of checkpointing. The creation of three new events makes this possible:

RegisterCPPtrs: This event is responsible for registering a set of regions of memory to be checkpointed. As saw in Section 2.2.2, before actually checkpointing, it is necessary to register the pointers and size of each memory region. This event also returns the information necessary to reload the buffer: the `id` of the buffer, the `MPI rank` that saved the checkpoint, and the `version` of the checkpoint.

Checkpoint: This event saves a checkpoint for all previously registered buffers. This function calls the checkpointing function and waits for its completion.

Recovery: This event is responsible for re-loading a saved buffer. In this stage of implementation, we used one event per buffer. The `Recovery` implements Algorithm 4 explained in Section 4.3.1 for the case of one buffer.

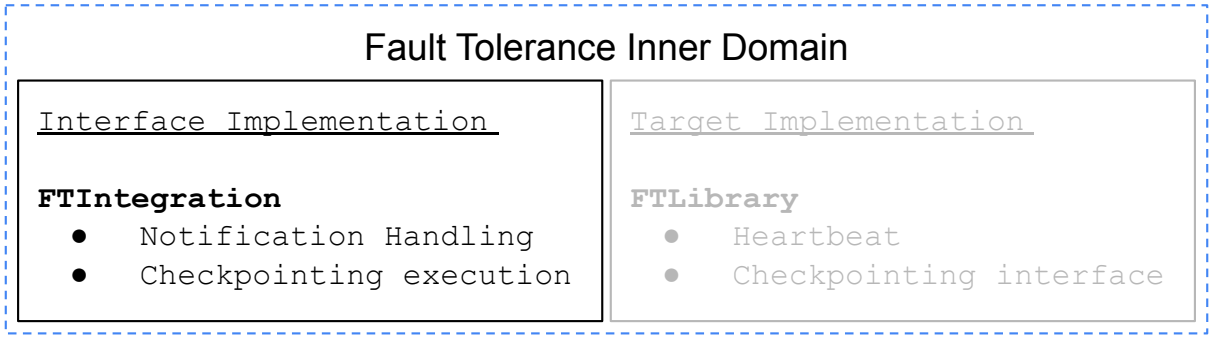
5.2 Integration at the interface level

The integration at the `OmpTarget interface implementation` level is more complex than in the `OmpTarget target implementation` level. At this level, there is the

coordination of the target task executions. The execution is restricted to the host process, differently of what happens at the `OmpTarget target implementation` level, which is executed by every MPI process. For any FT procedure that involves any worker process, a corresponding event is created, whose type is one of the events described in Section 5.1.

The integration at this level is divided in two parts: the `notification callback`, which is responsible for listening to the OCFTL FT notifications (see Section 4.4.2) and is further explained in Section 5.2.1; and the `notification handler` class, which is responsible for executing FT procedures on the `OmpTarget interface implementation` level and is further discussed in Section 5.2.2. Both parts of the integration are present in the highlighted part of architecture shown in Figure 11.

Figure 11 – Location of fault tolerance integration inside OmpTarget.



5.2.1 Notification Callback Function

The notification callback function is the most crucial part of the integration at the `OmpTarget interface implementation` level since it is responsible for determining what will be done depending on which notification is received. This function will be bound to the notification callbacks, explained in Section 4.4.2. The current implementation of the notification callback function is presented in Algorithm 3.

As seen in Algorithm 3, upon receiving a notification, the function acts based on the notification that was received. As seen in Section 4.4.2, there are four types of notifications: `FAILURE`, in which the `notification handler` will handle the failed device and execute the restart procedures; `FALSE_POSITIVE`, in which the `notification handler` will add back the now-alive process back to the alive devices list; `CHECKPOINT`, in which the procedures of checkpointing will be initialized; and `CHECKPOINT_DONE`, in which the procedures of checkpointing will be finalized.

Algorithm 3 – Notification callback function.

```
1  void FTNotificationCallback(FTNotification notify_handler) {
2    switch (notify_handler.notification_id) {
3    case FTNotificationID::FAILURE: {
4      ft_handler.handleFailedDevice(notify_handler.device_id);
5      ft_handler.executeRestart(notify_handler.device_id);
6      break;
7    }
8    case FTNotificationID::FALSE_POSITIVE: {
9      ft_handler.removeFailedDevice(notify_handler.device_id);
10     break;
11   }
12   case FTNotificationID::CHECKPOINT: {
13     ft_handler.startCP();
14     break;
15   }
16   case FTNotificationID::CHECKPOINT_DONE: {
17     ft_handler.finishCP();
18     break;
19   }
20   }
21 }
```

5.2.2 Notification Handler

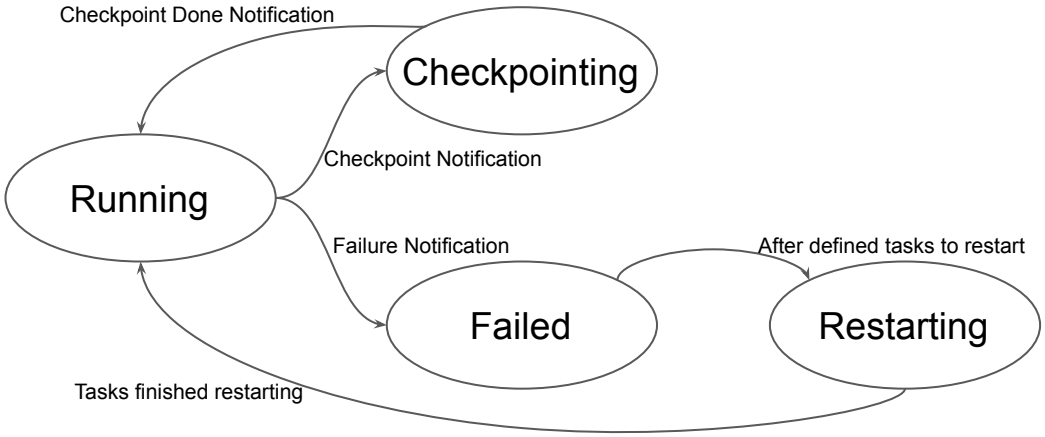
This class (another `C++` class) is responsible of executing the FT procedures on the `OmpTarget interface implementation` level based on a notification received by the `notification callback function`. To better organize the description of this class, it is subdivided in four categories: the `interface operation`, discussed in Section 5.2.2.1, the `tasking and mapping`, explained in Section 5.2.2.2, the `devices handling`, discussed by Section 5.2.2.3, and, the `checkpointing`, explained in Section 5.2.2.4.

5.2.2.1 Interface Operation

Before executing any fault tolerance operation at the `OmpTarget interface implementation` level, some action is needed to control OpenMP targets execution. As seen in Section 3.1, in the host process of OmpCluster, there will be local OpenMP threads running. Those threads are controlled by OpenMP and not by OmpCluster, so, before executing any fault tolerance operation at the `OmpTarget interface implementation` level, those threads are temporary suspended by the `NotificationHandler` class. To do that, the `NotificationHandler` class keeps a count of executing tasks (incrementing at the start of each target function and decrementing at the end). It also defines execution states: `RUNNING`, the normal state with no pending fault tolerance procedures; `CHECKPOINTING`, meaning that a checkpoint procedure is pending or executing; `FAILED`, meaning that the system encountered a failure and initial procedures are being done; and, `RESTARTING`,

indicating that the system is re-executing tasks that failed. Figure 12, shows the state transitions. From the `RUNNING` state, upon receiving a checkpointing of failure notification, the state will be changed to `CHECKPOINTING` or `FAILED` respectively. For the first case, after receiving the notification of checkpoint completion, the state will be changed back to `RUNNING`. For the second case, the `NotificationHandler` class will do the procedures to find the failed tasks to restart and load the required buffers before changing the state to `RESTARTING`, when the tasks will be restarted. When all the tasks have finished the restart procedure, the state will be changed to `RUNNING` again.

Figure 12 – States diagram for the execution states defined by the FT handler at interface level.



The executing tasks counter allows the handler to determine if the fault tolerance procedure can be executed or if it needs to wait further. During the wait, the handler holds a `conditional variable`. This `conditional variable` is used when a thread finishes the execution of a task and tries to execution a new one. If the conditional variable is held, the thread is suspended and can only start once the variable is freed. When all local thread finishes the currently executing tasks, the fault tolerance procedure can occur. This procedure is crucial and permits the `NotificationHandler` class to execute procedures without the need to care for the states of executing tasks (since there will be no tasks executing).

As explained in Section 3.1, every OpenMP target task is associated with a specific function[1] at the `OmpTarget interface implementation` level. Knowing which function was associated is essential for the restart process. So before starting each target task execution, the handler saves which function and arguments were associated with the target task.

[1] Currently, these functions are: `data_begin`, `data_update`, `data_end`, `target`,`target_teams`, and the `nowait` variants.

5.2.2.2 Tasking and Mapping

The `NotificationHandler` class also holds information about tasks and task mapping. The integration of OCFTL to OmpCluster demanded a copy of the task graph since it is used to search for tasks that are dependencies or dependents of failed tasks. OmpCluster already has a structure that holds the necessary information. For the `NotificationHandler` class, a reference to this data structure is used, and during the execution of the program, another structure is used, the `Device-Task` map.

The `Device-Task` map holds the information about which task is running on which worker process. This is essential when a failure occurs since it is the primary source of knowledge to determine if a task failed with the failed worker process. The other source of finding failed tasks is searching for tasks that have the associated event returned failure. However, if an event returns failure, we expect that a worker process failed, so the failed task would be captured by the failure notification.

5.2.2.3 Devices Handling

Device handling is also done by the `NotificationHandler` class. OmpCluster itself has a map for each device, which holds all the information about the device, including the data maps. For the integration, a map that only holds information about how many workers are available and the state of each worker is created.

The knowledge of how many worker processes are available is critical since each time a process state is modified, the task graph is re-scheduled. For that reason, the information about how many workers are available is used. The output of the scheduler is `id` of the device, for each scheduled `id` the handler translates to the actual worker `id` that represents that position. This is exemplified in the paragraph related to `Restarting Tasks`, in Section 5.2.2.4.

5.2.2.4 Checkpointing

Concerning checkpointing, since OmpCluster is distributed, each process needs to save the buffers that are stored. The more straightforward way is to save each allocated buffer. However, as data management permits various instances of the same buffer (in different devices), this would save different versions of the buffer, which make the loading procedure more difficult (selecting from which process the buffer would be loaded). Another problem is that some buffers may not have been used up to that point, so they are not allocated on any device. But, it is also necessary to save them since the remaining tasks will use them. There is three checkpointing operations in the integration which are discussed in the paragraphs below: how to execute the save procedure of a checkpoint; how to execute a

load procedure of a checkpoint; and how to re-execute the failed tasks after the checkpoint was loaded.

Saving a checkpoint: Before taking a checkpoint the handler evaluates each remaining task marking every buffer used to be saved. For each buffer, the handler will query the data mapping provided by the `OmpTarget interface implementation`. If the buffer does not have a mapping, it means that the buffer is not allocated on any device. In this case, the host process (MPI rank 0) will save that buffer. If there is at least one map associated with the buffer, the process that will save the buffer is chosen based on the type of the maps associated with the buffer.

OmpCluster's data management has three types of buffer mapping entries. A `LOCAL` entry, meaning it belongs only to a specific target region and will not be forwarded to other devices. A `PROXY` entry not associated to a specific target region, but can be forwarded to other devices. And, a `LINK` entry, which is a forwarded buffer always linked to a `PROXY` entry.

Since OpenMP's local threads will be temporarily suspended, no tasks will be executing, thus no `LOCAL` entries exist. Because of that, the handler only needs to take into account `PROXY` and `LINK` entries. Two situations are possible: when there are only a `PROXY` map associated with the buffer with no `LINK` entries; and when there are at least one `LINK` entry active. In the first case, the first device (MPI rank 1) saves the buffer to its checkpoint file, since every `PROXY` entry is held by the first device. In the second case, the process associated with the last `LINK` added will be chosen, since it represents the last version of the buffer.

After associating each buffer to a process, the handler creates a `RegisterCPPtrs` event for each process to register the marked buffers. In sequence, the handler creates a `Checkpoint` event to save the checkpoint and waits for the `Checkpoint Done` notification. Finally, the handler finishes the checkpoint and frees the local OpenMP threads, and the program execution will resume.

Loading a checkpoint: During restart, the handler loads all buffers that are needed by the tasks marked to restart. First, the handler iterates each task querying the checkpoint map[2] and registers the buffers to load. Second, the handler selects the process to load each buffer based on the checkpoint map. If it is the host process, no map for the buffer will be created. Otherwise, if the buffer was loaded and already has a map, that map is discarded, and a new map is created. For each buffer, a `Recovery` event is created.

The `Recovery` event implements the procedure described by Algorithm 4. This procedure is needed since Veloc does not permit complete control of the checkpoint file name. When a load procedure is done, Veloc appends its current rank (the one with it was

[2] The checkpoint map holds where each buffer was saved in the last checkpoint process.

Algorithm 4 – Given the three FT functions, the load process should follow the function LoadBuffers().

```
1  // Finalizes current checkpoint context
2  void cpLoadStart();
3  // Loads buffer "id" from an checkpoint context
4  int cpLoadMem(int id, int s_cp_rank, int ver, size_t count,
5                size_t base_size, void *memregion);
6  // Initalizes original checkpoint context
7  void cpLoadEnd();
8
9  void LoadBuffers() {
10   ...
11   ftObject.cpLoadStart(); // one call
12   for (auto buf : buffers)
13     ftObject.cpLoadmem(buf.id, buf.cp_rank, buf.cp_ver, buf.count, buf.size,
14                        buf.address); // one call per buffer
15   ftObject.cpLoadEnd(); // one call
16   ...
17 }
```

initialized) to the file name and loads the checkpoint. This makes the process of loading checkpoints created by other ranks more intricate. To do so, the current execution of Veloc is finalized and a new one (with the adequate rank, i.e., the rank that saved the checkpoint) is run. After the checkpoint is loaded, this new execution is finished and the execution with the original rank is restarted. This is done by the three steps in the algorithm: the function `cpLoadStart` finishes the current instance of Veloc; `cpLoadmem` starts an instance of Veloc in the rank that saved the buffer, loads the buffer from the file and finishes that instance of Veloc; and finally `cpLoadStart` starts the Veloc in the initial rank again.

Now that the saving and loading procedures are explained, the last part of exploring is the `Restarting` procedure:

Restarting Tasks: Before re-executing tasks, the OpenMP local threads are suspended temporary and the handler checks for any buffer that is used by the tasks that will be restarted. Then, each buffer is loaded from the last valid checkpoint.

In current version of OCFTL, the re-execution of tasks is done sequentially, based on the history of the executed tasks. The history maintains the order of the tasks execution. Since the tasks are re-executed sequentially, there is no concern about the dependencies. Further discussions about the impact of this type of re-execution are made in Section 6.3.1.2. Currently the `OmpTarget interface implementation` does not permit the scheduling of the tasks marked to restart, so OpenMP local threads that are suspended can not be used.

The procedure described in Section 4.3.1 is done for each failed task in a `failed-tasks` list to define what tasks will be restarted. Basically, any task with a direct or

indirect dependency with the failed task and was already executed (but not saved by a checkpoint) will also be marked to restart. As seen in Section 5.2.2.1, there will be no executing tasks, so the handler does not need to bother with that kind of task.

After re-executing the tasks, it is necessary to re-schedule the remaining tasks. This new schedule will be done with fewer available devices since the failed devices are excluded from the available devices. Figure 13 shows an example if the system had five available devices (device 0 to device 4), and device 3 failed, the scheduler will schedule the tasks with devices ranging from 0 to 3 (notice that only four devices are available after the failure, the devices 0, 1, 2 and 4). This is possible since the handler translates the scheduled device to the available devices. If the scheduled device were 0, 1, or 2, the translated device would remain the scheduled one, but if the scheduled device were 3, the translated device would be the device 4 (the fourth device in the list of the available devices).

Figure 13 – Translation of the output of the scheduler.

Finally, to execute restarting, every task's non-checkpointed information that is needed to re-execute the tasks is stored in memory by the host process (since OCFTL does not expect failures in the host process, as discussed in Section 6.4). The stored information contains all the data necessary to call the interface functions again: the associated target function and its arguments. So, the handler based on the task execution history, re-executes each task sequentially. The handler uses the data store about the task (which function and arguments) to replicate the call done by the OpenMP runtime.

6 Experimental Evaluation

This chapter discusses the experiments performed to assess different MPI implementation behaviors, to evaluate OCFTL performance characteristics, and to discuss the current limitations of this work. Section 6.1 presents the test environment, while Section 6.2 discusses the behavior of different MPI implementations in the presence of failures. Then, Section 6.3 describes the experiments to evaluate OCFTL's performance. And Section 6.4 presents the current limitations and possible solutions.

6.1 Test Environment

All tests were performed in a distributed environment. The cluster used was the Santos Dumont supercomputer (SDumont), located at the LNCC (Laboratório Nacional de Computação Científica). SDumont is one of the biggest supercomputers in Brazil. For the reported experimental results, the computing nodes used were the model named B710. These computing nodes feature 64Gb of RAM and 2xCPU Intel Xeon E5-2695v2 Ivy Bridge, each with 12 cores (24 threads) running at 2.4 GHz (3.2 GHz Turbo Boost). The network interface is InfiniBand (56 Gb/s). The number of cores was selected according to each test. The source code for all the tests of this research is available at the repository: <https://gitlab.com/phrosso/ftmpi-tests>.

To execute the tests, two MPI distributions with some configuration variations were used. MPICH version 3.4.2 (newest stable release at the time) and Open MPI version 4.1.0 (the last version compatible with ULFM). All MPI distributions were configured with support to multi-threading (`MPI_THREAD_MULTIPLE`), and compiled with UCX[1] and without. UCX stands for Unified Communication X and is used to accelerate the performance of networks in HPC and is used in OmpCluster. ULFM version 2.1 was also used in some tests. This version of ULFM is based on Open MPI version 4.1.0. During the experiments, one extra execution configuration parameter was used: the recovery flag, `"-enable-recovery"` for Open MPI and ULFM, or `"-disable-auto-cleanup"` for MPICH. Algorithm 15(Appendix E) shows all configuration options for each version of MPI.

6.2 MPI Behavior

OCFTL depends on the behavior of each MPI implementation. Each MPI implementation has ample freedom to implement the MPI standard, so it is expected that

[1] https://www.openucx.org/

different MPI implementations present different behavior in the presence of failures. To properly implement a fault tolerant solution in OCFTL, it was necessary to evaluate the behavior of each implementation in different critical execution scenarios. For example, *what happens if a collective operation is made using a communicator with the presence of a failed node?* or, *what happens if a blocking send call is made to a failed node?* These tests focus on evaluating such behaviors.

Some point-to-point and collective operations were selected for these tests. The operations were executed in all variations (blocking, non-blocking and synchronous when applicable) by two processes. Some operations have a source and a destination processes (*e.g.,* `MPI_Send`) while in others every process has both jobs (*e.g.,* `MPI_Allreduce`). Where applicable, two instances of the tests were being executed, one killing the MPI rank 0 and other killing the MPI rank 1. Each run is considered unsuccessful if it times out. The objective of these tests is to check if the operation deadlocks, not to evaluate the operation's correctness in the presence of failures. The benchmark programs and runtime configurations are available on the `Behavior` folder of the aforementioned Git repository.

Tables 2 and 3 show the results of the experiments with a few MPI operations on MPICH distributions and OpenMPI distributions, respectively. For both tables, each cell represents the results of each variation of a operation when applicable, in the order `Blocking` / `Non-Blocking` / `Synchronous`. Possible values are: `ok` if the program finished (with or without errors); `to` if the program timed out; and `(-)` if the variation does not exist. Finally, for each MPI distribution, the tests simulate the failure of a MPI process by killing this process (rank 0 or 1, where it is applicable).

Table 2 – Behavior of different MPI operations for MPICH and MPICH+UCX. (`ok` means program finished and `to` means program timed out)

	MPICH		**MPICH+UCX**	
	Kill P0	**Kill P1**	**Kill P0**	**Kill P1**
MPI_Allreduce	(ok / ok / -)	(ok / ok / -)	(to / to* / -)	(to / to* / -)
MPI_Barrier	(ok / ok / -)	(ok / ok / -)	(to / to / -)	(to / to / -)
MPI_Bcast	(ok / ok / -)	(ok / ok / -)	(to / to* / -)	(to / to / -)
MPI_Bsend	-	(ok / ok / -)	-	(to / to / -)
MPI_Gather	(ok / ok / -)	(ok / ok / -)	(to / to / -)	(to / to* / -)
MPI_Recv	-	(ok / ok / -)	-	(to / to* / -)
MPI_Reduce	(ok / ok / -)	(ok / ok / -)	(to / to / -)	(to / to* / -)
MPI_Send	-	(ok / ok / to)	-	(to / to / to)
MPI_Wait	-	(ok / - / -)	-	(to / - / -)

Representation: (Blocking / Non-Blocking / Synchronous)

Table 2 shows that for MPICH configured without UCX, in all tests, but the synchronous send operation, the program finished. This was expected since MPICH in its basic configuration has limited FT support. For MPICH with UCX, every program

timed out after 5 seconds (a regular execution of the test program runs in less than 1 second). The value with `(*)` next a result means that the program does not time out during the evaluation of the target function, but on the execution of `MPI_Request_free` or on `MPI_Finalize` functions.

Table 3 – Behavior of different MPI operations for OpenMPI and OpenMPI+UCX. (`ok` means program finished and to means program timed `out`)

	OpenMPI		**OpenMPI+UCX**	
	Kill P0	**Kill P1**	**Kill P0**	**Kill P1**
MPI_Allreduce	(to / ok / -)	(to / ok / -)	(to / to / -)	(to / ok / -)
MPI_Barrier	(to / to / -)	(to / to / -)	(to / to / -)	(to / to / -)
MPI_Bcast	(to / ok / -)	(to / to / -)	(to / to / -)	(to / to / -)
MPI_Bsend	-	(to / to / -)	-	(to / to / -)
MPI_Gather	(to / to / -)	(to / ok / -)	(to / to / -)	(to / to / -)
MPI_Recv	-	(to / ok / -)	-	(to / to / -)
MPI_Reduce	(to / to / -)	(to / ok / -)	(to / to / -)	(to / to / -)
MPI_Send	-	(to / ok / to)	-	(to / to / to)
MPI_Wait	-	(to / - / -)	-	(to / - / -)

Representation: (Blocking / Non-Blocking / Synchronous)

Table 3 shows that for Open MPI without UCX, for most of `Non-Blocking` operations the program finishes while for `Blocking` operations every program times out. For the configuration with UCX, every program, but the `Non-Blocking` `MPI_Allreduce` (when the rank 1 process was killed) has failed.

This experiment shows that different MPI distributions can have different behaviors depending on which operation is used. These results explain the reason OCFTL employs the function wrappers, discussed in Section 4.4.4. Although those wrappers are specific for OmpCluster, they will avoid the program of being deadlocked independent of what MPI distribution is used.

In general, the non-blocking operations are the most failure-friendly operations. However, these operations can demand the use of a `wait` operation or a `test` operation inside a loop, which could lead to a deadlock. OCFTL provides non-deadlockable `wait` and `test` operations to be used together with the non-blocking operations. This way, the program will not deadlock because of failures related to MPI.

6.3 OCFTL performance evaluation

This section presents the tests that aim to evaluate OCFTL performance characteristics. Section 6.3.1 discusses the integration of OFCTL and OmpCluster, looking at the overhead and correctness. Section 6.3.2 brings the results that evaluate the impact of heartbeat frequency and timeout properties. Section 6.3.3 evaluates the internal broadcast

of OCFTL compared to other broadcasting algorithms. Finally, Section 6.3.4 discusses the locality problem presented in Section 4.2.

6.3.1 Using OCFTL in OmpCluster

There are two main interests in testing OCFTL with OmpCluster. First, overhead of OCFTL on the execution of programs running on the OmpCluster architecture was evaluated (Section 6.3.1.1). Second, the correctness of programs using OmpCluster with OCFTL in presence of failures was verified (Section 6.3.1.2).

6.3.1.1 OCFTL Overhead

OmpcBench script was employed to evaluate the overhead imposed by OCFTL on OmpCluster programs. This script executes the TaskBench[2], with different task graphs as input. Since OmpCluster is in constant changes, these tests merge the latest modifications related to fault tolerance[3]. These tests dedicated to the evaluation of OCFTL overhead do not include failure simulation and leverages OmpCluster's container architecture, with a specific container built with the latest OCFTL modifications to execute the tests. Table 10 (Appendix E) shows the values of OCFTL variables used in this experiment.

Due to OmpCluster's current limitations[4], these tests use 1+8 nodes (1 for the head process and 8 for worker processes). The size of TaskBench graphs is 8×16 (width by depth). Four graphs were tested to explore different logical and communication patterns. `stencil_1d`, which has the pattern of a one dimensional stencil application. `fft`, which has the pattern of an FFT (Fast Fourier Transform) application. `trivial`, which is a pattern that has no dependencies and communication between the tasks. And `no_comm`, which has a logical depth dependency with no communication between the tasks. Three of the five OmpcBench defined sizes were used to define the tasks' duration: `micro`, which simulates $100\mu s$; `small`, for $50ms$ tasks; and `large`, that represents $1s$ tasks. TaskBench also permits configuring the ratio between computation and communication. For the sake of simplicity, these tests ran with a ratio of one to one, meaning tasks will compute roughly the same as they communicate. For each test, a total of 10 samples were used and the confidence interval of 95% was calculated using the Bootstrap method (EFRON; HASTIE, 2016) with 10000 iterations.

Table 4 shows OCFTL's overhead over OmpCluster (Baseline). As seen in Table 4, for the `no_comm`, `trivial` and the `micro` benchmarks of `fft` and `stencil_1d` no significant

[2] Task Bench is a configurable benchmark for evaluating the efficiency and performance of parallel and distributed programming models, runtimes, and languages (SLAUGHTER et al., 2020).

[3] The version of OmpCluster and OCFTL are available at <https://gitlab.com/phrosso/llvm-project> in the default branch `test/ft-feature-test`

[4] At the time of this experimental evaluation, OmpCluster is only able to run small programs with a limit of 255 tasks.

Table 4 – Comparision between OmpCluster with and without OCFTL enabled for different Task Bench graphs

		Micro		Small		Large		Overhead Geo. Mean
		Avg (s)	CI 95%	Avg (s)	CI 95%	Avg (s)	CI 95%	
fft	**Overhead**	**1.07x**		**0.80x**		**0.79x**		**0.87x**
	Baseline	0.1162	[0.1120, 0.1228]	3.4857	[3.4340, 3.5359]	58.572	[57.9310, 59.1593]	
	OCFTL	0.1084	[0.1079, 0.1127]	4.3827	[4.3125, 4.4656]	74.5599	[72.6821, 76.6640]	
no_comm	**Overhead**	**0.97x**		**1.02x**		**0.99x**		**0.99x**
	Baseline	0.0856	[0.0837, 0.0873]	2.4779	[2.4509, 2.5025]	38.0695	[37.5677, 38.7132]	
	OCFTL	0.0881	[0.0866, 0.0895]	2.4378	[2.3974, 2.4774]	38.3335	[37.8999, 38.7799]	
stencil_1d	**Overhead**	**1.01x**		**1.26x**		**1.25x**		**1.16x**
	Baseline	0.1164	[0.1150, 0.1183]	4.3735	[4.2812, 4.4838]	73.3837	[71.0218, 76.5664]	
	OCFTL	0.1158	[0.1137, 0.1177]	3.484	[3.4422, 3.5256]	58.6643	[57.9112, 59.3839]	
trivial	**Overhead**	**1.02x**		**1.00x**		**1.00x**		**1.01x**
	Baseline	0.0702	[0.0694, 0.0712]	2.1375	[2.1209, 2.1548]	35.1977	[35.0825, 35.3095]	
	OCFTL	0.0691	[0.0677, 0.0708]	2.1369	[2.1254, 2.1490]	35.0637	[34.8730, 35.2436]	

overhead can be observed. For the `Small` and `Large`, a difference can be observed, in the `fft` benchmark, both types showed improvement when comparing the use of OCFTL to the baseline, while in the `stencil_1d` benchmark, both types showed worse results when comparing the use of OCFTL to the baseline. Both `fft` and `stencil_1d` have similar patterns, where each (but the initials) task has three dependencies so it is expected the results for them to be similar. So the type of task graph should not impact on the overhead generated by OCFTL since the parallelization is similar. These outliers could be caused by external sources, like CPU speed changes, among others.

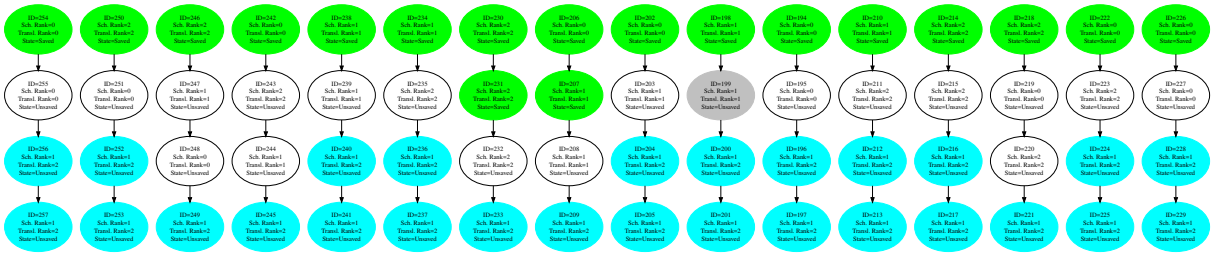
The tests were executed evaluating OCFTL with and without function wrappers with similar results. Results in Table 4 present the version without the wrappers since the program should not have failures. If a false positive failure is encountered, the function wrappers will start the execution of recovery approaches, like repairing a communicator, which would affect the execution time (note that these tests were intended to evaluate OCFTL without failures).

6.3.1.2 Execution Correctness

To evaluate the correctness of OCFTL as a failure mitigation approach in OmpCluster. The results of a `Block-Matrix-Multiplication` program were evaluated. This program does two things. First, it executes a block of code using the OpenMP directives, and since it is compiled with OmpCluster, it will offload the computation as discussed in Section 3.1 using OCFTL as FT library. Second, it executes a sequential version of the algorithm in the rank 0, not using OmpCluster's architecture. The configuration for the matrix was a square matrix of 2800×2800 and blocks of size 700, so 64 target tasks will be generated. Algorithm 17 (Appendix E) shows the two versions of the algorithm.

Four nodes (1 host and 3 workers) were used to execute the program. Comparating the execution times, the sequential version averages around $65s$ while the parallel version averages approaximatelly at $7s$, when a failure is injected, the time averages around $12s$.

Figure 14 – Final task graph of a block matrix multiplication program



To execute the test, the library first forces a checkpoint[5] And after that, injects a failure on the worker 2. Figure 14 shows the task graph. Highlighted in green are the tasks that were saved during the checkpoint. When the failure notification arrives, the executing task on worker 2 (Rank 1 in the graph) was the one with ID equal to 199 (highlighted in grey). Since all tasks before 199 were saved, it was the only task re-executed, recovering a total of 3 buffers. The tasks that had the scheduled device rank different from the actual rank are highlighted in cyan. Note that the translation will be only different after failures happen.

At the end of each execution, the resulting matrix of the sequential and parallel versions are compared. In all tests, including the one with checkpointing and restart, no non-matching values were found. It is also important to point out the impact of failures. The difference in the execution time has two causes: first, the program needs to save and load the buffers, which takes some time; and second, it re-executes tasks sequentially, which also takes some time. For the example case, only one task was re-executed, so no significant impact is seen. In comparison, considering a program where 8 tasks were restarted, the program will take $8 \times t$ time to re-execute all tasks (t being a task execution time), while if all those tasks were parallelizable, it could take up to t time to execute. In this small program, the time increase should not be a problem, but it could be more significative when various tasks need to be re-executed.

6.3.2 An Empirical Evaluation of Heartbeat Parameterization

This section evaluates the MPI side of OCFTL. The tests with OmpCluster give the evaluation about running OCFTL in OmpCluster programs, but does not give the evaluation of OCFTL running with stressed MPI programs[6]. This section aims to evaluate OCFTL under Intel MPI Benchmarks to define the limits of OCFTL based on those programs.

[5] In the example shown in Figure 17, 48 buffers were saved by the checkpoint. The host process saved all the buffers since there are no `PROXY` and `LINK` data maps.

[6] It is meant by `stressed` MPI programs, those where the program is overloaded by MPI message exchange. MPI benchmarks are examples of stressed MPI programs, since they overload the program with different MPI operations to establish the limits of a MPI distribution.

Intel MPI Benchmarks[7] is a set of MPI benchmarks intended to test MPI distributions of the MPI standard. The benchmark offers various subsets of benchmarks directed to specific MPI operations. These experiments leverage the MPI-1 subset that provides the evaluation of elementary MPI operations for communication. This subset has point-to-point (`P2P`, where two processes communicate with each other) and collective (`coll`, where a set of processes communicates with the entire set of processes) operations. For this experiment, the `ping-pong` application was selected as a `P2P` operation tester (since it is a `send-recv` operation), while the `allreduce` application was selected as a `coll` operation tester (since it is an all-to-all operation). One benchmark of each type was selected to make evaluation simpler, since the comparison is made with 5 different MPI distributions for different values of heartbeat parameters.

The configuration for these experiments uses 20 nodes with 24 processes each, totaling 480 processes. The benchmark configuration was set to test messages of length: `min`, meaning messages with a size of 0 bytes; `medium`; meaning messages with a size of 64 Kbytes; and `max`, meaning messages with a size of 4 Mbytes. The number of repetitions for each message size was the default of the tool (which is 1000 for 0 bytes messages, 640 for 64 Kbytes messages and 10 for 4 Mbytes messages). The `npmin` (minimum number of the participants in each benchmark) was also set to 480, meaning that for the `P2P` benchmark, there will be 240 pairs of processes, and all the 480 processes will do the MPI operation collectively for the `coll` benchmark. This number of processes evaluates the limits of heartbeat parameters and the scalability of OCFTL.

This experiment evaluated five MPI distributions: MPICH with UCX, MPICH without UCX, Open MPI with UCX, Open MPI without UCX, and ULFM. For the first four, OCFTL was the FT approach, while for the last, it used the ULFM's detector. The experiment tested different values of heartbeat periods[8]. At the same time, the timeout was set to 100 times higher than the period. Defining those parameters would evaluate the overhead of flooding the network with heartbeat messages. Tables 5, 6, 7, 8 and 9 (Appendix C) show the full results for MPICH with UCX, regular MPICH, Open MPI with UCX, regular Open MPI and ULFM respectively. The baseline in the tests represents the execution of the benchmarks without any fault tolerance support. Each test was executed 10 times and the confidence intervals are calculated using the Bootstrap method (EFRON; HASTIE, 2016) with 10000 iterations. Section E.4 (Appendix E) shows the modifications made in the Intel MPI Benchmarks to make it suitable for use with OCFTL and ULFM as well as the commands used to execute the tests.

Open MPI with UCX suffers from internal problems[9] and most of its tests could

[7] Available at <https://github.com/intel/mpi-benchmarks>

[8] For OCFTL, the tests defined the `heartbeat steptime` equal to the `heartbeat period`

[9] These are `segmentation fault` problems generated by the `mca_pml_ucx_recv_completion()` internal UCX function present in that distribution. This function is used, among others, by the `MPI_Mrecv`

not be completed. For that reason, this discussion will not include Open MPI with UCX. Some of the P2P benchmarks using ULFM reported false-positive failures (for the periods of $10, 20, 30$ and $100ms$). Finally, some samples of the first test (period of $10ms$) of the collective benchmark using MPICH with UCX also presented some internal errors related to the internal states of UCX. Every other tested completed successfully.

Figures 15, 16, 17, 18, 19 and 20 summarize the data of the tables. The plots represent the ratio between each benchmark and the associated baseline (same MPI distribution without FT). And, the confidence intervals are calculated using the Bootstrap method with 10000 iterations (EFRON; HASTIE, 2016).

Figure 15 – PingPong benchmark for messages with 0 bytes size

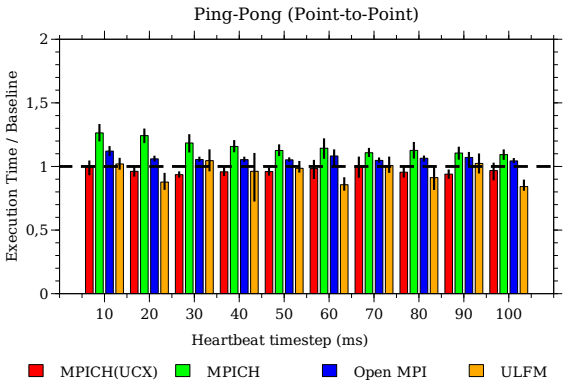
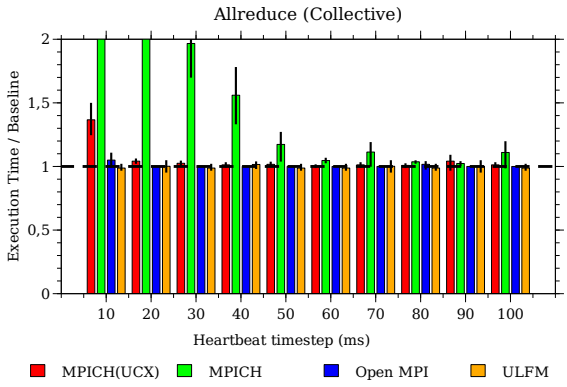


Figure 16 – AllReduce benchmark for messages with 0 bytes size



Note: Bars overlapping the plot limits in Figure 16 show a relation between tests (with `timestep` 10 and 20) and baseline of 7.33 and 3.17.

Figures 15 and 16 show that for messages with no data (0 bytes), the regular MPICH is the distribution where OCFTL has more overhead, especially for the collective benchmark. Excluding those and the test with `heartbeat timestep` of $10ms$ for MPICH with UCX, every other test showed overhead less than 10% when compared to the baseline.

Figures 17 and 18 show the overhead for messages with 64 Kbytes of data. For point-to-point benchmarks every distribution besides the regular MPICH showed no significant overhead. For collective benchmarks, all the distributions other than the regular OpenMPI showed significant overhead typically higher than 50%.

Finally, figures 19 and 20 show results for messages with 4 Mbytes of data. For the point-to-point benchmarks, only the regular MPICH showed significant overhead, with the maximum overhead of approximatelly 20%. For the collective benchmarks, with exception of the benchmark with `heartbeat timestep` of $10ms$ for MPICH with UCX, the other distributions showed up to 20% overhead. ULFM showed up to 40% overhead in some

function, which is heavily used by OCFTL in the procedure of checking if a message was received. This problem does not occur with the MPICH with UCX distribution, and do not occur with the non-UCX distributions.

Figure 17 – PingPong benchmark for messages with 64 Kbytes size

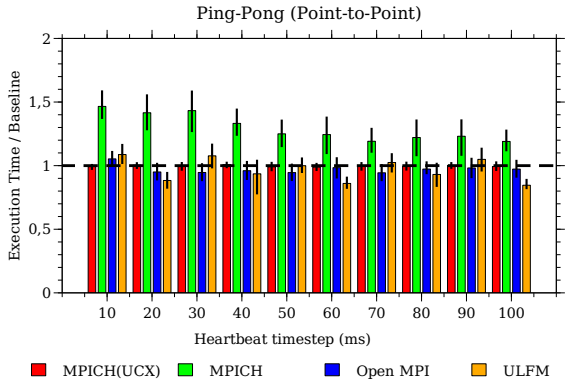
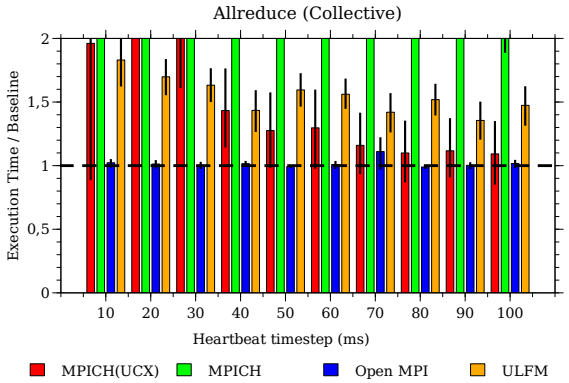


Figure 18 – AllReduce benchmark for messages with 64 Kbytes size



Note: Bars overlapping the plot limits in Figure 18 show a relation between tests and baseline ranging from 2.06 to 30.34.

Figure 19 – PingPong benchmark for messages with 4 Mbytes size

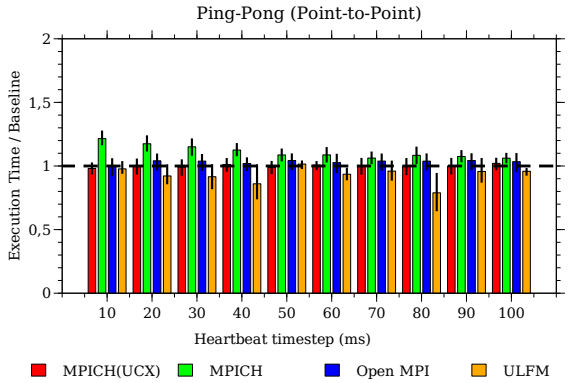
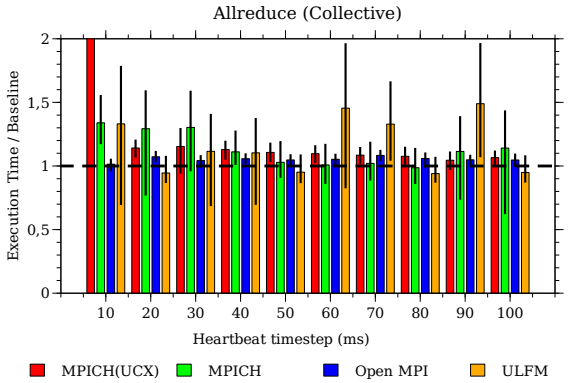


Figure 20 – AllReduce benchmark for messages with 4 Mbytes size



Note: Bars overlapping the plot limits in Figure 20 show a relation between tests with `timestep` 10 and baseline of 3.06.

benchmarks. In various benchmarks, the results show that ULFM performs better than the baseline, but Table 9 shows that the confidence intervals between the tested parameter and the baseline overlaps, which means that one can not consider those results as performance improvement.

It is important to highlight that the overhead should not be the only metric taken into account when chosing a library. For example, in the tests, OCFTL showed more overhead usings regular MPICH when compared to OCFTL using regular Open MPI. However, when looking at the time spent to execute the benchmarks through the tables in the Appendix C, the regular MPICH outperformed regular Open MPI in almost all of them, especially for collective operations with larger data exchanges. The same occurs when comparing ULFM, which showed almost no overhead, with OCFTL on top of regular MPICH.

Therefore, it is hard to point out an absolute value for the heartbeat properties since OCFTL can be used with different MPI distributions, and some distributions will perform better than others, depending on the case. Most of the tests showed that the heartbeat period could be $10ms$ with the timeout equal to $1s$, since it does not reported false-positive failures. These values are far lower than what this research expects to set as a default value for OmpCluster. At some point in the near future, OmpCluster will be running jobs that will take days or weeks to complete, so setting properties from ms to s (as the actual default values for OCFTL) should not make a huge difference.

Further tests were executed to define the `heartbeat timeout`. For OCFTL with regular MPICH and regular OpenMPI, the `heartbeat timestep` and `heartbeat period` value was set to $10ms$ and list of $10, 20, 40, 60, 80$ and 100 times the `heartbeat timestep` were used for the `heartbeat timeout`. The results showed that for regular MPICH, executions with the `timeout` equal or higher than $400ms$ false positives failures do not occur. To obtain the same behavior with Open MPI, `heartbeat timeout` values equal or higher than $600ms$ were needed. The `heartbeat timeout` only implies at the false-positive detection, it does not include significant extra overhead to the application, since it only checks if the heartbeat has timed out, it does not check for MPI messages.

Lastly, these tests were expanded to be executed with the UCX distribution variations since OmpCluster leverages UCX, but regular MPI (both MPICH and Open MPI) outperformed UCX. It is essential to point out that these executions with the regular variations simulated TCP over the InfiniBand, accelerating the speed of the library[10].

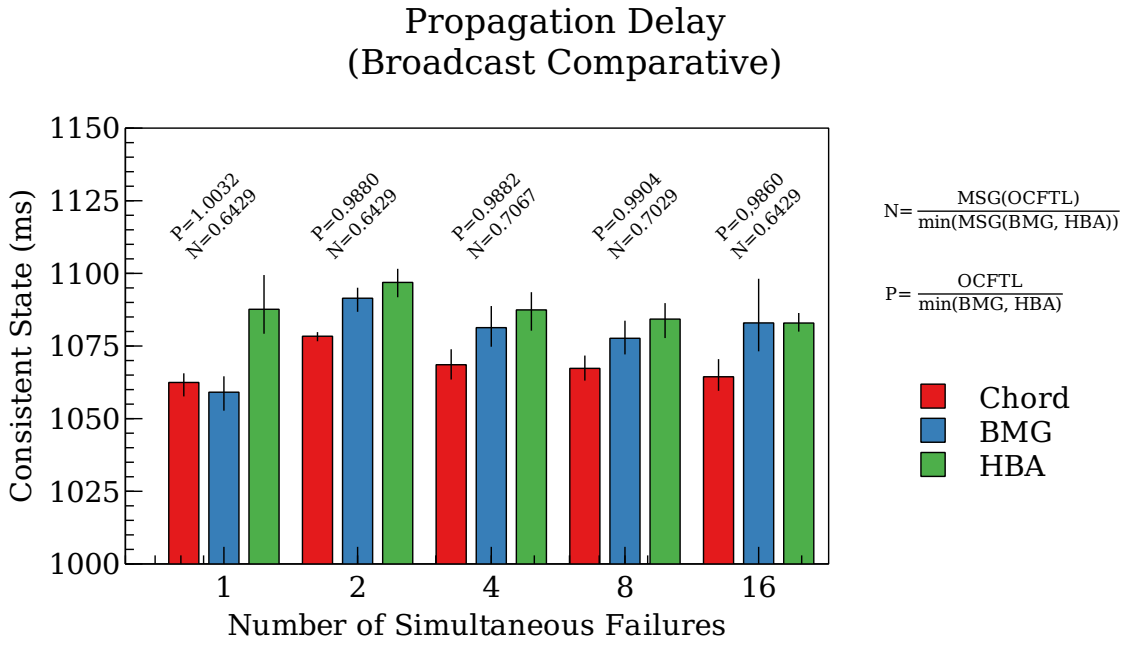
6.3.3 Internal Broadcast

To evaluate the internal broadcast, a comparison was made between the our approach, BMG and HBA algorithms (tested with ULFM – see Section 4.2). To compare the broadcasts, each broadcast was implemented inside OCFTL, each implementation (presented in the Appendix B) was made according to Algorithms 11, 12, and 13, for the Chord-like, BMG and HBA respectively.

The objective of this experiment is to evaluate the time to propagate a failure and the overhead they would impose on the application. So the test monitored the total time to propagate a set of simultaneous failures (each process remaining in the application to know about all the failures). The total number of messages each broadcast has used to complete the propagation was also counted. The configuration for this experiment uses a total of 480 processes distributed over 20 nodes and evaluates cases for 1, 2, 4, 8, and

[10] The runtime flag `-iface ib0` makes the regular MPICH to simulate TCP over the InfiniBand hardware. For Open MPI, the FAQ says that this is done automatically by the library (<https://www.open-mpi.org/faq/?category=tcp#tcp-auto-disable>)

Figure 21 – Time and total messages to achieve a consistent state through OCFTL, BMG and HBA broadcasts.



Note: Error bars represent the 95% confidence interval calculated using Bootstrap with 10000 iterations (EFRON; HASTIE, 2016).

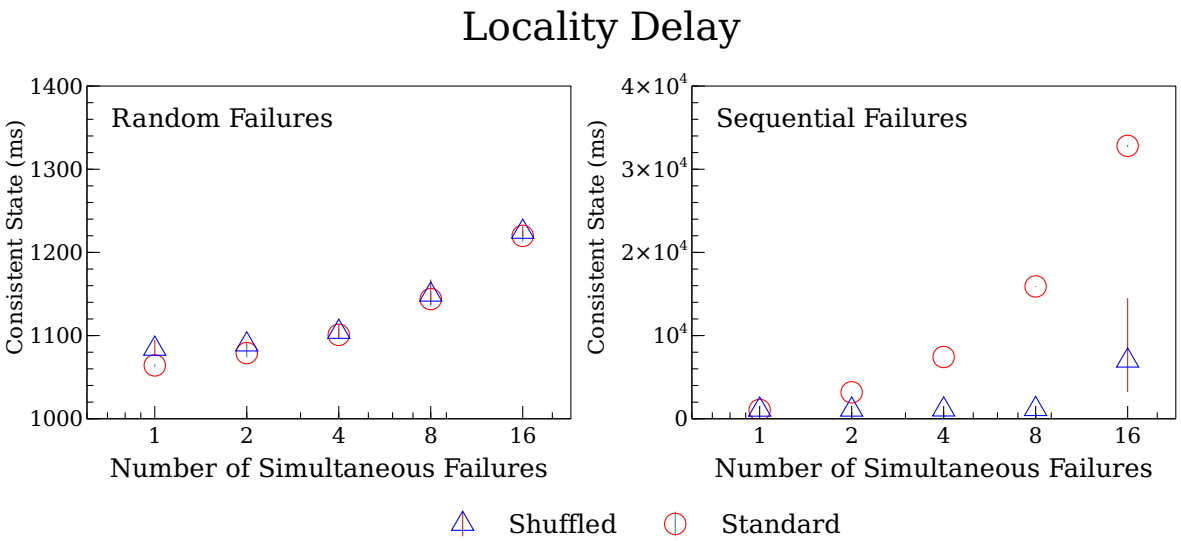
16 simultaneous failures. Each scenario consists of 10 samples. The full configuration for running these experiments is presented in Section E.5 in the Appendix E.

Figure 21 shows the comparison between the broadcasting algorithms. Although the figure shows the Chord-like is marginally better than BMG and HBA, the time taken to know about all failures (achieving a consistent state) is practically the same for all the broadcasting algorithms, as it is possible to see through the performance relation between the broadcast used by OCFTL and the best of BMG and HBA (`P` in the figure), where the values are close to 1.0 meaning the times are almost equal. The main difference is in the total number of messages (`N` in the figure). The proposed algorithm achieved the reduction of about 30% in most of the experiments (those represent received only broadcasts, messages lost during the execution and the broadcasts sent to already dead processes are not take into account). These results show that the Chord-like broadcast algorithm is a viable and better alternative to current state-of-the-art FT broadcasting algorithms.

6.3.4 Locality Problem

The rationale beyond the initial shuffling positions of the MPI rank in the ring to solve the locality problem was presented in Section 4.2. This experiment proposes the evaluation of the shuffling by simulating simultaneous failures (1, 2, 4, 8, and 16

Figure 22 – Comparision between standard and shuffled initial positions for random and sequential failures.



Note: Error bars represent the 95% confidence interval calculated using Bootstrap with 10000 iterations (EFRON; HASTIE, 2016).

simultaneous failures) for two cases: where the distribution of ranks would be sequential over the nodes (*e.g.,* 0 – 23 in the first node, 24 – 49 in the second node, and so on), which is the worst scenario if the positions are not shuffled and sequential failures happen (each dead process, except one of them, is observed by a dead process); and, where the distribution of process would be defined by a round-robin choice (*e.g.,* 0 in the first node, 1 in the second node, and so on), which is the best scenario if the positions are not shuffled and sequential failures happen (each dead process is observed by an alive process). For this experiment, 480 processes were used over 20 nodes and each scenario consists of 10 samples. The full configuration for running this experiments is presented in Section E.5 (Appendix E).

Figure 22 shows the comparison between two situations. The first case is for failures of pseudo-random MPI Rank. In this case, failures are defined by a round-robin algorithm that takes account of how many processes per node were allocated, in this case, 24, representing the best case for the standard algorithm (not shuffled). The second case is failures of sequentially distributed MPI ranks, which should be the worst case for the standard algorithm. The worst and best cases are theoretically defined, but as OCFTL includes a probability factor, this is not deterministic anymore. Figure 22 shows that for random failures, the standard and shuffle options take about the same time to achieve the consistent state, while for sequential failures, the standard option follows a linear function that increases the time as the number of simultaneous failure occur. The shuffle option is shown to be good for sequential failures, once the times to achieve a consistent state are less or equal to the standard option. In general, the shuffled positions shows to be good, having results even or better than the standard positions.

6.4 Limitations

The first observed limitation of OCFTL is in the procedure of repairing a communicator, discussed in Section 4.4.1. An inconsistent state of agreement between the processes occurs every time a request to repair a communicator is made, and OCFTL is in the middle of failure propagation. That will cause the processes to not agree with each other, and the procedure will not succeed. Solving this problem is not trivial. Since the request is made by one process, and the others receive this request through an internal broadcast, it is hard to determine its inconsistency. This topic will be tackled in the subsequent versions of OCFTL, in which data synchronization will be done using vector clocks.

Another limitation of OCFTL refers to the use of multiple communicators. Currently, the use of multiple communicators requires multiple OCFTL objects, which means that an MPI process that participates in more than one communicator would have the corresponding number of fault tolerance threads. Future studies will include handling multiple communicators by only one OCFTL object.

Concerning checkpointing, currently, OCFTL re-executes tasks sequentially since it has no control over the OpenMP threads that are stalled at the moment of the restart. One solution to this problem is scheduling those tasks and acting as the OpenMP runtime, redirecting the stalled OpenMP threads to execute normally the tasks marked to restart.

Finally, there is a limitation when OCFTL is used with OmpCluster. Currently, OmpCluster's runtime uses the process with rank 0 as the head process, which coordinates the execution and runs the core of OpenMP. OCFTL currently does not employ fault tolerance for the head process, which means that if it fails, the application needs to be restarted from the beginning. Providing FT for the head process is not trivial. Currently, OCFTL is not able to migrate or restart the core of OpenMP. That is a concern for future works, which might achieve it by replicating the head node and duplication of all messages exchanged by it, for example. Another problem with specific rank failure is the failure of MPI rank 1. This is the default "proxyable" process, which holds the proxy entries of any data mapping between the host and any worker. Currently, OmpCluster does not fix those tables when this specific process dies. The solution is simple, the map should be changed to the following process, but it is necessary to verify the correctness of the map after redefining the proxy entry of each map.

7 Conclusion

Combining FT and MPI is not a trivial task, and yet it is often done manually by MPI users. This document proposes a new library (OCFTL) that is an MPI implementation-independent approach. OCFTL tackles the problem of detecting and propagating failures, repairing invalid communicators, and proposes procedures to solve problems related to the behavior of synchronous and blocking operations. OCFTL was implemented to serve as the FT system for the OmpCluster project, which aims at easing the development of scientific applications for HPC, especially for researchers from non-computing areas.

The main objective of this research is to improve the resilience of MPI inside OmpCluster, providing it with a fault tolerance solution capable of detecting failures, surviving them, and completing the program execution correctly even if in the presence of failures. OCFTL offers: a heartbeat capable of detecting failures; an internal fault-tolerant broadcast algorithm with a reduced number of messages exchanged when compared to other broadcasting algorithms; checkpointing leveraging Veloc; and extra features, like wrappers to avoid deadlocking in MPI communications with dead processes, MPI communicators shrinkage, updated states of MPI communicators and processes, and notifications callbacks. This set of fault tolerance tools improves the resilience of MPI to the point where the program running OmpCluster can survive and complete with success even if in the presence of failures.

OCFTL is implemented in the core of OmpCluster, so, except for a few configurations, it will be transparent to the final user of OmpCluster, meaning the user will not have to add FT handling to their application source code. The library also is easier to maintain or update since it only employs MPI functions present in the MPI standard, which are well documented and heavily discussed by the MPI community. Its implementation is self-contained in a single `C++` class, so the maintaining and updating is easier since there are no scattered components. These characteristics fulfill the first specific objective of this research.

This work also explored, in some way, the use of more than one failure mitigation approach to resolve the impact of failures in the applications. In OCFTL, checkpointing was implemented, and a draft of how replication could be done was made, even if, at this time, OCFTL still lacks full support for alternative approaches to failure mitigation.. However, it improves the resilience by providing the shrinkage of the communicators (making it possible to execute collective operations in the presence of failures) and wrappers (to avoid the possible deadlock of MPI operations executed in the presence of failures). This characteristic fulfills the second specific objective of this work.

OCFTL is compatible with any standard-compliant MPI implementation. It was tested in different clusters (*Santos Dumont supercomputer* and *Sorgan* (ROSSO; FRANCESQUINI, 2021)) and was built with the exact same requirements of OmpCluster. Therefore it is compatible to all environments that support OmpCluster. Therefore, OCFTL fulfills the third specific objective of this research.

Results of this research include the development of techniques to workaround and avoid MPI deadlocks in the presence of failures. Experimental results show that for taskbench tests, OCFTL has a low impact on OmpCluster's performance. They also show that OCFTL portability could be used to overcome eventual overheads it may cause by the execution on faster MPI distributions. Additionally, the proposed algorithm for internal broadcast also provides a way to reduce message overloading while maintaining the robustness of an FT broadcasting algorithm. Finally, the technique of shuffling the initial position of the processes in the heartbeat ring was shown to be efficient, and better than the standard algorithm, while maintaining the simplicity.

OCFTL still has some limitations. First, it is restricted to one communicator. Second, the shrinkage procedure can not be completed if a failure occurs in the course of this procedure. Therefore, OCFTL cannot yet deal with failures in the host process, and for failures in the first worker, the data mapping will be broken. Finally, the checkpointing procedure currently can only re-execute tasks sequentially.

7.1 Publications

This research produced two articles published at the time of the production of this document. Both articles were published at the Escola Reginal de Alto Desempenho (Regional de São Paulo) - ERAD-SP, on the 2020 (XI) and 2021 (XII) editions. Both publications received the award of Best Paper in the Graduate Category of each edition.

The first article, published at the XI edition (ROSSO; FRANCESQUINI, 2020), discussed the subject of integrating fault tolerance and scheduling, testing choosing between using checkpointing or replication at the moment the application schedules the tasks based on the characteristics of the tasks. The tests were based on simulations using scientific workflows from Pegasus[1]. The results obtained show that choosing the fault tolerance approach on the fly according to the task's characteristic can reduce the total makespan of the application in the presence of failures. Although this article is more generic and is not directly related to OmpCluster, it gives insights into how fault tolerance approaches can be used in OmpCluster.

The second article, published in the XII edition (ROSSO; FRANCESQUINI, 2021),

[1] Available at <https://pegasus.isi.edu/>

discussed the subject of failure detection and propagation. This article is related to OCFTL and gave an initial evaluation about the heartbeat and propagation discussed in Section 4.2. The tests were similar to those presented in in Section 6.3.2, 6.3.3 and 6.3.4, but smaller. Results showed that the proposed algorithms are as good or better in some cases when compared to the state-of-the-art.

We also expect to publish one more article related to this research. This article would explore the results of OCFTL when used by OmpCluster to evaluate the correctness, the overhead of the library over OmpCluster, and other impacts OCFTL when used with OmpCluster.

Future Work

This research will be continued by the author as a doctoral thesis. So, most of the limitations and pending improvements will be tackled. This research will focus on the following topics:

- Extend fault tolerance to the outer domain of the fault tolerance in OmpCluster. This is especially related to failures in the host process. It is intended to use replication, so the replicas can control the OpenMP application if the host process fails.

- Extend the existing solutions (*e.g.,* checkpointing) to other devices. In the OmpCluster project, it is intended to have a second level of computation offloading, which means, for example, offloading to GPUs.

- Optimize the checkpointing, providing incremental checkpointing, which could be achieved by saving only the part of the used buffers. Moreover, support persistent memories, which are memories that have higher speed than HDs and SSDs.

- Provide replication as another fault tolerance approach for failure mitigation, which was already discussed and drafted by this work.

- Evaluate the inclusion of pro-active fault tolerance. The pro-active fault tolerance could help OmpCluster mitigate the impact of failures, taking actions before failure happens. Some status monitoring could be employed for this case. Like measuring CPU usage and temperature variation.

- Recover failed workers. Currently, the system is able to survive failures but not able to recover the failed processes. For this, OCFTL needs to spawn new processes or allocate more processes at the beginning of the program to be in reserve for failures.

Also, the limitations discussed in Section 6.4 are all topics to be tackled in the future, especially those concerning the tasks re-execution and failures on the host and first work device (the one that holds the proxy entries for all data maps).

Bibliography

AGBARIA, A.; FRIEDMAN, R. Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations. *Cluster Computing*, Springer, v. 6, n. 3, p. 227–236, 2003. Quoted in page 16.

ANSEL, J.; ARYA, K.; COOPERMAN, G. DMTCP: Transparent checkpointing for cluster computations and the desktop. In: IEEE. *2009 IEEE Intl. Symposium on Parallel & Distributed Processing (IPDPS'09)*. Rome, Italy, 2009. p. 1–12. Quoted in page 11.

AULWES, R. et al. Architecture of LA-MPI, a network-fault-tolerant mpi. In: *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings*. Santa Fe, NM, USA: IEEE, 2004. p. 15–. Quoted in page 16.

BATCHU, R. et al. MPI/FT: a model-based approach to low-overhead fault tolerant message-passing middleware. *Cluster Computing*, Springer, v. 7, n. 4, p. 303–315, 2004. Quoted in page 14.

BAUTISTA-GOMEZ, L. et al. FTI: High performance fault tolerance interface for hybrid systems. In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. New York, NY, USA: Association for Computing Machinery, 2011. (SC '11). ISBN 9781450307710. Disponível em: <https://doi.org/10.1145/2063384.2063427>. Quoted in page 11.

BLAND, W. et al. Post-failure recovery of MPI communication capability: Design and rationale. *The Intl. Journal of High Performance Computing Applications*, Sage Publications Sage UK: London, England, v. 27, n. 3, p. 244–254, 2013. Quoted 6 times in pages 2, 9, 13, 14, 23, and 31.

BOSILCA, G. et al. MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. In: *SC'02: Proc. of the 2002 ACM/IEEE Conf. on Supercomputing*. Baltimore, MD, USA: IEEE, 2002. p. 29–29. Quoted 3 times in pages 2, 8, and 9.

BOSILCA, G. et al. A failure detector for hpc platforms. *The Intl. Journal of High Performance Computing Applications*, SAGE Publications Sage UK: London, England, v. 32, n. 1, p. 139–158, 2018. Quoted 5 times in pages 10, 11, 14, 25, and 26.

BOUTEILLER, A. et al. MPICH-V project: A multiprotocol automatic fault-tolerant MPI. *The International Journal of High Performance Computing Applications*, Sage Publications Sage CA: Thousand Oaks, CA, v. 20, n. 3, p. 319–333, 2006. Quoted in page 16.

BRIDGES, P. et al. Users' guide to MPICH, a portable implementation of MPI. *Argonne National Laboratory*, v. 9700, p. 60439–4801, 1995. Quoted in page 16.

BUNTINAS, D. et al. Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI protocols. *Future Generation Computer Systems*, Elsevier, v. 24, n. 1, p. 73–84, 2008. Quoted in page 13.

CHAPMAN, B.; JOST, G.; PAS, R. V. D. *Using OpenMP: portable shared memory parallel programming*. London, England: MIT press, 2008. v. 10. Quoted in page 6.

DALY, J. T. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future generation computer systems*, Elsevier, v. 22, n. 3, p. 303–312, 2006. Quoted in page 12.

DI, S. et al. Characterizing and understanding hpc job failures over the 2k-day life of ibm bluegene/q system. In: IEEE. *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. Portland, OR, USA, 2019. p. 473–484. Quoted 2 times in pages 1 and 5.

DONGARRA, J.; HERAULT, T.; ROBERT, Y. Fault tolerance techniques for high-performance computing. In: *Fault-Tolerance Techniques for High-Performance Computing*. Cham: Springer, 2015. p. 3–85. Quoted in page 12.

DUBROVA, E. *Fault-tolerant design*. New York, NY: Springer, 2013. ISBN 978-1-4614-2112-2. Quoted in page 13.

EFRON, B.; HASTIE, T. *Computer age statistical inference*. Cambridge, United Kingdom: Cambridge University Press, 2016. v. 5. Quoted 10 times in pages 48, 51, 52, 55, 56, 77, 78, 79, 80, and 81.

EGWUTUOHA, I. P. et al. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing*, Springer, v. 65, n. 3, p. 1302–1326, 2013. Quoted 2 times in pages 1 and 5.

EL-ANSARY, S. et al. Efficient broadcast in structured p2p networks. In: SPRINGER. *International workshop on Peer-to-Peer systems*. [S.l.], 2003. p. 304–314. Quoted in page 26.

ELLIOTT, J. et al. Combining partial redundancy and checkpointing for hpc. In: IEEE. *2012 IEEE 32nd Intl. Conf. on Distributed Computing Systems*. Macau, China, 2012. p. 615–626. Quoted 2 times in pages 1 and 5.

FAGG, G. E.; DONGARRA, J. J. FT-MPI: Fault tolerant mpi, supporting dynamic applications in a dynamic world. In: DONGARRA, J.; KACSUK, P.; PODHORSZKI, N. (Ed.). *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000. p. 346–353. ISBN 978-3-540-45255-3. Quoted in page 16.

FORUM, M. P. I. *MPI: A Message-passing Interface Standard, Version 3.1 ; June 4, 2015*. High-Performance Computing Center Stuttgart, University of Stuttgart, 2015. Disponível em: <https://books.google.com.br/books?id=Fbv7jwEACAAJ>. Quoted 4 times in pages 8, 9, 23, and 36.

GABRIEL, E. et al. Open MPI: Goals, concept, and design of a next generation MPI implementation. In: SPRINGER. *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Berlin, Heidelberg, 2004. p. 97–104. Quoted in page 8.

GAMELL, M. et al. Exploring automatic, online failure recovery for scientific applications at extreme scales. In: IEEE. *SC'14: Proceedings of the Intl. Conf. for High Performance*

Computing, Networking, Storage and Analysis. New Orleans, LA, USA, 2014. p. 895–906. Quoted in page 15.

GEORGAKOUDIS, G.; GUO, L.; LAGUNA, I. Reinit++: Evaluating the performance of global-restart recovery methods for mpi fault tolerance. In: SADAYAPPAN, P. et al. (Ed.). *High Performance Computing*. Cham: Springer International Publishing, 2020. p. 536–554. ISBN 978-3-030-50743-5. Quoted in page 14.

HARGROVE, P. H.; DUELL, J. C. Berkeley lab checkpoint/restart (BLCR) for linux clusters. *Journal of Physics: Conference Series*, IOP Publishing, v. 46, p. 494–499, sep 2006. Disponível em: <https://doi.org/10.1088/1742-6596/46/1/067>. Quoted in page 11.

HASAN, M.; GORAYA, M. S. Fault tolerance in cloud computing environment: A systematic survey. *Computers in Industry*, Elsevier, v. 99, p. 156–172, 2018. Quoted 4 times in pages 1, 5, 9, and 10.

KOREN, I.; KRISHNA, C. *Fault-Tolerant Systems*. Elsevier Science, 2020. ISBN 9780128181065. Disponível em: <https://books.google.com.br/books?id=YrnjDwAAQBAJ>. Quoted 3 times in pages 5, 11, and 12.

LIANG, Y. et al. Bluegene/l failure analysis and prediction models. In: IEEE. *International Conference on Dependable Systems and Networks (DSN'06)*. Philadelphia, PA, USA, 2006. p. 425–434. Quoted in page 5.

LOUCA, S. et al. MPI-FT: Portable fault tolerance scheme for MPI. *Parallel Processing Letters*, World Scientific, v. 10, n. 04, p. 371–382, 2000. Quoted in page 15.

LU, C. D. *Scalable diskless checkpointing for large parallel systems*. Urbana, Illinois, 2005. Quoted in page 15.

MIKAILOV, M. et al. Scaling bioinformatics applications on HPC. *BMC bioinformatics*, BioMed Central, v. 18, n. 14, p. 163–169, 2017. Quoted in page 1.

MOODY, A. et al. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In: IEEE. *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. New Orleans, LA, USA, 2010. p. 1–11. Quoted in page 11.

NAEIMI, V. et al. Geophysical parameters retrieval from sentinel-1 sar data: A case study for high performance computing at eodc. In: *Proceedings of the 24th High Performance Computing Symposium*. San Diego, CA, USA: Society for Computer Simulation International, 2016. (HPC '16). ISBN 9781510823181. Disponível em: <https://doi.org/10.22360/SpringSim.2016.HPC.026>. Quoted in page 1.

NICOLAE, B. et al. Veloc: Towards high performance adaptive asynchronous checkpointing at large scale. In: IEEE. *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Rio de Janeiro, Brazil, 2019. p. 911–920. Quoted 3 times in pages 11, 12, and 28.

OLINER, A.; STEARLEY, J. What supercomputers say: A study of five system logs. In: IEEE. *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. Edinburgh, UK, 2007. p. 575–584. Quoted in page 5.

PANDA, D. K. et al. The MVAPICH project: Transforming research into high-performance MPI library for HPC community. *Journal of Computational Science*, Elsevier, v. 52, p. 101208, 2021. Quoted in page 8.

PATTERSON, D.; HENNESSY, J. *Computer Organization and Design ARM, Interactive Edition: The Hardware Software Interface*. Elsevier Science, 2016. (The Morgan Kaufmann Series in Computer Architecture and Design). ISBN 9780128018354. Disponível em: <https://books.google.com.br/books?id=Pz-XCgAAQBAJ>. Quoted in page 1.

PLANK, J. S. *An overview of checkpointing in uniprocessor and distributed systems, focusing on implementation and performance*. Knoxville, TN, 1997. Quoted in page 11.

RODRÍGUEZ, G. et al. CPPC: a compiler-assisted tool for portable checkpointing of message-passing applications. *Concurrency and Computation: Practice and Experience*, Wiley Online Library, v. 22, n. 6, p. 749–766, 2010. Quoted in page 11.

ROJAS, E. et al. Analyzing a five-year failure record of a leadership-class supercomputer. In: IEEE. *2019 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. Campo Grande, Brazil, 2019. p. 196–203. Quoted 2 times in pages 1 and 5.

ROSSO, P. H.; FRANCESQUINI, E. A fault tolerant scheduling model for directed acyclic graphs in cloud. In: SBC. *Anais Estendidos da XI Escola Regional de Alto Desempenho de São Paulo. Paper accepted for publication*. São Paulo, Brazil, 2020. Quoted 2 times in pages 3 and 60.

ROSSO, P. H.; FRANCESQUINI, E. Improved failure detection and propagation mechanisms for MPI. In: SBC. *Anais Estendidos da XII Escola Regional de Alto Desempenho de São Paulo. Accepted for publication.* São Paulo, Brazil, 2021. Disponível em: <http://cradsp.sbc.org.br/eradsp/2021/artigos/s1.2.pdf>. Quoted 2 times in pages 3 and 60.

SCHROEDER, B.; GIBSON, G. A. A large-scale study of failures in high-performance computing systems. *IEEE transactions on Dependable and Secure Computing*, Ieee, v. 7, n. 4, p. 337–350, 2009. Quoted in page 5.

SILVA, J. A. da; REBELLO, V. E. F. A hybrid fault tolerance scheme for easygrid mpi applications. In: *Proceedings of the 9th International Workshop on Middleware for Grids, Clouds and e-Science*. New York, NY, USA: Association for Computing Machinery, 2011. (MGC '11). ISBN 9781450310680. Disponível em: <https://doi.org/10.1145/2089002.2089006>. Quoted in page 15.

SLAUGHTER, E. et al. Task bench: A parameterized benchmark for evaluating parallel runtime performance. In: IEEE. *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. Atlanta, GA, USA, 2020. p. 1–15. Quoted in page 48.

STELLNER, G. CoCheck: Checkpointing and process migration for MPI. In: IEEE. *Proceedings of International Conference on Parallel Processing*. Honolulu, HI, USA, 1996. p. 526–531. Quoted in page 16.

STOICA, I. et al. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, ACM New York, NY, USA, v. 31, n. 4, p. 149–160, 2001. Quoted 2 times in pages 14 and 26.

SULTANA, N. et al. Mpi stages: Checkpointing mpi state for bulk synchronous applications. In: *Proceedings of the 25th European MPI Users' Group Meeting*. New York, NY, USA: Association for Computing Machinery, 2018. (EuroMPI'18). ISBN 9781450364928. Disponível em: <https://doi.org/10.1145/3236367.3236385>. Quoted in page 16.

TERANISHI, K.; HEROUX, M. A. Toward local failure local recovery resilience model using mpi-ulfm. In: *Proceedings of the 21st European MPI Users' Group Meeting*. New York, NY, USA: Association for Computing Machinery, 2014. (EuroMPI/ASIA '14), p. 51–56. ISBN 9781450328753. Disponível em: <https://doi.org/10.1145/2642769.2642774>. Quoted in page 15.

TIWARI, D.; GUPTA, S.; VAZHKUDAI, S. S. Lazy checkpointing: Exploiting temporal locality in failures to mitigate checkpointing overheads on extreme-scale systems. In: IEEE. *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. Atlanta, GA, USA, 2014. p. 25–36. Quoted in page 12.

YOUNG, J. W. A first order approximation to the optimum checkpoint interval. *Communications of the ACM*, ACM New York, NY, USA, v. 17, n. 9, p. 530–531, 1974. Quoted 2 times in pages 12 and 28.

YUAN, Y. et al. Job failures in high performance computing systems: A large-scale empirical study. *Computers & Mathematics with Applications*, Elsevier, v. 63, n. 2, p. 365–377, 2012. Quoted in page 5.

ZHONG, D. et al. Runtime level failure detection and propagation in hpc systems. In: *Proceedings of the 26th European MPI Users' Group Meeting*. New York, NY, USA: Association for Computing Machinery, 2019. (EuroMPI '19). ISBN 9781450371759. Disponível em: <https://doi.org/10.1145/3343211.3343225>. Quoted 5 times in pages 9, 10, 11, 14, and 26.

Appendix

APPENDIX A – Fault Tolerance MPI Wrappers Implementation

Algorithm 5 – OCFTL's costum MPI function to `MPI_Itest`.

```
int MPI_Itest(MPI_Request *request, int *flag, MPI_Status *status, int proc) {
  if (disable_ft || ft_handler == nullptr || proc == ft::FT_TEST_LEGACY) {
    return __real_MPI_Test(request, flag, status);
  }
  assertm((proc >= ft::FT_MPI_COLLECTIVE) && (proc < ft_handler->getSize()),
          "Waiting for request with invalid rank participating.");

  if (proc == ft::FT_MPI_COLLECTIVE) {
    // If it is a collective call
    if (ft_handler->getCommState() != ft::CommState::VALID) {
      MPI_Request_free(request);
      return ft::FT_ERROR;
    }
  } else {
    // If it is a point-to-point call
    if (ft_handler->getProcessState(proc) != ft::ProcessState::ALIVE) {
      MPI_Request_free(request);
      return ft::FT_ERROR;
    }
  }

  // If there is no errors associated to the processes. return the real call
  __real_MPI_Test(request, flag, status);
  return ft::FT_SUCCESS;
}
```

Algorithm 6 – OCFTL's wrapper function to `MPI_Free_comm`.

```
int __wrap_MPI_Comm_free(MPI_Comm *comm) {
  if (disable_ft || ft_handler == nullptr) {
    return __real_MPI_Comm_free(comm);
  }
  if (ft_handler->getCommState() == ft::CommState::VALID) {
    __real_MPI_Comm_free(comm);
    return ft::FT_SUCCESS;
  } else {
    if (ft_handler->getRank() == 0)
      FTDEBUG(
          "[Rank %d FT] - Could not free communicator with a failed process\n",
          ft_handler->getRank());
    return ft::FT_ERROR;
  }
}
```

Algorithm 7 – OCFTL's wrapper function to `MPI_Barrier`.

```
int __wrap_MPI_Barrier(MPI_Comm comm) {
  if (disable_ft || ft_handler == nullptr) {
    return __real_MPI_Barrier(comm);
  };
  int result = ft::FT_ERROR;
  MPI_Request barrier_req;
  if (ft_handler->getCommState() == ft::CommState::VALID) {
    MPI_Ibarrier(comm, &barrier_req);
    result = MPI_Iwait(&barrier_req, MPI_STATUS_IGNORE, ft::FT_MPI_COLLECTIVE);
  }
  // If comm is invalid or previouse call to MPI_Ibarrier failed
  while (result == ft::FT_ERROR) {
    // If comm is not valid, let us return an error, but try to fix the comm and
    // call MPI_Ibarrier
    comm = ft_handler->requestCommRepair();
    assertm(comm != MPI_COMM_NULL,
            "MPI_Ibarrier was called within an invalid MPI Comm. FT library "
            "could not the reestablish comm.");
    MPI_Ibarrier(comm, &barrier_req);
    result = MPI_Iwait(&barrier_req, MPI_STATUS_IGNORE, ft::FT_MPI_COLLECTIVE);
    if (result == ft::FT_SUCCESS) {
      result = ft::FT_SUCCESS_NEW_COMM;
      if (ft_handler->getRank() == 0)
        FTDEBUG("[Rank %d FT] - MPI_Barrier was called within an invalid MPI "
                "Comm. Executing in a repaired comm\n",
                ft_handler->getRank());
    }
  }
  return result;
}
```

Algorithm 8 – OCFTL's wrapper function to `MPI_Mprobe`.

```
int __wrap_MPI_Mprobe(int source, int tag, MPI_Comm comm, MPI_Message *message,
                      MPI_Status *status) {
  if (disable_ft || ft_handler == nullptr) {
    return __real_MPI_Mprobe(source, tag, comm, message, status);
  }
  int probe_flag = 0;
  MPI_Improbe(source, tag, comm, &probe_flag, message, status);

  while (!probe_flag) {
    if (ft_handler->getProcessState(source) != ft::ProcessState::ALIVE) {
      return ft::FT_ERROR;
    }
    MPI_Improbe(source, tag, comm, &probe_flag, message, status);
  }
  return ft::FT_SUCCESS;
}
```

Algorithm 9 – OCFTL's wrapper function to `MPI_Send`.

```
int __wrap_MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest,
                    int tag, MPI_Comm comm) {
  if (disable_ft || ft_handler == nullptr) {
    // If FT is disabled
    return __real_MPI_Send(buf, count, datatype, dest, tag, comm);
  }
  // Replaces regular Send by Isend
  MPI_Request w_send_request;
  MPI_Isend(buf, count, datatype, dest, tag, comm, &w_send_request);
  int result = MPI_Iwait(&w_send_request, MPI_STATUS_IGNORE, dest);
  if (result == ft::FT_ERROR) {
    MPI_Request_free(&w_send_request);
  }
  return result;
}
```

Algorithm 10 – OCFTL's wrapper function to `MPI_Recv`.

```
int __wrap_MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source,
                    int tag, MPI_Comm comm, MPI_Status *status) {
  if (disable_ft || ft_handler == nullptr) {
    // If FT is disabled
    return __real_MPI_Recv(buf, count, datatype, source, tag, comm, status);
  }
  MPI_Message msg;
  // Keep probing until the message is received and stored in msg
  int result = MPI_Mprobe(source, tag, comm, &msg, MPI_STATUS_IGNORE);
  if (result == ft::FT_SUCCESS) {
    // Get message and buf contents from msg
    MPI_Mrecv(buf, count, datatype, &msg, MPI_STATUS_IGNORE);
  }
  return result;
}
```

APPENDIX B – Implementation of Chord-like, BMG and HBA broadcasting algorithms

Algorithm 11 – Implementation for the Chord-like broadcasting algorithm.

```
// neighbors is a vector that represents the heartbeat ring
for (std::size_t i = 1; i < neighbors.size(); i *= 2) {
  int index = (n_pos + i) % neighbors.size();
  if (neighbors[index] != rank) {
    MPI_Isend(bc_message, 3, MPI_INT, neighbors[index], TAG_HB_BCAST,
              hb_comm, &send_request);
    MPI_Request_free(&send_request);
  }
}
```

Algorithm 12 – Implementation for the BMG broadcasting algorithm.

```
// neighbors is a vector that represents the heartbeat ring
for (std::size_t i = 1; i < neighbors.size(); i *= 2) {
  // Successor
  int index_s = (n_pos + i) % neighbors.size();
  if (neighbors[index_s] != rank) {
    MPI_Isend(bc_message, 3, MPI_INT, neighbors[index_s], TAG_HB_BCAST,
              hb_comm, &send_request);
    MPI_Request_free(&send_request);
  }
  // Predecessor
  int index_p = (n_pos - i) % neighbors.size();
  index_p = index_p < 0 ? neighbors.size() + index_p : index_p;
  if (neighbors[index_p] != rank) {
    MPI_Isend(bc_message, 3, MPI_INT, neighbors[index_p], TAG_HB_BCAST,
              hb_comm, &send_request);
    MPI_Request_free(&send_request);
  }
}
```

Algorithm 13 – Implementation for the HBA broadcasting algorithm.

```
// neighbors is a vector that represents the heartbeat ring
int k = static_cast<int>(std::floor(std::log2(neighbors.size())));
for (std::size_t i = 0; i < k; i++) {
  // Successor
  int index_s = (n_pos + i) % neighbors.size();
  if (neighbors[index_s] != rank) {
    MPI_Isend(bc_message, 3, MPI_INT, neighbors[index_s], TAG_HB_BCAST,
              hb_comm, &send_request);
    MPI_Request_free(&send_request);
  }
  // Predecessor
  int index_p = (n_pos - i);
  index_p = index_p < 0 ? neighbors.size() + index_p : index_p;
  if (neighbors[index_p] != rank) {
    MPI_Isend(bc_message, 3, MPI_INT, neighbors[index_p], TAG_HB_BCAST,
              hb_comm, &send_request);
    MPI_Request_free(&send_request);
  }
}
```

APPENDIX C – Intel MPIBench Results

Table 5 – Results for MPICH + UCX. First column represents the values of the heartbeat timestep. The heartbeat period was equal to the timestep and the timeout was set 100 times higher. The three other columns show the value of the benchmark for different message sizes.

Collective (allreduce benchmark)

HB	0 bytes		64 Kbytes		4 Mbytes	
TimeStep	**Avg (us)**	**95% CI**	**Avg (us)**	**95% CI**	**Avg (us)**	**95% CI**
10	0.33	[0.30, 0.36]	1262.88	[865.30, 1790.82]	204668.62	[188968.36, 220843.53]
20	0.25	[0.25, 0.25]	4432.03	[3759.82, 5076.37]	75817.62	[73878.96, 78055.52]
30	0.25	[0.24, 0.25]	1620.25	[1450.49, 1906.78]	76607.59	[69024.59, 89522.50]
40	0.24	[0.24, 0.25]	981.79	[931.93, 1036.40]	75019.84	[73645.35, 76485.05]
50	0.24	[0.24, 0.25]	874.23	[819.23, 951.95]	73505.86	[70235.02, 77175.05]
60	0.24	[0.24, 0.24]	888.86	[808.89, 1005.01]	72846.35	[71184.27, 74536.30]
70	0.24	[0.24, 0.25]	794.62	[758.16, 838.16]	72084.98	[69930.70, 74090.92]
80	0.24	[0.24, 0.25]	753.48	[725.53, 784.00]	71472.52	[69052.43, 73796.55]
90	0.25	[0.24, 0.27]	765.19	[707.43, 826.65]	69430.36	[67589.24, 71513.27]
100	0.24	[0.24, 0.25]	748.5	[687.56, 833.77]	70781.45	[69201.64, 72437.55]
Baseline	0.24	[0.24, 0.24]	828.67	[587.21, 1144.50]	66769.5	[63673.20, 69344.30]

Point-to-point (pingpong benchmark)

HB	0 bytes		64 Kbytes		4 Mbytes	
TimeStep	**Avg (us)**	**95% CI**	**Avg (us)**	**95% CI**	**Avg (us)**	**95% CI**
10	0.43	[0.42, 0.46]	52.24	[52.05, 52.45]	3322.44	[3211.95, 3402.30]
20	0.42	[0.41, 0.43]	52.86	[52.67, 53.16]	3379.62	[3254.89, 3466.25]
30	0.41	[0.40, 0.41]	52.28	[51.16, 52.89]	3359.87	[3240.33, 3465.47]
40	0.42	[0.41, 0.43]	53.08	[52.91, 53.32]	3431.36	[3348.90, 3482.41]
50	0.42	[0.41, 0.43]	52.17	[50.90, 52.89]	3351.42	[3225.16, 3473.48]
60	0.43	[0.41, 0.46]	52.03	[50.97, 52.81]	3415.46	[3310.31, 3480.73]
70	0.44	[0.41, 0.47]	52.31	[51.26, 53.12]	3389.79	[3253.80, 3487.13]
80	0.42	[0.40, 0.44]	52.44	[51.34, 53.14]	3381.82	[3247.99, 3481.96]
90	0.41	[0.40, 0.42]	52.76	[52.16, 53.25]	3389.46	[3255.43, 3485.06]
100	0.42	[0.41, 0.45]	52.29	[50.98, 53.01]	3461.42	[3428.59, 3484.64]
Baseline	0.44	[0.43, 0.45]	52.68	[51.82, 53.30]	3401.88	[3261.68, 3498.08]

Notes: Confidence intervals calculated using Bootstrap method with B=10000 (EFRON; HASTIE, 2016).

Table 6 – Results for MPICH. First column represents the values of the heartbeat timestep. The heartbeat period was equal to the timestep and the timeout was set 100 times higher. The three other columns show the value of the benchmark for different message sizes.

Collective (allreduce benchmark)

HB	0 bytes		64 Kbytes		4 Mbytes	
TimeStep	**Avg (us)**	**95% CI**	**Avg (us)**	**95% CI**	**Avg (us)**	**95% CI**
10	2.20	[2.11, 2.28]	27760.04	[27402.54, 28096.60]	52749.66	[51200.09, 55305.32]
20	0.95	[0.87, 1.03]	21610.24	[21267.92, 21932.17]	50928.56	[44011.85, 63328.82]
30	0.59	[0.52, 0.67]	17437.24	[16606.92, 18228.09]	51332.37	[43592.86, 60333.04]
40	0.47	[0.40, 0.53]	11820.09	[11207.82, 12400.70]	43767.17	[41193.04, 47146.26]
50	0.35	[0.32, 0.39]	7331.00	[6981.97, 7684.85]	40519.25	[39250.22, 41876.87]
60	0.31	[0.31, 0.32]	5478.90	[4980.91, 5952.76]	39709.90	[38237.29, 41389.11]
70	0.33	[0.31, 0.37]	3826.24	[3337.09, 4330.74]	40184.42	[38660.34, 41948.09]
80	0.31	[0.31, 0.31]	3239.07	[2735.64, 3766.80]	38848.18	[37802.44, 39940.02]
90	0.31	[0.30, 0.31]	2411.54	[1968.00, 3000.04]	43888.29	[36573.20, 56888.01]
100	0.33	[0.31, 0.37]	1888.59	[1759.97, 2019.38]	44941.63	[37340.96, 57920.79]
Baseline	0.30	[0.30, 0.30]	914.99	[883.80, 945.42]	43093.90	[35446.23, 55315.63]

Point-to-point (pingpong benchmark)

HB	0 bytes		64 Kbytes		4 Mbytes	
TimeStep	**Avg (us)**	**95% CI**	**Avg (us)**	**95% CI**	**Avg (us)**	**95% CI**
10	0.80	[0.76, 0.83]	31.73	[30.64, 32.90]	3005.70	[2948.76, 3072.03]
20	0.78	[0.77, 0.80]	30.64	[28.79, 32.40]	2905.38	[2795.92, 3003.84]
30	0.75	[0.72, 0.78]	31.01	[29.48, 32.66]	2846.11	[2788.24, 2913.84]
40	0.73	[0.72, 0.74]	28.84	[28.00, 29.63]	2783.72	[2748.16, 2820.57]
50	0.71	[0.70, 0.72]	27.06	[25.69, 28.43]	2687.52	[2628.29, 2748.91]
60	0.72	[0.69, 0.76]	26.93	[25.62, 28.38]	2688.93	[2639.15, 2742.39]
70	0.70	[0.68, 0.72]	25.79	[24.49, 27.11]	2628.14	[2565.94, 2695.57]
80	0.71	[0.69, 0.73]	26.43	[25.06, 27.86]	2680.83	[2611.87, 2750.90]
90	0.70	[0.68, 0.71]	26.65	[25.03, 28.51]	2658.18	[2603.94, 2716.85]
100	0.69	[0.68, 0.70]	25.76	[24.61, 26.83]	2625.86	[2568.07, 2684.82]
Baseline	0.63	[0.61, 0.66]	22.09	[20.21, 24.23]	2483.75	[2386.37, 2590.18]

Notes: Confidence intervals calculated using Bootstrap method with B=10000 (EFRON; HASTIE, 2016).

Table 7 – Results for Open MPI + UCX. First column represents the values of the heartbeat timestep. The heartbeat period was equal to the timestep and the timeout was set 100 times higher. The three other columns show the value of the benchmark for different message sizes.

Collective (allreduce benchmark)

HB	0 bytes		64 Kbytes		4 Mbytes	
TimeStep	**Avg (us)**	**95% CI**	**Avg (us)**	**95% CI**	**Avg (us)**	**95% CI**
10	0.05	[0.05, 0.05]	883.68	[883.68, 883.68]	-	-
20	0.05	[0.05, 0.05]	-	-	-	-
30	0.05	[0.05, 0.05]	472.00	[472.00, 472.00]	-	-
40	0.05	[0.05, 0.05]	1332.75	[1332.75, 1332.75]	-	-
50	0.05	[0.05, 0.05]	-	-	-	-
60	0.05	[0.05, 0.05]	604.82	[509.25, 700.38]	-	-
70	0.05	[0.05, 0.05]	-	-	-	-
80	0.05	[0.05, 0.05]	-	-	-	-
90	0.05	[0.05, 0.05]	-	-	-	-
100	0.05	[0.05, 0.05]	381.70	[381.70, 381.70]	33852.86	[33852.86, 33852.86]
Baseline	0.05	[0.05, 0.05]	399.37	[379.31, 425.72]	33398.40	[33255.59, 33610.15]

Point-to-point (pingpong benchmark)

HB	0 bytes		64 Kbytes		4 Mbytes	
TimeStep	**Avg (us)**	**95% CI**	**Avg (us)**	**95% CI**	**Avg (us)**	**95% CI**
10	-	-	-	-	-	-
20	-	-	-	-	-	-
30	-	-	-	-	-	-
40	-	-	-	-	-	-
50	-	-	-	-	-	-
60	0.53	[0.52, 0.54]	-	-	-	-
70	0.53	[0.53, 0.54]	100.99	[100.73, 101.25]	-	-
80	0.53	[0.53, 0.53]	101.49	[101.29, 101.71]	-	-
90	-	-	-	-	-	-
100	0.53	[0.52, 0.53]	-	-	-	-
Baseline	0.53	[0.52, 0.53]	102.86	[102.85, 102.88]	2432.18	[2426.31, 2439.22]

Notes: Confidence intervals calculated using Bootstrap method with B=10000 (EFRON; HASTIE, 2016). Cells without values means that the benchmark failed due to an internal UCX problem in Open MPI implementation (See Section 6.3.2)

Table 8 – Results for Open MPI. First column represents the values of the heartbeat timestep. The heartbeat period was equal to the timestep and the timeout was set 100 times higher. The three other columns show the value of the benchmark for different message sizes.

Collective (allreduce benchmark)

HB	0 bytes		64 Kbytes		4 Mbytes	
TimeStep	**Avg (us)**	**95% CI**	**Avg (us)**	**95% CI**	**Avg (us)**	**95% CI**
10	0.06	[0.06, 0.07]	2884.09	[2819.94, 2968.05]	284888.06	[278535.19, 291792.66]
20	0.06	[0.06, 0.06]	2856.18	[2795.31, 2935.17]	301115.31	[292471.75, 309861.00]
30	0.06	[0.06, 0.06]	2827.52	[2780.63, 2878.73]	292687.72	[286011.81, 299279.84]
40	0.06	[0.06, 0.06]	2863.80	[2822.40, 2918.97]	296865.19	[290263.72, 303168.41]
50	0.06	[0.06, 0.06]	2798.88	[2781.79, 2813.63]	294105.81	[286383.91, 302833.06]
60	0.06	[0.06, 0.06]	2844.16	[2791.27, 2915.37]	295667.50	[289678.31, 301543.84]
70	0.06	[0.06, 0.06]	3132.21	[2839.24, 3505.13]	304192.75	[297085.56, 311414.56]
80	0.06	[0.06, 0.06]	2790.31	[2775.98, 2804.87]	297292.56	[289374.94, 306874.31]
90	0.06	[0.06, 0.06]	2827.02	[2784.57, 2883.77]	294403.44	[289497.28, 299831.69]
100	0.06	[0.06, 0.06]	2867.30	[2811.93, 2939.35]	294058.94	[286951.31, 301824.28]
Baseline	0.06	[0.06, 0.06]	2820.44	[2806.90, 2835.48]	281245.53	[273970.12, 287504.94]

Point-to-point (pingpong benchmark)

HB	0 bytes		64 Kbytes		4 Mbytes	
TimeStep	**Avg (us)**	**95% CI**	**Avg (us)**	**95% CI**	**Avg (us)**	**95% CI**
10	2.51	[2.43, 2.60]	10.72	[10.14, 11.38]	2798.36	[2714.54, 2873.17]
20	2.37	[2.36, 2.39]	9.68	[9.64, 9.71]	2922.66	[2912.06, 2933.14]
30	2.36	[2.35, 2.37]	9.64	[9.61, 9.66]	2915.71	[2902.50, 2928.18]
40	2.36	[2.35, 2.37]	9.78	[9.62, 10.05]	2863.52	[2762.82, 2934.99]
50	2.35	[2.35, 2.36]	9.63	[9.61, 9.64]	2928.15	[2918.64, 2939.88]
60	2.42	[2.34, 2.56]	10.02	[9.64, 10.60]	2882.80	[2787.47, 2946.47]
70	2.34	[2.34, 2.35]	9.61	[9.59, 9.64]	2916.79	[2870.03, 2947.26]
80	2.38	[2.34, 2.45]	9.91	[9.62, 10.33]	2915.61	[2870.66, 2944.25]
90	2.40	[2.34, 2.50]	10.00	[9.62, 10.53]	2928.54	[2905.08, 2948.28]
100	2.33	[2.33, 2.34]	9.91	[9.63, 10.31]	2903.18	[2829.22, 2946.80]
Baseline	2.24	[2.22, 2.27]	10.30	[9.59, 11.07]	2827.86	[2663.85, 2941.29]

Notes: Confidence intervals calculated using Bootstrap method with B=10000 (EFRON; HASTIE, 2016).

Table 9 – Results for ULFM. First column represents the values of the heartbeat timestep. The heartbeat period was equal to the timestep and the timeout was set 100 times higher. The three other columns show the value of the benchmark for different message sizes.

Collective (allreduce benchmark)

HB	0 bytes		64 Kbytes		4 Mbytes	
TimeStep	**Avg (us)**	**95% CI**	**Avg (us)**	**95% CI**	**Avg (us)**	**95% CI**
10	0.07	[0.07, 0.07]	14297.21	[13763.95, 14962.67]	109771.94	[77852.02, 156242.72]
20	0.07	[0.07, 0.07]	13265.29	[12738.71, 13604.62]	77929.30	[77474.73, 78678.44]
30	0.07	[0.07, 0.07]	12753.40	[12575.05, 12933.02]	91985.09	[77562.16, 119740.33]
40	0.07	[0.07, 0.07]	11204.03	[10440.45, 11951.01]	90990.47	[77406.55, 116492.10]
50	0.07	[0.07, 0.07]	12459.55	[12290.93, 12647.85]	78466.07	[77481.96, 79973.49]
60	0.07	[0.07, 0.07]	12196.63	[12017.29, 12397.97]	119983.02	[77811.94, 162522.72]
70	0.07	[0.07, 0.07]	11091.47	[10338.63, 11812.62]	109623.98	[77591.31, 156432.28]
80	0.07	[0.07, 0.07]	11864.26	[11695.04, 12043.56]	77560.58	[77387.98, 77755.94]
90	0.07	[0.07, 0.07]	10587.95	[9760.86, 11404.33]	122851.02	[80845.29, 166670.03]
100	0.07	[0.07, 0.07]	11513.36	[11033.86, 11861.66]	78220.79	[77704.94, 78976.55]
Baseline	0.07	[0.07, 0.07]	7925.70	[7346.41, 8508.13]	89538.92	[77349.63, 113777.73]

Point-to-point (pingpong benchmark)

HB	0 bytes		64 Kbytes		4 Mbytes	
TimeStep	**Avg (us)**	**95% CI**	**Avg (us)**	**95% CI**	**Avg (us)**	**95% CI**
10	2.44	[2.35, 2.57]	11.26	[10.24, 12.77]	2755.84	[2594.89, 2873.93]
20	2.10	[2.04, 2.18]	9.15	[8.81, 9.62]	2599.37	[2384.57, 2718.48]
30	2.50	[2.38, 2.64]	11.13	[10.46, 11.85]	2584.57	[2345.41, 2816.22]
40	2.30	[2.03, 2.78]	9.67	[8.80, 11.01]	2427.94	[2010.55, 2712.90]
50	2.36	[2.35, 2.36]	10.33	[10.17, 10.62]	2865.91	[2850.34, 2880.61]
60	2.05	[2.03, 2.09]	8.89	[8.79, 9.06]	2640.45	[2543.86, 2699.63]
70	2.41	[2.36, 2.48]	10.58	[10.18, 11.15]	2710.36	[2502.20, 2875.15]
80	2.18	[2.06, 2.35]	9.62	[8.95, 10.40]	2227.86	[1802.37, 2589.42]
90	2.45	[2.35, 2.58]	10.84	[10.20, 11.55]	2701.73	[2434.69, 2890.82]
100	2.02	[2.01, 2.03]	8.76	[8.75, 8.78]	2702.91	[2695.57, 2710.38]
Baseline	2.41	[2.33, 2.56]	10.39	[10.10, 10.93]	2826.78	[2760.85, 2868.46]

Notes: Confidence intervals calculated using Bootstrap method with B=10000 (EFRON; HASTIE, 2016).

APPENDIX D – OCFTL Configuration

OCFTL is proposed as a transparent library to the final user of OmpCluster, which means that the final user does need to employ fault tolerance in the application source code. On the other hand, OCFTL provides few configurations that the final user can do via environment variables. These configurations are used to enable/disable the library, configure the heartbeat and Veloc. Following, each configuration variable is explained, and lastly, it is shown an example of how to configure the Veloc to be used as the checkpointing library.

OMPCLUSTER_FT_DISABLE: This variable is used to disable FT on the OmpCluster. Setting the value of 1 to the variable will disable all FT features. The default value of this variable is 0.

OMPCLUSTER_HB_TIMESTEP: This variable defines (in ms) the timestep of the heartbeat. The timestep is the interval that OCFTL will check the heartbeat events. Should not be confused with the `TIMEOUT` or `PERIOD`. The default value of timestep is 50.

OMPCLUSTER_HB_TIMEOUT: This variable defines (in ms) the timeout property of the heartbeat. The timeout is the amount of time a process will wait, without receiving alive messages from its emitter, until saying that its emitter failed. This timeout resets to its original value every time the process receives an alive message. The timeout should never be lower than the `OMPCLUSTER_HB_PERIOD`, which will lead to false-positive failures. The default timeout is $30s$. It is advised to set this value when using OCFTL.

OMPCLUSTER_HB_PERIOD: This variable defines (in ms) the period property of the heartbeat. The period is the amount of time a process will wait before sending the next alive message to its observer. Values lower than `OMPCLUSTER_HB_TIMESTEP` will use the value of the timestep configuration instead of the one set to this variable. The default period is $1s$. It is advised to set this value when using OCFTL.

OMPCLUSTER_CP_USEVELOC: Setting the value 1 to this variable would enable the use of Veloc as checkpointing interface for OCFTL. Other values will disable Veloc. Reminder: If the LLVM infrastructure compiles the OmpCluster without the presence of Veloc. It will be disabled regardless of the variable value. The default value of this variable is 0, disabling Veloc.

OMPCLUSTER_CP_EXECCFG: This variable sets the path to the configuration file of Veloc. This file will be used by veloc to define how Veloc will save the checkpoints. The value is unconsidered if `OMPCLUSTER_CP_USEVELOC` is set to 0. If `OMPCLUSTER_CP_USEVELOC` is set to 1 and this variable is unset, the Veloc will be disabled. There is no

default value for this configuration.

OMPCLUSTER_CP_TESTCFG: This is a test variable for the continuous integration system of OmpCluster. The final users of the library should not use it.

OMPCLUSTER_CP_MTBF: This variable defines the Mean Time Between Failures (MTBF) (in s) to be used by OCFTL to calculate the checkpointing interval. The default value of this variable is $86400s$ ($24h$)

OMPCLUSTER_CP_WSPEED: This variable defines the write speed of the disk (in MB/s) to be used by OCFTL to calculate the checkpointing interval. The default value of this variable is $10MB/s$.

When using Veloc, the user needs to provide a Veloc configuration file (`OMPCLUSTER_CP_EXECCFG`). This file contains the path for saving/loading checkpoints and some other veloc configurations. The Algorithm 14 shows an example of a basic working configuration file. More details and the full configurable Veloc variables can be checked in the Veloc documentation[1]. The basic example of Algorithm 14 configures three veloc options. First, `scratch` dir tells Veloc where to save the new checkpoints. Secondly, `persistent` dir tells Veloc where to transfer the now saved checkpoints. Lastly, `mode` equals to `async` tells Veloc to execute another program in the background to run Veloc operations; changing this to `sync` will make Veloc operation be executed by the user application, not the Veloc application.

Algorithm 14 – Basic configuration for Veloc.

```
1    scratch = /tmp/scratch
2    persistent = /tmp/persistent
3    mode = async
```

[1] Available at: <https://veloc.readthedocs.io/en/latest/userguide.html#configure-veloc>

APPENDIX E – Experimental Evaluation Configuration

E.1 Configuration

To execute the experiments, several tools were used. Algorithm 15 shows the configuration lines for the UCX and each MPI distribution tested.

Algorithm 15 – Configuration parameters for tools used

MPICH (UCX):

```
$ mpichversion | grep "config"
MPICH configure:         --prefix=<install-path> --with-device=ch4:ucx
↪ --with-ucx=<ucx-install-path>
```

MPICH:

```
$ mpichversion | grep "config"
MPICH configure:         --prefix=<install-path> --disable-silent-rules
↪ --with-hwloc-prefix=system --with-pm=hydra --with-slurm=yes
↪ --with-pmi=simple --with-device=ch3:nemesis
```

Open MPI (UCX):

```
$ ompi_info | grep "Configure comm"
Configure command line: --prefix=<install-path>
↪ --with-ucx=<ucx-install-path> --enable-mca-no-build=btl-uct
```

Open MPI:

```
$ ompi_info | grep "Configure comm"
Configure command line: --prefix=<install-path> --disable-silent-rules
```

ULFM:

```
$ ompi_info | grep "Configure comm"
Configure command line: --prefix=<install-path> --disable-silent-rules
↪ --with-ft
```

UCX:

```
$ ucx_info -v | grep "config"
configured with: --disable-logging --disable-debug --disable-assertions
↪ --disable-params-check --enable-mt --prefix=<install-path>
```

E.2 MPI Behavior

This Section shows the configuration used for the experiments discussed in Section 6.2. The configuration for building the benchmark is included in the source codes. Algorithm 16 shows the command used for each benchmark. First, there is set a timeout for each program. Following, in the command line, the variable `$recovery_flag` represents the recovery flag to disable the cleanup of processes, since MPICH was used to execute the tests, the value was `disable-auto-cleanup`. The variable `$benchmark` represents the MPI operation. The variable `$killed_proc` means which one of the two processes where killed and `$op_variant` represents the variant of the MPI operation (`blocking`, `non-blocking` or `synchronous`).

Algorithm 16 – Runtime command line for evaluation the behavior of MPI operations

```
$ export MPIEXEC_TIMEOUT=5
$ mpirun -np 2 --$recovery_flag ./$benchmark $killed_proc $op_variant
↪  >> $benchmark.out 2>> $benchmark.err
```

E.3 OmpCluster

This Section shows the configuration used for the experiments discussed in Section 6.3.1. These experiments leveraged the OmpCluster implementation in the taskbench (<https://gitlab.com/ompcluster/task-bench>) and the OmpcBench (<https://gitlab.com/ompcluster/ompcbench>) that are currently tools for internal testing of OmpCluster. Table 10 shows the OCFTL parameters for testing its overhead over OmpCluster — See Section 6.3.1.1. And Algorithm 17 shows the two versions of the `Block-Matrix-Multiplication` algorithm used discussed in Section 6.3.1.2.

Table 10 – Fault tolerance configurations for the task bench experiments

	OmpCluster	**OmpCluster + FT**
`OMPCLUSTER_FT_DISABLE`	1	0
`OMPCLUSTER_HB_TIMESTEP`	-	50
`OMPCLUSTER_HB_TIMEOUT`	-	5000
`OMPCLUSTER_HB_PERIOD`	-	1000
`OMPCLUSTER_CP_USEVELOC`	0	1
`OMPCLUSTER_CP_EXECCFG`	-	<path>/veloc.cfg

Algorithm 17 – Block Matrix Multiplication algorithms with and without OmpCluster

```
#define BS 512
#define N 2048

int BlockMatMul_OmpCluster(BlockMatrix &A, BlockMatrix &B, BlockMatrix &C) {
  #pragma omp parallel
  #pragma omp master
  for (int i = 0; i < N / BS; ++i)
    for (int j = 0; j < N / BS; ++j) {
      float *BlockC = C.GetBlock(i, j);
      for (int k = 0; k < N / BS; ++k) {
        float *BlockA = A.GetBlock(i, k);
        float *BlockB = B.GetBlock(k, j);
        #pragma omp target depend(in: BlockA[0], BlockB[0]) \
                           depend(inout: BlockC[0]) \
                           map(to: BlockA[:BS*BS], BlockB[:BS*BS]) \
                           map(tofrom: BlockC[:BS*BS]) nowait
        #pragma omp parallel for
        for(int ii = 0; ii < BS; ii++)
          for(int jj = 0; jj < BS; jj++) {
            for(int kk = 0; kk < BS; ++kk)
              BlockC[ii + jj * BS] += BlockA[ii + kk * BS] * BlockB[kk + jj * BS];
          }
      }
    }
  return 0;
}

void Matmul_Sequential(float *a, float *b, float *c) {
  for (int i = 0; i < N; ++i) {
    for (int j = 0; j < N; ++j) {
      float sum = 0.0;
      for (int k = 0; k < N; ++k) {
        sum = sum + a[i * N + k] * b[k * N + j];
      }
      c[i * N + j] = sum;
    }
  }
}
```

E.4 InteMPI Benchmarks

This Section shows the configuration used for the experiments discussed in Section 6.3.2. These tests were done with multiple MPI distributions, so each distribution was loaded before each experiment, to compile the IntelMPI Benchmarks it is necessary to configure the `CC=mpicc` and `CXX=mpic++` variables. For the experiments using OCFTL, the library object is needed (`<bench-dir>/ocftl/libocftl.a`. Algorithm 21 shows the modifications that were made in the benchmarks to run with OCFTL, and the Algorithm 19 shows the modifications made in the benchmarks to run with ULFM-2. Finally, Algorithm 20 shows how each benchmark was tests for each MPI tested. In these tests, the

heartbeat parameters discussed in Section 6.3.2 for OCFTL were configured via enviroment variables while for ULFM they are configure via runtime flags (`$HB_PERIOD`, `$HB_TIMEOUT` and the flag `--mca mpi_ft_enable <true|false>` to enable/disable ULFM).

Algorithm 18 – Diff output of the IntelMPI benchmarks with and without OCFTL

```
diff -br mpi-benchmarks/src_cpp/imb.cpp
↪  thesistests/IntelMPIBench/src_cpp/imb.cpp
@@ 62a63 @@
+ #include "../ocftl/ft.h"
@@ 323a325,328 @@
+         FaultTolerance *ft = new FaultTolerance(1000, 3000,
↪  MPI_COMM_WORLD);
+         ft->disableAsserts();
+         ft->start_hb = true;
+
@@ 354a360,361 @@
+
+         delete ft;
diff -br mpi-benchmarks/src_cpp/Makefile
↪  thesistests/IntelMPIBench/src_cpp/Makefile
@@ 86a87,92 @@
+ # OCFTL addition
+ override CXXFLAGS += -std=c++11 -L../ocftl -g
+
+ # OCFTL Wrappers
+ OCFTL_WRAPPERS = -Wl,--wrap=MPI_Wait -Wl,--wrap=MPI_Test
↪  -Wl,--wrap=MPI_Barrier -Wl,--wrap=MPI_Comm_free -Wl,--wrap=MPI_Mprobe
↪  -Wl,--wrap=MPI_Send -Wl,--wrap=MPI_Recv
+
@@ 160c166,167 @@
- scope.h
---
+ scope.h \
+ ../ocftl/ft.h
@@ 180c187 @@
-         $(CXX) $(CPPFLAGS) $(CXXFLAGS) -o $@ $^ $(LDFLAGS)
---
+         $(CXX) $(CPPFLAGS) $(CXXFLAGS) $(OCFTL_WRAPPERS) -o $@ $^
↪  $(LDFLAGS) -locftl -lpthread
```

E.5 OCFTL Benchmarks

This Section shows the configuration used for the experiments discussed in Sections 6.3.3 and 6.3.4. The configuration for building the benchmark is already present

Algorithm 19 – Diff output of the IntelMPI benchmarks with and without ULFM additions

```
diff -br mpi-benchmarks/src_cpp/imb.cpp
↪  thesistests/IntelMPIBench-ULFM/src_cpp/imb.cpp
@@ 63a64,76 @@
+ #include <mpi-ext.h>
+ #include <stdio.h>
+ #include <unistd.h>
+ #include <signal.h>
+
+ void ErrorHandler(MPI_Comm *pcomm, int *perr, ...) {
+   int len, ec;
+   char errstr[MPI_MAX_ERROR_STRING];
+   MPI_Error_string(*perr, errstr, &len);
+   MPI_Error_class(*perr, &ec);
+   fprintf(stderr, "Found Error: %s (Error class %d)\n", errstr, ec);
+ }
+
@@ 322a336,339 @@
+
+         MPI_Errhandler errh;
+         MPI_Comm_create_errhandler(ErrorHandler, &errh);
+         MPI_Comm_set_errhandler(MPI_COMM_WORLD, errh);
```

with the source codes. These tests were also executed using the regular MPICH, and both tests use the same program. Table 11 shows the enviroment variable values for each test. Algorithm 22 shows the command line for running the benchmarks, `$nprocs` represents the number of processes killed (same as `FTLIB_TOTAL_FAILURES`) and `$type` represents if the failure are sequential (value 0) or random (value 1). For the internal broadcast experiment, `$type` was equal to 1.

Table 11 – Runtime environment variables for the Locality and Internal Broadcast experiments

Environment Variable	Locality	Broadcasts
OMPCLUSTER_FT_DISABLE	0	
OMPCLUSTER_WRAPPERS_DISABLE	1	
OMPCLUSTER_HB_TIMESTEP	1	
OMPCLUSTER_HB_PERIOD	100	
OMPCLUSTER_HB_TIMEOUT	1000	
FTLIB_TOTAL_FAILURES	<1|2|3|4|5|8|16>	
FTLIB_RING_SHUFFLE	<0|1>	0
FTLIB_WHICH_BC	0	<0|1|2>
MPIEXEC_TIMEOUT	60	

Algorithm 20 – IntelMPI Benchmark run commands for the different MPI distributions

PingPong Benchmark:

```
mpirun -np 480 $flags ./IMB-MPI1 pingpong -thread_level multiple -multi 1
↪ -msglen lengths.txt >> res.out 2>> res.err # If OCFTL
mpirun -np 480 $flags --mca mpi_ft_detector_period $HB_PERIOD --mca
↪ mpi_ft_detector_timeout $HB_TIMEOUT ./IMB-MPI1 pingpong -thread_level
↪ multiple -multi 1 -msglen lengths.txt >> res.out 2>> res.err # If
↪ ULFM-2
```

Allreduce Benchmark:

```
mpirun -np 480 $flags ./IMB-MPI1 allreduce -thread_level multiple -npmin
↪ 480 -msglen lengths.txt >> res.out 2>> res.err # If OCFTL
mpirun -np 480 $flags --mca mpi_ft_detector_period $HB_PERIOD --mca
↪ mpi_ft_detector_timeout $HB_TIMEOUT ./IMB-MPI1 allreduce
↪ -thread_level multiple -npmin 480 -msglen lengths.txt >> res.out 2>>
↪ res.err # If ULFM-2
```

Algorithm 21 – Definition of length.txt file and runtime flags for each MPI distribution tested

lengths.txt:

```
0
65536
4194304
```

flags:

```
flags="-x UCX_NET_DEVICES=mlx5_0:1" # For MPICH (UCX)
flags="iface ib0" # For MPICH
flags="-mca pml ucx -mca btl ^uct -x UCX_NET_DEVICES=mlx5_0:1" # For Open
↪ MPI (UCX)
flags="" #For Open MPI
flags="--mca orte_enable_recovery true --mca mpi_ft_enable true --mca
↪ mpi_ft_verbose 1 --mca mpi_ft_detector true --mca
↪ mpi_ft_detector_thread true" # For ULFM-2
```

Algorithm 22 – Runtime command line for evaluation OCFTL's internal broadcast and the locality experiments

```
$ mpirun -np 480 -iface ib0 --disable-auto-cleanup ./locality $nprocs
↪ 24 $type >> log.out 2>> log.err
```