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

Pedro Henrique Di Francia Rosso[1,2][0000−0001−6482−8745], Emilio Francesquini[2][0000−0002−5374−2521]

[1] Universidade Estadual de Campinas (UNICAMP) – Campinas, SP – Brazil
p233687@dac.unicamp.br
[2] Universidade Federal do ABC (UFABC) – Santo André, SP – Brazil
{pedro.rosso, e.francesquini}@ufabc.edu.br

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. OCFTL can detect and propagate failures; provides false-positive detection; exchanges a reduced number of messages during failure propagation; employs checkpointing to reduce the impact of failures; and has a reduced delay to detect sequential simultaneous failures in comparison to related works.
Keywords: Fault tolerance · High-Performance Computing · Message Passing Interface

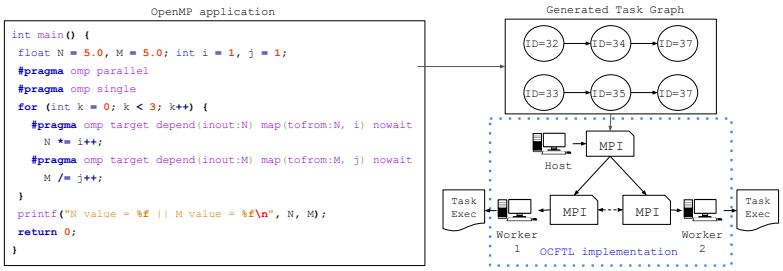
1 Introduction

Message Passing Interface (MPI) is a message passing protocol specification frequently used to parallelize and distribute jobs in High Performance Computing (HPC). Among several implementations, Open MPI [10] and MPICH [4] are the most commonly employed by the HPC community. HPC applications typically run on large clusters and dedicated computing environments. In such environments, Fault Tolerance (FT) is a common concern justified by the large number of computing nodes which ultimately leads to an increased failure rate [8].

Even today, the development of fault-tolerant MPI applications is done manually, with some exceptions like ULFM and MPICH. User Level Failure Migration (ULFM) [3] is implemented inside Open MPI whereas MPICH has its own FT routines [4]. However, these approaches are not portable, nor easy to use, and are far from complete FT solutions. Building a portable and implementation-independent FT library, OCFTL (OmpCluster Fault Tolerance Library), is the main focus of this work. OCFTL can be used in a plug-and-use style with any standard-compliant MPI implementation. OCFTL uses MPI functions defined by the specification, and it does not rely on modifications to the MPI implementation itself, which makes it portable.

OCFTL provides mechanisms to detect and propagate failures, survive them and complete the program execution correctly even if in the presence of failures. This paper describes the rationale of OCFTL and the use of it in OmpCluster (https://ompcluster.gitlab.io), a project that focuses on easing parallel programming on HPC clusters using OpenMP (https://www.openmp.org/).

Fig. 1: Flow of the parallelization process.



2 Background and Motivation

This work is inserted in the context of the OmpCluster research project, which aims at easing scientific programming for HPC clusters by leveraging the LLVM compiler infrastructure. This project aims at parallelizing applications with OpenMP tasks. These tasks are deployed by the runtime to the target cluster using MPI as shown in Figure 1. An application execution flow follows: from an OpenMP program, the runtime system creates the tasks that have been previously described by the programmer using OpenMP directives, building a Directed Acyclic Graph (DAG) that models the dependencies between the tasks. Those tasks are then scheduled and distributed across the computing nodes (either CPU or CPU-GPU nodes) using MPI. When the execution completes, the final results are sent back to the main program. The MPI task distribution is based on an event system, where each MPI process implicitly communicates within this event system.

To provide fault tolerance for OmpCluster, we need to take into account failure detection and propagation, and failure mitigation. The first consists in the library being able to detect and propagate failures, making every process on the application achieve a consistent state (*i.e.*, knowing all the failures in the system). The second consists in reducing the impact of failures, a common way to do this is to restart the program whenever a failure happens, which would cause a significant overhead. The mitigation of this impact is essential to reduce the time spent dealing with failures.

2.1 Failure Detection and Propagation

One of the most common approaches found in the literature to detect failures in HPC system is through the use of a heartbeat message. In this detection system, each process is an `emitter` (it sends alive messages to other processes), but also, an `observer` (it receives alive messages from other processes). In this work, we employ a ring topology, where a process is an emitter to the next process in the ring and an observer for the previous process in the ring. There are two properties of interest to the heartbeat mechanism: the `period` (period of time between each alive notification), and the `timeout` (amount of time to wait between notifications from the observed process to declare it as failed).

This kind of heartbeat-based failure detection mechanism was used by ULFM [3], and Zhong *et al.* [22] which uses heartbeats as a secondary failure detection mechanism. Another important concept is failure propagation. To achieve a consistent state between all processes in the application, every process

needs to know about every failure. To achieve that, the process that detects a failure broadcasts it to the remaining processes. This is done using a fault-tolerant broadcast. It is essential that every process receives the broadcast, so all of them have the same vision of the processes' states in the system, thus the necessity of a fault-tolerance broadcast.

ULFM [3] employs a broadcast based on HBA (hyper-cube), which consists of, from the starter process, sending the broadcast to k processes forward and k processes backward in the ring where N is the number of processes and $k = \lfloor log_2(N) \rfloor$. On the other hand, Zhong *et al.* [22] employ the use of circulant graphs, for instance the binomial graph BMG. In this case, the broadcast is sent by the process with rank k to processes with ranks $k + 2^0, k + 2^1, k + 2^2, ...$, forward and backward in the ring. In both cases, each process, except for the starter process, will replicate the broadcast the first time they receive it. One important thing to notice is that in this kind of broadcast, multiple messages will be received by each process, making it fault-tolerant.

2.2 Failure Mitigation

One can find in the literature several approaches, each with their own strengths and weaknesses, to failure mitigation. In this work, we employ checkpointing to deal with failures. In future works, we intend to employ replication as well.

Checkpointing is one of the most common failure mitigation techniques. It consists in saving program snapshots to a stable and reliable storage [13]. There are several categories of checkpointing. The category that more closely matches with the requirements of the OmpCluster project is the *application-level* category. In this type of checkpointing, applications are required to guide what data must be saved and loaded, giving total control of the checkpointing procedure to the application. In OmpCluster's case, the user is OmpCluster's runtime system itself. Examples of libraries that fall in this category include SCR [15], Veloc [16] and FTI [2]. Since Veloc allows the application to perform both coordinated and uncoordinated checkpointing[3] and SCR and FTI are strictly coordinated, Veloc was chosen as to be the checkpointing library for OCFTL.

3 Related Work

Fault tolerance and MPI are a frequent topic of research interest. Some works propose targeted MPI implementations. Open MPI, for example, refers users to a separate implementation, ULFM [3], as its main FT approach[4]. MPICH, on the other hand, has limited support to FT in the vanilla implementation for TCP-based communications, https://wiki.mpich.org/mpich/index.php/Fault_Tolerance. However, it is not clear what behavior can be expected on other configurations such as Infiniband or shared memory.

ULFM proposes a FT chapter for the MPI specification. By this date, it is implemented inside Open MPI and relies on it. The authors propose functions to limit the impact failures have on the application execution, allowing users to invalidate and shrink communicators (modifying it to contain only alive

[3] Coordinated checkpointing requires synchronization of checkpoint functions in every process, while uncoordinated checkpointing does not require synchronization.

[4] https://www.open-mpi.org/faq/?category=ft#ft-future

processes), agree to something (*e.g.*, a flag), and acknowledge that a failure happened. It uses the heartbeat failure detection mechanism and a hypercube-based algorithm (HBA) to propagate failure [5]. In a recent proposal [22], the authors employ a Binomial Graph (BMG) based algorithm for failure propagation. OCFTL also uses a heartbeat mechanism, however it does not employ any additional detection system such as discussed by Zhong (*e.g.* OS Signals) [22]. This lean mechanism makes our algorithm simpler and allows us to use a proven and scalable peer-to-peer topology for failure propagation based on Chord [7].

Reinit++ [12] is an approach that was tested with OpenMPI. 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++ failure detection mechanism is based on a root node which monitors some daemon processes that, in turn, monitor application MPI processes, to do that, the authors extended OpenMPI runtime process manager.

Other works such as LFLR [20] and Fenix [11] use ULFM to allow recovery after failures. LFLR focuses on local recovery, dealing only with the restart of the failed component. Thus, no global restart of the program is required. In this sense, it is much closer to the approach followed by OCFTL. Similarly, Fenix uses procedures proposed by ULFM to perform process recovery. Additionally, Fenix supports different types of checkpoints (implicit, asynchronous, coordinated, and selective). Contrary to these approaches, OCFTL employs its own failure detection and propagation mechanisms [17], not relying on other libraries like ULFM and thus making it portable.

A hybrid FT approach was proposed to take into account failures in the EasyGrid middleware (a framework for grid-type applications) using checkpointing and message logs, as well as self-healing properties [18]. On top of LAM/MPI and EasyGrid framework, the authors use those tools to detect and recover from failures. In this sense, OCFTL tends to be more generic, providing a more portable FT library as well as not relying on another tool.

MPI stages [19] proposes a brand-new MPI implementation with FT. However, it offers only basic MPI functionality as well as checkpointing. It is not clear how its failure detection and propagation system works. OCFTL, on the other hand, is based on the MPI specification, and can (and should) therefore be used with any MPI compliant implementation.

Finally, some works constitute the initial proposals for FT in MPI, that were later migrated to other projects. It is the case of LA/MPI [1] and MPI-FT [14]. As well as other projects focuses on old versions of MPI standard and are discontinued or outdated, like the FT-MPI [9].

4 An Implementation-Independent Fault Tolerance Library – OCFTL

OCFTL provides FT mechanisms compatible with any specification-compliant MPI implementation, such as OpenMPI and MPICH, employing user-level MPI functions only, making it easier to update as new MPI standards are released. To do so, OCFTL creates an additional thread for each MPI process to control the

FT heartbeat through a main loop. This loop checks, on each iteration, if there are any messages to send or receive, or if there are any other FT procedures pending. Thus, OCFTL requires MPI to be initialized with `MPI_THREAD_MULTIPLE`. Finally, for the time being, OCFTL is implemented and focuses on C/C++ only.

4.1 Failure Detection

Our failure detection mechanism was inspired by ULFM's [5]. The algorithm was modified to work as a user-level library, relying on the standard MPI functions and proposing some improvements.

OCFTL presents important differences from ULFM's failure detection mechanisms. First, the failure propagation is performed using a modified broadcast based on the Chord algorithm [7] which is similar to BMG broadcast. In this approach, 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 $\log_2(N)$, N is the size of the heartbeat ring, and pos_r and pos_s indicates the position of process r and s respectively. This is done once per process upon receiving the first broadcast from another process, replicating the broadcast $\log_2(N)+1$ times in total, which differentiates from the original chord-based algorithm, that avoids the message redundancy. Additionally, the expected time to achieve a consistent state is $\log_2(N)$ multiplied by the main loop iteration time. When a process fails, it is removed from the ring data structure present on every process.

OCFTL also provides *false positive* failure detection. This means that, when a process failure is detected but after a while, the process recovers (*e.g.* due to an intermittent network failure), the former observer will capture the message and start a procedure of adding it back to the ring. This is achieved using the broadcast procedure discussed before.

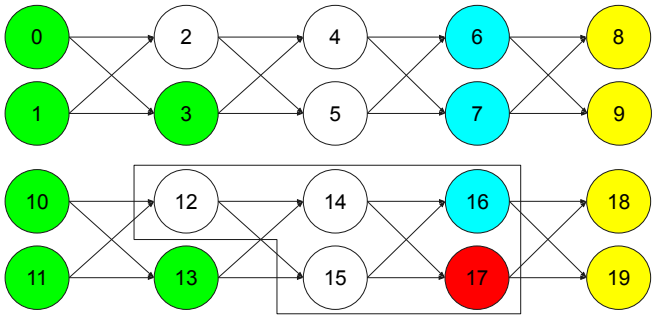
OCFTL also includes an optional initial procedure to shuffle the distribution of MPI processes throughout the heartbeat ring (this shuffling process only affects the heartbeat, not impacting on the applications). Often, MPI processes are distributed sequentially to the machines or rack, and when a failure occurs, if those MPI processes are observing other processes in the same machine/racks, the procedures taken by the heartbeat would take $(2(n-1)+1) \times hb_{to}$ to detect all failures, where hb_{to} represents the heartbeat timeout and n represents the total number of processes in the machine/rack. The shuffling procedure reduces the average time to detect failures since it spreads the MPI processes evenly across the ring, which reduces the probability of processes in the same machine or rack to appear sequentially on the ring. Assuming m machines with n consecutive processes each, $P_n = m \prod_{i=0}^{n-1} \frac{1}{(m*n)-i}$ represents the probability of having n consecutive process in the ring.

Since OCFTL is focused on HPC applications, which typically run on large clusters, we expect the number of failures to increase linearly with the number of machines. On the other hand, the time needed to detect and propagate a failure only increases at a logarithmic rate.

4.2 Handling Failures

To handle failures, we use checkpointing, and plan to add replication in the future. Checkpoint/restart consists of periodically saving states of the execution, and in case of failures, resuming the application from the last saved state.

Fig. 2: Task restart procedure illustration



To execute checkpointing, OCFTL leverages Veloc [16]. Veloc is a library that permits saving and loading checkpointing synchronizing or not the processes. It also provides an infrastructure that saves the process locally (same machine as the application process execution) and sends the checkpoint in background to a more reliable storage (like a distributed storage). In this work, OCFTL stands as an interface between the application and Veloc. Thus, the application depends on OCFTL's interface, making it possible to exchange the checkpointing library without the need to perform adaptations on the application code. This interface also provides the checkpoint interval calculation that takes into account the cost (δ) of the checkpoint (time to write it) and the MTTI (M) (Mean time to interruption)($\theta = \sqrt{2\delta M}$) [21], which is directly connected to OCFTL's notification system , emitting a notification when a checkpoint is necessary.

OmpCluster, in particular, employs coordinated checkpointing, *i.e.*, every OmpCluster process is suspended during checkpointing and resumed after it is done. The loading process, for the time being, is still coordinated due to limitations of OmpCluster's runtime infrastructure. When these limitations are lifted, checkpointing will be uncoordinated, so loading will be limited to only a few specific processes. Since OCFTL focuses on a task-based runtime environment, Figure 2 illustrates a procedure to identify which tasks need to be restarted. The green shaded tasks have already been saved, *i.e.*, their output are available in saved checkpoints; the white tasks indicate executed but not saved; the blue tasks are in execution; the red one is a task that began its execution but suffered from a failure; and yellow tasks are the not yet executed tasks.

After identifying a failure of a process, the system will check if any task has failed, in the case of the Figure 2, the task 17 failed. First, the system will wait for the currently executing tasks (6, 7 and 16) to finish, and then start a `restart` procedure. To define which tasks will be restarted, we evaluate each task that is a dependency or a dependent of the failed task, both directly or indirectly. This means that from the task 17, we recursively evaluate the dependencies and the dependents, repeating for each new task, and marking all tasks that were executed, but not saved, for restart. For the example of shown on Figure 2, we evaluate the tasks 14, 15, 18 and 19 first, and after the tasks 16, 12, 13 and 10. From all those tasks, the ones that fill the requirements to restart are 12, 14, 15 and 16. This procedure avoids the overhead of restarting unnecessary tasks, if, for example, restarting all non-saved tasks from the last checkpoint taken.

4.3 Repairing Communicators

MPI provides optimized collective communications between processes. However, they require the participation of every process from a given communicator. In an environment with nodes that might experience failures, this can be a frequent cause of deadlocks.

To tackle this problem, we propose a repair function similar to `MPI_Comm_Shrink` (available in ULFM [3]). This function shrinks the set of processes to contain only alive processes (as determined by the failure detection mechanism described in Section 4.1). The shrinkage only completes successfully if all MPI processes have a coherent view of the communicator state, *i.e.*, they have the same view of the whole system regarding which processes are alive or dead. An inconsistent state (their view are not coherent) might be reached when a failure happens during the propagation process performed by the library after the detection of a previous failure, this is currently a limitation for this project.

4.4 Gathering States

To give OCFTL users increased control over their applications, the library provides state gathering functions that return the current state of a process or communicator. Not only this can be used by users to verify the state of execution, but also, it is useful to implement pre- and post-processing in OCFTL wrappers for MPI operations. The state of a process can be *alive* or *dead* while a communicator can be *valid* or *invalid* (when a process fails). This is used in OCFTL wrappers for MPI functions in the case of OmpCluster.

4.5 MPI Wrappers

One of the problems of using MPI with the possibility of failures is the behavior of MPI functions, which could deadlock if a failed process is participating. To tackle that, OCFTL employs function wrappers. This is achieved leveraging linker option "`-Wl,--wrap=FUNCTION`" available in the GCC and Clang compilers. This option redefines the symbol `FUNCTION` to two new symbols, for example, in the case of a `MPI_Send` function: the `__real_MPI_Send`, expressing the original `MPI_Send` function; and, the `__wrap_MPI_Send`, a custom implementation for `MPI_Send`. Any call in the code for `MPI_Send` will in fact be calling `__wrap_MPI_Send`. These wrappers could be created for any MPI function. In the particular case of OmpCluster, the functions `MPI_Wait`, `MPI_Test` (used as a loop condition), `MPI_Barrier`, `MPI_Comm_free`, `MPI_Mprobe`, `MPI_Send` and `MPI_Recv` were wrapped[5], but any MPI function could be wrapped in the same way.

The general solution for these wrappers is to use the non-blocking variant existent for almost every MPI function and loop test the associated `MPI_Request` object while verifying the other part of the communication. For example, Algorithm 1 replicates the behavior of `MPI_Send`, waiting in the function while the operation does not complete. Line 3 replaces the blocking send for the non-blocking send, associating a request object to it. This request is tested in every loop iteration (lines 5 to 11), as well as the status of the destination is checked (lines 7 to 10). If the operation completes successfully (the test in line 6 completes), the function returns success, otherwise, if the destination process is not

[5] OCFTL's wrappers source codes can be found in the library implementation, available at https://gitlab.com/phrosso/ftmpi-tests/-/tree/master/OCFTLBench/ftlib

alive, the function returns an error and avoids the possible deadlock caused by an operation involving a dead process.

Algorithm 1: Wrapper function to `MPI_Send`.

```
int __wrap_MPI_Send(const void *buf, int count, MPI_Datatype datatype,
↪  int dest, int tag, MPI_Comm comm) {
  MPI_Request w_send_request;
  MPI_Isend(buf, count, datatype, dest, tag, comm, &w_send_request);
  int test_flag = 0;
  while (!test_flag) {
    MPI_Test(request, &test_flag, status);
    if (getProcessState(dest) != ProcessState::ALIVE) {
      MPI_Request_free(request);
      return ft::FT_ERROR;
    }
  }
  return ft::FT_SUCCESS;
}
```

4.6 Notification Callbacks

Integration between OCFTL and applications is done using an event notification system. Currently, OCFTL has four types of notifications: `Failure`, `False-Positive`, `Checkpoint` and `CheckpointDone`. For the application to receive these notifications, it needs to register a callback function using the OCFTL method `registerCallback`. Whenever one of those notifications is triggered, OCFTL calls all registered function (one or more functions can be registered). The treatment for these callbacks is specific to each application using OCFTL, and specifies what should be done after a notification is received.

5 Experimental Results and Discussion

This section discusses the experiments performed to assess different MPI implementation behaviors and to benchmark OCFTL[6]. To execute the tests, OpenMPI v4.1 and MPICH v3.4.2 were used along with the respective recovery flags (`"--enable-recovery"` and `"--disable-auto-cleanup"`), initializing with `MPI_THREAD_MULTIPLE` thread level option. All tests were performed in the Santos Dumont supercomputer (SDumont, LNCC - Brazil). For the reported experimental results, the computing nodes named B710 featuring 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) were used. The network interface is InfiniBand (56 Gb/s). The number of cores was selected according to each test.

5.1 MPI Behavior

Each MPI implementation has ample freedom to implement the MPI standard, so it is expected that different MPI implementations present different behavior in the presence of failures. To properly implement a fault tolerant solution in

[6] The tests and the library are available at: https://gitlab.com/phrosso/ftmpi-tests

Table 1: 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: 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)

OCFTL, it was necessary to evaluate the behavior of each implementation in different critical execution scenarios.

To evaluate the behavior of different MPI distributions, point-to-point and collective 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 the other killing the MPI rank 1. Each run is considered unsuccessful if it times out, since the objective of these tests is to check if the operation deadlocks or not. The benchmark programs and runtime configurations are available on the `Behavior` folder of the aforementioned Git repository.

Tables 1 and 2 show the results of the experiments with a few MPI operations on MPICH and OpenMPI distributions with and without UCX[7], respectively. For both tables, each cell represents the results of each variation of an operation when applicable, in the order `Blocking` / `Non-Blocking` / `Synchronous`. Pos-

[7] UCX is an optimized framework for communications between nodes in high-bandwidth and low-latency networks, which is commonly used with MPI.

Fig. 3: PingPong benchmark for messages with 64 Kbytes size

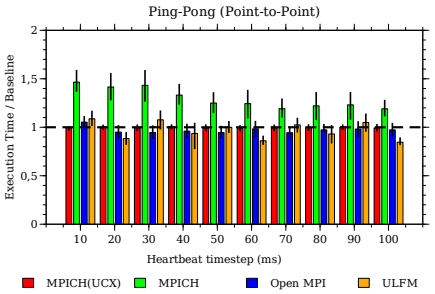
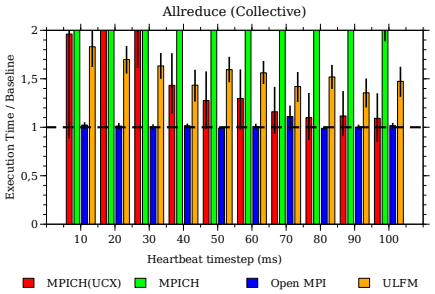


Fig. 4: AllReduce benchmark for messages with 64 Kbytes size



Note: Bars overlapping the plot limits in Figure 4 show a relation between tests and baseline ranging from 2.06 to 30.34.

sible values are: `ok` if the program finished (with or without errors); `to` if the program timed out; and `(-)` if the variation does not exist.

Table 1 shows that for MPICH configured without UCX, the program finished for all tests, but the synchronous send. This was expected since MPICH in its basic configuration has support for failure detection. 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 `(*)` next a result means that the program does not time out during the target function, but on the execution of `MPI_Request_free` or on `MPI_Finalize` functions.

Table 2 shows that for Open MPI without UCX, most of the `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.

These results show the importance of wrappers while working with fault tolerance and MPI. The idea of wrappers is replicating the function behavior with functions that not deadlock, and always check the other side of the communication, or the communicator state if it is a collective communication. This way, the behavior of these tests can be avoided.

5.2 Empirical OCFTL performance evaluation

This section evaluates the OCFTL under stressed MPI programs[8]. To perform such evaluations, we use the Intel MPI Benchmarks[9], which is a set of MPI benchmarks intended to test MPI distributions of the MPI standard.

This suit offers various subsets of benchmarks . These experiments leverage the MPI-1 subset, evaluating elementary MPI operations. It 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, we choose the `ping-pong` as a `P2P` operation (since it

[8] We consider an MPI program as `stressed` when it is overloaded by MPI message exchanges. Benchmarks are examples of stressed MPI programs, since they overload the MPI runtime with MPI operations to establish the limits of a MPI distribution.

[9] Available at https://github.com/intel/mpi-benchmarks

is a `send-recv` operation), and the `allreduce` application as a `coll` operation (since it is an all-to-all operation). To limit the combinatorial explosion of 4 different MPI distributions with different heartbeat parameters, we chose only one benchmark of each type.

These experiments use 20 nodes with 24 processes each (480 processes in total). Discussions in this paper are limited to the tests with the `medium` size of messages(64 Kbytes per message). But tests were also performed with a `min` length (0 Bytes per message) and `large` length (4 Mbytes per message)[10]. The number of internal repetitions of the operation, for each run, is 640 times (the default value for messages with 64 Kbytes). 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 every process 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.

Four MPI distributions were tested: MPICH with UCX, MPICH without UCX, Open MPI, and ULFM. For the first three, OCFTL was the FT approach, while for the last, it used the ULFM's detector. The experiment tested different values of heartbeat period, and the timeout was set to 100 times higher than the period. The baseline in the tests represents the execution of the benchmarks without fault tolerance support. Tests were executed 10 times and the confidence intervals are calculated using the Bootstrap method [6] with 10000 iterations. The Bootstrap method was chosen since we do not know, beforehand, the probability distribution of the data.

Figures 3 and 4 show the results for the messages with 64 Kbytes. `P2P` resultsshows that only MPICH without UCX has significant overhead. For the `collective` tests, Open MPI was the only distribution that did not show overheads. In particular, MPICH without UCX, showed a high overhead for lower values of the heartbeat period. One important aspect to notice is the total run time of the applications. Although MPICH showed higher overheads in the `collective` tests, it showed better performance (time wise) than Open MPI in all cases and ULFM (for values of heartbeat period higher than $40ms$). MPICH with UCX showed the best completion times among all distributions. Evaluating the execution times of `P2P`, Open MPI was as good as ULFM, while MPICH with and without UCX performed worse. Since OCFTL can be used with any standard-compliant distribution, this portability allows us to choose the best distribution, where the difference in execution times between MPI implementations can easily surpass the overhead caused by OCFTL.

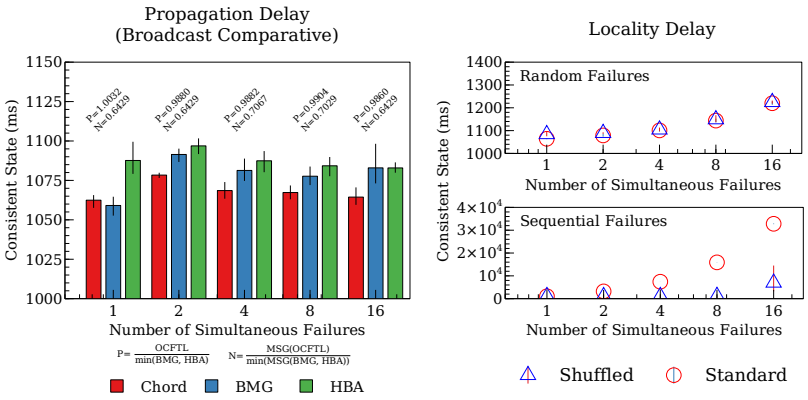
5.3 Internal Broadcast

To evaluate the internal broadcast, a comparison was made between our approach, BMG and HBA algorithms, where each broadcast was implemented within OCFTL to be evaluated.

The objective is to evaluate the time to propagate a failure and the overhead they would impose on the application. So the tests monitored the total time needed to propagate a set of simultaneous failures to all remaining application

[10] All test results are available in the aforementioned Git repository. Due to space limitations, here we show only results with the most significant overheads (*i.e.*, overheads for the remaining configurations is lower than those presented here).

Fig. 5: Internal Broadcast comparison (left) and Locality delay results for standard and shuffle heartbeat initial positions (right).



Note: Error bars represent the 95% confidence interval calculated using Bootstrap with 10000 iterations [6].

processes. 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 16 simultaneous failures. Each scenario consists of 10 samples.

Figure 5 (left) shows the comparison between the broadcasting algorithms. Our approach is marginally better than BMG and HBA (time taken to achieve a consistent state is practically the same for all of them) as the values of `P` (representing the performance relation between our approach and the best of BMG and HBA) are close to 1. 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 taken into account). These results show that our broadcast algorithm is a viable and better alternative to current state-of-the-art FT broadcasting algorithms.

5.4 Locality Problem

The rationale behind shuffling positions of the MPI rank in the ring is to solve the delay of multiple sequential failure detection. These tests evaluate two approaches: the standard, which does not shuffle the initial positions of the processes, and the shuffled, which is the approach we suggest that random shuffle the initial positions of the process in the ring. For this experiment, 480 processes were used over 20 nodes and each scenario consists of 10 samples.

This experiment proposes the evaluation of the detection delay by simulating simultaneous failures (1, 2, 4, 8, and 16 simultaneous failures) for two cases: sequential distribution of ranks over the nodes (*e.g.,* 0 – 23 in the first node, 24 – 49 in the second node, and so on), which is the worst option for the standard approach; and, round-robin distribution of processes (*e.g.,* 0 in the first node, 1 in the second node, and so on), which is the best option for the standard approach. The worst and best cases are theoretically defined, but as OCFTL includes a probability factor, this is not deterministic anymore. Figure 5 (right)

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 failures increases. The shuffle option is shown to be a better choice for sequential failures, since the times to achieve a consistent state are less or equal to the standard option. In general, experiments show that shuffling is a good option to avoid the detection delay for sequential failures.

5.5 Checkpointing

Since OmpCluster is our case of study, the checkpoint was configured to run accordingly to the OmpCluster runtime. Whenever OmpCluster head process receives the checkpoint notification, a coordinated checkpoint occurs, saving all data future tasks will need. When a task fails, all the necessary data to run the tasks marked to restart (like discussed in Section 4.2) is restored. This system was tested with a blocked matrix multiplication application, in which, after some time, a checkpoint is taken, then, a failure is injected causing a failure to at least one task. So, based on the tasks' evaluation, all necessary data is loaded, and then, the tasks are restarted. All the tests ran successfully. A general MPI application (whose communication graph does not follow that of a task-based application such as that of OmpCluster) is still a work in progress. The current checkpointing algorithm is suited to task-based applications, but when working with other types of applications whose tasks are not well-defined, the programmer still needs to use the available checkpoint functions to use OCFTL.

5.6 Limitations

OCFTL is under development. Therefore, limitations described in this section are expected to be removed in the near future. The first observed limitation is present in the repair communicator procedure, in which, an inconsistent state of agreement between the processes occurs if a request to repair a communicator is made while OCFTL is in the middle of failure propagation, where processes will not agree with each other, and the procedure will not succeed. Concerning OmpCluster, there are two main limitations: the restarting of failed tasks is currently sequential, given OmpCluster's runtime limitations; and, OCFTL is not able to deal with failures in the head process (rank 0), since this process holds the OpenMP core, and if it fails, the application needs to be restarted.

5.7 Stand-alone Use

This paper brings the discussion about OCFTL in the context of OmpCluster project. Although some solutions are specific for the project, the core of OCFTL can be used in a stand-alone way. The failure detection and propagation mechanisms, checkpointing, state gathering and communicator repair could be all used by other applications that leverages the task-based workflow paradigm. As OmpCluster deals with the FT notifications of OCFTL, the responsibility of dealing with each notification falls to the stand-alone application, which should be able to deal with each notification accordingly to its needs.

6 Conclusions and Future Work

Providing FT for MPI is a complex task. This paper proposes OCFTL, a FT library that focuses on the portability, compatible with any MPI standard-compliant distribution. OCFTL provides failure detection and propagation mechanisms, and although inspired by other FT MPI approaches, it does not rely on any of them. For checkpointing it employs Veloc, however the application only interfaces with OCFTL providing portability. It also provides the application with checkpoint interval calculation.

Results show that OCFTL wrappers can avoid deadlock of MPI operations in the presence of failures; that the portability of OCFTL can overcome the overhead it imposes to the different MPI distributions; that the internal broadcast reduces the number of messages employed while retaining FT characteristics; and, the delay when detecting simultaneous sequential failures can be reduced. It also demonstrates that OCFTL can be used in stand-alone mode with applications or systems other than OmpCluster.

This is an ongoing project, so as future works we plan to tackle the presented limitations, as well as improving the library, providing incremental checkpointing, support for FT in GPUs and FT for the head process of OmpCluster.

Acknowledgments

The authors are grateful to the Center of Petroleum Studies (CEPETRO-Unicamp/Brazil) and PETROBRAS S/A for the support to this work as part of BRCloud Project.

References

1. Aulwes, R., Daniel, D., Desai, N., Graham, R., Risinger, L., Taylor, M., Woodall, T., Sukalski, M.: Architecture of LA-MPI, a network-fault-tolerant mpi. In: 18th International Parallel and Distributed Processing Symposium, 2004. Proceedings. pp. 15–. IEEE, Santa Fe, NM, USA (2004). https://doi.org/10.1109/IPDPS.2004.1302920
2. Bautista-Gomez, L., Tsuboi, S., Komatitsch, D., Cappello, F., Maruyama, N., Matsuoka, S.: FTI: High performance fault tolerance interface for hybrid systems. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. SC '11, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/2063384.2063427, https://doi.org/10.1145/2063384.2063427
3. Bland, W., Bouteiller, A., Herault, T., Bosilca, G., Dongarra, J.: Post-failure recovery of mpi communication capability: Design and rationale. The Intl. Journal of High Performance Computing Applications **27**(3), 244–254 (2013)
4. Bosilca, G., Bouteiller, A., Cappello, F., Djilali, S., Fedak, G., Germain, C., Herault, T., Lemarinier, P., Lodygensky, O., Magniette, F., 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. pp. 29–29. IEEE (2002)
5. Bosilca, G., Bouteiller, A., Guermouche, A., Herault, T., Robert, Y., Sens, P., Dongarra, J.: A failure detector for hpc platforms. The Intl. Journal of High Performance Computing Applications **32**(1), 139–158 (2018)
6. Efron, B., Hastie, T.: Computer age statistical inference, vol. 5. Cambridge University Press, Cambridge, United Kingdom (2016)

7. El-Ansary, S., Alima, L.O., Brand, P., Haridi, S.: Efficient broadcast in structured p2p networks. In: International workshop on Peer-to-Peer systems. pp. 304–314. Springer (2003)
8. Elliott, J., Kharbas, K., Fiala, D., Mueller, F., Ferreira, K., Engelmann, C.: Combining partial redundancy and checkpointing for hpc. In: 2012 IEEE 32nd Intl. Conf. on Distributed Computing Systems. pp. 615–626. IEEE (2012)
9. 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. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. pp. 346–353. Springer Berlin Heidelberg, Berlin, Heidelberg (2000)
10. Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J.J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., et al.: Open mpi: Goals, concept, and design of a next generation mpi implementation. In: European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting. pp. 97–104. Springer (2004)
11. Gamell, M., Katz, D.S., Kolla, H., Chen, J., Klasky, S., Parashar, M.: Exploring automatic, online failure recovery for scientific applications at extreme scales. In: SC'14: Proceedings of the Intl. Conf. for High Performance Computing, Networking, Storage and Analysis. pp. 895–906. IEEE (2014)
12. Georgakoudis, G., Guo, L., Laguna, I.: Evaluating the performance of global-restart recovery for mpi fault tolerance. Tech. rep., Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States) (2019)
13. Koren, I., Krishna, C.: Fault-Tolerant Systems. Elsevier Science (2020), https://books.google.com.br/books?id=YrnjDwAAQBAJ
14. Louca, S., Neophytou, N., Lachanas, A., Evripidou, P.: MPI-FT: Portable fault tolerance scheme for MPI. Parallel Processing Letters **10**(04), 371–382 (2000)
15. Moody, A., Bronevetsky, G., Mohror, K., De Supinski, B.R.: Design, modeling, and evaluation of a scalable multi-level checkpointing system. In: SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–11. IEEE, New Orleans, LA, USA (2010)
16. Nicolae, B., Moody, A., Gonsiorowski, E., Mohror, K., Cappello, F.: Veloc: Towards high performance adaptive asynchronous checkpointing at large scale. In: 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 911–920. IEEE, Rio de Janeiro, Brazil (2019)
17. Rosso, P.H.D.F., Francesquini, E.: Improved failure detection and propagation mechanisms for mpi. In: Anais da XII Escola Regional de Alto Desempenho de São Paulo. pp. 45–48. SBC (2021)
18. da Silva, J.A., 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. MGC '11, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/2089002.2089006, https://doi.org/10.1145/2089002.2089006
19. Sultana, N., Skjellum, A., Laguna, I., Farmer, M.S., Mohror, K., Emani, M.: Mpi stages: Checkpointing mpi state for bulk synchronous applications. In: Proc. of the 25th European MPI Users' Group Meeting. pp. 1–11 (2018)
20. Teranishi, K., Heroux, M.A.: Toward local failure local recovery resilience model using mpi-ulfm. In: Proc. of the 21st european mpi users' group meeting. pp. 51–56 (2014)
21. Young, J.W.: A first order approximation to the optimum checkpoint interval. Communications of the ACM **17**(9), 530–531 (1974)
22. Zhong, D., Bouteiller, A., Luo, X., Bosilca, G.: Runtime level failure detection and propagation in hpc systems. In: Proc. of the 26th European MPI Users' Group Meeting. pp. 1–11 (2019)