MATCH: An MPI Fault Tolerance Benchmark Suite

Luanzheng Guo[†], Giorgis Georgakoudis[‡], Konstantinos Parasyris[‡], Ignacio Laguna[‡] and Dong Li[†]
[†] *Electrical Engineering and Computer Science*, University of California, Merced, USA,
{lguo4, dli35}@ucmerced.edu
[‡] *Center for Advanced Scientific Computing*,
Lawrence Livermore National Laboratory, USA,
{georgakoudis1, parasyris1, lagunaperalt1}@llnl.gov

***Abstract*—MPI has been ubiquitously deployed in flagship HPC systems aiming to accelerate distributed scientific applications running on tens of hundreds of processes and compute nodes. Maintaining the correctness and integrity of MPI application execution is critical, especially for these safety-critical scientific applications. Therefore, a collection of effective MPI fault tolerance techniques have been proposed to enable MPI application execution to efficiently resume the application execution and states in the occurrence of system failures. However, there is not a structured way to study and compare different MPI fault tolerance designs, and to guide the selection and development of efficient MPI fault tolerance techniques in distinct scenarios. To solve this problem, we design, develop, and evaluate a benchmark suite MATCH to characterize, research, and comprehensively compare different combinations and configurations of MPI fault tolerance designs. Our investigation derives useful findings: (1) Reinit recovery achieves better performance efficiency than ULFM recovery; (2) Reinit recovery is independent of the scaling size and the input problem size, however ULFM recovery is not; (3) Using Reinit recovery with FTI checkpointing is a highly efficient fault tolerance design. MATCH open-source is available at https://github.com/kakulo/MPI-FT-Bench.**

I. Introduction

As supercomputers continue to increase computational power and size, next-generation HPC systems are expected to incur a much higher failure rate than contemporary systems. For example, the Sequoia supercomputer located in Lawrence Livermore National Laboratory (LLNL) reported a mean time between node failures to be 19.2 hours in 2013 [1]. After that, in 2014 the Blue Waters supercomputer reported a mean time between node failures to be 6.7 hours [2]. Most recently, the Taurus system located in TU Dresden reported a mean time between node failures to be 3.65 hours [3].

This trend raises the concerns in the HPC community for MPI applications running on tens of thousands of processes and nodes to fail when facing an increasing number of process and node failures. Process and node failures frequently occur in production HPC systems due to power outages and other issues. MPI process and node failures are fail-stop failures. In this type of failure, the application execution cannot continue without a repairing to the communication and has to stop.

These crucial facts lead to an increasing importance of and challenges for developing efficient and effective fault tolerance designs for scaling HPC systems. There are numerous fault tolerance techniques proposed to protect MPI application execution from system failures. These MPI fault tolerance techniques can be assigned into two types with different focuses. Checkpointing [4], [5], [6], [7], [8], commonly used in HPC applications, is one type of fault tolerance technique that focuses on restoring application state. Checkpointing takes places in two separate phases: storing the system state and recovering from it in case of a failure. Checkpointing helps MPI applications to quickly restore application state from the latest checkpoints through saving application execution state periodically. The other type of MPI fault tolerance technique focuses on restoring MPI state. Restarting is a baseline solution for restoring MPI state, which immediately restarting an application after execution collapses due to a failure. Later, because of the inefficiency of restarting an application, HPC practitioners propose MPI recovery mechanisms to restore MPI state online. User-level Fault Mitigation (ULFM) [9] and Reinit [10], [11], [12] are the two pioneer MPI recovery frameworks in this effort. ULFM supports a wide range of recovery strategies including local forward recovery and global-restart recovery, whereas Reinit only supports global-restart recovery. ULFM is a powerful MPI recovery framework, but complicated to use. Reinit needs much less programming effort.

However, there is not an existing framework that enables a comprehensive comparison between different MPI fault tolerance techniques. To solve the problem, we design and develop a benchmark suite MATCH aiming to study the performance efficiency of a variety of MPI fault tolerance configurations. MATCH contains six proxy applications from the Exascale Computing Project (ECP) Proxy Apps Suite and LLNL Advanced Simulation and Computing (ASC) proxy application suite; MATCH uses Fault Tolerance Interface (FTI) for data recovery and uses ULFM and Reinit for MPI recovery. We pick a representative set of HPC applications, but our methodology is extensible to more HPC applications. In evaluation, we break down the execution time and compare the performance overhead, when using FTI with Restart, when using FTI with ULFM, and when using FTI with Reinit, respectively. All the above experiments are running in four different scaling sizes (64 processes, 128 processes, 256 processes, and 512 processes on 32 nodes), in three different input sizes (small, median, and large), and when with or without injecting process failures.

In particular, our contributions are:

1) We present MATCH, an MPI fault tolerance benchmark suite. This is the first benchmark suite designed to evaluate

multiple fault tolerance techniques in MPI. We illustrate the process and manifest the details of implementing a range of different fault tolerance designs to HPC proxy applications.

2) We develop a data dependency analysis tool for identifying the data objects for checkpointing, which are the only data objects necessary to guarantee the restoring of application state for the application execution correctness for the first time.
3) We comparatively and extensively investigate the performance efficiency of different configurations and different combinations of fault tolerance designs. Our evaluation reveals that, for MPI global-restart recovery, using FTI with Reinit is the most efficient design within the three evaluated fault tolerance designs, and Reinit recovery is 4 times faster than ULFM recovery on average, and 16 times faster than restarting on average.

II. Background

A. MATCH

There is not an existing benchmark suite aiming at benchmarking of MPI fault tolerance. We design, implement, and test a benchmark suite MATCH to understand, study, and comparatively evaluate the performance efficiency of different MPI fault tolerance designs and configurations. MATCH is composed of HPC proxy applications coming from representative HPC benchmark suites. MATCH contains six representative HPC applications. Our fault tolerance design has two interfaces: the checkpointing interface to preserve and protect the data, and the failure recovery interface to protect and repair the MPI communicator. We use the Fault Tolerance Interface (FTI) for checkpointing and Restart, ULFM and Reinit for MPI process recovery in this work.

B. Workloads

Our workloads are proxy applications getting from well-known benchmark suites: ECP proxy applications suite [13] and LLNL ASC proxy applications suite [14]. Proxy applications are small and simplified applications that allow HPC practitioners, operators, and domain scientists to explore and test key features of real applications with a quick turnaround. Our workloads represent the most important HPC application domains in scientific computing, such as iterative solvers, multi-grid, molecular dynamics, etc. We describe the six proxy applications used in MATCH below.

`AMG`: An algebraic multi-grid solver dealing with linear systems in unstructured grids problems. AMG is built on top of the BoomerAMG solver of the HYPRE library which is a large-scale linear solver library developed at LLNL. AMG provides a number of tests for a variety of problems. The default one is an anisotropy problem in the Laplace domain.

`CoMD`: A proxy application in Molecular Dynamics (MD) commonly used as a research platform for particle motion simulation. Different than previous MD proxy applications such as miniMD, the design of CoMD is significantly modularized which allows performing analyses individual modules.

`LULESH`: A proxy application that solves the hydrodynamics equation in a Sedov blast problem. LULESH solves the hydrodynamics equation separately by using a mesh to simulate the Sedov blast problem which is divided into a composition of volumetric elements. This mesh is an unstructured hex mesh, where nodes are points connected by mesh lines.

`miniFE`: A proxy application that solves unstructured implicit finite element problem. miniFE aims at the approximation of an unstructured implicit finite element.

`miniVite`: A proxy application that solves the graph community detection problem using the distributed Louvain method. The Louvain method is a greedy algorithm for the community detection problem.

`HPCCG`: A preconditioned conjugate gradient solver that solves the linear system of partial differential equations in a 3D chimney domain. HPCCG approximates practical physical applications that simulate unstructured grid problems.

C. Checkpointing Interface - FTI

Fault Tolerance Interface (FTI) [15] is a multi-level checkpointing interface for efficient multilevel checkpointing in large-scale high-performance computing systems. FTI provides programmers a number of APIs which are easy to use, and allows programmers to choose checkpointing strategy that fits the application. FTI enables multiple levels of reliability with different performance efficiency by utilizing local storage, data replication, and erasure codes. FTI is an application-level checkpointing. It requests users to decide which data objects to be checkpointed. Furthermore, FTI hides data processing details from users. Users only tell FTI the memory address and data size of the date object to be protected to enable checkpointing of the data object. Because failures can corrupt single or multiple nodes during the execution of an application, FTI provides multiple levels of resiliency to recover from failures of different severities. Namely the levels are the following:

- L1: This level stores checkpoints locally to each compute node. In case of a node failure, the application states cannot successfully restore.
- L2: This level is built on top of L1 checkpointing. In this level each application stores their checkpoint locally as well as to a neighboring node.
- L3: In this level, the checkpoints are encoded by the Read-Solomon (RS) erasure code. This implementation can survive the breakdown of half of the nodes. The lost data can be restored from the RS-encoded files.
- L4: This level flushes checkpoints to parallel file system. This level enables differential checkpointing.

FTI have proposed a multi-level checkpointing model, and have conducted an extensive study of correctness and reliability of the proposed checkpointing model. In our work, we use FTI in the context of MPI recovery which is for the first time.

D. Failure Recovery Interface - ULFM and Reinit

MPI failure recovery has multiple modes including *global*, *local*, *backward*, *forward*, *shrinking*, and *non-shrinking*.

Global: The application execution must roll back to a global state to fix a failure.

Local: The application can continues the execution by repairing the failed components such as a failed code block locally without starting over the execution.

Backward: The application execution must go back to a previous state in order to survive a failure.

Forward: The failure can be fixed with the current application state, and the execution can continue.

Non-shrinking: The application manages to bring all failed processes back to resume execution.

Shrinking: The application execution is able to continue with the remaining survivor processes.

We target global, backward, non-shrinking recovery in this work. Because the global, backward, non-shrinking recovery fits best into the Bulk Synchronous Parallel (BSP) paradigm of HPC applications.

1) ULFM: User-level Fault Mitigation (ULFM) [9] is a leading MPI failure recovery framework providing shrinking recovery and non-shrinking recovery. ULFM develops new MPI operations to add fault tolerance functionalities at the application level. These functionalities include fault detection, communicator repairing, and failure recovery. In particular, ULFM leverages the MPI error handler to notify process failures. Once a failure is detected and notified, ULFM uses an operation MPI_Comm_revoke() to revoke processes in the communicator. This operation interrupts communication pending on the communicator at all processes. ULFM then reduces the failed processes using an operation MPI_Comm_shrink(), which also creates a new communicator with survivor processes. ULFM then makes an agreement among processes of the new communicator. The shrinking recovery is done using the above steps. The other recovery mode is non-shrinking recovery. For non-shrinking recovery, ULFM further uses the MPI_Comm_spawn() operation to spawn new processes and create a new communicator. ULFM then uses the MPI_Intercomm_merge() operator to merge the communicator of survivor processes and the communicator of spawned processes, and create a new communicator. We provide an example implementation of ULFM non-shrinking recovery in the Appendix.

2) Reinit: Reinit [11], [16], [12] is an alternative recovery framework designed particularly for global backward non-shrinking recovery. Reinit implements the recovery process into the MPI runtime, which is transparent from users. Therefore, the programming effort of using Reinit is much less than using ULFM. Programmers only need to set a global restarting point, the remaining recovery is done by Reinit. Also, Reinit is much more efficient than ULFM because of Reinit recovery transparently handled in the MPI runtime [12], whereas ULFM recovery is handled not only in the MPI runtime but also in the application.

III. Design

We present the design details in this section. In particular, we describe the algorithm that we use to find data objects for checkpointing through data dependency analysis.

A. Find Data Objects for Checkpointing

Different than many fault tolerance frameworks that request programmers to decide data objects for checkpointing, we develop a practical analytic tool to guide programmers to identify data objects to be checkpointed, in order to recover the application execution to the same state as before the failure. We identify data objects for checkpointing through data dependency analysis across iterations following three **principles**.

1) The data objects for checkpointing across iterations must be defined before the iterative computation. Data objects defined locally within the main computation loop must be excluded for checkpointing.
2) The data objects for checkpointing must be used (read or written) across iterations of the main computation loop.
3) The value of data objects for checkpointing must vary across iterations of the main computation loop.

Following the three principles, we design and develop the data dependency analysis tool. The **input** to the tool is a dynamic execution instruction trace generated using LLVM-Tracer [17]. The trace contains detailed information of dynamic operations, such as the register name and memory address, the operator, and the line number in the source code where the operation performs. We describe the algorithm of the data dependency analysis tool in Algorithm 1. The input to the algorithm is the set of locations used within the main computation loop, and the set of locations allocated before the main computation loop. Here locations are registers and memory locations. We create the two sets of locations by traversing the instruction trace once. After that, we first check values of locations, and make sure the invocation values of the same location within the main computation loop are different. We then remove repetitions from both sets of locations. Lastly, for each location in the set of the main computation loop we search for a match in the location set before the main computation loop. If a match is found, the matched location is used to localize data objects for checkpointing. The **output** of the tool is a set of locations for checkpointing. Note that the tool only outputs the locations for checkpointing, runs separately, and has not supported automatic generation of checkpointing code at this stage. We leave it for future work.

IV. Implementation

A. FTI Implementation

The Fault Tolerance Interface (FTI) is a checkpointing library widely used by HPC developers for checkpointing. We illustrate a sample usage of FTI in Figure 1. We find a challenge while implementing FTI to MATCH workloads.

The challenge is the programming complexity of enabling FTI checkpointing to data objects, when the number of data objects for checkpointing is large. FTI requests users to manually add FTI checkpointing to every data object. This significantly increases the programming effort when the number of data objects for checkpointing is large and when the data object is a complicated data structure. This is a common issue

Algorithm 1: Find Data Objects for Checkpointing

Input: $Locs\_in\_loop$: the set of locations used in the main computation loop; $Locs\_before\_loop$: the set of locations defined or allocated before the main computation loop

Output: CPK_Locs: the set of locations for checkpointing

```
// Check values of locations in Locs_in_loop
for l ∈ Locs_in_loop do
    if The invocation values of l are not the same then
        Keep l in Locs_in_loop;
    else
        Remove l from Locs_in_loop;
// Remove repetition in Locs_in_loop and Locs_before_loop
for l ∈ Locs_in_loop do
    Remove repetition;
for l ∈ Locs_before_loop do
    Remove repetition;
// Check if locations in Locs_in_loop can find a match in Locs_before_loop
for l_i ∈ Locs_in_loop do
    for l_j ∈ Locs_before_loop do
        if l_i matches l_j then
            CPK_Locs ← l_i;
```

in application level checkpoint libraries such as FTI, VeloC, and SCR. These libraries cannot automatically enable checkpointing to target data objects.

B. FTI with Reinit Implementation

Reinit is the state-of-the-art MPI global non-shrinking recovery framework. Reinit hides all recovery implementations to the MPI runtime, which makes it ease-to-use. We provide a sample implementation of Reinit with FTI checkpointing in Figure 2. We can see that Reinit recovery only adds less than five lines of code. Line 4 and 5 are for Reinit recovery, while Line 14 is used for other functionalities. FTI is completely independent of Reinit. To implement FTI with Reinit, the only thing to notice is to move the FTI_Init() and FTI_Finalize() functions into the resilient_main() function as well.

C. FTI with ULFM Implementation

ULFM is a pioneer MPI recovery framework. ULFM provides five new MPI interfaces to support MPI fault tolerance. ULFM gives flexibility to programmers to use the provided interfaces to implement the MPI recovery functionality. Also, ULFM allows programmers to use both shrinking and non-shrinking recovery. However, it takes a significant learning and programming effort before a programmer can successfully implement ULFM process recovery. As most HPC applications follow the Bulk Synchronous Parallel (BSP) paradigm, we

```
1  int main(int argc, char *argv[]) {
2      MPI_Init(&argc, &argv);
3
4      // Initialize FTI
5      FTI_Init(argv[1], MPI_COMM_WORLD);
6
7      // Right before the main computation loop
8      // Add FTI protection to data objects
9      FTI_Protect();
10
11     // the main computation loop
12     while (...) {
13         // At the beginning of the loop
14         // If the execution is a restart
15         if ( FTI_Status() != 0){
16             FTI_Recover();
17         }
18
19         // do FTI checkpointing
20         if (Iter_Num % cp_stride == 0) {
21             FTI_Checkpoint();
22         }
23     }
24
25     FTI_Finalize();
26     MPI_Finalize();
27 }
```

Fig. 1: A sample implementation of FTI.

```
1  int main(int argc, char *argv[])
2  {
3      MPI_Init(&argc, &argv);
4      OMPI_Reinit(argc, argv, resilient_main);
5      MPI_Finalize();
6      return 0;
7  }
8  // Move the original main() into resilient_main()
9  int resilient_main(int argc, char** argv,
       OMPI_reinit_state_t state) {
10     FTI_Init(argv[1], MPI_COMM_WORLD);
11     ...
12     // the main computation loop
13     ...
14     FTI_Finalize();
15     return 0;
16 }
```

Fig. 2: A sample implementation of Reinit.

focus on ULFM global non-shrinking recovery. In order to implement ULFM non-shrinking recovery, we add more than 200 lines of code for each benchmark, which is less efficient comparing to the implementing effort (less than five lines of code) for Reinit recovery. We provide a sample implementation of ULFM global non-shrinking recovery with FTI in Figure 3.

When combining ULFM global non-shrinking recovery with FTI, it is important to notice that the MPI_COMM_WORLD at Line 4 in Figure 1 must be implemented as a global variable with external declaration. Such that, the world communicator is immediately updated after repaired by ULFM recovery, and FTI is able to use the repaired world communicator for MPI

```c
/* world will swap between worldc[0] and worldc[1]
     after each respawn */
MPI_Comm worldc[2] = { MPI_COMM_NULL, MPI_COMM_NULL };
int worldi = 0;

//the MPI communicator must be implemented as a global
     variable to enable immediately update after ULFM
     recovery for FTI to use
#define world (worldc[worldi])

int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    // set long jump
    int do_recover = _setjmp(stack_jmp_buf);
    int survivor = IsSurvivor();
    /* set an errhandler on world, so that a failure is
         not fatal anymore */
    MPI_Comm_set_errhandler(world);
    FTI_Init(argv[1], world);
    ...
    // the main computation loop
    ...
    FTI_Finalize();
    MPI_Finalize();
}

/* error handler: repair comm world */
static void errhandler(MPI_Comm* pcomm, int* errcode,
    ...)
{
    int eclass;
    MPI_Error_class(*errcode, &eclass);

    if( MPIX_ERR_PROC_FAILED != eclass &&
        MPIX_ERR_REVOKED != eclass ) {
        MPI_Abort(MPI_COMM_WORLD, *errcode);
    }

    /* swap the worlds */
    worldi = (worldi+1)%2;

    MPIX_Comm_revoke(world);
    MPIX_Comm_shrink();
    MPI_Comm_spawn();
    MPI_Intercomm_merge();
    MPIX_Comm_agree();

    _longjmp( stack_jmp_buf, 1 );
}
```

Fig. 3: A sample implementation of ULFM non-shrinking recovery.

communication without incurring communication faults.

D. Fault Injection

We emulate MPI process failures through fault injection. In particular, we raise a SIGTERM signal at the selected MPI process in the selected iteration of the main computation loop. We illustrate the fault injection code in Figure 4. Note that we choose to evaluate different fault tolerance techniques by triggering a process failure, which does not mean that the MPI recovery frameworks do not support recovery in a node failure. Reported in a recent study [12], Reinit can recover in a node failure, while ULFM cannot. In our case, it is sufficient to

```c
// simulation of proc failures
if (procfi == 1 && numIters==Selected_Iter){
    if (myrank == Selected_Rank){
        printf("KILL rank %d\n", myrank);
        kill(getpid(), SIGTERM);
    }
}
```

Fig. 4: A sample implementation of fault injection.

evaluate on MPI process failures to compare the performance difference when using FTI checkpointing in ULFM and Reinit.

V. Evaluation

We seek for answers for a few questions in the analyses and discussion of the evaluation results with respect to fault tolerance efficiency.

- Can fault tolerance interfaces (such as ULFM) delay the application execution or not?
- Can the checkpointing interface and the MPI recovery interface interfere with each other?
- Can ULFM perform better or Reinit perform better in different scaling sizes and different input problem sizes?

A. Artifact Description

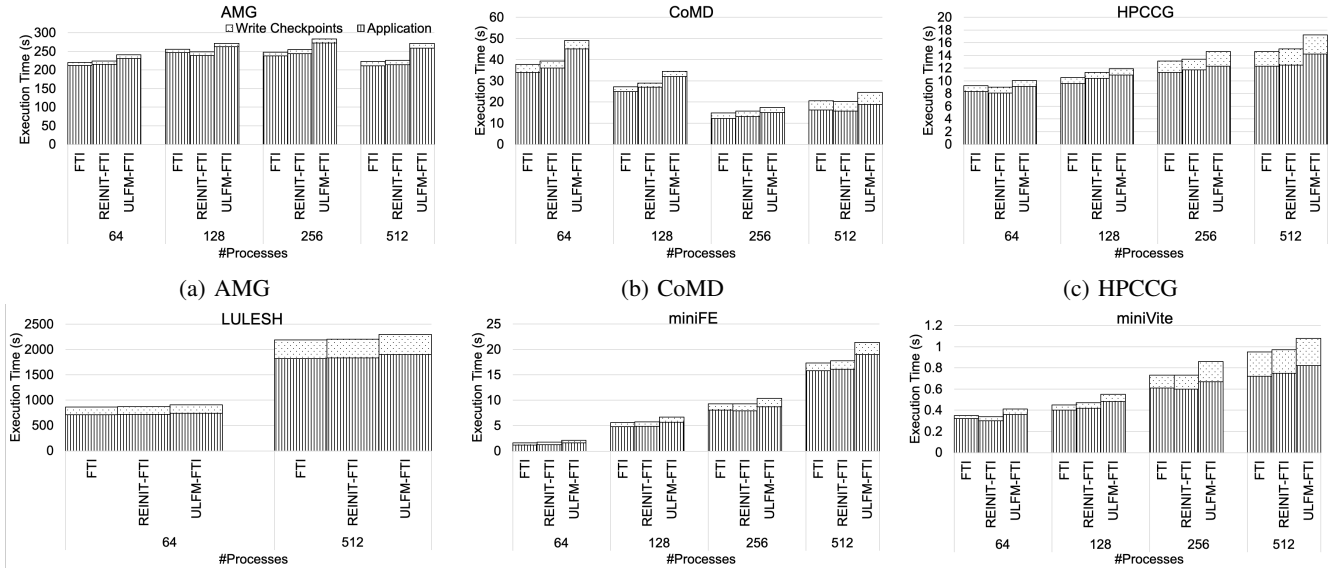
We run experiments on a large-scale HPC cluster having 752 nodes. Each node is equipped of two Intel Haswell CPUs, 28 CPU cores, 128 GB shared memory, and 8 TB local storage.

B. Experimentation Setup

This section provides the configuration details of the experimentation setup. We aim to test, evaluate, and compare the performance efficiency of different combinations and configurations of fault tolerance designs. In our experiments, we evaluate three fault tolerance designs. They are FTI checkpointing only, FTI checkpointing with ULFM recovery, and FTI checkpointing with Reinit recovery. "FTI checkpointing only" means that we restart the execution in a process failure for MPI recovery.

For FTI checkpointing, we use the L1 checkpointing mode. FTI L1 checkpointing allows users to store checkpoints to the local SSD or to do in-memory checkpointing. In our evaluation, we use the faster way that saves checkpoints to the local memory associated with the nodes in use using RAMFS through "/dev/shm". Although there are L1, L2, L3, and L4 modes for checkpointing, we do not evaluate all of them. The efficiency comparison between the four FTI checkpointing modes has been fully investigated in the FTI paper [15]. We save checkpoints every **ten** iterations. For ULFM, we use the latest version "ULFM v4.0.1ulfm2.1rc1" based on Open MPI 4.0.1. For Reinit, we use its latest version based on Open MPI 4.0.0.

We implement all the three fault tolerance designs to the MATCH benchmarks. Each evaluation is run on *three input problem sizes with the default scaling size (64 processes)* with and without fault injection. Also, each evaluation is run on *four scaling sizes (64 processes on 32 nodes, 128 processes on*



(a) AMG (b) CoMD (c) HPCCG

(d) LULESH (e) miniFE (f) miniVite

Fig. 5: Execution time breakdown recovering in different scaling sizes with no process failures

TABLE I: Experimentation configuration for proxy applications (**default scaling size: 64 processes; default input problem: small**)

Application	Small Input	Medium Input	Large Input	Number of processes
AMG	-problem 2 -n 20 20 20	-problem 2 -n 40 40 40	-problem 2 -n 60 60 60	64, 128, 256, 512
CoMD	-nx 128 -ny 128 -nz 128	-nx 256 -ny 256 -nz 256	-nx 512 -ny 512 -nz 512	64, 128, 256, 512
HPCCG	64 64 64	128 128 128	192 192 192	64, 128, 256, 512
LULESH	-s 30 -p	-s 40 -p	-s 50 -p	64, 512
miniFE	-nx 20 -ny 20 -nz 20	-nx 40 -ny 40 -nz 40	-nx 60 -ny 60 -nz 60	64, 128, 256, 512
miniVite	-p 3 -l -n 128000	-p 3 -l -n 256000	-p 3 -l -n 512000	64, 128, 256, 512

32 nodes, 256 processes on 32 nodes, and 512 processes on 32 nodes) with the default input problem size (small) with and without fault injection. We show the experimentation configuration in Table I. Note that LULESH needs to run on a cube number of processes. We can only run LULESH on 64 and 512 processes.

For fault injection, we choose a certain iteration and a certain process to inject a fault. This enables us to fairly compare the efficiency of different fault tolerance configurations.

Notably, we run experiment of each configuration for five times, and calculate the average execution time to avoid any system noise. We use '-O3' for mpicc or mpicxx compilation.

C. Performance Comparison on Different Scaling Sizes

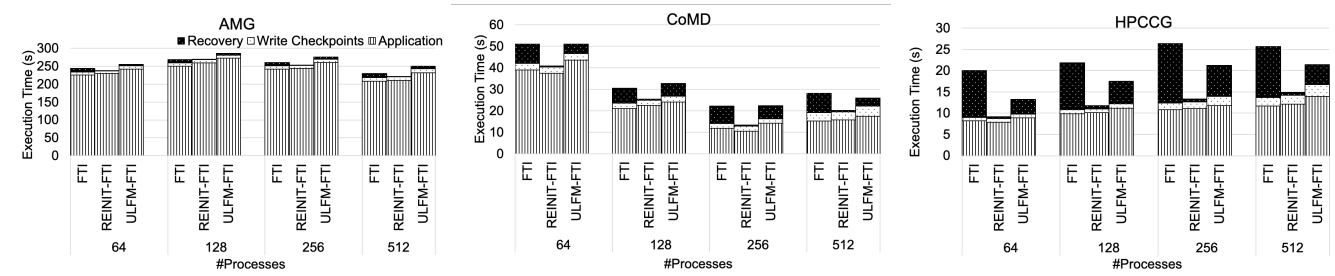
In this experiment, we run each evaluation on four scaling sizes with the default input problem size (small). We seek to compare the scaling efficiency of the three fault tolerance designs with and without process failures.

Without A Failure: Figure 5 shows the average execution time when no failure occurs. We break down the execution time to *the pure application execution time* and *the time for writing checkpoints*.

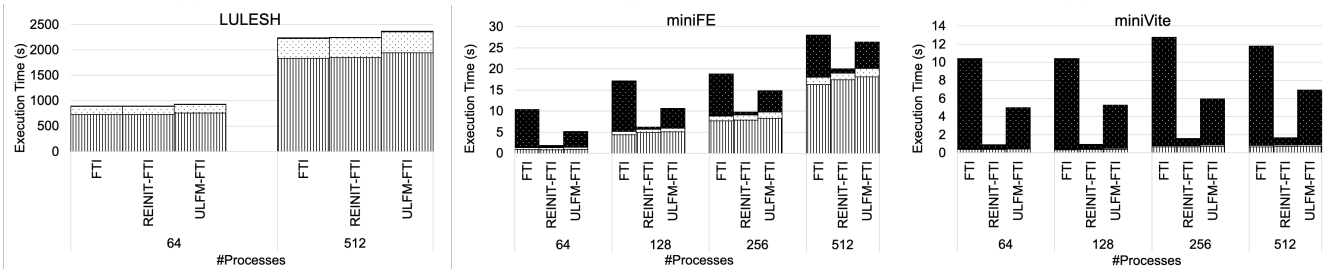
Overall, we can see that among the three fault tolerance designs, the FTI checkpointing with ULFM recovery case performs worst. The FTI checkpointing only and the FTI checkpointing with Reinit recovery perform similar and better than "ULFM-FTI".

We first observe that FTI L1 checkpointing scales well. The time spent on writing checkpoints gently increases with more processes. This verifies that there are a number of collective operations implemented in FTI L1 checkpointing. The average time for writing checkpoints is accounted for 13% of the total execution time.

Second, we observe that Reinit has no impact to application execution when there is no failure. We use the FTI application execution time as the baseline for comparison because FTI is an application-level checkpointing library, whereas ULFM and Reinit modify the MPI runtime. We can see that the *application execution time* of "REINIT-FTI" is very close to the *application execution time* of cases using FTI checkpointing only. However, the "ULFM-FTI" cases using ULFM recovery introduce some overhead to the application execution time. This overhead increases as the number of processes goes up. This is understandable. ULFM is known as a framework implemented

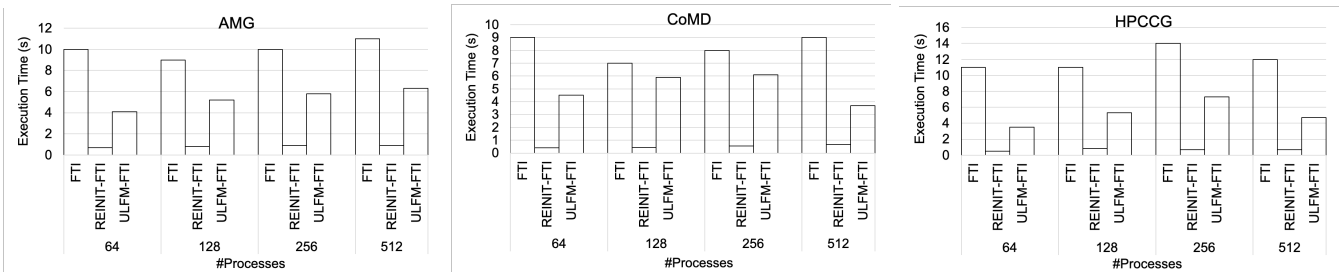


(a) AMG (b) CoMD (c) HPCCG

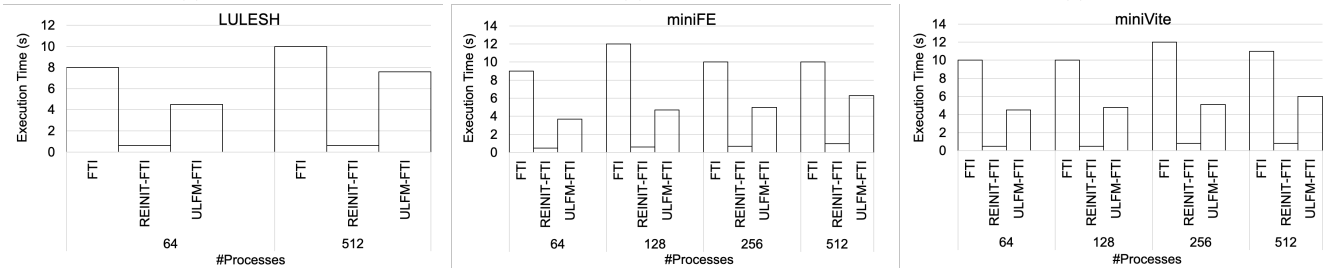


(d) LULESH (e) miniFE (f) miniVite

Fig. 6: Execution time breakdown recovering from a process failure in different scaling sizes



(a) AMG (b) CoMD (c) HPCCG



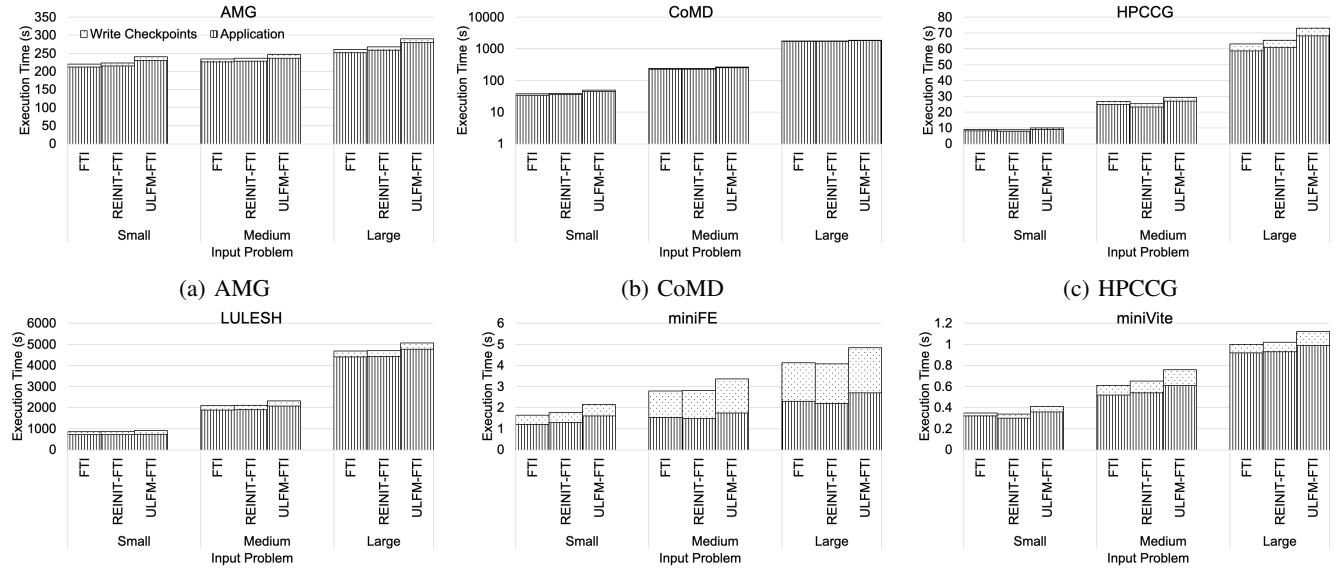
(d) LULESH (e) miniFE (f) miniVite

Fig. 7: Recovery time for different scaling sizes

across MPI runtime and application levels. It can introduce memory and communication latency to the application execution and further affect the application execution efficiency. As reported in a ULFM paper [18], ULFM implements a constantly heartbeat mechanism for failures detection, and also amends MPI communication interfaces for failure recovery operations. These changes must have impact on the application execution. Different than ULFM, Reinit incurs overhead only when a failure happens because it does not perform any other background operation in the MPI runtime during execution.

Furthermore, we observe that the times for writing checkpoints in FTI checkpointing only and "REINIT-FTI" cases are close. This indicates that Reinit has no interference on FTI checkpointing, yet ULFM has a small impact on FTI checkpointing in some cases such as HPCCG and miniVite. This is reasonable. Reinit implements the process recovery at the MPI runtime level, which has minimal impact on application-level operations, where the FTI operations run. Whereas ULFM does a significant amount of collective operations for periodic heartbeat in the MPI runtime, which leads to background overhead.

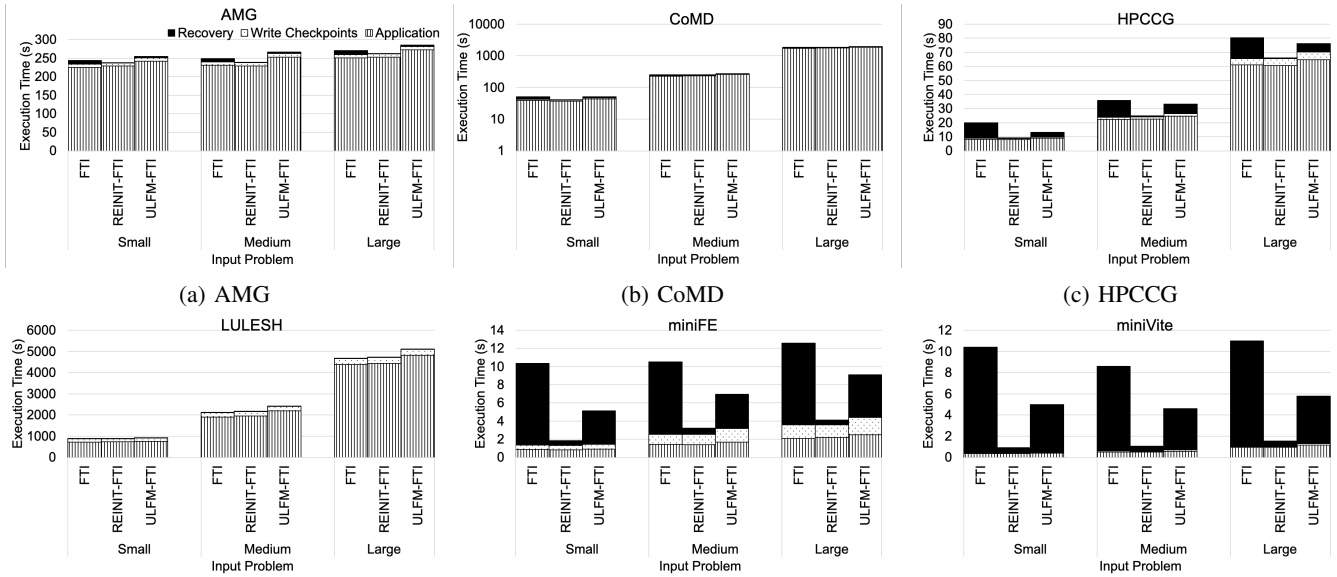
Conclusion 1. *"REINIT-FTI" cases achieve similar perfor-*



(a) AMG (b) CoMD (c) HPCCG

(d) LULESH (e) miniFE (f) miniVite

Fig. 8: Execution time breakdown in different input problem sizes with no process failures



(a) AMG (b) CoMD (c) HPCCG

(d) LULESH (e) miniFE (f) miniVite

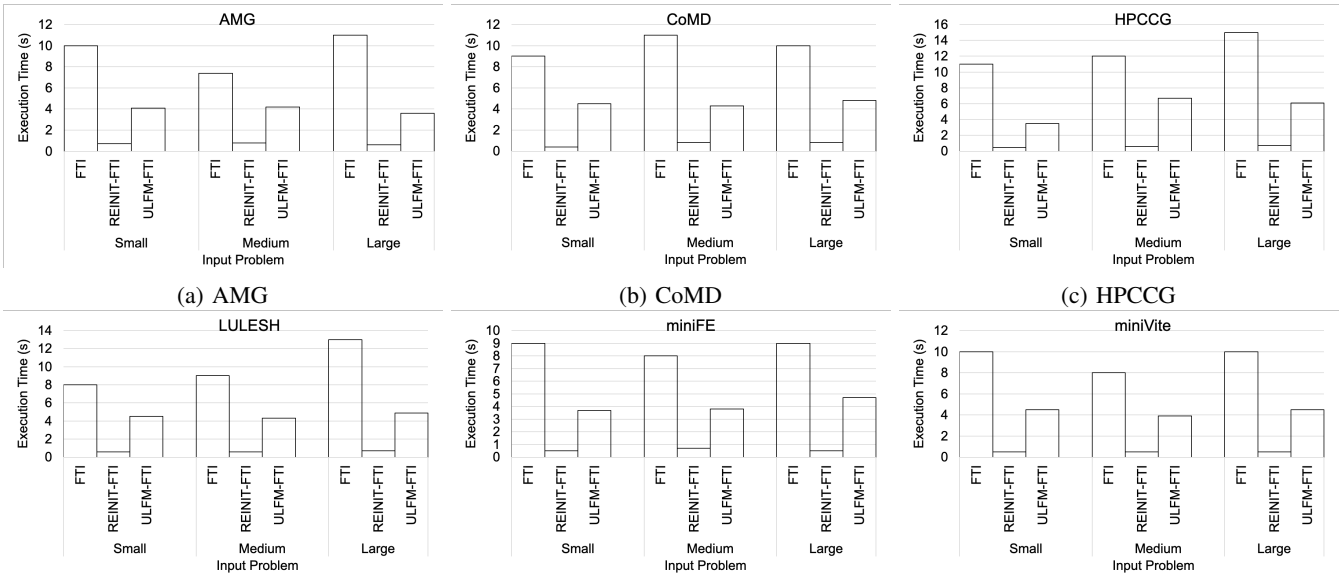
Fig. 9: Execution time breakdown recovering from a process failure in different input problem sizes

mance to "FTI checkpointing only" cases. This suggests using "REINIT-FTI" and "FTI checkpointing only" when there is no MPI process failure.

With A Failure: Figure 6 shows the breakdown of execution time recovering from a process failure on different scaling sizes. Note that reading checkpoints only happens once in the execution, and has values in the order of milliseconds, which is difficult to observe, and we exclude it from the figure. Figure 7 shows the MPI recovery time for different scaling sizes.

Overall, we observe that "REINIT-FTI" achieves the best performance compared to the other two cases "FTI checkpointing only" and "ULFM-FTI". There are two essential reasons. First, "REINIT-FTI" does not affect the performance of writing checkpoints. Second, Reinit recovery achieves the best performance for MPI recovery than restarting and ULFM recovery. We can make the similar observations we made from Figure 5. Furthermore, we can make new observations. First, we can compare the time of MPI recovery for cases using restarting, Reinit, and ULFM. Also, we can find that restarting and ULFM recovery are significantly slower than Reinit recovery in many cases.

ULFM recovery vs. Reinit Recovery. By observation, we



(a) AMG (b) CoMD (c) HPCCG

(d) LULESH (e) miniFE (f) miniVite

Fig. 10: Recovery time for different input problem sizes

find that the ULFM recovery time can be up to 13 times larger than Reinit recovery time, and 4 times larger on average. We can also see a trend that the ULFM recovery time increases as the number of processes increases, not scaling well. *Different than ULFM, after counting numbers, we find that Reinit recovery time looks constant in many cases and is independent of the number of processes.* This makes sense. ULFM enforces a variety of fault tolerance collective operations on all MPI processes to enable the MPI global non-shrinking recovery. Even worse, ULFM implements these fault tolerance operations at the application level, which needs to synchronize with other fault tolerance operations implemented at the MPI runtime. On the contrast, Reinit is implemented at the MPI runtime level, which requests much fewer collective operations.

Restarting vs. Reinit recovery. By calculation, we find that the restarting recovery can be up to 22 times slower than Reinit recovery, and 16 times slower on average. This is acceptable. Redeployment of the MPI setup and allocation of resources for restarting the execution is very expensive. Whereas Reinit recovery repairs the MPI state online.

Restarting vs. ULFM recovery. Restarting recovery is 2 to 3 times slower than ULFM recovery. Similarly, ULFM recovery is online recovery, which is much more efficient than redeployment.

Conclusion 2. "REINIT-FTI" outperforms "FTI checkpointing only" and "ULFM-FTI" in case of a failure. This suggests to use "REINIT-FTI" for MPI fault tolerance.

D. Performance Comparison on Different Input Sizes

In this experiment, we perform the performance comparison of three fault tolerance designs on three input problem sizes with the default scaling size (64 processes), with and without fault injection. Each configuration runs for five times, and we count the average of the five runs to avoid any system noise.

Without A Failure: Figure 8 presents the results of application execution in different input problem sizes with no process failures. The execution time is divided into the time of writing checkpointing and pure application execution time. We make several observations. Again, we use the pure application execution time of FTI as the baseline for comparison.

First, we can see an increment on the pure application execution time and FTI checkpointing time when running on larger input problem sizes because the amount of data to process increases.

We can also observe the performance latency in application execution time in "ULFM-FTI" cases using ULFM recovery. This latency increases as the input problem size grows. This indicates that ULFM is intensively involved into the application execution, where ULFM fault tolerance operations run a large number of collective MPI operations. These inefficient operations significantly affect the application execution, causing a huge communication latency, especially when there is a large amount of data to process and communicate. Different than ULFM, Reinit does not delay the application execution. We can observe that the application execution time of "REINIT-FTI" cases is very close to the execution time of the "FTI checkpointing only" cases. This is expected as Reinit is implemented in the MPI runtime. Also, Reinit uses much fewer collective operations than ULFM used.

With A Failure: Figure 9 shows the results of execution time breakdown when recovering from a process failure in different input problem sizes. Note that we omit the time of reading checkpoints because it is in the order of milliseconds. Also, Figure 10 shows the recovery time for different input problem sizes.

From the results, we can make the same observation we make through Figure 8 and results of the scaling experimentation.

However, the new observation is that, after counting numbers, we find that either the recovery times of ULFM or Reinit only has a negligible change when the input problem sizes increase. This is an interesting finding, but makes sense. When a failure occurs, ULFM starts collecting messages among daemons and processes, which cannot be affected by application execution because the application stops computing and communicating data. Reinit is fully implemented in the MPI runtime, which is even more difficult to be affect. We find that ULFM and Reinit process recovery are independent of input problem size.

Conclusion 3. *Through the performance comparison results on different input sizes, we again find that "REINIT-FTI" is the most efficient design within the three fault tolerance designs.*

VI. Related Work

Data Recovery. Checkpointing [19], [20], [21], [22], [23], [24], [25], [26] is the commonly used approach to restart an MPI application when a failure occurs. Programmers need to have a good sense of the application algorithm and the code structure before they can pinpoint which date objects for checkpointing. On ther other hand, writing checkpoints to the file system typically brings at least 20% percent performance overhead. There are many works trying to make checkpointing easier-to-use and to improve checkpointing efficiency.

Hargrove et al. [19] develop a system-level checkpointing library–the Berkeley Lab Checkpoint/Restart (BLCR) library– to run checkpointing at system-level using the Linux kernel. Furthermore, Adam et al. [21], SCR [27], and FTI [28] propose multi-level checkpointing aiming to significantly advance checkpointing efficiency. CRAFT [29] provide a fault tolerance framework that integrates checkpointing to ULFM shrinking and non-shrinking recovery. In this work, we choose FTI for checkpointing for data recovery because the high efficiency and well documenting of FTI. We attempt to integrate and evaluate more checkpointing mechanisms in addition to FTI in future work. Furthermore, different than existing works, we also provide a data dependency analytics tool to aid programmers to identify data objects for checkpointing.

MPI Recovery. ULFM [9], [30] is a leading MPI recovery framework that is in progress with the MPI Fault Tolerance Working Group. ULFM provides new MPI interfaces to remove failed processes and add new processes to communicators, and to agree between processes. ULFM requests programmers to implement shrinking- or non-shrinking recovery using these interfaces. ULFM provides flexibility to programmers, but there is a great effort of learning before programmers can correctly use ULFM interfaces to implement ULFM recovery. A large number of works [31], [32], [33], [4], [5], [6], [8] have explored and extended the applicability of ULFM. Teranishi et al. [34] replace failed processes with spare processes to accelerate ULFM process recovery. Bosilca et al. [35], [18] and Katti et al. [36] propose a series of efficient fault detection mechanisms for ULFM. Fenix [37] provides a user-friendly abstraction layer on top of ULFM. Fenix reduces the effort to implement ULFM recovery, but it does not solve the scalability problems of ULFM reported by previous works [34], [38], also demonstrated in our evaluation.

Reinit [39], [12] is a more efficient solution for global recovery. Reinit hides the process recovery from programmers by implementing it to MPI runtime. Reinit provides only one interface to programmers which sets up the global restart point, protects the target function, and returns the state of spawned and survivor processes. The early versions [11], [16], [39], [40] of Reinit have limited usage because these versions are not compatible with common job schedulers. Most recently, Georgakoudis et al. [12] fix the design and reimplement Reinit into the OpenMPI runtime.

Later, researchers realize the efficiency of combining the two aspects to achieve higher efficiency of MPI fault tolerance. For example, FENIX [37] and CRAFT [29] both design and develop a checkpointing interface that supports data recovery for ULFM shrinking and non-shrinking recovery. However, developers must explicitly manage and redistribute the restored data among survivor processes in case of a non-shrinking recovery. This can easily cause load imbalance problems. Also, they only evaluate their frameworks on two applications, and do not compare their fault tolerance frameworks to other fault tolerance designs. For example, using Reinit for recovery, and testing other checkpointing interfaces. In conclusion, there is not an existing work that either benchmarks the design and implementation of MPI fault tolerance, or comprehensively compare the performance efficiency of different combinations of fault tolerance mechanisms and different configurations of fault tolerance designs.

Different than FENIX and CRAFT, we evaluate and comprehensively compare fault tolerance designs that combine FTI checkpionting and MPI recovery frameworks (ULFM and Reinit) on a collection of a variety of representative HPC proxy applications.

MPI Fault Tolerance Benchmarking. There have been many benchmark suites [41], [42], [43] developed for performance modeling of programming models using MPI. SKaMPI [44] is an early benchmark suite that evaluates different implementations of MPI. Bureddy et al. [45] develop a benchmark suite to evaluate point-to-point, multi-pair, and collective MPI communication on GPU clusters. Dosanjh et al. [46] propose the first micro benchmark suite to study the multi-threading Remote Memory Access performance in MPI. However, there is not a MPI benchmark suite that focuses on fault tolerance and evaluates fault tolerance designs in MPI. This paper proposes a benchmark suite MATCH for benchmarking MPI fault tolerance.

VII. Conclusions

MPI fault tolerance is becoming an increasingly critical problem as supercomputers continue to grow in size and add new components. We have designed and implemented a benchmark suite MATCH with an emphasis on MPI fault tolerance. Our benchmark suite has six representative HPC proxy applications selected from flagship benchmark suites. We comprehensively evaluate and compare the performance

efficiency of the three fault-tolerance designs we implement into the six workloads. The evaluation results reveal that FTI checkpointing with Reinit recovery is the most efficient fault tolerance design within the three designs. Our analytics and insights will inspire future MPI fault tolerance designs.

VIII. Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-812453). This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This research is partially supported by U.S. National Science Foundation (CNS-1617967, CCF-1553645 and CCF-1718194). We wish to thank the Trusted CI, the NSF Cybersecurity Center of Excellence, NSF Grant Number ACI-1920430, for assisting our project with cybersecurity challenges.

References

[1] J. Dongarra, "Emerging heterogeneous technologies for high performance computing," in *International Heterogeneity in Computing Workshop*, 2013.

[2] C. Di Martino, Z. Kalbarczyk, R. K. Iyer, F. Baccanico, J. Fullop, and W. Kramer, "Lessons learned from the analysis of system failures at petascale: The case of blue waters," in *IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2014.

[3] S. Ghiasvand, F. M. Ciorba, R. Tschüter, and W. E. Nagel, "Lessons learned from spatial and temporal correlation of node failures in high performance computers," in *Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE, 2016.

[4] A. Katti, G. Di Fatta, T. Naughton, and C. Engelmann, "Scalable and fault tolerant failure detection and consensus," in *Proceedings of the 22nd European MPI Users' Group Meeting*, 2015, p. 13.

[5] T. Herault, A. Bouteiller, G. Bosilca, M. Gamell, K. Teranishi, M. Parashar, and J. Dongarra, "Practical scalable consensus for pseudo-synchronous distributed systems," in *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, pp. 1–12.

[6] A. Bouteiller, G. Bosilca, and J. J. Dongarra, "Plan b: Interruption of ongoing mpi operations to support failure recovery," in *Proceedings of the 22nd European MPI Users' Group Meeting*, 2015, p. 11.

[7] M. M. Ali, P. E. Strazdins, B. Harding, and M. Hegland, "Complex scientific applications made fault-tolerant with the sparse grid combination technique," *The International Journal of High Performance Computing Applications*, vol. 30, no. 3, pp. 335–359, 2016.

[8] I. Laguna, D. F. Richards, T. Gamblin, M. Schulz, and B. R. de Supinski, "Evaluating user-level fault tolerance for mpi applications," in *Proceedings of the 21st European MPI Users' Group Meeting*, ser. EuroMPI/ASIA '14. New York, NY, USA: ACM, 2014, pp. 57:57–57:62. [Online]. Available: http://doi.acm.org/10.1145/2642769.2642775

[9] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra, "Post-failure recovery of mpi communication capability: Design and rationale," *The International Journal of High Performance Computing Applications*, vol. 27, no. 3, pp. 244–254, 2013.

[10] I. Laguna, D. F. Richards, T. Gamblin, M. Schulz, B. R. de Supinski, K. Mohror, and H. Pritchard, "Evaluating and extending user-level fault tolerance in mpi applications," *The International Journal of High Performance Computing Applications*, vol. 30, no. 3, pp. 305–319, 2016.

[11] S. Chakraborty, I. Laguna, M. Emani, K. Mohror, D. K. Panda, M. Schulz, and H. Subramoni, "Ereinit: Scalable and efficient fault-tolerance for bulk-synchronous mpi applications," *Concurrency and Computation: Practice and Experience*, vol. 0, no. 0, p. e4863, e4863 cpe.4863. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4863

[12] G. Georgakoudis, L. Guo, and I. Laguna, "Reinit++: Evaluating the performance of global-restart recovery methods for mpi fault tolerance," in *ISC*, 202O.

[13] D. Richards, O. Aaziz, J. Cook, S. Moore, D. Pruitt, and C. Vaughan, "Quantitative performance assessment of proxy apps and parentsreport for ecp proxy app project milestone adcd-504-9," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2020.

[14] J. R. Neely and B. R. de Supinski, "Application modernization at llnl and the sierra center of excellence," *Computing in Science & Engineering*, 2017.

[15] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka, "Fti: high performance fault tolerance interface for hybrid systems," in *International conference for high performance computing, networking, storage and analysis (SC)*, 2011.

[16] I. Laguna, D. F. Richards, T. Gamblin, M. Schulz, B. R. de Supinski, K. Mohror, and H. Pritchard, "Evaluating and extending user-level fault tolerance in mpi applications," *The International Journal of High Performance Computing Applications*, vol. 30, no. 3, pp. 305–319, 2016. [Online]. Available: https://doi.org/10.1177/1094342015623623

[17] Y. S. Shao and D. Brooks, "ISA-Independent Workload Characterization and its Implications for Specialized Architectures," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2013.

[18] G. Bosilca, A. Bouteiller, A. Guermouche, T. Herault, Y. Robert, P. Sens, and J. Dongarra, "A failure detector for hpc platforms," *The International Journal of High Performance Computing Applications*, vol. 32, no. 1, pp. 139–158, 2018. [Online]. Available: https://doi.org/10.1177/1094342017711505

[19] P. H. Hargrove and J. C. Duell, "Berkeley lab checkpoint/restart (blcr) for linux clusters," in *Journal of Physics: Conference Series*, vol. 46, no. 1, 2006, p. 494.

[20] S. Sankaran, J. M. Squyres, B. Barrett, V. Sahay, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman, "The lam/mpi checkpoint/restart framework: System-initiated checkpointing," *JHPCA*, vol. 19, no. 4, pp. 479–493, 2005.

[21] J. Adam, M. Kermarquer, J.-B. Besnard, L. Bautista-Gomez, M. Pérache, P. Carribault, J. Jaeger, A. D. Malony, and S. Shende, "Checkpoint/restart approaches for a thread-based mpi runtime," *Parallel Computing*, vol. 85, pp. 204–219, 2019.

[22] O. Subasi, T. Martsinkevich, F. Zyulkyarov, O. Unsal, J. Labarta, and F. Cappello, "Unified fault-tolerance framework for hybrid task-parallel message-passing applications," *The International Journal of High Performance Computing Applications*, vol. 32, no. 5, pp. 641–657, 2018.

[23] Z. Wang, L. Gao, Y. Gu, Y. Bao, and G. Yu, "A fault-tolerant framework for asynchronous iterative computations in cloud environments," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 8, pp. 1678–1692, 2018.

[24] J. Cao, K. Arya, R. Garg, S. Matott, D. K. Panda, H. Subramoni, J. Vienne, and G. Cooperman, "System-level scalable checkpoint-restart for petascale computing," in *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)*, 2016.

[25] J. Adam, J.-B. Besnard, A. D. Malony, S. Shende, M. Pérache, P. Carribault, and J. Jaeger, "Transparent high-speed network checkpoint/restart in mpi," in *Proceedings of the 25th European MPI Users' Group Meeting*, 2018, p. 12.

[26] N. Kohl, J. Hötzer, F. Schornbaum, M. Bauer, C. Godenschwager, H. Köstler, B. Nestler, and U. Rüde, "A scalable and extensible checkpointing scheme for massively parallel simulations," *The International Journal of High Performance Computing Applications*, vol. 33, no. 4, pp. 571–589, 2019.

[27] K. Mohror, A. Moody, G. Bronevetsky, and B. R. de Supinski, "Detailed modeling and evaluation of a scalable multilevel checkpointing system," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 9, pp. 2255–2263, Sep. 2014.

[28] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka, "Fti: High performance fault tolerance interface for hybrid systems," in *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2011, pp. 1–12.

[29] F. Shahzad, J. Thies, M. Kreutzer, T. Zeiser, G. Hager, and G. Wellein, "Craft: A library for easier application-level checkpoint/restart and automatic fault tolerance," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 3, pp. 501–514, 2018.

[30] W. Bland, H. Lu, S. Seo, and P. Balaji, "Lessons learned implementing user-level failure mitigation in mpich," in *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2015.

[31] N. Losada, I. Cores, M. J. Martín, and P. González, "Resilient mpi applications using an application-level checkpointing framework and ulfm," *The Journal of Supercomputing*, vol. 73, no. 1, 2017.
[32] S. Pauli, M. Kohler, and P. Arbenz, "A fault tolerant implementation of multi-level monte carlo methods," *Parallel computing: Accelerating computational science and engineering (CSE)*, vol. 25, pp. 471–480, 2014.
[33] A. Hori, K. Yoshinaga, T. Herault, A. Bouteiller, G. Bosilca, and Y. Ishikawa, "Sliding substitution of failed nodes," in *Proceedings of the 22nd European MPI Users' Group Meeting*. ACM, 2015, p. 14.
[34] K. Teranishi and M. A. Heroux, "Toward local failure local recovery resilience model using mpi-ulfm," in *Proceedings of the 21st european mpi users' group meeting*, 2014, p. 51.
[35] G. Bosilca, A. Bouteiller, A. Guermouche, T. Herault, Y. Robert, P. Sens, and J. Dongarra, "Failure detection and propagation in hpc systems," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 312–322.
[36] A. Katti, G. Di Fatta, T. Naughton, and C. Engelmann, "Epidemic failure detection and consensus for extreme parallelism," *The International Journal of High Performance Computing Applications*, vol. 32, no. 5, pp. 729–743, 2018.
[37] M. Gamell, D. S. Katz, H. Kolla, J. Chen, S. Klasky, and M. Parashar, "Exploring automatic, online failure recovery for scientific applications at extreme scales," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 895–906. [Online]. Available: https://doi.org/10.1109/SC.2014.78
[38] M. Gamell, K. Teranishi, M. A. Heroux, J. Mayo, H. Kolla, J. Chen, and M. Parashar, "Local recovery and failure masking for stencil-based applications at extreme scales," in *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, pp. 1–12.
[39] I. Laguna, D. F. Richards, T. Gamblin, M. Schulz, and B. R. de Supinski, "Evaluating user-level fault tolerance for mpi applications," in *Proceedings of the 21st European MPI Users' Group Meeting*, ser. EuroMPI/ASIA '14. New York, NY, USA: ACM, 2014, pp. 57:57–57:62. [Online]. Available: http://doi.acm.org/10.1145/2642769.2642775
[40] N. Sultana, M. Rüfenacht, A. Skjellum, I. Laguna, and K. Mohror, "Failure recovery for bulk synchronous applications with mpi stages," *Parallel Computing*, vol. 84, pp. 1 – 14, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819118303260
[41] J. M. Bull, J. P. Enright, and N. Ameer, "A microbenchmark suite for mixed-mode openmp/mpi," in *International Workshop on OpenMP*. Springer, 2009.
[42] P. R. Luszczek, D. H. Bailey, J. J. Dongarra, J. Kepner, R. F. Lucas, R. Rabenseifner, and D. Takahashi, "The hpc challenge (hpcc) benchmark suite," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006.
[43] T. Agarwal and M. Becchi, "Design of a hybrid mpi-cuda benchmark suite for cpu-gpu clusters," in *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*. IEEE, 2014.
[44] R. Reussner, P. Sanders, L. Prechelt, and M. Müller, "SKaMPI: A detailed, accurate MPI benchmark," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 1998.
[45] D. Bureddy, H. Wang, A. Venkatesh, S. Potluri, and D. K. Panda, "OMB-GPU: a micro-benchmark suite for evaluating MPI libraries on GPU clusters," in *European MPI Users' Group Meeting*. Springer, 2012.
[46] M. G. Dosanjh, T. Groves, R. E. Grant, R. Brightwell, and P. G. Bridges, "RMA-MT: a benchmark suite for assessing MPI multi-threaded RMA performance," in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, 2016.