Towards Ad Hoc Recovery For Soft Errors

Nuria Losada
Universidade da Coruña
A Coruña, Spain
nuria.losada@udc.es

Leonardo Bautista-Gomez, Kai Keller, Osman Unsal
Barcelona Supercomputing Center
Barcelona, Spain
{leonardo.bautista, kai.keller, osman.unsal}@bsc.es

***Abstract*—The coming exascale era is a great opportunity for high performance computing (HPC) applications. However, high failure rates on these systems will hazard the successful completion of their execution. Bit-flip errors in dynamic random access memory (DRAM) account for a noticeable share of the failures in supercomputers. Hardware mechanisms, such as error correcting code (ECC), can detect and correct single-bit errors and can detect some multi-bit errors while others can go undiscovered. Unfortunately, detected multi-bit errors will most of the time force the termination of the application and lead to a global restart. Thus, other strategies at the software level are needed to tolerate these type of faults more efficiently and to avoid a global restart. In this work, we extend the FTI checkpointing library to facilitate the implementation of custom recovery strategies for MPI applications, minimizing the overhead introduced when coping with soft errors. The new functionalities are evaluated by implementing local forward recovery on three HPC benchmarks with different reliability requirements. Our results demonstrate a reduction on the recovery times by up to 14%.**

***Index Terms*—Fault Tolerance, Checkpoint/Restart, ABFT.**

I. Introduction

Exascale offers great opportunities for HPC applications. Even though failures may be handled by resiliency techniques, they entail extended execution times, for instance due to a restart of the application. While the mean time to failure (MTTF) of one compute node could be in the order of a century, a machine with 100 000 nodes will statistically encounter a failure every 9 hours; furthermore, a machine consisting of 1 000 000 of those nodes will be hit by a failure every 53 minutes on average [1]. Comprehensive research has been performed studying those failures [2]. The study by "Di Martino et al." [3] have studied failures in the Cray supercomputer Blue Waters during 261 days, reporting that 1.53% of applications running on the machine failed because of system-related issues. The cost for electricity within this period that could have been avoided by introducing appropriate fault tolerance mechanisms in the applications, was estimated to be almost 0.5 million dollars. Future exascale systems will have tens of thousands of nodes and millions of cores, and they are expected to present high failure rates due to their scale and complexity. Following the taxonomy of Avižienis and others [4]–[6], faults such as physical defects, cause incorrect system states (i.e. errors). An error may lead to a failure when it causes the incorrect service of the system, i.e., an incorrect system's functionality and performance that can be externally perceived. Corrupted memory regions are among the most common root causes of failures, and various studies are focusing on DRAM and SRAM faults [7], [8]. Memory faults can be classified as *hard errors*, when bits are repeatedly corrupted due to a physical defect (e.g. bits permanently at "0" or "1"), and *soft errors*, which transiently corrupt bits [9]. Depending on whether memory errors can be detected and/or corrected, they are commonly classified in detectable correctable error (DCE), detectable uncorrectable error (DUE), and silent data corruption (SDC). DCEs are managed by the hardware and are oblivious to the applications. DUEs can lead to the interruption of the execution, while SDCs can lead to a scenario in which the application returns incorrect results but the user might not be aware of it. Most modern systems employ hardware mechanisms such as ECCs, parity checks or Chipkill-Correct ECC against data corruption in order to prevent SDCs and reduce DUEs [10]. The study by the Di Martino et al. study [3] shows the value of those protection mechanisms reducing failures in large supercomputers. However, even though the number of DUEs over the total machine check events is low, they still represent around 6.34% of the hardware errors leading to single/multiple node failures. Data also shows how GPU memory is 100 times more sensitive to uncorrectable errors because only ECC protection is used (i.e., no Chipkill). Therefore, software techniques are necessary to reduce the overhead caused by the remaining DUEs, which otherwise will trigger fail-stop failures.

Long-running scientific applications need to use software fault tolerance techniques to ensure their successful completion. Most popular parallel programming models, such as MPI, lack fault tolerance support and a fail-stop failure in one of the MPI processes results in the termination of the entire application. Traditional fault tolerant solutions for these applications rely on checkpoint/restart mechanisms: the application state is periodically saved into checkpoint files that allow the restart into intermediate states of the execution in case of failure. However, in this scheme all the running processes are terminated and need to be re-initialized and restarted. For large applications with several tens of thousands of processes, this can introduce high overheads. More efficient solutions can be implemented. Software mechanisms can prevent soft errors from causing fail-stop failures, and alternatives to coordinated rollback can be used to get the application to a global consistent state after a memory corruption. This is the case of local and/or forward recovery protocols.

Increasing the locality of the recovery process, i.e. restricting the resiliency actions to a subset of the application

© 2019 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works

processes, can reduce the failure recovery overhead. The same applies for forward recoveries, in which the application attempts to construct a new state to successfully continue the execution instead of using a previously checkpointed state and repeating computation already done, as in roll-back strategies. In this work, we extend the Fault Tolerance Interface (FTI) [11] checkpointing library to facilitate the implementation of flexible resilience strategies for MPI applications that allow the handling of soft errors during run time. We evaluate these software extensions on three different benchmarks, implementing forward local recovery protocols that reduce the failure overhead by 14%.

The rest of this paper is structured as follows. Section II provides an insight into related work. The FTI extensions to support forward recovery are presented in Section III. The integration of these new functionalities on three HPC benchmarks is described in Section IV. The experimental evaluation of the tested benchmarks is presented in Section V. Section VI comments on the applicability of these extensions and the implementation of recovery techniques and finally, Section VII concludes this paper.

II. Related Work

Checkpoint/restart techniques, such as [11]–[14], have been extensively studied in the past decades. These techniques usually target fail-stop failures, enabling the application's restart when a failure causes the interruption of execution and recovering the computation from the most recent set of checkpoint files. In large scale applications, failures usually affect a small part of the computational resources being used, thus, terminating and re-initializing all the application processes imposes unnecessary performance penalties. The User Level Failure Mitigation (ULFM) [15] corresponds to the most recent effort for the inclusion of resilience capabilities in the MPI standard, enabling applications to detect and react to failures without stopping their execution. Several works have implement resilient applications using the ULFM features [16]–[22]. ULFM enables the deployment of different recovery strategies after repairing the communication environment when a failure hits the application, thus, avoiding the overheads of re-initializing the entire MPI application.

Detectable soft errors related to memory corruption may be handled by the application before triggering a failure, and hence, avoiding the interruption of the execution when they arise. Some of these techniques are based on redundancy [23]–[25]. Comparing the results of the replicas enables error detection, and error correction in the case of triple-redundancy. However, replication requires substantially more hardware resources which rapidly becomes prohibitively expensive.

Other approaches exploit the characteristics of the particular application/algorithm by implementing ad hoc recovery techniques, which can introduce significant performance benefits in the recovery process. Traditionally, checkpoint/restart relies on a backward recovery, in which all processes in the application rollback to a previous committed state, and repeat the computation done from that point on. Increasing the checkpointing frequency reduces the amount of computation to be repeated, decreasing the failure overhead but increasing the checkpointing overhead in terms of performance, storage and bandwidth. Forward recovery strategies attempt to build a new application state from which the execution can resume, without rolling back to a past checkpointed state and repeating all the computation already done, thus, significantly reducing the recovery overhead. This is the case of partial re-computation [26] (which is focused on limiting the scope of the recomputation after a failure), and Algorithm Based Fault Tolerance (ABFT) techniques. ABTF was originally introduced by Huang and Abraham [27] to detect and correct permanent and transient errors on matrix operations. The method is based on the encoding of data at a high level and the design of algorithms to operate on the encoded data. ABFT has been used in combination with disk-less checkpointing for its usage in matrix operations [28], [29], and it has been implemented on algorithms such as the High Performance Linpack (HPL) benchmark [30], Cholesky factorization [31], algorithms using sparse matrices and dense vectors [32], and tasks based applications [33]. Another key aspect of the recovery process is the locality, i.e. the number of processes not affected by the error that need to be involved in the recovery. Restricting the recovery actions to only those processes affected by the error, or a subset of the processes (i.e., for those scenarios in which the failed processes cannot recover on their own, and require the participation of neighbour processes) also contributes towards the efficiency of the fault tolerance solution.

Generally, in order to tolerate fail-stop failures, multilevel checkpointing APIs enable programmers to save a subset of the application variables in different levels of the storage hierarchy at a specific frequency (called the checkpoint frequency). However, ad hoc recoveries, such as ABFT, may require access to past values of variables that differ from the ones checkpointed, and may require to do so at a different frequency to cope with soft errors. This work builds on top of those strategies exploiting the particularities of the application to boost the recovery and aims to provide a generic and intuitive way for applications to implement ad hoc recovery strategies. In particular, these new extensions are added to the FTI library in order to provide the necessary flexibility to facilitate the implementation of custom recovery mechanisms.

III. Extensions for Local/Forward Recovery

HPC applications that use any type of fault tolerance usually focus on fail-stop failures which terminate the execution of one or several processes running the application. The most widely used technique is checkpoint/restart, which enforces a global coordinated rollback to the last checkpoint upon a failure. Soft errors (i.e., errors affecting/corrupting part of the data in a non-permanent fashion) are usually handled in the same way. When a soft error is detected, the recovery process is determined by how the application has been protected against it. No protection will force the complete re-execution of the application from the beginning. Protecting the application with checkpoint/restart enables the recovery from the last

```
1  void sig_handler_sigdue(int signo){
2      FTI_RankAffectedBySoftError();
3      /* Algorithm recovery code */
4      [ ... ]
5  }
6  int main( int argc, char* argv[] )
7      /* SIGDUE: signal reporting a DUE */
8      signal(SIGDUE, sig_handler_sigdue);
9      /* Application initialization */ [ ... ]
10     for(i=0; i<N; i++){ /* Main loop */
11         int ret=FTI_Snapshot();
12         if (ret==FTI_SB_FAIL) {
13             int* status_array;
14             FTI_RecoverLocalVars( { ... } )
15             /* Algorithm recovery code */
16             [ ... ]
17         }
18         [ ... ]
19     }
20     [ ... ]
21 }
```

Fig. 1: Detection of a soft error.

checkpoint. This is inefficient because many of those soft errors could be tolerated without performing a global rollback, in which all processes running the application are recovered from the last checkpoint. Instead, they could be handled locally in a much more time and energy efficient way. Strategies such as ABFT allow the implementation of local-forward recovery, that exploit particularities of the application, to provide considerable performance boosts in the recovery process. In this work two types of soft errors are considered: i) DUEs, and ii) SDCs that are detected by software mechanisms [32], [34].

We provide a framework that allows developers to exploit the characteristics of their MPI applications in order to achieve a more efficient resilience strategy against soft errors. We leverage the FTI library [11] with an extended API for this purpose. FTI is an application-level multilevel checkpointing library that provides fault tolerance support by adding instrumentation blocks in the application code for: (i) marking those variables necessary for the recovery for their inclusion in the checkpoint files using the `FTI_Protect` routine, and (ii) inserting the `FTI_Snapshot` in the main loop of the application to generate checkpoint files.

We implement a signal handler to inform the local process about the occurrence of a soft error. This is achieved by the handling of the signal sent to the affected process by the operating system when a DUE occurs, or by means of software error/data-corruption detection (e.g., online data monitoring). The OS can provide information of the affected memory page which allows to derive the affected variables. In any case, the MPI process handles the error and triggers a recovery (example code shown in Figure 1). Once the soft error is locally detected, the process will notify FTI of the error event by means of the extended API function `FTI_RankAffectedBySoftError()`. This function is non-collective and merely sets a flag in the local address space.

A custom recovery process is coupled to the application and the particular characteristics of the algorithm. The goal is to protect as much application data as possible against soft errors by implementing an ad hoc mechanism that allows the regeneration of the protected variables to correct values when corrupted. Two different scenarios are possible: a) the process affected by the soft error can regenerate the affected data using only local information; b) the process affected by the soft error needs the neighbour processes to be involved in the recovery.

The second scenario requires knowledge of the soft error by those unaffected peers that need to participate in the recovery. The `FTI_Snapshot` function, which is inserted in the main loop of the application, has been modified to check for soft error status at a user defined interval. The default soft error detection mechanism is global, involving all processes running the application. However, the programmer may also implement a custom detection mechanism in which only a subset of the application processes detect the soft error, i.e., scenarios in which the application processes are divided in groups, and only those in the group of the failed peer need to participate in its recovery.

Techniques such as forward recovery or ABFT most commonly focus on regenerating the affected data using local information, therefore in this work we focus on the first scenario, in which data can be recovered locally.

In most situations the regeneration of the protected variables will imply accessing data from a past state of the computation. The extended API function `FTI_RecoverLocalVars()` recovers only a subset of the checkpointed variables for the calling process from its last checkpoint. This is a non-collective function, thus it can be called by only a subset of the processes (for completely local recoveries, only by the processes actually affected by the soft error). Relying on the data from the last checkpoint to implement an ad hoc local recovery is useful for some applications, however, this is not the general case. For instance, local forward recovery usually requires to use information from the current iteration which is not included in the last checkpoint (i.e, checkpoint frequency has a much coarser granularity). Let's consider an iterative application in which a relevant amount of the data used by the algorithm can be protected against soft errors by using the values of some variables at the beginning of the last iteration. Those variables may not be needed to perform a global rollback when a fail-stop failure occurs. Thus, checkpointing them will lead to larger checkpoint files, and therefore, higher overheads. In addition to this, the ad hoc recovery will imply checkpointing at every iteration of the execution, which will be translated in most cases into an inadequate checkpointing frequency for the application. To prevent this situation, a fast memory-saving mechanism is provided. The `FTI_MemSave()` and `FTI_MemLoad()` routines perform a copy in-memory of a given variable and restore its contents, respectively. These routines allow the programmer to save only the subset of variables that are mandatory for the ad hoc recovery procedure, and to do so with the required frequency.

The flexibility of these new extensions to the FTI API enables developers to implement a more efficient recovery strategy exploiting the particular characteristics of the appli-

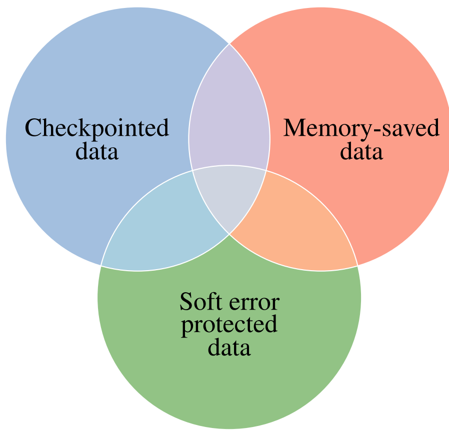

Fig. 2: Memory Protection

cation over a wide range of scenarios. As commented before, some variables can be checkpointed and some variables can be memory-saved using a different frequency in each case. Figure 2 shows a schematic view of the memory used by the application. Note that there is no restriction regarding the checkpointed, memory-saved, and protected data: the intersections between these three sets may or may not be empty. Protected data is defined as data that can be regenerated by using the checkpointed variables, the memory-saved ones, both, or neither of them (e.g., read-only data initialized from files). The fact that a variable is checkpointed and/or memory-saved depends only on the application requirements when checkpointing or when doing an ad hoc recovery.

IV. Implementing Ad Hoc Recovery on HPC Applications

In order to demonstrate the value of the new extensions to the FTI library, this section describes the implementation of an ad hoc local forward recovery on three different HPC benchmarks. All codes are instrumented with FTI to obtain checkpoint/restart fault tolerance support. Then, the new functionalities are used to protect a relevant part of the application data against soft errors.

A. Himeno

The Himeno benchmark [35] solves a 3D Poisson equation in generalized coordinates on a structured curvilinear mesh. A simplified overview of the benchmark code instrumented with FTI is shown in Figure 3a. The call to `FTI_Snapshot` is located in the main loop. The only variables that need to be checkpointed by FTI are the array `p` (pressure) and the loop index `n`. The other variables in Himeno are either initialized at the beginning of the execution and read-only (`a`, `b`, `c`, `wrk1`, `bnd`), initialized in each iteration to constant values, or merely depend on `p(n)`. When a soft error hits, the failed process can re-initialize the read-only variables locally at any point of the execution, as shown in Figure 3b.

For Himeno, data protected against soft errors correspond to the read-only variables, which account for 85% of the memory used by the benchmark, as it will be presented in more detail in Section V. Himeno does not need to memory-save any variable to tolerate soft errors. Also, none of the checkpointed variables (i.e, pressure) can tolerate a soft error through local forward recovery. The data protected against soft errors are read-only variables that can be regenerated without being saved or checkpointed.

B. CoMD

CoMD [36] is a reference implementation of classical molecular dynamics algorithms and workloads as used in Materials Science. It is created and maintained by The Exascale Co-Design Center for Materials in Extreme Environments (ExMatEx). Figure 4a presents the code instrumented with FTI. To tolerate fail-stop failures, FTI checkpoints the variables `nAtoms`, `gid`, `iSpecies`, `r`, `p`, `f`, and `iStep`.

In contrast to the Himeno benchmark, there are no read-only variables within each timestep. However, some variables are read-only within each invocation of the `computeForce` method, which consumes around 93% of the total run time. These read-only variables can be protected against soft errors by memory-saving their contents before the invocation of the `computeForce` method, and replacing the default error handler with a custom one within the scope of this function. Within the error handler (shown in Figure 4b), their value will be set to those valid when the method was invoked. In CoMD, the protected variables that can be regenerated are both memory-saved and checkpointed.

C. TeaLeaf

TeaLeaf [37] is a mini-app from the mantevo project that solves the linear heat conduction equation. Pawelczak et al. have implemented ABTF for TeaLeaf [32] as an alternative method of protecting sparse matrices and dense vectors from data corruptions, which can be combined with this proposal to obtain a fast local forward recovery upon a soft error corrupting the protected variables.

A simplified overview of the benchmark code instrumented with FTI is shown in Figure 5a. A CG (Conjugate Gradient) solver is performed within each step of the main loop. Most of the variables are initialized for each CG solver run. Therefore, locating the checkpoint call in the most external main loop avoids checkpointing all the internal data of the CG solver, reducing the checkpointed data by 87% (and its overhead) while providing an adequate checkpointing frequency. To tolerate fail-stop failures, only the main loop index and the `density` and `energy` arrays need to be checkpointed.

There are read-only variables within a CG solver that are initialized using the local `density` array. These read-only variables can be protected against soft errors: they can be regenerated using the version of the `density` array when the CG solver was invoked. Figure 5b shows the error handler for TeaLeaf to regenerate those read-only variables. The handler can be invoked at any time during the CG solver execution, thus, the `density` array may have been modified. In order to preserve the `density` array, it is copied to a temporary

```
1  [ ... ]
2  FTI_Protect { n, p }
3  signal(SIGDUE, sig_handler_sigdue);
4  for(n=0 ; n<NN ; ++n){ /* Application main loop */
5      int checkpointed = FTI_Snapshot();
6      for i=1:imax-1, j=1:imax-1, k=1:imax-1 {
7          /* Read only data: { a,b,c,p,wrk1,bnd } */
8          /* Write only data: { s0,ss,gosa,wrk2 } */
9      }
10     for i=1:imax-1, j=1:imax-1, k=1:imax-1 {
11         /* Read only data: { wrk2 } */
12         /* Write only data: { p }   */
13     }
14 }
15 [ ... ]
```

(a) Instrumenting for checkpointing and soft error correction.

```
1  void sig_handler_sigdue(int signo){
2
3      FTI_RankAffectedBySoftError();
4
5      /* Algorithm recovery code */
6
7      /* Re-initializate the     */
8      /* read only data:         */
9      /*  { a, b, c, wrk1, bnd } */
10
11 }
```

(b) Handler for forward recovery.

Fig. 3: Himeno simplified pseudocode.

```
1  [ ... ]
2  FTI_Protect { nAtoms,gid,iSpecies,r,p,f,iStep }
3  /* Application main loop */
4  for (iStep=0; iStep<nSteps; iStep++){
5      int checkpointed = FTI_Snapshot();
6      if(iStep%printRate==0)sumAtoms(s);
7      advanceVelocity( ... );
8      advancePosition( ... );
9      redistributeAtoms( ... );
10     signal(SIGDUE, sig_handler_sigdue);
11     FTI_MemSave { gid, iSpecies, r, p }
12     computeForce( ... );
13     signal(SIGDUE, SIG_DFL);
14     advanceVelocity( ... );
15     kineticEnergy(s);
16 }
17 [ ... ]
```

(a) Instrumenting for checkpointing and soft error correction.

```
1  void sig_handler_sigdue(int signo){
2
3      FTI_RankAffectedBySoftError();
4
5      /* Algorithm recovery code */
6      /* Recover values of read  */
7      /* only variables within   */
8      /* computeForce:           */
9
10      FTI_MemLoad { gid, iSpecies, r, p }
11
12 }
```

(b) Handler for forward recovery.

Fig. 4: CoMD simplified pseudocode.

```
1  [ ... ]
2  FTI_Protect { tt, density, energy }
3  signal(SIGDUE, sig_handler_sigdue);
4  /* Application main loop: diffuse method */
5  for(tt = 0; tt < end_step; ++tt){
6      int checkpointed = FTI_Snapshot();
7      /* CG Solver */
8      FTI_MemSave { density }
9      cg_init_driver(chunks, settings, rx, ry, &rro);
10     for(t = 0; t < max_iters; ++t){
11         cg_main_step_driver(chunks, settings, t, &rro, error);
12         halo_update_driver(chunks, settings, 1);
13         if(fabs(*error) < eps) break;
14     }
15     solve_finished_driver(chunks, settings);
16 }
17 [ ... ]
```

(a) Instrumenting for checkpointing and soft error correction.

```
1  void sig_handler_sigdue(int signo){
2      FTI_RankAffectedBySoftError();
3
4      /* Algorithm recovery code */
5      memcpy(aux_density, density,
6            sizeof(double)*sizeD);
7
8      FTI_MemLoad { density }
9      /* Regenerate read-only variables
10      * using density from the previous
11      * ckpt file
12      */
13      [...]
14
15      memcpy(density, aux_density,
16           sizeof(double)*sizeD);
17 }
```

(b) Handler for forward recovery.

Fig. 5: TeaLeaf simplified pseudocode.

variable. Then, the protected variables are generated using the `density` array that was memory-saved on invocation. In contrast to CoMD, in TeaLeaf not all the checkpointed variables are memory-saved, and none of the checkpointed or memory-saved variables can be regenerated upon a soft error.

V. Experimental Evaluation

The experimental evaluation was performed in the CTE-KNL cluster at the Barcelona Supercomputing Center (BSC-CNS), based on Intel Xeon Phi Knights Landing processors, a Linux Operating System and an Intel OPA interconnection.

TABLE I: Weak scaling configurations and baseline run time.

App	NbProcs	Parameters	Run time (s)
Himeno	64	gridsize:513x2049x1025	498.54
	128	gridsize:1025x1025x2049	501.92
	256	gridsize:2049x2049x1025	502.61
CoMD	64	x=128, y=128, z=256, N=100	561.69
	128	x=128, y=256, z=256, N=100	582.10
	256	x=256, y=256, z=256, N=100	591.66
TeaLeaf	64	cells:3000x3000, timesteps=20	256.71
	128	cells:6000x3000, timesteps=20	482.70
	256	cells:6000x6000, timesteps=20	765.25

TABLE II: Memory characterization of the tested benchmarks.

		Used memory (%avail.)	Ckpt'd memory (%used)	Memory-saved (%used)	Protected memory (%used)
Himeno	64p	82.03	7.14	0.00	85.71
	128p	82.03	7.14	0.00	85.71
	256p	82.03	7.14	0.00	85.71
CoMD	64p	10.92	89.24	62.42	62.42
	128p	10.90	89.29	62.46	62.46
	256p	10.92	89.24	62.42	62.42
TeaLeaf	64p	1.05	13.88	6.94	51.37
	128p	1.05	13.88	6.94	51.37
	256p	1.05	13.88	6.94	51.37

Each node of the cluster has one Intel(R) Xeon Phi(TM) CPU 7230 @ 1.30GHz 64-core processor, 94 GB of main memory with 16 GB high bandwidth memory (HBM) in cache mode, and 120 GB SSD as local storage. The technique proposed in this work leverages all four storages levels efficiently: HBM in cache mode is used to store the computation variables, the main memory is used to store the extra data necessary for the local forward recovery, the SSDs are used to store the multilevel checkpoint files and the file system is used to store checkpoints required to comply with batch scheduler limitations (i.e., 24-48 hours jobs).

In order to quantify the results we determined the relative overheads introduced by our modifications with respect to the original code. The measurements are performed by increasing the number of processes while keeping the problem size per process constant (i.e., *weak scaling*). The parameters used for each experiment are given in Table I, together with the original completion run times without any FTI instrumentation. For statistical robustness, we ran the experiments multiple times. Thus, both in the table and in the rest of this section, each reported number corresponds to the mean of ten executions.

A. Memory Characterization

First, we characterize the memory footprint of the benchmarks in Table II. The table reports as Used Memory the percentage of memory used by the application in comparison to the available memory, that is, the total memory available in the running nodes. Additionally, the table presents (i) the percentage of the used memory that needs to be Checkpointed, (ii) the percentage of the used memory that needs to be Memory-saved using the new FTI extensions, and (iii) the percentage of the used memory that is Protected, i.e., that can be recovered from memory corruptions.

As observed, Himeno is the benchmark with the largest amount of used memory (82.03% of the total memory that is available) which was expected as Himeno is a memory bounded benchmark. In contrast to that, CoMD and TeaLeaf only use a small fraction of the available memory. Most molecular dynamic applications such as CoMD have small memory footprint. On the other hand, TeaLeaf is a sparse-matrix computationally-bounded benchmark, which explains its reduced memory consumption. All in all, these three benchmarks give us a wide spectrum with significantly different memory usages to analyze the impact of the proposed resilience schemes in deep-memory hierarchies.

The checkpointed datasets are also heterogeneous across the different testbed benchmarks. Himeno checkpoints 7.14% of the used memory which corresponds to aggregated checkpoint file sizes varying between 5.51GB and 22.03GB when increasing the number of processes. CoMD generates checkpoint file sizes ranging between 9.16GB and 36.65GB as the benchmarks scale out, which correspond to checkpoining around 89.3% of the used memory. Finally, TeaLeaf checkpoints 13.88% of the used memory, and generates the smallest checkpoint file sizes of the testbed benchmarks, which range between 0.14GB and 0.55GB. This shows that not all the data used by the benchmarks needs to be checkpointed to perform a successful global rollback after a failure, and accounts for significantly different checkpointing overheads.

Another relevant difference between the three benchmarks is the memory footprint of the forward recovery strategy (the data that is memory-saved) and the protection coverage that is achieved by doing so. The table reports both quantities (`Memory-saved` and `Protected` data) as the percentage of the used memory. As commented before, a memory corruption on the protected data is tolerated using the local forward recovery strategies described in Section IV, while corruptions on other datasets will lead to a global rollback from the last checkpoint. The potential benefits of this technique will be defined by the amount of protected, memory-saved, and checkpointed data; as well as by the amount of computation that needs to be repeated when doing a global rollback. Scenarios in which a large protection coverage at low cost (i.e. small amounts of memory-saved data) can introduce important performance benefits in the recovery upon a soft error, and, thus, in the total execution run time. This scenario in which a large amount of data can be regenerated from a small portion of memory-saved data is ideal, however it is not always the case. The testbed benchmarks cover very different scenarios regarding protected and memory-saved data. For the Himeno benchmark, we can protect 85.71% of the used data without needing to memory-save any variables, thus, at no cost. In the case of CoMD, we can protect 62.42% of the application data from soft errors by memory-saving the same

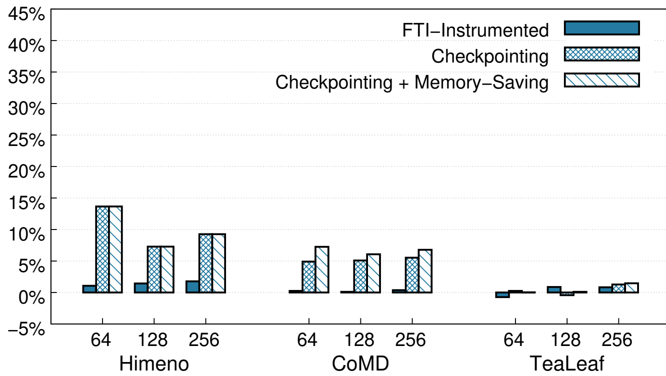

Fig. 6: Relative overheads in a fault-free execution with respect to benchmarks original run times.

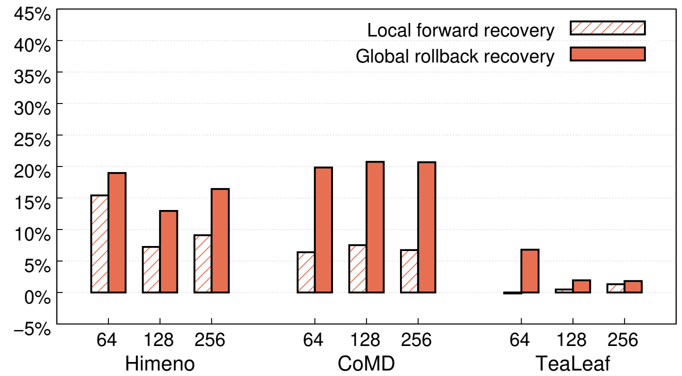

Fig. 7: Relative overheads when introducing a failure with respect to benchmark original run times.

amount of data, which accounts for datasets 30% smaller than the checkpointed data. Finally, for TeaLeaf, 51.37% of the used data is protected by memory-saving only 6.94% of the datasets, which corresponds to half of the checkpointed data.

The next sections evaluate the overheads of the local forward recovery implementation in the fault-free execution and study the benefits obtained when recovering from soft errors.

B. Overhead in the Absence of Failures

This section studies the overheads introduced in the fault-free execution. Figure 6 shows the relative overheads with respect to the original execution run times (reported in Table I) for three different experiments. Firstly, `FTI-instrumented` experiments measure the performance penalty that is introduced by the calls to the FTI library added to the application code, but neither checkpointing nor memory-saving any dataset. The relative instrumentation overhead is low, on average, 0.66% and always below 1.77%.

Secondly, we study the overhead introduced when checkpointing for a global rollback. The `Checkpointing` experiments generate checkpoint files at the optimal frequency, calculated as defined by Young [38] and Daly [39]. These experiments show the overhead that is introduced when checkpointing to tolerate fail-stop failures, enabling a global rollback recovery after a failure. The relative checkpointing overhead is, on average, 5.92% and never exceeds 14.88%. This overhead is tied to the checkpoint file size (i.e., the time writing the data) and the synchronization cost that is introduced by the coordinated checkpointing provided by FTI. For TeaLeaf, the overheads are sometimes negative, as the instrumentation modifies the application code, the compiler's optimizations (and their benefit) may differ.

Lastly, `Checkpointing`+`Memory-saving` experiments study the overhead introduced when not only checkpoints are taken but also the necessary datasets are memory-saved. Thus, allowing the recovery using a global rollback upon a fail-stop failure and enabling the usage of the local forward recovery when a soft error hits any of the protected variables. In these experiments, checkpointing is performed at the optimal checkpointing frequency, while memory-saving data is performed at every iteration of the main loop of the application. The overhead introduced by the memory-saving operations is negligible, and the cost of the in-memory copy accounts for an increase over the checkpointing overhead of, on average, 0.57%, and always below 2.36%. As we see in Figure 6, Himeno does not incur any extra overhead because no data is memory-saved. On the other hand, for CoMD and TeaLeaf the local forward recovery strategies described in Section IV require a memory copy of a subset of the application data in every iteration for the regeneration of the protected variables.

C. Overhead in the Presence of Failures

This section compares the performance upon errors of the global rollback recovery and the local forward recovery. In these experiments, a soft error is introduced when approximately 75% of the execution has been completed by signalling one of the processes running the benchmark. In the local forward recovery, the affected process handles the signal and recovers from the soft error to continue the execution, as depicted in Section IV. On the other hand, in the global rollback the soft error triggers a fail-stop failure, the signal terminates the execution and a global restart takes place, recovering all the processes running the benchmark from the most recent available checkpoint file.

Figure 7 presents the overheads introduced when handling the failure with each strategy. The overheads are calculated as the difference between the original execution run times (reported in Table I) and the run times when introducing a soft error. Thus, the overheads correspond to the extra time consumed by each proposal in order to tolerate the error. Additionally, Table III details the reduction in the failure overhead that the local forward recovery achieves over the global rollback, both in absolute and relative terms.

On average, the reduction corresponds to 7.38% of the execution time, with a maximum of 14.01% for CoMD. Note that this reduction is determined by two factors: the size of the checkpoint files and the amount of re-computations that has to be performed. For Himeno and CoMD, the reduction in the overhead increases as more processes run the benchmarks.

TABLE III: Reduction in the overhead when using the local forward recovery instead of the global rollback upon a soft error: absolute value (in seconds) and percentage value (normalized with repect to the original execution run time).

NProcs	Reduction in the overhead: seconds [%]					
	Himeno		CoMD		TeaLeaf	
64	17.93	[3.60%]	75.75	[13.49%]	17.65	[6.88%]
128	29.06	[5.79%]	77.01	[13.23%]	7.07	[1.46%]
256	37.51	[7.46%]	82.91	[14.01%]	3.93	[0.51%]

On the other hand, for TeaLeaf, the reduction decreases when scaling out, because when using the optimal checkpointing frequency, the amount of computation to be repeated in the global rollback decreases, e.g. in the 256 processes experiment a checkpoint is taken in every iteration and barely no computation is repeated when rolling back. The local forward recovery avoids that all processes read the checkpoint files at the same instant, as it occurs in a global rollback recovery. More importantly, it avoids rolling back the processes running the application, avoiding the repetition of computation already done, and, therefore, reducing the failure overhead.

VI. Discussion

The new functionalities introduced on FTI enable the implementation of ad hoc recovery techniques to improve the performance when tolerating soft errors in MPI applications. The usage of these new extensions does not disturb the operation of traditional checkpoint/restart to handle any other type of failures by means of a global rollback to the last valid set of checkpoint files. The evaluation of these new functionalities was done implementing forward local recovery strategies on three different benchmarks. In all of them, read-only data is protected against soft errors in the scope of functions that consume most part of the execution run time, and the protected data account for an important part of the memory used by the application, thus, providing a good coverage ratio. In order to provide this protection, one benchmark does not need to memory-save any data, while the other two do so: in one, the amount of memory-saved and protected data are the same, while in the other one the protected data is 7.4 times larger than the memory-saved data. In all cases, any process can recover locally from the soft error corrupting the protected data by re-initialize/regenerate it. The protection of read-only data has been proved to reduce the failure overhead. In addition, using algorithm specific knowledge, the protection coverage can be extended to other datasets, not necessarily read-only, which will introduce further reduction in the recovery overhead. Table IV summarizes the programming effort of this protection coverage introduces in the testbed benchmarks. The table reports the number of lines of code that are added to each benchmark to obtain global rollback support with FTI, and to extend the fault tolerance support with an ad hoc recovery to manage soft errors.

Although this approach requires that the user identifies the read-only variables in the computationally most expensive parts of the applications, the benefits in the recovery of soft errors affecting the protected variables can be important, and these type of variables can represent an important fraction of the data used by HPC applications. In addition, as this is a static analysis of the application code, it could be automatically done by a compiler.

TABLE IV: Original number of lines of code (LOCs) and extra lines of instrumentation code to obtain fault tolerance support for global rollback and for global rollback combined with ad hoc recovery for soft errors.

	LOCs original code	Extra LOCs global rollback	Extra LOCs global rollback + ad hoc soft error recovery
Himeno	284	15	57
CoMD	5638	30	82
TeaLeaf	4415	15	99

In order to quantify the performance benefits that this technique can offer in production environments, we have modelled (i) the overhead introduced when all failures are managed by global rollback, and (ii) the overhead when those failure originated from soft errors that can not be corrected by hardware mechanisms, i.e. uncorrectable errors (UEs), are managed by ad hoc recoveries. The overall relative overheads of the global rollback and the ad hoc recovery are estimated as follows (notations detailed in Table V):

$$O_{Global} = Checkpointing + Restart(F_t) + Recompute(F_t) \quad (1)$$

$$O_{AdHoc} = Checkpointing + Restart(F_{nDUE}) + Recompute(F_{nDUE}) + ExtraAH + Recovery_{AH}(F_{DUE}) \quad (2)$$

The different terms in the equations (1) and (2) are defined as follows:

$$Checkpointing = 100\frac{C_{cost}(Wr_{bw}) \cdot N_{ckpts}}{R_{time}}$$

$$Restart(F) = F \cdot 100\frac{C_{cost}(Rd_{bw})}{R_{time}}$$

$$Recompute(F) = F \cdot 100\frac{C_{int}/2}{R_{time}}$$

$$Recovery_{AH}(F) = F \cdot \frac{Restart(1) + Recompute(1)}{Reduction_{factor}}$$

The checkpoint cost, interval, number of checkpoints, and number of failures and number of failures are denoted as:

$$C_{cost}(bw) = \frac{C_{size}}{bw}, \quad C_{int} = \sqrt{2 \cdot MTBF \cdot C_{cost}(Wr_{bw})}$$

$$N_{ckpts} = \frac{R_{time}}{C_{int}}, \quad F_t = \frac{R_{time}}{MTBF}, \quad F_{DUE} = \frac{R_{time}}{MTBF} \cdot \frac{\%_{DUEs}}{100}$$

$$F_{nDUE} = F_t - F_{DUE}$$

Finally, the simplified equations used to calculate the relative overhead are to the following ones:

$$O_{Global} = \frac{100 \cdot C_{size}}{Wr_{bw} \cdot C_{int}} + \frac{100 \cdot C_{size}}{Rd_{bw} \cdot MTBF} + \frac{100 \cdot C_{int}}{2 \cdot MTBF}$$

$$O_{AdHoc} = \frac{100 \cdot C_{size}}{Wr_{bw} \cdot C_{int}} + \frac{(\%_{nDUE}) \cdot C_{size}}{Rd_{bw} \cdot MTBF} + \frac{(\%_{nDUE}) \cdot C_{int}}{2 \cdot MTBF} + ExtraAH + \frac{\%DUE}{MTBF} \cdot \frac{2 \cdot C_{size} + Rd_{bw} \cdot C_{int}}{2 \cdot Rd_{bw} \cdot Reduction_{factor}}$$

TABLE V: Summary of key model notations.

NOTATION	DESCRIPTION
O_{Global}	Overall overhead using a global rollback.
O_{AdHoc}	Overall overhead using ad hoc recovery for soft errors.
$Checkpointing$	Checkpointing overhead.
$Restart(F)$	Restart overhead of F failures.
$Recompute(F)$	Recomputation overhead of F failures.
$ExtraAH$	Extra overhead introduced by the ad hoc recovery operations to obtain FT support. On average: 0.57%.
$Recovery_{AH}(F)$	Overhead of tolerating F failures with the ad hoc recovery.
R_{time}	Application run time.
$Reduction_{factor}$	Average factor by which the global failure overhead is reduced when using the ad hoc recovery. Average: 2.1.
F_t, F_{nDUE}, F_{DUE}	Number of failures: total, non DUEs, DUEs.
$\%_{nDUE}, \%_{DUE}$	Percentage of failures originated by: non DUEs, DUEs.
$C_{cost}, Rd_{bw}, Wr_{bw}$	Checkpoint cost; reading and writing bandwidth local SSD.
$C_{int}, C_{size}, N_{ckpts}$	Checkpoint interval, checkpoint size, number of checkpoints.
$MTBF$	Mean Time Between Failures.

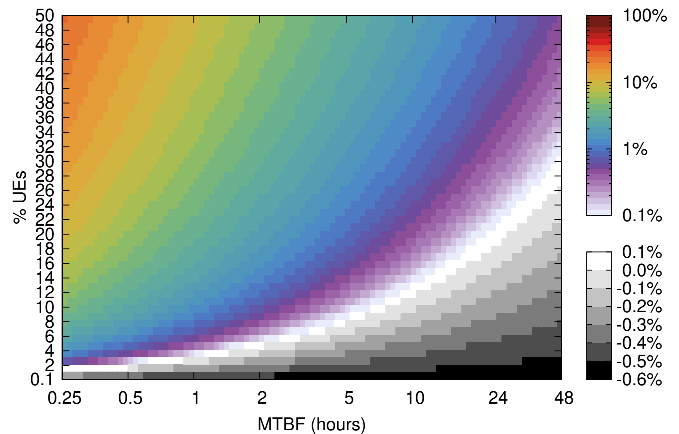

Fig. 8: Reduction in the relative overhead varying MTBF and percentage of DUEs. Parameters: reading bandwidth per node 0.5GB/s, writing bandwidth per node 0.4GB/s, checkpoint file size per node of 200GB, and 128 nodes.

The benefit of the proposal is estimated as the reduction in the relative overhead when incorporating the ad hoc recovery ($O_{Global} - O_{AdHoc}$) to tolerate DUEs (while every other failure is tolerated using a global rollback). Figure 8 shows this gain for different MTBF and different percentage of DUEs over the total number of failures. The simulation fixes the checkpoint file size (200GB per node and using 128 nodes), and the I/O bandwidth (0.5GB/s for reading, and 80% of that performance for writing using local SSD). The positive values of the reduction in the relative overhead are represented in log scale. Even though a performance penalty is introduced by the ad hoc recovery in systems with large MTBFs and a low percentages of DUEs, the extra overhead introduced is very low (less than 0.6%). On the other hand, the performance benefit is notable for other parameters configurations, specially for low MTBF, which are expected in the exascale era. Although memory protection mechanisms (e.g., ECC Chipkill) contribute to keep the rate of UEs low, new hardware devices such as GPUs and FPGAs are more prone to suffer this type of errors as they still do not implement all those technique in their products.

All in all, these extensions to the FTI library do not impose a particular strategy for the management of soft errors. Their flexibility enables the programmer to adapt the recovery strategy to the algorithm and to the mathematical and/or domain-specific properties of the underlying problem that the code is modelling. These strategies would require a more extensive knowledge of the application and its domain, and can introduce some programming overhead if the user is not familiar with the application code. Alternative recovery techniques may include the regeneration of the corrupted datasets by means of re-computation, for instance, to obtain a good enough approximation of a partial result by using other non-corrupted and/or memory-saved datasets. Although the most efficient recovery techniques most probably would correspond with those in which the processes affected by a soft can recover locally, this proposal is not restricted to this layout. In other scenarios, processes may need to use information from remote peers, requesting the participation of other neighbour processes in the recovery from a soft error. The goal of these FTI extensions is to facilitate the catching and handling of error signals, to free the user of the duties of saving and restoring data in the applications at different frequencies and to allow the easy extraction of partial information from the checkpoint files in order to leverage from local forward recovery strategies.

VII. CONCLUDING REMARKS

This work presents the extension of the FTI checkpointing library to facilitate the implementation of ad hoc recovery strategies for HPC applications, to provide protection against those soft errors that cannot be corrected by hardware mechanisms. The new functionalities provide programmers different mechanisms to save and access data from a past state of the computation, without requiring it to be checkpointed nor imposing restrictions on the saving frequency. The flexibility of the extensions enables the implementation of more efficient recoveries by exploiting the particular characteristics of the application over a wide range of scenarios, while still being compatible with multilevel checkpoint/restart. These extensions have been evaluated on three different HPC benchmarks to implement a local forward recovery. In those benchmarks, a relevant part of the application data is protected against soft errors by taking into account the particularities of each algorithm. The experimental evaluation demonstrates the low overhead introduced by the proposal, while it provides important performance benefits when recovering from soft errors.

The usage of these new features in combination with a more sophisticated knowledge of the algorithm particular characteristics will allow the implementation of recovery protocols with further performance benefits.

Future work includes the application of these new extensions to Monte Carlo applications and Genetic Algorithms (among others), in which memory corruptions over random data can be regenerated on the fly.

ACKNOWLEDGEMENTS

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie grant agreement No 708566 (DURO). This research is also supported by the Ministry of Economy and Competitiveness of Spain and FEDER funds of the EU (Projects TIN2016-75845-P and the predoctoral grant of Nuria Losada ref. BES-2014-068066), and by the Galician Government (Xunta de Galicia) under the Consolidation Program of Competitive Research (ref. ED431C 2017/04).

REFERENCES

[1] J. Dongarra, T. Herault, and Y. Robert, "Fault tolerance techniques for high-performance computing," in *Fault-Tolerance Techniques for High-Performance Computing*, 2015, pp. 3–85.

[2] B. Schroeder and G. Gibson, "A large-scale study of failures in high-performance computing systems," *Transactions on Dependable and Secure Computing*, vol. 7, no. 4, pp. 337–350, 2010.

[3] C. Di Martino, Z. Kalbarczyk, and R. Iyer, "Measuring the Resiliency of Extreme-Scale Computing Environments," in *Principles of Performance and Reliability Modeling and Evaluation*, 2016, pp. 609–655.

[4] A. Avižienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.

[5] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson *et al.*, "Addressing failures in exascale computing," *The International Journal of High Performance Computing Applications*, vol. 28, no. 2, pp. 129–173, 2014.

[6] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir, "Toward exascale resilience: 2014 update," *Supercomputing frontiers and innovations*, vol. 1, no. 1, pp. 5–28, 2014.

[7] L. Bautista-Gomez, F. Zyulkyarov, O. Unsal, and S. McIntosh-Smith, "Unprotected computing: a large-scale study of DRAM raw error rate on a supercomputer," in *High Performance Computing, Networking, Storage and Analysis (SC)*, 2016, pp. 645–655.

[8] B. Schroeder, E. Pinheiro, and W.-D. Weber, "DRAM errors in the wild: a large-scale field study," in *SIGMETRICS Performance Evaluation Review*, vol. 37, 2009, pp. 193–204.

[9] R. Baumann, "Soft errors in advanced computer systems," *IEEE Design & Test of Computers*, vol. 22, no. 3, pp. 258–266, 2005.

[10] T. J. Dell, "A white paper on the benefits of chipkill-correct ECC for PC server main memory," *IBM Microelectronics Division*, vol. 11, 1997.

[11] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka, "FTI: high performance fault tolerance interface for hybrid systems," in *High Performance Computing, Networking, Storage and Analysis (SC)*, 2011, pp. 1–12.

[12] P. H. Hargrove and J. C. Duell, "Berkeley lab checkpoint/restart (BLCR) for linux clusters," in *Journal of Physics: Conference Series*, vol. 46, 2006, pp. 494–499.

[13] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill, "C 3: A system for automating application-level checkpointing of MPI programs," in *International Workshop on Languages and Compilers for Parallel Computing*, 2003, pp. 357–373.

[14] G. Rodríguez, M. J. Martín, P. González, J. Touriño, and R. Doallo, "CPPC: a compiler-assisted tool for portable checkpointing of message-passing applications," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 749–766, 2010.

[15] 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.

[16] S. Pauli, M. Kohler, and P. Arbenz, "A fault tolerant implementation of multi-level Monte Carlo methods," *Parallel Computing: Accelerating Computational Science and Engineering*, vol. 25, p. 471, 2014.

[17] W. Bland, K. Raffenetti, and P. Balaji, "Simplifying the recovery model of user-level failure mitigation," in *Workshop on Exascale MPI at High Performance Computing, Networking, Storage and Analysis (SC)*, 2014, pp. 20–25.

[18] I. Laguna, D. F. Richards, T. Gamblin, M. Schulz, and B. R. de Supinski, "Evaluating user-level fault tolerance for MPI applications," in *European MPI Users' Group Meeting*, 2014, p. 57.

[19] 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.

[20] F. Rizzi, K. Morris, K. Sargsyan, P. Mycek, C. Safta, B. Debusschere, O. LeMaitre, and O. Knio, "ULFM-MPI implementation of a resilient task-based partial differential equations preconditioner," in *Workshop on Fault-Tolerance for HPC at Extreme Scale*, 2016, pp. 19–26.

[21] 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, pp. 100–113, 2017.

[22] 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," *CoRR*, vol. abs/1708.02030, 2017. [Online]. Available: http://arxiv.org/abs/1708.02030

[23] J. Elliott, K. Kharbas, D. Fiala, F. Mueller, K. Ferreira, and C. Engelmann, "Combining Partial Redundancy and Checkpointing for HPC," in *I. C. on Distributed Computing Systems*, 2012, pp. 615–626.

[24] C. Engelmann, H. Ong, and S. L. Scott, "The case for modular redundancy in large-scale high performance computing systems," in *IASTED I. C. on Parallel and Distributed Computing and Networks*, vol. 641, 2009, p. 046.

[25] K. Ferreira, J. Stearley, J. H. Laros, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold, "Evaluating the viability of process replication reliability for exascale systems," in *High Performance Computing, Networking, Storage and Analysis (SC)*, 2011, pp. 1–12.

[26] J. Sloan, R. Kumar, and G. Bronevetsky, "An algorithmic approach to error localization and partial recomputation for low-overhead fault tolerance," in *Dependable Systems and Networks*, 2013, pp. 1–12.

[27] K.-H. Huang and J. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations," *Transactions on Computers*, vol. 33, no. 6, pp. 518–528, 1984.

[28] G. Bosilca, R. Delmas, J. Dongarra, and J. Langou, "Algorithm-based fault tolerance applied to high performance computing," *Journal of Parallel and Distributed Computing*, vol. 69, no. 4, pp. 410–416, 2009.

[29] Z. Chen and J. Dongarra, "Algorithm-based fault tolerance for fail-stop failures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 12, pp. 1628–1641, 2008.

[30] T. Davies, C. Karlsson, H. Liu, C. Ding, and Z. Chen, "High performance linpack benchmark: a fault tolerant implementation without checkpointing," in *I. C. on Supercomputing (ICS)*, 2011, pp. 162–171.

[31] D. Hakkarinen and Z. Chen, "Algorithmic Cholesky factorization fault recovery," in *International Symposium on Parallel & Distributed Processing*, 2010, pp. 1–10.

[32] G. Pawelczak, S. McIntosh-Smith, J. Price, and M. Martineau, "Application-Based Fault Tolerance Techniques for Fully Protecting Sparse Matrix Solvers," in *Cluster Computing*, 2017, pp. 733–740.

[33] J. Yeh, G. Pawelczak, J. Sewart, J. Price, A. A. Ibarra, S. McIntosh-Smith, F. Zyulkyarov, L. Bautista-Gomez, and O. Unsal, "Software-level Fault Tolerant Framework for Task-based Applications," in *Poster session at High Performance Computing, Networking, Storage and Analysis (SC)*, 2016.

[34] L. Bautista-Gomez and F. Cappello, "Detecting and correcting data corruption in stencil applications through multivariate interpolation," in *Cluster Computing*, 2015, pp. 595–602.

[35] Himeno Benchmark, http://accc.riken.jp/en/supercom/himenobmt/.

[36] CoMD website, http://proxyapps.exascaleproject.org/apps/comd/.

[37] TeaLeaf website, https://github.com/UoB-HPC/TeaLeaf.

[38] J. W. Young, "A first order approximation to the optimum checkpoint interval," *Comm. of the ACM*, vol. 17, no. 9, pp. 530–531, 1974.

[39] J. T. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *Future Generation Computer Systems*, vol. 22, no. 3, pp. 303–312, 2006.