Design and Study of Elastic Recovery in HPC Applications

Kai Keller
Barcelona Supercomputing Center
(BSC-CNS)
Barcelona, Spain
kai.keller@bsc.es

Konstantinos Parasyris
Lawrence Livermore National Laboratory
(LLNL)
Livermore, CA, US
parasyris1@llnl.gov

Leonardo Bautista-Gomez
Barcelona Supercomputing Center
(BSC-CNS)
Barcelona, Spain
leonardo.bautista@bsc.es

***Abstract*—The efficient utilization of current supercomputing systems with deep storage hierarchies demands scientific applications that are capable of leveraging such heterogeneous hardware. Fault tolerance, and checkpointing in particular, is one of the most time-consuming aspects if not handled correctly. High checkpoint performance can be achieved using optimized multilevel checkpoint and restart libraries. Unfortunately, those libraries do not allow for restarts with a modified number of processes or scientific post-processing of the checkpointed data. This is because they typically use an N-N checkpointing scheme and opaque file-formats. In this article, we present a novel mechanism to asynchronously store checkpoints into a self-descriptive file format and load the data upon recovery with a different number of processes. We provide an API that defines the process-local data as part of a globally shared dataset. Our measurements demonstrate a low overhead between 0.6% and 2.5% for a 2.25 TB checkpoint with 6K processes.**

I. Introduction

During the last decades, supercomputers have increased in size and complexity, and exascale computing is on the way. However, several challenges come with this quest, and one of the most important is fault tolerance. As the number of components in *high-performance computing* (HPC) systems increase, the systems become more error-prone. It is expected that the next generation of HPC systems experiences failures every few hours [10], [28]. Consequently, most long-running HPC applications will experience multiple failures during the execution due to the reduced *mean time between failures* (MTBF) [18], [24]. Usually, HPC applications employ *checkpoint and restart* (CR) to recover from failures [11]. In CR, the application state is periodically stored and upon a failure recovered from the last checkpoint. CR has been highly optimized by multilevel checkpoint libraries to take advantage of multiple storage levels and minimize the IO bottleneck.

Despite their high performance, multilevel CR libraries still suffer from some shortcomings. They often use an opaque file format, and the data layout in the checkpoint files is agnostic to the developer. Thus, although HPC applications may potentially run with a variable number of processes, CR libraries do not support recovery from a checkpoint with an adjustable number of processes (a.k.a. elastic restart). Consequently, the recovery from a failure needs to take place with the exact number of processes as the application was formerly executed with. On the other hand, the developer cannot extract the data from the checkpoint files manually and restart on a changed decomposition of the application without knowledge of the underlying file format. The same applies for utilizing the checkpointed data for visualization or data analyses, as the developer cannot extract the data from the checkpoint files.

In this article, we propose a novel multilevel checkpoint API and runtime that empowers developers to expose the topological information of the decomposition, and to control the data layout in the checkpoint files. As the checkpointed data is often also the object of the analysis, we make two friends with one gift. The methods described in this article will be applicable in numerous categories, i.a., cosmological applications, climate simulations, and reanalyses. For instance, GISELA5D, a plasma simulation for fusion reactions, already uses the HDF5 format to combine resiliency and analysis [5].

The contributions of this work are summarized as follows:

- We propose API extensions able to achieve both objectives (i.e., resiliency and IO). Our API simplifies the code, increases productivity and improves readability.
- We implement an asynchronous strategy that leverages multiple storage levels for fast checkpointing. We demonstrate a 5x speedup on comparison with ADIOS Sustainable Staging Transport (SST).
- Our results show that our approach has negligible checkpoint and restart overheads and high scalability.
- We showcase online recovery using a resilient MPI implementation and demonstrate automatic elastic restart.
- Our study highlights some of the challenges for irregular applications. We analyze the trade-off between fast checkpoint vs fast restart and propose several solutions.

The remainder of the paper is structured as follows: Section II explains the foundation of this paper. Section III focuses on the implementation of our extensions. Section IV describes the methodology for our detailed analysis. Section V presents and discusses the results of our experiments. Section VII explores related work, and Section VIII concludes this article.

II. Background

In this section, we introduce the main libraries and concepts necessary to understand the contributions of this work.

© 2020 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. DOI 10.1109/HiPC50609.2020.00040

A. Fault Tolerance

1) Multilevel CR libraries: Rollback-recovery is one of the most used resilience technique for HPC applications and several libraries focus on this technique. In this work, we are particularly interested in multilevel CR libraries because of their speed and their efficient use of deep storage hierarchies. Among others, common libraries from this class are SCR [29], FTI [4] and VeloC [7]. Any of these libraries can be adapted to support elastic restart as proposed in this paper.

Multilevel checkpointing is characterized by providing a variety of checkpointing methods, called *levels*. The levels differ in performance and reliability. Common levels are: a) Node level, b) Partner, c) Ecnoded, d) Global checkpoint. Additionally, some libraries offer the possibility to perform the checkpoints in two stages. The first stage consists of writing the checkpoint data locally to node storage. Afterward, the checkpoint is processed further, depending on the level of reliability. For instance, flushing the files to the *parallel file system* (PFS), or sending a copy of the checkpoint files to another node. Those post-processing stages can then be performed by dedicated processes asynchronously.

2) MPI Layer Fault Tolerance: Fault-tolerant MPI mechanisms have been an object of investigation for many years now [15], [19], [34]. Some popular mechanisms for fault tolerance in MPI are ULFM [2], FT-MPI [34] and MPI_Reinit [8], [22]. The common goal of these frameworks is to provide a mechanism for the developers to cope with process failures, allowing them to continue the execution without the need to launch a new MPI job.

B. General Purpose IO

1) HDF5: HDF5 (Hierarchical Data Format) [35] is a file format that allows the storage of complex datasets embedded inside a hierarchical folder-like structure [13]. Inter alia, HDF5 allows the creation of named groups (similar to folders in file systems) and named datasets (continuing the analogy, the files in a file system). The HDF5 file format represents the standard format for scientific datasets. The API is very complete and actively maintained. Furthermore, there exist bindings to a great variety of programming languages and applications such as C/C++, Fortran, Python, MatLab and R, and it has been optimized for multiple file systems [17], [25].

2) ADIOS: ADIOS [23] is a state-of-the-art IO library for HPC applications. ADIOS can be operated with several output engines. The more interesting for this article is *sustainable staging transport* (SST) [21] in combination with HDF5. The latter can be used to stage global checkpoint data locally using memory buffers and *remote direct memory access* (RDMA). Afterwards, the data is combined to a single HDF5 file on the PFS asynchronically to the application.

III. Implementation

In this section we outline the design objectives of our extensions and present the basic concepts and mechanisms. The proposed interface gives developers a tool at hand that solves the disagreement mentioned in section I. In short, it consolidates the IO work for resiliency with the IO work for data processing and it also allows for elastic restart from the checkpoint files using a variable number of processes. Deploying our extensions does not restrict the other features of the CR library in any way. It remains possible to perform checkpoints in all other reliability levels. We implemented our extensions on top of FTI (without loss of generality).

A. Design Objectives

The design objectives for our API extensions are:

Complex Data Representation: The core of the problems that HPC applications try to solve is formulated using complex data structures. These structures are usually organized either as structure of arrays or as an array of structures. Our interface should provide an intuitive mechanism to define both.

Accessibility to the Data: Checkpoints in HPC CR libraries do not provide access to the data from outside of internal library calls. Our objective is to remove this restriction and to give the developer the capacity of customizing the structure of the checkpoint data to access the data processing decoupled from the checkpoint library.

Intuitive API: Developers prefer using libraries and APIs which they already know before investing their time to learn a new specification. Consequently, we align our extensions to the structure and functions of the HDF5 API, which is the file format that we use to achieve the accessibility of the data.

B. API Specification

In this section we showcase the proposed API extensions. We use a simple example to demonstrate how the extensions can be used to structure the data inside the checkpoint file. This will clarify i) how to organize the data for scientific post-processing and visualization and ii) how to prepare the application buffers to allow the elastic recovery.

1) Complex Data Representation: In our example, we simulate the movement of particles, in the 3-dimensional space, that are exposed to a force. Each particle state is represented by its position and velocity. The structure that defines a particle is presented in Listing 1.

```
1 typedef struct coord_t {
2     double x,y,z;
3 } coord_t;
4
5 typedef struct particle_t{
6     coord_t position;
7     coord_t velocity;
8 } particle_t;
```

Listing 1: The particles data-type is represented by a structure with two members, which are represented by the respective data-structures. This is an example for a nested composite data-type. We explain how to expose this type with our API in Listing 2.

Our interface provides mechanisms to describe complex datasets of the application and store those to the checkpoint file with the corresponding information about shape, type and relationship. The first step is to expose the data-types of the datasets to the checkpoint library. Besides the standard types

(integer, floating-point, char, etc.), which are predefined, the user can define derived data-types that correspond to structures or classes. Essentially, derived types are defined by calling `FTI_InitCompositeType` to initialize a complex data-type and by adding further information (members, names, etc.) with calls to `FTI_AddScalarField` for scalar members and `FTI_AddVectorField` for array members. Listing 2 shows this in detail for the particle type from Listing 1. Since the members of this type are also of a derived data-type, we need to expose this type first.

```
1 fti_id_t FTI_COORD = FTI_InitCompositeType("COORD",
      sizeof(coord_t), NULL);
2 FTI_AddScalarField(FTI_COORD, "X", FTI_DBLE,
      offsetof(coord_t, x));
3 FTI_AddScalarField(FTI_COORD, "Y", FTI_DBLE,
      offsetof(coord_t, y));
4 FTI_AddScalarField(FTI_COORD, "Z", FTI_DBLE,
      offsetof(coord_t, z));
5
6 fti_id_t FTI_PARTICLE = FTI_InitCompositeType("PARTICLE",
      sizeof(particle_t), NULL);
7 FTI_AddScalarField(FTI_PARTICLE, "POSITION", FTI_COORD,
      offsetof(particle_t, position));
8 FTI_AddScalarField(FTI_PARTICLE, "VELOCITY", FTI_COORD,
      offsetof(particle_t, velocity));
```

Listing 2: To expose the nested particle data-type from Listing 1, we need to expose the coordinate type first. Composite data-types are defined with `FTI_InitCompositeType`. Their members are added with `FTI_AddScalarField` (or `FTI_AddVectorField` for array members).

2) Accessibility of the Data: Now that the data-types are exposed, the next step is to define the global dataset representation. Listing 3 shows all the necessary steps to take. The properties of a global dataset are the name, the location in the file (parent group) and the dimensions. Those properties can be defined using the function `FTI_DefineGlobalDataset()`. The global dataset in our example is the superset of all the particles before the domain decomposition. Once the global dataset is defined, every process needs to define its proportion (i.e. subset) of the dataset. A subset is sufficiently described by its coordinates relative to the global dataset and the number of elements in each coordinate direction. Variables such as the iteration counter or communicator size, which take the same value on all ranks can be defined in the same way. In that case, the values for count and offset need to be identical on all MPI ranks. The dimensions of the global dataset in our example are represented by the 3 variables `gX`, `gY` and `gZ`, whereas the dimension of the subsets for each process are represented by the variables `lX`, `lY` and `lZ`.

```
1 lX = gX;
2 lY = gY;
3 lZ = gZ/nbProcs;
4
5 size_t MEMSIZE = lx*ly*lz*sizeof(particle_t);
6 particle_t *particles = (particle_t*)malloc( MEMSIZE );
7
8 int globalDim = {gZ, gY, gX};
9 size_t offset[3] = {0, 0, rank*lZ};
10 size_t count[3] = {lZ, lY, lX};
11
12 FTIT_H5Group GRID;
13 FTI_InitGroup(&GRID, "GRID", NULL);
14
15 int VAR_ID_PARTICLES = 0;
16 int DATASET_ID_PARTICLES = 0;
17
18 FTI_DefineGlobalDataset(DATASET_ID_PARTICLES, 3,
      globalDim, "PARTICLES", &GRID, FTI_PARTICLE);
19 FTI_Protect(VAR_ID_PARTICLES, particles, lX*lY*lZ,
      FTI_PARTICLE);
20 FTI_AddSubset(VAR_ID_PARTICLES, 3, offset, count,
      DATASET_ID_PARTICLES);
```

Listing 3: In this example, we create an HDF5 group `GRID` to show how to create a hierarchy in the checkpoint file. The particle dataset is then added to this group. Further, we define the global dimensions of the dataset (line 15) and specify the process local region inside the dataset (line 17) according to the domain decomposition (here we merely partition along the z-axis.)

C. Scientific Processing of the Checkpoint Files

The underlying file format of our implementation is HDF5, however, the proposed mechanism works as well with other formats such as parallel netCDF [20] or ADIOS. We implemented functions to enable HDF5 features such as named groups and named datasets allowing to structure the data inside the checkpoint file. Reading the data from the checkpoint file is straightforward, as the developer has defined the structure of the HDF5 checkpoint file by himself. Consequently, the data can be extracted and further processed using third-party scripts/tools that provide an interface to HDF5.

D. Elastic restart

After defining the global view of the problem, all necessary information is available for the checkpoint library to perform an elastic restart. Upon recovery the CR library automatically determines, which data share needs to be assigned to which rank in order to perform the elastic recovery and continue the execution on a different number of processes. What remains, is to indicate a global checkpoint by passing the level flag `FTI_L4_H5_SINGLE` to `FTI_Checkpoint()`.

E. Checkpoint Strategies

```
1 int iter;
2 for (iter=1; iter<=MAX_ITER; iter++){
3   if( FTI_Status() != 0 ) FTI_Recover();
4   else if(iter%8000==0) FTI_Checkpoint(iter,
      FTI_L4_H5_SINGLE);
5   else if(iter%4000==0) FTI_Checkpoint(iter, FTI_L3);
6   else if(iter%2000==0) FTI_Checkpoint(iter, FTI_L2);
7   else if(iter%1000==0) FTI_Checkpoint(iter, FTI_L1);
8   simulateSystem(grid);
9 }
```

Listing 4: Example of a simple checkpoint strategy that combines checkpointing into a shared HDF5 file with other levels of reliability. Higher levels are preferred. The recovery is performed automatically upon restart.

It is important to emphasize, that our extensions are meant to be used along the native features of the library. The resiliency strategy should comprise a combination of all reliability levels available. Most of the CR libraries provide very fast and highly scalable checkpoint levels (e.g. in memory checkpointing, leveraging local NVMe's, etc.) which are suitable to perform checkpoints with high frequency. At the same time, the applications may not need to write data for analyses that often. Typically, the best strategy deploys a mixture of all levels. Listing 4 shows an example of a simple checkpoint strategy that uses 4 different levels of reliability.

F. Asynchronous Checkpoint

The global HDF5 checkpoint creation can be divided into two stages. Initially, the application processes store the data

to fast local storage on the nodes (stage 1). Afterwards, the local files are merged to a shared file on the PFS in the background (stage 2). This method is called *asynchronous checkpointing*, because the second stage is performed asynchronously to the application by one dedicated MPI process per node. The structure of the shared file will be the same as in the synchronous mode as defined by the user with our API extensions. We implemented this functionality in alignment to the pre-existing FTI head feature. When operating in this mode, the job allocation has to be extended by 1 process per node, since the feature does not use spawned but integrated MPI processes.

IV. Methodology

A. Analytical Modeling and Metrics

To evaluate the techniques proposed in this paper we use an analytical model that provides a clear definition of the checkpoint and recovery overheads. To measure and estimate the overheads of our implementation we performed executions within the following scenarios:

S0: The application is executed without any fault tolerance support, consequently this is the execution time of the application alone.

S1: The application is protected and performs N_{ckpt} checkpoints during execution.

S2: This scenario is the same as S1, except that the execution is interrupted by a failure and the recovery.

By subtracting the execution time of S0, $T(\text{S0})$, from the execution time of S1, $T(\text{S1})$, we can determine the total checkpoint overhead as:

$$\Delta T_{ckpt} = T(\text{S1}) - T(\text{S0}) \quad (1)$$

Equivalently, we get the total recovery overhead by subtracting S1 from S2:

$$\Delta T_{reco} = T(\text{S2}) - T(\text{S1}) \quad (2)$$

The average overhead per checkpoint, becomes:

$$t_{\text{ckpt}} = \Delta T_{\text{ckpt}} / N_{ckpt}. \quad (3)$$

and the time for the interrupt and recovery becomes:

$$t_{reco} = \Delta T_{reco} - T_{recompute} \quad (4)$$

where we have subtracted the time spent in recomputations from the total recovery overhead.

To provide a normalized quantity to compare between measurements for different scales and applications, we use the relative checkpointing overhead, and the relative recovery overhead:

$$\delta_{\text{ckpt}} = \frac{t_{\text{ckpt}}}{t_{\text{opt}}}, \quad \textit{relative checkpointing overhead} \quad (5)$$

$$\delta_{\text{reco}} = \frac{t_{\text{reco}}}{t_{\text{mtbf}}}, \quad \textit{relative recovery overhead} \quad (6)$$

where t_{mtbf} corresponds to the MTBF and $t_{\text{opt}} = \sqrt{2 t_{\text{ckpt}} t_{\text{mtbf}}}$ to the optimal checkpointing interval as suggested in [37].

The relative overheads defined as in equations 5 and 6, are parametrized by the three quantities t_{mtbf}, t_{ckpt}, and t_{reco}. Let T_{exp} be the expected time to completion of the application without protection and failures (for instance, $T_{\text{exp}} = T(S0)$ in our experiments). Once t_{mtbf}, t_{ckpt}, and t_{reco} are determined, one can predict the absolute checkpoint overhead of an experiment on any cluster using:

$$\Delta T_{\text{tot}} = (\delta_{\text{ckpt}} + \delta_{\text{reco}}) T_{\text{exp}} + \Delta T_{\text{corr}} \quad (7)$$

where:

$$\delta_{\text{ckpt}} T_{\text{exp}} = \frac{T_{\text{exp}}}{t_{\text{opt}}} t_{\text{ckpt}} = N_{\text{ckpt}} t_{\text{ckpt}} = \Delta T_{\text{ckpt}}$$

and:

$$\delta_{\text{reco}} T_{\text{exp}} = \frac{T_{\text{exp}}}{t_{\text{mtbf}}} t_{\text{reco}} = N_{\text{fail}} t_{\text{reco}} = \Delta T_{\text{reco}}$$

With this, equation 7 gets simply:

$$\Delta T_{\text{tot}} = \Delta T_{\text{ckpt}} + \Delta T_{\text{reco}} + \Delta T_{\text{corr}}.$$

ΔT_{corr} is a correction accounting for the re-executions when rolling back to a checkpoint after a failure and the additional overhead, if the difference between the total execution time and the expected time-to-completion exceeds the MTBF.

B. Checkpoint and Restart Strategies

In addition to clear analytical metrics, it is important to evaluate all the relevant configurations, both for checkpointing and restart. To evaluate the performance of our new checkpoint implementation, we have tested it against the traditional checkpointing mechanism that generates one file per process. Experiments that use the traditional interface are labeled with **Trad** and experiments leveraging the new implementation are labeled with **Novel**. Each mode, in turn, was measured with two different configurations for checkpoint and recovery. The checkpoint configurations are labeled **Sync** and **Async** and denote respectively synchronous and asynchronous checkpointing. The recovery configurations are labeled **Off** and **On** and denote respectively offline and online recovery.

For asynchronous checkpointing the CR library leverages dedicated checkpoint processes. In that case, the checkpoint is separated into a pre- and post-processing stage. During the pre-processing stage, the checkpoint data is stored to local storage such as DRAM, *non-volatile memory* (NVMe) or *solid-state drives* (SSD) on the compute-nodes. During the post-processing, the local files are converted into levels of higher reliability, leveraging the dedicated checkpoint processes. The second stage happens in parallel to the application, i.e., asynchronously. In contrast, for synchronous checkpointing the application processes perform both stages sequentially.

Online recovery refers to the recovery without termination of the application upon failure. This becomes possible using ULFM, Re-Init or another flavor of resilient MPI (See Section II-A2). The recovery takes place on the remaining processes during runtime. In offline recovery on the contrary, the application is terminated upon failure and restarted afterward to perform the recovery of the execution. We have listed the various configurations and their labels in Table I.

File Format		Checkpoint Methodology		Recovery methodology	
Trad	Traditional interface, binary file format (N-N)	**Sync**	Checkpoint post-processing by the application processes	**Off**	Offline recovery; application terminates upon failure.
Novel	Checkpointing into shared HDF5 file (N-1) for independent data post-processing and elastic restarts	**Async**	Checkpoint post-processing by the dedicated processes	**On**	Online recovery; application stays alive upon failure, reinitialization and restart with remaining processes.

TABLE I: Different C/R scenarios tested in the evaluation section with respect to the file format, the checkpoint method and the recovery method.

C. Applications

We want to study how the techniques proposed in this paper can be applied to different types of applications. The HPC ecosystem has a wide spectrum of scientific codes, some of them are based on regular static grids that are easy to work with, while others are more irregular and significantly harder to work with. This complexity has an impact on both the checkpoint and the restart. Therefore, we must analyze both sides of the spectrum to have a complete picture.

1) Heat2D: Heat2D is a 2-dimensional stencil C++ application simulating a heat distribution process in a chamber. It operates on a 1-dimensional domain decomposition. The body of the application is arranged in three parts 1) initialization 2) mainloop 3) finalization. This structure is very common for HPC applications. In spite of its simplicity, the application is a good representative of many HPC codes (i.e., stencils) and a good test case for an elastic restart with fewer processes on an online recovery scenario (Novel/On).

2) xPic: xPic is a *particle in cell (PIC)* application for space plasma simulations. It is derived from the iPIC3D code [31]. It studies charged particle streams propagating inside the magnetic field of the earth (i.e., space weather) [1]. The simulation space is decomposed into a 3-dimensional grid of cells and each cell is initialized with a certain amount of particles at the beginning. The number of cells for each direction is customizable as is the number of particles per cell. The electric and magnetic fields are discretized and defined at fixed grid points (nodes). On the contrary, the particle positions are continuous and the particles can move between the cells of the grid. As a consequence, the number of elements per cell is constant for the field arrays, but not for the particle arrays. This poses a challenge for the elastic restart.

The grid is partitioned into cells as its smallest unit. The field values on the nodes form a 3-dimensional array. Each rank operates on a sub-volume of the grid, dictated by the MPI decomposition. The global field data in the checkpoint file is organized in row-major data-alignment (3D to 1D mapping). Writing the fields is straight-forward and every rank can determine the offset and count of the data in the file independently. The same applies to the recovery. Every rank can determine the offset and count corresponding to the current decomposition due to the fixed number of nodes. This does not apply to the particles. Given that particles move around the grid during the execution, the number of particles per cell varies, thus, at checkpoint time the ranks cannot know the offset in the shared file without communication among the other ranks. Even more challenging is the recovery in that case. Similar challenges are faced by general IO libraries, as the user has to hand-code some application-specific sections that are hard to handle automatically (e.g., particle redistribution upon recovery). Thus, xPic is a good example of an irregular application to evaluate the generality of the techniques proposed in this paper. We will come back to this in section V-G, where we discuss the elastic restart for irregular applications.

V. Evaluation

In this section, we evaluate the performance of our implementation V-B, compare to the state-of-the-art IO library (ADIOS) V-C, and demonstrate its scalability V-D. We investigate the benefits of online recovery V-E, and evaluate the overhead that is imposed due to the increased execution time when restarting with fewer processes V-F. Finally, we analyse some of the challenges of elastic restart on irregular applications and we demonstrate some solutions to achieve good performance for both checkpoint and restart V-G.

A. HPC Environment

All experiments are performed on MareNostrum IV, the supercomputer at the Barcelona Supercomputing Center (BSC). Each compute node is equipped with 2 Intel Xeon Platinum CPUs (24 cores each), 12x8 GB DDR4 main memory, a 100Gbit network and 10Gbit ethernet to the PFS [30]. All local checkpoints are performed in-memory, using the node-local RAM Drive (`/dev/shm`).

B. Performance Measurements

Recent studies show that modern HPC systems have several failures per day [16], [32], consequently, we set MTBF in equations 5 and 6 equal to 6 hours. This choice corresponds to the estimated MTBF when the application uses 15K cores [9]. Our evaluation uses the methodology presented in Section IV and the evaluation parameters are presented in Table II.

Evaluation Parameters	xPic	Heat2D
Nodes	16	16
MPI-processes per Node	12	47
Threads Per Rank	3	1
CP size per Node (GB)	11	18
Num of MPI-ranks	192	752
Total CP size (GB)	176	288

TABLE II: Configuration and scale for the benchmark experiments.

Table III lists the outcome of our measurements for both applications and Figure 1c shows the results in greater detail. As we can see, most of the overhead during the recovery for xPic comes from the re-distribution of the particles (we

will analyse this in Section V-G). However, in both cases, the overhead for recovery and checkpointing is very low. The total relative overhead (i.e., checkpoint and recovery) exposed by our implementation (Novel) varies from 0.61% (Heat2D/Async) to 2.99% (xPic/Sync). Compared to the traditional interface (Trad), we achieve slightly higher values in the synchronous case and slightly lower values in the asynchronous case[1]. With the help of equation 7 we can now estimate the maximal and minimal overhead for a 12h-execution (i.e., $T_{exp} = 12$ h):

$$\Delta T_{tot} \approx \begin{cases} 22 \text{ minutes} & \textit{(Synchronous)} \\ 4 \text{ minutes} & \textit{(Asynchronous)} \end{cases}$$

In Section V-G we will see, that we can reduce the recovery overhead for xPic even further by avoiding the re-distribution of the particles upon the restart.

		δ_{ckpt} [%]		δ_{reco} [%]	
		xPic	**Heat2D**	**xPic**	**Heat2D**
Trad	Sync	1.56	1.81	0.09	0.15
	Async	0.95	1.03	0.11	0.09
Novel	Sync	2.40	2.12	0.59	0.05
	Async	0.80	0.57	0.61	0.04

TABLE III: Results for the relative CR overheads.

Overall, these results demonstrate that the new technique proposed in this article performs comparably well to the state-of-the-art multilevel checkpoint techniques that leverage the local storage of current deep-memory architectures. The overhead imposed on both applications is extremely low ($< 1.5\%$) when applying asynchronous post-processing.

C. Comparison to ADIOS

After comparing the performance of the novel interface to the traditional interface within the CR library, we compared it to the asynchronous staging feature (SST) of ADIOS. The feature leverages fast memory buffers and RDMA to stage the files though the network. The shared HDF5 file is created in the background by several dedicated processes, after receiving the checkpoint data from the application processes. This feature is similar to the asynchronous case that we have explained before (See Section II-A1). ADIOS can also stage the files through node memory, however, this doesn't match the functionality of our interface, since the consolidation of the node-local data needs additional steps to be taken care of manually. The consolidation of the data in our implementation, however, is part of the runtime and is transparent for the developer. Thus, the fair comparison is to the ADIOS solution that stages and consolidate the files transparently, which is the solution that stages over the network using RDMA. ADIOS allows to allocate an arbitrary amount of dedicated processes, which can as well be located on a separate set of nodes. Due to hard-coded limitations in the buffer size per dedicated process, we needed to allocate as many staging processes as application processes. This is a disadvantage if the total number of processes in the allocation is limited. In our implementation, the number of dedicated staging processes is fixed, however, occupies only 1 process per node.

		$\mathbf{t_{ckpt}}$ **(sec)**	$\mathbf{t_{reco}}$ **(sec)**	**Bandwidth (GB/s)**
Novel	Sync	19.7	22.6	14.6
	Async	0.92	6.9	157.9
ADIOS	Sync	18.9	21.4	15.2
	Async	5.14	10.4	28.2

TABLE IV: The resulting values for the relative CR overheads for N-1 (shared HDF5 file on PFS). Comparison between our proposal and ADIOS.

Overall, the flexibility of ADIOS' staging feature did not match the performance of our asynchronous checkpoint implementation. Table IV summarizes the results of our measurements. Writing directly to the PFS (Sync) does not show any significant difference in performance between the two libraries. When staging the data though (Async), we achieve a significant performance advantage. Our implementation is over five times faster, writing the checkpoint files asynchronously, which could be considered as a 5x increase in terms of bandwidth. This difference is mostly because ADIOS stages the data over the network, hence, with more effort on developers side, as mentioned earlier, this gap could be closed. An important takeaway from this experiment is that with our interface, we achieve very high performance with little effort for the developer, compared to other IO libraries.

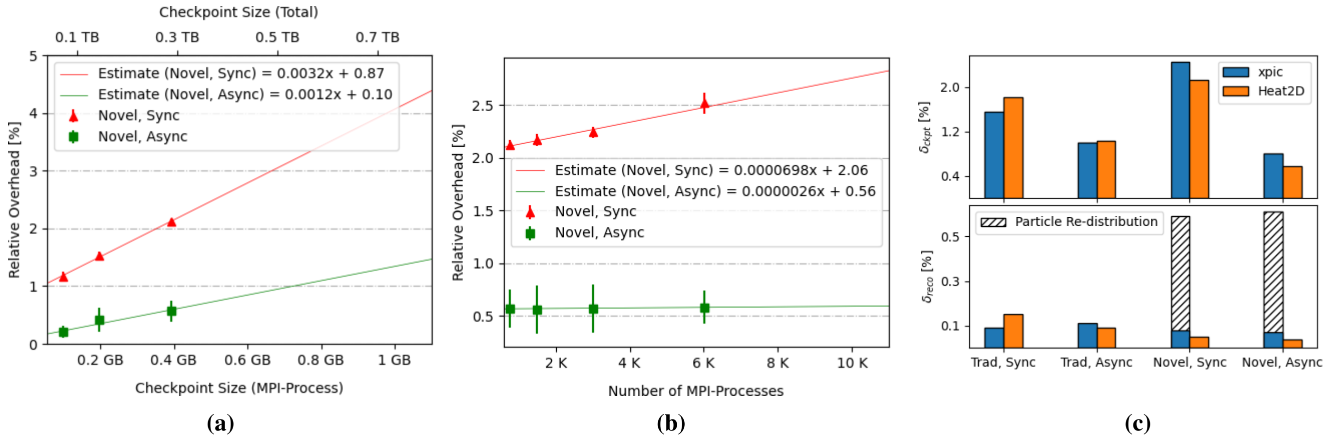
D. Scaling

We demonstrated the overall low overhead of our implementation and its high performance in comparison to a state-of-the-art IO library. Now we want to study its scalability. We performed experiments that vary the checkpoint load per process (similar to strong scaling) and experiments that vary the total checkpoint size, but keep the size per process constant (weak scaling).

1) Strong Scaling: Figure 1a shows the outcome of the strong scaling experiments. A weighted linear regression model on the data allows an estimation for higher loads. If we consider, for instance, 1GB per process, we obtain relative checkpointing overheads for Novel/Sync of about 4% and Novel/Async of about 1.3%. Both cases show that the overhead remains low even while increasing the amount of data per process.

2) Weak Scaling: Figure 1b shows the results of the weak scaling experiments. We performed experiments on 16, 32, 64 and 128 nodes and 47 processes on each node, while keeping the checkpoint data per process constant. We can see that the overhead for the synchronous checkpointing (Sync) increases linearly with the number of processes. The overhead for the asynchronous checkpointing (Async) on the other hand, remains practically constant. Using again a weighted linear regression model, we can estimate the overheads for higher scaled runs. The overheads for runs with 15K processes

[1]The latter one results from the fact that the new technique does not create metadata files, or checksums for integrity checks.



(a) **(b)** **(c)**

Fig. 1: **(a)** Relative checkpoint overhead for our new extensions for varying checkpoint data per process (strong scaling) and **(b)** for varying number of nodes and constant checkpoint data per process (Weak scaling). **(c)** Top: Relative checkpoint overhead. Bottom: Relative recovery overhead. For xPic we separated the recovery overhead into read and re-distribution of the particles.The difference in latency between online and offline recovery depending on the number of nodes.

estimate to 3.1% and 0.6% respectively. The constant behavior of the asynchronous checkpointing is expected since the checkpoints are performed on each node independently. Thus, as the number of nodes increases the bandwidth increases together with the global amount of data.

E. Offline vs Online Elastic Recovery

The checkpoint technique introduced in this paper opens new opportunities, such as automatic elastic online recovery on a reduced number of processes. This can be achieved using our interface in combination with any fault tolerant MPI implementation. For the experiments of this section, we implemented elastic online recovery in Heat2D using ULFM.

To get notified upon a process failure, we set the standard MPI error handler to `MPI_ERRORS_THROW`. In addition to that, we wrapped the computation block (99.99% of the execution time) of the main loop inside a try-catch statement. Controlled failure injection takes place within the computation block. After the failure injection, the affected processes terminate execution and the surviving processes throw an exception and call a customized handler. Inside the handler, the processes invoke the ULFM function `MPIX_Comm_shrink` which excludes the failed processes from the MPI communicator. The remaining processes are then rolled back to the last checkpoint and continue execution.

The experiments use the same scale as the benchmarking experiments from section V-B (Table II). We simulated the loss of 1 node and then restarted on the remaining 15 nodes using either online recovery (On) or offline recovery (Off). Figure 2a shows the time difference between the both methods (x-axis uses a logarithmic scale!). As expected, online recovery is faster than offline recovery (and this is assuming an immediate restart of the job). We can see that the effect is noticeable already at a small scale. Towards large executions, the effect becomes ever more important. Applying a linear model, we can see that the effect doubles when we double the number of nodes. The effect is likely to depend on the application complexity and the initial memory allocation. That is to say, the effect could be even more drastic for applications with complex initialization processes. For the simple case of Heat2d, we can estimate a difference of about 10 minutes at a scale of 10K nodes ($\sim$ 500K processes). If we consider numerous failures and recoveries, which is expected to happen at such scale, this is a significant overhead which can be avoided applying online recovery.

F. Restart on a Reduced Number of Processes

An important effect to consider when restarting on a reduced number of processes, is the increased iteration time due to the higher workload per process. In this section, we analyse the additional overhead resulting from such a situation. We performed experiments on $k = n - i$ nodes, where $n = 50$ and $i = 0, \ldots, 10$. The baseline is T_0, the iteration time at $k = 50$. The additional relative overhead due to the reduced number of nodes is defined as:

$$\delta_{iter,i} = \frac{T_i - T_0}{T_0}. \tag{8}$$

With this, the additional overhead, $\Delta T_{iter,k}$, after the k-th recovery and the total additional overhead, $\Delta T_{iter,tot}$, become:

$$\Delta T_{iter,k} = \delta_{iter,k} \cdot T_0 \tag{9}$$

$$\Delta T_{iter,tot} = \sum_{k=0}^{F} \delta_{iter,k} \cdot T_0 \tag{10}$$

where F corresponds to the total number of failures during the execution.

Figure 2b shows the result of these measurements. The y-axis corresponds to $\delta_{iter,i}$ and the x-axis to the percentage of failed nodes w.r.t. the initial number of nodes. Please note that a loss of 20% of the nodes is very unrealistic. In practice, we will be confronted with failures of less than 5%. A linear regression up to 5% shows an almost direct proportional slope, i.e. 1% loss corresponds to 1% overhead. Hence, assuming a failure with maximal 5% of the nodes, We can express

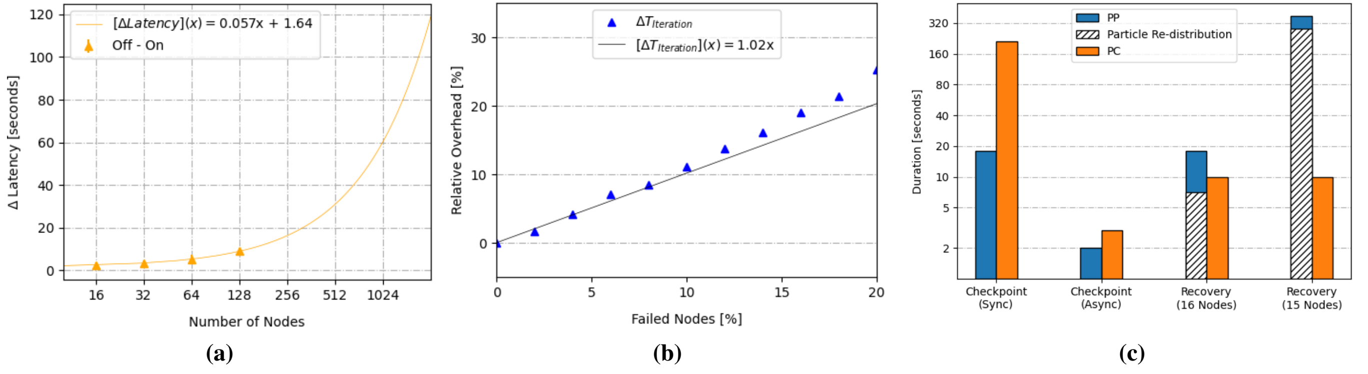


Fig. 2: (a) The difference in latency between online and offline recovery depending on the number of nodes. **(b)** Relative additional overhead for executions on fewer nodes with constant total load. The x-axis shows the percentage of nodes lost upon failure. **(c)** Comparison of the 2 techniques (PP and PC) to organize the particle data in the checkpoint file. Recovery on 16 nodes means recovery on the same number of nodes, whereas recovery on 15 nodes means elastic recovery on a reduced number of nodes (i.e., simulates the loss of 1 node upon failure).

equation 10, the total additional overhead due to the reduced number of nodes by:

$$\delta_{iter,k} = 0.01 \cdot n_k, \quad n_k < 5\% \tag{11}$$

$$\Delta T_{iter,tot} = \sum_{k=0}^{F} 0.01 \cdot n_k \cdot T_0 \tag{12}$$

Where n_k is the percentage of nodes lost during the k-th failure w.r.t. the initial number of nodes. Considering an execution time of 15h and a MTBF of 6h, we can compute an additional overhead of 7.2 minutes when losing 1% of the nodes each failure ($n_1 = 1\%, n_2 = 2\%$). Assuming a loss off 5% of the nodes each time ($n_1 = 5\%, n_2 = 10\%$), the overhead reaches 36 minutes. Clearly the model we presented is rather simple, and load balance issues could increase the additional overhead. Nevertheless, a few extra minutes of overhead is short in comparison to the time spent in a job scheduler queue waiting for a new allocation.

G. Data Distribution on Irregular Applications

In section IV-C2, we mentioned the issue about the irregular number of particles per cell in xPic. Particles in xPic move within the cells and MPI ranks process a continuously changing number of particles. This poses a challenge for elastic restart because it is not trivial to know where in the file the particles of a process start and where they end with respect to the other processes. Here we propose and analyse the generality and performance benefits of two different data layouts, i) per process (*PP*) and ii) per cell (*PC*).

The first layout (PP) is based on the number of particles per process. This approach is oriented towards the best performance while checkpointing. Before writing the particles, the ranks communicate the number of particles they will write. With this information, each rank computes its offset within the global particle dataset and writes the particles accordingly. No additional information is kept. Thus, upon recovery with a different number of ranks, we have not enough information to read the correct particles, that belong to the ranks, from the file directly. Instead, upon restart, we divide the total number of particles in the dataset by the number of ranks and read from the dataset in equal parts. Afterward, we re-distribute the particles among the ranks until all particles arrive at the correct destination.

The second layout (PC) is based on the number of particles per cell. This approach is oriented towards the best performance upon recovery by eliminating the necessity of particle re-distributions after the restart. Each rank writes the particles according to a pre-defined decomposition to the global dataset. The decomposition is aligned to the smallest grid units, the cells. Additionally, we store the number of particles in each cell at the time of the checkpoint. Upon restart, the ranks read first the dataset that contains the particle number per cell and with those, compute the file offset to directly read the correct particles from the file.

We compare both methods in this section. To have a certain grade of complexity, we performed the experiments on a cubic grid (i.e., the number of cells is equal in each coordinate direction). Table V summarizes the most important parameters. Figure 2c shows a comparison between the two techniques.

Evaluation Parameters	xPic
Nodes	16
MPI-processes per Node	32
Grid Dimensions	120x120x120
Number of Cells	$2{\cdot}10^6$
Number of Particles	$3{\cdot}10^9$
Accumulated CP size	280GB

TABLE V: Configuration for the measurements of the 2 different data organization pattern in the checkpoint file for xPic.

As we can see, writing the data directly to the PFS (Sync) imposes significantly more overhead for PC (210 seconds) than for PP (18 seconds). This is because PC requires to write the buffers occasionally discontinuous, which is not optimal for the PFS. However, if we apply the asynchronous checkpoint mechanism, the overhead of both techniques becomes almost identical (2 and 3 seconds for PP and PC respectively). The recovery, on the other hand, is much faster with PC, given that the time spent in particle redistribution is completely

removed. Therefore, PP is faster at checkpointing because it is somehow simplistic but that cost is paid at the restart. PC is faster at restart because a significant effort is done during checkpointing. This more cost, however, is hidden when using asynchronous checkpointing. Hence, with our implementation for the asynchronous write of the shared file, we can efficiently perform elastic restarts in particle-in-cell applications as an example for an irregular application.

VI. Discussion

In this work we proposed API extensions and a runtime to allow multilevel CR libraries to achieve both resilience and general IO through a simple interface while guaranteeing high performance. Nonetheless, it would be naive to think that the proposed API could cover all the corner cases that the entire HDF5 interface allow. For instance, we did not implement HDF5 attributes and the API currently does not offer the possibility to create strided datasets. We expect that the limitations of the proposed API will become apparent with time and practical application. However, the runtime that we developed to consolidate the contributions of the ranks to the shared file in the background, could be integrated into HDF5 directly. It could represent an alternative to the HDF5 subfiling mechanism [6].

The asynchronous checkpoint tested in this work can be applied to any multilevel CR and IO library. We implemented the mechanism using dedicated MPI processes, yet it could be adapted to threads or spawned processes. While IO libraries have generally focused more on staging through IO nodes [3], [21], [33], recent deep storage hierarchies (i.e., NVME, 3DDRAM) are pushing toward solutions that leverage the local storage, such we have presented in this work.

We demonstrated that elastic restart is viable at large scale, and that we can reduce the overhead further using online recovery. We evaluated the overhead when executing on a reduced number of nodes and arrived to an acceptable value compared to the alternative, which is, spending time in the scheduler queue. It is important to highlight that elastic restart is useful for other purposes different to resilience (e.g., malleability of ensemble runs), which increases the scope of our proposal.

Finally, we analyzed the generality of our approach by testing with a highly irregular application. This raised several challenges and highlighted the trade-off between checkpoint performance and restart performance. The first approach (PP) was far more general and did not involve any application-specific measure other than particle redistribution upon restart. The second one (PC) is more application-specific but can leverage the advantages proposed in this work to achieve both fast checkpoint and fast recovery.

VII. Related Work

The work in this article focusses on two aspects. The first addresses the consolidation of multi-level checkpointing and scientific IO, and the second addresses the elastic restart from a different number of processes. Regarding the first aspect, besides application-specific solutions, there are several IO-libraries that can be used for resiliency as well as for scientific IO. However, as we emphasized, there is no library combining the virtues of both families optimized for both purposes.

Among the most important IO libraries are ADIOS, SIONlib [14], netCDF and HDF5. From the libraries above, ADIOS is the closer work to this paper. It allows asynchronous staging to the PFS and node-local writes and provides with those, valuable features that may be used to implement resiliency in HPC applications. However, basic multilevel techniques as for instance, partner checkpointing or encoded checkpoint files are not available. The most important multilevel libraries such as FTI, SCR and VeloC do not always provide access to their checkpoint files, as they sometimes store the data into binary files and do not offer interfaces to extract the data contained inside those files.

As for the other aspect of our proposal, the elastic restart, several ways to continue the execution without the missing processes have been proposed. Redundancy schemes [12], context migration from checkpointed migratable objects [26] or process migration using failure prediction [36] are some of the examples. However, these techniques usually prescribe certain conditions as, for instance, the allocation of shadow nodes for proactive process migration and redundancy, or performance loss when using over-subscription. Migratable objects, such as used in Charm++ [27], need to be considered at program creation as it requires the application to use a specific programming language and code layout. Certainly, there are other ways to allow for an elastic restart and resilient scientific IO inside an application using an application-specific solution. However, the method presented in this work can be applied to a wide range of applications, with limited effort from the developer.

VIII. Conclusion

We presented a novel checkpointing mechanism, that allows an elastic restart to an arbitrary number of processes. We extended a multilevel CR library with functions that allow creating self-descriptive checkpoint files that can be used for scientific IO and resilience. We demonstrated that the checkpoint overhead of our method is comparable to multilevel checkpoint techniques. We have shown that we can reduce the overhead further with asynchronous checkpointing, showing a negligible overhead of about 0.6% (at scale). We have compared our technique with the state-of-the-art IO library ADIOS, and demonstrated that our method is five times faster when leveraging staging methods. We accomplished online recovery using ULFM together with an elastic restart and showed that it reduces significantly the latency at restart compared to traditional offline recovery. Our analysis on the iteration times at varying loads show that for the loss of 1% of the nodes, we can expect an additional overhead of about 1%. We also analyzed the challenges raised by irregular applications and their trade-offs regarding the performance at checkpointing and recovery, and we have proposed methods to solve those challenges.

IX. Acknowledgements

Part of the research presented here has received funding from the Horizon 2020 (H2020) funding framework under grant/award number: 824158; Energy oriented Centre of Excellence II (EoCoE-II). The present publication reflects only the authors' views. The European Commission is not liable for any use that might be made of the information contained therein. This work was partially performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DEAC52-07NA2734 (LLNL-CONF-811845).

References

[1] ipic3d: implicit particle-in-cell code for space weather applications (kth).

[2] Ulfm 2.0, fault tolerance research hub, 2019.

[3] Hasan Abbasi, Matthew Wolf, Greg Eisenhauer, Scott Klasky, Karsten Schwan, and Fang Zheng. Datastager: scalable data staging services for petascale applications. *Cluster Computing*, 13(3):277–290, 2010.

[4] 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*, pages 1–12, Nov 2011.

[5] Bigot, Julien, Latu, Guillaume, Cartier-Michaud, Thomas, Grandgirard, Virginie, Passeron, Chantal, and Rozar, Fabien. An approach to increase reliability of hpc simulation, application to the gysela5d code***. *ESAIM: Proc.*, 53:248–270, 2016.

[6] Suren Byna, Mohamad Chaarawi, Quincey Koziol, John Mainzer, and Frank Willmore. Tuning hdf5 subfiling performance on parallel file systems. 5 2017.

[7] Franck Cappello, Kathryn Mohror, and Bogdan Nicolae. Overview veloc documentation, 2019. Available at https://veloc.readthedocs.io/en/latest/.

[8] Sourav Chakraborty, Ignacio Laguna, Murali Emani, Kathryn Mohror, Dhabaleswar K. Panda, Martin Schulz, and Hari Subramoni. Ereinit: Scalable and efficient fault-tolerance for bulk-synchronous mpi applications. *Concurrency and Computation: Practice and Experience*, 32(3):e4863, 2020. e4863 cpe.4863.

[9] Camille Coti. chapter Fault Tolerance Techniques for Distributed, Parallel Applications, pages 221–252. IGI Global, Hershey, PA, USA, 2016.

[10] D. Dauwe, S. Pasricha, A. A. Maciejewski, and H. J. Siegel. Resilience-aware resource management for exascale computing systems. *IEEE Transactions on Sustainable Computing*, 3(4):332–345, 2018.

[11] Ifeanyi P Egwutuoha, David Levy, Bran Selic, and Shiping Chen. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing*, 65(3):1302–1326, 2013.

[12] James Elliott, Kishor Kharbas, David Fiala, Frank Mueller, Kurt Ferreira, and Christian Engelmann. Combining partial redundancy and checkpointing for hpc. In *2012 IEEE 32nd International Conference on Distributed Computing Systems*, pages 615–626. IEEE, 2012.

[13] Christoph Ertl, Jérôme Frisch, and Ralf-Peter Mundani. Design and optimisation of an efficient hdf5 i/o kernel for massive parallel fluid flow simulations. *Concurrency and Computation: Practice and Experience*, 29(24):e4165, 2017.

[14] Wolfgang Frings, Felix Wolf, and Ventsislav Petkov. Scalable massively parallel i/o to task-local files. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 17:1–17:11, New York, NY, USA, 2009. ACM.

[15] William Gropp and Ewing Lusk. Fault tolerance in message passing interface programs. *The International Journal of High Performance Computing Applications*, 18(3):363–372, 2004.

[16] Saurabh Gupta, Tirthak Patel, Christian Engelmann, and Devesh Tiwari. Failures in large scale systems: long-term measurement, analysis, and implications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2017.

[17] Mark Howison. Tuning hdf5 for lustre file systems. 2010.

[18] Saurabh Hukerikar and Christian Engelmann. Resilience design patterns: A structured approach to resilience at extreme scale. *arXiv preprint arXiv:1708.07422*, 2017.

[19] Joshua Hursey, Richard L. Graham, Greg Bronevetsky, Darius Buntinas, Howard Pritchard, and David G. Solt. Run-through stabilization: An mpi proposal for process fault tolerance. In Yiannis Cotronis, Anthony Danalis, Dimitrios S. Nikolopoulos, and Jack Dongarra, editors, *Recent Advances in the Message Passing Interface*, pages 329–332, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[20] Jianwei Li, Wei-keng Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. Parallel netcdf: A high-performance scientific i/o interface. In *SC '03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, pages 39–39, 2003.

[21] Oak Ridge National Laboratory. Adios 2: The adaptable input/output system version 2 adios2 2.5.0 documentation, 2018.

[22] Ignacio Laguna, David F. Richards, Todd Gamblin, Martin Schulz, Bronis R. de Supinski, Kathryn Mohror, and Howard Pritchard. Evaluating and extending user-level fault tolerance in mpi applications. *International Journal of High Performance Computing Applications*, 30(3), 1 2016.

[23] Qing Liu, Jeremy Logan, Yuan Tian, Hasan Abbasi, Norbert Podhorszki, Jong Youl Choi, Scott Klasky, Roselyne Tchoua, Jay Lofstead, Ron Oldfield, et al. Hello adios: the challenges and lessons of developing leadership class i/o frameworks. *Concurrency and Computation: Practice and Experience*, 26(7):1453–1473, 2014.

[24] C. McNairy. Exascale fault tolerance challenge and approaches. In *2018 IEEE International Reliability Physics Symposium (IRPS)*, pages 3C.4–1–3C.4–10, 2018.

[25] Kshitij Mehta, John Bent, Aaron Torres, Gary Grider, and Edgar Gabriel. A plugin for hdf5 using plfs for improved i/o performance and semantic analysis. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 746–752. IEEE, 2012.

[26] E. Meneses, X. Ni, G. Zheng, C. L. Mendes, and L. V. Kal. Using migratable objects to enhance fault tolerance schemes in supercomputers. *IEEE Transactions on Parallel and Distributed Systems*, 26(7):2061–2074, 2015.

[27] E. Meneses, X. Ni, G. Zheng, C. L. Mendes, and L. V. Kal. Using migratable objects to enhance fault tolerance schemes in supercomputers. *IEEE Transactions on Parallel and Distributed Systems*, 26(7):2061–2074, July 2015.

[28] Z. Miao, J. Calhoun, and R. Ge. Energy analysis and optimization for resilient scalable linear systems. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 24–34, 2018.

[29] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, Nov 2010.

[30] BSC Operations. Technical information marenostrum 4, 2019. Available at https://www.bsc.es/marenostrum/marenostrum/technical-information.

[31] Stefano Markidis; Giovanni Lapenta; Rizwan-uddin. Multi-scale simulations of plasma with ipic3d. *Mathematics and Computers in Simulation*, 80, 2010.

[32] Elvis Rojas, Esteban Meneses, Terry Jones, and Don Maxwell. Analyzing a five-year failure record of a leadership-class supercomputer. In *2019 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 196–203. IEEE, 2019.

[33] Wolfram Schenck, Salem El Sayed, Maciej Foszczynski, Wilhelm Homberg, and Dirk Pleiter. Evaluation and performance modeling of a burst buffer solution. *SIGOPS Oper. Syst. Rev.*, 50(2):1226, January 2017.

[34] A. D. Selvakumar, P. M. Sobha, G. C. Ravindra, and R. Pitchiah. Design, implementation and performance of fault-tolerant message passing interface (mpi). In *Proceedings. Seventh International Conference on High Performance Computing and Grid in Asia Pacific Region, 2004.*, pages 120–129, 2004.

[35] The HDF Group. Hierarchical data format version 5, 2000-2010.

[36] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott. Proactive process-level live migration in hpc environments. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–12, 2008.

[37] John W. Young. A first order approximation to the optimum checkpoint interval. *Commun. ACM*, 17(9):530–531, September 1974.