Recovery of Distributed Iterative Solvers for Linear Systems Using Non-Volatile RAM

Yehonatan Fridman[1, 2], Yaniv Snir[1, 3], Harel Levin[4, 5], Danny Hendler[1], Hagit Attiya[6] and Gal Oren[5, 6] ✉

[1]Department of Computer Science, Ben-Gurion University of the Negev
[2]Israel Atomic Energy Commission
[3]Google
[4]Mobileye Vision Technologies
[5]Scientific Computing Center, Nuclear Research Center – Negev
[6]Department of Computer Science, Technion – Israel Institute of Technology

{fridyeh, yanivsn}@post.bgu.ac.il, harellevin@nrcn.org.il,
hendlerd@cs.bgu.ac.il, {hagit, galoren}@cs.technion.ac.il

Abstract—HPC systems are a critical resource for scientific research and advanced industries. The increased demand for computational power and memory ushers in the *exascale era*, in which supercomputers are designed to provide enormous computing power to meet these needs. These complex supercomputers consist of numerous compute nodes and are consequently expected to experience frequent faults and crashes.

Mathematical solvers, in particular, *iterative linear solvers* are key building block in numerous large-scale scientific applications. Consequently, supporting the recovery of *distributed* solvers is necessary for scaling scientific applications to exascale platforms. Previous recovery methods for iterative solvers are based on Checkpoint-Restart (CR), which incurs high fault tolerance overhead, or intrinsic fault tolerance, which require extra computation time to converge after failures.

Exact state reconstruction (*ESR*) was proposed as an alternative mechanism to alleviate the impact of frequent failures on long-term computations. ESR has been shown to provide exact reconstruction of the computation state while avoiding the need for costly checkpointing. However, ESR currently relies on volatile memory for fault tolerance, and must therefore maintain redundancies in the RAM of multiple nodes. This not only incurs high memory overhead but also prevents ESR from being *fully resilient*, that is, resilient against a full system crash.

Recent supercomputer designs feature emerging *non-volatile RAM* (*NVRAM*) technology, for example, the exascale *Aurora* that is planned to consist of Intel Optane™ DCPMM. This paper investigates how NVRAM can be utilized to devise an enhanced ESR-based recovery mechanism that is more efficient and provides full resilience. Our mechanism, called *in-NVRAM ESR*, provides full resiliency while significantly reducing both the memory footprint and the time overhead in comparison with the original ESR design (*in-RAM* ESR). In-NVRAM ESR is based on a novel MPI One-Sided Communication (OSC) over RDMA implementation, which was optimized and applied for using NVRAM to store recovery data for iterative linear solvers.

The source code used in this work, as well as the benchmarks and other relevant sources, are available at: **https://github.com/Scientific-Computing-Lab-NRCN/In-NVRAM-ESR.git**.

Index Terms—Iterative Solvers, Recovery, HPC, Exascale, NVRAM, Intel Optane DCPMM, MPI OSC, RDMA, ESR, PCG

I. Introduction

A. Fault Tolerance in Supercomputing

The past decade has seen a skyrocketing increase in the demand for high-performance computing (HPC) systems, in order to meet the computing power needs of resource-hungry applications in various science domains. This ushered in a new exascale era of stronger and more complex supercomputers. For example, the first exascale supercomputer, *Frontier* [1], consists of more than 8 million compute cores that provide peak computing power of 1.102 Exaflop/s. With the increase in complexity and the number of compute nodes, comes increased vulnerability to failures. Already for earlier generations of petascale supercomputers, Schroeder and Gibson [2] showed that in certain situations, applications are forced into recovery more than twice a day. It is further anticipated that exascale systems will experience various kinds of faults every few hours or even every few minutes, in spite of hardware-based protection mechanisms [3], [4], leading to a growing need for mechanisms to ensure efficient recovery from failures. *Mathematical solvers* are a key component of scientific applications, especially in HPC, accounting for 50-90% of their operations [5]. Efforts were made to develop and maintain distributed, efficient and scalable libraries for these solvers. A prominent example is *Trilinos* [6], a collection of distributed and parallel reusable scientific software libraries that is widely used in high-performance scientific computing and includes linear, non-linear, transient and optimization solvers. Supporting the recovery of mathematical solvers is therefore necessary for scaling scientific applications and performing them on exascale platforms. Indeed, prior research [7]–[11] studied soft and hard faults resilience aspects in Trilinos.

Checkpoint-Restart (CR) techniques using non-volatile storage such as HDDs and SSDs are the most general and direct mechanisms to support recovery of scientific applications [12], [13]. Extensive work was done to provide improved CR models in order to minimize performance loss [14], [15]. However, CR

has inherent limitations, especially for scientific computations with large data sets, where checkpointing the partial image of an application (let alone the full one) requires to transfer large amounts of data to high-latency devices. Excessive checkpointing will result in total performance degradation.

Other work investigates fault-tolerance aspects in mathematical solvers, exploiting intrinsic attributes of solvers to tolerate faults, alleviating the need for checkpointing [11], [16]–[18]. For example, Sao and Vuduc [18] investigate self-stabilizing iterative solvers that require only lightweight tests for fault-detection. However, these solutions typically do not guarantee correct recovery, or require a significant extra computation time to converge after recovery.

B. NVRAM in HPC

Next-generation supercomputers are expected to incorporate novel *non-volatile memory* (*NVM*); for example, the flagship *Aurora* [19] will integrate NVM devices such as the Intel Optane™ DCPMM. When configured in App-Direct mode, these devices are byte-addressable and can be used by processes as *non-volatile random access memory* (*NVRAM*).[1] NVRAM's ability to retain data even after node failures opens new possibilities for HPC, most notably, as a medium for data persistence upon node or process failure and recovery.

Prior work on the use of NVRAM in HPC environments was confined to three main direct use-cases: (1) memory expansion to enable larger memory scientific workloads [20], [22]–[24], (2) fast storage for diagnostics [22], [25], [26], and (3) fast persistence area for checkpointing [22], [27]. Use cases (2) and (3) rely on the non-volatility of NVRAM for fast storage as a substitute for standard storage mediums [22], [25], [26]. For example, the DAOS storage server [25] is one of the promising storage systems for massively distributed NVM (NVMeSSD/+NVRAM); it is already in use by supercomputers at the top of IO500 list [28].

In contrast to these use-cases of NVRAM (for diagnostics, checkpointing and memory expansion), recent research investigates the recovery of concurrent data objects using NVRAM [29]–[35]. These works focus on achieving correct recovery with marginal footprint of the recovery data and minimal time overhead mechanisms. NVRAM has limited endurance and can tolerate a limited number of writes [36] — a fact that increases the need for clever recovery models that emphasize on minimizing memory footprint of the persisted data. However, all of those works are mostly theoretical and were not geared towards HPC applications.

C. Exact State Reconstruction of Linear Iterative Solvers

Iterative linear solvers, commonly used in HPC, offer inherent recoverability capabilities. They are mostly suitable for linear problems involving many variables (sometimes on the order of millions), where direct methods would be prohibitively expensive [37]. In Trilinos, these solvers are provided in the

Anasazi [38] and *Belos* [39] libraries, and include CG [40], GMRES [41], Jacobi [42], SOR [43] and more. Given initial approximations, such methods converge to the correct result by improving the quality of their result over time. This provides intrinsic tolerance to errors, as the solver has the potential to converge from any initial guess.

The intrinsic fault tolerance to errors of iterative linear solvers introduces a trade-off with CR mechanisms: Iterative solvers can inherently tolerate inconsistent data during convergence, although this might require extra computation time to converge [27], [44]. In contrast, CR mechanisms checkpoint the exact state of the solver and computation continues in recovery from the last checkpoint. Calculations performed after the last checkpoint are lost upon a failure, hence frequent checkpointing minimizes the rollback cost. However, excessive checkpointing creates significant storage access overheads and might double or even triple the memory footprint of the application. This is a major downside for scientific simulations with large data sets [27]. The gap between these two recovery mechanisms and their downsides calls for recovery methods that reduce the memory overhead without inaccuracies or imperfections, while crucial data is persisted directly into non-volatile yet fast memory, for seamless fault tolerance.

Yet another recovery scheme is to design recovery algorithms that tolerate failures according to the specific characteristics of an application [45]–[51]. A key example of such a scheme is *exact state reconstruction* (ESR). ESR is applicable to many distributed linear iterative solvers. Introduced by Chen et al. [46], [52], and refined by Pachajoa et al. [53]–[56], ESR is a recovery mechanism that achieves exact reconstruction of the computation state under failures while incurring significantly lower overhead than CR [53]. ESR provides fault tolerance by keeping redundancies of chosen process' state variables in the memory of other processes in run time; we therefore refer to it as *in-RAM* ESR.[2] Chen [52] introduced a way to use the *sparse matrix–vector multiplication* (*SpMV*) to redundantly store the input vector, exploiting the already existing transmission of data between processes. In this manner, redundant copies of the input vector can be produced with relatively low memory and runtime overheads. Pachajoa et al. [53]–[55] extended *in-RAM* ESR to support multiple node failures. However, to maximize fault tolerance, processes replicate their state vectors on each other compute node which comes with a significant memory expense. Additionally, in this case redundancies are sent all-to-all after each iteration (or after a certain period), leading to a surge in network traffic. To alleviate these problems, ESR can replicate the state at only a fraction α of the cluster, for example, at half of the nodes ($\alpha = 0.5$). It is also possible to use different strategies for selecting the nodes to keep the redundant copies for minimizing

---

[1]Alternatively, they may be configured in Memory or Flat mode, in which case that provide an extension to the volatile memory pool of the application [20], [21].

[2]We use the terminology *in-X*, where *X* is the target of the redundancy needed for the ESR operation. Specifically, *in-RAM* (as in the original ESR), *in-SSD* and *in-NVRAM*. We note that the target device does not directly imply the access fashion. Specifically, while RAM will be byte-addressable in the context of this paper, NVRAM can be either byte- (with *DAX* and RDMA operations, or byte-oriented PMFS) or block-addressable (with simple PMFS). Of course, SSDs and HDDs will be referred as block-oriented only.

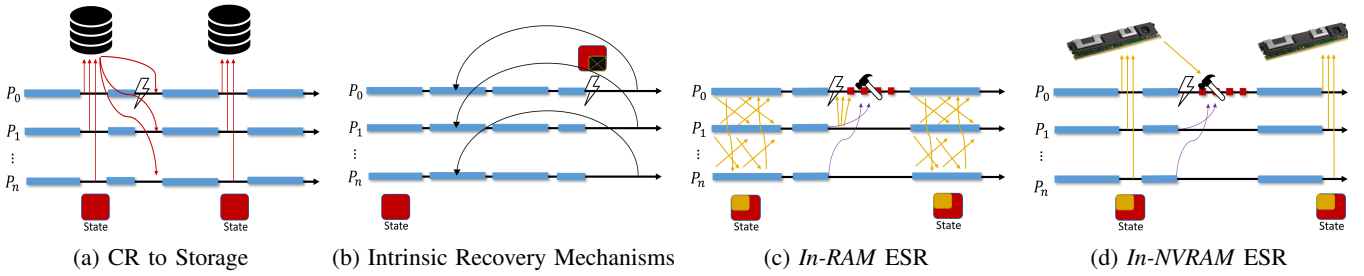| (a) CR to Storage | (b) Intrinsic Recovery Mechanisms | (c) *In-RAM* ESR | (d) *In-NVRAM* ESR |

Fig. 1: Recovery models for iterative solvers: (a) Checkpointing the state of the computation frequently to storage device; when a failure occurs, reading the complete state back and continuing. (b) fault corrupts parts of the state, accuracy is therefore lost and recomputation is required. (c) Saving copies of recovery data in other processes' RAM; when a failure occurs, collecting recovery data from other survived processes via messages, reconstructing the full state and continuing (d) Exact state reconstruction with saving recovery data to NVRAM directly; when a failure occurs, reading recovery data from NVRAM, reconstructing the full state and continuing.

	CR to Storage	Intrinsic Recovery Mechanisms	*In-RAM* ESR	*In-NVRAM* ESR
Pros	direct recovery, no extra calculations are needed to reconstruct the computation	minimal memory footprint and manipulation of recovery data	exact state reconstruction of failed processes	exact state reconstruction with marginal memory footprint of recovery data
Cons	frequent checkpoints to standard SSD or HDD are expensive (in memory and time)	reached state is lost, recovery requires extra calculations to make up for the lost accuracy	huge memory and networking overheads	NVRAM performances are still not comparable to RAM performances

TABLE I: Prominent Pros and Cons of the various recovery models described in Fig. 1.

communication overheads during SpMV [55]. Nevertheless, the original problem remains as the scale of computers grows (see Section III-A), leaving open the challenge of achieving full resilience while reducing the costs of ESR and improving its scalability.

D. Our contribution: In-NVRAM ESR

In this work we investigate how to significantly improve the performance of *in-RAM* ESR. Specifically, we show how its extended memory footprint and network traffic can be dramatically reduced. Our work rests on three pillars: (1) recently enabled capabilities of *direct access (DAX)* to NVRAM, (2) the access to such memory using MPI One-Sided Communication (OSC) over RDMA, and (3) the observation that these two capabilities allow maintaining all of the advantages of the original *in-RAM* ESR while persisting only a single copy of recovery data during each persistence cycle, instead of maintaining multiple redundancies. This yields the enhanced *in-NVRAM* ESR, which instead of relying on and populating the RAM with many redundancies for fault tolerance, sends just a single copy DAX-wise through RDMA directly to the persistent NVRAM. Accessing byte-addressable NVRAM directly, without incuring the latency of moving data to and from the I/O bus, with comparable performances to RAM, and while incurring only a small overhead, yields an enhanced ESR mechanism, without compromising data and recovery consistency.

We implement *in-NVRAM* ESR using the extension of MPI One-Sided Communication (OSC) over RDMA [57], [58] under the setting of NVRAM. We study two possible NVRAM placements architectures:

1) *Homogeneous NVRAM cluster*, in which each compute node is equipped with its own NVM module, enabling the persistence of ESR state variables to local NVRAM by using either the *persistent memory development kit (PMDK)* [59] libraries or a local MPI window.
2) *NVRAM persistent recovery data (PRD) sub-cluster*, in which recovery data is persisted in dedicated PRD sub-cluster nodes via remote MPI one-sided communication implemented using RDMA.

In the PRD sub-cluster architecture, we assume RAID between nodes to provide fault tolerance to errors in the sub-cluster. Otherwise, each node of the sub-cluster behaves as a single point of failure. We stress that while *in-RAM* ESR's data transportation increases quadratically with the cluster size (as explained in Section III-A), increase in writes for RAID is linear and depends on RAID level.

We note that, to the best of our knowledge, planned exascale supercomputers are expected to use the remote PRD NVRAM sub-cluster architecture [60]–[62]. This choice of architecture is mainly because integrating Optane DCPMM modules in each compute node would reduce the number of available DDR DIMM memory slots, thereby reducing the size of available DRAM, which is one of the crucial resources in HPC nodes [63]. Hence, future supercomputers integrating Optane DCPMM modules are designed with remote NVM storage nodes, e.g., the DAOS storage server [25] in Aurora [19], [60], [61]. Nevertheless, this work evaluates also the homogeneous NVRAM cluster architecture, as future systems might include NVRAM in each compute node, for fast storage or relatively cheap and fast byte-addressable memory expansion (see [22], [20]). Specifically, network and remote I/O performance char-

acteristics play a crucial role in the optimization of HPC applications running in HPC cloud systems (HPCaaS). Thus, future HPC cloud systems are integrating storage devices such as NVMe-based SSD locally [64]–[66], and even foundations for NVRAM integration are already established [67].

Fig. 1 describes the different types of recovery models for iterative solvers, while Table I summarizes their pros and cons. In summary, our work incorporates NVRAM recovery with exact state reconstruction techniques for distributed linear iterative solvers. We propose, implement and evaluate *in-NVRAM* ESR, a novel NVRAM-based mechanism for scalable and resource-efficient exact state reconstruction techniques. We implemented *in-NVRAM* ESR in the PRD sub-cluster architecture by using MPI one-sided communication over InfiniBand's remote direct memory access (RDMA) towards NVRAM and have optimized its usage by ESR's persistence iterations (see Section IV-A).

To the best of our knowledge, our work is the first to report on a scientific application implementation that accesses remote NVRAM in this manner.

We conducted a comprehensive performance evaluation, comparing *in-NVRAM* ESR to other implementations that store recovery data, either in DRAM (*in-RAM ESR*) or in SSD storage device (*in-SSD ESR*). Our evaluation shows that *in-NVRAM* ESR is significantly superior to *in-RAM* ESR in terms of the size of the memory footprint required for storing the recovery data. *In-NVRAM* ESR is also superior to *in-RAM* ESR and *in-SSD* ESR in terms of the time overhead incurred by writing the recovery data. Based on these results, we estimate that the small memory and time overheads of *in-NVRAM* ESR will allow it, unlike ESR, to be deployed in future exascale supercomputers for providing high resiliency to the important class of distributed iterative solver for linear systems algorithms for which ESR is suitable.

Organization: The rest of the paper is organized as follows. Section II describes the *in-RAM* ESR technique and its challenges for the scalability of linear iterative solvers to exascale systems. In section III an NVRAM-based solution is described and the *in-NVRAM* ESR model is suggested, applicable to the homogeneous NVRAM cluster architecture and the PRD sub-cluster architecture. In section III-A we compare the memory utilization of *in-RAM* ESR and *in-NVRAM* ESR and demonstrate significant savings in memory resources when persisting data to NVRAM with *in-NVRAM* ESR. In section IV we describe the implementation details of *in-NVRAM* ESR, focusing on the technologies and software that enable *in-NVRAM* ESR to access NVRAM locally and remotely. Section V presents the evaluation results of *in-NVRAM* ESR in a key example of the widely used PCG solver for sparse systems.

II. *In-RAM* ESR and Its Challenges

Exact state reconstruction (ESR) is a technique for recovering the state of a linear algebra iterative solver after a failure, avoiding the checkpointing of the entire state of the computation. ESR takes advantage of concurrent data distributed

between nodes to reconstruct the state [55], [68]. Specifically, it exploits *sparse matrix-vector multiplication* (*SpMV*) operations to produce redundant copies to vectors with low memory and runtime overheads. Therefore, ESR is applicable to iterative solvers that involve *SpMV* and perform a finite-term recurrence (hence the state can be reconstructed from a bounded amount of previously calculated data).

ESR first identifies the state of the solver. Upon recovery, the redundancy of vectors participating in the SpMV operations is used to reconstruct the other state vectors, by solving local equations on a replacement node. When the full state of the failed process is reconstructed, the computation can proceed on the replacement node. In ESR, whenever the SpMV operation is applied, *i.e.*, $Av^{(j)}$ is computed for some state variable v at iteration j, the transition of v-values is augmented to create redundancy for all its entries. This augmented SpMV operation is denoted by ASpMV [54]. Redundancies are created on other processes' RAM, hence this model is referred to as *in-RAM* ESR (see Fig. 2a).

A generic method for this state identification for iterative solvers is described in by Pachajoa et al. [56]. This meta-algorithm uses the dependency graph of an iterative algorithm to decide which variables have to be saved and which can be reconstructed by using other variables. Two main examples demonstrated are the PCG [40] and the BiCGStab [69] solvers. We stress that these two solvers are examples of a larger class of iterative solvers that can be adjusted to ESR. For example, in Jacobi, SOR and Gauss-Seidel the solution approximation variable will keep redundancies, and in MINRES and GMRES it will be the Arnoldi vectors [41]. For a given distributed linear iterative solver produced with an ESR algorithm, we denote the variables of the full state by the set V, and the subset of ESR variables that participate in SpMV operations and create redundancies by V_{SpMV} ($V_{SpMV} \subseteq V$). We denote by n the size of each global vector of the solver. We refer to the set of all indices as $I = \{1 \ldots n\}$. The indices corresponding to a certain process p are denoted by I_p. Specifically, f denotes a failed process, and its indices are denoted I_f. The value of variable v at the jth iteration is denoted by v^j.

In the reconstruction phase of failed process f, the redundancies of $(V_{SpMV})_{I_f}$ are collected to a replacement node, together with the values of $V_{I \setminus I_f}$ and $V_{I \setminus I_f}$ that are needed for reconstruction from the surviving nodes. For some multi-term recurrence solvers, reconstruction requires k successive values of $(V_{SpMV})_{I_f}$, and therefore redundancies for such variables are kept for k iterations. For example, in the ESR algorithm for the Preconditioned Conjugate Gradient (PCG) solver, presented in [52], $k = 2$ as it stores the last two search directions via ASpMV (the PCG solver is a two-term recurrence). Solving some local linear equations, $(V \setminus V_{SpMV})_{I_f}$ is reconstructed, ending up with the full state of the process, which enables to continue the computation.

To tolerate multiple node failures, the redundancy should be saved in multiple copies. Thus, even when several nodes

crash together, values can still be recovered from the RAM of processes on the surviving nodes. If c nodes may fail simultaneously, c redundant copies should be made. Copies should be saved on different nodes, since if a node crashes, all of its processes fail together. In this case, I_f represents the indices of all the failed processes together, and the reconstruction algorithm is executed on several nodes, solving the local equation systems together distributively. The reconstruction effort depends on the number of failed processes. As the number of processes that fail simultaneously increases, the size of the equations that must be solved for reconstructing the lost data grows.

There is a delicate balance between the runtime overhead required to save the recovery data of the application securely, and the time it takes to recover after a failure. ESRP [54] is a modification of ESR, where redundant copies are created periodically to alleviate the networking overhead for each iteration. ESRP demonstrates a trade-off, where increasing the period of ESR decreases the runtime overhead, but increases the cost of discarding the iterations performed since the last copies were made when recovery is required.

ESR aims to minimize the amount of data being persisted, by a careful analysis of the application that identifies a minimum-sized set of variables whose state should be made persistent. These variables must be chosen such that all other significant variables can be reconstructed from their values. A generic method for this state identification for iterative solvers is described in [56]. This generic meta-algorithm produces ESR algorithms for various solvers. Two main examples demonstrated by Pachajoa are the PCG [40] and the BiCGStab [69] solvers. We stress that these two solvers are examples of a larger class of iterative solvers that can be adjusted to ESR. For example, in Jacobi, SOR and Gauss-Seidel the solution approximation variable will keep redundancies, and in MINRES and GMRES it will be the Arnoldi vectors [41]. *In-RAM* ESR has three main limitations. First, a large number of copies should be made to redundant data to ensure the recovery when a large number of nodes fail simultaneously. This creates a dramatic increase in memory overhead, and therefore effectively reduces the size of the problems that can be solved. Second, *In-RAM* ESR suffers from networking and time overheads, as the redundancies are sent between nodes simultaneously — leading to a surge in network traffic. Finally, *In-RAM* does not rely on NVRAM, which is a main disadvantage for future supercomputers that are planned to integrate NVRAM. NVRAM-based systems offer an attractive alternative solution to these challenges, as we describe next.

III. *In-NVRAM* ESR: Overview

To exploit NVRAM for ESR recovery, we use it for persisting the redundant data required for it instead of keeping it in the memory of other nodes. We name this mechanism *in-NVRAM* ESR. In NVM-based systems, processes have access to persistent memory space, in addition to a unique volatile memory space available for every process. Nodes crash arbitrarily and independently of each other. Upon a crash of a certain node,

the content of its volatile memory is lost, and the data in its persistent memory (if it includes NVM) becomes inaccessible until the node recovers.

There are two key possible ways to integrate NVRAM devices into a cluster architecture. In the first, more traditional architecture, called *homogeneous NVRAM cluster*, each node has an NVRAM device attached to it (see Fig. 2b). Each node's state is saved into its local NVRAM whose coherency should be ensured by the application. If a node fails, the persisted data can be recovered from the NVM once it recovers. Therefore the reconstruction algorithm for failed process f is executed on the same node that f executed on before the failure, and only after it recovers.

In the second architecture, there is a sub-cluster of one or more *persistent recovery data* (*PRD*) nodes, each containing NVRAM modules, which stores recovery data for the rest of the cluster's *compute nodes* (see Fig. 2c). In this architecture, recovery data is saved via RDMA operations with the remote memory MPI's One-sided API (to reduce overheads). (The details of how this is done are explained in Section IV-A.) When a process f fails, its recovery data is accessible on the NVRAM sub-cluster and can be retrieved by a replacement node. For each of the failed processes, the reconstruction can be executed on every compute node that has access to the NVRAM sub-cluster. A failure of a PRD node renders its memory inaccessible, which may make it a single point of failure. This type of failure can be addressed by adding redundancy within the PRD sub-cluster itself (for example using RAID), but this is outside the scope of this paper.

The *in-NVRAM* ESR persistence stage, as well as the *in-NVRAM* ESR reconstruction phase for both these architectures, appear in Algorithm 1 and Algorithm 2, respectively. We do not deal here with the trade-off between the ESR period and the number of "wasted" iterations upon a failure demonstrated in ESRP [54] and simply assume that this period is chosen optimally according to the problem and the cluster character-istics. We focus instead on a single persistence iteration during the calculation.

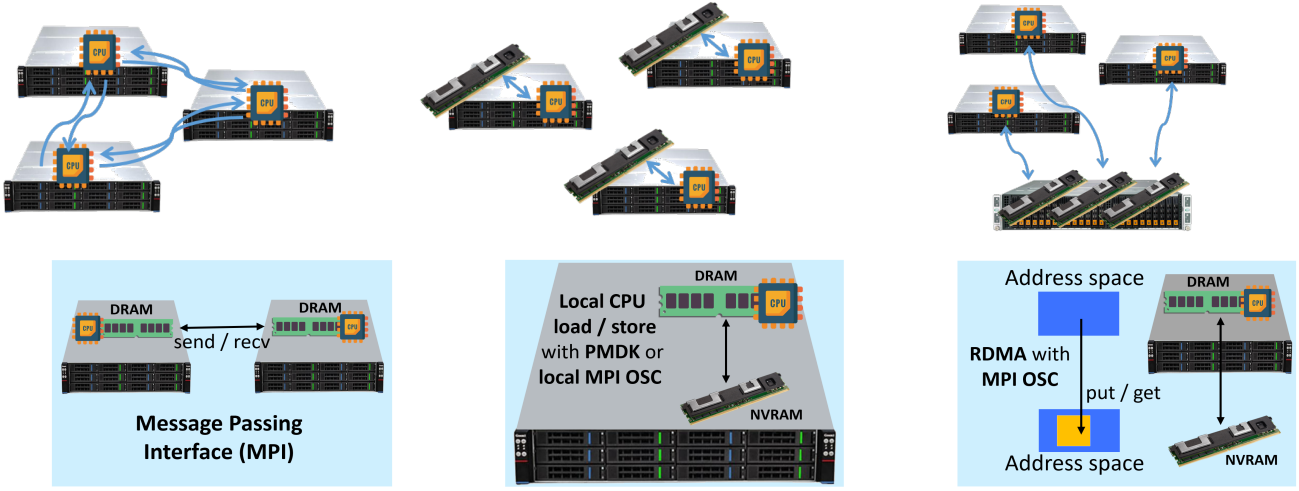### A. Comparing in-RAM ESR and in-NVRAM ESR

Let N denote the number of compute nodes; The number of the actual compute processes used by the solver is $proc \leq t \cdot N$. $M_R(n, proc, \phi)$ denotes the amount of memory overhead required by a recovery method R to support the recoverability of up to ϕ simultaneous node failures. We assume that the

---

Algorithm 1 *In-NVRAM* ESR for a persistence iteration j of process p

---

Compute j^{th} iteration of the solver
   $\vdots$
if Homogeneous Cluster **then**
   persist $(V_{SpMV})_{I_p}^j$ to **local** NVM
if PRD sub-cluster **then**
   persist $(V_{SpMV})_{I_p}^j$ to **remote** NVM

(a) Volatile memory architecture.   (b) Homogeneous NVRAM cluster.   (c) Persistent recovery data (PRD) sub-cluster.

Fig. 2: Cluster architectures with and without the usage of NVRAM.

Algorithm 2 *In-NVRAM* ESR Reconstruction phase of iteration j of failed process f

---

if HOMOGENEOUS CLUSTER **then**

    Wait for failed nodes to recover and have access to **local NVM**

if PRD SUB-CLUSTER **then**

    Run reconstruction from any spare nodes that have access to **remote** NVRAM Sub-Cluster

---

Retrieve the static data of the solver

Gather required variables from $V_{I \setminus I_f}^{(j)}$

if HOMOGENEOUS CLUSTER **then**

    Read $\{(V_{SpMV})_{I_f}^i\}_{i=j-k+1}^j$ from **local** NVM

if PRD SUB-CLUSTER **then**

    Read $\{(V_{SpMV})_{I_f}^i\}_{i=j-k+1}^j$ from **remote** NVM

$\vdots$

Solve local linear systems to reconstruct $(V \setminus V_{SpMV})_{I_f}^j$

---

vulnerability of the system is proportional to the number of nodes, hence $\phi = \alpha \cdot N$ for some constant $\alpha < 1$. In the following, we estimate $M_R(n, proc, \phi)$, where R={*in-RAM* ESR, *in-NVRAM* ESR}. In the *in-RAM* ESR mode, redundancies for the variables in V_{SpMV} are saved for successive k iterations in ϕ nodes, therefore

$$M_{in\text{-}RAM\ \text{ESR}}(n, proc, \phi) = \Sigma_p k \cdot |(V_{SpMV})_{I_p}| \cdot \phi$$
$$= k \cdot |V_{SpMV}| \cdot \phi = O(|V_{SpMV}| \cdot \phi)$$

elements in memory. In the *in-NVRAM* ESR mode, a single copy (up to certain RAID level) is saved in NVRAM for the variables in V_{SpMV}, therefore

$$M_{in\text{-}NVRAM\ \text{ESR}}(n, proc, \phi) = O(\Sigma_p k \cdot |(V_{SpMV})_{I_p}|)$$
$$= O(|V_{SpMV}|)$$

elements in persistent area (where the $O()$ notation hides the constant of RAID level).

For sparse linear problems of matrices of size $n \times n$, the representation of the problem is linear in n. In addition, the size of the problem is estimated to be proportional to N (as more nodes add more RAM). Moreover, $|V_{SpMV}|$ is proportional to n as V_{SpMV} consists of global vectors of size n. Therefore we conclude that $|V_{SpMV}|=O(N)$. Remember that typically ϕ is chosen in proportion to N ($\phi = \alpha \cdot N$). Finally, we write $M_{in\text{-}RAM\ \text{ESR}}(n, proc, \phi) = O(N \cdot N) = O(N^2)$, while $M_{in\text{-}NVRAM\ \text{ESR}}(n, proc, \phi) = O(N)$. These estimations demonstrate that *in-NVRAM* ESR is much more scalable than *in-RAM* ESR as the former incurs fault tolerance overheads in memory that increase linearly with the cluster size, while the latter incurs memory overheads that increase quadratically. Moreover, while *in-RAM* ESR uses the RAM to save the huge amount of recovery data, *in-NVRAM* ESR uses persistent memory for its recovery data, leaving more space in RAM for larger computations.

B. Comparing In-NVRAM ESR and In-SSD ESR

Checkpointing scientific applications, either transparently (for example with DMTCP [13]) or explicitly (for example with SCR [12]), persists all the data allocated by the application to a block device, so restarting is enabled by reading the last available checkpoint into the application's buffers. Usually, and especially for distributed linear iterative solvers, not all the data is necessary for exact reconstruction, as the exact state can be reconstructed from only a partial checkpoint. ESR alleviates the full state's checkpoint by persisting only a subset of the state and computationally reconstructing the rest of the state upon recovery.

In-SSD ESR uses a block SSD device to persist recovery data. This includes all the I/O stages, from system call invocation, block granularity writes, and data transferring via the I/O bus. All these add additional overhead to the high latency of block SSD devices. In contrast, the *in-NVRAM* ESR mechanism persists the recovery data to the NVRAM directly, either locally via the memory bus or remotely via RDMA. Unlike

block-based systems, this eliminates the intervention of the operating system by directly accessing the byte-addressable NVRAM. Accessing the NVRAM in this manner is much faster and, in fact, persistence operation only access the application's address space, unlike when accessing SSD.

IV. The Implementation in Detail

This section describes the various ways in which *in-NVRAM* ESR can use the system's capabilities to persist data to NVRAM.

A. MPI One-Sided Communication (OSC) over RDMA

Remote direct memory access (RDMA) support in networks became mainstream, particularly through the widespread adoption of InfiniBand as a commodity network fabric [70]. MPI-1 provides a powerful and complete interface for the message-passing approach. MPI-2 added (and MPI-3 greatly extended) remote memory operations that provide a way to access the memory of remote processes directly, through operations that put data to, get data from, or update data at a remote process. Unlike message passing (using standard *send* and *receive* operations), the program running on the remote process does not need to call any routines to match the *put* or *get* operations. Thus, remote memory operations can provide better performance for distributed and parallel programs. This functionality of remote memory access is implemented via *memory windows*. The term *window* is used since MPI limits what part of a process's address space is accessible to other processes.

Several works [57], [71], [72] present different persistence schemes to correctly utilize RDMA over NVRAM. Dorożyński et al. [57] incorporate non-volatile RAM into wrappers over MPI One-Sided API, in order to provide persistence of data stored in MPI windows. They extend the RDMA MPI One-Sided Communication (OSC) to persist the data stored in windows to NVRAM. Data consistency is ensured by copying necessary data into a separate location. More specifically, the data is saved into two locations alternately in order to have at least one proper "checkpoint" even if a failure occurs during checkpointing creation. An implementation of this method [58] provides a new programming model by allowing processes to communicate freely using standard OSC functions and fall back to a state saved during synchronization. It supports synchronization calls of OSC, as explained next.

RMA communication calls of the MPI ISC API must occur in the invoking process only within an *access epoch* for the window. The transferred data is available only when exiting the access epoch. Such an epoch starts with an RMA synchronization call on the window; it then proceeds with zero or more RMA communication calls (e.g., `MPI_PUT` or `MPI_GET`) on the window; it completes with another synchronization call on the window [73]. RMA communications fall in two categories: *active* and *passive* target communication.

In *active target communication*, the data is moved from one process to another, and both processes are explicitly involved in the communication. In contrast to standard message passing,

in active target communication RMA operations are controlled only by the invoking process, and the target process only participates in the synchronization. In active target communication, a target window can be accessed by RMA operations only within an *exposure epoch*. Distinct exposure epochs at a process on the same window must be disjoint, but such an exposure epoch may overlap with multiple access epochs for the same window. MPI provides two synchronization mechanisms for active target communication:

1) A general collective RMA synchronization, in which an access epoch at an origin process or an exposure epoch at a target process are started and completed by calls to `MPI_Win_fence`.
2) Synchronization in which only pairs of communicating processes synchronize using the *Post-Start-Complete-Wait* (PSCW) protocol. In PSCW, an access epoch is started at the origin process by a call to `MPI_Win_Start` and is terminated by a call to `MPI_Win_Complete`. An exposure epoch is started at the target process by a call to `MPI_Win_Post` and is completed by a call to `MPI_Win_Wait`. The post-call has a group argument that specifies the set of origin processes for that epoch.

In *passive target communication*, the target process does not execute RMA synchronization calls, and there is no notion of an exposure epoch. Instead, passive target synchronization is accomplished by using MPI locks at the origin process with `MPI_Win_Lock` and `MPI_Win_Unlock`. It is used to ensure that RMA operations from other processes do not modify the data unexpectedly.

Dorożyński et al. [58] support both communication mechanisms in their extension of MPI OSC over NVRAM. For the active target synchronization, exposure epochs are closed with `MPI_Win_Fence_persist` (for active target communication) or `MPI_Win_Wait_persist` (for passive target communication), to ensure that data reaches NVRAM before exiting the exposure epoch.

To implement *in-NVRAM* ESR in the NVRAM PRD sub-cluster architecture, it is necessary to ensure that data is persisted successfully to the NVRAM in the PRD node after each persistence iteration and before the successive persistence iteration attempts to access the window. Since the target process must know when the exposure epoch is closed, an active target mechanism is more suitable for *in-NVRAM* ESR. Since a persistence iteration usually requires a significant period of time, we can optimize by releasing the access epochs of the compute processes while the target process is still persisting the data in its exposure epoch. This allows compute processes to proceed to the next compute iteration. For this reason, we choose the PSCW mechanism to be applied in *in-NVRAM* ESR. Within the access epoch, a process executes a `MPI_Win_Put_pmem` to transmit data to the remote process, and `MPI_Win_Get_pmem` to read the data when recovery is needed. Whenever a compute process completes its RMA operations with `MPI_Win_Complete`, it exits the access
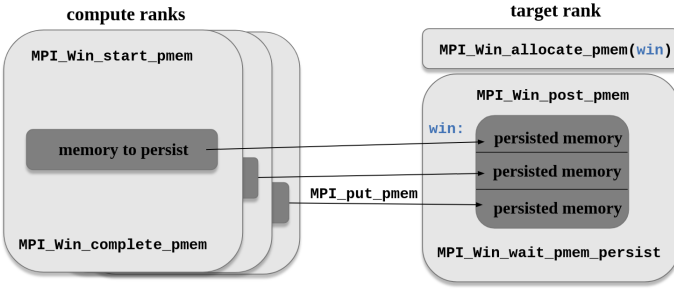
Fig. 3: Persisting ESR data using MPI OSC epoch over RDMA to a remote NVRAM node, using PSCW.
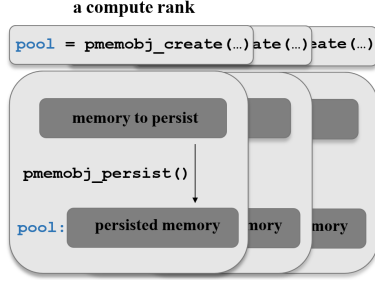

Fig. 4: Persisting data with *libpmemobj* directly to local NVRAM in the homogeneous NVRAM cluster architecture.

epoch and proceeds. Fig. 3 illustrates a PSCW epoch for the MPI OSC to an NVRAM PRD node in a persistence iteration. A similar implementation for the homogeneous NVRAM cluster architecture, which we also consider, uses a separate local window for each process. In this architecture, each process accesses its own local window and persists its data locally.

B. Persistent Memory Development Kit (PMDK)

PDMK [59] offers a set of user-space APIs and interfaces to interact with non-volatile memory, with support for multiple abstraction layers, over Linux and Windows. PMDK libraries are designed to leverage the direct access allowed by persistent memory as much as possible. Persistent libraries in PMDK help applications maintain data structure consistency in the presence of failures. One such library is *libpmemobj*, which helps the programmer manage persistent memory arrays and data structures. We implement *in-NVRAM* ESR using the *libpmemobj* library to persist ESR data directly to local NVRAM in the homogeneous NVRAM cluster architecture. Each process first creates a persistent memory pool using a call to pmemobj_create and then, at each persistence iteration, persists ESR data using pmemobj_persist. Fig. 4 illustrates a persistence iteration using *libpmemobj* in the homogeneous NVRAM cluster architecture.

PMDK also provides *librpmem*, a remote RDMA access library, which supports remote access to persistent memory, with a synchronous write model: The local initiator writes and all of the remotely replicated writes must complete before the local write returns to the application. This library can be useful in the implementation of *in-NVRAM* ESR in the PRD architecture, although this is out of the scope of this paper.

C. Persistent Memory File Systems (PMFSs)

We also consider persisting data to NVRAM using PMFSs, which can be either local or distributed. PMFSs often exploit the byte-addressability of NVRAM, and support a special Direct Access (DAX) mode, which enables memory mapping directly from the NVRAM to the application memory space. DAX bypasses the kernel, page cache, and I/O subsystem, avoids interrupts and context switching, and allows the application to perform byte-addressable *load/store* operations [74]. Fridman et al. [22] present an evaluation of local PMFSs (e.g., ext4-DAX and SplitFS [75]) for writing and reading diagnostics of scientific applications, as well as for performing C/R of these applications using transparent or explicit checkpointing. While these local PMFSs are good choices for single-node workloads, scientific computing applications typically require a distributed PMFS operating on multiple nodes. *in-NVRAM* ESR can utilize local PMFSs in the homogeneous NVRAM cluster architecture to persist the recovery data of each process locally in its node. Distributed PMFSs, however, control how data is stored and retrieved from NVM when accessed by more than one node, using network communication. Distributed PMFSs can be utilized by *in-NVRAM* ESR in the PRD sub-cluster architecture to persist recovery data remotely on the NVRAM present on PRD nodes.

In this work, we implement *in-NVRAM* ESR for the homogeneous NVRAM cluster architecture by using ext4-DAX as a local PMFS over the local NVRAM.

V. KEY EXAMPLE: PCG SOLVER

We focus on the *preconditioned conjugate gradient* (*PCG*) iterative solver also studied in prior ESR research, because it is commonly used for solving sparse linear systems and employed by the representative HPCG scientific benchmark [76]. PCG solves the linear equation $Ax = b$ for a symmetric positive definite matrix $A_{n \times n}$. An ESR algorithm for PCG has been derived by the generic meta-algorithm described in [56]. To recover from node failures, PCG saves redundancies for the search direction variable. (see Appendix A for more details). We use our experimental cluster (which contains 8 nodes), described in Table II, to simulate the recovery patterns in a larger cluster of 256 nodes, as each core acts as a difference compute node. Hence, to support full fault tolerance ($\phi = \#processes$), each process sends redundancies to all other processes. To support the recovery in more reliable systems, each process sends redundancies to a certain fraction of the processes ($\phi = \alpha \cdot \#processes$). In our experiments we examine half fault tolerance, when recovery data is copied to half of the processes ($\alpha = 0.5$). Fig. 5 depicts the memory usage of PCG for the 7-point stencil of a 3-D Poisson equation. RAM utilization is fixed per process, so the total memory usage increases linearly with the number of processes. The figure shows that the size of problems that can be handled using *in-RAM* ESR decreases because recovery data occupy a significant part of the node DRAM.

	Compute Nodes	NVRAM Storage Node
#Nodes	8	1
#Sockets	2 (per node)	2
CPU Spec.	16 Cores × Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz (per socket)	10 Cores × Intel(R) Xeon(R) Gold 5215 CPU @ 2.50GHz (per socket)
L1 Cache	32KB i-Cache 32KB d-Cache (per core)	
L2 Cache	1024KB (per core)	
L3 Cache	22528KB (shared, per socket)	14080KB (shared, per socket)
DRAM Spec.	32GB DDR4 DRAM 2666 MT/s	16GB DDR4 DRAM 2933 MT/s
Total DRAM	128GB (per node)	192GB
NVM Spec.	none	256GB Intel Optane™ DCPMM 2666 MT/s Apache Pass
Total NVM	none	1024GB [(2 sockets)× (2 channels)×256GB]
SSD Spec.	none	240GB 6GB/s Intel SATA 2.5" SSD
Network	56Gb/s Mellanox Infiniband FDR	
HT	disabled	
OS	Linux CentOS 7	Linux CentOS 7.9

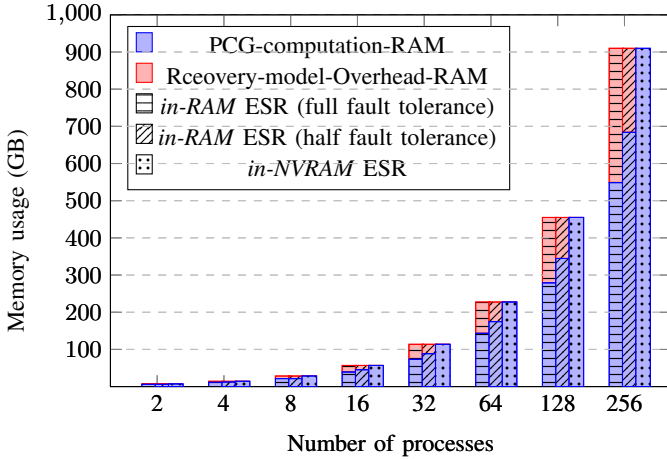TABLE II: Experimental cluster specifications.

Fig. 5: RAM usage for calculation and recoverability PCG for a 7-point stencil of a 3-D Poisson equation.
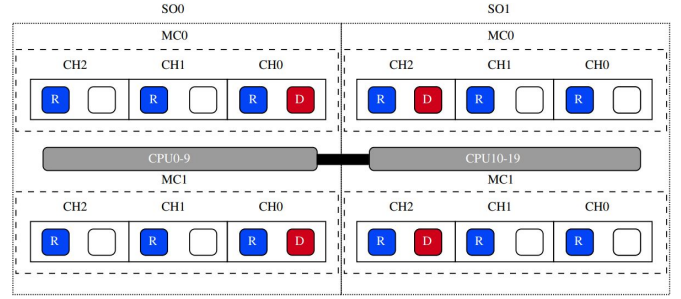

Fig. 6: DCPMM population configuration of the NVRAM Storage Node, with two sockets (SO) connected via an interconnect. Each CPU has two memory controllers (MC), each providing three memory channels (CH); each memory channel contains two DIMM slots. D (red) denotes DCPMM, R (blue) denotes DRAM (RDIMM); white denotes vacancy.
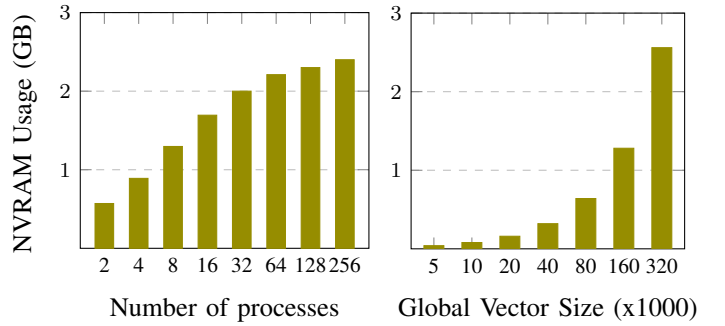

Fig. 7: NVRAM usage by *in-NVRAM* ESR as a function of the number of processes in a node (left) and as a function of the problem's vector size (right).

Extrapolation to Aurora: Aurora [77] will consist of 9000 compute nodes, each with 112 CPU cores ($\sim 10^6$ cores total). Total system memory is estimated at 10PB. It is expected to have over 230PB of high-performance storage, including Intel Optane™ DC SSDs and DCPMMs (with DAOS). For a PCG 7-point stencil of a 3-D Poisson equation, extrapolating the *in-RAM* ESR PCG RAM consumption presented in Fig. 5 into Aurora scale, we estimate *in-RAM* full fault tolerance ESR RAM consumption to be $\sim$30% of the system memory, hence $\geq$ 3PB. *In-NVRAM* ESR suggests eliminating this memory overhead at the expense of only 3PB $\div$9000 $=\sim$ 0.3TB usage of NVRAM because every value that resides in the RAM of $\sim$ 9000 nodes can now be persisted to NVRAM only once (or with certain RAID level).

A. Evaluation

To evaluate *in-NVRAM* ESR's performance, we employed it for the PCG solver and compared it with the fully fault tolerant ESR. Our experimental cluster specifications are listed in Table II. Our cluster consists of 8 compute nodes (each with 32 compute cores and 128GB DRAM) and a single NVRAM node (with 20 compute cores, 192GB DRAM, and 1TB Intel Optane™ DCPMM, populated as depicted by Fig. 6).

We evaluated both the homogeneous NVRAM cluster and the NVRAM PRD sub-cluster architectures. As our cluster contains only a single NVRAM node, our evaluation of the homogeneous NVRAM cluster architecture is limited to 20 processes. Nevertheless, this evaluation can be reliably extrapolated to a multi-node homogeneous NVRAM cluster architecture, since persisting data from each process to its local node is an embarrassingly parallel workload at the node level. To evaluate *in-NVRAM* ESR/PRD, we use the NVRAM node in our experimental cluster as a single PRD node. Multiple NVRAM nodes can serve as the PRD sub-cluster, distributing the persistent data to eliminate bandwidth bottlenecks or creating RAID over the sub-cluster to increase fault tolerance. Fig. 7 shows an estimate of NVRAM utilization of *in-NVRAM* ESR on our cluster. The graph on the left shows the amount of NVRAM required for different numbers of processes when a fixed amount of RAM is available to each process, and the size of the problem grows to fit available RAM. The graph on the right shows the NVRAM utilization for different sizes of the global input vector of the problem. As the global vector is split between the processes and each process persists its local parts to the NVRAM, the total number of values that are persisted to NVRAM is equal to the global vector size. We next present the time overheads of a single persistence
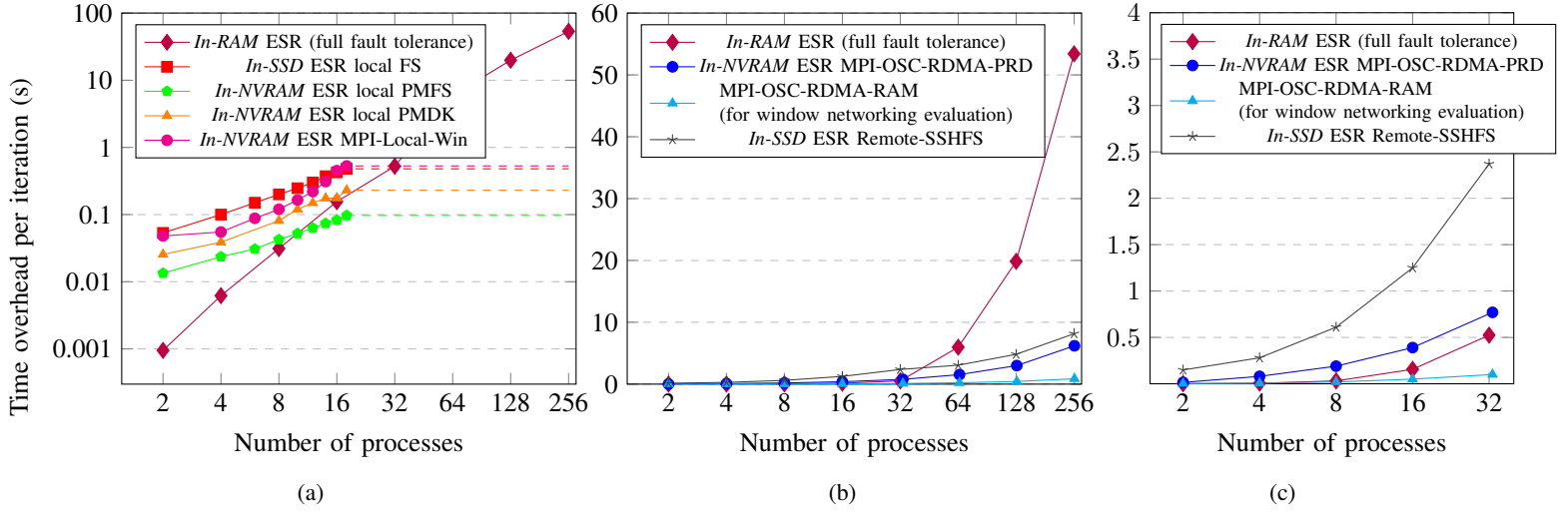
Fig. 8: (a) Time overhead (in log-scale) for *in-NVRAM* ESR and *in-RAM* ESR for single persistence/redundancy iteration in crash-free CG computation in the homogeneous cluster architecture. (b) Time overhead for *in-NVRAM* ESR and *in-RAM* ESR for single persistence/redundancy iteration in crash-free CG computation in the PRD sub-cluster architecture. (c) Zoom in (in log-scale) on chart 8b with ≤ 32 processes.

iteration in *in-NVRAM* ESR PCG for a 7-point stencil of a 3-D Poisson equation (with a fixed size for local vectors of 176,400 entries each). The time overheads of a single redundancy iteration of *in-RAM* ESR with full fault tolerance are presented as well. We focus on the time overhead of a single persistence iteration. We do not show time overheads for the reconstruction phase because the number of reconstruction phases throughout the execution can be assumed to be much smaller than that of persistence iterations. Moreover, the reconstruction phase is heavily governed by the full-state reconstruction calculations, as explained in [52].

Fig. 8a presents the time overheads of ESR and *in-NVRAM* ESR in the homogeneous cluster architecture. We have implemented persistence to the local NVRAM using the ext4-dax local PMFS, PMDK, and MPI local windows over NVRAM. For reference, we also measured the time overhead of persisting the ESR data to a local SATA-SSD device. Dashed lines refer to a natural extrapolation of the results beyond a single NVRAM node[3], based on the simple observation that local persistence operations in different nodes proceed in parallel. Above 32 processes, ESR's time overhead increases significantly since persistence data must be sent to the RAM of processes in remote nodes.

Fig. 8b presents the evaluation of a single persistence iteration of *in-NVRAM* ESR in the PRD sub-cluster architecture implemented with MPI OSC over RDMA to NVRAM. To show the cost of implementing MPI OSC over NVRAM, we also present the time overhead of MPI OSC over RDMA when the windows are on *RAM* without persistent operations. For reference, we measure the time overhead of persisting the ESR data to a remote SATA-SSD device via SSH-FS. It can be seen that the overhead of ensuring persistence is relatively

small. It can also be seen that MPI-OSC over RDMA access to NVRAM is faster than accessing a remote SSD storage device, especially at high process counts.

These results show that *in-NVRAM* ESR/PRD is significantly superior to ESR in terms of the time overhead incurred by writing the recovery data, except when the number of processes is small, and they all fit inside a single node.

---

³We remind the reader that our single NVRAM node contains 20 cores.

REFERENCES

1 Schneider, D., "The exascale era is upon us: The frontier supercomputer may be the first to reach 1,000,000,000,000,000,000 operations per second," *IEEE Spectrum*, vol. 59, no. 1, pp. 34–35, 2022.

2 Schroeder, B. *et al.*, "Understanding failures in petascale computers," *Journal of Physics: Conference Series*, vol. 78, p. 012022, jul 2007. [Online]. Available: https://doi.org/10.1088/1742-6596/78/1/012022

3 Cappello, F. *et al.*, "Toward exascale resilience," *The International Journal of High Performance Computing Applications*, vol. 23, no. 4, pp. 374–388, 2009.

4 Canal, R. *et al.*, "Predictive reliability and fault management in exascale systems: State of the art and perspectives," *ACM Computing Surveys (CSUR)*, vol. 53, no. 5, pp. 1–32, 2020.

5 Heroux, M. A., "Recent trends and challenges for high performance sparse linear algebra." 2019.

6 Heroux, M. A. *et al.*, "An overview of the trilinos project," *ACM Transactions on Mathematical Software (TOMS)*, vol. 31, no. 3, pp. 397–423, 2005.

7 Heroux, M. A., "Software challenges for extreme scale computing: Going from petascale to exascale systems," *The international journal of high performance computing applications*, vol. 23, no. 4, pp. 437–439, 2009.

8 Chien, A., "Final report,"exploiting global view for resilience"," Univ. of Chicago, IL (United States), Tech. Rep., 2017.

9 Chien, A. *et al.*, "Exploring versioned distributed arrays for resilience in scientific applications: global view resilience," *The International Journal of High Performance Computing Applications*, vol. 31, no. 6, pp. 564–590, 2017.

10 Dun, N. *et al.*, "Flexible error recovery using versions in global view resilience," in *2015 IEEE International Conference on Cluster Computing*. IEEE, 2015, pp. 512–513.

11 Bridges, P. G. *et al.*, "Fault-tolerant linear solvers via selective reliability," *arXiv preprint arXiv:1206.1390*, 2012.

12 Moody, A. *et al.*, "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*. IEEE, 2010, pp. 1–11.

13 Ansel, J. *et al.*, "Dmtcp: Transparent checkpointing for cluster computations and the desktop," in *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, 2009, pp. 1–12.

14 Naksinehaboon, N. *et al.*, "Reliability-aware approach: An incremental checkpoint/restart model in hpc environments," in *2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*. IEEE, 2008, pp. 783–788.

15 Liu, Y. *et al.*, "An optimal checkpoint/restart model for a large scale high performance computing system," in *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 2008, pp. 1–9.

16 Heroux, M. A., "Toward resilient algorithms and applications," *arXiv preprint arXiv:1402.3809*, 2014.

17 Bosilca, G. *et al.*, "Recovery patterns for iterative methods in a parallel unstable environment," Citeseer, Tech. Rep., 2004.

18 Sao, P. *et al.*, "Self-stabilizing iterative solvers," in *Proceedings of the workshop on latest advances in scalable algorithms for large-scale systems*, 2013, pp. 1–8.

19 Argonne National Laboratory, "Aurora," https://www.alcf.anl.gov/aurora/, 2022.

20 Patil, O. *et al.*, "NVM-based energy and cost efficient hpc clusters," 2021.

21 Peng, I. B. *et al.*, "System evaluation of the intel optane byte-addressable NVM," in *Proceedings of the International Symposium on Memory Systems*, 2019, pp. 304–315.

22 Fridman, Y. *et al.*, "Assessing the use cases of persistent memory in high-performance scientific computing," in *2021 IEEE/ACM 11th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS)*. IEEE, 2021, pp. 11–20.

23 Patil, O. *et al.*, "Performance characterization of a dram-NVM hybrid memory architecture for HPC applications using Intel Optane dc persistent memory modules," in *Proceedings of the International Symposium on Memory Systems*, 2019, pp. 288–303.

24 Weiland, M. *et al.*, "An early evaluation of intel's optane dc persistent memory module and its impact on high-performance scientific applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–19.

25 Hennecke, M., "Daos: A scale-out high performance storage stack for storage class memory," *Supercomputing frontiers*, p. 40, 2020.

26 Weiland, M. *et al.*, "Exploiting the performance benefits of storage class memory for hpc and hpda workflows," *Supercomputing Frontiers and Innovations*, vol. 5, no. 1, pp. 79–94, 2018.

27 Ren, J. *et al.*, "Easycrash: Exploring non-volatility of non-volatile memory for high performance computing under failures," *arXiv preprint arXiv:1906.10081*, 2019.

28 Kunkel, J. *et al.*, "Establishing the io-500 benchmark," *White Paper*, 2016.

29 Attiya, H. *et al.*, "Nesting-safe recoverable linearizability: Modular constructions for non-volatile memory," in *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, 2018, pp. 7–16.

30 Aksenov, V. *et al.*, "Execution of nvram programs with persistent stack," in *International Conference on Parallel Computing Technologies*. Springer, 2021, pp. 117–131.

31 Golab, W. *et al.*, "Recoverable mutual exclusion in sub-logarithmic time," in *Proceedings of the ACM Symposium on Principles of Distributed Computing*, 2017, pp. 211–220.

32 Friedman, M. *et al.*, "Brief announcement: A persistent lock-free queue for non-volatile memory," in *31st International Symposium on Distributed Computing (DISC 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

33 Attiya, H. *et al.*, "Tracking in order to recover-detectable recovery of lock-free data structures," in *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, 2020, pp. 503–505.

34 Coburn, J. *et al.*, "Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1, pp. 105–118, 2011.

35 Venkataraman, S. *et al.*, "Consistent and durable data structures for non-volatile byte-addressable memory," in *9th USENIX Conference on File and Storage Technologies (FAST 11)*, 2011.

36 Qureshi, M. K. *et al.*, "Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling," in *2009 42nd Annual IEEE/ACM international symposium on microarchitecture (MICRO)*. IEEE, 2009, pp. 14–23.

37 Pommerell, C. *et al.*, "Memory aspects and performance of iterative solvers," *SIAM Journal on Scientific Computing*, vol. 15, no. 2, pp. 460–473, 1994.

38 Baker, C. G. *et al.*, "Anasazi software for the numerical solution of large-scale eigenvalue problems," *ACM Transactions on Mathematical Software (TOMS)*, vol. 36, no. 3, pp. 1–23, 2009.

39 Bavier, E. *et al.*, "Amesos2 and belos: Direct and iterative solvers for large sparse linear systems," *Scientific Programming*, vol. 20, no. 3, pp. 241–255, 2012.

40 Nazareth, J. L., "Conjugate gradient method," *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 1, no. 3, pp. 348–353, 2009.

41 Saad, Y. *et al.*, "Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems," *SIAM Journal on scientific and statistical computing*, vol. 7, no. 3, pp. 856–869, 1986.

42 Rutishauser, H., "The jacobi method for real symmetric matrices," *Numerische Mathematik*, vol. 9, no. 1, pp. 1–10, 1966.

43 Hadjidimos, A., "Successive overrelaxation (sor) and related methods," *Journal of Computational and Applied Mathematics*, vol. 123, no. 1-2, pp. 177–199, 2000.

44 Agullo, E. *et al.*, "Towards resilient parallel linear krylov solvers: recover-restart strategies," Ph.D. dissertation, INRIA, 2013.

45 Arnold, D. C., "Reliable, scalable tree-based overlay networks," Ph.D. dissertation, University of Wisconsin–Madison, 2008.

46 Chen, Z. *et al.*, "Algorithm-based checkpoint-free fault tolerance for parallel matrix computations on volatile resources," in *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*. IEEE, 2006, pp. 10–pp.

47 Chen, Z. *et al.*, "Algorithm-based fault tolerance for fail-stop failures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 12, pp. 1628–1641, 2008.

48 Chen, Z., *Scalable techniques for fault tolerant high performance computing*. The University of Tennessee, 2006.

49 Chen, Z., "Extending algorithm-based fault tolerance to tolerate fail-stop failures in high performance distributed environments," in *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 2008, pp. 1–8.

50 Davies, T. *et al.*, "Fault tolerant linear algebra: Recovering from fail-stop failures without checkpointing," in *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*. IEEE, 2010, pp. 1–4.

51 Hakkarinen, D. *et al.*, "Algorithmic cholesky factorization fault recovery," in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 2010, pp. 1–10.

52 Chen, Z., "Algorithm-based recovery for iterative methods without checkpointing," in *Proceedings of the 20th International Symposium on High Performance Parallel and Distributed Computing*. ACM, 2011, pp. 73–84.

53 Pachajoa, C. *et al.*, "Extending and evaluating fault-tolerant preconditioned conjugate gradient methods." in *2018 IEEE/ACM Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS)*. IEEE, 2018, pp. 49—58.

54 Pachajoa, C. *et al.*, "Algorithm-based checkpoint-recovery for the conjugate gradient method," in *49th international conference on parallel processing-icpp*, 2020, pp. 1–11.

55 Pachajoa, C. *et al.*, "How to make the preconditioned conjugate gradient method resilient against multiple node failures," in *Proceedings of the 48th international conference on parallel processing*, 2019, pp. 1–10.

56 Pachajoa, C. *et al.*, "A generic strategy for node-failure resilience for certain iterative linear algebra methods," in *2020 IEEE/ACM 10th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS)*. IEEE, 2020, pp. 41–50.

57 Dorożyński, P. *et al.*, "Checkpointing of parallel mpi applications using mpi one-sided api with support for byte-addressable non-volatile ram," *Procedia Computer Science*, vol. 80, pp. 30–40, 2016.

58 Dorożyński, P. *et al.*, "Extension of mpi one-sided communication api," https://github.com/pmem/mpi-pmem-ext/raw/master/mpi_one_sided_extension/doc/mpi-one-sided_extension.pdf, 2016. [Online]. Available: h t t p s : //github.com/pmem/mpi-pmem-ext/raw/master/mpi_one_sided_e xtension/doc/mpi-one-sided_extension.pdf

59 Scargall, S., "Introducing the persistent memory development kit," in *Programming Persistent Memory*. Springer, 2020, pp. 63–72.

60 Yasaman Ghadar, T. W., "An overview of aurora, argonne's upcoming exascale system," 2022, https://web.cse.ohio-state.edu/~panda.2/6422/cl ass_slides/Aurora_Overview.pdf.

61 Damkroger, T., "Argonne national labratory's aurora exascale system will enable new science, engineering," 2022, https://www.intel.com/content/ dam/www/public/us/en/documents/case-studies/argonne-aurora-case-stud y.pdf.

62 SC *et al.*, "Io500," 2022, https://io500.org/.

63 Li, S. *et al.*, "A performance & power comparison of modern high-speed dram architectures," in *Proceedings of the International Symposium on Memory Systems*, 2018, pp. 341–353.

64 Kashyap, A. *et al.*, "NVMe-oAF: Towards adaptive NVMe-oF for io-intensive workloads on HPC cloud," in *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, 2022, pp. 56–70.

65 Guz, Z. *et al.*, "Performance characterization of NVMe-over-fabrics storage disaggregation," *ACM Transactions on Storage (TOS)*, vol. 14, no. 4, pp. 1–18, 2018.

66 Gupta, A. *et al.*, "Evaluation of hpc applications on cloud," in *2011 Sixth Open Cirrus Summit*. IEEE, 2011, pp. 22–26.

67 Fridman, Y. *et al.*, "The case for non-volatile ram in cloud hpcaas," 2022. [Online]. Available: https://arxiv.org/abs/2208.02240

68 Levonyak, M. *et al.*, "Scalable resilience against node failures for communication-hiding preconditioned conjugate gradient and conjugate residual methods," in *Proceedings of the 2020 SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, 2020, pp. 81–92.

69 Sleijpen, G. L. *et al.*, "Bicgstab (l) and other hybrid bi-cg methods," *Numerical Algorithms*, vol. 7, no. 1, pp. 75–109, 1994.

70 Gropp, W. *et al.*, *Using Advanced MPI*. Massachusetts Institute of Technology, 2014.

71 Du, J. *et al.*, "Fast and consistent remote direct access to non-volatile memory," in *50th International Conference on Parallel Processing*, 2021, pp. 1–11.

72 Liu, X. *et al.*, "Write-optimized and consistent rdma-based NVM systems," *arXiv preprint arXiv:1906.08173*, 2019.

73 Forum, M. P. I., "Mpi: a standard message passing interface," https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf, 2015.

74 "Quick start guide part 1: Persistent memory provisioning introduction," https://software.intel.com/content/www/us/en/develop/articles/qsg-intro-to-provisioning-pmem.html, Intel.

75 Kadekodi, R. *et al.*, "Splitfs: Reducing software overhead in file systems for persistent memory," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 494–508.

76 Marjanović, V. *et al.*, "Performance modeling of the hpcg benchmark," in *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*. Springer, 2014, pp. 172–192.

77 Stevens, R. *et al.*, "Aurora: Argonne's next-generation exascale supercomputer," Argonne National Lab.(ANL), Argonne, IL (United States), Tech. Rep., 2019.

78 Park, R. I., "NegevHPC Project," https://www.negevhpc.com, 2019, [Online].

79 Eller, P. R. *et al.*, "Using performance models to understand scalable krylov solver performance at scale for structured grid problems," in *Proceedings of the ACM International Conference on Supercomputing*, 2019, pp. 138–149.

Preconditioned Conjugate Gradient (PCG) is a main example of linear iterative solver, as it is commonly used to solve linear systems for a wide range of applications including computational fluid dynamics, wind energy, and particle physics [79]. PCG solves the linear equation $Ax = b$ for a symmetric positive definite matrix $A_{n \times n}$ (see Algorithm 3). The dependency graph of PCG is shown in Fig. 9. An ESR algorithm for PCG has been derived by the generic meta-algorithm described in [56], from PCG's dependency graph. To recover from node failures, PCG saves redundancies for the search direction variable, i.e., the variable p (hence $V_{SpMV} = \{p\}$). To reconstruct the complete state of a failed process f, two successive values of p_{I_f} are required. Therefore, the redundancy of p is saved for two successive iterations (hence $k = 2$). Algorithm 4 describes a persistence iteration of *in-RAM* ESR for PCG (Algorithm 3), where SpMV operations with p are augmented to ASpMV operations to create p's redundancies. Algorithm 5 shows the reconstruction stage of *in-RAM* ESR, where redundancies of p are collected from the RAM of the surviving nodes to reconstruct the failed processes.

With *in-NVRAM* ESR the redundancies of p are saved for each process in the NVRAM. Algorithm 6 shows a persistence iteration of PCG, in which values of p are saved in NVRAM (locally or remotely). Algorithm 7 shows the reconstruction stage of PCG under *in-NVRAM* ESR, collecting p's redundancies by accessing the NVRAM.

Algorithm 3 PCG solver for $Ax = b$.

1: $r^{(0)} \leftarrow b - Ax^{(0)}$, $z^{(0)} \leftarrow Pr^{(0)}$, $p^{(0)} \leftarrow z^{(0)}$
2: **for** $j = 0, 1, ...$ until convergence **do**
3:      $\alpha^{(j)} \leftarrow r^{(j)T} z^{(j)} / r^{(j)T} A p^{(j)}$
4:      $x^{(j+1)} \leftarrow x^{(j)} + \alpha^{(j)} p^{(j)}$
5:      $r^{(j+1)} \leftarrow r^{(j)} - \alpha^{(j)} A p^{(j)}$
6:      $z^{(j+1)} \leftarrow Pr^{(j+1)}$
7:      $\beta^{(j)} \leftarrow r^{(j+1)T} z^{(j+1)} / r^{(j)T} z^{(j)}$
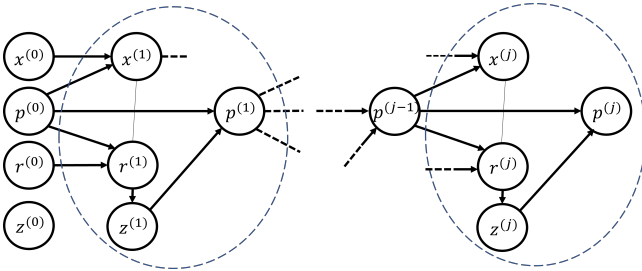8: $p^{(j+1)} \leftarrow z^{(j+1)} + \beta^{(j)} p^{(j)}$
9: **end for**

Fig. 9: PCG Dependency Graph.

Algorithm 4 *in-RAM* ESR for each iteration j of two successive redundancy iterations of PCG. Line numbering refers to the lines of Algorithm 3.

3: $\alpha^{(j)} \leftarrow r^{(j)T} z^{(j)} / r^{(j)T} ASpMV(A, p^{(j)})$
    ⋮
8: $p^{(j+1)} \leftarrow z^{(j+1)} + \beta^{(j)} p^{(j)}$

Algorithm 5 *In-RAM* ESR Reconstruction phase of PCG.

1: Retrieve the static data $A_{I_f, I}$, $P_{I_f, I}$ and b_{I_f}
2: Gather $r^{(j)}_{I \setminus I_f}$ and $x^{(j)}_{I \setminus I_f}$
3: Retrieve the redundant copies of $\beta^{(j-1)}$, $p^{(j-1)}_{I_f}$ and $p^{(j)}_{I_f}$ from the RAM of other processes
4: Compute $z^{(j)}_{I_f} \leftarrow p^{(j)}_{I_f} - \beta^{(j-1)} p^{(j-1)}_{I_f}$
5: Compute $v \leftarrow z^{(j)}_{I_f} - P_{I_f, I \setminus I_f} r^{(j)}_{I \setminus I_f}$
6: Solve $P_{I_f, I_f} r^{(j)}_{I_f} = v$ for $r^{(j)}_{I_f}$
7: Compute $w \leftarrow b_{I_f} - r^{(j)}_{I_f} - A_{I_f, I \setminus I_f} x^{(j)}_{I \setminus I_f}$
8: Solve $A_{I_f, I_f} r^{(j)}_{I_f} = w$ for $x^{(j)}_{I_f}$

Algorithm 6 *In-NVRAM* ESR for a persistence iteration j of PCG. Line numbering refers to the lines of Algorithm 3.

3: $\alpha^{(j)} \leftarrow r^{(j)T} z^{(j)} / r^{(j)T} A p^{(j)}$
    ⋮
8: $p^{(j+1)} \leftarrow z^{(j+1)} + \beta^{(j)} p^{(j)}$
    if HOMOGENEOUS CLUSTER **then**
       persist $p^{(j+1)}$ to **local NVM**
    if PRD SUB-CLUSTER **then**
       persist $p^{(j+1)}$ to **remote NVM**

Algorithm 7 *In-NVRAM* ESR Reconstruction phase of PCG. Line numbering refers to the lines of Algorithm 5.

if HOMOGENEOUS CLUSTER **then**
    Wait for failed nodes to recover and have access to **local NVM**
if PRD SUB-CLUSTER **then**
    Run reconstruction from any spare nodes that have access to the
    remote NVRAM Sub-Cluster

    Retrieve the static data $A_{I_f, I}$, $P_{I_f, I}$ and b_{I_f}
    Gather $r^{(j)}_{I \setminus I_f}$ and $x^{(j)}_{I \setminus I_f}$
    if HOMOGENEOUS CLUSTER **then**
3:     Read $\beta^{(j-1)}$, $p^{(j-1)}_{I_f}$ and $p^{(j)}_{I_f}$ from **local NVM**
    if PRD SUB-CLUSTER **then**
3:     Read $\beta^{(j-1)}$, $p^{(j-1)}_{I_f}$ and $p^{(j)}_{I_f}$ from **remote NVM**
    ⋮
8: Solve $A_{I_f, I_f} r^{(j)}_{I_f} = w$ for $x^{(j)}_{I_f}$