Research Paper

International Journal of HIGH PERFORMANCE COMPUTING APPLICATIONS

The International Journal of High Performance Computing Applications
2017, Vol. 31(6) 564–590

DOI: 10.1177/1094342016664796
journals.sagepub.com/home/hpc
SAGE

Exploring versioned distributed arrays for resilience in scientific applications: global view resilience

A Chien[1,2], P Balaji[2], N Dun[1,2], A Fang[1], H Fujita[1,2], K Iskra[2], Z Rubenstein[1], Z Zheng[3], J Hammond[4], I Laguna[5], D Richards[5], A Dubey[6], B van Straalen[6], M Hoemmen[7], M Heroux[7], K Teranishi[7] and A Siegel[2]

Abstract

Exascale studies project reliability challenges for future HPC systems. We present the Global View Resilience (GVR) system, a library for portable resilience. GVR begins with a subset of the Global Arrays interface, and adds new capabilities to create versions, name versions, and compute on version data. Applications can focus versioning where and when it is most productive, and customize for each application structure independently. This control is portable, and its embedding in application source makes it natural to express and easy to maintain. The ability to name multiple versions and "partially materialize" them efficiently makes ambitious forward-recovery based on "data slices" across versions or data structures both easy to express and efficient. Using several large applications (OpenMC, preconditioned conjugate gradient (PCG) solver, ddcMD, and Chombo), we evaluate the programming effort to add resilience. The required changes are small ($<$ 2% lines of code (LOC)), localized and machine-independent, and perhaps most important, require no software architecture changes. We also measure the overhead of adding GVR versioning and show that overheads $<$ 2% are generally achieved. This overhead suggests that GVR can be implemented in large-scale codes and support portable error recovery with modest investment and runtime impact. Our results are drawn from both IBM BG/Q and Cray XC30 experiments, demonstrating portability. We also present two case studies of flexible error recovery, illustrating how GVR can be used for multi-version rollback recovery, and several different forward-recovery schemes. GVR's multi-version enables applications to survive latent errors (silent data corruption) with significant detection latency, and forward recovery can make that recovery extremely efficient. Our results suggest that GVR is scalable, portable, and efficient. GVR interfaces are flexible, supporting a variety of recovery schemes, and altogether GVR embodies a gentle-slope path to tolerate growing error rates in future extreme-scale systems.

Keywords

Resilience, fault-tolerance, exascale, scalable computing, application-based fault tolerance

1 Introduction

With the widely documented end of Dennard scaling (Borkar and Chien, 2011; Dennard et al., 1974; Esmaeilzadeh et al., 2011), power is an increasingly critical concern for systems from mobile to supercomputer scale. As a result, both aggressive voltage scaling and the physical challenges of deep submicrometer technologies (14 nm, 7 nm, and beyond) give rise to increased error rates, wearout, and manufacturing variability. Reliability is already a serious concern in today's supercomputers at 1 million cores, where system-wide mean time between failures (MTBF) can be as short as a few hours (Di Martino et al., 2014; Schroeder and Gibson, 2006; Zheng et al., 2011). With the growing the growing scale of high performance computers scale (to 1–10 million cores in pre-exascale systems (Antypas et al., 2014)

[1]University of Chicago, USA
[2]Argonne National Laboratory, USA
[3]HP Vertica, USA
[4]Intel Corp, USA
[5]Lawrence Livermore National Laboratory, USA
[6]Lawrence Berkeley National Laboratory, USA
[7]Sandia National Laboratories, USA

Corresponding author:
A Chien, University of Chicago and Argonne National Laboratory, 1100 E 58th Street, Ryerson 257C, Chicago, IL 60637, USA.
Email: achien@cs.uchicago.edu

(see also https://www.olcf.ornl.gov/summit/), and perhaps 100 million cores in exascale (Peter Kogge, et. al., 2008)) exacerbates both power and reliability concerns (Borkar and Chien, 2011; Cappello et al., 2009; Elnozahy 2009; Peter Kogge, et. al., 2008).

Recent studies of modern supercomputers have shown failure as the norm rather than the exception, with system-wide mean time between failures of a few hours (Schroeder and Gibson, 2006; Zheng et al., 2011). Future exascale systems are projected to have mean time to interrupt (MTTI) (Bergman and et al., 2008; Borkar and Chien, 2011; Ferreira et al., 2011; Moody et al., 2010) as low as 10–30 minutes. If large-scale applications are to succeed under these circumstances, they will need strong resilience support. Without it, such large-scale applications will struggle to make efficient progress.

A wide array of checkpoint–restart research has explored techniques to efficiently apply checkpointing (Daly, 2006; Young, 1974) and improve its performance (Antypas et al., 2014; Bautista-Gomez et al., 2011; Cappello et al., 2011; Fang and Chien, 2015; Moody et al., 2010). Notably, recent advances that exploit high bandwidth non-volatile memories both reduce checkpoint cost dramatically, and because their efficiency reduces the optimal checkpoint interval, can do "micro-checkpointing" (fast checkpointing with interval of seconds), reducing the work lost per detected error or process failure. Such checkpointing enables systems to tolerate high rates of "fail-stop" (immediately detected) errors (Schlichting and Schneider, 1983).

The Global View Resilience (GVR) project seeks to address a larger class of errors, including not only fail-stop (immediately detected failures such as an error-correcting code (ECC) or checksum detected partial data loss) or node crash addressed by classical checkpoint–restart (Hargrove and Duell, 2006), but also growing concerns about *latent errors*, or often called silent data corruption (SDC). Latent errors are errors which are not detected immediately, but may eventually manifest themselves as incorrect results, severe performance degradation, or even application crash (Fiala and et al., 2012; Shantharam et al., 2011). Interestingly, shortening the checkpoint interval such as enabled by recent advanced techniques may increase the number of latent errors, as the likelihood that the error propagation to detection (fail-stop) increases as the checkpoint interval is shortened.

Further, numerous reports document SDC errors as more frequent than previously believed. For example, the BlueGene/L system (106,496 nodes) experiences an L1 cache error (parity error) every 4–6 hours (Moody et al., 2010). The Cray XT5 at Oak Ridge National Laboratory experiences an uncorrectable double bit error on a daily basis (Fiala and et al., 2012). Latent errors are generally invisible (silent) until the corrupted data is used (Lu and Reed, 2004), so detection latency is generally much longer, and can be heavily algorithm-dependent or application-dependent. As shown in previous studies (Fiala and et al., 2012; Shantharam et al., 2011), the latent errors can create severe application problems such as incorrect results or extreme performance degradation.

Our approach, GVR, uses versioned, distributed arrays to enable computational scientists to build portable, resilient applications. Beyond process/node crashes, GVR also enables resilience for more difficult *latent* or *silent* errors (Lu et al., 2013a).

Key features of GVR include:

- multi-version distributed arrays that enable complex and latent error recovery;
- multi-stream versioning that gives the programmer control of when versions are created for an array; and
- unified error signaling and handling, customized per GVR distributed array, that enable algorithm-based fault-tolerance (ABFT) (Huang and Abraham, 1984) error-checking and recovery.

We explore use of the GVR library for resilience in several large applications (OpenMC, preconditioned conjugate gradient (PCG) solver, ddcMD, and Chombo) that are collectively nearly 1 million lines of code. Through this experience, we gain significant insight into the programmer effort required (code changes) to adopt version-based resilience and its performance impact. Our experience and resulting resilient applications show that introducing version-based resilience can often be achieved with small, localized code changes. These changes are portable (machine-independent) and create a gentle-slope programming path to tolerate growing error rates in future systems.

Performance is a central concern of high-performance computing, so with the same four applications, we evaluate the overhead of version-based resilience based on a prototype GVR system GVR Team (GVR Team, 2014b), reporting overhead results. These studies involve a range of application types, run sizes, and machine sizes, and that acceptable overheads can be achieved.

Finally, we present two studies. To explore the benefit of multi-stream versioning, we conduct a study using PCG code. We first apply same versioning rate to different data structures and then vary the versioning rate. We compare the versioning overhead and recovery performance of these configurations. To explore the flexibility of error recovery supported by GVR, we undertake a case study using the OpenMC code in the presence of latent errors. We implement and compare empirically three error recovery techniques, varying error latency, and show the resulting performance.

Specific contributions include the following.

- A description of the GVR resilience model, including the API for multi-stream, versioned distributed arrays, and flexible cross-layer error signalling and recovery.
- Study of adding resilience to applications and proxy codes (OpenMC, PCG/Trilinos, ddcMD, and Chombo), and description of the variety of different approaches used and how they reflect GVR's flexibility.

– Error Checking: Error checks can rely on hardware, exploit system software checks, or exploit application-semantics, to detect both immediate and silent data corruption.
– Redundancy: Applications indicate versioning rates for each distributed array, and can exploit the existing versions as they see fit for recovery.
– Error recovery: can be achieved by rollback, multi-version restoration by approximation, and other methods forward error recovery (Randell et al., 1978).

- Empirical study of required source code changes to add resilience; the results show that only modest, localized code changes (<2 % lines of code (LOC)) are required and no software architecture changes are required to enable GVR-based application resilience.
- Performance evaluation that of GVR versioning and scalability, documenting achievable overheads $<2\%$ even at versioning frequencies higher than needed for today's error rates, and with up to 16,000 processes (1000–16,000 varying by application).
- A case study of multi-stream versioning on PCG solver. The study shows that multi-stream versioning provides control to reduce overhead and flexible choices to achieve efficient recovery.
- A case study of flexible error recovery using GVR on the OpenMC application. The study illustrates three different error recovery schemes that can tolerate latent errors: rollback, basic forward, and forward with additional batches. Performance measurements show that forward error recovery can give significant benefits when error latency increases.

In short, our results show GVR version-based resilience is a portable and gentle-slope. Further, it enables flexible application and system error checking and reporting as well as many different methods for error recovery. As such, GVR is a promising approach to scale forward to the higher error rates expected in extreme-scale hardware.

1.1 Paper organization

The rest of the paper is organized as follows. In Section 2, we describe the GVR model, introducing key concepts such as distributed arrays, versioning, multi-stream, and application-controlled flexible error recovery. In Section 3, we describe addition of version-based resilience to mini-apps and full applications, closing with a summary of the needed source code changes to achieve resilience. In Section 3.3, we present performance experiments which explore the runtime costs of versioning for a range of application sizes, machine configurations, and error rates. To give deeper insights into the use of GVR, we present two deep dives that illustrate multi-stream versioning in preconditioned conjugate gradient (CG), and design and evaluation of several different GVR-based resilience techniques in OpenMC in Section 4. Related work and our results in context are presented in Section 5. We close in Section 6 with a brief review of our results and highlight several interesting future research directions.

2 GVR model and APIs

The GVR model enables portable application-controlled resilience. Applications control redundancy (per data structure), error checking, and recovery (exploit application semantics) in a portable fashion with versioned distributed arrays.

GVR's interface has two parts: one for basic data access, update, and version creation, and the other for error signalling and handling. These interfaces further introduce the concept of multi-stream, multi-version, and enable flexible recovery.

2.1 The GVR model and interfaces

2.1.1 Global-view array basic interface. GVR distributed arrays each have a global name, but are distributed across multiple nodes (Bariuso and Knies, 1994; Nieplocha et al., 2006; Numrich and Reid, 1998). The global name supports flexible programming of irregular applications and, in the context of resilience, eases recovery programming when the number of physical resources have changed.

The basic GVR APIs are illustrated in Figure 1, and we briefly introduce those needed to understand the program examples. Notable are GVR's operations to create, navigate, and otherwise manipulate versions. GVR's full API is documented in the GVR API reference (GVR Team, 2014a).

- *GDS_alloc,GDS_create* create a global array from scratch or by federating existing local memory regions (collective).
- *GDS_put,GDS_get block* data access to the distributed array.
- *GDS_acc* block accumulate operations against the global array.

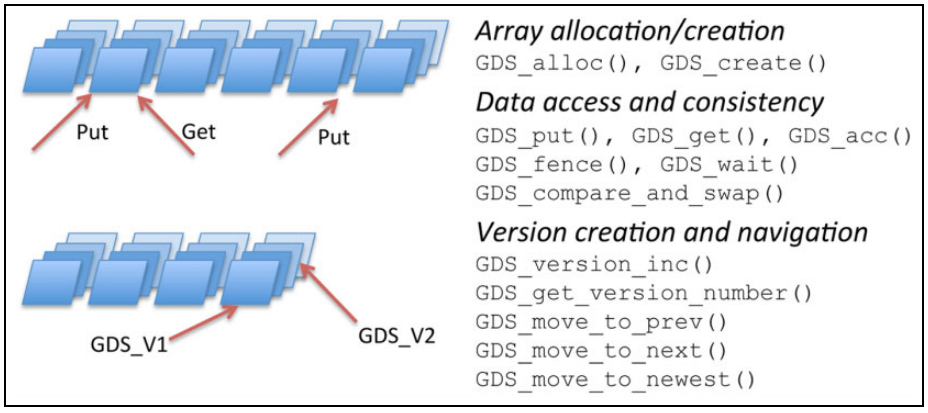

Figure 1. Distributed arrays and versioning in GVR.

- *GDS_compare_and_swap* compares single value with an element of the global array and swaps if they match.
- *GDS_fence, GDS_wait* are collective and local synchronization operations.
- *GDS_version_inc* advances the version for the given global array. Collectively marks a computation boundary where the array data is consistent. GVR runtime decides if a version is actually created based on storage, performance, failure-rate considerations.
- *GDS_descriptor_clone* creates a copy so we navigate and manipulate several versions at the same time.
- *GDS_get_version_number* returns the version number from a global array descriptor.
- *GDS_move_to_prev,GDS_move_to_next,GDS_move_to_newest* move the global array descriptor to the next, previous, newest version available.

2.1.2 GVR error handling interface. GVR includes a unified signalling and error-handling interface that supports flexible application and cross-layer handling (e.g. Bridges et al., 2012; Chen, 2013; Huang and Abraham, 1984). When an error occurs, the hardware/system/application invokes the GVR library, passing an error descriptor that describes the error.[1] GVR allows errors to be signalled with two different scopes (local and global), and corresponding different error recovery scopes (local and global). Local error handling involves a single process, and global error handling involves collective action by all the processes sharing the GVR array. Of course, local errors can be escalated.[2]

Beyond a unified error-signalling and -handling structure (see examples in Figures 6 and 18), GVR further incorporates the *Open Resilience* (OR) interface (GVR Team, 2014b). The OR interface is designed to support a resilience ecosystem, where investment in detailed error signaling and sophisticated error recovery gives incremental benefit to vendors, application scientists, and middleware developers. The full description of the OR system is beyond the scope of the paper, but the basic ideas include the following.

- An extensible set of error descriptors; each with set of error attributes (key-value pairs) such as "corrupted memory address = 0xffe0" or "failed process ranks = {24, 25}". These attributes are designed to be composable with other attributes, in order to allow flexible combination of them.
- A predicate language that is used to supply a predicate for the triggering of each error handler. With this description, the GVR system selects the closest match, allowing modular refinement of error handling based on incremental addition of error reporting or handling.

Figure 3 shows how the OR interface can be used. OR allows generalize existing error handler to another kinds of errors, as shown in the center of the figure, leading to maximizes the benefit of error handlers. It also allows applications to add a specialized error handler, without affecting other handlers. For example, if an application is ported to another platform, it can add a new error handler to implement a recovery logic which fully utilizes the knowledge of a new platform. Overall, OR allows applications to incremental investment towards more flexible error handling.

A simple example of the OR interface is presented in Figure 7. Detailed description and evaluation of the OR interface will be the subject of future work.

2.2 Using GVR to introduce versioning and flexible recovery

Applications using GVR can take a snapshot of an individual distributed array at a time of their choosing, what we call multi-stream versioning, to match redundancy to application structure as needed. Applications

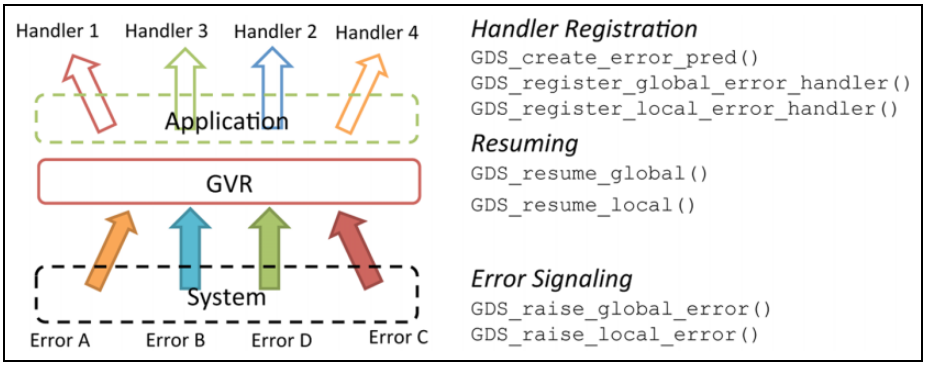

Figure 2. Unified error signaling and handling in GVR.

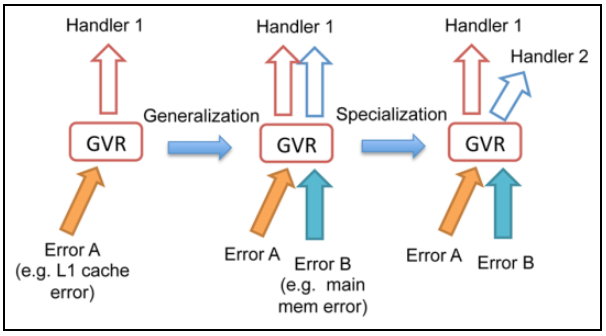

Figure 3. Conceptual use cases of the OR interface. Handler 1 handles Error A (L1 cache error). It can be generalized to handle Error B (main memory error). Handler 2 is a specialized error handler adapted for another platform to handle both Error A and Error B.

can select the timing of versions to minimize synchronization cost and maximize recovery value (keep as small a state as possible for maximum coverage). Versioning is a convenient idiom, as the versioning rate can be increased if error rates and types increase, providing a "gentle slope" for resilience. Another application tactic could be to add error checking and recovery techniques (also supported by GVR) as errors increase.

Figure 4 compares the traditional checkpoint–restart approach to GVR's flexible recovery. As shown on the left, a checkpoint–restart system maintains a single checkpoint and, upon error, rolls back to the checkpoint and restarts the program. In comparison, the GVR system provides a flexible set of options based on versioned application state.

Each call to *version_inc* creates a version and increments the current version number.[3] Calls to *version_inc* control the rate and timing of versioning for a given array. The key idea here is *multi-stream versioning*, that is, versioning can be done individually, at different timings for different arrays, based on each data structure's characteristic and resiliency requirement. Multi-stream versioning allows applications to optimize cost of versioning and provided resilience. For example, a read-only object needs only be preserved once, an object that is easy to calculate may not need to be preserved at all, and objects that take up less memory can be efficiently preserved at a greater frequency than objects that take up more memory.

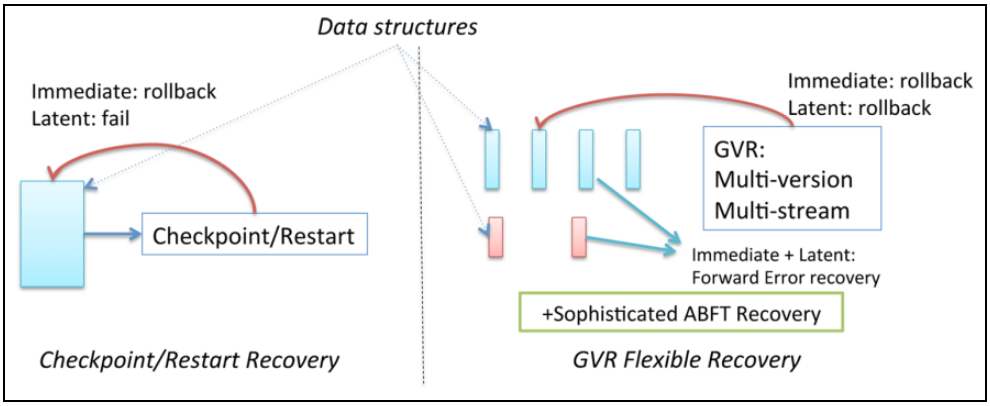

Figure 4. Comparison between the classical checkpoint–restart (left) and GVR (right). Checkpoint–restart cannot handle latent errors while GVR expands the possibilities, enabling Flexible Error Recovery.

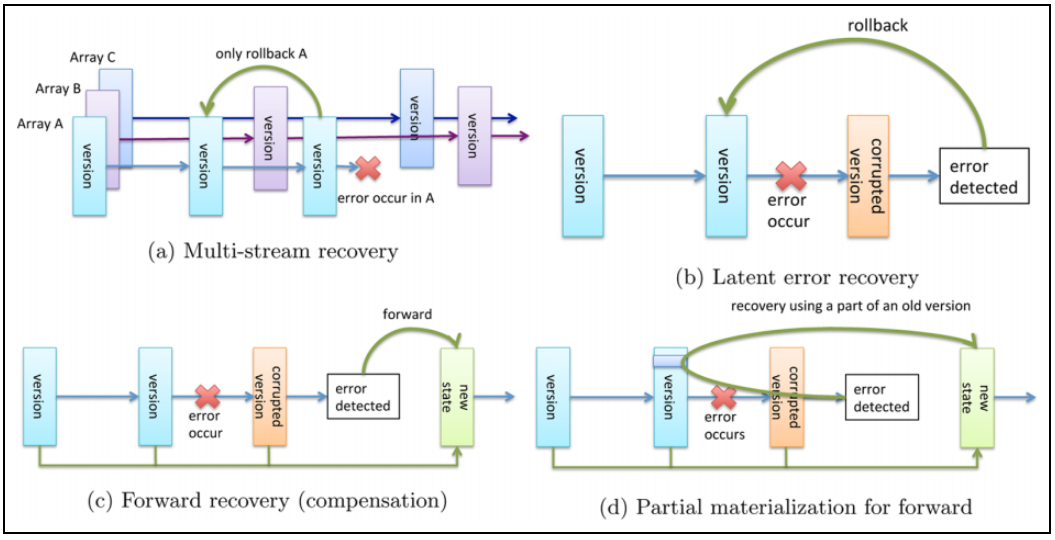

Figure 5. Illustrating the flexibility of application recovery using GVR.

GDS_move_to_prev and *GDS_move_to_next* update an array handle to point to a previous/next version. For recovery, GVR provides convenient primitives to name and navigate multiple versions. This differs from checkpoint–restart systems, where checkpoints have no application names, nor can you manipulate parts of several checkpoints simultaneously to aid complex recovery. When an error is detected (by the system or perhaps an application consistency check routine), an error handler is invoked through the GVR unified error handling interface, with an error descriptor as a parameter. A descriptor is generated by a component which detects an error.

GVR enables a wide range of recovery options based on multiple versions, multiple streams (data structures), and the use of application semantics for forward error recovery. This wide range is illustrated in Figure 5. First, a GVR program can conveniently version different data structures at different rates, in several different streams (we call this *multi-stream*). This feature enables multi-stream recovery scenario in Figure 5a. If the error occurs in one data structure, the program can recover it independently. Second, when an error occurs, recovery can happen from the most recent version or from older versions, enabling recovery from latent/silent errors. Figure 5b shows that error detection latency results in corrupted versions. The program can still recover from previous good versions. Traditional checkpoint–restart approach cannot handle latent errors, because it provides an access only to the latest checkpoint, which might have already been corrupted by a latent error. If the latest checkpoint is corrupted, the only way to recovery the program is to restart it from the beginning, which is prohibitively expensive under high rate of latent errors (Lu et al., 2013a). Third, instead of a rollback recovery, a GVR program can choose to do a forward error recovery (Figure 5c) by computing an approximation from the current application data and any collection of versions. For example, if it can be isolated, an error handler might recover by subtracting/removing the contribution of a corrupted value. This is particularly useful if there are several good phases of computation after the corrupted version which could be used (the phases are commutative) as in OpenMC. Finally, we have intentionally designed the GVR library implementation to ensure that version data retrieval cost is proportional to the size of data accessed. This allows applications to flexibly use many versions for recovery as shown in Figure 5d (we call this *partial materialization* of a version). An example of this is shown in Figure 15, in our OpenMC deep dive.

As we will revisit again in our performance evaluation, key performance aspects of the GVR model include:

1. versioning runtime overhead is independent from the number of existing versions;
2. data retrieval cost from a version is independent from the number of existing versions (given that the data size accessed is fixed); and
3. data retrieval cost from a version is proportional to the size of data accessed (*partial materialization*).

All four of these capabilities are unique to the GVR interface: checkpoint–restart provides no support for multi-stream, naming of versions, flexible version recovery, or partial materialization in a larger, more complex recovery. Of course all of these things can be programmed with traditional checkpoint–restart approaches, but at significantly greater manual effort and complexity.

3 Application studies

In this section we demonstrate real use cases of GVR for several existing scientific applications. We show that introducing resilience by GVR requires very small programming effort and almost negligible runtime overhead.

3.1 Applications

We have applied GVR to a set of complex, scientifically useful, large-scale community application codes, several of which are broadly used by diverse computational communities.

- OpenMC (Romano and Forget, 2013) is a production Monte Carlo neutron transport simulation code. It was originally developed by members of the Computational Reactor Physics Group at the Massachusetts Institute of Technology and now is used by the DOE CESAR co-design center to explore scalable nuclear reactor modeling.
- Comparing to other Monte Carlo production codes, such as MCNP (Goorley et al., 2012) and MC21 (Sutton et al., 2007), OpenMC is designed to leverage current and future high-performance computing platforms and explore large parallelism. OpenMC is capable of simulating 3D models based on constructive solid geometry with second-order surfaces, which makes possible high-fidelity simulations of nuclear reactors. OpenMC also demonstrate the ability to model large models with considerable geometric and material complexity. By using a mapping technique for fast scoring bin indexing, the implementation of tallies in OpenMC was shown to be efficient with respect to tallying large numbers of quantities. Its parallel fission bank algorithm also allows for parallel scaling up to tens of thousands of processors.
- PCG (Golub and Van Loan, 1996) is an efficient and widely used method to iteratively solve the linear system $Ax = b$, where A is a matrix of size $m \times m$, and x and b are vectors of length m where the value of b is known a priori while the value of x is unknown. A large number of scientific applications require the solution to this linear system. A popular approach to this problem are iterative solvers. Iterative solvers approximate x with increasing accuracy at every iteration. Examples of iterative solvers include stationary methods such as successive over-relaxation (SOR), and Krylov subspace methods such as CG or generalized minimal residual method (GMRES). In this study, we focus on PCG. PCG speeds up the convergence by applying a preconditioner M to A and b and then solving the equation $MAx = Mb$ (Saad, 1993). The justification behind this procedure is that it may be less expensive to solve $MAx = Mb$ than to solve $Ax = b$. There are a number of different choices for M. We opt for incomplete Cholesky factorization (Kershaw, 1978) using an arbitrary drop threshold of 0.001 (Saad, 1993). In relation to PCG, we will later refer to the vector p, which records the direction that x is moved in the current iteration; the vector r, which is principally identical to the residual $b - Ax$, but is updated in-place for optimization purposes rather than being calculated as $Ax - b$ in each iteration; and ρ, which records the norm of r from the previous iteration.
- ddcMD (Glosli. et al., 2007; Streitz et al., 2006) is a parallel classical molecular dynamics application developed by Lawrence Livermore National Laboratory. It is highly scalable and efficient. It has twice won the Gordon Bell prize for high-performance computing (Glosli. et al., 2007; Streitz et al., 2005). On BlueGene/L, ddcMD was used to demonstrate application assisted fault tolerance when BlueGene/L was discovered to have fatal L1 cache errors. The L1 cache on BG/L can detect, but not correct parity errors due to bit flips. On a system with over 200,000 cores L1 parity errors occur roughly every 5 hours. Therefore, ddcMD exploits a checkpoint/rollback scheme and utilizes application-level error recovery strategy. It periodically takes a fast checkpointing of the full computation state in memory. When an unrecoverable parity error is detected, the error handler sets an application-level global flag. The application continues execution until it reaches a designated rally point, when all tasks check the error flag, discard current results, and restore to the previous backup state. We replicate and extend the fault tolerance of ddcMD by using GVR (Fang and Chien, 2014).
- Chombo (Colella et al., 2009) is a library that implements block-structured adaptive mesh refinement (SAMR) technique (Berger and Colella, 1989; Berger and Oliger, 1984) to efficiently achieve higher resolution in regions of interest. Chombo defines patches of uniform resolution and embeds them within other patches of lower resolution. This is useful when available resources do not permit a simulation with uniformly high resolution that is demanded by some regions in the domain. All patches with the same resolution are grouped into a level, but distributed arbitrarily in the physical space. Thus a level can be viewed either as a logical entity (same resolution) or a physical entity (union of all patches at the same resolution). Depending upon the scientific domain and the specific application the AMR hierarchy can get very deep. The solution advances with the same timestep everywhere on a given level, though it might differ from all other levels. Similarly load distribution and regridding are also managed on individual levels.

In the following, we explore how each of these applications were enhanced for resilience with GVR.

3.2 How applications use GVR

We studied our four applications, using GVR to explore the programming effort, models of adding resilience, and performance impact of versioning. We quickly learned that there are two ways to integrate GVR version-based resilience.

1. Deep: Federate primary application data structures as a GVR distributed array.
2. Shallow: add a new distributed array and copy application data into it for versioning.

Amongst our four applications, OpenMC used deep integration and ddcMD, PCG, and Chombo use the shallow model. In general, we expect that many GVR adopters would begin with shallow integration, and then if memory pressure is a significant concern, proceed to deep integration (eliminates one copy). Below, we discuss how GVR was added to each of the four applications in turn.

3.2.1 OpenMC. Among several in-memory data structures in OpenMC, we introduced a distributed array to represent the tally data. The tally data is the essential computation structure where the results of the Monte Carlo samples are collected. Thus, it is the natural place to introduce versioning. In the original version of OpenMC, tally data was represented as a set of local buffers, one in each process. This limited the simulation scaling, as each process maintained a tally array sized for the entire application in its memory. By introducing the distributed array using GVR, OpenMC can take advantage of globally shared data and gain scalability (Dun et al., 2014). Tally data is region-based and accumulated (i.e. fetch-and-add) data, where the region, or tally region, is the volume over which the tallies should be integrated. The size of total tally data is directly proportional to the number of physical quantities to be tallied and the number of tally regions. In a realistic reactor simulation, that tally size could reach terabytes. Tally is a write (acc)-only data structure; read (get) never happens during simulation. During the simulation, OpenMC simulates a batch, a set of particles. When it completes simulation of one batch, it creates a version, then proceeds to the next batch.

3.2.2 PCG. We make PCG resilient by periodically taking snapshots of critical variables and restoring these variables if PCG fails checks for algorithm-specific invariants (Chen, 2013). Our implementation of PCG is built on top of Trilinos (Heroux et al., 2005). The Trilinos project is a C++ library that provides scalable primitives for linear algebra operations, linear and nonlinear solvers, and other useful scientific computing algorithms. In this study, we utilized Trilinos' linear algebra primitives in order to implement a PCG solver. We take advantage of the abstraction that Trilinos offers by building GVR-provided resilience into linear algebra primitives rather than requiring the application developer to interact with GVR directly. We decorated Trilinos vector objects with methods to snapshot (take a version) and restore state on demand with GVR (Rubenstein et al., 2013). These methods were then used in conjunction with application-directed error detection in order to find errors and restore to a previous application state as appropriate. GVR enabled convenient expression of application-level error checks connected to application-driven recovery.

Our experiments with the PCG solver show that the choice of detection methods make a good deal of difference when correcting errors in PCG (Rubenstein et al., 2013). Inexpensive methods based on monitoring the norm residual and more expensive, algorithm-aware methods that performed extra linear algebra operations to verify PCG-specific invariants were implemented and explored empirically.

3.2.3 ddcMD. The ddcMD code includes in-memory snapshot, as well as a trap handler to capture the L1 cache parity error. When applying GVR we exploited the snapshot and restore handler with minor code changes. As shown in Figure 6, the GVR-enhanced version has similar structure, but cleanly separates the error-handling code. The GVR version protects the essential data in ddcMD such as positions, velocities in global arrays by creating versions (see the left-hand side of Figure 6). Furthermore, the original error handler is transformed into a GVR-style error handler utilizing the unified error signaling interface of GVR. With GVR extended error signaling and handling, we generalize the original error handler that was designed only for L1 cache errors, to many other kinds of errors such as main memory errors and application-detected data corruptions.

Errors identified by the error detectors trigger the GVR library to identify the right error handler (with application help). As shown in the code, types of error detectors include hardware detection to middleware to application-semantic checks. An example application-level check uses energy conservation in the simulation ("total energy change threshold"), to identify errors. Specifically, in ddcMD, typically the total energy shows minor changes (i.e. 1×10^{-6}) between time steps. Larger changes (i.e. 1×10^{-2}) in total energy generally indicate an error. Fallible detectors such as this can be combined with recomputation to increase error coverage at some overhead.

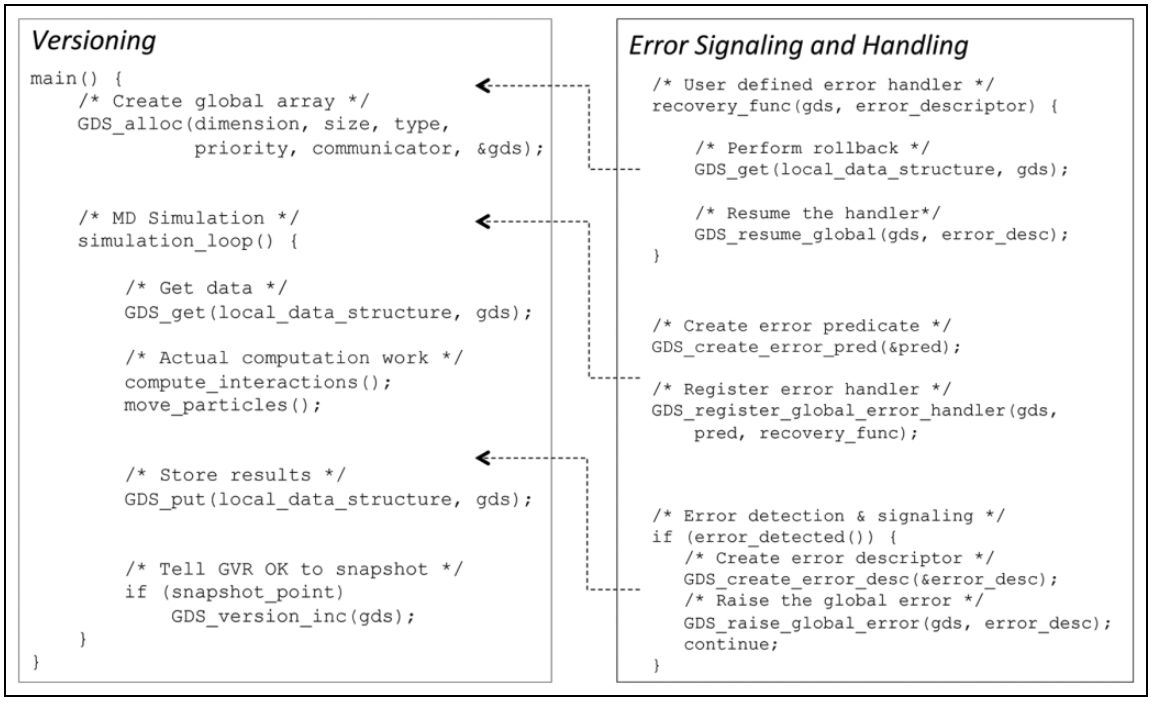

Figure 6. ddcMD error checking, signaling, and recovery using GVR.

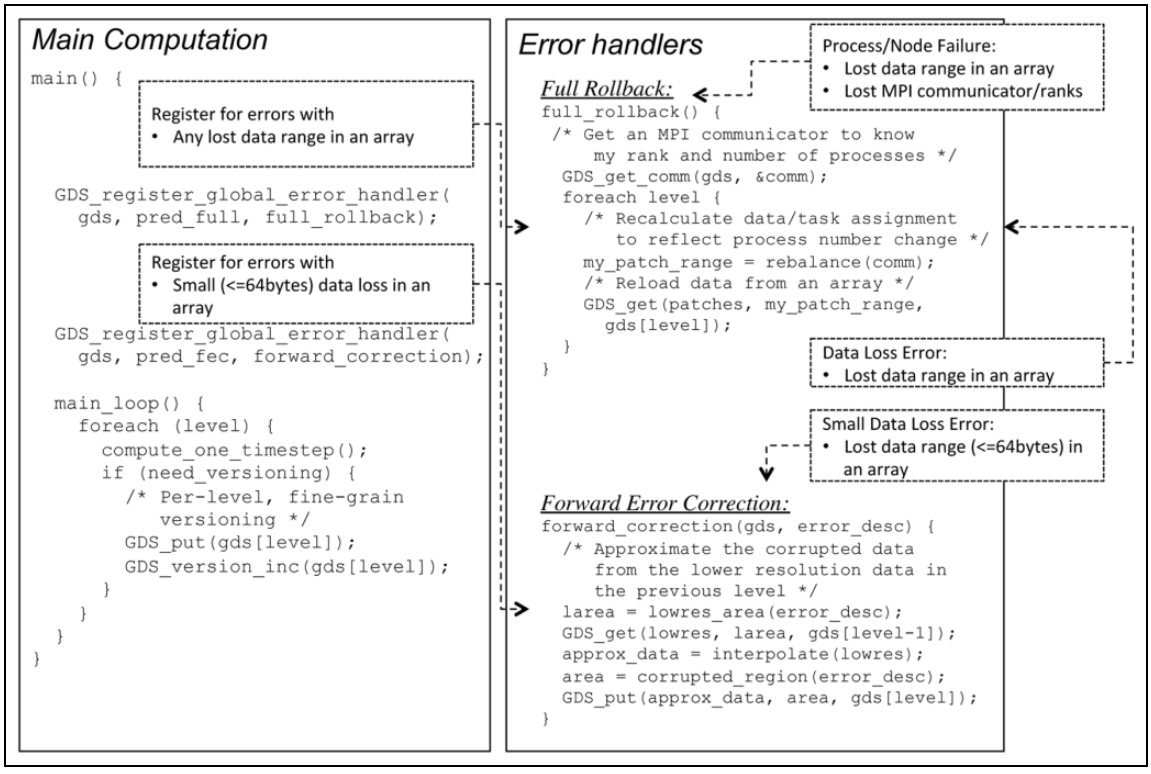

Figure 7. App-level data versioning and several recoveries in Chombo, enabled by GVR.

3.2.4 Chombo. For Chombo we introduced one array per level, plus one additional array for storing global metadata information. Figure 7 illustrates how GVR is applied to Chombo. We exploited a part of the existing HDF5-based checkpoint–restart functionality in Chombo for state preservation and restoration. In the figure, we demonstrate the power of the OR interface. Two error handlers with different recovery strategies

Table 1. Summary of target applications and their GVR usage.

Application	Percentage changed	Application lines of code	Leverage global view	Change software architecture
OpenMC	<2%	30,000	Yes	No
PCG/Trilinos	<1%	300,000	Yes	No
ddcMD	<0.3%	110,000	Yes	No
Chombo	<1%	500,000	Yes	No

are presented. They are registered to the system at the same time, and GVR invokes the most appropriate handler upon error. The first error handler performs full rollback of the data structure. This handler is registered for errors which contains lost data range as error attributes. This handler can handle two kinds of errors, process failures and data loss errors, as both errors contain lost data range. This is an example of generalization of error handlers. It is possible to use another kind of error handler which exploits the idea of forward error correction. In AMR, finer-level data has its counterpart in coarser level, so if data is corrupted in finer level it can be approximated using values in coarser level (Dubey et al., 2013). However, since the resulting values are not exact, caution must be exercised to ensure that the approximated values are within acceptable error bounds. It is not yet clear whether in Chombo's context which of the two kind of error handlers is more suitable for small data corruption since we have not yet implemented the forward error correction handler. In the example, the forward error correction handler is associated with small data corruption, so if the size of corruption is small, this handler is called. Otherwise (if the size is bigger), the first handler will be invoked for full rollback.

Across all four applications, our experience is that modest source code changes are sufficient to add GVR resilience to an application. Further, these changes are localized to a small part of the program, requiring no software architecture change, and are portable (machine-independent, involving only array manipulation at the level of the GVR interface). Table 1 documents these results quantitatively, with overall results of typically less than 1% code change. Even for OpenMC, which incorporates the global view model in the code deeply, the code changes were still below 2%. In all four applications, we found the global view for state preservation and recovery to be a powerful tool to understand how to identify errors and select appropriate recovery strategies. And the basic infrastructure for versioned arrays added to each appears to be a sufficient platform to scale forward to the high error rates anticipated in extreme-scale systems. In short, we conclude that GVR enables gentle-slope migration to higher resilience for existing scalable scientific applications.

3.3 Performance experiments

We explore the runtime overhead of adding GVR to an application by measuring execution time for several applications, varying versioning frequency and comparing with a base case. Specifically, we first run *native* applications with no modifications made to the original application codes and use that runtime as the baseline.[4]. Second we run applications linked with GVR library, but without any GVR operations (no puts, gets, or version_inc's). In runtime, GVR launches a server thread (called target server thread) per process which is used to implement collective operations such as allocation. This measurement isolates the target server thread overhead. Finally we measure the runtime of GVR versioning applications, which create versions of global arrays at varied frequencies. We choose versioning frequencies of every 30 minutes, 15 minutes, and 5 minutes (which leads to take 1 version, 2 versions, and 6 versions respectively in our experiments of a 30-minute run), successively increasing the versioning overhead. These times correspond to much shorter periods than would be used on today's systems and current MTTIs. The shortest of these periods (5 minutes) correspond to predicted MTTIs on extreme resilience scenarios for future exascale systems. We then compare the performance of each configuration and characterize the overhead of GVR versioning.

We configured each application run in the following ways.

- *OpenMC*: This uses PWR Performance Benchmark (Hoogenboom et al., 2011) with a total of 8,352,100 tally bins.
- *PCG*: This uses a sparse matrix generated according to HPCG benchmark (Heroux et al., 2013) of 105 million $\times$ 105 million with 26 non-zero values per row. Each of the 16 processes has a $820,000 \times 820,000$ sub-matrix. The data size of per sub-matrix, thus per process, is approximately 256 MB.
- *ddcMD*: The simulation is a system of 500 (Ta) atoms per process. Atom interaction is modeled with EAM potentials. The simulation runs for about 130,000 to 140,000 time steps.

- *Chombo*: We use a 3D gas-dynamics example reflecting a shockwave along a ramp (Colella, 1990). It uses purely explicit hyperbolic solve with space and time refinement, where AMR hierarchy has four levels and the refinement ratio is two. Refinement in time implies that for every time step dt taken by a coarse level, the next fine level takes n timesteps of size dt/n where n is the refinement ratio. The coarsest level has $128 \times 32 \times 128$, $128 \times 64 \times 128$, $128 \times 128 \times 128$, $128 \times 128 \times 256$ cells for 1024, 2048, 4096, and 8192 process run, respectively.

3.3.1 Hardware platforms. GVR is a portable system, as are many of the applications. So we performed experiments on the Mira Blue Gene/Q system at Argonne (see http://www.alcf.anl.gov/mira), the Edison Cray XC30 system at NERSC (see http://www.nersc.gov/users/computational-systems/edison/), and the UChicago Midway Linux cluster (see https://rcc.uchicago.edu/). We report ddcMD, PCG, and OpenMC measurements for the Midway cluster (284 nodes, dual 8-core Intel 2.6 GHz Xeon E5-2670, 32 GB). For Chombo, we report Edison measurements (5576 nodes, dual 12-core Intel IvyBridge 2.4 GHz, 64 GB) connected by Cray Aries with Dragonfly topology.

3.3.2 Results and discussion. Figure 8 shows the runtime (single run) of the applications in each configuration. The versioning overhead is less than 2% in all cases except the ddcMD run with 512 processes and versioning every 5 minutes. Specifically, for versioning every 30 minutes, which is a reasonable frequency under today's failure rates, the overhead is less than 1% for all applications. We also observe some negative overheads for ddcMD and PCG solver, which we conjecture may result from the unstable network of the cluster. The results also show that GVR scales at least up to 8000 processes. Figure 9 summarizes the versioning overheads in different applications.

Overall, the results indicate that adding GVR versioning incurs low overhead that can be managed by the application, which data, how frequently, to provide appropriate coverage, and thereby GVR provides a gentle-slope for application resilience.

4 GVR deep dives: multi-stream and forward error correction

4.1 Multi-stream in PCG linear system solvers

In this section, we use a deep-dive with the PCG solver to illustrate how multi-stream control of versioning enables tailoring of resilience to an application. Specifically, we show how the flexible control can be used to reduce application overhead both for versioning and recovery. We apply GVR *multi-stream* versioning to the sparse matrix (A), direction vector (p), and solution vector (x) as depicted in Figure 10, a notional example of different versioning intervals for each data structure. We vary these intervals in the experiments that follow. Versioning of all of these structures are done through the enhanced Trilinos vector and matrix classes as described in Section 3.2.

4.1.1 Experiments. We first explore how multi-stream versioning can be used to control and reduce the overhead of versioning in error-free PCG, varying versioning and versioning rate (see Figures 11 and 12). We run PCG with single process on the UChicago Midway cluster. Experiments use bcsstk18 from the University of Florida sparse matrix collection (Davis and Hu, 2011), a large symmetric and positive definite matrix. bcsstk18 is $11,948 \times 11,948$ matrix with 149,090 nonzeros. We run experiments with a convergence threshold of 0.001, leading to convergence in 739 iterations. All results are the average of five runs to reduce variation and measurement error. Starting with "full versioning" of all three structures every iteration (red bar), we consider eliminating versioning of first one (blue bars) and then two (green bars) of the structures. Eliminating versioning can reduce overhead dramatically, and reducing A's versioning gives the greatest benefit because it is largest.

Next we compare versioning all three structures in PCG to only one structure (see Figure 12), and vary the versioning rates. Eliminating versioning can reduce overhead dramatically, and at frequent versioning. But as the versioning frequency decreases (interval increases), the overall versioning overhead becomes quite small.

To achieve resilience with the best overall performance, the versioning overhead must be balanced against the cost of recovery. We illustrate how GVR's multi-stream versioning control can be used to optimize across these two often opposing considerations. In our experiment, we consider a single error injection at an arbitrary iteration, number 139, and inject a large bit-flip error (Elliott et al., 2013; Shantharam et al., 2011). We assume the error is detected immediately (end of the iteration). We consider error injection in the p vector and x vector in turn, but not the A matrix. The matrix A is versioned exactly once at the beginning of execution. We use τ_A, τ_p and τ_x to represent the versioning interval of A, p and x .

First, we compare the error-free execution to version-based recovery (see Figure 13). These three experiments all version A only once, p every 100 iterations and x every 10 iterations, that is, $\tau_p = 100$ and $\tau_x = 10$ (the recovery baseline). Our recovery procedure restores the most recent version of the direction (p)

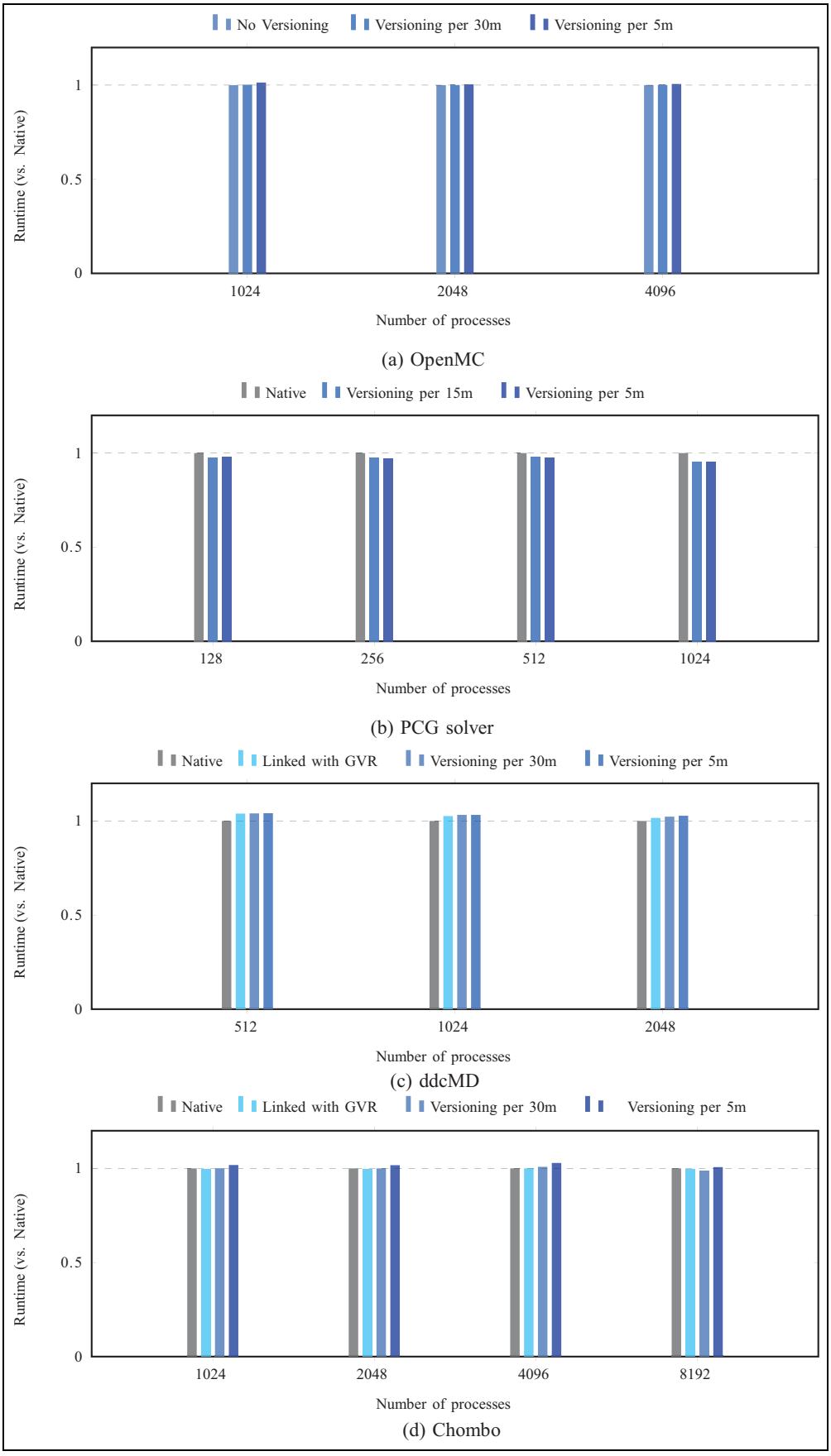

Figure 8. Versioning performance impact adding resilience using GVR.

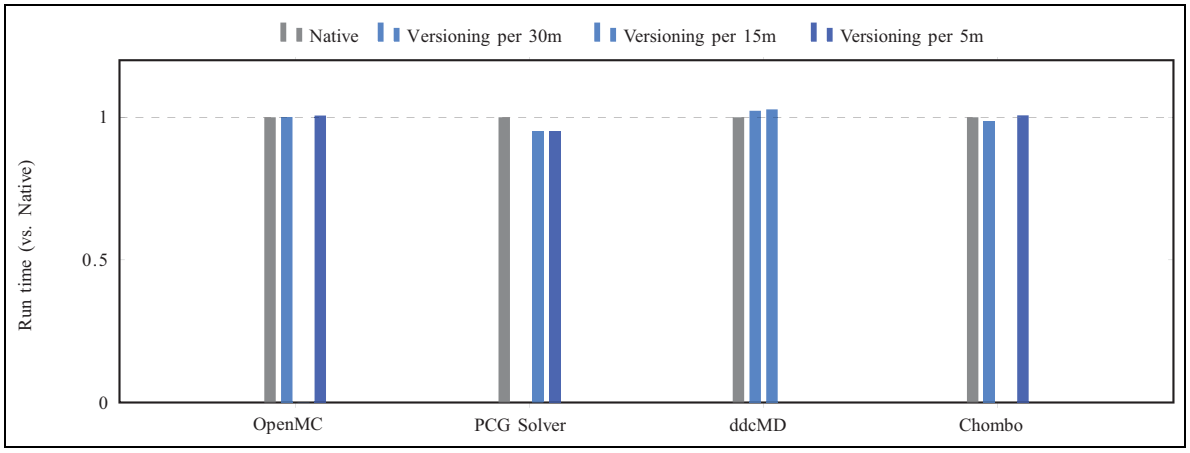

Figure 9. Summary of versioning overhead.

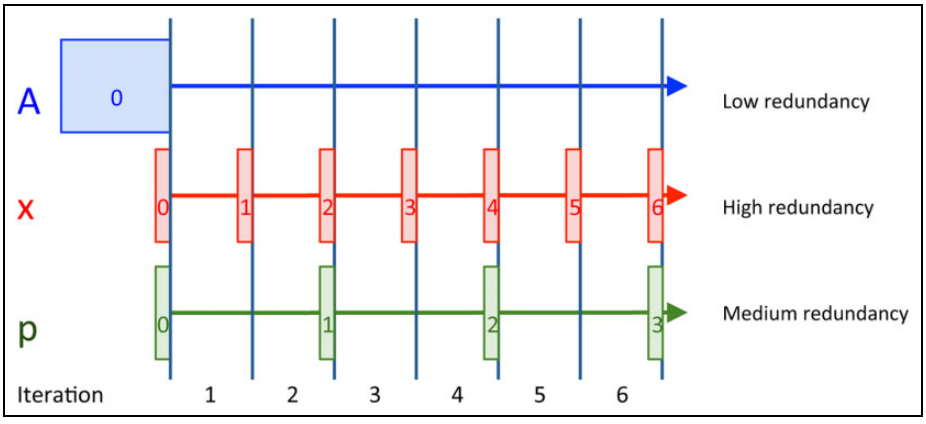

Figure 10. Example of multi-stream versioning: *A*, *x*, and *p* intervals are under programmer control.

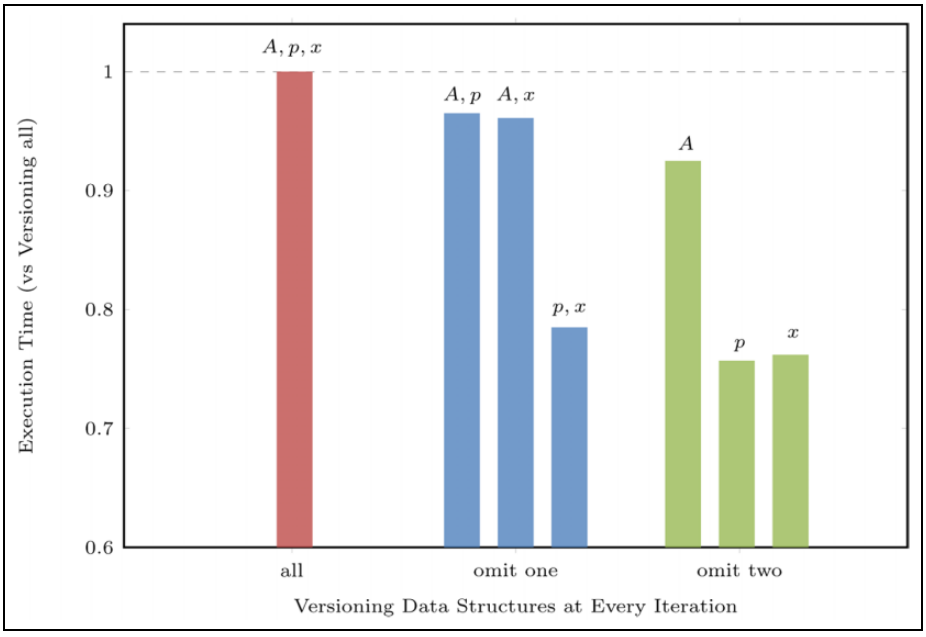

Figure 11. Selectively versioning *A*, *p* and *x* : all, omit one, omit two. Results are normalized to versioning all (*A*, *p*, *x*).

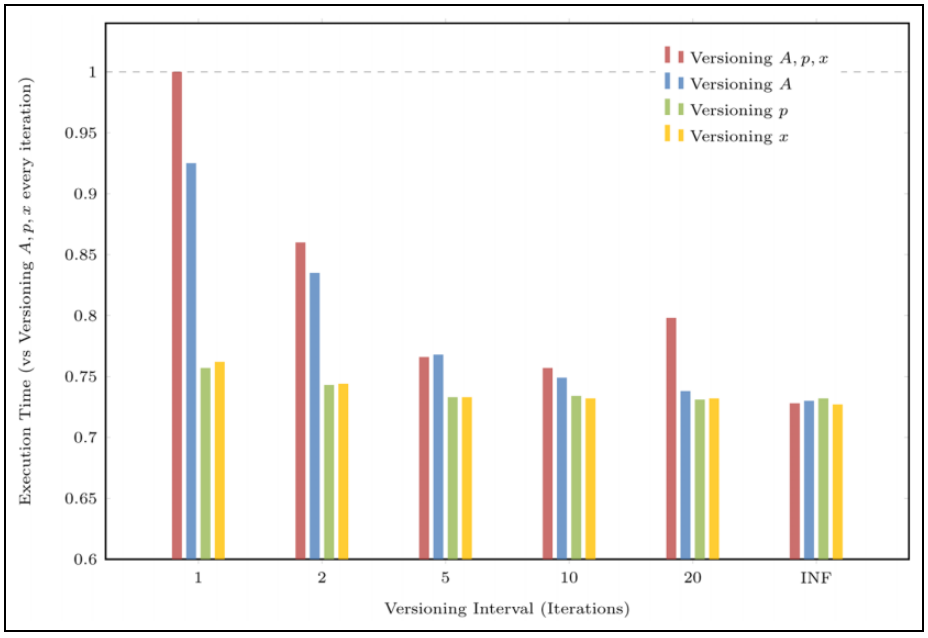

Figure 12. Versioning of *A*, *p* and *x* and omit two versus versioning interval. Results are normalized to versioning *A*, *p*, *x* at every iteration.

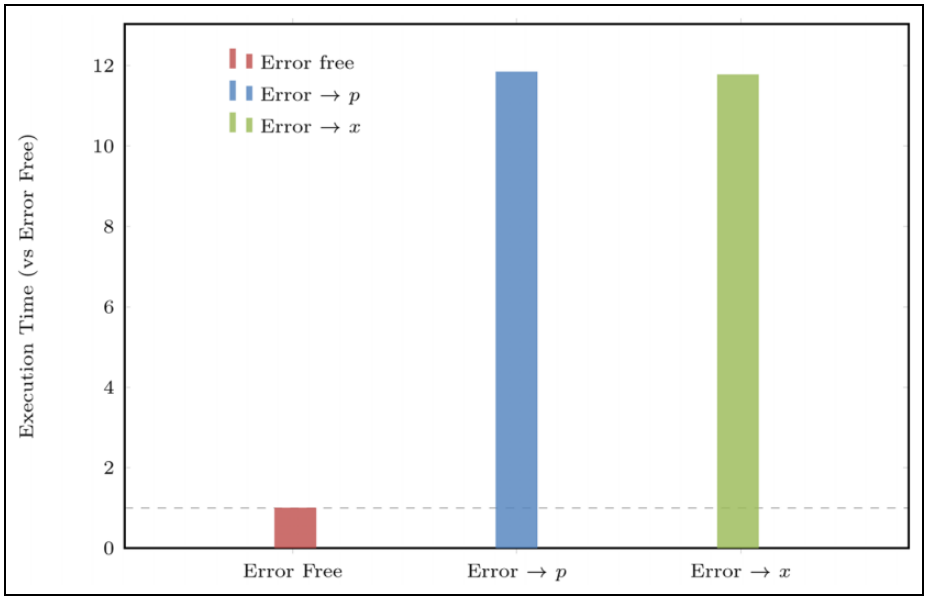

Figure 13. Runtime with error recovery. Varied error location, and with versioning intervals $\tau_p = 100$, and $\tau_x = 10$. Results are normalized to the error-free case.

and solution (x) vectors. In our experiments, the residual and residual norm vectors are never corrupted. Figure 13 shows empirically that the cost of error recovery can be high, increasing runtime by up to 12 × over the error-free runtime. The principal cause for increased runtime in our experiments is a large increase in the number of iterations from 739 to 8677. This increase is a product of error propagation into other solver state. The compute cost per iteration varies only slightly with versioning interval.

Next, we vary the versioning intervals for p and x to find the best intervals for each that maximize performance in the presence of errors and recovery. As shown in Figure 14(a) and (b), varying the versioning interval for x has a modest impact on the overall runtime while varying that for p has a much larger effect (over 10-fold). In fact, changing the versioning interval for x cannot reduce overall overhead and recovery cost to a useful range. However, the right choice of versioning

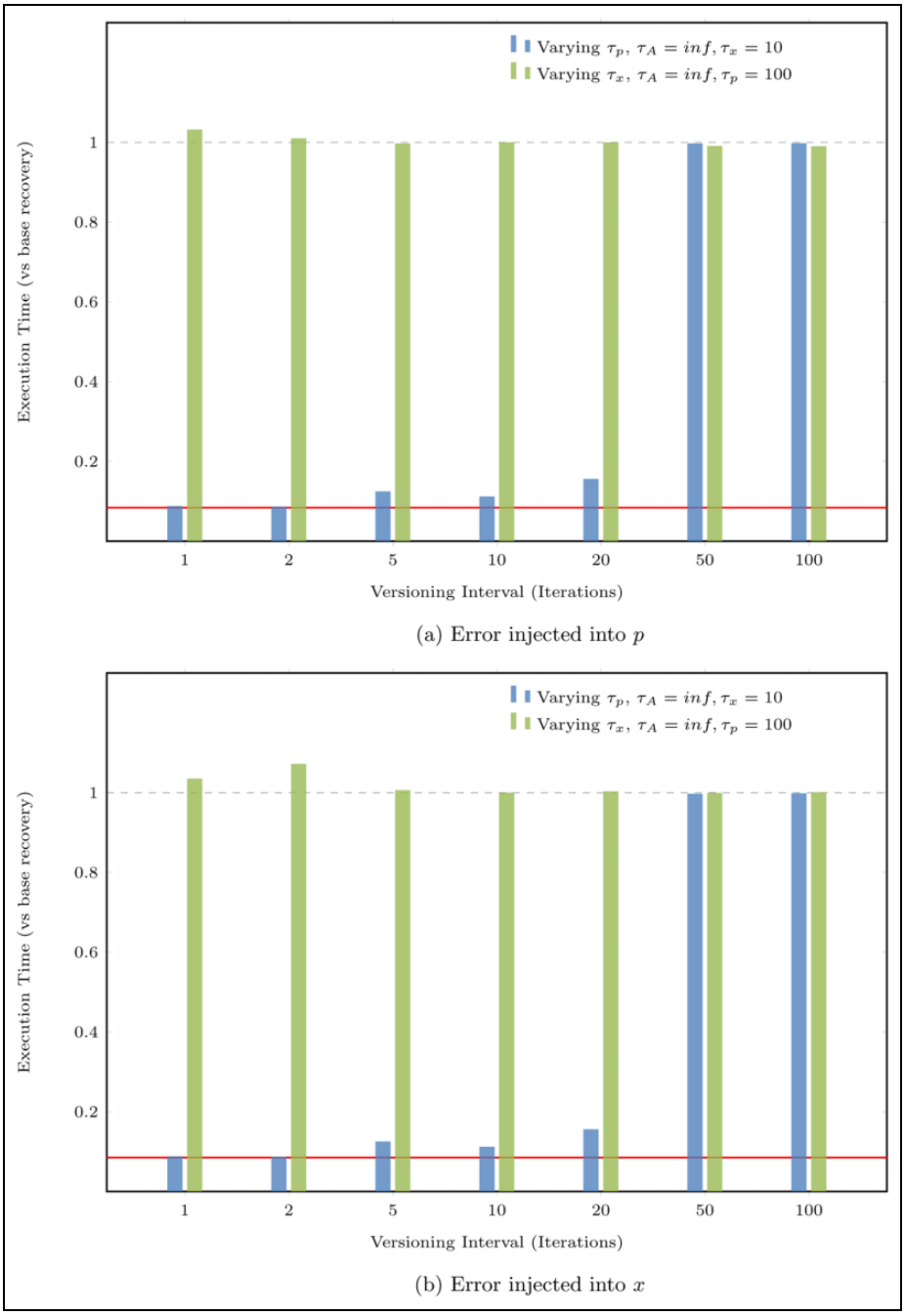

Figure 14. Comparison of runtime with recovery by varying versioning intervals τ_p and τ_x . Error is injected at iteration 139. The red line represents runtime of the error-free case. Execution times normalized to a recovery baseline with versioning intervals $\tau_p = 100$ and $\tau_x = 10$.

interval for p can, achieving only 3% overhead for the best interval of two iterations.

Overall, these results in PCG show that multi-stream versioning control can (1) provide control to separately reduce overhead per-data structure, and (2) provide flexible choice of versioning intervals to achieve efficient recovery. In the best cases, the tuned multi-stream versioned recovery can approach within 3% of the error-free case.

4.2 Forward error correction in OpenMC

Monte Carlo methods have shown a number of potential advantages over conventional deterministic methods in carrying out nuclear reactor core simulations (Martin, 2012; Siegel et al., 2013): the capability of simulating arbitrary geometrical and physics complexity, no approximation for neutron energy dependence, and inherent extreme parallelism for modern HPC architectures. OpenMC (Romano and Forget, 2013), a

Table 2. Data reliability in OpenMC.

Datastructure	Property	Management	Recovery
Geometry	Read-only	Caching	Reread from non-volatile storage
Cross section	Read-only	Caching	Recompute from cached good data
Tally data	Accumulate	Versioning	Forward error correction

production Monte Carlo neutron transport code, is capable of simulating 3D models based on constructive solid geometry with second-order surfaces. The application is written in FORTRAN, with support for a hybrid MPI/OpenMP parallelism.

However, there are still two major challenges that prevent Monte Carlo methods from being a realistic choice for full-core simulation. One is the enormous computational effort required to achieve acceptable statistics and source convergence, and the other is excessive demand of memory due to the large cross-section and tally data (Martin, 2012). Current tally accumulation approaches include either simple data replication (as in native OpenMC), or are based on application-controlled decomposition such as domain decomposition (Horelik et al., 2014) or client/server model-based data decomposition (Romano et al., 2013), which are limited by either memory cost, programmability, load imbalance, or performance loss. Accordingly, effective algorithms and implementations of Monte Carlo simulation are still required as a matter of urgency to enable scientists to harness the power of current and future exascale systems to conduct full-scale simulation.

During a simulation, there are three categories of data need to be stored in memory.

- Geometry: Geometry in Monte Carlo simulation is non-mesh based, read-only data and can be represented using *constructive solid geometry*.
- Interaction cross sections: The random events of each particle are determined by experimentally premeasured probability distributions, i.e. cross sections. The cross section is also read-only data and accessed randomly by each process during the simulation.
- Tallies: Tally data is region-based and accumulated (i.e. fetch-and-add) data, where the region, or tally region, is the volume over which the tallies should be integrated. The size of total tally data is directly proportional to the number of physical quantities to be tallied and the number of tally regions. In a realistic reactor simulation, that tally could reach terabytes size of data. Unlike geometry and cross sections, tally is only output data and not required for particles simulation; it is possible to process tally data in an asynchronous way.

4.2.1 Forward error correction. The reliability of OpenMC data structures are categorized as shown in Table 2.

Algorithm 1 Comparison of local accumulation, global view, and tally server algorithms

```
Global view: Create global arrays for tallies
for i ← 1 to M do
  for j ← 1 to N/p do
    while Particle j is alive do
      Process next event
      if Event satisfies filter criteria then
        for all Scoring functions do
          Calculate score
          Accumulate score to global view array
        end for
      end if
    end while
  end for
  Flush outstanding accumulate operations
end for
Write tally results to state point file
```

Cross-section and geometry data are stored on low-level, read-only, and is replicated over nodes. Therefore, they can be recovered simply by rereading from storage or other nearby nodes with replicas. It is also possible to recompute cross-section data from cached good data.

Data versioning is applied to tally data. At the end of each batch simulation (e.g. batch i), tally data can be snapshotted as a version T_i . Thus, we have a history of tally data $T_1 \ldots T_n$. Since the tally scoring is Monte Carlo accumulation only, if one *latent error* is detected at the latest batch n but this error occurred in batch i, then we are able to correct the contaminated T_n to the correct $T_n^{'}$ by removing the *contribution*, i.e. $\delta T = T_i - T_{i-1}$, caused by this latent error:

$$T_n^{'} = T_n - (T_i - T_{i-1}) \tag{1}$$

This *forward error correction* especially allows the application to recover from latent errors without rolling back to a previous state and conserving the effort in contaminated but correctable computation (Randell et al., 1978). For example, if using checkpointing/restart, the computation needs to roll back to batch $i-1$ with T_{i-1} and compute from batch i to batch n again. Figure 15 illustrates this scheme.

4.2.2 Tally implementation. Algorithm 1 shows the pseudocode of applying global arrays approach in

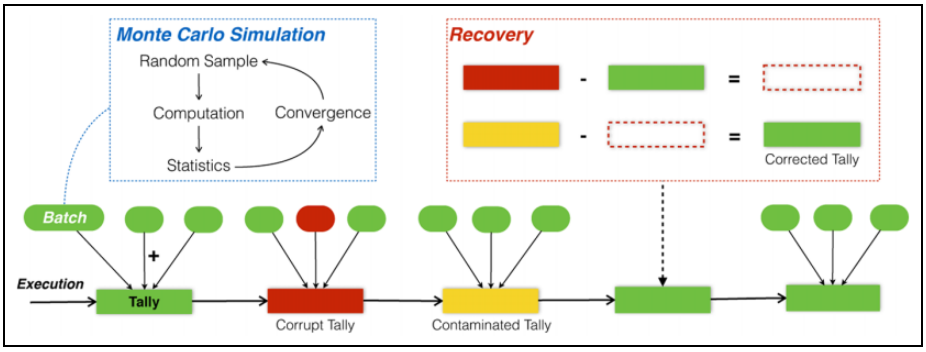

Figure 15. Illustration of forward error recovery in OpenMC.

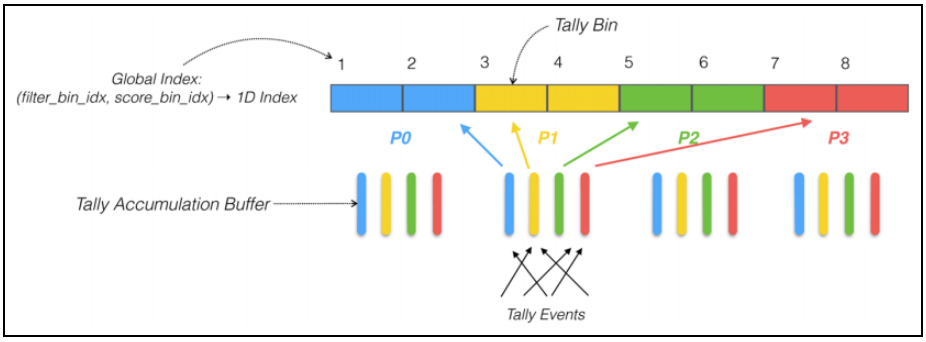

Figure 16. Illustration of global view implementation of tally data, using block distribution.

OpenMC. During initialization, each user-defined tally is allocated separately as a global array. In particle tracking routines, the only modification is to replace the original tally scoring function with GVR array accumulation calls when a scoring event occurs. When a batch of particles simulation finishes, the variances of tally score bins can be calculated by referencing corresponding tally array.

In native OpenMC implementation, one tally array consists of all tally bins/values corresponding to a specific tally (e.g. type of nuclides), which is represented as a 2D array indexed by *filter bin index* and *score bin index*. In global view implementation, as shown in Figure 16, we use a 1D GVR array to implement the original 2D array, where the mapping from 2D index to 1D global index is straightforward. Since the tally is accumulated into random bins during the simulation and thus data layout does not help access locality, we simply use block distribution to partition the array evenly on processes.

Using global view array naturally expresses the particle-base parallelism in OpenMC thus significantly enhances the programmability. Integrating GVR arrays requires fairly small changes (less than 1% LOC) of OpenMC code. Figure 17 shows the actual code changes required by using GVR library.

As shown on the right of Figure18 the application programmer would typically define an error recovery function to handle a class of errors and register it. The error can be recovered by forward correction approach or rollback fully, depending on the properties of errors. Through GVR, applications can utilize the OR interface for error signaling and handling. Figure 18 shows the GVR unified dispatch code, by using OR interface. An error handler `do_recovery` is newly introduced and registered to an array. The error handler is also associated with error matching predicates, to express what kind of errors can be handled by this handler. By registering an error handler through the OR interface, the handler can receive all error signals from any kind of sources such as hardware, operating system, runtime libraries, or the application itself, as long as the error consists of a set of error attributes which satisfies the pre-registered predicate.

4.2.3 Experiments. Figure 19 shows results up to 16,384 processes of GVR-enabled OpenMC with 2.4 TB array size on 1366 nodes of Edison Cray XC30 system (weak scaling). At maximum size, OpenMC with GVR achieves 85% efficiency for 16,384 processes with 1024 processes as the baseline.

To illustrate how forward error correction can help OpenMC efficiently recover from *latent errors*, we compare the recovery efficiencies of using following three error recovery approaches.

```
! === Tally Allocation ===
subroutine setup_tally_arrays()
#ifdef GVR
  ndim = 1
  arr_size = total_score_bins * total_filter_bins
  call gds_alloc(ndim, arr_size, ..., tally_arr)
#else
  allocate(tally_arr(total_score_bins, total_filter_bins))
#endif
end subroutine

!=== Tally Accumulation ===
subroutine score_analog_tally()
  ! score calculation
#ifdef GVR
  index = score_index * total_score_bins + filter_index
  call gds_acc(score, index, ..., tally_arr)
#else
  tally_arr(score_index, filter_index) % value += score
#endif
end subroutine
```

Figure 17. Code snippet of tally allocation and accumulation.

Table 3. System parameters for flexible recovery efficiency study.

	Higher error rate	Lower error rate
Nodes (N)	1024	1024
Node FIT (α)	36,621	8610
Versioning overhead (δ)	0.3 seconds	0.3 seconds
Versioning interval (τ)	250 seconds	500 seconds

- Rollback/restart: the application rolls back to a previous version/snapshot of tally data right before the error occurred, and compute all batches again. For example, some errors occurred in batch i and there is a version of good tally data at batch i, but the detection of errors are delayed until batch $i + k$, then the application calls *GDS_move_to_prev()* to checkout the good version and redo k batches.
- Forward error correction with additional batches: the application uses the forward error correction described in Section 4.2.1 to remove the contribution between to consecutive versions that enclose the error, then redo all the contaminated batches between these two consecutive versions. For example, similar as above, some errors occurred at batch i but detected at batch $i + k$. If there is a version at batch $i + l$ ($l < k$), then the δ is computed by using version $i + l$ and version i and removed from current version. Finally, l batches are re-run to compensate lost accumulation.
- Forward error correction: this approach is the same as above but without re-running l batches to compensate. Instead, the application can decide how to deal with error-removed data: to compensate it or simply tolerate it. This approach is useful if the detector can indicate specific error locations while the errors are significant but it is sufficient to remove errors without re-run additional batches. For example, whether to re-run additional batches or not can be determined by statistical convergence criteria.

To exploring the impact of failure rate and detection latency on recovery efficiency, we use the failure model proposed by Snir et al. (2014) to manually inject errors and set versioning interval and detection latency. Specifically, the optimal versioning interval τ is calculated by $\sqrt{2\delta M}$ (Young, 1974), where δ is the time to take a version and M is the MTBF that can be derived from empirical node FIT values. Table 3 documents the experiment parameters.

As shown in Figure 20, using forward error correction achieves better recovery efficiency as error detection latency increases. Because rollback discards all of the computation since error occurrence, the loss for each error is proportional to the detection latency since it has to rollback to the version before the error happened. Note that this may be better than checkpoint restart, which will fail if the error detection latency exceeds the detection latency. In contrast, forward correction is able to conduct the in-place compensation for current version of data with optional additional batches, saving much of the intervening computation. Note that in Figure 20a, since the versioning interval is

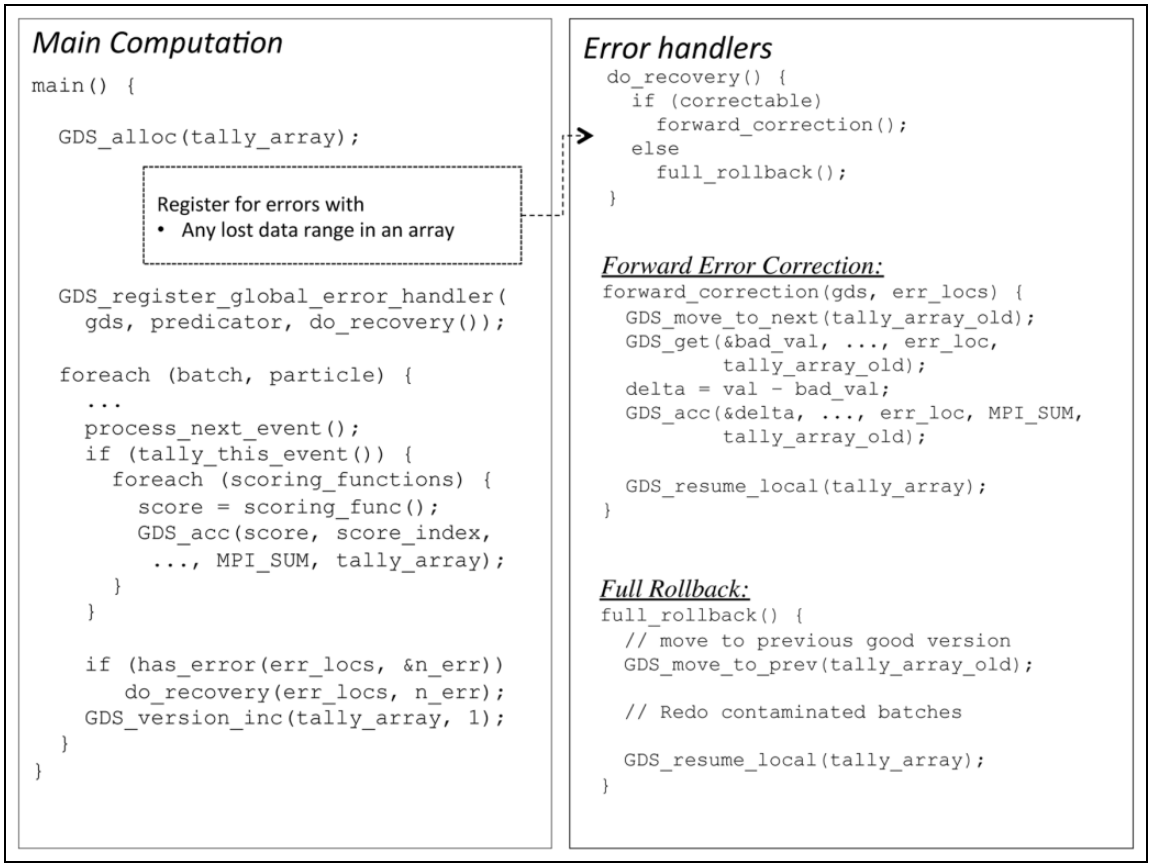

Figure 18. Expandable GVR error signalling and handling in OpenMC.

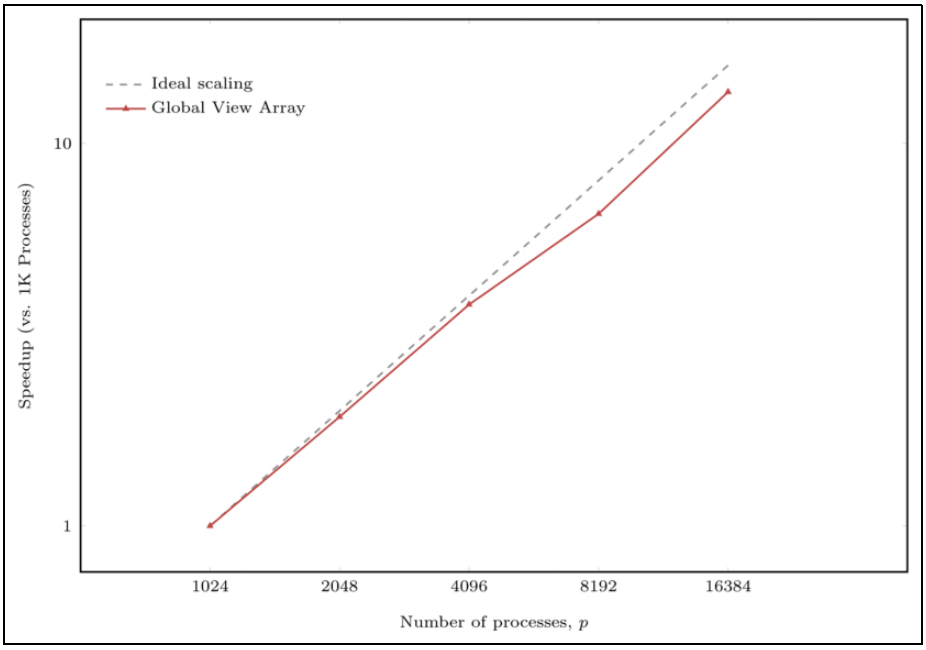

Figure 19. Scalability of OpenMC using GVR.

taken every 5 batches, the number of additional rerun batches (i.e. blue bars) is always 5 for 15, 30, and 45 detection latency. With 10-batch versioning interval, as shown in Figure 20b the number of re-run batches is 5 for 15 and 45 detection latency, and 10 for 30 detection latency.

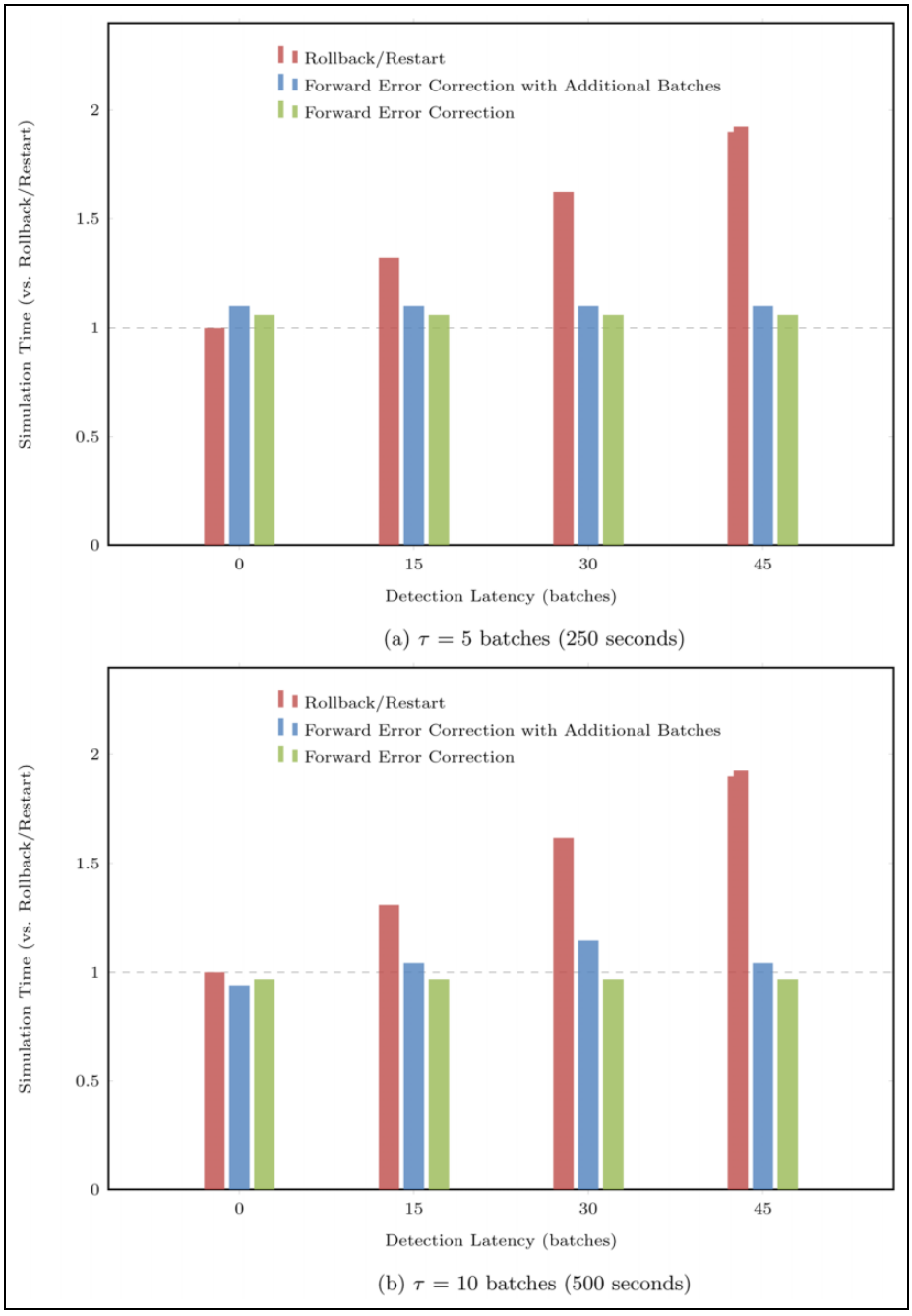

Figure 20. OpenMC runtime for 50 batches with varied error recovery (1000 processes, 8 GB tallies).

5 Discussion and related work

For decades, checkpoint–restart has been the mainstay of reliability in high-performance computing, and a wealth of research in appropriate checkpoint frequencies (Daly, 2006; Young, 1974), data compression (Hargrove and Duell, 2006; Hogan et al., 2012; Lakshminarasimhan et al., 2013), and recently efficient exploitation of non-volatile storage as in SCR and FTI (Bautista-Gomez et al., 2011; Moody et al., 2010) has steadily increased its efficiency.

GVR's global arrays trace their heritage to PGAS-style libraries and languages such as Global Arrays (GA) (Nieplocha et al., 2006), UPC (Carlson et al., 1999), Co-Array Fortran (Numrich and Reid, 1998), X10 (Charles et al., 2005), and Chapel (Chamberlain et al., 2007). GVR programmers access arrays with APIs that are derived from the Global Array library (Nieplocha et al., 2006), extending them with new operations that to create versions of each distributed array, names for each version, and navigation amongst versions for an array. In contrast, checkpoint–restart

approaches typically maintain only a single checkpoint, so they can be used to recover from errors detected immediately (checkpoints are uncorrupted). In contrast, GVR versions can be used to recover from latent errors (Aupy et al., 2013; Hogan et al., 2012), and to do so effectively in the face of high error rates (Lu et al., 2013b).

While applications using checkpoint–restart can control the checkpoint rate, that is a single knob for the entire application. GVR provides per-data structure redundancy, with application control of when versions can be created, so each array's redundancy can be tuned to match application need.[5]

Finally, GVR empowers applications to use the full complement of versions captured for each array to do sophisticated application data recovery. In checkpoint–restart systems that lack the ability to name multiple checkpoints, partially materialize them, and manipulate them conveniently, any such flexible recovery is difficult. For example, SCR (Moody et al., 2010) and FTI (Bautista-Gomez et al., 2011) preserve multiple checkpoints internally, they provide a way for applications to name and manipulate multiple versions. To these, GVR adds flexible version labels, navigation, and flexible read access to arbitrary parts of versions to support a wide range of latent error recovery and forward error recovery.

While a relatively new concept in high-performance fault resilience, multi-version concepts have been explored in other concurrent systems including database views (sometimes called snapshot views), time-stamped values for a key in Google's BigTable (Chang et al., 2006) and Apple's Time machine (see http://www.apple.com/osx/apps/#timemachine), and a variety of research snapshot extensions of Linux filesystems (Konishi et al., 2006; Rodeh et al., 2013).

Several research planning reports have advocated cross-layer resilience (DeHon et al., 2011) to both increase the number of recoverable errors and make recovery efficient. However, existing examples of cross-layer error recovery (Bridges et al., 2012; Glosli. et al., 2007) are "stovepipes", with a one-to-one mapping of error handler and error event. GVR's approach here has two novelties: associating error handling with data (distributed arrays) and flexibly associating handlers and errors. GVR's OR uses pattern matching to flexibly pair error handlers and error events. And by doing so, to increase the return on investment for both error signallers and handlers. An early unified error signaling interface can be found in CIFTS (Gupta et al., 2009), but that work is currently inactive. Checkpoint/restart also provides no means to any form of extensible error checking or flexible recovery.

The wealth of work on application-specific techniques such as approximate execution (Carbin et al., 2013; Misailovic et al., 2014; Sampson et al., 2011) and fault-tolerant algorithms (Bronevetsky and de Supinski, 2008; Hari et al., 2012; Heroux, 2014) (such as ABFT) can enable efficient error detection and recovery. This work complements our research on GVR, and we hope that not only these published approaches, but many more will be implemented atop GVR. As described in Section 2, it is our goal to support both the application error checking and recovery aspects of these applications with GVR.

Other reliability techniques such as replicated (Lidman et al., 2012; Lyons and Vanderkulk, 1962) have been proposed, and evaluated at scale. There are situations where they can be appropriate, but they often incur high overhead during failure-free execution, and have a limited set of options on the range of error rate and cost. GVR enables flexible balance of error coverage and overhead.

6 Summary and future work

We presented GVR, a library that enables the construction of portable, resilient applications. By adding GVR calls to capture key data structures, one can implement resilience that is matched to the data structure needs, and then tune coverage to match to increasing error rates. Programming experience with GVR shows that it can be added to large applications with little effort ($<2\%$ code changes), and support both rollback recovery and a broad range of multi-version and forward recovery approaches that may be important for future extreme-scale systems. Adding GVR versioning to an application is low cost (<2 % runtime) at today's error rates. The PCG solver case study shows that multi-stream versioning can provide per-data structure control to reduce overhead and enable flexibility to achieve efficient recovery. While we have shown only one example of multi-stream versioning, there are many possible uses. For example, in a multi-physics simulation where different model variables (e.g. velocity, temperature, density) evolve at different time scales. Varying the versioning rate as the computational modeling rate is varied may be appropriate. Besides many application have immutable table data (e.g. cross-section data for neutronics). In such cases, immutable data can be versioned only once with other versioned appropriately for resilience. Our experiments with flexible forward error correction demonstrate that large performance gains are possible compared with rollback restart schemes when latent errors occur. Future directions include incorporating GVR into high-level programming models and tools, such the X-stack program research (see https://xstackwiki.modelado.org/Extreme_Scale_Software_Stack) and deeper exploration of the opportunities presented by a resilience ecosystem as posited by OR. In addition, techniques to further optimize version implementation, including

efficient differences, compression, and exploitation of NVRAM are all promising directions.

Declaration of conflicting interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, US Department of Energy, under Award DE-SC0008603 and Contract DE-AC02-06CH11357. This work was completed in part with resources provided by: the University of Chicago Research Computing Center, the resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the US Department of Energy under Contract No. DE-AC02-05CH11231, and resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

Notes

1 Error detection is a rich and long-standing research area. GVR enables errors from all layers of the system, including application, system software, and hardware to be signalled, and handled through its unified error-handling interface.
2 Specific error-handling strategies can be defined for a system, or specific to an application. We give several examples later, but the general question of error handling strategy is left to the application.
3 The increment can be specified by the user, and if desired a version label can be applied.
4 Note that the native version of OpenMC did not scale beyond 64 nodes for the data set we use due to a design limitation of the tally data structure. So instead we compare with the GVR base version for OpenMC.
5 While application programmers could implement this manually using some application-based checkpointing techniques, GVR provide direct naming and API support for this.

References

Antypas K, Wright N, Cardo NP, Andrews A and Cordery M (2014) Cori: A Cray XC pre-exascale system for NERSC. In: *Cray User Group Proceedings*. Cray Research, Inc.

Aupy G, Benoit A, Herault T, Robert Y, Vivien F and Zaidouni D (2013) On the combination of silent error detection and checkpointing. In: *2013 IEEE 19th Pacific Rim International Symposium on Dependable Computing (PRDC)*, pp. 11–20. DOI:10.1109/PRDC.2013.10.

Bariuso R and Knies A (1994) Shmem user's guide. Cray Research, Inc.

Bautista-Gomez L, Tsuboi S, Komatitsch D, Cappello F, Maruyama N and Matsuoka S (2011) FTI: High performance fault tolerance interface for hybrid systems. In: *SC '11*. DOI:10.1145/2063384.2063427.

Berger M and Colella P (1989) Local adaptive mesh refinement for shock hydrodynamics. *Journal of Computational Physics* 82(1): 64–84.

Berger M and Oliger J (1984) Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics* 53(3): 484–512.

Bergman K, et al (2008) Exascale computing study: Technology challenges in achieving exascale systems. DARPA IPTO Technical Report.

Borkar S and Chien AA (2011) The future of microprocessors. *Communications of the ACM* 54: 67–77.

Bridges PG, Hoemmen M, Ferreira KB, Heroux MA, Soltero P and Brightwell R (2012) Cooperative application/OS DRAM fault recovery. In: *Resilience'11*, pp. 241–250. DOI:10.1007/978-3-642-29740-3_28.

Bronevetsky G and de Supinski B (2008) Soft error vulnerability of iterative linear algebra methods. In: *ICS*.

Cappello F, Casanova H and Robert Y (2011) Preventive migration vs. preventive checkpointing for extreme scale supercomputers. *Parallel Processing Letters* 21(02): 111–132.

Cappello F, Geist A, Gropp W, Kale L, Kramer W and Snir M (2009) Towards exascale resilience. *International Journal of High Performance Computing Applications* 23(4): 374–388.

Carbin M, Misailovic S and Rinard MC (2013) Verifying quantitative reliability for programs that execute on unreliable hardware. In: *OOPSLA '13*, pp. 33–52. DOI:10.1145/2509136.2509546.

Carlson W, Draper J, Culler D, Yelick K, Brooks E and Warren K (1999) Introduction to UPC and language specification. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences.

Chamberlain BL, Callahan D and Zima HP (2007) Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications* 21(3): 291–312.

Chang F, Dean J, Ghemawat S, Hsieh WC, Wallach DA, Burrows M, Chandra T, Fikes A and Gruber RE (2006) Bigtable: A distributed storage system for structured data. In: *OSDI '06*, p. 15.

Charles P, Grothoff C, Saraswat V, Donawa C, Kielstra A, Ebcioglu K, von Praun C and Sarkar V (2005) X10: An object-oriented approach to non-uniform cluster computing. In: *OOPSLA '05*, pp. 519–538. DOI:10.1145/1094811.1094852.

Chen Z (2013) Online-ABFT: an online algorithm based fault tolerance scheme for soft error detection in iterative methods. In: *PPoPP '13*, pp. 167–176. DOI:10.1145/2442516.2442533.

Colella P (1990) Multidimensional upwind methods for hyperbolic conservation laws. *Journal of Computational Physics* 87(1): 171–200.

Colella P, Graves D, Keen N, Ligocki T, Martin D, McCorquodale P, Modiano D, Schwartz P, Sternberg T and Van Straalen B (2009) Chombo software package for AMR applications design document. Technical report, LBNL,

Applied Numerical Algorithms Group, Computational Research Division.

Daly JT (2006) A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Computer Systems* 22(3). DOI:10.1016/j.future.2004.11.016.

Davis TA and Hu Y (2011) The university of florida sparse matrix collection. *ACM Transaction on Mathematical Software* 38(1): 1:1–1:25.

DeHon A, Carter N and Quinn H (2011) Final report for CCC cross-layer reliability visioning study. Available at: http://www.cra.org/ccc/xlayer.php.

Dennard RH, Gaensslen FH, Rideout VL, Bassous E and LeBlanc AR (1974) Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits* 9(5): 256–268.

Di Martino C, Kalbarczyk Z, Iyer R, Baccanico F, Fullop J and Kramer W (2014) Lessons learned from the analysis of system failures at petascale: The case of Blue Waters. In: *DSN 2014*, pp. 610–621. DOI:10.1109/DSN.2014.62.

Dubey A, Mohapatra P and Weide K (2013) Fault tolerance using lower fidelity data in adaptive mesh applications. In: *Proceedings of the 3rd Workshop on Fault-tolerance for HPC at Extreme Scale*. New York: ACM Press, pp. 3–10. DOI:10.1145/2465813.2465817.

Dun N, Fujita H, Tramm J, Chien AA and Siegel AR (2014) Data decomposition in Monte Carlo particle transport simulations using global view arrays. Technical Report TR-2014-09, Department of Computer Science, University of Chicago.

Elliott J, Mueller F, Stoyanov M and Webster C (2013) Quantifying the impact of single bit flips on floating point arithmetic. Technical Report TR 2013-2, Department of Computer Science, North Carolina State University.

Elnozahy M (2009) System resilience at extreme scale: A white paper. DARPA Resilience Report for ITO, William Harrod.

Esmaeilzadeh H, Blem E, St Amant R, Sankaralingam K and Burger D (2011) Dark silicon and the end of multicore scaling. In: *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, pp. 365–376.

Fang A and Chien AA (2014) Applying GVR to molecular dynamics: Enabling resilience for scientific computations. Technical Report TR-2014-04, University of Chicago.

Fang A and Chien AA (2015) How much SSD is useful for resilience in supercomputers. Manuscript submitted for publication.

Ferreira K, Stearley J, Laros JH III, Oldfield R, Pedretti K, Brightwell R, Riesen R, Bridges PG and Arnold D (2011) Evaluating the viability of process replication reliability for exascale systems. In: *SC '11*.

Fiala D, Mueller F, Engelmann C, Riesen R, Ferreira K and Brightwell R (2012) Detection and correction of silent data corruption for large-scale high-performance computing. In: *Proceedings of Supercomputing*, p. 78.

Glosli JN, Richards DF, Caspersen KJ, Rudd RE, Gunnels JA and Streitz FH (2007) Extending stability beyond CPU millennium: a micron-scale atomistic simulation of Kelvin–Helmholtz instability. In: *Proceedings of SC '07*, pp. 1–11. DOI:10.1145/1362622.1362700.

Golub GH and Van Loan CF (1996) *Matrix Computations* (3rd ed.). Baltimore, MD: Johns Hopkins University Press.

Goorley T, James M, Booth T, Brown F, Bull J, Cox LJ, Durkee J, Elson J, Fensin M, Forster RA, Hendricks J, Hughes HG, Johns R, Kiedrowski B, Martz R, Mashnik S, McKinney G, Pelowitz D, Prael R, Sweezy J, Waters L, Wilcox T and Zukaitis T (2012) Initial MCNP5 release overview. *Nuclear Technology* 180(3): 298–315.

Gupta R, Beckman P, Park BH, Lusk E, Hargrove P, Geist A, Panda D, Lumsdaine A and Dongarra J (2009) CIFTS: A coordinated infrastructure for fault-tolerant systems. In: *Proceedings of ICPP '09*, pp. 237–245. DOI:10.1109/ICPP.2009.20.

GVR Team (2014a) Global View Resilience (GVR) API documentation, version 1.0. Technical report, University of Chicago, Department of Computer Science.

GVR Team (2014b) Global View Resilience (gvr) documentation, release 1.0. Technical Report TR-2014-13, University of Chicago, Department of Computer Science.

Hargrove PH and Duell JC (2006) Berkeley Lab checkpoint/restart (BLCR) for Linux clusters. *Journal of Physics: Conference Series* 46: 494.

Hari SKS, Adve SV and Naeimi H (2012) Low-cost program-level detectors for reducing silent data corruptions. In: *IPDPS*.

Heroux MA (2014) Toward resilient algorithms and applications. *CoRR* Preprint abs/1402.3809.

Heroux MA, Bartlett RA, Howle VE, Hoekstra RJ, Hu JJ, Kolda TG, Lehoucq RB, Long KR, Pawlowski RP, Phipps ET, et al. (2005) An overview of the trilinos project. *ACM Transactions on Mathematical Software* 31(3): 397–423.

Heroux MA, Dongarra J and Luszczek P (2013) HPCG technical specification. Technical report, Sandia National Laboratories.

Hogan S, Hammond JR and Chien AA (2012) An evaluation of difference and threshold techniques for efficient checkpoints. In: *2012 IEEE/IFIP 42nd International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, pp. 1–6.

Hoogenboom JE, Martin WR and Petrovic B (2011) The Monte Carlo performance benchmark test - aims, specifications and first results. In: *ANS M&C*.

Horelik N, Forget B, Smith K and Siegel A (2014) Domain decomposition and terabyte tallies with the OpenMC Monte Carlo neutron transport code. In: *PHYSOR 2014 – Advances in Reactor Physics – The Role of Reactor Physics toward a Sustainable Future*.

Huang KH and Abraham JA (1984) Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers* 33(6): 518–528.

Kershaw DS (1978) The incomplete Cholesky–conjugate gradient method for the iterative solution of systems of linear equations. *Journal of Computational Physics* 26(1): 43–65.

Konishi R, Amagai Y, Sato K, Hifumi H, Kihara S and Moriai S (2006) The Linux implementation of a log-structured file system. *SIGOPS Operating Systems Review* 40(3): 102–107.

Lakshminarasimhan S, Shah N, Ethier S, Ku SH, Chang CS, Klasky S, Latham R, Ross R and Samatova NF (2013)

Isabela for effective in situ compression of scientific data. *Concurrency and Computation: Practice and Experience* 25(4): 524–540.

Lidman J, Quinlan DJ, Liao C and McKee SA (2012) ROSE::FTTransform-a source-to-source translation framework for exascale fault-tolerance research. In: *FTXS'12*.

Lu Cd and Reed DA (2004) Assessing fault sensitivity in MPI applications. In: *Proceedings of Supercomputing*.

Lu G, Zheng Z and Chien AA (2013a) When is multi-version checkpointing needed? In: *FTXS '13*. New York: ACM Press. DOI:10.1145/2465813.2465821.

Lu G, Zheng Z and Chien AA (2013b) When is multi-version checkpointing needed? In:*Proceedings of the 3rd Workshop on Fault-tolerance for HPC at Extreme Scale (FTXS '13)*. New York: ACM Press. DOI:10.1145/2465813.2465821.

Lyons RE and Vanderkulk W (1962) The use of triple-modular redundancy to improve computer reliability. *IBM Journal of Research and Development* 6(2): 200–209.

Martin WR (2012) Challenges and prospects for whole-core Monte Carlo analysis. *Journal of Nuclear Engineering Technology* 44(2): 151–160.

Misailovic S, Carbin M, Achour S, Qi Z and Rinard MC (2014) Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels. In: *OOPSLA '14*, pp. 309–328. DOI:10.1145/2660193.2660231.

Moody A, Bronevetsky G, Mohror K and Supinski BRd (2010) Design, modeling, and evaluation of a scalable multi-level checkpointing system. In: *SC '10*. IEEE Computer Society. DOI:10.1109/SC.2010.18.

Nieplocha J, Palmer B, Tipparaju V, Krishnan M, Trease H and Aprà E (2006) Advances, applications and performance of the Global Arrays shared memory programming toolkit. *The International Journal of High Performance Computing Applications* 20(2): 203–231.

Numrich RW and Reid J (1998) Co-array fortran for parallel programming. *SIGPLAN Fortran Forum* 17(2): 1–31.

Peter Kogge, et al (2008) Exascale computing study: Technology challenges in achieving exascale systems. DARPA IPTO Study Report. `Available at: http://users.ece.gatech.edu/mrichard/ExascaleComputingStudyReports/exascale_final_report_100208.pdf`.

Randell B, Lee P and Treleaven PC (1978) Reliability issues in computing system design. *ACM Computer Surveys* 10(2): 123–165.

Rodeh O, Bacik J and Mason C (2013) Btrfs: The Linux B-tree filesystem. *Transactions on Storage* 9(3): 9:1–9:32.

Romano PK and Forget B (2013) The OpenMC Monte Carlo particle transport code. *Annals of Nuclear Energy* 51: 274–281.

Romano PK, Siegel AR, Forget B and Smith K (2013) Data decomposition of Monte Carlo particle transport simulations via tally servers. *Journal of Computational Physics* 252: 20–36.

Rubenstein Z, Fujita H, Zheng Z and Chien A (2013) Error checking and snapshot-based recovery in a preconditioned conjugate gradient solver. Technical Report TR-2013-11, Department of Computer Science, University of Chicago.

Saad Y (1993) A flexible inner-outer preconditioned GMRES algorithm. *SIAM Journal on Scientific Computing* 14(2): 461–469.

Sampson A, Dietl W, Fortuna E, Gnanapragasam D, Ceze L and Grossman D (2011) EnerJ: Approximate data types for safe and general low-power computation. In: *PLDI '11*, pp. 164–174. DOI:10.1145/1993498.1993518.

Schlichting RD and Schneider FB (1983) Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computing Systems* 1(3): 222–238.

Schroeder B and Gibson GA (2006) A large-scale study of failures in high-performance computing systems. In: *DSN*.

Shantharam M, Srinivasmurthy S and Raghavan P (2011) Characterizing the impact of soft errors on iterative methods in scientific computing. In: *Proceedings of Supercomputing*.

Siegel AR, Smith K, Romano PK, Forget B and Felker K (2013) The effect of load imbalances on the performance of Monte Carlo codes in LWR analysis. *Journal of Computational Physics* 235: 901–911.

Snir M, Wisniewski RW, Abraham JA, Adve SV, Bagchi S, Balaji P, Belak J, Bose P, Cappello F, Carlson B, Chien AA, Coteus P, Debardeleben NA, Diniz P, Engelmann C, Erez M, Fazzari S, Geist A, Gupta R, Johnson F, Krishnamoorthy S, Leyffer S, Liberty D, Mitra S, Munson TS, Schreiber R, Stearley J and Hensbergen EV (2014) Addressing failures in exascale computing. *International Journal of High Performance Computing* 28(2): 129–173.

Streitz FH, Glosli JN, Patel MV, Chan B, Yates RK and de Supinski BR (2005) 100 + TFlop solidification simulations on BlueGene/L. In: *SC '05*.

Streitz FH, Glosli JN, Patel MV, Chan B, Yates RK, de Supinski BR, Sexton J and Gunnels JA (2006) Simulating solidification in metals at high pressure: The drive to petascale computing. *Journal of Physics: Conference Series* 46(1): 254.

Sutton TM, Donovan TJ, Trumbull TH, Dobreff PS, Caro E, Griesheimer DP, Tyburski LJ, Carpenter DC and Joo H (2007) The MC21 Monte Carlo transport code. In: *Joint International Topical Meeting on Mathematics and Computation and Supercomputing in Nuclear Applications*.

Young JW (1974) A first order approximation to the optimum checkpoint interval. *Communications of the ACM* 17(9).

Zheng Z, Yu L, Tang W, Lan Z, Gupta R, Desai N, Coghlan S and Buettner D (2011) Co-analysis of RAS log and job log on Blue Gene/P. In: *Proceedings of IPDPS*.

Author biographies

A Chien is the William Eckhardt Professor in Computer Science at the University of Chicago. He is also a Senior Fellow at UC's Computation Institute and a Senior Computer Scientist at Argonne National Laboratory. His research interests include parallel computing, computer architecture, and cloud computing. From 2005 to 2010, Chien was Vice President of Research at Intel Corporation where he launched new initiatives in parallel software, mobile computing,

cloud computing, and exascale research. From 1998 to 2005, He was the SAIC Endowed Chair Professor in the Department of Computer Science and Engineering where he founded the Center for Networked Systems at the University of California San Diego. From 1990 to 1998, he was a Professor of Computer Science at the University of Illinois at Urbana-Champaign and the National Center for Supercomputing Applications (NCSA). He has served on numerous advisory committees for the National Science Foundation, Department of Energy, and universities such as Stanford, EPFL, and Cal-Berkeley. He earned BS, MS, and PhD degrees at the Massachusetts Institute of Technology, and is a Fellow of the ACM, IEEE, and AAAS.

P Balaji holds appointments as a Computer Scientist and Group Lead at the Argonne National Laboratory, as an Institute Fellow of the Northwestern-Argonne Institute of Science and Engineering at Northwestern University, and as a Research Fellow of the Computation Institute at the University of Chicago. He leads the Programming Models and Runtime Systems group at Argonne. His research interests include parallel programming models and runtime systems for communication and I/O on extreme-scale supercomputing systems, modern system architecture, cloud computing systems, data-intensive computing, and big-data sciences. He has more than 130 publications in these areas and has delivered nearly 150 talks and tutorials at various conferences and research institutes. His work has been cited nearly 2000 times in the literature. He is a recipient of several awards including the US Department of Energy Early Career award in 2012, TEDxMidwest Emerging Leader award in 2013, Crain's Chicago 40 under 40 award in 2012, Los Alamos National Laboratory Director's Technical Achievement award in 2005, Ohio State University Outstanding Researcher award in 2005, six best paper awards, one best paper finalist, and one best poster finalist. He has served as a chair or editor for nearly 50 journals, conferences and workshops, and as a technical program committee member in numerous conferences and workshops. He is a senior member of the IEEE and a professional member of the ACM.

N Dun is a postdoctoral researcher working jointly at the Large-Scale Systems Group, University of Chicago and Argonne National Laboratory. His current research focuses on developing resilient models and techniques that enable reliable large-scale scientific applications. He received his PhD and MS in computer science from the University of Tokyo, and BS in computer science from Peking University. His research interests include parallel and distributed systems, system software, and large-scale applications.

A Fang is a third year PhD student working in GVR group at Computer Science Department of University of Chicago. Her current research focuses on resilience and high performance computing areas. Her PhD advisor is Andrew A Chien. She received her BS in computer science from Beijing Institute of Technology, Beijing, China in 2012.

H Fujita is a postdoctoral scholar at Large-scale Systems Group, University of Chicago. He is also a joint staff at Argonne National Laboratory. He is currently working with Professor Andrew A Chien in the GVR project. He received his PhD in 2012, MS in 2008, and BS in 2006, from The University of Tokyo, Japan, respectively. His research interest includes large-scale systems, systems software, and computer networks.

K Iskra received his MS in computer science from AGH University of Science and Technology in Cracow, Poland in 1999. In 1999–2000 he was a scientific programmer at the University of Amsterdam, Netherlands, where he worked on task migration for parallel applications. He got his PhD in computer science from the University of Amsterdam in 2005, in the area of parallel discrete event simulation. He has been working at the Argonne National Laboratory in the US since 2005, first as a postdoctoral researcher, then as an assistant computer scientist, and since 2013 as a computer scientist. He works on operating systems and I/O forwarding for massively parallel machines. He has worked on projects Argo, GVR, NoLoSS, IO-VIS, IOFSL, and ZeptoOS.

Z Rubenstein received a BA in Computational and Applied Mathematics at Rice University in 2011 and a MS in Computer Science at University of Chicago in 2014. He currently works as a software engineer at the Computational Institute at University of Chicago.

Z Zheng obtained the BS and MS degrees from the University of Electronic Science and Technology of China in 2003 and 2006, respectively, and the PhD degree in computer science from Illinois Institute of Technology in 2012. He was a postdoctoral scholar at the University of Chicago in 2013. He is currently a software engineer in HP Vertica. His research focuses on fault tolerance in large-scale computer systems. He is a member of IEEE computer society.

J Hammond is a research scientist in the Parallel Computing Lab at Intel Labs. His research interests include: one-sided and global view programming models, load-balancing for irregular algorithms, and shared- and distributed-memory tensor contractions. He has a long-standing interest in enabling the

simulation of physical phenomena, primarily the behavior of molecules and materials at atomistic resolution, with massively parallel computing. Previously, he was a Assistant Computational Scientist and Director's Postdoctoral Fellow at Argonne National Laboratory. He received a PhD in chemistry from the University of Chicago and undergraduate degrees in chemistry and mathematics from the University of Washington.

I Laguna is a computer scientist at the Center for Applied Scientific Computing (CASC) at the Lawrence Livermore National Laboratory. He received the PhD degree in the school of electrical and computer engineering from Purdue University, West Lafayette, Indiana, in 2012. He received the ACM and IEEE-CS George Michael Memorial Fellowship in 2011 for his work on large-scale failure diagnosis techniques. His research interests include software reliability, fault tolerance, and debugging in high-performance computing.

D Richards is a computational physicist in the Physical and Life Sciences Directorate at Lawrence Livermore National Laboratory. He received a BS in Physics from Harvey Mudd College in 1992 and a PhD in Physics from the University of Illinois at Urbana-Champaign in 1999. He has over 15 years of experience in scientific computing as both a user and application developer in academic, industrial, and national lab settings. He is the applications lead for the ExMatEx Co-Design center and also leads a team that is working with scientists at IBM to develop advanced cardiac modeling techniques. He was a recipient of the IEEE/ACM Gordon Bell Award in 2007 and an R&D 100 award in 2013. His research interests include large-scale parallel scientific computing and atomic scale simulation of materials.

A Dubey is a member of the Applied Numerical Algorithms Group at Lawrence Berkeley National Laboratory (LBNL). Before joining LBNL she was the Associate Director of the Flash Center for Computational Science at the University of Chicago. She received her PhD in Computer Science (1993) from Old Dominion University and BTech in Electrical Engineering Indian Institute of Technology Delhi (1985). Her research interests are in parallel algorithms, computer architecture, and software engineering applicable to high-performance scientific computing.

B van Straalen is a member of the Applied Numerical Algorithms Group at Lawrence Berkeley National Laboratory (LBNL) since 1998. Before working at LBNL he worked at Beam Technologies in NY, and Bell Northern Research in Ottawa, Canada. He received his MMath in Applied Mathematics and BASc in Mechanical Engineering from University of Waterloo, Canada in 1995 and 1993 respectively. His research focus is HPC techniques for scientific computing.

M Hoemmen finished his PhD in computer science at the University of California Berkeley in 2010. He specializes in the boundary between numerical algorithms and computer architectures. His latest research areas are algorithmic fault tolerance and very fine-grained parallel programming models. He is also a Trilinos (trilinos.org) developer.

M Heroux is a distinguished member of the Technical Staff at Sandia National Laboratories and Scientist in Residence at St. John's University, MN, working on new algorithm development, and robust parallel implementation of solver components for problems of interest to Sandia and the broader scientific and engineering community. He leads development of the Trilinos Project, an effort to provide state-of-the-art solution methods in a state-of-the-art software framework. He works on the development of scalable parallel scientific and engineering applications and maintains his interest in the interaction of scientific/engineering applications and high-performance computer architectures. He leads the Mantevo project, which is focused on the development of open source, portable mini-applications and mini-drivers for scientific and engineering applications. He is also the lead developer and architect of the HPCG benchmark, intended as an alternative ranking for the TOP 500 computer systems. He is a member of the Society for Industrial and Applied Mathematics (SIAM) and past chair of the SIAM Activity Group on Supercomputing. He is a Distinguished Member of the Association for Computing Machinery (ACM). He is the Editor-in-Chief for the *ACM Transactions on Mathematical Software*, Subject Area Editor for the *Journal on Parallel and Distributed Computing* and Associate Editor for the *SIAM Journal on Scientific Computing*.

K Teranishi is a principal staff member of Scalable Modeling and Analysis Systems at Sandia National Laboratories at California. His has broad interests in HPC research, including application resilience, programming models and numerical linear algebra. He holds MS degree from University of Tennessee and PhD from Pennsylvania State University.

A Siegel is a scientist with joint appointments in the Mathematics and Computer Science and Nuclear Engineering Divisions of Argonne National Laboratory, where he currently serves as the Director of the Center for Exascale Simulation of Advanced Reactors (CESAR). His research focuses on the intersection between advanced computational methods for

particle transport and next generation computer architectures, with particular recent focus on stochastic approaches. Previously, he served as the founder and lead of the SHARP (Simulation-based High-Fidelity Advanced Reactor Prototyping) group, the National Technical Director of the Nuclear Energy Advanced Modeling and Simulation Program, the Deputy Director for the Fusion Simulation Program, lead of the Petaflops Application Group, and Chief Architect of the Flash code. He is also an Adjunct Professor of Computer Science at the University of Chicago, where he has helped design a curriculum and taught over fifty courses in parallel computing, numerical methods, and software design.