Versioned Distributed Arrays for Resilience in Scientific Applications: Global View Resilience

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

[1] University of Chicago [2] Argonne National Lab [3] Lawrence Livermore National Lab
[4] HP [5] Intel Corp. [6] Lawrence Berkeley National Lab [7] Sandia National Labs

Abstract
Exascale studies project reliability challenges for future high-performance computing (HPC) systems. We propose the Global View Resilience (GVR) system, a library that enables applications to add resilience in a portable, application-controlled fashion using versioned distributed arrays. We describe GVR's interfaces to distributed arrays, versioning, and cross-layer error recovery. Using several large applications (OpenMC, the preconditioned conjugate gradient solver PCG, ddcMD, and Chombo), we evaluate the programmer effort to add resilience. The required changes are small (<2% LOC), localized, and machine-independent, requiring no software architecture changes. We also measure the overhead of adding GVR versioning and show that generally overheads <2% are achieved. We conclude that GVR's interfaces and implementation are flexible and portable and create a gentle-slope path to tolerate growing error rates in future systems.

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

1 Introduction

One of the most daunting challenges for high-performance computing as we approach extreme scale is growing hardware and software error rates. Already a serious concern in today's supercomputers at 100,000 cores, systemwide mean time between failures (MTBF) can be as short as a few hours [1, 2, 3]. Future exascale systems, with a billion threads, are projected to have mean time to interrupt (MTTI) [4, 5, 6, 7] as low as 10 to 30 minutes. The most successful applications on such systems will benefit from application resilience.

Our approach, called Global View Resilience (GVR), uses versioned distributed arrays to enable computational scientists to build portable, resilient applications. Beyond process/node crashes, GVR also enables resilience to more difficult *latent* or *silent* errors [8].

Key features of GVR include the following:

- 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.
- Unified error signaling and handling, customized per GVR distributed array, that enable algorithm-based fault-tolerance (ABFT) [9] error-checking and recovery.

We have applied the GVR approach to several large applications (PCG solver, OpenMC, ddcMD, and Chombo). Based on this experience, we evaluate the programmer effort required (code changes) to adopt version-based resilience and its performance impact.

Specific contributions of the paper include the following:

- Description of the GVR version-based resilience model, including the API for multi-stream, versioned distributed arrays and flexible cross-layer error signaling and recovery.
- Study of GVR with application proxies and large applications (PCG/Trilinos, OpenMC, ddcMD, and Chombo) showing that only modest code changes (<2% LOC) are required in order to achieve application-controlled, portable, version-based resilience.
- Demonstration of a variety of approaches to application-level resilience by using a version-based distributed array model. These include recovery from immediately detected errors, latent or silent errors (where detection is delayed), and forward-error recovery techniques. The latter two particularly benefit from GVR's multi-version and multi-stream capability that enables a broad range of novel recovery techniques.
- Performance studies across the same applications documenting that the cost of running with versioning code generally results in <2% runtime overhead, at versioning frequencies much higher than needed for today's error rates.

In short, our results show that GVR's version-based resilience using distributed arrays is a portable, gentle-slope resilience approach. It enables flexible error reporting and recovery and thus is promising for scaling to the higher error rates expected in extreme-scale hardware.

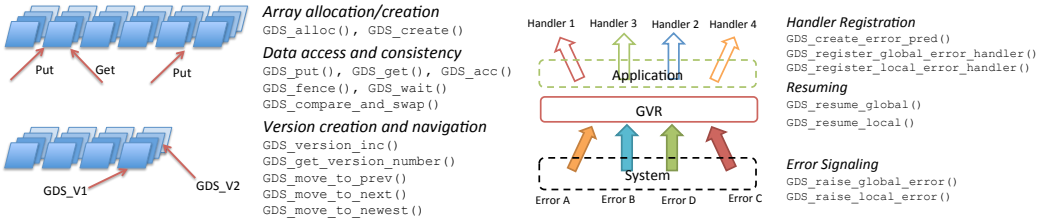
2 Global View Resilience Model and APIs

The Global View Resilience 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: 1) basic data access, update, and version creation, and 2) error signaling and handling.

GVR Global-Array Interface. GVR distributed arrays each have a global name but are distributed across multiple nodes [10, 11, 12]. The global name supports flexible programming of irregular applications and, in the context of resilience, eases recovery programming when the number of physical resources has changed. In addition to distributed array creation, GVR supports block-based *put/get* operations on multidimensional arrays, synchronization operations (*wait/fence*), and accumulate operations (*acc/get_acc*). Beyond these traditional operations, GVR adds novel operations to create and label versions with *version_inc*. All of these APIs are illustrated in Figure 1a.

GVR Error-Handling Interface. GVR includes the *Open Resilience* (OR) interface, designed to support flexible application and cross-layer handling (e.g., [9, 13, 14]). Open resilience supports a wide variety of error types, including process crash, node failure, memory error, network error, and application-detected error, and is extensible to more as they arise. The OR interface allows applications to define error-checking and recovery routines, exploiting both application and systems semantics for efficiency and robust recovery.

A key element of GVR's Open Resilience interface is unified error signaling, where programs define errors, handlers, and predicates that govern matching (see Figure 1b). A unified



(a) Data Access and Versioning APIs

(b) Error Signaling and Handling APIs

Figure 1: GVR's Application Programming Interface

approach both maximizes the opportunity for applications to recover from errors and supports flexibility and composability in error handling. Note that the error handling is functionally unified but need not be centralized in implementation. OR supports an extensible set of error descriptors, each composed of a set of error attributes (key-value pairs), describing the error, such as "corrupted memory address=0xffe0" or "failed process ranks={24, 25}". When an error occurs, the signaler creates an appropriate error descriptor and invokes the GVR signaling library. The GVR library's open resilience system then filters the descriptor with error predicates to determine the most appropriate handler.

2.1 Using GVR to Introduce Versioning and Flexible Recovery

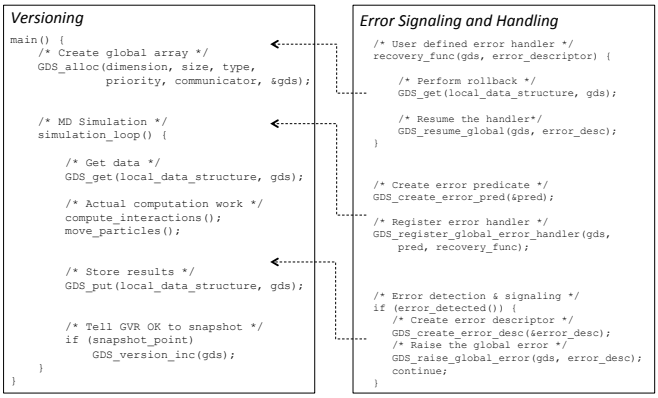


Figure 2: Example Program. First a distributed array is defined, and then snapshots (versions) are created. These persistent versions enable a wide range of application resilience techniques

Applications using GVR can take a snapshot of an individual distributed array at a time of their choosing, which we call multi-stream versioning, to match redundancy to application structure as needed. Applications can select the timing of versions to minimize synchronization cost and maximize recovery value (keeping as small a state as possible for maximum coverage). Versioning is a convenient idiom because 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. For example, in Figure 2, the MD simulation used *get* operations to pull data from a GVR array for computation and then put the data to the GVR array each time step. For versioning, a single call to *version_inc* is added.

Each call to *version_inc* creates a version and increments the current version number.[1] Calls

[1]The increment can be specified by the user, and if desired a version label can be applied.

to *version_inc* control the rate and timing of versioning for a given array. For example, a read-only object needs to be preserved only once, an object that is easy to calculate may not need to be preserved at all, and objects that consume less memory can be efficiently preserved at a greater frequency than objects that consume more memory.

GDS_move_to_prev and *GDS_move_to_next* update an array handle to point to a previous or next version, respectively. For recovery, GVR provides convenient primitives to name and navigate multiple versions. This approach differs from checkpoint/restart systems, where checkpoints have no application names, nor can one manipulate parts of several checkpoints simultaneously to aid complex recovery. As shown on the right of Figure 2, with GVR the application programmer typically defines an error recovery function to handle a class of errors and register it. 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 that detects an error. GVR currently handles two types of errors, local and global.

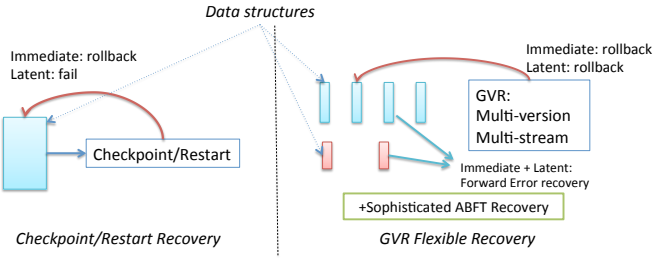


Figure 3: GVR Flexible Recovery

Figure 3 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. First, a GVR program can conveniently version different data structures at different rates, in several different streams. Second, when an error occurs, recovery can happen from the most recent version or from older versions, enabling recovery from latent or silent errors. Third, instead of a rollback recovery, a GVR program can choose to do a forward error recovery by computing an approximation from the current application data and any collection of versions. Fourth, at the point of error recovery, the application can do any general computation across all the versions of all the data structures, as well as the current computation state to both diagnose the error and decide how to proceed. This computation is not without cost, but GVR provides efficient implementation of versions, including efficient "partial materialization" to support the most flexible approaches. 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 these capabilities can be programmed—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 little small programming effort and incurs almost negligible runtime overhead.

3.1 Applications

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

- OpenMC [15] is a production Monte Carlo neutron transport code, capable of simulating 3D models based on constructive solid geometry with second-order surfaces. OpenMC is used by the DOE CESAR co-design center to explore scalable nuclear reactor modeling.
- Preconditioned Conjugate Gradient (PCG) [16] is an efficient and widely used method to iteratively solve the linear system $Ax = b$. In addition, it is the simplest of the class of Krylov subspace solvers that solve linear systems by moving the approximate answer in one dimension of Krylov subspace at a time.
- ddcMD [17, 18] 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 [17, 19] and was used for fault tolerance research when Blue Gene/L was discovered to have fatal L1 cache errors.
- Chombo [20] is a library that implements a block-structured adaptive mesh refinement (SAMR) technique [21, 22] to efficiently achieve higher resolution in regions of interest. Chombo defines patches of uniform resolution and embeds them within other patches of lower resolution. 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 as a physical entity (union of all patches at the same resolution).

3.2 How Applications Use GVR

GVR is used in two major ways in existing applications. One is to directly introduce a distributed array as the primary application data structure. OpenMC is an example of this. The other is to keep the existing data structures, using a distributed array as a copy of recovery data. GVR in ddcMD, PCG, and Chombo periodically dump critical data to a recovery data array creating versions.

OpenMC: Among several in-memory data structures in OpenMC, we introduced a distributed array to represent the tally data. In the original version of OpenMC, tally data was represented as a local buffer, thus limiting the scalability of the simulation because each process had to keep the entire tally data in its memory. By introducing the distributed array using GVR, OpenMC can take advantages of globally shared data and gain scalability [23]. 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 the 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 can reach terabytes. Tally is a write (put)-only data structure; read (get) never happens during simulation. During the simulation, OpenMC simulates a batch, or set, of particles. When it completes simulation of one batch, it creates a version and then proceeds to next batch.

PCG: This solver can be made resilient by periodically taking snapshots of critical variables. It restores these variables if algorithm-specific invariants [13] fail. Our PCG implementation uses Trilinos [24], a C++ library that provides scalable primitives for linear algebra operations, linear and nonlinear solvers, and other useful scientific computing algorithms. Trilinos was augmented with GVR by adding versioning to Tpetra (fundamental class for vectors and matrices) [25]. Hence, this general investment could be exploited by many Trilinos applications.

ddcMD: The original ddcMD had an in-memory snapshot functionality to tolerate uncorrectable (yet detectable) L1 parity errors on the BG/L platform [17]. When applying GVR

we exploited part of this infrastructure, namely, the state preservation functionality and the recovery handler. The original error handler is transformed into a GVR-style error handler with small code changes so that it utilizes the unified error signaling interface of GVR. By doing this, we generalize the original error handler, which was designed only for L1 cache errors, to many other kinds of errors such as main memory errors and application-detected data corruptions.

Chombo: For Chombo we introduced one array per level, plus one additional array for storing global metadata information. We exploited part of the existing HDF5-based checkpoint/restart functionality in Chombo for state preservation and restoration.

Table 1: Application Code Changes for GVR Resilience

Application	% Changed	Application Lines of Code	Leverage Global View	Change Software Architecture
OpenMC	<2%	30 K	Yes	No
PCG/Trilinos	<1%	300 K	Yes	No
ddcMD	<0.3%	110 K	Yes	No
Chombo	<1%	500 K	Yes	No

Table 1 demonstrates that introducing GVR requires very small code changes to existing applications, typically less than 1%. Even for OpenMC, which most intensively incorporates the global view model in the code, the code changes were still below 2%. Not only OpenMC but also other applications leverage global view for state preservation and recovery. Also, these changes are localized to a small part of the program and are machine-independent (array manipulation on the GVR interface). Moreover, these applications did not require architectural changes, thus greatly reducing the amount of work when introducing GVR resilience. From these results we conclude that introducing GVR is not intrusive and thus enables gentle migration to higher resilience for existing 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 the results 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.[2] Next we run applications linked with GVR library but without any GVR operations (no puts, gets, or version_inc's). This measurement isolates the target server thread overhead, which is used by the GVR library to implement collective operations such as allocation. We then 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, 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) corresponds to predicted MTTIs on extreme resilience scenarios for future exascale systems. Versions are captured in local memory. 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:* We use the PWR Performance Benchmark [26] with a total of 8M tally bins.

[2]Note that the native version of OpenMC did not scale beyond 64 nodes for the data set we use, because of a design limitation of the tally data structure. So instead we compare it with the GVR base version for OpenMC.

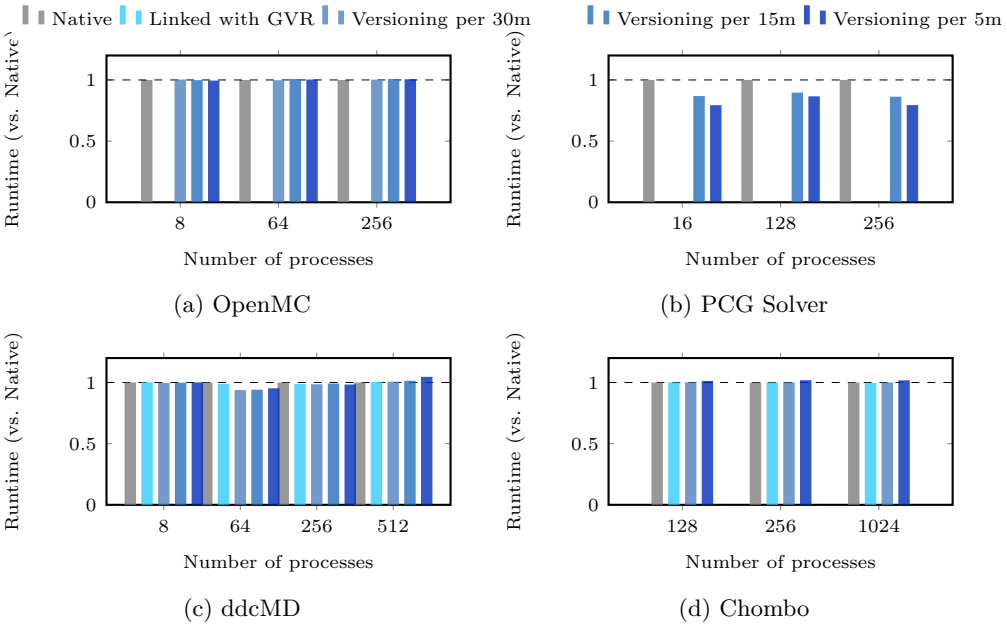


Figure 4: Summary of Versioning Performance Impact Adding Resilience Using GVR

- *PCG:* We use a sparse matrix generated according to HPCG benchmark [27] of $320K$x$320K$ with 26 non-zero values per row. Each of the 16 processes has a $320K$x$20K$ sub-matrix.
- *ddcMD:* The simulation is a system of 2,000 (Ta) atoms per process. Atom interaction is modeled with EAM potentials. The simulation runs for 500,000 time steps.
- *Chombo:* A 3D gas-dynamics simulation, reflecting a shock-wave along a ramp [28]. A purely explicit hyperbolic solve with space and time refinement with an AMR hierarchy with four levels and refinement ratio of 2. Refinement in time implies that for each time step dt taken by a coarse level, the next finer level takes n timesteps of size dt/n (where n is the refinement ratio). The coarsest level has 128x32x16, 128x32x32, and 128x32x128 cells for 128, 256, and 1024-process runs, respectively.

3.3.1 Hardware Platforms

GVR is a portable system, as are many of the applications. We therefore did experiments on the Mira Blue Gene/Q system at Argonne [29], the Edison Cray XC30 system at NERSC [30], and the UChicago Midway Linux cluster [31]. 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 (5,576 nodes, dual 12-core Intel IvyBridge 2.4 GHz, 64 GB) connected by Intel Aries with Dragonfly topology.

3.3.2 Results and Discussion

Figure 4 shows the runtime 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 the PCG solver, which we conjecture may result from the unstable network of the cluster.

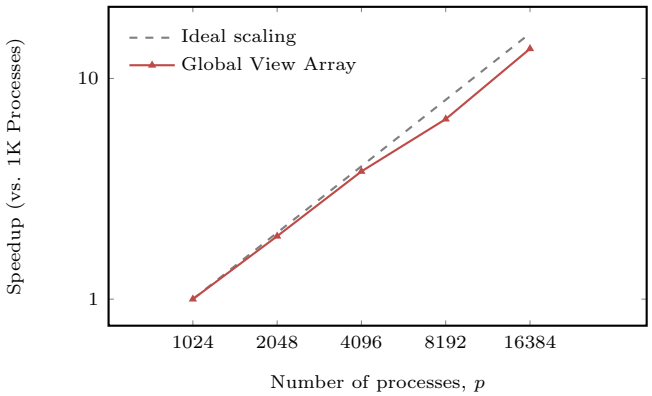


Figure 5: Scalability of OpenMC Using GVR

Figure 5 shows the weak-scaling results up to 16K processes of GVR-enabled OpenMC with a 2.4 TB array size on Edison Cray XC30 system, where it achieves 85% efficiency for 16K processes compared with the run on 1K processes. 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 Discussion and Related Work

GVR's global arrays trace their heritage to ideas found in PGAS-style libraries and languages such as Global Arrays (GA) [11], UPC [32], Co-Array Fortran [10], X10 [33], and Chapel [34]. The GVR APIs follow the well-known interfaces implemented by the Global Arrays library [11]. The key innovations in GVR include the ability to create and name multiple versions of a distributed array, as well as support for flexible labeling and navigation among versions. Further, GVR attaches error handling to distributed arrays.

While multi-version data interfaces are a relatively new concept in high-performance fault resilience, other systems have provided this capability. Examples include database views (sometimes called snapshot views), timestamped values for a key in Google's BigTable [35] and Apple's Time machine [36], and a variety of research snapshot extensions of Linux filesystems [37, 38].

Several reports have advocated cross-layer resilience [39] both to increase the number of recoverable errors and to make recovery efficient. However, existing examples of cross-layer error recovery [14, 17] are "stovepipes," with a one-to-one mapping of error handler and error event. GVR's Open Resilience approach, on the other hand, seeks to provide more flexible matching between error handlers and error events and, by doing so, to increase the return on investment for both error signalers and handlers. An early unified error signaling interface can be found in CIFTS [40], but that work is currently inactive.

Other fault tolerance techniques exist, with checkpoint/restart (CR) being the most popular one. CR suffers, however, from obvious scalability problems on accessing stable storage from multiple nodes on large systems, although efforts are under way to alleviate this problem by utilizing on-node storage; see, for example, SCR [7] and FTI [41]. With access only to the latest checkpoint, however, CR has no way of dealing with latent errors. Checkpoint/restart also provides no means of extensible error checking or flexible recovery.

Other reliability techniques such as replicated execution [42, 43] often incur high overhead on failure-free execution. GVR enables flexible balance of error coverage and overhead.

Application-specific techniques such as approximate execution [44, 45, 46] and fault-tolerant algorithms (ABFT) [47, 48, 49] can enable efficient error detection and recovery. Such techniques would be natural to implement portably atop GVR.

5 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 structures and tuned to increasing error rates. The programming experience with GVR shows that it can be added to large applications with little effort (<2% code changes), and adding GVR versioning is low cost (<2% runtime) at today's error rates. Future directions include incorporating GVR into high-level programming models and tools, such as these being developed in the X-stack program [50], as well as techniques to further optimize version implementation, including efficient differences, compression, and exploitation of NVRAM.

Acknowledgements

This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. 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 U.S. 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.

References

[1] B. Schroeder et al. A large-scale study of failures in high-performance computing systems. In *DSN*, 2006.

[2] Z. Zheng et al. Co-analysis of RAS log and job log on Blue Gene/P. In *Proceedings of IPDPS*, 2011.

[3] C. Di Martino et al. Lessons learned from the analysis of system failures at petascale: The case of blue waters. In *DSN 2014*, pages 610–621, June 2014.

[4] K. Ferreira et al. Evaluating the viability of process replication reliability for exascale systems. In *SC '11*.

[5] K. Bergman et al. Exascale computing study: Technology challenges in achieving exascale systems. *DARPA IPTO Tech. Rep*, 2008.

[6] S. Borkar et al. The future of microprocessors. *CACM*, 54:67–77, May 2011.

[7] A. Moody et al. Design, modeling, and evaluation of a scalable multi-level checkpointing system. SC '10.

[8] G. Lu et al. When is multi-version checkpointing needed? In *FTXS '13*, New York, NY, USA, 2013. ACM.

[9] K.-H. Huang et al. Algorithm-based fault tolerance for matrix operations. *IEEE Trans. Comput.*, 33(6):518–528, June 1984.

[10] R. W. Numrich et al. Co-Array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, August 1998.

[11] J. Nieplocha et al. Advances, applications and performance of the Global Arrays shared memory programming toolkit. *IJHPCA*, 20(2):203–231, 2006.

[12] R. Bariuso et al. SHMEM user's guide. Cray Research, Inc., 1994.

[13] Z. Chen. Online-ABFT: an online algorithm based fault tolerance scheme for soft error detection in iterative methods. PPoPP '13, pages 167–176, 2013.

[14] P. G. Bridges et al. Cooperative application/os dram fault recovery. In *Resilience'11*, pages 241–250, 2012.

[15] P. K. Romano et al. The OpenMC Monte Carlo particle transport code. *Ann. Nucl. Energy*, 51, 2013.

[16] G. H. Golub et al. *Matrix Computations (3rd ed.)*. Johns Hopkins University Press, Baltimore, MD, 1996.

[17] J. N. Glosli. et al. Extending stability beyond cpu millennium: a micron-scale atomistic simulation of kelvin-helmholtz instability. In *Proceedings of SC '07*, pages 1–11, Nov. 2007.

[18] F. H. Streitz et al. Simulating solidification in metals at high pressure: The drive to petascale computing. *Journal of Physics: Conference Series*, 46(1):254, 2006.

[19] F. H. Streitz et al. 100+ TFlop solidification simulations on BlueGene/L. In *SC '05*, Nov. 2005.

[20] P. Colella et al. Chombo software package for AMR applications design document. Technical report, LBNL, Applied Numerical Algorithms Group, Computational Research Division, 2009.

[21] M. Berger et al. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53(3):484–512, 1984.

[22] M. Berger et al. Local adaptive mesh refinement for shock hydrodynamics. *Journal of Computational Physics*, 82(1):64–84, 1989.

[23] N. Dun et al. Data decomposition in Monte Carlo particle transport simulations using global view arrays. Technical Report TR-2014-09, Department of Computer Science, University of Chicago, May 2014.

[24] M. A. Heroux et al. An overview of the trilinos project. *ACM Transactions on Mathematical Software (TOMS)*, 31(3):397–423, 2005.

[25] Z. Rubenstein et al. Error checking and snapshot-based recovery in a preconditioned conjugate gradient solver. Technical Report TR-2013-11, Department of Computer Science, University of Chicago, Nov. 2013.

[26] J. E. Hoogenboom et al. The Monte Carlo performance benchmark test - aims, specifications and first results. In *ANS M&C*, 2011.

[27] M. A. Heroux et al. HPCG technical specification. Technical report, Sandia National Laboratories, 2013.

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

[29] Argonne Mira. `http://www.alcf.anl.gov/mira`.

[30] NERSC Edison. `http://www.nersc.gov/users/computational-systems/edison/`.

[31] UChicago RCC Midway. `https://rcc.uchicago.edu/`.

[32] W. Carlson et al. Introduction to UPC and language specification. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, 1999.

[33] P. Charles et al. X10: An object-oriented approach to non-uniform cluster computing. OOPSLA '05, pages 519–538, 2005.

[34] B. L. Chamberlain et al. Parallel programmability and the Chapel language. *IJHPCA*, 21(3), 2007.

[35] F. Chang et al. Bigtable: A distributed storage system for structured data. OSDI '06, page 15, 2006.

[36] Apple Inc. Time machine built-in backup. http://www.apple.com/osx/apps/#timemachine.

[37] R. Konishi et al. The Linux implementation of a log-structured file system. *SIGOPS Oper. Syst. Rev.*, 40(3):102–107, July 2006.

[38] O. Rodeh et al. BTRFS: The Linux B-tree filesystem. *Trans. Storage*, 9(3):9:1–9:32, August 2013.

[39] A. DeHon et al. Final report for CCC cross-layer reliability visioning study. `http://www.cra.org/ccc/xlayer.php`, 2011.

[40] R. Gupta et al. CIFTS: A coordinated infrastructure for fault-tolerant systems. In *Proceedings of ICPP '09*, pages 237–245, 2009.

[41] L. Bautista-Gomez et al. FTI: High performance fault tolerance interface for hybrid systems. In *SC '11*.

[42] R. E. Lyons et al. The use of triple-modular redundancy to improve computer reliability. *IBM J. Res. Dev.*, 6(2):200–209, April 1962.

[43] J. Lidman et al. ROSE::FTTransform—a source-to-source translation framework for exascale fault-tolerance research. In *FTXS'12*, 2012.

[44] A. Sampson et al. EnerJ: Approximate data types for safe and general low-power computation. PLDI '11.

[45] M. Carbin et al. Verifying quantitative reliability for programs that execute on unreliable hardware. OOPSLA '13, pages 33–52, 2013.

[46] S. Misailovic et al. Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels. OOPSLA '14, pages 309–328, 2014.

[47] G. Bronevetsky et al. Soft error vulnerability of iterative linear algebra methods. In *ICS*, 2008.

[48] S. K. S. Hari et al. Low-cost program-level detectors for reducing silent data corruptions. In *IPDPS*, 2012.

[49] M. A. Heroux. Toward resilient algorithms and applications. *CoRR*, abs/1402.3809, 2014.

[50] U.S. Department of Energy. Extreme scale software stack. `https://xstackwiki.modelado.org/Extreme_Scale_Software_Stack`.