MCEM: Multi-Level Cooperative Exception Model for HPC Workflows

Stephen Herbein
herbein1@llnl.gov
Lawrence Livermore National Laboratory
Livermore, CA

David Domyancic
domyancic1@llnl.gov
Lawrence Livermore National Laboratory
Livermore, CA

Paul Minner
minner2@llnl.gov
Lawrence Livermore National Laboratory
Livermore, CA

Ignacio Laguna
ilaguna@llnl.gov
Lawrence Livermore National Laboratory
Livermore, CA

Rafael Ferreira da Silva
rafsilva@isi.edu
Information Sciences Institute, University of Southern California
Marina del Rey, CA

Dong H. Ahn
ahn1@llnl.gov
Lawrence Livermore National Laboratory
Livermore, CA

ABSTRACT

As fault recovery mechanisms become increasingly important in HPC systems, the need for a new recovery model for workflows on these systems grows as well. While the traditional approach in which each system component attempts its own independent recovery after a fault works well at each individual application level, this model does not scale to the new level demanded by workflow-level exception handling. As today's workflows must often run many components simultaneously (e.g., workflow manager components, many simulation instances, data analytics etc), any uncoordinated model can quickly result in redundant or contradictory recovery actions. In this paper, we propose a multi-level cooperative exception model (MCEM), a novel exception handling approach that solves this coordination challenge for HPC workflows. We present our model, describe how it can be applied to common system faults and other workflow specific exceptions, and demonstrate how it reduces redundant I/O in the case of a file-system quota exception.

CCS CONCEPTS

• **Software and its engineering → Distributed systems organizing principles**; *Cooperating communicating processes.*

KEYWORDS

Workflow Exception Model; Coordinated Recovery; Fault-Tolerance

ACM Reference Format:
Stephen Herbein, David Domyancic, Paul Minner, Ignacio Laguna, Rafael Ferreira da Silva, and Dong H. Ahn. 2019. MCEM: Multi-Level Cooperative Exception Model for HPC Workflows. In *9th International Workshop on Runtime and Operating Systems for Supercomputers (ROSS'19), June 25, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3322789.3328745

1 INTRODUCTION

With the current push towards exascale systems, the HPC community anticipates a dramatic increase in the rate of fault occurences, shrinking the mean-time-between-failures (MTBF) from days to hours [1]. Existing techniques like checkpoint-restart will no longer be sufficient at these short MTBF timescales. Additional resiliency strategies will need to be deployed to ensure applications can make meaningful progress in spite of the frequent faults.

Moving beyond application-centric resiliency strategies and adding fault-tolerance throughout a system hierarchy presents challenges of its own. In particular, today's large HPC workflows employ different distributed components, and when each of them tries to recover from the same fault, redundant, or worse, contradictory recovery operations can arise. As HPC systems grow larger, with more levels of hardware and software, a more holistic fault-tolerance approach is necessary. Workflow as well as system's components on future exascale systems must cooperate to collectively detect, isolate, and recovery from faults, rather than relying on the existing uncoordinated approaches.

In this paper, we present a novel workflow exception model for HPC systems. Our model enables more efficient recovery by coordinating the recovery efforts of system components to avoid redundant and contradictory recovery actions. We present our model and its design principles, how it can be implemented on top of modern resource managers, and how it can be applied to common system faults. We evaluate how our model can be leveraged to reduce the I/O performed by up to 90% during the recovery from one of those common faults, a file system quota exception.

The paper is organized as follows. Section 2 provides background on fault-tolerance primitives and two motivating examples for our work. Section 3 presents our novel workflow exception model, MCEM. We present our evaluation and results in Section 4. Section 5 outlines related work and is followed by a summary of the work in Section 6.

ROSS'19, June 25, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6755-4/19/06…$15.00
https://doi.org/10.1145/3322789.3328745

2 BACKGROUND AND MOTIVATION

In this section, we provide background on fault-tolerance primitives (i.e., detection, isolation, and recovery) and two motivating examples that demonstrate the state of the practice of workflow exception handling in HPC. We present the state of the practice through the common-place examples of a node failure and exceeding a file system quota. We show how these exceptions are currently handled in a production LLNL workflow manager, the UQ Pipeline (UQP) [2].

2.1 Fault-Tolerance Primitives

The three fault-tolerance primitives relevant to parallel or distributed systems are detection, isolation, and recovery. Here we define each term:

Detection: the observation of a fault, error, or degradation

Isolation: the identification of the root cause of the detected fault

Recovery: the remediation of the fault by affected components

In the current state of the practice in HPC, each individual system component is responsible for independently implementing its own detection, isolation, and recovery. This lack of coordination produces suboptimal results as each component only has limited knowledge. With only limited knowledge, a failure may be attributed to the wrong root cause or missed entirely. Considerable effort has gone into improving the fault-tolerance primitives within a single component of the system (e.g., parallel runtime or resource manager) [3–6], but ideally, system components would synchronize their knowledge and coordinate their recovery efforts. Some state-of-the-art work enables system components to coordinate their fault detection and isolation [7–9]. While other work focuses on coordinating recovery between two specific system components, like the parallel application and the resource manager [10, 11]. Unfortunately, none of these works directly handles the coordination of recovery for workflows across arbitrary levels of system components.

2.2 Motivating Example

Many types of failures can occur for HPC workflows that range from system to application- or workflow-level exceptions (e.g., mesh tangling, normal simulation completion whose results do not add to the overall workflow objectives, etc). We focus on two common failures, compute node failure and the exhaustion of disk quota, as motivating examples. The node failure is a local fault, while the disk quota error is a global fault. Both of these failure have huge impacts on the performance of the HPC system.

Node-level failures are becoming increasingly common on large-scale HPC systems and can happen for many reasons, including an OS crash, system reboot, or power loss. In the current state of the practice, after the node fails, several components will detect, isolate, and recover from the failure in an independent and uncoordinated way. First, the cluster's resource manager will detect the unresponsive node, most likely through a heartbeat timeout, isolate the failure to that particular node, and prevent future jobs from running on the node. Second, if the failed node was part of a parallel application, the application will detect an unresponsive rank during communication, isolate the failure to the node that the unresponsive rank is running on, and begin recovery specific to the application (e.g., restart from a checkpoint). Unfortunately, this detection method is susceptible to false positives from, for example, a slow or failed network link, resulting in unnecessary or suboptimal recovery. Third, if the node failure causes the application as a whole to crash, the workflow manager will detect the failed application but will be unable to isolate the cause to any particular node. In the UQ Pipeline specifically, if a node is involved in a failure three separate times, then the UQP estimates that the node is unreliable or failed and prevents that particular node from running any of the remaining work in the workflow.

This example highlights a current lack of coordination between system components, resulting in each component having its own mechanism to detect, isolate, and recover from failures, with some of these mechanism being more accurate and consistent than others. Ideally, when any system component detects and isolates an exception, it would notify the other components of the exception and begin a coordinated recovery. In this ideal scenario, the resource manager would detect the node failure and notify the parallel application and workflow manager. The parallel application would now be confident that the unresponsive rank was indeed due to a failed node, rather than just a slow network link, and the workflow manager would have the information necessary to isolate the problem to a single node. We re-visit this example in more detail in Section 3.4.

Another common fault on large-scale HPC systems is I/O write failures due to exceeding file system quotas. Many HPC centers impose limitations on the total disk space and the total number of files that can be owned by any one user on a given file system. This increases fairness amongst users and ensures that a single user cannot monopolize an entire file system, creating write faults for other users, but these quotas pose a challenges for workflows that generate large amounts of data or files. Users with these data intensive workflows must be careful not to exceed their quotas and to spread their data across multiple file systems when necessary.

In the current state of the practice, when a user exceeds their quota, all of their applications' writes immediately begin to fail, which from an application's perspective, is indistinguishable from a total file-system failure. At this point, the user's applications will either crash or hang until their wallclock time is exceeded and they are killed. In some cases, the application may be able to copy it's data from the primary file system that is full, over to a secondary file system that has space available, and then continue execution. Unfortunately, with enough simulations performing this action in an uncoordinated fashion, the storm of data movement to the secondary file system can create a cascading failure due to performance degradation or the exceeding of another quota.

Ideally, when a user is approaching their file-system quota, a *soft exception* would be raised to notify system components of an impending *hard exception* that will occur once the quota is exceeded. In this ideal scenario, the workflow manager would coordinate with the user's running simulations to only migrate the minimal subset of simulations necessary for every simulation to continue running without exceeding the primary file system's quota. This would not only minimize the I/O load placed on the secondary file system, but it would also eliminate the possibility of a cascading failure due to uncoordinated recovery attempts by the individual simulations. We re-visit this example in more detail in Section 4.

3 MCEM: MULTI-LEVEL COOPERATIVE EXCEPTION MODEL

In this section, we give a description of our exception model, MCEM, that enables coordinated recovery amongst all components within the system software stack. This includes our fault model as well as how we forsee MCEM being implemented in real systems.

3.1 Fault Model

We consider both hard faults, like a process segmentation fault or a node power off, and soft faults, like network performance degradation or a file system that is nearly full. We only consider permanent faults and temporary faults whose effects last long enough to be reliably detected, isolated, and recovered from (i.e., effects lasting on the order of minutes rather than seconds). We make no assumptions about the sources of faults other than there exists a technique for isolating the fault to the point of origin. Said another way, our model is general enough to account for faults originating from anywhere in the system, including hardware and software. Figure 1 depicts an example system with various distributed components, potential faults, and where the faults may occur within the system. We refer to exceptions that affect only a single node or job, like a segmentation fault or ECC error, as *local exceptions*, and we refer to exceptions that affect an entire system or workflow, like network performance degradation or exceeding PFS quotas, as *global exceptions*.

3.2 Coordinated Exception Recovery via MCEM

MCEM takes the idea of exceptions in languages like C++ or Java and extends them to an entire distributed system. After a software component has detected and isolated a fault, system components can begin a coordinated recovery through MCEM. MCEM starts exception recovery at a single component, which then has the option of "re-raising" the same exception or raising a new exception (i.e., creating a chained exception), which is then propagated to the next component in the system software hierarchy. Chaining exceptions is useful in scenarios where the current software component is unable to accomplish a full recovery by itself and needs help from other components in the system. For example, in the case of a process crashing due to a segmentation fault, the resource manager may partially recover by re-launching the process and then re-raising the exception so that the parallel application can complete the recovery by wiring the newly launched process into the application.

Ideally, recovery would start at the exception's point of origin, but exceptions can affect components such that they cannot participate in recovery (e.g., total node failure). We must determine which component should begin the recovery process and which direction chained exceptions should propagate. While entirely configurable, we present a possible configuration that determines the starting component and propagation direction based on the properties of the exception. This configuration is depicted in Figure 1. In the case of a *local exception*, we propose starting recovery at the lowest software component still alive/unaffected by the exception. Chained exceptions should then propagate up the system hierarchy. This is illustrated in Figure 1a. In the case of a *global exception*, we propose starting recovery at the highest software component in the system hierarchy affected by the exception and then propagating chained exceptions down the system hierarchy. This is illustrated in Figure 1b.

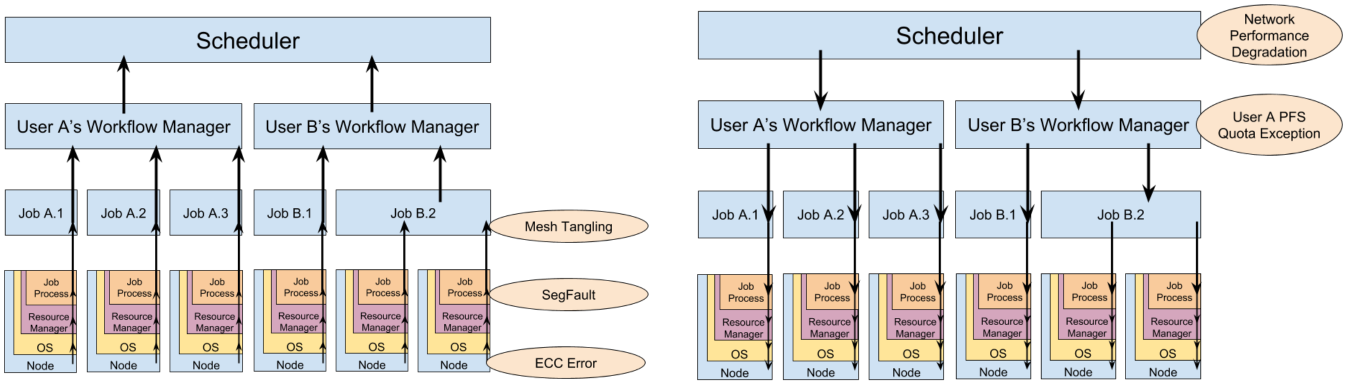
3.3 Implementation

In order to implement MCEM and the fault detection and isolation that MCEM relies on, the various components in the system hierarchy need a reliable mechanism for communication. Each of the black arrows in Figure 1 represents a point-to-point communication that can occur between components during a chained exception. The communication mechanism must support point-to-point communication between system components and should be itself tolerant of most faults that occur on the system. One option is to implement MCEM on top of a resource manager (RM) like Flux, which supports communication via RPCs as well as event publication and subscription [12]. Implemenation within or on top of an RM is natural since RMs must also be resilient to node-failure and other common exceptions. Using the communication mechanisms provided by an RM like Flux, system components could chain exceptions by sending an RPC with the exception information to their parent in the system hierarchy.

3.4 Application

In the case of a node-failure, which we detailed in Section 2.2, MCEM can coordinate the recovery amongst components so that no contradictory or redundant actions are taken. While recovery would typically begin at the exception's point of origin, in this case, the node-failure exception originates at the failed node; so the exception recovery begins at the next level up in the system hierarchy. In the configuration that we proposed in Section 3.2, the next component is the parallel job. If the job uses a parallel runtime, like Charm++ [13], or checkpointing library, like SCR [14], that automatically and independently recover from single-node failures, then the job marks the exception as handled and recovery is complete.

If the job cannot completely recover independently, it must re-raise the exception so that other system components participate in the recovery. To continue the example, the chained node failure exception propagates up to a workflow manager, which may determine that the job's preliminary results are uninteresting and kill the job, completing the recovery. Alternatively, the workflow manager may deem the job important enough to warrant re-launching the job on new nodes, completing the recovery. If the workflow manager is unable to independently complete the recovery, it must re-raise the exception, propagating it up to the system scheduler, where the process repeats. Without MCEM and a coordinated recovery, the parallel job may be restoring from a checkpoint at the same time that the workflow manager begins re-launching the job on new nodes, which could result in redundant work in the best case and data corruption in the worst. In the next section, we discuss how MCEM can coordinate the recovery of a file system quota exception.



(a) Local exceptions that are configured to propagate up the system component hierarchy, from the narrowest scope that they affect to the broadest.

(b) Global exceptions that are configured to propagate down the system component hierarchy, from the broadest scope that they affect to the narrowest.

Figure 1: Examples of chained handling of local and global exceptions in MCEM

4 EVALUATION

The need for generating a set of simulations (i.e., an ensemble of simulations) on HPC systems has become an integral component of science and engineering research. An ensemble may be composed of a few tens of simulations and can scale to the tens of thousands of simulations. An ensemble is used to study the behavior of a set of output responses as related to the perturbation of a set of input variables, sensitivity analysis, and uncertainty quantification.

A unique challenge for high performance computing and the use of ensembles is the amount of disk space required and the number of files generated. The multi-physics simulation used in this study consumes 47.34GB of disk space and approximately 24,000 files are created. Out of the 47.34GB, 47.32GB is used to store the restart dumps, each of which consumes 565MB on average and is created every 21.68sec wall clock on average. Each restart dump consists of 288 files. The disk space requirements for a simulation can vary greatly depending the complexity of the simulation such as the types of physics simulated and the number of materials in the simulation and their respective properties, requiring much more disk space. Also, the disk space requirements for supporting simulation files can occupy a greater percentage of the total disk space needs.

An evaluation of the uncoordinated approach and the MCEM approach in the event of a disk quota expiration demonstrates advantages of MCEM from the perspective of disk usage. The ensemble manager receives a notification that the user disk quota is expiring and, in response, moves the sets of simulations and the contents of each simulation to another file system to continue the generation of the ensemble. In an uncoordinated recovery, the ensemble manager and each simulation receives the notification and acts accordingly by recreating the simulation run directory on the secondary target file system and populating the simulation run directory with the required files to continue the simulation on the secondary file system from where the simulation last made its own restart checkpoint on the primary file system.

The problem with this approach is no coordination between the individual simulations and the ensemble manager occurs. The simulation will populate its own simulation run directory that is separate from the ensemble manager. The ensemble manager meanwhile will create its own directories and populate those directories with files including directories and files required for the ensemble to continue generation on the secondary file system. In effect, duplicate simulation directories and files are created, thereby increasing the load on the file system.

Using the example simulation, the size of the duplicate and unneeded data that is copied is a function of the simulation setup data, in this case 0.02GB, and the number of restart dumps, each of which is 565MB in size and is composed of 288 files. Significant time could be saved in not needing to perform duplicate data in the file transfer and the load of the file system would also be reduced. Also, if only a subset of the ensembles can be moved due to limitations on the secondary file system, then the ensemble manager will have to counteract the independent recovery actions of some of the simulations, cancelling them and restarting them back on the primary file system.

With the MCEM approach and its multi-level model, the ensemble manager receives notification of the exception first and then responds by coordinating with each simulation to stop the evolution of the simulation and transfer all of the required files to the secondary file system. If only a subset of the ensembles can be moved, then the ensemble manager will only move that exact subset over to the secondary file system, minimizing the cost of recovery. Figure 2 depicts a model of the reduction in file system I/O (i.e., metadata operations, bandwidth, and storage) when using MCEM's coordinated recovery rather than an uncoordinated recovery.

To explain the results of our model, we explore two cases in Figure 2: when 100% and 10% of jobs must be moved to a secondary filesystem. In the first case, a hard quota exception has occurred and 100% of an ensemble of jobs must be moved to a secondary filesystem in order to continue execution. In this case, under the uncoordinated approach, the ensemble manager and the simulations will both perform the transfer simultaneously, resulting

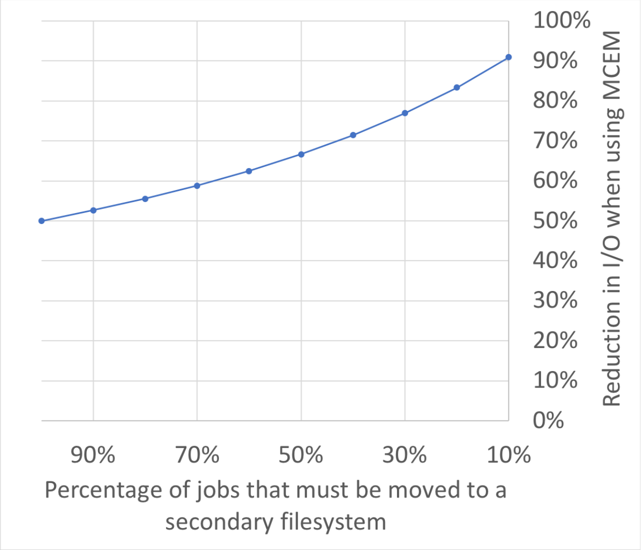


Figure 2: Reduction in I/O performed during recovery when using MCEM rather than an uncoordinated approach

in double the I/O than is necessary. The MCEM model avoids this redundancy, reducing the I/O performed by 50%. In the other case, a soft quota exception has occurred and only 10% of the jobs need to be moved to the secondary filesystem to avoid exceeding the quota. Under the uncoordinated approach, every simulation will move its data to the secondary filesystem, and to avoid a cascading failure, the ensemble manager will kill the 90% of simulations that should not have migrated and restart them on the original, primary filesystem. The MCEM approach enables the ensemble manager to coordinate the migration of exactly 10% of the simulations, avoiding the back-and-forth of simulations between filesystems and reducing the total I/O performed by 90%.

5 RELATED WORK

Many workflow management systems have been developed for improving the efficiency of large-scale applications on distributed systems. They span domains such as HPC, Grid, Cloud, among others. To cope with failures, several fault-tolerance mechanisms were developed [15–19], e.g. replication, checkpointing, and also the use of ML-based approaches to identify and mitigate faulty conditions; however only few works have actually focused on exception handling.

An attempt to handle exceptions in workflow managers is presented as an extension to Kepler for enabling resilient execution of bionformatics workflows [20]. There, exception handling is performed at the collection level, subset of computing tasks, in which a collection-aware actor that catches an external application error (or other exception) may add an exception token to the collection that caused the error. This actor may then proceed to operate on the next collection. A downstream exception-catching actor can filter out collections that contain exception tokens. Exception handling in workflows was also addressed in the form of QoS violations [21]. In their approach, workflow exceptions were tackled via workflow rescheduling: once low performance was detected (defined by a threshold derived by empirical experiments), the system would reschedule the workflow to meet its QoS constraints.

In [22], an adaptive exception handling for workflow managers tackled multi-level exceptions: workflow, middleware, and resource. Exceptions were classified into patterns, which guided system decisions, inspired by modular programming language, to replace the faulty element by another one at runtime. In our proposed model, rather than isolating exception into patterns, we define chains of exceptions that can be handled at different levels of the system stack.

To the best of our knowledge this is the first work that proposes workflow exception handling in HPC platforms by coordinating the recovery efforts of system components.

6 CONCLUSIONS

In this paper, we presented a novel workflow exception data model that can enable coordinated recovery amongst the many components in an HPC system. Using an example of a file system quota exception, we showed how the MCEM model can reduce the I/O performed during recovery by 90% over an uncoordinated recovery. Future work includes implementing MCEM on top of a resource manager like Flux and comparing the recovery latency to an uncoordinated approach and state of the art approaches like CIFTS.

7 ACKNOWLEDGMENTS

This article has been authored by Lawrence Livermore National Security, LLC under Contract No. DE-AC52-07NA27344 with the U.S. Department of Energy. Accordingly, the United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this article or allow others to do so, for United States Government purposes (LLNL-CONF-771601).

REFERENCES

[1] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir, “Toward exascale resilience,” *The International Journal of High Performance Computing Applications*, vol. 23, no. 4, pp. 374–388, 2009.

[2] T. L. Dahlgren, D. Domyancic, S. Brandon, T. Gamblin, J. Gyllenhaal, R. Nimmakayala, and R. Klein, “Poster: Scaling uncertainty quantification studies to millions of jobs,” in *Proceedings of the 27th ACM/IEEE International Conference for High Performance Computing and Communications Conference (SC)*, November 2015.

[3] B. H. Park, T. J. Naughton, P. Agarwal, D. E. Bernholdt, A. Geist, and J. L. Tippens, “Realization of user level fault tolerant policy management through a holistic approach for fault correlation,” in *2011 IEEE International Symposium on Policies for Distributed Systems and Networks*, June 2011, pp. 17–24.

[4] X. Ouyang, S. Marcarelli, R. Rajachandrasekar, and D. K. Panda, “Rdma-based job migration framework for mpi over infiniband,” in *2010 IEEE International Conference on Cluster Computing*, Sep. 2010, pp. 116–125.

[5] K. Uhlemann, C. Engelmann, and S. L. Scott, “Joshua: Symmetric active/active replication for highly available hpc job and resource management,” in *2006 IEEE International Conference on Cluster Computing*, Sep. 2006, pp. 1–10.

[6] K. Ferreira, J. Stearley, J. H. Laros, III, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold, “Evaluating the Viability of Process Replication Reliability for Exascale Systems,” *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 44:1–44:12, 2011.

[7] R. Gupta, P. Beckman, B. Park, E. Lusk, P. Hargrove, A. Geist, D. Panda, A. Lumsdaine, and J. Dongarra, "Cifts: A coordinated infrastructure for fault-tolerant systems," in *International Conference on Parallel Processing*, ser. ICPP, Sep. 2009, pp. 237–245.

[8] R. Rajachandrasekar, X. Besseron, and D. K. Panda, "Monitoring and predicting hardware failures in hpc clusters with ftb-ipmi," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*, May 2012, pp. 1136–1143.

[9] S. Di, R. Gupta, M. Snir, E. Pershey, and F. Cappello, "Logaider: A tool for mining potential correlations of hpc log events," in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, May 2017, pp. 442–451.

[10] S. Chakraborty, I. Laguna, M. Emani, K. Mohror, D. K. Panda, M. Schulz, and H. Subramoni, "Ereinit: Scalable and efficient fault-tolerance for bulk-synchronous mpi applications," *Concurrency and Computation: Practice and Experience*, p. e4863, 2018.

[11] "Failure management support," https://slurm.schedmd.com/nonstop.html, SchedMD, 2014, retrieved March 29, 2019.

[12] D. H. Ahn, J. Garlick, M. Grondona, D. Lipari, B. Springmeyer, and M. Schulz, "Flux: A next-generation resource management framework for large HPC centers," in *Proceedings of the 10th International Workshop on Scheduling and Resource Management for Parallel and Distributed Systems*, September 2014.

[13] L. V. Kale and S. Krishnan, "Charm++: A portable concurrent object oriented system based on c++," in *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '93. New York, NY, USA: ACM, 1993, pp. 91–108. [Online]. Available: http://doi.acm.org/10.1145/165854.165874

[14] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11. [Online]. Available: https://doi.org/10.1109/SC.2010.18

[15] G. Kandaswamy, A. Mandal, and D. A. Reed, "Fault tolerance and recovery of scientific workflows on computational grids," in *2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID).* IEEE, 2008, pp. 777–782.

[16] M. Lackovic, D. Talia, R. Tolosana-Calasanz, J. A. Banares, and O. F. Rana, "A taxonomy for the analysis of scientific workflow faults," in *2010 13th IEEE International Conference on Computational Science and Engineering.* IEEE, 2010, pp. 398–403.

[17] D. Poola, M. A. Salehi, K. Ramamohanarao, and R. Buyya, "A taxonomy and survey of fault-tolerant workflow management systems in cloud and distributed computing environments," in *Software Architecture for Big Data and the Cloud.* Elsevier, 2017, pp. 285–320.

[18] R. Ferreira da Silva, T. Glatard, and F. Desprez, "Self-healing of workflow activity incidents on distributed computing infrastructures," *Future Generation Computer Systems*, vol. 29, no. 8, pp. 2284–2294, 2013.

[19] R. Ferreira da Silva, R. Filgueira, E. Deelman, E. Pairo-Castineira, I. M. Overton, and M. P. Atkinson, "Using simple pid-inspired controllers for online resilient resource management of distributed scientific workflows," *Future Generation Computer Systems*, 2019.

[20] T. M. McPhillips and S. Bowers, "An approach for pipelining nested collections in scientific workflows," *ACM Sigmod Record*, vol. 34, no. 3, pp. 12–17, 2005.

[21] X. Liu, J. Chen, Z. Wu, Z. Ni, D. Yuan, and Y. Yang, "Handling recoverable temporal violations in scientific workflow systems: a workflow rescheduling based strategy," in *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing.* IEEE, 2010, pp. 534–537.

[22] R. Tolosana-Calasanz, J. A. Bañares, O. F. Rana, P. Álvarez, J. Ezpeleta, and A. Hoheisel, "Adaptive exception handling for scientific workflows," *Concurrency and computation: Practice and experience*, vol. 22, no. 5, pp. 617–642, 2010.