Using Performance Tools to Support Experiments in HPC Resilience

Thomas Naughton[1,2], Swen Böhm[1], Christian Engelmann[1], and Geoffroy Vallée[1] *

[1] Computer Science and Mathematics Division
Oak Ridge National Laboratory, Oak Ridge, TN, USA.
{naughtont,bohms,engelmannc,valleegr}@ornl.gov
[2] School of Systems Engineering
The University of Reading, Reading, UK.

Abstract. The high performance computing (HPC) community is working to address fault tolerance and resilience concerns for current and future large scale computing platforms. This is driving enhancements in the programming environments, specifically research on enhancing message passing libraries to support fault tolerant computing capabilities. The community has also recognized that tools for resilience experimentation are greatly lacking. However, we argue that there are several parallels between "performance tools" and "resilience tools". As such, we believe the rich set of HPC performance-focused tools can be extended (*repurposed*) to benefit the resilience community.

In this paper, we describe the initial motivation to leverage standard HPC performance analysis techniques to aid in developing diagnostic tools to assist fault tolerance experiments for HPC applications. These *diagnosis* procedures help to provide context for the system when the errors (failures) occurred. We describe our initial work in leveraging an MPI performance trace tool to assist in providing global context during fault injection experiments. Such tools will assist the HPC resilience community as they extend existing and new application codes to support fault tolerances.

1 Introduction

As large-scale systems increase in size and complexity the high performance computing (HPC) community is exploring ways to cope with the emerging and projected dependability challenges. This is driving research in the area of fault tolerance (FT) and resilience for HPC. The efforts range from algorithmic approaches that seek to make applications failure-oblivious or naturally fault tolerant down to system level changes that offer modular redundancy and differentiated resource allocations based on resilience requirements. These advances have been highlighted in recent whitepapers and workshop reports [4, 6, 11, 5].

The dominant programming model for current HPC systems is based on the message passing interface (MPI). Therefore it is understandable that a number of efforts have focused on enhancing MPI to support fault tolerant computing. The resilience

* ORNL's work was supported by the U.S. Department of Energy, under Contract DE-AC05-00OR22725.

community has also recognized that there is a significant need for tools to assist in resilience experimentation. Unfortunately, there exist very few resilience tools. Most existing tools provide monitoring capabilities to gather metrics about the health of a system, but there is a significant lack of tools to analyze and interpret this data in order to create a global view of the system status and resiliency. Furthermore, users, researchers and system administrators have very different views of the resiliency of the system and existing tools are not appropriate for capturing meaningful resilience data for users and researchers (including data mining tools).

Performance tools on the other hand have been widely studied, deployed and used in the context of HPC. We believe that reusing and possibly extending these "performance tools" can assist the HPC resilience community as it begins to develop "resilience tools". Then the rich set of HPC tools for MPI applications can be extended or *repurposed* to benefit the resilience community. For instance, much of the focus on resilience for MPI is limited to fail-stop errors [2] but tools are lacking to capture the local & global state of the job in the event of a failure. We believe it is possible to use performance tools for MPI on HPC systems to obtain data that can be useful in this resilience oriented context and expose valuable information about the job state in the context of failures. In addition, fault injection (FI) is a standard approach to controllably introduce synthetic errors into a target to aid in the study of the system's fault tolerance properties. These FI tools are beginning to emerge in the context of HPC to assist in the study of application resilience. These tools provide the basis for experimental studies and must be adapted to work in a large-scale computing context. Beyond the basic mechanisms, these tools should also provide support for understanding the state of the system under test to aid in the analysis phase of the experiments. These *diagnosis* procedures help to provide context for the system when the errors (failures) occurred.

In this paper, we describe the motivation to leverage an existing HPC technique to help develop a diagnostic tool that can benefit users and developer experimenting with MPI FT. We describe our initial work in leveraging an MPI performance trace tool to assist in providing global context during resilience experiments, e.g., fault-injection studies.

The following sections are organized as follows: in Section 2 we provide background on resilience and performance tools. In Section 3 we outline the scope and discuss our initial approach for realizing a diagnostics tool for MPI resilience to assist with application fault tolerance experiments. Then in Section 4 we experiment with the approach using the MPI Fault Tolerance Working Groups (MPI-FTWG)'s reference prototype. Section 5 mentions some related work that has leveraged performance oriented tools for resilience experimentation. Finally, in Section 6 we discuss plans for future work and conclusions for this paper.

2 Background

It is common for HPC software developers to employ tools to aid with the analysis of their parallel applications. Performance tools enable the user to obtain supplementary information to better understand the execution behavior of the parallel applications running on the machine. There are different approaches and methods to obtaining this performance data. A common approach is to trace the execution, recording the time for each event under observation. These traces can then be used in postmortem analysis to identify functions or regions of the execution where performance degraded as reflected by the timing data.

In this section, we present an overview of HPC performance tools, their benefits, how they could be used in the context of system resiliency, as well as the associated challenges. We more precisely focus on tracing tools for MPI applications and their usage for postmortem analysis to gather data about system and application resilience, as well as the impact of failures on application performance (including the impact on fault-tolerant or failure-aware applications). The MPI trace file provides useful data for establishing local and global context for the tasks in the parallel job. This can include very simple information like which rank(s) failed? Which ranks were communicating when failure was detected? How long did collectives delay when failure occurred? What functions had the failed rank(s) executed before the failure? Some of these questions can be aided by the fact that it is possible to get traces for "golden" and "failed" runs of the application and do postmortem comparisons. This includes looking at the performance/resilience trade-offs for the application. This can help to answer questions like what MPI functions did the application spend the most time in: (i) during non-error cases?, and (ii) during error cases?

2.1 Benefits of Performance Tools in the Context of Resilience

As previously said, performance tools typically provide the capability of producing an execution trace associated with timing details. From this data, it is possible to extract both the *dependencies* between parts of the parallel applications (between ranks in the context of MPI), as well as *timing* details for the entire parallel application by combining timing data from all the nodes involved in the computation. This data is useful for creating diagnostic data, especially when applications incorporate API's for managing errors: this trace data can be used to explain where additional time is being spent for fault-tolerance and possibly to understand the propagation of errors throughout the parallel application. Therefore, these "performance tools" may provide the basis to gather data to assist in developing HPC "resilience tools".

For instance, in the context of MPI, trace files provide several useful pieces of data in the form of a log of events from the execution of a parallel job. These are time stamped event logs that may include all or a subset of the MPI functions that the application has executed. The logs provide the ordering of functions on a per rank basis within the overall job. Depending on the tool used, the traces may also contain the arguments/parameters and return values for the functions. In the case of MPI, this can include the communicator identifier that is in use. The function entry and exit time are also included, which is typically used to establish performance bottlenecks and attribute portions of the overall wall-clock and CPU time to sections of the execution.

In fault injection experiments, establishing the context for the target is crucial to understanding how failures propagate, etc. The distributed nature of MPI applications adds to the complexity of this context problem because the dependencies are not just in a single location, but potentially distributed across multiple processes in the parallel job. Therefore, establishing the context at both the local and global levels is beneficial.

What functions are active (executing) at the target rank and at other ranks is useful information.

2.2 Challenges Created by the Use of Performance Tools

One challenge of tracing that must be controlled for when using these tools in a resilience context is file generation. In some instances, if a rank fails, the trace file might not be generated. Also, the file may only contain a partial trace for the failed rank(s) due to buffering. Again, this requires that the methods used to introduce synthetic errors (failures) for resilience testing may have to modify the tracing system to flush logs before injecting the failure. This idealized failure to aid experiments/data gathering is typical for FI tools to provide controlled behavior under failure. The point being that care must be taken to avoid losing or corrupting the trace data when performing resilience experiments.

2.3 MPI-FT API

The MPI Fault Tolerance Working Group (MPI-FTWG) has developed additions for the standard that are to be reviewed for inclusion by the MPI Forum. The extensions for MPI are defined in the *User Level Failure Mitigation (ULFM)* proposal, which is described in [2]. These ULFM extensions provide support for detection and notification of process failures for communicating MPI processes. These changes are to provide a basis for building applications that can be extended to be failure-aware, i.e., applications can be notified of rank failures during their execution. The following provides a very brief summary of the ULFM extensions.

The proposed specification [2] adds two new MPI error types: `MPI_ERR_REVOKED` and `MPI_ERR_PROC_FAILED`. Failure notification is done through communicator error handlers. An application can change the default handler (`MPI_ERRORS_ARE_FATAL`) to `MPI_ERRORS_RETURN` and when an error occurs the associated error handler will be called to allow the application to cope with the issue. The ULFM API provides a function, `MPI_Comm_agree()`, to establish consensus over a group of processes in a communicator. When an application is notified of failed ranks, it can revoke (`MPI_Comm_revoke()`) the communicator to force other members of the group to recognize the failure as well. Once revoked, the failed ranks can be removed from a communicator by calling `MPI_Comm_shrink()`, which reduces the size of the associated group. This function is similar to `MPI_Comm_split(comm, newcomm)` with the live ranks of `comm` forming the group returned in `newcomm`. There are functions to locally acknowledge failures on the communicator (`MPI_Comm_failure_ack()`), which quells future notifications about that rank's failure. Additionally, a user can query for failures that have been locally acknowledged via `MPI_Comm_failure_get_acked()`. The MPI-FTWG's reference prototype is implemented in Open-MPI.

3 Requirements & Approach

This investigation is interested in adding assistance for applications taking advantage of the fault-tolerance mechanisms that are being introduced for MPI. We are focused on two classes of users: application developers and fault-tolerance implementers. We will limit the error model to fail-stop, where a fault in a MPI process results in a detectable error that causes the process to fail and stop executing. This is the error model that is being focused upon by the MPI Forum's Fault Tolerance Working Group (MPI-FTWG). The model assumes that failure detection occurs when communication takes place with the failed process(es), e.g., point-to-point, collectives, etc. (Note, the proposed MPI-FTWG extensions also provide a method for forcing the propagation of errors to other members of a group, e.g., `MPI_Comm_revoke()`.)

It can be very difficult to have a global view of the execution state for a parallel job. A MPI tracing tool provides details about what actions have been performed by each process (rank) in the parallel job. The results of these traces can provide details on application characteristics without doing detailed source code analysis as the actual functions and the communication patterns among the ranks are available for postmortem analysis. The trace files provide empirical data that can be used to establish dependencies. The dependencies can then be used to focus further resilience experiments, e.g., fault-injection campaigns. They can also be used to reason about the state of the application upon failure and the propagation of the failure throughout the application.

We assume the testing environment can be managed to avoid trace data corruption and/or loss. This includes controlling the I/O buffering within the trace to ensure we get output from the ranks even if a failure occurs in the job. This likely requires some additional considerations when introducing the errors, e.g., ensuring that failing ranks call `exit()` before terminating. These are part of the controlled environment that should be provided to support fault-injection experiments.

Our approach is to instrument each rank of the MPI job to generate trace files. These files capture all MPI functions at a given rank and the arguments passed to the functions. The application should not need to be modified for the purpose of gathering the data, with the exception of possibly adding support to control failure behavior to ensure traces are generated (i.e., process signal handlers for controlled exit, etc.). The FT-enabled applications will run in both a standard (failure-free) and failure mode.

To evaluate this approach we have prototyped the idea using existing performance tools. We used the DUMPI MPI tracing library that was developed by Sandia as part of their SST/macro project [8]. The traces are recorded in a binary format with utilities to convert the results to plain text. The library records both the input arguments and return values[1] for the MPI functions. The library can also support tracing individual functions, and can be configured to record performance counter data using PAPI [8]. There are utilities for converting DUMPI trace files to Open Trace Format (OTF) files, which could be used with existing performance visualization tools like Vampir.

The DUMPI library was extended to support the MPI-FTWG's proposed ULFM extensions [2]. This allows us to gather trace data for the existing MPI-1 and MPI-2 functions supported by DUMPI, as well as the ULFM functions while experimenting with applications that have been enhanced to take advantage of the FT extensions.

[1] Note, the documentation states the return codes are tracked but the `dumpi2ascii` utility does not appear to display the return codes, with the exception of `MPI_Init()`.

4 Experimentation

We performed a set of experiments on a single desktop and a small Linux testbed to investigate our approach with trace data for example codes. The tests used a basic molecular dynamics code (*simpleMD*) that performs a Lennarad-Jones simulation. The application is written in C and accepts an input files for parameters (e.g., number of time steps, report frequency, etc.). At set intervals the rank-0 of the application prints computed properties (e.g., temperature, kinetic, potential and total energies) and writes to a checkpoint file. The application support restart from the checkpoint file using a command-line option on the `mpirun` command line.

The existing MPI application was enhanced to use the MPI-FTWG's ULFM API. The changes added additional error checking to the MPI calls and forced roll-back to a previous time step if an error occurred during the simulation. This roll-back does not require a full restart of the application but does use the checkpoint data from previous successful steps. The application uses a single communicator, but to accommodate ULFM's shrink functionality, we duplicate `MPI_COMM_WORLD` to have a consistent handle (`smd_comm`) that can be used throughout the application. Lastly, to simulate failures, we modified the `simpleMD` application to force a rank to terminate if the environment variable `ULFM_TEST_FAIL` was enabled. In summary, the steps to add ULFM support to the simpleMD application where:

Step-1: Change all instances of `MPI_COMM_WORLD` to use "smd_comm" handle
Step-2: Change error handlers to `MPI_ERRORS_RETURN`
Step-3: Modify main simulation loop to recognize process failures (`MPI_ERR_REVOKED` and `MPI_ERR_PROC_FAILED`)
Step-4: On error, all call `MPI_Comm_revoke()` and `MPI_Comm_shrink()`
Step-5: Replace "smd_comm" handle with newcomm obtained from shrink
Step-6: Roll back to previous iteration and continue from previous checkpoint

4.1 Performance+Resilience

We ran the `simpleMD` application on our test cluster to gather non-failure and induced-failure trace data. The non-failure traces showed that the application was entirely symmetric and used a large number of explicit synchronizations (`MPI_Barrier().` This is understandable because this is a demonstration application and not necessarily intended to scale to large process counts. We confirmed that none of the ULFM functions were used during the non-failure runs. We also noticed that during our induced-failure tests we called the revoke function twice when only one should be necessary. The tests used 32 nodes of the cluster with 1 rank per node. We could see from the test runs that the wall-time for the shrink was small (less than a half second) but the time spent in the two revokes was between 2-5 seconds (approx. $1-2\%$ of job wall-time) for the job with 1 of the ranks in the forced failure.

As highlighted in Section 3, the trace data provides useful diagnostic information to help explain where additional time is being spent for fault-tolerance and to help understand error propagation through the parallel application. The experiments are performed with specific care to ensure the DUMPI trace files are generated when the failures are introduced into the ULFM enabled application.

4.2 Global Context for Fault Injection

The application was run under normal circumstances to generate a failure-free trace. As a way to evaluate the benefits for fault-injection experiments (Section 3), we use the trace results to identify instances where the communication log could be used to gain insights into the behavior of the application when performing fault-injection experiments. Here we discuss two instances where the trace data was useful for gathering global context during two tests. Note, these tests used the `simpleMD` application with 1000 time steps and status reported every 10 steps.

Note, the trace data in the failure test case can also provide diagnostic information for instances where the target application hangs and global-context can be useful to determine what the overall job is doing (beyond the failed process), i.e., which ranks are waiting, etc. This can be an indicator for portions of the application that are not receiving the notification about rank failures (i.e., problems with detection). This trace data provided diagnostic information that was useful when investigating hangs and other unexpected behavior.

During experiments with the ULFM enhanced application we run a basic test that caused a specific rank to fail to trigger the ULFM detection and force a rollback to previous checkpoint. The steps for the test are as follows:

1. Set an environment variable to signal a failure at runtime.
2. Start 3 ranks with executable linked to trace library ('md-dumpi'). The application runs for 1000 time steps.
3. At hard-coded time step 500, and victim-rank set to rank 1, we simulate a failure by calling `exit()`.
4. Ranks 0 and 2 detect this process failure via ULFM based on return codes from MPI calls (`MPI_ERR_PROC_FAILED` or `MPI_ERR_REVOKED`). They call revoke and then shrink to get a new communicator (`smd_comm`). They fall back to earlier phase and re-initialize data and restart from a saved checkpoint.
5. The program finishes and you see the same result for the last time step as in the non-failure case.

After running this test we discovered unexpected results in the trace files. The root rank (rank 0) finished properly and the trace ended with a call to `MPI_Finalize()`. The forced-fail rank (rank 1) had a shorter trace that ended in calls to get rank, which was expected for the error reporting function that prints the rank. However, the other rank (rank 2) also had a shorter trace with more time steps than rank 1, but fewer than rank 0 and the final lines were not `MPI_Finalize()` but instead resembled that of rank 1. The final result output from the application matched the results from previous runs so the application exited properly and the fault-tolerance mechanisms worked. After closer review of output logs we also noticed that the application only reported the execution time of one rank. This combined with the unexpected results in the trace files made us review the ULFM enhancements to the application. We then recognized a flaw in the fault-injection logic as outlined in the steps above. The ULFM shrink routine reduces the size of the communicator. During the re-initialization the ranks update their rankID for the restored communicator. The hard-coded selection of rank 1 for the simulated failure was repeated on the "new" rank 1 (which was originally rank 2). After adjusting our victim rank selection method, we achieved the expected results and had both ranks complete properly. The revised steps are as follows (*italics indicate the updated parts*:

1. Set an environment variable to signal a failure at runtime.
2. Start 3 ranks with executable linked to trace library ('md-dumpi'). The application runs for 1000 time steps.
3. At hard-coded time step 500, *and victim-rank is set to original `MPI_COMM_WORLD` size minus 1 (*`mcw_size - 1`), we simulate a failure by calling `exit()`.
4. *Ranks 0 and 1* detect this process failure via ULFM based on return codes from MPI calls (`MPI_ERR_PROC_FAILED` or `MPI_ERR_REVOKED`). They call revoke and then shrink to get a new communicator (`smd_comm`). They fall back to earlier phase and re-initialize data and restart from a saved checkpoint.
5. The program finishes and you see the same result for the last time step as in the non-failure case.

This error in the fault-injection logic was simple but illustrates the sort of problems that occur without having some global context. The trace files were a clear indication that the application completed properly but the global job (all ranks) did not operate as we expected/intended. The fault-tolerance methods were operating properly so we achieved the correct results and if not for the additional review from the unexpected trace output we likely would have missed this FI logic error until much later in the testing process.

The previous tests were performed on a single machine. When we moved the same tests over to the Linux cluster we encountered unexpected hangs when we forced a failure. We could tell from the console logs that the rank terminated. We then forced all the stalled ranks to terminate and write their trace files to disk using a signal handler to force them to call `exit()`. When looking at the traces from all the ranks we could see two useful pieces of information. First, all of the ranks were a few steps beyond the failed rank (based on trace file time steps) and all live ranks were at the same time step in a blocking collective. Second, we could see that none of the expected ULFM routines for communicator revocation or shrinking had occurred. This helped to explain why they were all stalled in the collective, because the failure had not been propagated to the different ranks in the job. To remedy this issue we changed our synthetic failure function to use `MPI_Abort(MPI_COMM_SELF,-1)` instead of `exit(1)`. This allowed the failure information to be propagated by the MPI runtime system and allowed the test to run as expected. This is either a configuration problem for the ULFM prototype or possibly a bug/limitation of the current prototype code.

5 Related Work

Recently, research on hardware performance characterization has extended preexisting tools to aid in resilience studies for soft errors in HPC applications. In [10], a fault injection tool was developed that leveraged the binary instrumentation tool *PIN*. Their tool, *BIFIT*, was used to introduce errors into specific symbols/data-structures of three real-world HPC applications to study the effect of simulated "soft errors". They used the instrumentation tool to inject bit-flips into application-specific regions of global, heap and stack data objects. They also leveraged the memory profiling support of their performance tools to identify candidate injection sites. In contrast to this work, we use performance traces to gather context about the application to identify errors during fault-injection experiments and to provide global context about the application. The trace data could also provide candidate ranks for injecting process failures based on the communication patterns.

The following research has also leveraged HPC performance tools to develop resilience and diagnostic tools. The Extreme-scale Simulator (xSim) is a performance investigation toolkit for running MPI applications at extreme scale by running on existing machines in a heavily oversubscribed mode [3]. The Sandia SST/Macro [8] project has similar goals but employs a different approach. Both of these performance tools employ parallel discrete simulation to study the effects of different hardware characteristics on application performance as the systems scale up in size. The xSim project is currently working to extend the performance toolkit to provide support for resilience investigations. Another related area is that of large-scale debugging and diagnosis for parallel HPC applications [1, 7]. The challenges are similar in that you must be able to gather data about the distributed application and provide details for diagnosis to identify the cause of the error. The AutomaDeD [9] project combined sampling and classification/clustering methods to help identify abnormalities that can provide signatures for software faults (bugs) to aid in debugging while maintaining reasonable performance.

6 Conclusion

There are many well established tools and techniques for performance evaluation and experimentation with HPC applications. This pre-existing infrastructure can provide useful data for resilience experiments. We have briefly discussed some initial efforts to extend an existing MPI tracing package to support the MPI FTWG's proposed ULFM specification. The data obtained from these traces can assist application developers and FT implementers for diagnosing problems and help with postmortem analysis.

To investigate the usefulness of the trace tools we extended a simple molecular dynamics application to use the ULFM enhancements to MPI. Our initial experiments used the trace files from the tests to help gain insights into the context of the job during resilience experiments. The traces helped to highlight two problems we encountered during fault injection experiments: i) a fault-injection logic error that resulted in correct results (application output), but more ranks than anticipated being killed; ii) an issue in failure detection/propagation with the ULFM prototype that was effected by the method used to simulate the rank failure. The trace files can also help to explain changes to overall performance when MPI fault tolerance mechanisms are employed.

In future work, we plan to continue our initial efforts to use the MPI tracing library for other work in resilience tools. The next steps will be to add support for generating these traces to the Extreme-scale Simulator (xSim). The xSim performance analysis toolkit was recently extended to support capabilities to assist in resilience studies, e.g., process and job fault-injection. Using the simulator will improve control for several items (e.g., time skew, IO buffer flushing, failure scheduling). Another potential benefit of traces is that they can provide a "replay" log for communication when combined with simulation tools. This could allow for reviewing complex interleaving of events when working on adding fault tolerance to existing codes.

References

1. Dong H. Ahn, Bronis R. de Supinski, Ignacio Laguna, Gregory L. Lee, Ben Liblit, Barton P. Miller, and Martin Schulz. Scalable temporal order analysis for large scale debugging. In *Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2009, November 14-20, 2009, Portland, Oregon, USA*. ACM, 2009.
2. Wesley Bland, Aurelien Bouteiller, Thomas Herault, Joshua Hursey, George Bosilca, and Jack J. Dongarra. An evaluation of user-level failure mitigation support in MPI. In *Proceedings of the 19th European conference on Recent Advances in the Message Passing Interface*, EuroMPI'12, pages 193–203, Berlin, Heidelberg, 2012. Springer-Verlag.
3. Swen Böhm and Christian Engelmann. xSim: The extreme-scale simulator. In *Proceedings of the International Conference on High Performance Computing and Simulation (HPCS)*, pages 280–286, Istanbul, Turkey, July 4-8, 2011. IEEE Computer Society, Los Alamitos, CA, USA. Acceptance rate 28.1% (48/171).
4. J. Daly, B. Harrod, T. Hoang, L. Nowell, B. Adolf, S. Borkar, N. DeBardeleben, M. Elnozahy, M. Heroux, D. Rogers, R. Ross, V. Sarkar, M. Schulz, M. Snir, P. Woodward, R. Aulwes, M. Bancroft, G. Bronevetsky, B. Carlson, A. Geist, M. Hall, J. Hollingsworth, B. Lucas, A. Lumsdaine, T. Macaluso, D. Quinlan, S. Sachs, J. Shalf, T. Smith, J. Stearley, B. Still, and J. Wu. Inter-Agency Workshop on HPC Resilience at Extreme Scale, February 2012.
5. Nathan DeBardeleben, James Laros, John T. Daly, Stephen L. Scott, Christian Engelmann, and Bill Harrod. High-end computing resilience: Analysis of issues facing the HEC community and path-forward for research and development. Whitepaper, December 2009.
6. J. Dongarra, P. Beckman, and et al. The international exascale software roadmap. *International Journal of High Performance Computer Applications*, 25(1), 2011.
7. Joshua Hursey, Chris January, Mark O'Connor, Paul H. Hargrove, David Lecomber, Jeffrey M. Squyres, and Andrew Lumsdaine. Checkpoint/restart-enabled parallel debugging. In *EuroMPI 2010: Proceedings of the 17th EuroMPI Conference*, Stuttgart, Germany, September 2010.
8. Curtis L. Janssen, Helgi Adalsteinsson, Scott Cranford, Joseph P. Kenny, Ali Pinar, David A. Evensky, and Jackson Mayo. A simulator for large-scale parallel computer architectures. *International Journal of Parallel and Distributed System Technology*, 1(2):57–73, April 2010.
9. Ignacio Laguna, Todd Gamblin, Bronis R. de Supinski, Saurabh Bagchi, Greg Bronevetsky, Dong H. Anh, Martin Schulz, and Barry Rountree. Large scale debugging of parallel tasks with AutomaDeD. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 50:1–50:10, New York, NY, USA, 2011. ACM.
10. Dong Li, Jeffrey S. Vetter, and Weikuan Yu. Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. ACM, November 2012.
11. M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, A. A. Chien, P. Coteus, N. A. DeBardeleben, P. Diniz, C. Engelmann, M. Erez, S. Fazzari, A. Geist, R. Gupta, F. Johnson, S. Krishnamoorthy, S. Leyffer, D. Liberty, S. Mitra, T. Munson, R. Schreiber, J. Stearley, and E. V. Hensbergen. Workshop report: Addressing failures in exascale computing, April 2013.