Simplifying the Recovery Model of User-Level Failure Mitigation

Wesley Bland, Kenneth Raffenetti, Pavan Balaji
Argonne National Laboratory
Mathematics and Computer Science Division
Argonne, IL USA
{wbland, balaji}@anl.gov, raffenet@mcs.anl.gov

***Abstract*—As resilience research in high-performance computing has matured, so too have the tools, libraries, and languages that result from it. The Message Passing Interface (MPI) Forum is considering the addition of fault tolerance to a future version of the MPI standard, and a new chapter called User-Level Failure Mitigation (ULFM) has been proposed to fill this need. However, as ULFM usage has become more widespread, many potential users are concerned about its complexity and the need to rewrite existing codes. In this paper, we present a usage model that is similar to the usage already common in existing codes but that does not require the user to restart the application (thereby incurring the costs of re-entering the batch queue, startup costs, etc.). We use a new implementation of ULFM in MPICH, a popular open source MPI implementation, and demonstrate the ULFM usage using the Monte Carlo Communication Kernel, a proxy-app developed by the Center for Exascale Simulation of Advanced Reactors. Results show that the approach used incurs a level of intrusiveness into the code similar to that of existing checkpoint/restart models, but with less overhead.**

I. Introduction

Fault tolerance research in high performance computing (HPC) has moved from an academic study to a serious concern as we grow ever closer to exascale. More and more evidence continues to point to the eventuality that machines will become less reliable at large scale. Even some current machines see failure rates of more than one failure per day. Applications will face a variety of potential failures, including failures impacting specific components of the system, such as volatile memory, long-term storage, and communication interfaces, and full-node failures, such as processor failures, power failures, and thermal issues. Therefore, an application must have a complete set of tools to detect, recover from, and repair a variety of failures. This tool set is only now being constructed, but clearly it must mix both interfaces with which application and library developers are already familiar and new interfaces and libraries that provide novel solutions to resilience issues.

The Message Passing Interface (MPI) [22] is the most ubiquitous interface for communication in HPC applications. The interface was first written in 1993 and has since been adapted as the landscape of HPC has changed. With the addition of new features such as one-sided memory access, I/O, and dynamic process management, the MPI interface remains applicable on today's machines even after five orders of magnitude of growth. To remain applicable to new scales, however, MPI must confront the problems of future machines, including resilience. The MPI Forum's Fault Tolerance Working Group [2] has been working toward this goal and has put forward a proposed chapter using the working title "User Level Failure Mitigation" (ULFM) [17]. This chapter is specific to process fault tolerance because this is currently the most studied area of resilience. The working group will continue to explore and potentially standardize (when necessary) a more comprehensive set of tools to help applications recover from other types of failures.

As ULFM has become more publicized and studied, one of the most common responses has been to criticize its complexity for existing large-scale codes, which are unlikely to be completely redesigned to fully take advantage of ULFM's new features. Arguably, the applications that will initially be the simplest to adapt to use ULFM are those that can perform local recovery or continue executing with fewer processes. Nevertheless, other types of applications will also gain from the new APIs provided.

In this paper, we investigate how a common class of applications — iterative refinement stencil codes — can use ULFM to achieve fault tolerance without requiring heavyweight disk-based checkpoints or restarting the entire MPI application. We show that using ULFM to achieve this does not require any more code changes than does a traditional fault tolerance model (i.e. checkpoint/restart). We also demonstrate that by using ULFM, the total time to solution is reduced.

To demonstrate these advances, we use a new implementation of ULFM being added to MPICH [3], the most widely used MPI implementation on HPC systems. We also use the Monte Carlo Communication Kernel (MCCK) [20] as a representative code for the class of applications discussed here. MCCK is a proxy-app developed by the Center for Exascale Simulation of Advanced Reactors (CESAR) as a way for computer scientists to investigate new tools to be used in large-scale codes without requiring the developer of the tool to understand hundreds of thousands of lines of domain-specific code. MCCK uses a standard stencil computation with a halo communication style to simulate the movement of particles in a nuclear reactor between computation domains. We adapt this proxy-app in two ways to add resilience. First, we use the traditional checkpoint/restart style of execution using checkpointing to disk in order to demonstrate the performance of the common model. Then, we modify the app again to use ULFM to show the improvement gained by using the new tools.

The rest of the paper is organized as follows. In Section II, we discuss the background of the proposal and related work.

Section III briefly describes our implementation of ULFM in MPICH. Section IV describes our work with MCCK. Section V discusses the performance of MCCK after the modifications. Section VI wraps up the paper and discusses future work.

II. Background & Related Work

Resilience in MPI is not a new subject. Research specifically into the checkpoint/restart model has be going on for many years. Berkeley Lab Checkpoint/Restart (BLCR) [11] is the checkpoint/restart library most commonly used by applications and MPI implementations. It was initially developed in 2005 and remains the defacto model for saving full-system checkpoints. BLCR support was added to many MPI implementations including the most popular open source implementations MPICH and Open MPI [12] as well as predecessors such as LAM/MPI [19].

As the footprint of applications became larger and the time spent writing checkpoints increased, researchers began to explore ways to improve the time necessary to write each checkpoint. Initially, the focus was on moving the checkpointing model from a synchronous model, where all processes would simultaneously write a checkpoint to disk after quiescing the network, to a more asynchronous one, where checkpoints could be taken independently and messages between the checkpoints were logged to allow the system to replay them after rolling back to a previous checkpoint [8]. This work improved the checkpoint times but also increased the memory requirement of checkpointing due to the message logging. More work was then done to attempt to make checkpoints even smaller and store only the most necessary information. This led to a large body of work involving application-level checkpointing including FTI [5], SCR [16], or GVR [1].

While resilience in MPI has been studied extensively [7], [9], [10], [15], ULFM is a relative newcomer. The chapter was first proposed to the MPI Forum's Fault Tolerance Working Group in 2012 [6] as a reimagining of previous proposals for MPI fault tolerance, including, most recently, Run-Through Stabilization [13]. The goal of ULFM, as is true of MPI as a whole, was not to make a particular model of resilience easy, but to form the foundational tools necessary to allow any model of resilience (roll-back recovery, roll-forward recovery, application-based fault tolerance, natural fault tolerance, transactions, etc.) to be constructible based on the provided interface.

ULFM provides the foundational tools necessary for resilience: failure notification, discovery, and recovery. Notification takes place by using the existing MPI error notification system through error codes and `MPI_ERRHANDLER`s. ULFM adds new error classes to inform the user when a process failure has been detected. To discover which processes have failed, the application can call the new functions `MPI_COMM_FAILURE_ACK` and `MPI_COMM_FAILURE_GET_ACKED`, which will provide a group of all processes that the local processes knows have failed. This group is not synchronized across all processes in a communicator. If such global knowledge is required, the user can call `MPI_COMM_REVOKE` to enforce it. This nonlocal, noncollective function causes all future nonlocal MPI operations to return with the error class `MPI_ERR_REVOKED`. This function is not matched on remote processes, so once it is called locally, it will return as soon as the revoke messages have been sent to all necessary remote processes, and the application can continue, knowing that at some point in the future, all processes will be notified of the failure. Recovery takes place via the new communicator creation function `MPI_COMM_SHRINK`. This function creates a new communicator after internally agreeing on a list of processes that have failed and excluding those processes from the new communicator. The final new function, `MPI_COMM_AGREE`, is used to perform a manual, fault-tolerant agreement operation at the user level. This allows the application to determine global status when necessary, such as at the end of algorithm segments or before calling `MPI_FINALIZE`.

In [6], the authors evaluate the performance of a ULFM implementation based on Open MPI. The evaluation is done using micro-benchmarks and some small applications, but it is focused largely on failure-free performance in an effort to demonstrate the low overhead introduced by the changes required by ULFM. This is important work in showing how ULFM can be implemented with minimum impact on existing applications, but it does not demonstrate the use of ULFM to actually recover from failures. In this paper, we provide a demonstration of how ULFM can be used with existing applications and evaluate the performance including recovery.

Recently, St Pauli and Schwab demonstrated the use of ULFM in real applications [21]. They showed how to implement fault tolerance in a naturally fault-tolerant algorithm by implementing redundancy within the algorithm to allow the application to ignore failed processes and continue using the stabilization features of ULFM. We demonstrate a more practical, immediately accessible usage here.

III. ULFM Implementation

As part of the work of demonstrating the implementation of a resilient application, we have implemented ULFM inside MPICH. This implementation required modifications to some of the existing pieces of MPICH but was largely kept isolated from performance-critical code. Error checking remains disabled by default and is activated only when the user provides the configure flag `--enable-error-checking=all`. This approach allows MPICH to internally check for more errors and for unique situations such as revoked communicators. It remains optional if fault tolerance is not required because it introduces a new conditional statement that could have a minor performance impact on extremely latency-sensitive applications.

The algorithms used for the implementation of the new ULFM functions are not complex or optimized, but they still provide reasonable performance and will be better optimized in future versions of the code. We provide some details about the algorithms here:

a) Failed Processes: Acquiring the list of failed processes is implemented by using the Process Management Interface (PMI) [4]. PMI is responsible for all processes involved in an MPICH job and tracks a variety of information about those processes, including whether they have failed. This information is aggregated at the PMI server and is shared

among the local PMI clients on each process. When a failure occurs, the PMI client alerts MPICH via a signal that will be handled the next time the user makes a call into the MPICH communication library (but not for calls that act only locally). To produce the list of failed processes in order to construct the group for `MPI_COMM_FAILURE_GET_ACKED`, MPICH gets the list of failed processes from PMI. However, to prevent MPICH from needing to cache the list of failed processes internally per communicator in order to satisfy the requirement that MPI not provide a group of failed processes that is different from the last time the user called the function `MPI_COMM_FAILURE_ACK`, we track only the last acknowledged process failure per communicator. When constructing the group of failed processes, we parse the list from PMI (which is in the same order every time it is retrieved) and stop when we reach the last failed process.

b) Agree: The agreement algorithm is implemented as a gather/scatter algorithm to determine the group of failed processes. Then two calls to an internal allreduce function determine the status of the flag provided as an argument and the return value of the completed function. According to the specification, these two values must be synchronized across all processes. If a failure occurs during the agreement, the remaining processes will return `MPI_ERR_PROC_FAILED` if the failure occurs before the determination of the return code, or `MPI_SUCCESS` if it occurs afterward.

c) Shrink: The implementation of the shrinking function is similar to that of the agreement function. The failed processes are determined via a gather/scatter; success is determined happens via an internal allreduce. The difference comes with the addition of the communicator creation. The communicator is constructed by using the MPI-3 function, `MPI_COMM_CREATE_GROUP`, with some modifications to ensure that a failure will not cause internal deadlock.

d) Revoke: Implementation of revocation is the most complex of the new functions. This complexity is introduced because `MPI_COMM_REVOKE` is a completely new type of function. It is not a collective function, which provides matching calls at all participating processes. Rather, the revoke operation must act as an active message, or a one-sided operation, which is handled in a packet handler without user involvement. Because only one process can be involved in starting the revoke algorithm and because redundant paths between all processes may not exist, the current version of revoke is implemented as a cascading all-to-all function. When one process first calls `MPI_COMM_REVOKE`, it sends a revoke message to all other processes in the communicator and initializes a counter to track how many other processes have yet to send a revoke message to the local process. As each process receives the message to revoke the communicator, it initializes its own counter and sends its own revoke message to all other processes. As each process receives the revoke messages, it decrements its own counter. Receiving a message about a process failure also decrements the counter. When the counter reaches 0, it deletes the communicator and frees the internal context id to be reused by future communicators. Obviously, this algorithm is not scalable for future machines, but for current hardware it completes in a reasonable time and will be called relatively infrequently. Improved versions of this algorithm will be developed as future work.

IV. Monte Carlo Communication Kernel

In this section, we detail the work done to extend the Monte Carlo Communication Kernel (MCCK) [20] mini-app to add process failure resilience. MCCK is a representative application for a variety of domain decomposition applications that perform stencil computation by sending and receiving data only with their direct neighbors. This application is developed as part of the Center for Exascale Simulation of Advanced Reactors (CESAR) at Argonne National Laboratory.

We modified MCCK in two ways to demonstrate how applications can use ULFM in a minimally intrusive fashion. First, we used standard application-level checkpointing. Second, we used ULFM to employ a similar protection strategy, but keeping data in-memory. Both modifications require a minimal amount of data to be protected in order to continue executing after a failure. MCCK tracks the position of particles as they move through a region that is divided among the different processes in the computation. Hence, we needed to preserve the location of these particles. In this section, we describe how this data is protected and repaired using our two mechanisms.

A. Application-Level Checkpointing Modifications

The first method of data protection is the most familiar to most application developers. We used a basic checkpointing scheme to store the position of the particles in checkpoint files. After a predetermined number of intervals, we captured the location of the particles and checkpointed that information to disk. While this approach makes data protection relatively simple, recovery after a failure is more involved. Because the application does not attempt to continue to communicate after a process failure, it must shut down and restart after each failure. In practice, this approach might also involve re-entering the batch queue and waiting for the job to be restarted. When the job starts again, the data is re-initialized and the checkpoint file read back into memory. Thus, the simulation is brought back to the point where the failure occurred, and the job can continue.

Two major sources of overhead are found with this form of resilience: writing checkpoint files, and the recovery time after a failure. Writing checkpoint files can be expensive because the well known relative speed of writing data to disk as opposed to storing the data in memory (even remote memory). The problem of determining the optimal checkpoint overhead is well-studied [14], [18], [23], and we will not cover it here.

The second source of overhead does not occur during executions without failures; but if periodic failures become commonplace (as expected with exa-scale) recovery time will be an important metric to track the total time to completion. In the checkpoint/restart model, recovery overhead involves the time to restart the job, time to restart the application, and time to restore the application data. While in some systems the time to restart the job may be minimal (such as systems where no batch scheduler exists or has a job is not aborted after a process failure), on many systems this is not currently the case. For example, on IBM Blue Gene machines, the batch scheduler will abort any job in which a process has failed because of technical limitations. Therefore, we need to account for this time when considering the overhead of recovery. Moreover, the

time to restart the application must be considered. Usually, this is the amount of time necessary to start an MPI job. While this time has improved dramatically in recent releases of most MPI implementations, it is still nonzero. This time also includes any time necessary to reinitialize the application, including setting up MPI communicators and data structures. In addition, we must consider the time spent actually recovering the data written to the checkpoint files. Recent advancements in checkpointing techniques have minimized this time, and applications are beginning to use less heavyweight checkpointing schemes; but as with writing checkpoints, this recovery time involves reading data from the disk and possibly communicating some of the data to other processes. We evaluate all these costs in Section V.

B. ULFM Modifications

For our ULFM version of the MCCK mini-app, we wanted to mirror the checkpoint/restart version as closely as possible in order to demonstrate a common use case for ULFM. However, because the ULFM version of the benchmark does not require the application to restart, the checkpoint data does not need to be stored to disk. Instead, the data is kept by neighboring processes. At the end of an iteration, each process sends the location of its particles to its neighbor. Therefore, after a failure, the recovery data can be acquired by asking a neighbor instead of reading the data from disk. Our method of repairing the execution after a process failure also changes slightly because the application will not be restarting. Instead, the application needs to replace the failed process in one of two ways: using a replacement process created at the start of the execution or spawning a replacement process as needed during the execution. While the first method of including spare processes during application startup is valid and compatible with more existing systems at the current time, the method of spawning replacement processes will allow an application to tolerate an arbitrary number of process failures, so we use it here. Our code can easily be modified to use the alternative method if desired.

Because we no longer rely on MPI to automatically abort after a process has failed, we must do some minor additional work to determine the status of the execution after an iteration of the algorithm is completed. In this instance, we use the new function introduced in UFLM, `MPI_COMM_AGREE`. This function performs a fault-tolerant agreement algorithm among all alive processes in the execution by accepting an integer value as input and performing a bitwise `AND` operation over the value collected from each process. If a process failed before calling `MPI_COMM_AGREE`, it will cause all other processes to return `MPI_ERR_PROC_FAILED`. Therefore, we can detect a failure from either the return value or the input value. If we detect that a process has failed, then all processes return their data to the last known good state (the state of the data after the last call to `MPI_COMM_AGREE`), which is kept in local memory. At this point, each process will discover the rank of the failed process by using `MPI_COMM_FAILURE_ACK` and `MPI_COMM_FAILURE_GET_ACKED` and construct a working communicator by calling `MPI_COMM_SHRINK`. This shrunken communicator is used to spawn a replacement process using `MPI_COMM_SPAWN`, which is patched into the position vacated by the failed process using `MPI_INTERCOMM_MERGE` and `MPI_COMM_SPLIT`. At this point, the resulting communicator contains the correct number of processes, and all processes have the same ranks that they did in the original communicator (with the replacement process in the place of the failed process). In order to continue the execution, the replacement processes must acquire the current iteration number and data about its particles by querying its neighbors.

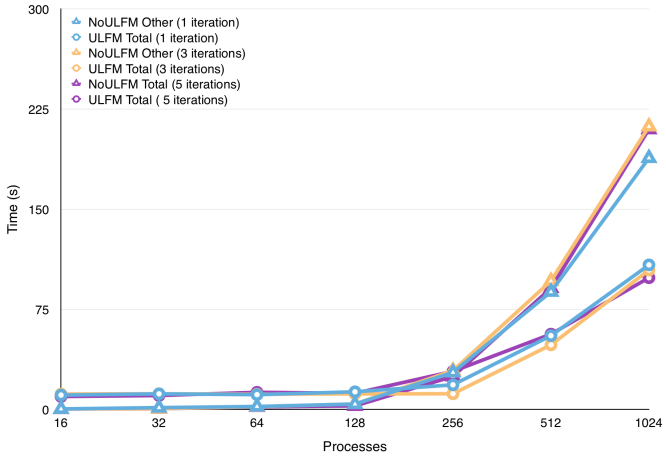

Fig. 1: Runtime of MCCK benchmark with application-level checkpointing and ULFM

V. Performance Evaluation

Performance analysis was done using the Fusion cluster at Argonne National Laboratory. Fusion nodes have 8 compute cores (2 Pentium Xeon Processors), and 36 GB of memory and are connected by QDR InfiniBand and Gigabit Ethernet. Runtime measurements were done for process counts up to 1,024, starting with 100,000 particles per process, and averaged of three runs. For reproducibility, we used an MCCK-provided option to always seed the randomness function with the same value.

In these experiments, we evaluated our two versions of the MCCK benchmark (one using application-level checkpointing and one using ULFM) and modified the frequency of the data protection. For the application-level checkpointing benchmark, checkpoints were taken more or less frequently; for the ULFM benchmark, the agreement was also performed more or less frequently. We varied this value between every iteration, every third iteration, and every fifth iteration of the mini-app. Because the relatively short runtime of the benchmark we were unable to produce meaningful results for higher iteration counts, but we can make interesting observations from the trends that emerge. During all the executions, we kill one process near the midpoint of the job and allow the application to recover using the predetermined strategy.

In Figure 1, we see the overall runtime of the MCCK benchmark in multiple configurations. Across all executions, at scales higher than 256 processes, the total execution time is better with ULFM than with application-level checkpointing. We expected to see these results because the time necessary to

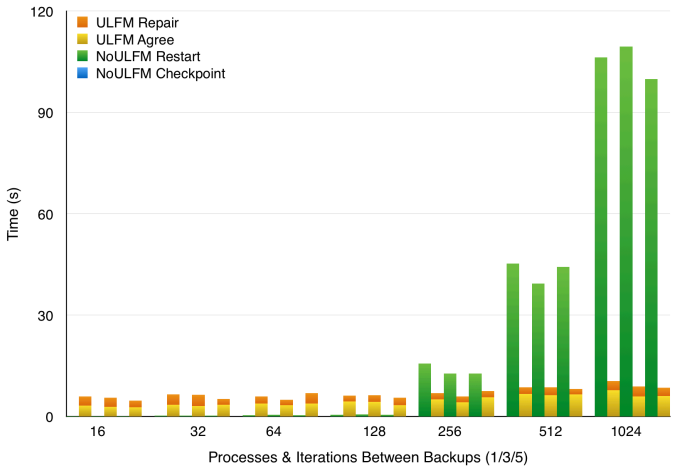

Fig. 2: Overhead of resilience in MCCK benchmark with backups every 1, 3, and 5 iterations

restart an MPI application and simultaneously read the checkpoint files from disk increases with the number of processes in the execution at a rate much faster than performing collective operations inside an existing MPI application (which is the requirement of the ULFM execution).

For a more detailed analysis of the execution time and the impact of the resilience schemes, we analyzed the execution in more detail. In Figure 2, we can see a breakdown of the time spent protecting the application data from failures (Checkpoint in the checkpoint/restart execution and Agree in the ULFM execution) and the time spent repairing the execution after a failure (Restart in the checkpoint/restart execution and Repair in the ULFM execution). In this figure, each group of bars represents a number of processes. Each green and blue (not seen due to small size) bar represents the overhead introduced by traditional checkpoint restart, and each orange and yellow bar represents the overhead introduced by ULFM. The first two bars in each group demonstrate a checkpoint every iteration, the second two show checkpoints every third iteration, and the final two every fifth iteration.

We can see that the time spent protecting the data is actually greater in the ULFM execution (approx. 6 seconds) than in the checkpoint/restart execution (less than 1 second). This is not surprising since the ULFM implementation on which this work is based has not yet been optimized to use sophisticated agreement algorithms. We expect this number to decrease as future versions of the agreement algorithms will scale better and minimize communication compared with the current naive implementation. Another point to consider when evaluating the overhead of the both MCCK implementations is that while the MCCK mini-app performs representative communication, it does not perform all of the computation that would be present in a full large-scale application. Therefore, at large scale, we would expect the relative overhead introduced by the both resilience strategies to drop significantly as the time spent performing computation will eclipse the time spent writing checkpoints or performing agreements.

When we evaluate the recovery time, we see that the ULFM implementation vastly outperforms the checkpointing implementation. This is expected because of the overhead of completely restarting the MPI runtime and reading a full set of checkpoints simultaneously. To determine how the recovery time is being spent, we evaluated an execution of MCCK with checkpointing running with 512 processes and checkpointing every 5 iterations. We found that the amount of time spent restarting the MPI job took more than 99% of the recovery time, and relatively little time was spent re-reading the checkpoint. Alternatively, we see that the repair time of ULFM grows at a much slower rate. The reason is that the ULFM implementation does not have to restart the MPI job or read data from disk. Instead, it only needs to spawn a replacement process and recreate its communicator (which involves performing a small number of collective operations). The data is read from memory instead of disk, thus dramatically decreasing the recovery time. While the ULFM algorithms used during the recovery period have also not yet been optimized, their impact is much lower since they are called only once a failure has actually occurred and therefore do not impact normal runtime.

We also see that for the MCCK mini-app, the checkpoint frequency has relatively little impact on the overall runtime. We can see a minor drop in total runtime as the number of checkpoints or agreements decreases (specifically in the ULFM implementation as the previous performance discussion would indicate). For applications that have larger checkpoint times, the impact of checkpoint frequency will be much more dramatic.

VI. Conclusion and Future Work

As users have begun to evaluate how their applications can take advantage of possible new fault tolerance additions coming to the MPI standard, early feedback indicates that not all the tools will be necessary for all applications. In this paper, we demonstrate how an application that already uses a traditional checkpoint/restart recovery model can easily convert to using ULFM and improve its recovery time dramatically by not requiring the application to restart the entire MPI job and read and write checkpoints to disk. We used the Monte Carlo Communication Kernel mini-app as a representative application for large scale executions and showed that the total time to completion for some executions can improve by as much as 75% by using ULFM.

We intend to improve the performance of the various new ULFM functions as they are implemented in MPICH. We also plan to demonstrate ULFM's usage in more complex applications that will use the full set of ULFM features.

Acknowledgment

This material is based upon work supported by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research, under contract number DE-AC02-06CH11357.

References

[1] "Global View Resilience: GVR." [Online]. Available: https://sites.google.com/site/uchicagolssg/lssg/research/gvr

[2] "MPI Forum Fault Tolerance Working Group." [Online]. Available: https://svn.mpi-forum.org/trac/mpi-forum-web/wiki/FaultToleranceWikiPage

[3] "MPICH." [Online]. Available: http://www.mpich.org/

[4] P. Balaji, D. Buntinas, D. Goodell, W. D. Gropp, J. Krishna, E. L. Lusk, and R. Thakur, "PMI: A Scalable Parallel Process-Management Interface for Extreme-Scale Systems," in *17th EuroMPI Conference, Lecture Notes in Computer Science*, Stuttgart, Germany, 2010, 11/2009 2009. [Online]. Available: http://www.springerlink.com/content/q9u361j4q6800773/

[5] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka, "FTI: High Performance Fault Tolerance Interface for Hybrid Systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 32:1–32:32. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063427

[6] W. Bland, A. Bouteiller, T. Herault, J. Hursey, G. Bosilca, and J. J. Dongarra, "An evaluation of user-level failure mitigation support in MPI," in *Recent Advances in the Message Passing Interface*. Springer, 2012, pp. 193–203.

[7] W. Bland, P. Du, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra, "A Checkpoint-on-Failure protocol for algorithm-based recovery in standard MPI," in *Euro-Par 2012 Parallel Processing*. Springer, 2012, pp. 477–488.

[8] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov, "MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes," in *Supercomputing, ACM/IEEE 2002 Conference*, Nov 2002, pp. 29–29.

[9] G. E. Fagg and J. J. Dongarra, "FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world," pp. 346–353, 2000.

[10] W. Gropp and E. Lusk, "Fault Tolerance in Message Passing Interface Programs," *International Journal of High Performance Computing Applications*, vol. 18, no. 3, pp. 363–372, 2004. [Online]. Available: http://hpc.sagepub.com/content/18/3/363.abstract

[11] P. H. Hargrove and J. C. Duell, "Berkeley lab checkpoint/restart (blcr) for linux clusters," *Journal of Physics: Conference Series*, vol. 46, no. 1, p. 494, 2006. [Online]. Available: http://stacks.iop.org/1742-6596/46/i=1/a=067

[12] J. Hursey, J. Squyres, T. Mattox, and A. Lumsdaine, "The design and implementation of checkpoint/restart process fault tolerance for open mpi," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, March 2007, pp. 1–8.

[13] J. Hursey, R. L. Graham, G. Bronevetsky, D. Buntinas, H. Pritchard, and D. G. Solt, "Run-through stabilization: An MPI proposal for process fault tolerance," in *Recent Advances in the Message Passing Interface*. Springer, 2011, pp. 329–332.

[14] W. M. Jones, J. T. Daly, and N. DeBardeleben, "Impact of sub-optimal checkpoint intervals on application efficiency in computational clusters," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. New York, NY, USA: ACM, 2010, pp. 276–279. [Online]. Available: http://doi.acm.org/10.1145/1851476.1851509

[15] A. Kanevsky, A. Skjellum, and A. Rounbehler, "MPI/RT - an emerging standard for high-performance real-time systems," vol. 3, pp. 157–166 vol.3, 1998.

[16] A. Moody, G. Bronevetsky, K. Mohror, and B. de Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, Nov 2010, pp. 1–11.

[17] MPI Fault Tolerance Working Group, "User level failure mitigation." [Online]. Available: https://svn.mpi-forum.org/trac/mpi-forum-web/wiki/User_Level_Failure_Mitigation

[18] J. S. Plank and M. G. Thomason, "Processor allocation and checkpoint interval selection in cluster computing systems," *Journal of Parallel and Distributed Computing*, vol. 61, no. 11, pp. 1570 – 1590, 2001. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731501917575

[19] S. Sankaran, J. M. Squyres, B. Barrett, V. Sahay, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman, "The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing," *International Journal of High Performance Computing Applications*, vol. 19, no. 4, pp. 479–493, 2005. [Online]. Available: http://hpc.sagepub.com/content/19/4/479.abstract

[20] A. Siegel, K. Smith, P. Fischer, and V. Mahadevan, "Analysis of communication costs for domain decomposed Monte Carlo methods in nuclear reactor analysis," *Journal of Computational Physics*, vol. 231, no. 8, pp. 3119–3125, 2012.

[21] P. A. St Pauli and C. Schwab, "Intrinsic fault tolerance of multi level Monte Carlo methods," ETH Zurich, Computer Science Department, Tech. Rep., 2012.

[22] The MPI Forum, "MPI: A Message-Passing Interface Standard, Version 3.0," Tech. Rep., 2012.

[23] J. W. Young, "A first order approximation to the optimum checkpoint interval," *Commun. ACM*, vol. 17, no. 9, pp. 530–531, Sep. 1974. [Online]. Available: http://doi.acm.org/10.1145/361147.361115