Toward Recovery Capabilities in Message Passing Environments

Wesley Bland
Department of Electrical Engineering and Computer Science
University of Tennessee, Knoxville
wbland@icl.utk.edu

I. Dissertation Statement

The goal of the dissertation is to demonstrate novel methods of fault tolerance supporting Algorithm Based Fault Tolerance (ABFT) for large scale systems using the message passing paradigm with extensions to facilitate concurrent approaches to cope with failures. By allowing applications to continue execution after a hardware failure, algorithms can support larger scale and longer running executions previously found to be unattainable.

II. Motivation

As High Performance Computing (HPC) reaches petascale and exascale, new challenges have emerged which necessitate a change in the way large scale operations are designed. As of the June 2012 Top500 list[1], machines at the top of the list have now surpassed the million-core mark. At this **scale**, reliability becomes a major concern. Reliability, availability and serviceability data mainly provided by the Los Alamos National Laboratory, National Energy Research Scientific Computing Center, Pacific Northwest National Laboratory, Sandia National Laboratory, and Laurence Livermore National Laboratory analyzed by Schroeder and Gibson [22] show an average number of failures per year between 100 and 1000 depending on the system. Projections found in Cappello's paper [8] predict a future mean time to failure of approximately one hour. With failures of that frequency, capacity applications will not be able to complete without considering a model for handling hardware failures.

Beyond traditional high performance computing environments, other new areas of distributed computation have also emerged which produce similar needs. Volatile resources such as cloud computing environments have been considered unsuitable for more traditional distributed computing models because of their constantly changing set of resources. If the underlying programming model could support the kind of drop in, drop out behavior that volatile environments need, they could become a lower cost set of tools available for developers. Another area which could benefit from a resiliency model is energy efficient computing. An application could choose exactly the number of processors necessary at any point in the application lifecycle and allow unnecessary resources to shut down, vastly reducing cost and energy consumption. In addition to more efficiently using processors when designing an application, by allowing applications to become **resilient** and continue execution beyond failures, expensive recalculations become unnecessary, saving energy and computation hours. For both of these computing models, a new, resilient programming model is necessary.

While the need for resilience at scale has been proven through many previous studies, an important factor for wide adoption which is often ignored is **usability**. Many previous efforts to introduce fault tolerance methods into high performance computing have gone unutilized because they were too difficult to use or required large changes to existing codes. For any tool to be employed, it must not only fulfill the need of the community, but also be compatible with the other existing tools.

In this work, we provide two new fault tolerance models for application and library developers to choose from to solve these problems. We used the de-facto programming environment for parallel applications, the Message Passing Interface (MPI), to provide a familiar, portable, and high performing programming paradigm on which users can base their work. The new models are called Checkpoint-on-Failure (CoF) and User Level Failure Mitigation (ULFM).

III. Related Work

The current industry standard for failure handling is rollback recovery with periodic checkpoints to disk, and many libraries have been implemented to support this behavior [13], [19], [20], [23]. This has been an effective recovery model for many years and continues to be an area of research to prolong its usefulness [2], [7], [14], despite the increase in fault frequency and the hierarchization of the underlying hardware architecture. However as machines continue to scale, concerns have been raised about the scalability of rollback recovery [9]. The time spent performing recovery operations is expected to exceed the MTBF in the next generation of supercomputers, causing any large-scale applications to enter a cycle of recovery where no useful computation occurs.

In February 2012, the Department of Defense and Department of Energy conducted research [11] to outline the necessity for resilience at extreme scale, specifically for exascale computing. They affirmed the likelihood that exascale systems will have a shorter MTBF than existing systems.

[1] http://www.top500.org

They also recommended that "a 'light-weight' approach, i.e., effective and easy to implement, is preferable to a 'full-featured' alternative." By attempting fewer features, each library can focus on creating a scalable and efficient implementation while promoting simplicity and reducing unnecessary features. The agencies also discussed possible redundancy approaches (discussed in [6], [16]), describing them as "not practical because they require 2-3x more energy". The price of redundancy must not only include the obvious loss of computing time from executing duplicate codes across multiple processors, but also the up-front costs of purchasing extra hardware on which to execute the redundancy.

Out of these problems have arisen a new class of algorithms that allow different methods of resilience. These algorithms support what is called Algorithm Based Fault Tolerance (ABFT). They have been designed to support a recovery model that does not require all processes to participate in the recovery simultaneously or, perhaps, ever at all. Huang and Abraham first explored these algorithms [17] for soft errors in systolic arrays, but research has expanded to create ABFT techniques for many algorithms [12], [21].

To support these new algorithms, applications require support from their libraries. One of the most popular communication libraries used by large scale applications is the Message Passing Interface (MPI), the contents of which are determined by the MPI Forum. The MPI Forum has previously considered proposals to add fault tolerance capabilities to the MPI Standard. In 2011, a proposal was brought forth [18] titled "Run-Through Stabilization" which discussed a wide range of fault tolerance capabilities to provide for users by defining the behavior of MPI following a process failure as well as introducing new constructs to provide strong consensus between processes. The MPI Forum decided to reject the proposal, but the work led to more research in the field of resilience in MPI, including the work being proposed in Section V.

Outside of the MPI Forum, work has been done to provide fault tolerance to MPI applications. FT-MPI [15] was an MPI-1 compliant implementation of the MPI Standard which added new capabilities for fault tolerance. It included automatic recovery models for failures including: shrinking MPI communicators to automatically remove failed processes, leaving holes in the communicators while allowing communication to continue, and destroying and rebuilding the communicators to retain communication patterns and groups. FT-MPI was never adopted into the MPI Standard but continued as a research project for many years.

IV. Checkpoint-on-Failure

Checkpoint-on-Failure (CoF) [5] combines the familiarity of checkpointing with the flexibility of ABFT. As discussed in Section III, coordinated checkpointing is no longer considered a scalable option for fault tolerance due to the increasing frequency and size of the individual checkpoints. However, for some types of applications which can take advantage of it, CoF can reduce the necessary number of checkpoints. In the failure-free case, no checkpoints are made, and in all other cases, only the optimal number of checkpoints are required.

The current MPI Standard (version 2.2 [1]) intentionally does not provide a guideline for the behavior of the MPI library after a process failure. Section 2.8 states: "*MPI does not provide mechanisms for dealing with failures in the communication system. [...] Whenever possible, such failures will be reflected as errors in the relevant communication call. Similarly, MPI itself provides no mechanisms for handling processor failures.*" Later, in the same section: "*This document does not specify the state of a computation after an erroneous MPI call has occurred. The desired behavior is that a relevant error code be returned, and the effect of the error be localized to the greatest possible extent.*" Using these two sections from the standard, it is possible to construct a minimal failure handling method using roll-forward recovery.

Section 8.3 (Error Handling) defines the behavior of a "high quality" implementation pertaining to error handling: "*A good quality implementation will, to the greatest possible extent, circumscribe the impact of an error, so that normal processing can continue after an error handler was invoked.*" By using a "high quality" MPI implementation as defined here, an application will receive a notification from the MPI library through a custom `MPI_ERRHANDLER` when a failure occurs and before the library shuts down operation. MPI makes no guarantee about the availability of the communication library following a process failure, therefore standard roll-forward recovery mechanisms are no longer possible. However, by using CoF methods, an application still has an opportunity to save its state and exit gracefully, then be restarted with the original number of processes and perform recovery.

The usage of the CoF model can be described as follows. When a failure eliminates a process, other processes are notified and control is returned from ongoing MPI calls. Surviving processes assume the MPI library is dysfunctional as described in the MPI Standard and make no further calls to the MPI library (in particular, they do not yet undergo ABFT recovery). Instead, they checkpoint their current state independently and abort. When all processes have exited, the job is usually automatically terminated, and the user (or a managing script, batch scheduler, runtime support system, etc.) can launch a new MPI application, which reloads processes from checkpoint. In the new application, the MPI library is functional and communication is once again possible; the ABFT recovery procedure is called to restore the data of the process(es) that did not restart with a checkpoint (because they were the processes which failed). When the global state has been repaired by the

ABFT procedure, the application is ready to resume normal execution.

CoF has many advancements beyond traditional periodic checkpointing. First, it is always optimized both in space and time. It takes checkpoints only after a failure has occurred, meaning that there is no need to calculate an optimal checkpoint interval [10] or pause failure-free execution to write state to disk. This can provide a measurable speedup in the application performance, especially at scale because it reduces resource contention for the I/O system. Also, because checkpoints are only written by processes which are known to be alive, the checkpoint can be written to local stable storage rather than requiring I/O be communicated between a remote, safe data store. The overhead of CoF is also comparable to the standard overhead introduced by other ABFT techniques. The application might need to do extra computation, even in the absence of failures, to maintain internal redundancy used to recover data damaged by failures, however, this is necessary for all ABFT techniques.

A. MPI Requirements for Checkpoint-on-Failure

Regaining Control After Failures: In most MPI implementations, `MPI_ERRORS_ABORT` is the default (and often, only functional) error handler. However, the MPI Standard also defines the error handler, `MPI_ERRORS_RETURN` and provides mechanisms for custom error handlers defined by the application. To support CoF, the MPI library should never deadlock because of failures, but invoke the error handler, at least on processes doing direct communications with the failed process. The error handler takes care of cleaning up at the library level and returns control to the application.

Termination After Checkpoint: A process that detects a failure ceases to use MPI. It checkpoints to storage, local or remote, and exits without calling `MPI_FINALIZE` (it may choose to call `MPI_COMM_ABORT`, but this behavior is not required). Exiting without calling `MPI_FINALIZE` is an error from the MPI perspective, hence the failure notification becomes fully propagated and MPI eventually calls the error handler on all processes, which trigger their own checkpoint procedure and termination.

B. Open MPI Implementation

Open MPI is an MPI 2.2 implementation architected such that it contains two main levels, the runtime (ORTE) and the MPI library (OMPI). As with most MPI library implementations, the default behavior of Open MPI is to abort after a process failure. This policy was implemented in the runtime system, preventing any kind of decision from the MPI layer or the user level. The major change required by the CoF protocol was to make the runtime system resilient, leaving the policy for failure handling to the MPI library, and ultimately the user application.

Resilient Runtime: The ORTE runtime layer provides an out-of-band (OOB) communication mechanism that relays messages based on a routing policy. Node failures not only impact the MPI communications, but also disrupt routing at the OOB level. The default routing policy in ORTE has been amended to allow for self-healing behaviors; this effort is not entirely necessary, but it avoids the significant downtime imposed by a complete redeployment of the parallel job with resubmission into execution queues. The underlying OOB topology is automatically updated to route around failed processes. In some routing topologies, such as a star, this is a trivial operation and only requires excluding the failed process from the routing tables. For more elaborate topologies, such as trees, the healing operation involves computing the closest neighbors in the direction of the failed process and reconnecting the topology through them. The repaired topology is not rebalanced, which could result in improved performance but would disrupt expected communication patterns, however it does retain complete functionality to allow the MPI implementation to maintain internal communication and facilitate failure notification. Although in-flight messages that were currently "hopping" through the failed processes are lost, other in-flight messages are safely routed by the repaired topology. Thanks to self-healing topologies, the runtime remains responsive, even when MPI processes leave.

Failure Notification: The runtime has been further augmented with a failure detection service. To track the status of the failures, an incarnation number has been included in the process names. Following a failure, the name of the failed process (including the incarnation number) is broadcasted over the OOB topology. By including this incarnation number, we can identify transient process failures, prevent duplicate detections, and track message status. ORTE processes monitor the health of their neighbors in the OOB routing topology. Process failure detection relies on a resilient broadcast operation that overlays on the OOB topology. However, the underlying OOB routing algorithm has a significant influence on failure detection and propagation time, as the experiments will show. On each node, the ORTE runtime layer forwards failure notifications to the MPI layer, which has been modified to invoke the appropriate MPI error handler.

V. User Level Failure Mitigation

While CoF can be useful for many types of applications which aim to include some level of fault tolerance in their execution, many other applications could benefit from a more complete fault tolerance solution that allows their application to continue normal execution beyond a process failure. For these applications, we propose an addition to the MPI Standard titled User Level Failure Mitigation (ULFM).

When constructing ULFM, we had three primary requirements of the proposal. In all correct applications, the MPI

library must not uncontrollably deadlock due to process failure. The proposal makes no guarantees about incorrect MPI codes being deadlock-free, but the library must provide tools that, whenever possible, can help resolve deadlocks. Furthermore, for most simple cases, ULFM should be understandable and should solve the needs of applications. For more complex applications, ULFM should be flexible and extensible enough to allow other fault tolerance models to build upon the proposal.

The last two goals, simplicity and extensibility, have always been foundational ideas of the MPI Standard. The standard has always provided the ability for library extensions to allow developers to add their own MPI constructs. We encourage this behavior to give developers more freedom to construct their own fault tolerance libraries which provide more complex consistency levels than ULFM provides.

To this end, the following set of MPI functions is considered to be the minimum necessary to achieve our goals:

`MPI_COMM_FAILURE_ACK` & `MPI_COMM_FAILURE_GET_ACKED`: These two calls allow the application to determine which processes within a communicator have failed. The acknowledgement function serves to mark a point in time which will be used as a reference. The function to get the acknowledged failures refers back to this reference point and returns the group of processes which were locally known to have failed. After acknowledging failures, the application can resume `MPI_ANY_SOURCE` point-to-point operations between non-failed processes, but operations involving failed processes (such as collective operations) will likely continue to raise errors.

`MPI_COMM_REVOKE`: Because failure detection is not global to the communicator, some processes may raise an error for an operation, while others may not. This inconsistency in error reporting may result in some processes continuing their normal, failure-free execution path, while others have diverged to the recovery execution path. As an example, if a process, unaware of the failure, posts a reception from another process that has switched to the recovery path, the matching send will never be posted. Yet no failed process participates in the operation and therefore it will not raise an error. The receive operation is effectively deadlocked. The `MPI_COMM_REVOKE` operation provides a mechanism for the application to resolve such situations before entering the recovery path. A revoked communicator can not be used for further communication, and all future or pending communications on this communicator will be interrupted and completed with the new error code `MPI_ERR_REVOKED`. It is notable that although this operation is not collective (a process will enter it alone), it affects remote ranks without a matching call.

`MPI_COMM_SHRINK`: The shrink operation allows the application to create a new, working communicator by removing all failed processes from a revoked communicator.

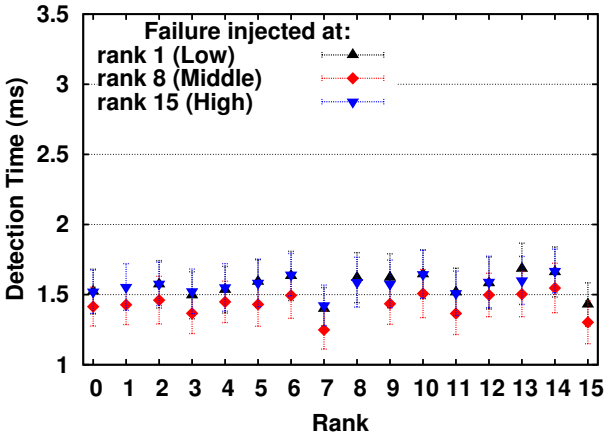

Figure 1. Failure detection time, sorted by process rank, depending on the OOB overlay network used for failure propagation.

The operation is collective and executes a consensus algorithm to ensure that all participating processes complete the operation with equivalent groups in the new communicator. This function cannot return an error due to process failure. Instead, such failures are absorbed as part of the consensus algorithm and will be excluded from the new communicator.

`MPI_COMM_AGREE`: This operation provides an agreement algorithm which can be used to determine a consistent state between processes when such strong consistency is necessary. The function is collective and forms an agreement over a boolean value, even when failures have happened. The agreement can be used to resolve a number of consistency issues after a failure, such as uniform completion of an algorithmic phase or collective operation, or as a key building block for strongly consistent failure handling approaches (such as transactional fault tolerance).

VI. Early Results

This section will demonstrate some of the first results of the CoF and ULFM work. Results are from the "Dancer" test platform unless noted otherwise. Dancer is a 16-node cluster where each node consists of two 2.27GHz quad-core Intel E5520 CPUs, with 20GB/s Infiniband interconnect.

A. Runtime Results

Our changes to the Open MPI Runtime Environment (ORTE) demonstrate the low amount of overhead introduced in failure free execution as well as fast failure detection when failures do occur.

We created a micro-benchmark to measure failure detection time on different MPI processes. The benchmark uses an `MPI_BARRIER` to roughly synchronize the processes, stores the reference start time, then enters a ring algorithm while injecting a failure at a predetermined location. When the failure is detected, MPI stores the detection time to determine a relative length for the failure detection. Figure 1

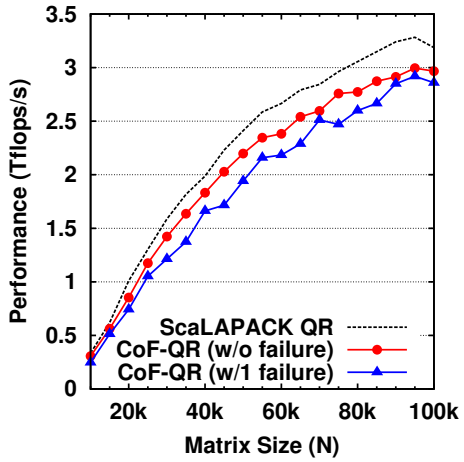

Figure 2. CoF QR on Kraken (Lustre)

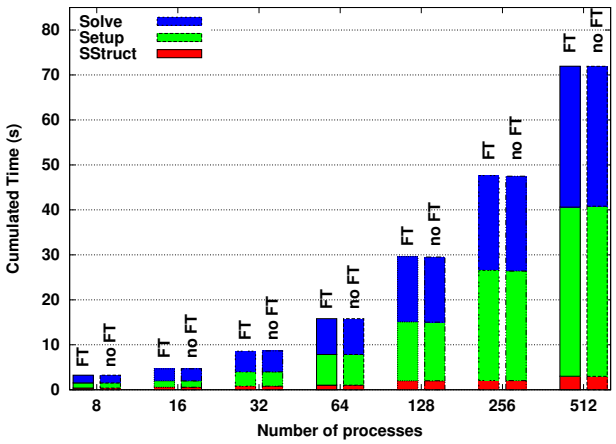

Figure 3. Comparison of the vanilla and ULFM versions of Open MPI running Sequoia-AMG at different scales (Smoky).

demonstrates the detection time for failures in three different locations in the system, ranks 1 (low), 8 (middle), and 15 (high). It demonstrates a binomial OOB topology which allows for fast, scalable failure detection due to its tree-based topology. The experiment shows an average of 20 runs with error bars showing the standard deviation of the runs.

B. CoF Results

To demonstrate the usefulness of the CoF work, Peng Du implemented a CoF compatible version of a QR factorization. Details of the CoF-QR can be found in [5]. Figure 2 presents the performance of the CoF-QR code on the Kraken supercomputer. This test was run on a process grid of 24x24 with a block size of 100. The figure illustrates both the failure-free and failure-resilient executions compared to the reference ScaLAPACK implementation.

The result of this experiment demonstrates the feasibility of CoF as a recovery scheme for some ABFT applications. The CoF-QR code was able to achieve performance results of at least 90% of the reference ScaLAPACK implementation, even in the presence of failures, with even better results in the failure-free case. This shows that compared to a possible performance loss of 25% or more with traditional checkpointing, CoF can significantly improve runtime.

C. ULFM Results

The ULFM proposal is in its early stages of implementation and testing, however there have been some comparisons between ULFM-compatible MPI and 2.2-Standard MPI to measure the overhead introduced by the new requirements. These tests were performed on the Smoky system at Oak Ridge National Laboratory. Each node contains four quad-core 2.0 GHz AMD Opteron processors with 2 GB of memory per compute core. Nodes are connected with gigabit Ethernet and InfiniBand. Some shared-memory benchmarks were conducted on Romulus, a 6×8 AMD Opteron 6180 SE with 256 GB of memory (32 GB per socket) at the University of Tennessee. We compare the vanilla version of Open MPI (r26237) with the ULFM enabled version.

Experiments using the NetPIPE benchmark have demonstrated that for all types of interconnect, the difference between the ULFM implementation performance and the vanilla version of Open MPI is negligible. All of the differences are within the standard deviation of both measurements which shows that any variation is as likely to be caused by network noise as implementation changes.

To measure the impact of the prototype on a real application, we used the Sequoia AMG benchmark[2]. This MPI intensive benchmark is an Algebraic Mult-Grid (AMG) linear system solver for unstructured mesh physics. A weak scaling study was conducted up to 512 processes following the problem *Set 5*. In Figure 3, we compare the time slicing of three main phases (Solve, Setup, and SStruct) of the benchmark, with, side by side, the vanilla version of the Open MPI implementation, and the ULFM enabled one. The application itself is not fault tolerant and does not use the features proposed in ULFM. The goal of this benchmark is to demonstrate that a careful implementation of the proposed semantic does not impact the performance of the MPI implementation, and ultimately leaves the behavior and performance of legacy applications unchanged. The results show that the performance difference is negligible.

Both of these results are encouraging for future work along the ULFM path. They provide proof that an implementation of the new constructs for ULFM does not need to cause an increase in overhead generated by the MPI implementation. This should lead to greater adoption of the ULFM proposal by multiple MPI implementations.

[2]https://asc.llnl.gov/sequoia/benchmarks/#amg

VII. Conclusion and Ongoing Work

To support exascale executions, applications must now consider resilience in their designs. Current fault tolerant models, including rollback recovery, are not expected to remain productive at extreme scales. Other efforts to introduce scalable resilience have not seen ubiquitous adoption due, in part, to their high barrier to entry from the large code additions they require, or the new levels of complexity which they introduce. Balancing the needs of usability and scalability in resilience has led to the work presented here.

This work has presented two new programming models to support resiliency in parallel computing when using the Message Passing Interface. Checkpoint-on-Failure shows how the current MPI standard can be used to create a resilient application by taking advantage of a "high quality implementation" of the MPI Standard. When failures occur, the application performs local checkpoints, restarts MPI, loads its local checkpoints, performs Algorithm Based Fault Tolerance recovery procedures to regenerate any lost data and continues application execution. For more complete failure handling, User Level Failure Mitigation provides extensions to the MPI Standard which give the application the ability to continue execution after a process failure. By repairing communicators when necessary, the application does not need to roll back to a checkpoint but can continue to run to completion.

These two new resilient programming models provide many useful new tools for developers to enhance their applications. More investigation is necessary to measure the exact impact of these tools on application performance. ULFM will also continue to be presented to the MPI Forum with the intention of eventually being adopted as a new chapter in the MPI Standard so developers will have a truly portable resilient parallel programming library. There are still challenges to overcome before the standardization process can take place. ULFM must be proven to not impact performance on a wide variety of communication environments. Also, it will be applied to more real world applications to prove that it can be helpful in a production environment. However, when the process is finished, it will provide many opportunities for new types of applications which were previously either impossible or difficult to architect.

VIII. Related Publications

- An Evaluation of User-Level Failure Mitigation Support in MPI, EuroMPI 2012 [4]
- A Checkpoint-on-Failure Protocol for Algorithm-Based Recovery in Standard MPI, Distinguished Paper Award, Euro-Par 2012 [5]
- Enabling Application Resilience With and Without the MPI Standard, CCGrid 2012 [3]

REFERENCES

[1] M. F. 2.2, *MPI: A Message-Passing Interface Standard Version 2.2*. [Online]. Available: http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf

[2] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka, "FTI: High performance Fault Tolerance Interface for hybrid systems," in *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM Request Permissions, Nov. 2011, pp. 1–12.

[3] W. Bland, "Enabling Application Resilience with and without the MPI Standard," *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pp. 746–751, 2012.

[4] 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 *19th EuroMPI*, J. L. Träff, S. Benkner, and J. Dongarra, Eds. Vienna, Austria: Springer, Sep. 2012.

[5] W. Bland, P. Du, A. Bouteiller, T. Herault, G. Bosilca, and J. J. Dongarra, "A checkpoint-on-failure protocol for algorithm-based recovery in standard MPI," in *18th Euro-Par*. Rhodes Island, Greece: Springer, Aug. 2012.

[6] G. Bosilca, A. Bouteiller, É. Brunet, F. Cappello, J. Dongarra, A. Guermouche, T. Herault, Y. Robert, F. Vivien, and D. Zaidouni, "Unified Model for Assessing Checkpointing Protocols at Extreme-Scale," Innovative Computing Laboratory - ICL , Département Informatique - INF , GRAND-LARGE - INRIA Saclay - Ile de France , Joint Laboratory for Petascale Computing [Illinois] - JLPC , Laboratoire de Recherche en Informatique - LRI , ROMA - ENS Lyon / CNRS / Inria Grenoble Rhône-Alpes , Laboratoire de l'Informatique du Parallélisme - LIP, Tech. Rep., May 2012.

[7] D. Buntinas, C. Coti, T. Herault, P. Lemarinier, L. Pilard, A. Rezmerita, E. Rodriguez, and F. Cappello, "Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI Protocols," *Future Generation Computer Systems*, vol. 24, no. 1, pp. 73–84, 2008.

[8] F. Cappello, "Fault Tolerance in Petascale/ Exascale Systems: Current Knowledge, Challenges and Research Opportunities," *International Journal of High Performance Computing Applications*, vol. 23, no. 3, pp. 212–226, Jul. 2009.

[9] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir, "Toward Exascale Resilience," *International Journal of High Performance Computing Applications*, vol. 23, no. 4, pp. 374–388, Oct. 2009.

[10] J. T. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *Future Generation Computer Systems*, vol. 22, no. 3, pp. 303–312, Feb. 2006.

[11] J. Daly, B. Harrod, T. Hoang, and L. Nowell, "Inter-Agency Workshop on HPC Resilience at Extreme Scale," National Security Agency, Advanced Computing Systems, Tech. Rep., Feb. 2012.

[12] P. Du, A. Bouteiller, G. Bosilca, T. Herault, and J. Dongarra, "Algorithm-based fault tolerance for dense matrix factorizations," in *PPoPP '12: Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*. ACM Request Permissions, Feb. 2012.

[13] J. Duell, "The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart," Tech. Rep. LBNL-54941, Dec. 2002.

[14] E. Elnozahy, L. Alvisi, Y. Wang, and D. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Computing Surveys (CSUR)*, vol. 34, no. 3, pp. 375–408, Jan. 2002.

[15] G. E. Fagg and J. J. Dongarra, "FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, J. Dongarra, P. Kacsuk, and N. Podhorszki, Eds. Springer Berlin / Heidelberg, 2000, pp. 346–353.

[16] K. Ferreira, J. Stearley, J. Laros, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. Bridges, and D. Arnold, "Evaluating the viability of process replication reliability for exascale systems," in *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*. ACM Request Permissions, 2011, pp. 1–12.

[17] K.-H. Huang and J. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations," *IEEE Transactions on Computers*, no. 6, pp. 518–528, 1984.

[18] J. Hursey, R. Graham, G. Bronevetsky, D. Buntinas, H. Pritchard, and D. Solt, "Run-Through Stabilization: An MPI Proposal for Process Fault Tolerance," in *Lecture Notes in Computer Science*, Y. Cotronis, A. Danalis, D. Nikolopoulos, and J. Dongarra, Eds. Springer Berlin / Heidelberg, 2011, pp. 329–332.

[19] M. Litzkow, T. Tannenbaum, and J. Basney, "Checkpoint and migration of UNIX processes in the Condor distributed processing system," Tech. Rep., 1997.

[20] J. S. Plank, M. Beck, and G. Kingsley, "Libckpt: Transparent checkpointing under unix," Knoxville, TN, Tech. Rep., 1994.

[21] J. Plank, Y. Kim, and J. Dongarra, "Algorithm-based diskless checkpointing for fault tolerant matrix operations," *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on*, pp. 351–360, 1995.

[22] B. Schroeder and G. A. Gibson, "Understanding Failures in Petascale Computers," *SciDAC, Journal of Physics: Conference Series*, vol. 78, 2007.

[23] H. Zhong and J. Nieh, "CRAK: Linux checkpoint/restart as a kernel module," Tech. Rep., Nov. 2001.