A Checkpoint-on-Failure Protocol for Algorithm-Based Recovery in Standard MPI

Wesley Bland, Peng Du, Aurelien Bouteiller, Thomas Herault, George Bosilca, and Jack J. Dongarra

Innovative Computing Laboratory, University of Tennessee
1122 Volunteer Blvd., Knoxville, TN 37996-3450, USA
{bland, du, bouteill, herault, bosilca, dongarra}@eecs.utk.edu

Abstract. Most predictions of Exascale machines picture billion way parallelism, encompassing not only millions of cores, but also tens of thousands of nodes. Even considering extremely optimistic advances in hardware reliability, probabilistic amplification entails that failures will be unavoidable. Consequently, software fault tolerance is paramount to maintain future scientific productivity. Two major problems hinder ubiquitous adoption of fault tolerance techniques: 1) traditional checkpoint based approaches incur a steep overhead on failure free operations and 2) the dominant programming paradigm for parallel applications (the MPI standard) offers extremely limited support of software-level fault tolerance approaches. In this paper, we present an approach that relies exclusively on the features of a high quality implementation, as defined by the current MPI standard, to enable algorithmic based recovery, without incurring the overhead of customary periodic checkpointing. The validity and performance of this approach are evaluated on large scale systems, using the QR factorization as an example.

1 Introduction

The insatiable processing power needs of domain science has pushed High Performance Computing (HPC) systems to feature a significant performance increase over the years, even outpacing "Moore's law" expectations. Leading HPC systems, whose architectural history is listed in the Top500 [1] ranking, illustrate the massive parallelism that has been embraced in the recent years; current number 1 – the K-computer – has half a million cores, and even with the advent of GPU accelerators, it requires no less than 73,000 cores for the Tsubame 2.0 system (#5) to breach the Petaflop barrier. Indeed, the International Exascale Software Project, a group created to evaluate the challenges on the path toward Exascale, has published a public report outlining that a massive increase in scale will be necessary when considering probable advances in chip technology, memory and interconnect speeds, as well as limitations in power consumption and thermal envelope [1]. According to these projections, as early as 2014, billion way parallel machines, encompassing millions of cores, and tens of thousands of nodes,

[1] www.top500.org

will be necessary to achieve the desired level of performance. Even considering extremely optimistic advances in hardware reliability, probabilistic amplification entails that failures will be unavoidable, becoming common events. Hence, fault tolerance is paramount to maintain scientific productivity.

Already, for Petaflop scale systems the issue has become pivotal. On one hand, the capacity type workload, composed of a large amount of medium to small scale jobs, which often represent the bulk of the activity on many HPC systems, is traditionally left unprotected from failures, resulting in diminished throughput due to failures. On the other hand, selected capability applications, whose significance is motivating the construction of supercomputing systems, are protected against failures by ad-hoc, application-specific approaches, at the cost of straining engineering efforts, translating into high software development expenditures. Traditional approaches based on periodic checkpointing and rollback recovery, incurs a steep overhead, as much as 25% [2], on failure-free operations. Forward recovery techniques, most notably Algorithm-Based Fault Tolerant techniques (ABFT), are using mathematical properties to reconstruct failure-damaged data and do exhibit significantly lower overheads [3]. However, and this is a major issue preventing their wide adoption, the resiliency support ABFT demands from the MPI library largely exceeds the specifications of the MPI standard [4] and has proven to be an unrealistic requirement, considering that only a handful of MPI implementations provide it.

The current MPI-2 standard leaves open an optional behavior regarding failures to qualify as a "high quality implementation." According to this specification, when using the MPI_ERRORS_RETURN error handler, the MPI library should return control to the user when it detects a failure. In this paper, we propose the idea of Checkpoint-on-Failure (CoF) as a minimal impact feature to enable MPI libraries to support forward recovery strategies. Despite the default application-wide abort action that all notable MPI implementations undergo in case of a failure, we demonstrate that an implementation that enables CoF is simple and yet effectively supports ABFT recovery strategies that completely avoid costly periodic checkpointing.

The remainder of this paper is organized as follows. The next section presents typical fault tolerant approaches and related works to discuss their requirements and limitations. Then in Section 3 we present the CoF approach, and the minimal support required from the MPI implementation. Section 4 presents a practical use case: the ABFT QR algorithm and how it has been modified to fit the proposed paradigm. Section 5 presents an experimental evaluation of the implementation, followed by our conclusions.

2 Background & Related Work

Message passing is the dominant form of communication used in parallel applications, and MPI is the most popular library used to implement it. In this context, the primary form of fault tolerance today is rollback recovery with periodical checkpoints to disk. While this method is effective in allowing applications

to recover from failures using a previously saved state, it causes serious scalability concerns [5]. Moreover, periodic checkpointing requires precise heuristics for fault frequency to minimize the number of superfluous, expensive protective actions [6–10]. In contrast, the work presented here focuses on enabling *forward recovery*. Checkpoint actions are taken only *after* a failure is detected; hence the checkpoint interval is optimal by definition, as there will be one checkpoint interval per effective fault.

Forward recovery leverages algorithms properties to complete operations despite failures. In Naturally Fault Tolerant applications, the algorithm can compute the solution while totally ignoring the contributions of failed processes. In ABFT applications, a recovery phase is necessary, but failure damaged data can be reconstructed only by applying mathematical operations on the remaining dataset [11]. A recoverable dataset is usually created by initially computing redundant data, dispatched so as to avoid unrecoverable loss of information from failures. At each iteration, the algorithm applies the necessary mathematical transformations to update the redundant data (at the expense of more communication and computation). Despite great scalability and low overhead [3, 12], the adoption of such algorithms has been hindered by the requirement that the support environment must continue to consistently deliver communications, even after being crippled by failures.

The current MPI Standard (MPI-2.2, [4]) does not provide significant help to deal with the required type of behavior. Section 2.8 states in the first paragraph: "*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.*" Failures, be they due to a broken link or a dead process, are considered resource errors. 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.*" So, for the current standard, process or communication failures are to be handled as errors, and the behavior of the MPI application, after an error has been returned, is left unspecified by the standard. However, the standard does not prevent implementations from going beyond its requirements, and on the contrary, encourages high-quality implementations *to return* errors, once a failure is detected. Unfortunately, most of the implementations of the MPI Standard have taken the path of considering process failures as unrecoverable errors, and the processes of the application are most often killed by the runtime system when a failure hits any of them, leaving no opportunity for the user to mitigate the impact of failures.

Some efforts have been undertaken to enable ABFT support in MPI. FT-MPI [13] was an MPI-1 implementation which proposed to change the MPI semantic to enable repairing communicators, thus re-enabling communications for applications damaged by failures. This approach has proven successful and applications have been implemented using FT-MPI. However, these modifica-

tions were not adopted by the MPI standardization body, and the resulting lack of portability undermined user adoption for this fault tolerant solution.

In [14], the authors discuss alternative or slightly modified interpretations of the MPI standard that enable some forms of fault tolerance. One core idea is that process failures happening in another MPI world, connected only through an inter-communicator, should not prevent the continuation of normal operations. The complexity of this approach, for both the implementation and users, has prevented these ideas from having a practical impact. In the CoF approach, the only requirement from the MPI implementation is that it does not forcibly kill the living processes without returning control. No stronger support from the MPI stack is required, and the state of the library is left undefined. This simplicity has enabled us to actually implement our proposition, and then experimentally support and evaluate a real ABFT application. Similarly, little effort would be required to extend MPICH-2 to support CoF (see Section 7 of the Readme[2]).

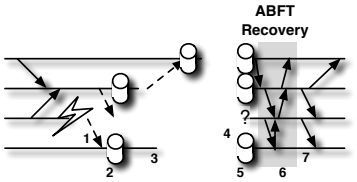
3 Enabling Algorithm-Based Fault Tolerance in MPI

3.1 The Checkpoint-on-Failure Protocol

In this paper, we advocate that an extremely efficient form of fault tolerance can be implemented, strictly based on the MPI standard, for applications capable of taking advantage of forward recovery. ABFT methods are an example of forward recovery algorithms, capable of restoring missing data from redundant information located on other processes. This forward recovery step requires communication between processes, and we acknowledge that, in light of the current standard, requiring the MPI implementation to maintain service after failures is too demanding. However, a high-quality MPI library should at least allow the application to regain control following a process failure. We note that this control gives the application the opportunity to save its state and exit gracefully, rather than the usual behavior of being aborted by the MPI implementation.

Algorithm 1 The Checkpoint-on-Failure Protocol

1. MPI returns an error on surviving processes
2. Surviving processes checkpoint
3. Surviving processes exit
4. A new MPI application is started
5. Processes load from checkpoint (if any)
6. Processes enter ABFT dataset recovery
7. Application resumes



Based on these observations, we propose a new approach for supporting ABFT applications, called Checkpoint-on-Failure (CoF). Algorithm 1 presents

[2] http://www.mcs.anl.gov/research/projects/mpich2/documentation/files/mpich2-1.4.1-README.txt

the steps involved in the CoF method. In the associated explanatory figure, horizontal lines represent the execution of processes in two successive MPI applications. When a failure eliminates a process, other processes are notified and regain control from ongoing MPI calls (1). Surviving processes assume the MPI library is dysfunctional and do not further call MPI operations (in particular, they do not yet undergo ABFT recovery). Instead, they checkpoint their current state independently and abort (2, 3). When all processes exited, the job is usually terminated, but the user (or a managing script, batch scheduler, runtime support system, etc.) can launch a new MPI application (4), which reloads processes from checkpoint (5). In the new application, the MPI library is functional and communications possible; the ABFT recovery procedure is called to restore the data of the process(es) that could not be restarted from checkpoint (6). When the global state has been repaired by the ABFT procedure, the application is ready to resume normal execution.

Compared to periodic checkpointing, in CoF, a process pays the cost of creating a checkpoint only when a failure, or multiple simultaneous failures have happened, hence an optimal number of checkpoints during the run (and no checkpoint overhead on failure-free executions). Moreover, in periodic checkpointing, a process is protected only when its checkpoint is stored on safe, remote storage, while in CoF, local checkpoints are sufficient: the forward recovery algorithm reconstructs datasets of processes which cannot restart from checkpoint. Of course, CoF also exhibits the same overhead as the standard ABFT approach: the application might need to do extra computation, even in the absence of failures, to maintain internal redundancy (whose degree varies with the maximum number of simultaneous failures) used to recover data damaged by failures. However, ABFT techniques often demonstrate excellent scalability; for example, the overhead on failure-free execution of the ABFT QR operation (used as an example in Section 4) is inversely proportional to the number of processes.

3.2 MPI Requirements for Checkpoint-on-Failure

Returning Control over Failures: In most MPI implementations, MPI_ERRORS_ABORT is the default (and often, only functional) error handler. However, the MPI standard also defines the MPI_ERRORS_RETURN handler. 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 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 only checkpoints on some storage and exits without calling MPI_Finalize. Exiting without calling MPI_Finalize is an error from the MPI perspective, hence the failure cascades and MPI eventually returns with a failure notification on every process, which triggers their own checkpoint procedure and termination.

3.3 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 requested by the CoF protocol was to make the runtime system resilient, and leave the decision in case of failure to the MPI library policy, and ultimately to the user application.

Failure Resilient Runtime: The ORTE runtime layer provides an out-of-band communication mechanism (OOB) 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 the Open MPI runtime 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 in 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 a binomial tree, 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, resulting in degraded performance but complete functionality after failures. Although in-flight messages that were currently "hopping" through the failed processes are lost, other in-flight messages are safely routed on the repaired topology. Thanks to self-healing topologies, the runtime remains responsive, even when MPI processes leave.

Failure Notification: The runtime has been 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. Detection of other processes rely on a failure resilient broadcast that overlays on the OOB topology. This algorithm has a low probability of creating a bi-partition of the routing topology, hence ensuring a high accuracy of the failure detector. 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.

4 Example: the QR Factorization

In this section, we propose to illustrate the applicability of CoF by considering a representative routine of a widely used class of algorithms: dense linear factorizations. The QR factorization is a cornerstone building block in many applications, including solving $Ax = b$ when matrices are ill-conditioned, computing eigenvalues, least square problems, or solving sparse systems through the GMRES iterative method. For an $M \times N$ matrix A, the QR factorization produces Q and R, such that $A = QR$ and Q is an $M \times M$ orthogonal matrix and R is an $M \times N$ upper triangular matrix. The most commonly used implementation of the QR algorithm on a distributed memory machine comes from the ScaLAPACK linear algebra library [15], based on the block QR algorithm. It uses a 2D block-cyclic distribution for load balance, and is rich in level 3 BLAS operations, thereby achieving high performance.

4.1 ABFT QR Factorization

In the context of FT-MPI, the ScaLAPACK QR algorithm has been rendered fault tolerant through an ABFT method in previous works [12]. This ABFT algorithm protects both the left (Q) and right (R) factors from fail-stop failures at any time during the execution. At the time of failure, every surviving process is notified by FT-MPI. FT-MPI then spawns a replacement process that takes the same grid coordinates in the $P \times Q$ block-cyclic distribution. Missing checksums are recovered from duplicates, a reduction collective communication recovers missing data blocks in the right factor from checksums. The left factor is protected by the Q-parallel panel checksum, it is either directly recovered from checksum, or by recomputing the panels in the current Q-wide section (see [12]). Although this algorithm is fault tolerant, it requires continued service from the MPI library after failures – which is a stringent requirement that can be waived with CoF.

4.2 Checkpoint-on-Failure QR

Checkpoint Procedure: Compared to a regular ABFT algorithm, CoF requires a different checkpoint procedure. System-level checkpointing is not applicable, as it would result in restoring the state of the broken MPI library upon restart. Instead, a custom MPI error handler invokes an algorithm specific checkpoint procedure, which simply dumps the matrices and the value of important loop indices into a file.

State Restoration: A ScaLAPACK program has a deep call stack, layering functions from multiple software packages, such as PBLAS, BLACS, LAPACK and BLAS. In the FT-MPI version of the algorithm, regardless of when the failure is detected, the current iteration of the algorithm must be completed before entering the recovery procedure. This ensures an identical call stack on every process and a complete update of the checksums. In the case of the CoF protocol, failures

interrupt the algorithm immediately, the current iteration cannot be completed due to lack of communication capabilities. This results in potentially diverging call stacks and incomplete updates of checksums. However, because failure notification happens only in MPI, lower level, local procedures (BLAS, LAPACK) are never interrupted.

To resolve the call stack issue, every process restarted from checkpoint undergoes a "dry run" phase. This operation mimics the loop nests of the QR algorithm down to the PBLAS level, without actually applying modifications to or exchanging data. When the same loop indices as before the failure are reached, the matrix content is loaded from the checkpoint; the state is then similar to that of the FT-MPI based ABFT QR after a failure. The regular recovery procedure can be applied: the current iteration of the factorization is completed to update all checksums and the dataset is rebuilt using the ABFT reduction.

5 Performance Discussion

In this section, we use our Open MPI and ABFT QR implementations to evaluate the performance of the CoF protocol. We use two test platforms. The first machine, "Dancer," is a 16-node cluster. All nodes are equipped with two 2.27GHz quad-core Intel E5520 CPUs, with a 20GB/s Infiniband interconnect. Solid State Drive disks are used as the checkpoint storage media. The second system is the Kraken supercomputer. Kraken is a Cray XT5 machine, with 9,408 compute nodes. Each node has two Istanbul 2.6 GHz six-core AMD Opteron processors, 16 GB of memory, and are connected through the SeaStar2+ interconnect. The scalable cluster file system "Lustre" is used to store checkpoints.

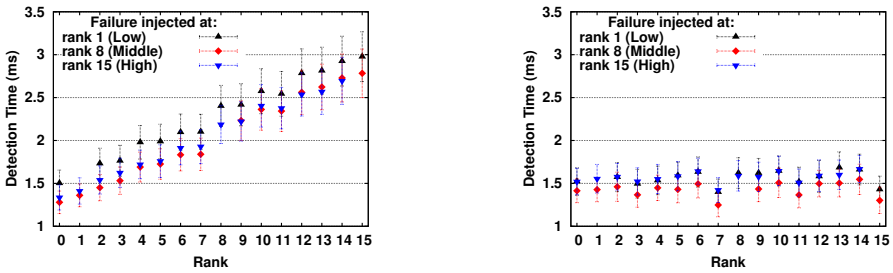
5.1 MPI Library Overhead

One of the concerns with fault tolerance is the amount of overhead introduced by the fault tolerance management additions. Our implementation of fault detection and notification is mostly implemented in the non-critical ORTE runtime. Typical HPC systems feature a separated service network (usually Ethernet based) and a performance interconnect, hence health monitoring traffic, which happens on the OOB service network, is physically separated from the MPI communications, leaving no opportunity for network jitter. Changes to MPI functions are minimal: the same condition that used to trigger unconditional abort has been repurposed to trigger error handlers. As expected, no impact on MPI bandwidth or latency was measured (Infiniband and Portals results not shown for lack of space). The memory usage of the MPI library is slightly increased, as the incarnation number doubles the size of process names; however, this is negligible in typical deployments.

5.2 Failure Detection

According to the requirement specified in Section 3.2, only in-band failure detection is required to enable CoF. Processes detecting a failure checkpoint then

exit, cascading the failure to processes communicating with them. However, no recovery action (in particular checkpointing) can take place before a failure has been notified. Thanks to asynchronous failure propagation in the runtime, responsiveness can be greatly improved, with a high probability for the next MPI call to detect the failures, regardless of communication pattern or checkpoint duration.



(a) Linear OOB Routing

(b) Binomial OOB Routing

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

We designed a micro-benchmark to measure failure detection time as experienced by MPI processes. The benchmark code synchronizes with an MPI_Barrier, stores the reference date, injects a failure at a specific rank, and enters a ring algorithm until the MPI error handler stores the detection date. The OOB routing topology used by the ORTE runtime introduces a non-uniform distance to the failed process, hence failure detection time experienced by a process may vary with the position of the failed process in the topology, and the OOB topology. Figure 1(a) and 1(b) present the case of the linear and binomial OOB topologies, respectively. The curves "Low, Middle, High" present the behavior for failures happening at different positions in the OOB topology. On the horizontal axis is the rank of the detecting process, on the vertical axis is the detection time it experienced. The experiment uses 16 nodes, with one process per node, MPI over Infiniband, OOB over Ethernet, an average of 20 runs, and the MPI barrier latency is four orders of magnitude lower than measured values.

In the linear topology (Figure 1(a)) every runtime process is connected to the *mpirun* process. For a higher rank, failure detection time increases linearly because it is notified by the *mpirun* process only after the notification has been sent to all lower ranks. This issue is bound to increase with scale. The binomial tree topology (Figure 1(b)) exhibits a similar best failure detection time. However, this more scalable topology has a low output degree and eliminates most contentions on outgoing messages, resulting in a more stable, lower average detection time, regardless of the failure position. Overall, failure detection time is on the order of milliseconds, a much smaller figure than typical checkpoint time.

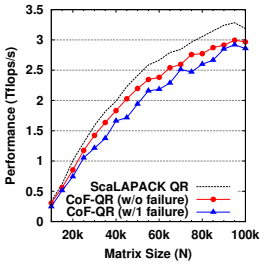


Fig. 2. ABFT QR and one CoF recovery on Kraken (Lustre).

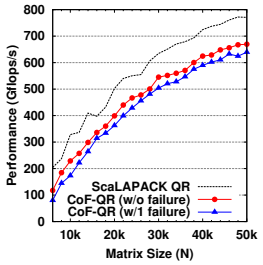


Fig. 3. ABFT QR and one CoF recovery on Dancer (local SSD).

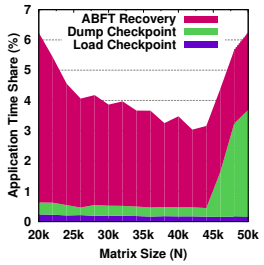


Fig. 4. Time breakdown of one CoF recovery on Dancer (local SSD).

5.3 Checkpoint-on-Failure QR Performance

Supercomputer Performance: Figure 2 presents the performance on the Kraken supercomputer. The process grid is 24×24 and the block size is 100. The CoF-QR (no failure) presents the performance of the CoF QR implementation, in a fault-free execution; it is noteworthy, that when there are no failures, the performance is exactly identical to the performance of the unmodified FT-QR implementation. The CoF-QR (with failure) curves present the performance when a failure is injected after the first step of the PDLARFB kernel. The performance of the non-fault tolerant ScaLAPACK QR is also presented for reference.

Without failures, the performance overhead compared to the regular ScaLAPACK is caused by the extra computation to maintain the checksums inherent to the ABFT algorithm [12]; this extra computation is unchanged between CoF-QR and FT-QR. Only on runs where a failure happened do the CoF protocols undergoe the supplementary overhead of storing and reloading checkpoints. However, the performance of the CoF-QR remains very close to the no-failure case. For instance, at matrix size N=100,000, CoF-QR still achieves 2.86 Tflop/s after recovering from a failure, which is 90% of the performance of the non-fault tolerant ScaLAPACK QR. This demonstrates that the CoF protocol enables efficient, practical recovery schemes on supercomputers.

Impact of Local Checkpoint Storage: Figure 3 presents the performance of the CoF-QR implementation on the Dancer cluster with a 8×16 process grid. Although a smaller test platform, the Dancer cluster features local storage on nodes and a variety of performance analysis tools unavailable on Kraken. As expected (see [12]), the ABFT method has a higher relative cost on this smaller machine. Compared to the Kraken platform, the relative cost of CoF failure recovery is smaller on Dancer. The CoF protocol incurs disk accesses to store and load checkpoints when a failure hits, hence the recovery overhead depends on I/O performance. By breaking down the relative cost of each recovery step in CoF, Figure 4 shows that checkpoint saving and loading only take a small percentage of the total run-time, thanks to the availability of solid state disks on every node. Since checkpoint reloading immediately follows checkpointing, the OS cache satisfy most disk access, resulting in high I/O performance. For matrices larger than

N=44,000, the memory usage on each node is high and decrease the available space for disk cache, explaining the decline in I/O performance and the higher cost of checkpoint management. Overall, the presence of fast local storage can be leveraged by the CoF protocol to speedup recovery (unlike periodic checkpointing, which depends on remote storage by construction). Nonetheless, as demonstrated by the efficiency on Kraken, while this is a valuable optimization, it is not a mandatory requirement for satisfactory performance.

6 Concluding Remarks

In this paper, we presented an original scheme to enable forward recovery using only features of the current MPI standard. Rollback recovery, which relies on periodic checkpointing has a variety of issues. The ideal period of checkpoint, a critical parameter, is particularly hard to assess. Too short a period wastes time and resources on unnecessary Input/Output. Overestimating the period results in dramatically increasing the lost computation when returning to the distant last successful checkpoint. Although Checkpoint-on-Failure involves checkpointing, it takes checkpoint images at optimal times by design: only after a failure has been detected. This small modification enables the deployment of ABFT techniques, without requiring a complex, unlikely to be available MPI implementation that itself survives failures. The MPI library needs only to provide the feature set of a high quality implementation of the MPI standard: the MPI communications may be dysfunctional after a failure, but the library must return control to the application instead of aborting brutally.

We demonstrated, by providing such an implementation in Open MPI, that this feature set can be easily integrated without noticeable impact on communication performance. We then converted an existing ABFT QR algorithm to the CoF protocol. Beyond this example, the CoF protocol is applicable on a large range of applications that already feature an ABFT version (LLT, LU [16], CG [17], etc.). Many master-slave and iterative methods enjoy an extremely inexpensive forward recovery strategy where the damaged domains are simply discarded, and can also benefit from the CoF protocol.

The performance on the Kraken supercomputer reaches 90% of the non-fault tolerant algorithm, even when including the cost of recovering from a failure (a figure similar to regular, non-compliant MPI ABFT). In addition, on a platform featuring node local storage, the CoF protocol can leverage low overhead checkpoints (unlike rollback recovery that requires remote storage).

The MPI standardization body, the MPI Forum, is currently considering the addition of new MPI constructs, functions and semantics to support fault-tolerant applications[3]. While these additions may decrease the cost of recovery, they are likely to increase the failure-free overhead on fault tolerant application performance. It is therefore paramount to compare the cost of the CoF protocol with prospective candidates to standardization on a wide, realistic range of applications, especially those that feature a low computational intensity.

[3] https://svn.mpi-forum.org/trac/mpi-forum-web/wiki/FaultToleranceWikiPage

References

1. J. Dongarra, P. Beckman *et al.*, "The international exascale software roadmap," *IJHPCA*, vol. 25, no. 11, pp. 3–60, 2011.
2. B. Schroeder and G. A. Gibson, "Understanding Failures in Petascale Computers," *SciDAC, Journal of Physics: Conference Series*, vol. 78, 2007.
3. F. Luk and H. Park, "An analysis of algorithm-based fault tolerance techniques," *Journal of Parallel and Distributed Computing*, vol. 5, no. 2, pp. 172–184, 1988.
4. The MPI Forum, "MPI: A Message-Passing Interface Standard, Version 2.2," Tech. Rep., 2099.
5. F. Cappello, A. Geist, B. Gropp, L. V. Kalé, B. Kramer, and M. Snir, "Toward exascale resilience," *IJHPCA*, vol. 23, no. 4, pp. 374–388, 2009.
6. J. W. Young, "A first order approximation to the optimum checkpoint interval," *Commun. ACM*, vol. 17, pp. 530–531, September 1974.
7. E. Gelenbe, "On the optimum checkpoint interval," *JoACM*, vol. 26, pp. 259–270, 1979.
8. J. S. Plank and M. G. Thomason, "Processor allocation and checkpoint interval selection in cluster computing systems," *JPDC*, vol. 61, p. 1590, 2001.
9. J. T. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *Future Gener. Comput. Syst.*, vol. 22, pp. 303–312, February 2006.
10. F. Cappello, H. Casanova, and Y. Robert, "Preventive migration vs. preventive checkpointing for extreme scale supercomputers," *PPL 21:2*, pp. 111–132, 2011.
11. K. Huang and J. Abraham, "Algorithm-based fault tolerance for matrix operations," *Computers, IEEE Transactions on*, vol. 100, no. 6, pp. 518–528, 1984.
12. P. Du, A. Bouteiller, G. Bosilca, T. Herault, and J. Dongarra, "Algorithm-based Fault Tolerance for Dense Matrix Factorizations," in *17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2012.
13. G. Fagg and J. Dongarra, "FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world," *EuroPVM/MPI*, 2000.
14. W. Gropp and E. Lusk, "Fault tolerance in message passing interface programs," *Int. J. High Perform. Comput. Appl.*, vol. 18, pp. 363–372, August 2004.
15. J. Dongarra, L. Blackford, J. Choi *et al.*, "ScaLAPACK user's guide," *Society for Industrial and Applied Mathematics, Philadelphia, PA*, 1997.
16. T. Davies, C. Karlsson, H. Liu, C. Ding, , and Z. Chen, "High Performance Linpack Benchmark: A Fault Tolerant Implementation without Checkpointing," in *Proceedings of the 25th ACM International Conference on Supercomputing (ICS 2011)*. ACM.
17. Z. Chen, G. E. Fagg, E. Gabriel, J. Langou, T. Angskun, G. Bosilca, and J. Dongarra, "Fault tolerant high performance computing by a coding approach," in *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPoPP '05. New York, NY, USA: ACM, 2005, pp. 213–223. [Online]. Available: http://doi.acm.org/10.1145/1065944.1065973