Enabling On-Demand Checkpointing and Algorithmic Failure Recovery Without Changing the MPI Standard

Wesley Bland*, Peng Du*, Aurelien Bouteiller*, Thomas Herault*, George Bosilca*, Jack J. Dongarra*†
* *Innovative Computing Laboratory, University of Tennessee*
1122 Volunteer Blvd., Knoxville, TN 37996-3450, USA
Email: {bland, du, bouteill, herault, bosilca, dongarra}@eecs.utk.edu

† *Oak Ridge National Laboratory*
Oak Ridge, TN 37996-3450, USA

***Abstract*—The International Exascale Software Project roadmaps predicts, as soon as 2014, billion way parallel machines 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 of paramount importance to maintain future scientific productivity. Currently, 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 and its implementations) offers extremely limited support of software-level fault tolerance approaches. In this paper, we present an approach that relies exclusively on the current MPI standard definition of a high quality implementation and enables algorithmic based recovery to complete the computation despite failures without incurring the overhead of customary periodic checkpointing.**

***Keywords*-Fault Tolerance, Message Passing Interface, ABFT, On-Demand Checkpointing**

I. 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 how massive parallelism has been embraced in the recent years, leading to ever growing systems featuring an impressive number of computing units; 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 Exaflop class machines, 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 soon as 2014, billion way parallel machines, encompassing not only millions of cores, but also tens of thousands of nodes, 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 of workload, composed of a large amount of medium to small scale jobs, which often represent the bulk of the activity on many HPC systems, has traditionally been 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 in high software development expenditures. Two long lasting issues have hindered ubiquitous adoption of streamlined fault tolerance techniques: first, the traditional checkpoint based approaches incur a steep overhead on failure-free operations; second, the dominant approach for programming parallel distributed memory systems, the MPI standard [2] and its implementations, offer extremely limited support for software fault tolerance approaches, which effectively limit the spectrum of options available to applications to periodic checkpointing and rollback recovery.

Several propositions have emerged from the ongoing MPI-3 forum[2], toward improving the expressivity of MPI with regard of fault tolerant techniques. However, it is yet unclear as to wether these propositions will prove successful enough to be blessed by the forum as they still incur synchronization overhead on failure-free scenarios. The current MPI-2 standard leaves open an optional behavior to qualify as a "high quality implementation", regarding failures: according to this specification in the case of an MPI_ERRORS_RETURN

[1] www.top500.org

[2] http://meetings.mpi-forum.org/mpi3.0_ft.php

error handler, when the MPI library detects a failure it should return control to the caller. This is at the opposite of the default mass-suicide action that all notable MPI implementations undergo in case of a failure. In this paper, we investigate the modifications that are required, inside the MPI implementation, to enable this behavior strictly within the scope of the current standard. We then describe how algorithm-based recovery techniques, illustrated by the most useful QR factorization, are able to leverage this behavior, to perform an inexpensive recovery that completely avoids costly periodic checkpointing, and rollback recovery. Although the proposed technique uses checkpointing to save the state of the application, it does so only after the system was affected by a failure, and the recovery operation could be qualified as forward recovery, since it does not make the application rollback.

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, we present in Section III the On-Demand Checkpointing approach, and the minimal support required from the MPI implementation. Section IV presents the use case: a fault tolerant version of the QR algorithm, and how it has been modified to fit the proposed paradigm. Section V presents a performance model to assess the efficiency of both periodic checkpointing with rollback recovery and On-Demand Checkpointing, and Section VI presents an experimental evaluation of the implementation.

II. Background & Related Work

Background

Message passing is the dominant form of communication used in parallel applications, and MPI is the most popular library used to implement it. However, as fault tolerance becomes a growing concern for application developers, users have encountered some challenges with the current MPI Standard that limit their options of fault tolerance methods. The primary form of fault tolerance today is to periodically write a checkpoint to disk. While this method is effective in allowing applications to recover from failures by restarting the work from a previously saved point, it causes serious concerns on the scalability [3]. Moreover, such proactive approach to fault tolerance requires a good idea of how many faults might hurt the system, with which frequency and on what nodes. Many works have discussed the optimal checkpointing period in the hope that as few as possible of these preventive actions are taken by the application [4], [5], [6], [7], [8]. Unlike these works, the work presented here focuses on *forward recovery*: checkpoint actions are taken only *a*fter a failure is detected, make it unnecessary to hypothesize on an optimal checkpoint interval. The checkpoint interval is optimal, by definition, as there will be one checkpoint interval by effective fault.

An alternative approach to rollback recovery is to take advantage of the properties of the application to design it as naturally fault-tolerant. This technique is traditionally called Algorithmic Based Fault Tolerance [9]. The algorithm itself includes modifications, or additional steps, to cope with the loss of some of its data. It includes a modification of the algorithm, usually to maintain redundant information in the data during the life of the application, and a recovery procedure that works only with the data remaining after the failure is detected, and reconstructs the missing data using additional computation and communication. To support such an algorithm, the underlying programming environment must however provide a way to communicate after the failure occurs on one of the processes.

The current MPI Standard (MPI-2.2, [2]) does not provide significant help to deal with that 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 as 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 to go 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. The runtime system then returns with an error code, signaling the failure of the run, leaving no other choice to the user but to run a new parallel execution.

The MPI forum is currently examining options for the future direction of MPI for MPI-3. One of the workgroups is dedicated to propose a standard form of MPI-supported fault tolerance. The proposal outlines a method of run-through stabilization which allows the application to acknowledge and repair communications, both collectively and between specific ranks in a point-to-point way [10]. The emphasis of the proposal is a set of "validation" functions which the application is required to call to repair and re-enable communication within an MPI communicator containing a failed process. To repair point to point wildcard receives, the application needs to collectively call the function MPI_COMM_REENABLE_ANY_SOURCE. To repair

collective communication within a communicator, the application needs to call the function MPI_COMM_VALIDATE. These functions give the MPI implementation an opportunity to acknowledge failures and discover or ensure that other MPI processes also acknowledge the same failures. It also gives the MPI library a chance to repair communication channels between remaining processes, optimizing communication topologies if possible and necessary.

While this method of fault tolerance is sufficient for Algorithmic Based Fault Tolerance, it is not without its drawbacks. The calls necessary to recover from collectives incur a non-trivial overhead even during the fault free case. MPI_COMM_VALIDATE requires a distributed consensus algorithm which is currently best implemented at log scale [11]. While this level of overhead is better than the current state of the art of periodic checkpointing, it still presents a significant cost that not all applications want or need to pay to check the validity of the communicators. Most importantly, this proposal does not yet include process recovery, which is left to a future proposal to the MPI forum.

Related Work

FT-MPI [12] is an MPI-1 implementation which added extensions to the MPI standard to give users options for their Algorithmic Based Fault Tolerance. FT-MPI proposed to change the MPI semantics of some of the calls, to enable continuing the execution of the parallel application after a failure hits the system, and to rebuild the communicators, thus re-enabling communications. This approach has been proven successful, and some applications have been implemented relying on the features of FT-MPI. However, these modifications of the standard were not imported in the official MPI standard, and no other MPI implementation took the same approach. The lack of large distribution of the FT-MPI implementation prevented a large base of users from implementing their solution based on this proposition.

Besides the works that have been cited previously to present the problem statement, the different approaches that have been proposed, and how this approach is original, the article by W. Gropp and E. Lusk in 2004 [13] is the work closest to the On-Demand Checkpointing, from the MPI requirement perspective. In this article, the authors explain how the standard can be interpreted, or slightly modified, to allow for a form of fault tolerance. They consider different approaches: periodical checkpointing; using inter-communicators and separate MPI applications to contain an error in an MPI application; modifying the MPI semantics; or propose new extensions. However, the last three propositions demand more from the MPI implementation than we require in this work: for example, the MPI library is supposed to continue its normal execution, if the error was located in another MPI application, connected with the one subject to the error through an inter-communicator. In our work we do not even require such a step: the only demand on the MPI implementation is that it does not forcibly kill the living processes without letting them take a checkpoint, but returns an error. Once this is ensured, no requirement from any MPI call is needed.

Moreover, we illustrate the well soundness of our approach using a non-trivial algorithm: a QR factorization, that is made fault tolerant using the modified Open MPI, and the On-Demand Checkpointing technique. We demonstrate that this approach is functional, and evaluate its performance at large scale.

III. Enabling Software Fault Tolerance in MPI

A. Errors in MPI and On-Demand Checkpointing

Contrary to all the proposed additions to the MPI standard introduced in Section II, we advocate in this paper that an extremely efficient form of fault tolerance can be implemented, based on the MPI standard, for those applications which can take advantage of it. According to the paragraphs of the standard cited in the previous section, when implementing a high-quality MPI library, the application should regain control following a process failure. 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 itself. In most current MPI implementations, MPI_ERRORS_ABORT is the default (and often, only functional) MPI_Errhandler. However, the MPI standard also defines another handler called MPI_ERRORS_RETURN. This handler should perform any necessary cleanup at the library level and then return to the application, giving it the opportunity to continue if possible. The MPI standard does not require that any MPI function calls continue to operate after an error is returned (thus after a failure), but even without MPI calls, we will show that it is possible for the application to complete its computation, or save enough information after the failure is detected to allow for continuing the computation in a new application.

Algorithmic Based Fault Tolerance algorithms are capable of restoring missing data from redundant information located in the other processes. This requires, of course, to communicate between processes, and we acknowledge that requiring from the MPI implementation to maintain functioning MPI calls after a failure hit one of the nodes is too demanding in front of the current standard. However, expecting that the living processes continue to work after a failure, and to be able to access the other functionalities of the system seems significantly less perturbing for the MPI implementors. We will present in the second part of this section how this was done in the Open MPI implementation.

Under this assumption, consider the figure 1. Horizontal lines represent the execution of different processes in two successive MPI applications. Living processes are allowed to call any other type of operations that does not depend on MPI after a fatal error is returned. In the proposed general approach, processes that detect a failure will stop

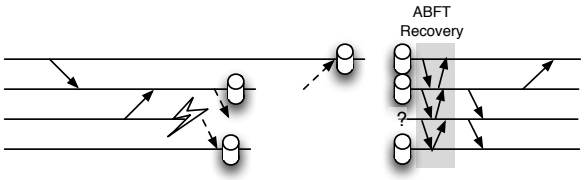


Figure 1. Execution diagram of an On-Demand Checkpointing approach with one failure

immediately to use MPI, save, on a file (local or remote, depending on the machine capability), all necessary information to proceed with the Algorithmic Based Fault Tolerance algorithm and exit without calling MPI_Finalize. This operation does not require communication if a filesystem, or any other way to do Input/Output operations remains available. Because process exiting will be considered as failed from the MPI perspective, all processes of the MPI application will eventually discover the initial failure, and the application that was hurt by a failure will then terminate. The user (or a controlling script), that launched the MPI application in the first place, can detect the failure by checking the return code of the MPI runtime system.

Usually, the execution is terminated at this point. However, the user or the script can launch a new MPI application on the same resources. This new application will load the local states (second series of lines in the figure), when available. If the file is not available, it means that it was not saved for the requesting ranks, and thus that these ranks were the ones hit by a failure in the previous run. In the new MPI application, the MPI system is functional. So, communications are enabled, and the recovery procedure of the Algorithmic Based Fault Tolerance approach can be called to restore the data of the process(es) that did not find it locally. From this point on, the application can continue its normal execution, as the global state has been restored by the Algorithmic Based Fault Tolerance recovery procedure. If another failure hits the system again during the recovery, the local states are not updated, and the relaunch starts from the beginning. If another failure hits the system after the Algorithmic Based Fault Tolerance recovery, the same procedure is followed to handle it.

B. On-Demand Checkpointing and Rollback Recovery

An On-Demand Checkpointing application behaves similarly to the familiar checkpoint/restart approach. However, by not periodically saving a checkpoint to stable storage, the application gains an enormous overhead reduction. Rather than needing to tune an optimal checkpoint interval and periodically save that checkpoint to stable storage, the application only needs to save the "checkpoint" once after the failure has already occurred. This has the added benefit of incurring zero checkpoint-related overhead in the fault-free case, a consideration that is very important on reliable systems with a low likelihood of failure. This means that the same code can be run on both reliable and non-reliable hardware without modification. This method of on-demand checkpointing gives the user a safe place to write its checkpoint before being aborted by the MPI library.

In traditional checkpointing, a complete set of checkpoints is saved periodically. With on-demand checkpointing, the failed processes will not write their checkpoints as they have already failed and will most likely be unable to do so (although link failures could be taken into consideration: in this case, all processes are able to save the local state, and the recovery procedure will simply consist in determining the global state at the moment of failure). Because of this, the algorithm will need to be able to recover the failed process and generate all necessary information to bring it back up to the same point as the other non-failed processes.

There is another fundamental difference between On-Demand Checkpointing and Coordinated Checkpointing, or most of the rollback recovery approaches. In traditional rollback recovery, the checkpoint images must be saved to a reliable media. It is, in particular, critical to recover the checkpoint image of the failed process to ensure the recovery. In [14] it is proposed to keep a local copy of the checkpoint image for the living processes, if local storage is available, in order to reduce the I/O stress on the remote storage system, but the failed process must have their images stored remotely. Since the system does not know what processes are going to fail at the time of checkpoint, all images must be also stored remotely. Thus, the (distributed) checkpoint can be considered complete only when all processes have stored their image on a resource that has a low probability to fail simultaneously with them.

In the case of On-Demand Checkpointing, the situation is completely different: only the checkpoint image of the living processes at the moment of failure is required. Thus, local storage can be considered if the same resources can be reused for the restarted MPI application. Similarly, traditional rollback recovery cannot rely on local caching of the shared filesystem, while On-Demand Checkpointing can use this operating system feature to accelerate significantly the time it takes to save the checkpoint image, and reload it, if memory is available.

The cost of the On-Demand Checkpointing approach is of course similar to the cost of any Algorithmic Based Fault Tolerance approach: the application might need to do extra computation during the whole execution, to maintain internal redundancy that will enable it to restore the missing data when a failure hits any of the nodes. However, Algorithmic Based Fault Tolerance techniques often have an excellent scalability. For example, the Algorithmic Based Fault Tolerance QR operation that we use to illustrate On-Demand Checkpointing in this work has an overhead on the fault-free execution inverse proportional to the number of participating processes, and therefore nodes.

C. 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 implementation (OMPI), with an additional level to provide support to the other two (OPAL). 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 implemented for this work was to make the runtime system resilient, and leave the policy decision in case of failure to the MPI library, and ultimately to the user application. To do so, we included an incarnation number in the process names, as they are known by the runtime system. Incarnation numbers are used to count how many times a process has failed and are necessary to track the status of all of the processes in order to know which processes are alive, which have failed, and reconcile contradictory views. Once the incarnation numbers are included in the runtime layer, the runtime error manager needs to change its default behavior during a failure. Rather than notifying the head node process (HNP) and starting an abort, we introduced a notification phase, where all runtime processes are notified of the failure so they can propagate this information to the MPI layer which appropriately handles the failure according to the application's instructions.

The runtime layer provides an out-of-band communication mechanism (OOB) that relays messages through a routing policy implemented in the routed component. This OOB layer is used to detect failures at the runtime level, and propagate the failures notifications. We have changed the routing algorithms to allow for a more dynamic network, were processes of the OOB can leave the network due to a failure. The underlying routing algorithm also has a significant influence on the time it takes to detect a failure and notify all MPI processes, as the experiments will show.

Like the runtime layer, the OMPI layer has a default behavior during a process failure of calling MPI_Abort. The alternative error handler MPI_ERRORS_RETURN is made functional by receiving the failure notification from the runtime system. When the case happens, all outstanding communication requests involving the communicator with the failed process are cancelled. This prevents any outstanding communications with the failed process from causing a dependency issue with future communications and ensures that the application will be notified of the process failure when the MPI function that it is calling returns a non-MPI_SUCCESS error code.

In addition to allowing the application to select the MPI_Errhandler MPI_ERRORS_RETURN, the application can choose to implement custom MPI_Errhandlers, another feature of the current MPI standard. In these custom MPI_Errhandlers, the application can immediately perform any post-failure operations, such as saving the local state. These custom MPI_Errhandlers provide enormous flexibility to the user.

IV. Fault Tolerant Algorithm for the QR factorization

To demonstrate the effectiveness of the On-Demand Checkpointing fault-tolerance mechanism, the QR factorization implementation from ScaLAPACK [15] is chosen as the testbed. QR factorization is a popular method for solving the linear least squares problem and QR algorithm which is at the center of a special version of eigenvalue algorithm. The fault tolerance utilities proposed in this work can be extended straightforwardly to other linear algebra operations like LU and Cholesky.

In this section, we first review the fault tolerance QR factorization in [16]. Then the situation where failure occurs during lower level routines such as PDLARFB is addressed. Such situation has been avoided by most of the related work in the area for the compleixty it introduces. In this work we proposed a novel technique with which failure during these lower level routines can be also recovered.

A. QR factorization on Distributed Memory System

For an $M \times N$ matrix A, 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. For simplicity of expression, we use a square matrix $M \times M$ in this work, but the result applies also to rectangular matrices. There are several methods for computing the QR factorization, such as the Gram-Schmidt process, the Householder transformations, and the Givens rotations. ScaLAPACK uses a block version of the QR factorization by accumulating a few steps of the Householder matrix. This method is rich in level 3 BLAS operations and therefore can achieve high performance. Q is stored under the lower diagonal of the input matrix in the form of a WY representation of the Householder transformation products[17], [18].

ScaLAPACK implements the block QR factorization as follow. At step i , an $m \times m$ submatrix A_i is partitioned and factorized as

$$A_i = \begin{bmatrix} A_1 & A_2 \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = Q \times \begin{bmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{bmatrix}$$

Here A_{11} is of size $nb \times nb$, where nb is called the block size. A_{21} is of size $(m - nb) \times nb$. $A_1 = [A_{11}, A_{12}]^T$ constitutes the area for the panel QR factorization. Since ScaLAPACK uses the Householder method, Q is expressed as a series of Householder transformations in the form $H_i = I - \tau_i v_i v_i^T, i = 1 \cdots nb$. v_i has 0 for the first $i - 1$ entries, 1 on the $i - th$ entry and $\tau_i = 2/v_i^T v_i$. In ScaLAPACK, v_i is stored below the diagonal of A and when Q is applied to the trailing matrix $A_2 = [A_{21}, A_{22}]^T$, Q is computed by $Q = H_1 \cdots H_{nb} = I - VTV^T$, where T is

an upper triangular matrix of size $nb \times nb$ and V has v_i as its $i-th$ column. With this expression, the trailing matrix update becomes

$$\tilde{A}_2 = \begin{bmatrix} \tilde{A}_{12} \\ \tilde{A}_{22} \end{bmatrix} = Q^T A_2 = (I - VT^T V^T) A_2 \quad (1)$$

This finishes one iteration of the block QR factorization. This process is repeated from $\tilde{A}_{22}$ until the whole matrix is factorized.

B. Checksum Generation

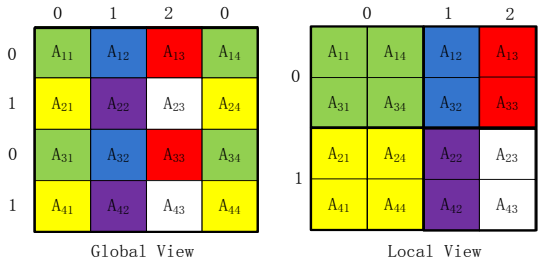


Figure 2. Example of a 2D block-cyclic data distribution

For high performance dense matrix factorization, data layout plays a critical role in the scalability and performance on distributed memory systems [19], [20]. In 2D block-cyclic distributions, data is split into equally sized blocks, and all computing units are organized into a virtual two-dimension grid. Each data block is distributed to computing units in a round robin fashion, following the two dimensions of the virtual grid. figure 2 is an example of a 2×3 grid applied to a global matrix of 4×4 blocks. The same color represents the same process while numbering in A_{ij} indicates the location in the global matrix. This layout helps with load balancing and reduces data communication frequency, since in each step of the algorithm, many computing units can be engaged in computations concurrently, and communications pertaining to blocks positioned on the same unit can be grouped. Because of these advantages, many prominent software libraries (like ScaLAPACK [15]) use a 2D block-cyclic distribution.

The first step of our algorithm is generating the checksum for the whole matrix A by $A_c = A \times G$, where G is a generator matrix. This checksum establishes a row-wise summation relationship between matrix entries and checksum. However, with a 2D block-cyclic data distribution, the loss of a single process, usually a computing node which keeps several non-contiguous blocks of the matrix, results in holes scattered across the whole matrix. Therefore checksum is computed for each $M \times (Q \times nb)$ blocks separately.

Let C_i be the i^{th} checksum item, and A_i^j, be the i^{th} data item on process j, $1 \le i \le \lceil \frac{N}{Q} \rceil$, $1 \le j \le Q$:

$$C_k = \sum_{k=1}^{Q} A_k^k \quad (2)$$

Another issue caused by the 2D block cyclic distribution is that checksum itself can be lost simultaneously during a failure, if it is held by the same processor, which renders the Algorithmic Based Fault Tolerance recovery impossible. To prevent this, a copy of each checksum block column is made and put on the side of its original column, resulting in $M \times \left\lceil \frac{2N}{Q} \right\rceil$ storage requirement for checksum. Compared with the $M \times N$ storage for data, the ratio is $\frac{2}{Q}$ which becomes negligible when Q is large.

C. Q-parallel Checksum to Protect the Left Factor

The left factor Q is stored under the lower triangular in the form of v. Since one panel of v of width nb is produced in each iteration, to protect this portion of data from failure, the checksum should be generated during the factorization at a regular interval. Simple vertical checksum using (2) that applies vertically on a column suffers from scalability issue since only up to P processes are involved, and the checksum generation is on the critical path of LU factorization, depriving the rest $P \times (Q-1)$ processes of performing trailing update that takes up most of the computation FLOPS in the factorization. Thus, in this work, we adopt the scalable Q-parallel checksum generation technique presented in [16] to protect the left factor Q.

For the Q-parallel checksum, $P \times Q$ processes are grouped by sections of width Q, called a *panel scope*. When the panel operation starts applying to a new section, the processes of this panel scope make a local copy of the impending column and the associated checksum, called a *snapshot*. This operation features the maximum $P \times Q$ parallelism due to the fact that all local copies are performed on each process locally without communication. The memory overhead involves only the space to keep at most two extra columns the whole time, one for saving the state before the application of the panel to the target column, and one for the checksum column associated to these Q columns. The algorithm then proceeds as usual, without any further update of the redundant memory until entering the next Q trailing updates. Because of the availability of this extra protection column, the original checksum can be modified to protect the trailing matrix without threatening the recovery of the panel scope, which can rollback to that previous dataset should a failure occur.

At the completion of a panel scope, the $P \times Q$ processes perform checkpointing simultaneously. Effectively, P simultaneous SUM reduction operations are taking place along the block rows, involving the Q processes of that row to generate a new protection block. This scheme enables the maximum parallelism for the checksum generation, hence decreasing its global impact on the failure free overhead.

D. Algorithmic Based Fault Tolerance for the Right factor

The protection of the right factor differs from that of the left factor because mathematical properties of the algorithm

can be utilized such that no additional step is needed to keep the checksums up to date, dropping the necessity of frequently generating checksum to keep up with the state of the mtraix.

The use of this technique is illustrated with a 2×2 block matrix. For an input matrix

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

Checksum is generated using the generator matrix G as

$$A_c = A \times G = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \end{bmatrix}$$

A_{13} and A_{23} are checksum blocks, and

$$\begin{cases} A_{13} = A_{11} + A_{12} \\ A_{23} = A_{21} + A_{22} \end{cases} \tag{3}$$

After performing a QR factorization on A_C, we have

$$A_c = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \end{bmatrix} = \begin{bmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \end{bmatrix} \times \begin{bmatrix} R_{11} & R_{12} & R_{13} \\ 0 & R_{22} & R_{23} \end{bmatrix} \tag{4}$$

Using Equations (3) and (4), the following equation can be obtained:

$$\begin{bmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \end{bmatrix} \times \begin{bmatrix} R_{11} + R_{12} - R_{13} \\ -(R_{23} - R_{22}) \end{bmatrix} = Q \times \begin{bmatrix} R_{11} + R_{12} - R_{13} \\ -(R_{23} - R_{22}) \end{bmatrix} = 0 \tag{5}$$

Because Q is non-singular, Equation (5) can be solved as:

$$\begin{cases} R_{11} + R_{12} &= R_{13} \\ R_{22} &= R_{23} \end{cases} \tag{6}$$

Equation (6) shows that at the end of the QR factorization, the checksum blocks on the right of the matrix still hold the same relationship with the data blocks as before the factorization except only the upper triangular matrix is involved. It can be further shown that the checksum relationship is also valid during factorization at the end of each iteration. The proof is beyond the scope of this paper, but this feature is used to recover lost data in the right factor at time of failure recovery.

E. Failure in PBLAS routines

An important condition for the effectiveness of Algorithmic Based Fault Tolerance is the completion of the current iteration. When a failure interrupts the program execution during an iteration, the checksum ends up in intermediate form and as a result cannot be used for recovery. This problem worsens when a failure occurs during a lower level routine, like a PBLAS, causing a partial trailing matrix update. In this case, updates have been applied to parts of the dataset, possibly without having updated accordingly the corresponding checksums. In the case of the QR algorithm, this problem is solved by saving the local state when a failure is detected in PDLARFB rather than in PDGEQRF. The recovery process in this case is described as follow.

As shown in Equation (1), the trailing update of QR carries out operation $Q^T A_2 = (I - VT^T V^T)A_2 \rightarrow \tilde{A}_2$. The right arrow means the updated trailing matrix is written in-place to A_2. The trailing matrix update of QR is similar to PDGEMM for LU which has been shown to hold the checksum relationship only at the end [16]. Therefore the procedure to recover from a failure in PDLARFB is:

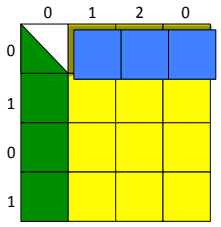
1) Survived processes mark the progress and dump critical data to disk
2) After re-spawning, all processes dry-run to the failure point
3) All except the replacement process load checkpoint from disk
4) All processes resume computing from the failure point to the end of PDLARFB
5) At the exit of PDLARFB, recover all lost data in checksum and the whole matrix
6) Execution of PDGEQRF returns to normal

The 'dry-run' step is to re-establish the calling stack of all processes to the failing point. Therefore PBLAS and ScaLAPACK routines for computing, for example, PDGEQR2, PDLARFT, etc. are skipped over during the dry run.

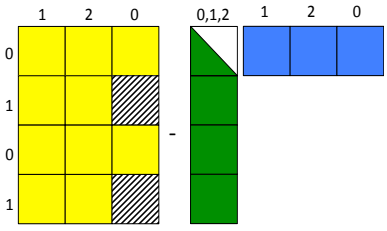
The recovery is demonstrated with an example of a 4×4 blocks matrix on a 2×3 grid where failure occurs during PDLARFB.

PDLARFB implements $Q^T A_2 = (I - VT^T V^T)A_2$ in three steps:

1) $W \leftarrow V^T A_2$



(a) After the first step: $A_2^T V$



(b) $A_2 - V\tilde{W}^T$

Figure 3. PDLARFB

2) $\tilde{W} \leftarrow W \times T$
3) $\tilde{A}_2 \leftarrow A_2 - V\tilde{W}^T$

Suppose the failure occurs right after step 1 on process (1,0). In step 1, as shown in figure 3(a), V is stored in the green trapezoid and A_2 is in the yellow blocks. V is first broadcast row-wise to all columns, then GEMM is called on each process that owns A_2 with the local V and A_2. Finally, the result is produced with column-wise block summation and the result is stored on the first row of processes that process the first row of A_2 (blue blocks). From the MPI facilities presented in Section III-C, the failure location is broadcasted to all surviving processes and matrix data are dumped to the disk, including peripheral data like the TAU array and workspace. Surviving processes also keep a record on whether they have finished the DGEMM in step 1.

After critical data is saved to disk, the program exits and is re-spawn with a replacement process in the failed process's location. The re-launched program dry-runs to the failure point in step 1 of PDLARFB. All previously surviving processes load their checkpoint from disk while the replacement process stays with its blank data. Then the program resumes execution of PDLARFB. Since failure is on process (1,0), W survives the data loss.

Step 2 of PDLARFB is $W \times T$ where W is the blue blocks in figure 3(a) and T is a $nb \times nb$ upper triangular matrix. Since T resides on each process in the row that owns W, the correctness of T can be always guaranteed, and therefore $\tilde{W}$ has no lost block in it after calling DTRMM. $\tilde{W}$ is broadcasted column-wise for step 3.

Step 3 of PDLARFB is shown in figure 3(b). In $\tilde{A}_2 \leftarrow A_2 - V\tilde{W}^T$, besides $\tilde{W}^T$, V is also correct since V has been broadcast row-wise to all process in step 1, therefore even if blocks of V are destroyed by the failure, the result on the replacement process can be recovered from its neighbour processes in the same row. The result of step 3, also that of PDLARFB, is affected by the incorrect result in A_2, expressed in shadowed blocks in figure 3(b). These incorrect blocks remain in the result of DGEMM in this step. They are fixed later in the recovery process in PDGEQRF using both the ABFT and Q-parallel checksum.

For PDLARFB, both V and T can be guaranteed correct no matter when and where failure occurs, the only variable factor is W. However if the failure does punch holes in W, more shadow blocks appear in the result of PDLARFB, and they can still be fixed by the recovery in PDGEQRF.

V. Models of Fault Tolerance Approaches

In this section, we present a simple model that we use to evaluate the efficiency of our approach, when compared to the classical approach of a coordinated, blocking checkpointing algorithm. For the following, we will consider an application with a fixed number of processes, and a fixed amount of work to execute. We consider that without any fault-tolerance approach enabled, the execution time of this work with this number of processes is T.

The ABFT technique, as proposed in this article, can be characterized with a few parameters:

- C, the time it takes the living processes, to save their state after the failure occurs, exit, and for the runtime system to discover that all processes have exited with an error code
- α, a multiplicative factor, that encompasses all costs related to making the operation ABFT-capable. Such modification of the operation introduces additional operations and communications, distributed among the processes. Based on our experience with ABFT, we consider, in this model, that these operations introduce a slowdown of the execution time that can be captured with a single multiplicative factor, $\alpha > 1$.
- R, the time it takes for processes to restart from scratch, load the checkpoints of the living processes, exchange the missing information, and recover the missing data.

Since the number of processors and the problem size are fixed, C, R, and α, can all be considered as constants.

Our approach is deterministic, given these assumptions: the time it takes to complete the execution of the work with a number n of failures is

$$T^{ABFT}(n) \quad = \quad \alpha T + n(C + R) \tag{7}$$

Indeed, the execution is slowed down by the α parameter to maintain the different checksums incurred by the ABFT approach, and each failure triggers first a checkpoint phase, that is immediately followed by a recovery phase. At the end of the recovery phase, the algorithm is ready to proceed at the step that follows the failure, so no additional cost is incurred.

The periodical coordinated blocking checkpointing technique can be characterized with the following parameters:

- c, the time it takes for all processes to save their state when the timer expires
- r, the time it takes all processes to restart from this checkpoint (load the checkpoint from file)
- t is the time interval between two checkpoints. This value is usually set to something as close as possible to the expected MTBF, to avoid unnecessary overheads. It must be set low enough, though, to allow for progress.

In the coordinated checkpointing approach, the duration of the execution depends upon the moment of the failures: if a failure hits the system just after a checkpoint, no time is lost to recompute the state between the last checkpoint and the current step of the computation. If a failure hits the system just before a checkpoint, then the recovery restars from the last checkpoint and the last time interval must be entirely re-executed. To model this, we define two extreme cases: the best case, noted $T_B^{CC}(n)$ (for Time, Coordinated

Checkpointing, Best case), and $T_W^{CC}(n)$ (for Time, Coordinated Checkpointing, Worst case). These functions can be defined recursively as follows:

$$\begin{cases} T_B^{CC}(0) = T + cT/t \\ T_B^{CC}(n) = T_B^{CC}(n-1) + r + cr/t \end{cases}$$

$$\begin{cases} T_W^{CC}(0) = T + cT/t \\ T_W^{CC}(n) = T_W^{CC}(n-1) + r + t + c(r+t)/t \end{cases}$$

The cost on the execution without recovery is to stop the execution every t times units, and take a checkpoint. The recovery cost depends upon the case that is considered. In the best case, all failures will happen just after the last valid checkpoint. Hence the cost of recovery will be simply the time to load this checkpoint. In the worst case, all failures will happen just before the next checkpoint is completed, hence the cost of recovery will be the time to load the checkpoint, plus the time to redo the missing part of the execution, every time. Since recoveries increase the duration of the execution, they also increase the number of checkpoints that are taken.

We can solve these recursive definitions to obtain the following general forms:

$$T_B^{CC}(n) = (c+t)(nr+T)/t \quad (8)$$

$$T_W^{CC}(n) = (c+t)(n(r+t)+T)/t \quad (9)$$

To compare the two models, we can assume that if a local storage is present, $C \ll c$: in the case of the ABFT approach, if the same nodes remain accessible to the application to restart its runs, and a local storage is accessible, saving the checkpoint will be a local, embarrassingly parallel, operation. The Coordinated Checkpointing approach, however, cannot rely on local storage: a checkpoint image of all the nodes, including the failed nodes, will be necessary at recovery time. Thus, in the best case, a buddy algorithm must be used if local storage is to be used. Introducing additional communications, the checkpointing time is expected to be larger in that case.

On the other hand, if a local storage is not accessible, one can safely assume that $C \sim c$, since they save the same amount of information (we are considering a user-level checkpointing in both cases). This checkpointing time can be influenced, in the case of a shared filesystem, by the number of nodes that checkpoint image together, but since we assume one failure simultaneously, the fact that the ABFT algorithm saves one less image might be not significant at large scale.

Similarly, if no local storage is available, $R \gg r$: they both incur the communication cost linked to loading the checkpoint image, but then the ABFT technique needs to introduce additional communications to recover the state. If a local storage is available, this will strongly depend on the amount of data saved, and the amount of communication required by the ABFT algorithm to implement its recovery.

Both models present a linear progression with the number of faults. When $n = 0$, the On-Demand Checkpointing approach is slowdown by the factor α, while the periodical Checkpointing approach incurs a slowdown proportional to $(c+t)/t = 1 + c/t$. As was demonstrated in [16], $alpha$ decreases when the number of processes increase. In the case of the QR factorization, it increases with the square root of the problem size, to maintain the checksum operation. c/t is proportional to the duration of the checkpoint operation, hence to the problem size, and in non-scalable storage system with the number of processes. Moreover, it is highly dependent on the t parameter, that is often be set conservatively to guarantee progress in case of failures.

When the system is subject to failures, the cost of the recovery of the On-Demand Checkpointing technique is proportional to the number of failures, and to the duration of the recovery, R. As stated above, this overhead depends on the disk capabilities, but also on the efficiency of the recovery procedure given by the Algorithmic Based Fault Tolerance algorithm, and on the runtime system of the MPI to detect the incomplete termination, and launch a new application, efficiently. In the case of QR, this recovery increases with the size of the problem and the square root of the number of computing elements. The periodical checkpointing approach on the other hand incurs a recovery cost that is constant, but must redo part of the execution. A large t parameter in this case is damageable because it increase the probability that a long part of the computing time is lost. The On-Demand Checkpointing technique does not suffer from this parameter, and the cost of recovery is entirely defined by the size of the problem and the number of processors.

VI. Performance Discussion

In this section, we evaluate the performance of the On-Demand Checkpointing technique using an implementation of the proposed approach in the Open MPI framework. We first analyze the overheads and performance in the Open MPI implementation, then the performance of an Algorithmic Based Fault Tolerance QR operation implemented using On-Demand Checkpointing.

A. MPI Fault Detection

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 propagation introduces very little overhead to the MPI library. In the fault-free case, the failure detection algorithm introduces no overhead as it is not activated. It does introduce a small memory increase as the library also needs to track the incarnation number of each process, however this amount is relatively low as the incarnation number is only 8 bytes.

In the case where a fault occurs, the implementation is very efficient at discovering the fault and propagating

the information to the rest of the application quickly. We performed a micro-benchmark a small local cluster as a proof of concept. The cluster has 16 nodes with two 8-core Intel Xeon processors each. We used only one core on each node because the increased number of ranks on each core would not have an impact on the result of our measurements.

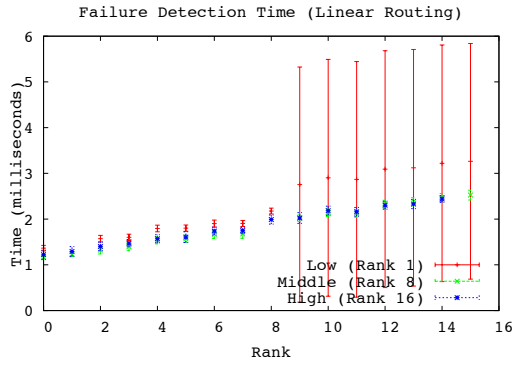


Figure 4. Failure Detection and Propagation with a Linear Topology

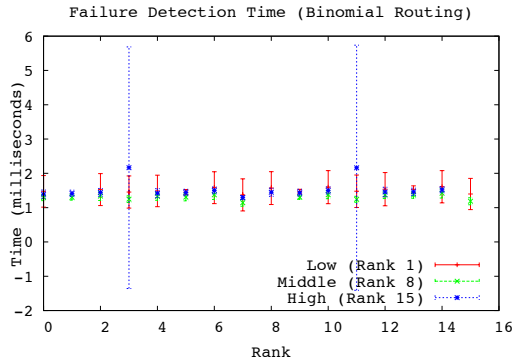


Figure 5. Failure Detection and Propagation with a Binomial Topology

Our evaluations used 16 nodes and ran 20 times for each test. The average value is presented here. In the micro-benchmark, the code performs an MPI_Barrier, injects a failure at a specific rank, and then waits for a custom MPI_Errhandler to be called, alerting the MPI application to the process failure. The time shown is the difference between the point directly before the failure is injected and the moment the MPI_Errhandler is called. Figures 4 and 5 show the time taken with two communication topologies. The linear topology in Figure 4 is a simple topology where every node is connected to the HNP. Figure 5 shows a binomial tree topology. Because of the small scale available to us, the failure propagation time is not effected by the communication topology. We performed other tests on a larger machine, but due to a machine specific issue, the results demonstrated too much noise to show useful data. However, it did demonstrate that a tree-based topology can significantly improve the communication time once the number of connections to the HNP would overwhelm the linear communication pattern. Regardless of the small scale, these results still acheive the goal of demonstrating the small amount of time necessary to detect an propagate failures through the runtime.

This failure notification happens at the runtime system level, asynchronously from any MPI call, ensures that with a high probability, MPI processes will be notified of the failure during one of the next MPI calls they enter. If the runtime system were not propagating the failure notification, failures could be discovered by MPI processes only when they try to communicate directly with a failed process. Since in the On-Demande Checkpointing scheme, after they are notified of a failure any process saves its state and then exits (thus behaving as a failed process for the rest of the living processes), all communicating processes would eventually be notified of the initial failure. However, depending on the communication scheme of the application, this notification using the in-band channels closure could take significantly longer. Note that the Algorithmic Based Fault Tolerance algorithm would be able to recover from both scenarios. The out-of-band notification of the runtime system is thus there to improve the performance by reducing the latency needed to enter the recovery procedure.

B. *On-Demand Checkpointing for QR*

The On-Demand Checkpointing QR checkpoints data from memory to disk on the living processes at the time of failure. Therefore disk I/O access time is a critical component of the performance overhead.

To evalue the performance impact of disk access, the implenentation of the On-Demand Checkpointing algorithm based on the ScaLAPACK QR is tested on two cluster systems at different scale. The first machine, “Dancer”, is a 16-node cluster at the University of Tennessee, Knoxville. All nodes are equipped with two 2.27GHz quad-core Intel E5520 CPUs, connected by 20GB/s Infiniband. Solid State Drive disks are used as the checkpoint storage media. The second system is the Kraken supercomputer by Cray Inc. at the Oak Ridge National Lab. Kraken has 9,408 compute nodes. Each node has two Istanbul 2.6 GHz six-core AMD Opteron processors, 16 GB of memory, and a highly scalable cluster file system “Lustre”. All the nodes are Connected by the Cray SeaStar2+ interconnect. In all experiments, the block size is set to 100.

Figure 6 presents the performance of this QR implementation on the Dancer cluster with a 8×16 process grid. The FT-QR (no failure) presents the performance of the On-Demand Checkpointing implementation, in a fault-free execution, while the FT-QR (with failure) curves present the performance of the same implemenation, when the failure is injected after the first step of PDLARFB that performs

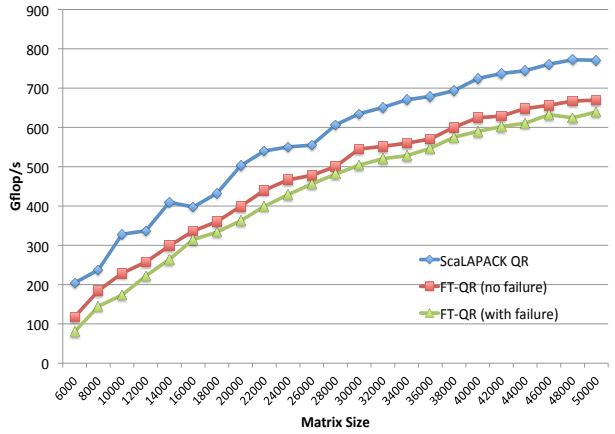


Figure 6. Performance on Dancer (16×8 grid)

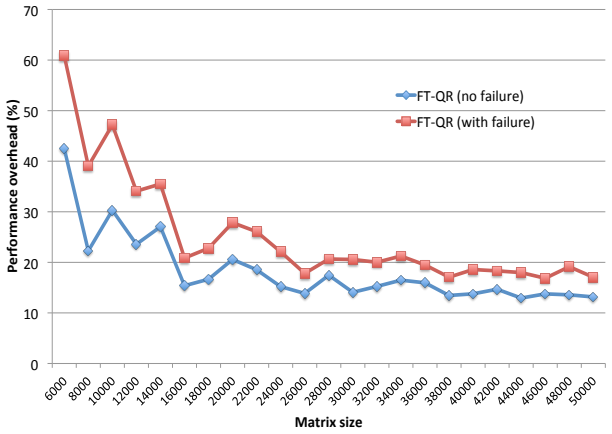


Figure 7. Overhead over ScaLAPACK QR on Dancer (16×8 grid)

$W \leftarrow V^T A_2$. The performance of the non-fault tolerant ScaLAPACK QR is also presented to serve as a reference.

The difference with the ScaLAPACK QR is caused by the parallel-Q checksum and the Algorithmic Based Fault Tolerance algorithm. This overhead has been shown in [16] to scale down with larger number of processes and matrices. In the case of a run with an error, the following overheads adds up: the times to store the checkpoint to disk, re-launch an application, re-establish the position of all processes by dry running in the application until the failing point, loading checkpoint from disk and perform the Algorithmic Based Fault Tolerance recovery using the checksum found in the checkpoint of the previously living processes.

On Dancer, the performance of QR with on-demand checkpointing and recovery follows closely with the "no failure" performance. Figure 7 shows that as the matrix size increases, the recovery overhead falls below 5% more than the "no failure" overhead. By breaking down the run-time of each recovery elements, Figure 8 shows that checkpoint saving and loading only take a small percentage of the total run-time. On a problem of this size, the additional overheads are dominated by the time it takes to terminate the failing MPI application and relaunch a new one. Other than the fast solid state drive disks, the fast checkpointing can also be attributed to the disk cache provided by the OS. Since loading is performed immediately after saving, high disk cache hits can largely speed up the process. After matrix size 44,000 the memory usage on each node came close to limit and since no swap space is available on the Dancer cluster, disk cache support started to decrease and cause slight increase in disk access time, which however does not affect the overhead percentage from performing recovery.

Figure 9 presents the performance on Kraken with a larger grid and a different filesystem to store the checkpoint images. A similar effect of a small checkpointing saving and loading time is observed. The performance of the "with failure" case shows the same trend of closely following the "no failure" case performance. At size matrix 100,000 for instance, FT-QR successfully recovered from the failure and achieved 2.86 Tflop/s, which is 90% of the performance of the ScaLAPACK QR. This verified that the On-Demand Checkpointing QR also performs well at larger scales.

VII. Concluding Remarks

In this paper, we presented an original scheme to deal with failures using the current MPI standard, and still avoiding periodical checkpointing. Periodical checkpointing is subject to a critical parameter that is particularly hard to assess: the ideal period of checkpoint. A period too short will loose time and resources, doing unnecessary Input/Output. A period too long will also loose time and resources, by increasing the amount of the execution that must be re-done, if a failure hits the system a long time after the last successful checkpoint.

On-Demand Checkpointing takes checkpoint images at optimal times by design: only after a failure has been detected. It relies on Algorithmic Based Fault Tolerance techniques to complement this checkpoint with redundant

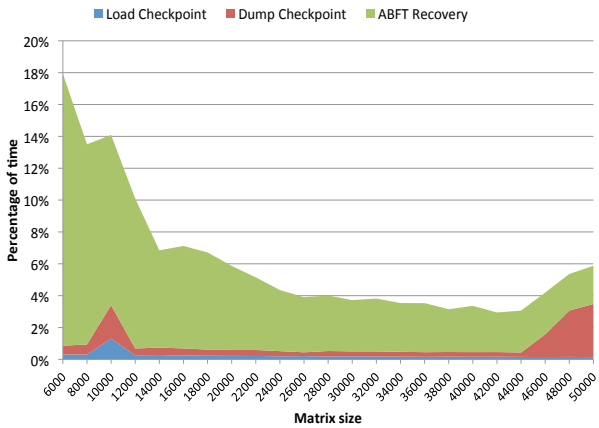


Figure 8. Time Breakdown of FT-QR on Dancer (16×8 grid)

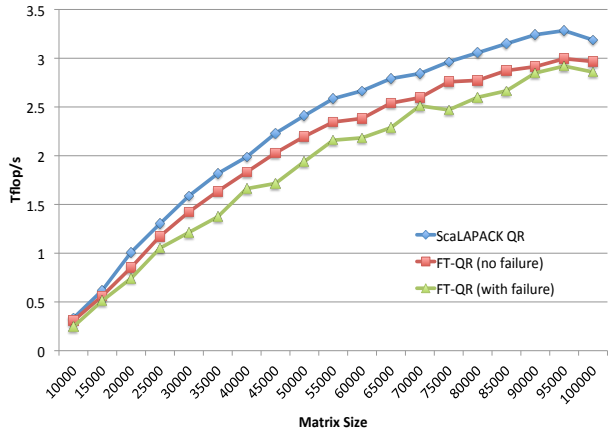


Figure 9. Performance on Kraken (24×24 grid)

information and enable recovering the whole application data at restart time. This simple scheme is easily integrable with high quality MPI implementations, as long as they let the application survive (even with MPI non-functionning) when a failure is detected and reported. We proposed performance models to evaluate and compare the benefits and limitations of On-Demand Checkpointing and periodical Checkpoint. These models highlight another feature of On-Demande Checkpointing: since the checkpoint images of the processes that were not subject to failure is needed to recover the state, local disks can be used, providing a much higher bandwidth than the remote storage needed by periodical Checkpointing. Last, but not least of the advantages of On-Demand Checkpointing, the restart does not trigger a rollback recovery, since the recovery procedure is able to reconstruct the data at the moment of the crash. The performance evaluation, based on the Algorithmic Based Fault Tolerance version of the QR factorization show that the disk Input/Output required by the method after a fault occured does not prevent to obtain excellent performance, and that a high quality MPI implementation can be obtained without decreasing its efficiency.

REFERENCES

[1] J. Dongarra, P. Beckman, and et al., "The international exascale software roadmap," *Intl. Journal of High Performance Computer Applications*, vol. 25, no. 11, p. to appear, 2011.

[2] The MPI Forum, "MPI: A Message-Passing Interface Standard, Version 2.2," Tech. Rep., 2099.

[3] 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.

[4] J. W. Young, "A first order approximation to the optimum checkpoint interval," *Commun. ACM*, vol. 17, pp. 530–531, September 1974.

[5] E. Gelenbe, "On the optimum checkpoint interval," *J. ACM*, vol. 26, pp. 259–270, April 1979.

[6] 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, p. 1590, 2001.

[7] 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.

[8] F. Cappello, H. Casanova, and Y. Robert, "Preventive migration vs. preventive checkpointing for extreme scale supercomputers," *Parallel Processing Letters*, vol. 21, no. 2, pp. 111–132, 2011.

[9] K. Huang and J. Abraham, "Algorithm-based fault tolerance for matrix operations," *Computers, IEEE Transactions on*, vol. 100, no. 6, pp. 518–528, 1984.

[10] 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 *EuroMPI 2011: Proceedings of the 18th EuroMPI Conference*, Santorini, Greece, September 2011.

[11] J. Hursey, T. Naughton, G. Vallee, and R. L. Graham, "A log-scaling fault tolerant agreement algorithm for a fault tolerant MPI," in *EuroMPI 2011: Proceedings of the 18th EuroMPI Conference*, Santorini, Greece, September 2011.

[12] G. Fagg and J. Dongarra, "FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world," *EuroPVM/MPI*, 2000.

[13] 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.

[14] F. Bouabache, T. Herault, G. Fedak, and F. Cappello, "Hierarchical replication techniques to ensure checkpoint storage reliability in grid environment," in *8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2008), 19-22 May 2008, Lyon, France*. IEEE Computer Society, 2008, pp. 475–483.

[15] J. Dongarra, L. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, S. Hammarling, G. Henry, A. Petitet *et al.*, "ScaLAPACK user's guide," *Society for Industrial and Applied Mathematics, Philadelphia, PA*, 1997.

[16] P. Du, A. Bouteiller, G. Bosilca, T. Herault, and J. Dongarra, "Algorithm-based fault tolerance for dense matrix factorizations." LAPACK Working Note, Tech. Rep. 253, Aug. 2011. [Online]. Available: http://www.netlib.org/lapack/lawnspdf/lawn253.pdf

[17] R. Schreiber and C. Van Loan, "A storage-efficient WY representation for products of householder transformations." *SIAM J. Sci. Stat. Comput.*, vol. 10, no. 1, pp. 53–57, 1989.

[18] C. Bischof and C. Van Loan, "The WY representation for products of householder matrices," in *Selected Papers from the Second Conference on Parallel Processing for Scientific Computing*. Society for Industrial and Applied Mathematics, 1985, pp. 2–13.

[19] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. Whaley, "ScaLAPACK: a portable linear algebra library for distributed memory computers–design issues and performance," *Computer Physics Comm.*, vol. 97, no. 1-2, pp. 1–15, 1996.

[20] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to parallel computing: design and analysis of algorithms*. Benjamin/Cummings, 1994, vol. 400.