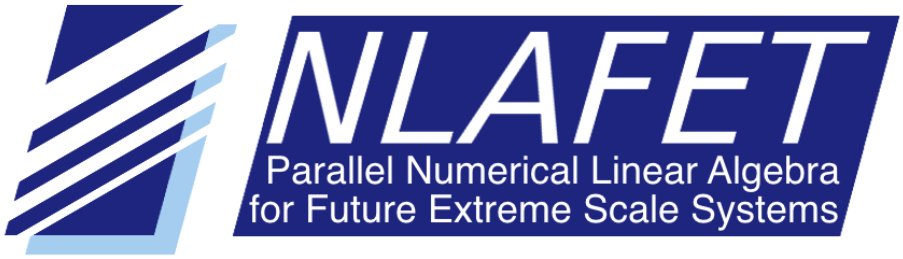



H2020–FETHPC–2014: GA 671633

# D6.6

# Algorithm-Based Fault Tolerance Techniques

October 2017

## Document information

| | |
|---|---|
| Scheduled delivery | 2017-10-31 |
| Actual delivery | 2017-10-31 |
| Version | 1.1 |
| Responsible partner | UNIMAN |

## Dissemination level

PU — Public

## Revision history

| Date | Editor | Status | Ver. | Changes |
|---|---|---|---|---|
| 2017-10-28 | Mawussi Zounon | Version | 1.1 | Feedback from internal reviewers added. |
| 2017-09-13 | Mawussi Zounon | Draft | 1.0 | Feedback from ICL reviewers added. |
| 2017-05-10 | Mawussi Zounon | Draft | 0.1 | Initial version of document produced. |

## Author(s)

Negin Bagherpour (UNIMAN)
Sven Hammarling (UNIMAN)
Nick Higham (UNIMAN)
Jack Dongarra (UNIMAN)
Mawussi Zounon (UNIMAN)

## Internal reviewers

Lars Karlsson (UMU)
Aurelien Bouteiller (University of Tennessee, Knoxville)
George Bosilca (University of Tennessee, Knoxville)
Thomas Herault (University of Tennessee, Knoxville)
Piotr Luszczek (University of Tennessee, Knoxville)
Yves Robert (ENS Lyon)

## Copyright



## Acknowledgements

This project has received funding from the *European Union's Horizon 2020 research and innovation programme* under the grant agreement number 671633.

# Table of Contents



# List of Figures

# 1 Introduction

The *Description of Action* document states for Deliverable D6.6:

> *D6.6: "Report on algorithm-based fault tolerance applied to the tiled Cholesky, LU, and/or QR factorizations."*

This deliverable is in the context of Task–6.3 (Algorithm-Based Fault Tolerance).

Over the last few decades, many large-scale science and engineering challenges have been successfully addressed owing to advances in high-performance computing (HPC). While the increased performance of HPC systems plays a tremendous role in numerical simulation breakthroughs, parallel numerical linear algebra solvers remain the primary tool for harnessing this improved computational power and empowering these scientific applications. In fact, numerical linear algebra libraries are the innermost numerical kernels in many scientific and engineering applications and are, consequently, one of the most time consuming parts to develop. Moreover, because the performance increase of HPC systems is commonly achieved using a huge number of complex and heterogeneous computing units, these large-scale systems suffer from a very high fault rate, where the mean time between two consecutive faults (MTBF) is getting shorter. Otherwise stated, numerical linear algebra solvers are more likely to experience faults on modern and future HPC systems, and it is absolutely critical that we develop parallel linear algebra solvers that can survive these faults.

According to the studies by Bianca et al. [68], faults may emanate from many sources, including hardware, software, network, human error, and the environment (Figure 1). However, among those possible sources, hardware faults are the predominant factor, as reported by Schroeder et al. in [69] and Gainaru et al. in [41].

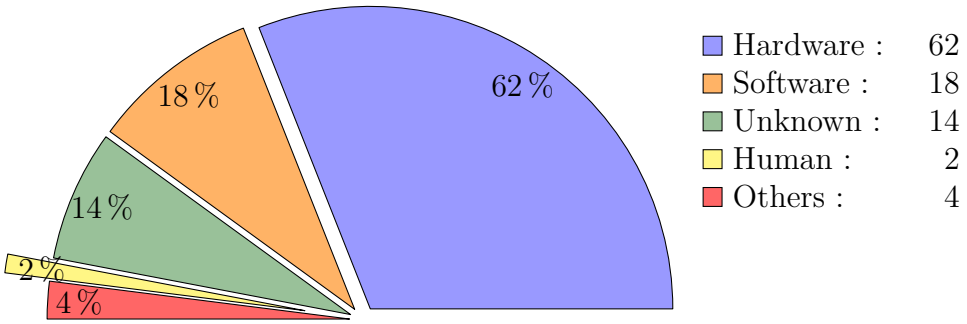



Figure 1: Root causes of faults according to Bianca et al. [68].

Depending on the systemic impact of a given fault, it may be labeled as a *hard* fault or a *soft* fault. A hard fault, also called a "fail-stop," is a fault that causes an immediate routine interruption. On the other hand, a soft fault is an inconsistency that does not lead directly to application interruption but can have other serious consequences. Typical soft faults include bit flips and data corruption [22]. There are a set of well-studied, system-based approaches for detecting hard faults. However, system-based approaches for detecting soft faults can be cost prohibitive; consequently, soft fault detection mechanisms often exploit sophisticated, application-specific techniques instead.

The main target of this work is the design of efficient fault tolerant solvers. There are two deliverables: (1) a report on the state-of-the-art fault tolerant techniques with a special focus on algorithm based fault tolerant (ABFT) techniques and (2) the design of a fully resilient linear algebra solver. The current document serves as the first deliverable.

Since soft errors and hard errors require different mechanisms for both detection and correction, most studies focus on designing either a soft error–tolerant application or a hard error–tolerant application. However, the work outlined here covers both, so we try to be as generic as possible when describing the algorithms, before giving specific details on how the techniques should be adapted specifically to soft or hard errors.

The rest of the paper is organized as follows. In Section 2, we present the state-of-the-art fault tolerant techniques for HPC applications. Section 3 is dedicated to ABFT techniques for silent error detection and correction, while mechanisms for hard fault recovery are discussed in Section 4. In Section 5, we present fault tolerant techniques specific to task-based applications and discuss how they can be combined with ABFT techniques to design a fully featured fault tolerant dense matrix factorization algorithm. Finally, we provide our concluding remarks in Section 6.

# 2 Review of HPC Fault Tolerant Techniques

Fault tolerance is a key feature required in many error-prone applications. The spectrum of fault tolerant techniques is wide, and these techniques differ based on the requirements and sensitivity of the target applications. For instance, while web service applications may afford using many replicates, and a critical embedded application may tolerate recomputing the same instruction two or three times to guarantee error-free results, HPC applications are very sensitive to variations in execution time and resource exploitation and require more efficient fault tolerance techniques. This section presents the most common fault tolerance techniques used in HPC applications and discusses their advantages and their drawbacks.

## 2.1 Checkpoint-restart techniques

In HPC, almost all fault-tolerant applications are based on checkpoint-restart techniques. The underlying principle consists of periodically saving data onto a reliable storage device such as a remote disk. The application can then recover from the most recent and consistent checkpoint whenever a fault occurs.

Checkpoint-restart techniques can be further classified into two categories: (1) system-level approaches and (2) user-level approaches. The system-level approach is commonly known as a "kernel-level checkpoint," and it provides resilience features without any change to the applications or libraries. The main advantage of this approach is its fine granularity access to memory, which is useful for optimizing the volume of data to checkpoint. As an out-of-the-box approach, it does not take advantage of application properties since it aims at handling any generic application.

On the other hand, the user-level checkpoint technique may take advantage of any relevant application features. While these techniques require more analysis and maybe some application modification, they have the advantage of reducing the checkpointing overhead because they can be tuned to match each specific application.

### 2.1.1 Coordinated checkpoint versus uncoordinated checkpoint

There are different ways to design a checkpoint scheme. On one hand, there are coordinated techniques which consist of synchronizing all processes in order to perform periodic checkpoints, simultaneously. Coordinated checkpointing techniques are more likely to be

the best candidates for system-level checkpointing, because they may be application agnostic. For example, the Local Area Multicomputer (LAM) Message Passing Interface (MPI) checkpoint-restart support [67] is based on the Berkeley Lab Checkpoint-Restart (BLCR) [35] kernel, which implements system-level coordinated checkpointing techniques.

Although implementing coordinated checkpointing is fairly straight forward, the extra synchronizations introduced by the coordination can significantly degrade application performance, as was pointed out in 1992 by Elnozahy et al. [37] and confirmed in 2008 by Liu et al. [54]. On the other hand, non-blocking coordinated checkpoints have been introduced by Cappello et al. [30] and El-Sayed et al. [36] in an attempt to mitigate synchronization overhead. The basic idea is to perform consistent checkpoints without blocking all the processes during the checkpoint. The checkpoint can then be managed by a coordinator, and an active process may delay its checkpoint to avoid interrupting ongoing computation. For full synchronization avoidance, an uncoordinated checkpoint is combined with message logging protocols, which save exchanged messages and their corresponding chronology to external storage [43]. When a process fails, the process can be restarted from its initial state or from a recent checkpoint. The logged messages can be replayed in the same order to guarantee identical, pre-fault behavior. For more details, we refer readers to the three main classes of message logging protocols: (1) the optimistic protocols [31, 70], (2) the pessimistic protocols [14, 49], and (3) the causal protocols [3, 65].

The fault tolerant MPI based on the combination of uncoordinated checkpoint and message logging protocols have been successfully implemented in MPICH, a high performance implementation of the MPI standard. The fault-tolerant MPICH, MPICH-V [15], is an automatic fault tolerant MPI–level technique based on an optimal combination of uncoordinated checkpoint-restart and message logging schemes. The fault tolerant techniques proposed by MPICH-V are transparent to the application. With respect to the MPI standard, MPICH-V does not require any modification in existing MPI application codes, but instead only requires re-linking the codes with the MPICH-V library. The first version, MPICH-V1, uses a remote pessimistic message logging protocol. The pessimistic message logging protocol has also been implemented in the Open Multi-Processing (OpenMP) application programming interface (API) [18]. Some performance penalty issues exhibited by the pessimistic approach have been addressed in the second version, MPICH-V2 [19]. Despite the improvement, MPICH-V2 suffers from the synchronizations intrinsic to pessimistic message logging protocols. This penalty is removed thanks to the causal message logging protocol implemented in the MPICH-V project [21].

### 2.1.2 Incremental checkpoint

In addition to the synchronization penalty associated with coordinated checkpointing techniques, there is another severe drawback: congestion on I/O resources when the checkpoints are written to disk [44]. This problem is discussed in [53, 63], and the authors propose techniques based on checkpoint data compression to reduce the checkpoint volume. Another approach to this problem is the practice of incremental checkpointing [61], which consists of optimizing the checkpoint volume by writing only modified data at memory-page granularity. The very first checkpointing might be a full checkpoint, but each subsequent checkpoint iteration includes only the data modified since the last checkpoint. While this approach seems attractive, it requires more frequent checkpoints to minimize the checkpoint volume. On the other hand, checkpointing at memory-page granularity may require a complex and time consuming algorithm to rollback to a consistent state when a fault occurs. A practical technique may limit the incremental checkpointing

to some specific data and then use full checkpointing whenever appropriate. This idea has been explored by Wang et al. in [72].

### 2.1.3 Diskless checkpoint techniques

The time spent writing the checkpoints to external disks represents an important part of the overhead associated with classical checkpointing techniques. One alternative uses fast nonvolatile memory to enhance checkpoint writing performance, as discussed in [58] and [57], where the authors propose a model using a solid state disk (SSD). This attractive solution is most beneficial for applications that require large storage capacity for the checkpoint data. On the other hand, when the checkpoint volume is not excessively large, node memory can be exploited in lieu of writing to disk. This approach is called *diskless checkpointing* [60], where the checkpoints are distributed on a node's main memory in a way that guarantees a recovery from one to a few faults. For example, a coordinated, diskless checkpoint technique that used extra nodes for checkpointing was first presented in 1994 by Jim Plank et al. in [64] and was then implemented in 1997 by James Plank et al. in [62] for well-known linear algebra algorithms, including Cholesky factorization, lower-upper (LU) factorization, QR factorization, and the preconditioned conjugate gradient (PCG) method. The algorithm forces each node to allocate a certain amount of memory to be used for checkpointing. Thus, each node writes its local checkpoint to its physical memory, and—using common encoding algorithms like those used in a redundant array of independent disks (RAID) [26]—a parity checkpoint is stored on extra nodes that are dedicated to checkpointing. In this model, failed nodes are recovered using checkpoints from the computing nodes and the parity nodes. However, this model can tolerate only a limited number of faults. In addition, when a computing node and a parity node fail simultaneously, lost data cannot be recovered. Another diskless checkpoint technique proposes a model in which each node stores its local checkpoint in a neighbor node's memory. This approach is called *buddy checkpointing*. A compromise between the encoding approach and the neighbor-based approach is proposed in [29].

### 2.1.4 Multilevel checkpointing

The traditional checkpointing technique is robust but requires external storage resources. In addition, coordinated checkpointing has to rollback *all* processes to recover from a single process fault, and it may not scale well in certain cases [25]. However, the diskless variant of checkpointing is promising for applications that do not require large checkpoint allocations, though it may require extra resources like memory and network bandwidth [28], which are precious resources in the context of HPC. In fact, the fault-rate metric (MTBF) commonly used in HPC, only provides information about overall reliability of the whole system, not the reliability (or failure susceptibility) of individual components. For this reason, using such a blunt metric for determining application checkpoint frequency could lead to sub-optimal results and consequently increase the checkpointing overhead. This issue is addressed in [5, 6] through the design of a multilevel checkpointing model. The authors consider the fault rate of each critical fault-prone component (e.g., main memory) and provide, for each critical component, a customized optimal checkpoint. The authors also suggest using different types of storage depending on the data being stored/protected. While stable storage (e.g., external disks) seems like a reasonable choice for node-level checkpointing, the authors proposed simple, on-node, diskless checkpointing to cope with transient errors in the main memory.

## 2.2 ABFT techniques

To overcome the drawbacks of the checkpoint-restart technique, application developers must focus on exploiting their applications' particularities in order to design the most appropriate fault tolerant techniques for a given application. An alternative approach may consist of investigating algorithms which are resilience friendly. For example, [42] proposed new fault tolerant super-scalable algorithms that can complete successfully despite node faults in parallel-distributed environments. Though not all applications can be redesigned to be inherently fault tolerant, some applications like meshless finite difference algorithms have demonstrated natural fault tolerance. A possible drawback of this approach is that it requires application modification and a deep involvement/investment from application developers. The imperative, though, is that to exploit extreme-scale machines, HPC application developers cannot continue to relegate fault tolerance to second place and expect general fault tolerant techniques to be used with low overhead. Application developers need to leverage the applications themselves, and the routines and algorithms within, to achieve efficient fault tolerance at extreme scales.

ABFT, a class of approaches that combines algorithmic techniques to detect and correct data loss or data corruption during computation, is one way to improve fault tolerance of the applications themselves. Whereas traditional fault-tolerant techniques save raw data in memory or on disk and restore that data when a fault occurs, ABFT techniques exploit mathematical properties to regenerate lost data. There are two classes of ABFT techniques: (1) checksum ABFT, which adds extra data and properties to the application that can be checked against the running application to detect and correct faults; and (2) checksum-less ABFT, which exploits mathematical properties intrinsic to the application itself to detect and recover data loss.

### 2.2.1 Checksum ABFT techniques

The checksum ABFT approach involves modifying algorithms to include extra data (a checksum) that can be used to recover from faults at a lower cost than traditional checkpoint-restart. The basic idea consists of keeping an invariant bijective relation between the extra encoded data and the original data through the execution of the algorithm. When a fault occurs, the extra encoded data are exploited to recover lost data. The checksums are computed to allow regeneration of not just data with the same numerical properties as the lost ones, but also to recompute exactly the data lost during a fault. Consequently, this approach is the best candidate for direct dense linear algebra solvers, which aim to compute exact solutions. The checksum ABFT technique was first introduced in 1984 by Abraham et al. [46] to design low-cost, fault-tolerant algorithms for matrix-based computations. The effectiveness of the technique was demonstrated in a parallel environment by the same author for signal processing on highly concurrent computing structures [50]. More recently, ABFT techniques have been reviewed for the purpose of designing resilient versions of different linear algebra solvers. The great advantage of ABFT is that it can be easily integrated into existing applications with affordable overhead in terms of labor.

### 2.2.2 Checksum-less ABFT techniques

Some applications have inherent mathematical properties that can be exploited for soft fault detection and/or correction. A simple approach would consist of checking the out-

put correctness at the end of the computation. However, to save computation time, Zizhong Chen introduced an on-line fault detection technique for Krylov subspace iterative methods to detect faults as soon as possible during application execution [27]. The basic premise involves a periodic check of the data/properties that the application should have and comparing this information against the application's current state to detect silent errors. This approach checks, for example, the orthogonality between vectors of the Krylov basis or the equality of $r^{(k)} = b - Ax^{(k)}$ between the iterate $x^{(k)}$ and the current residual $r^{(k)}$ that are true in exact arithmetic but are only satisfied up to a small relative perturbation in finite precision.

A similar approach was proposed by Langou et al. [52], to recover lost data in solvers of linear systems of equations, then extended by Agullo et al. [2] with variants of local recovery techniques and further numerical analysis. As reported in [1] such techniques are also relevant for eigensolvers. In general, the missing or corrupted entries are generated with the constraint of maintaining the application properties. These approaches are very attractive for iterative solvers since they require only a result that satisfies a given criterion but not an exact solution. Another attractive feature of this approach is that it has no overhead in a fault-free execution. However, not all applications have interesting data redundancy properties to rely on nor do all applications tolerate approximate solutions. In addition, these techniques are only for one class of silent errors, namely CPU errors, and do not cover bit flips in cache/memory.

## 2.3 Fault tolerance support in MPI

The primary goal of this work is to investigate ABFT for recovering lost data after a fault. However, these fault recovery algorithms cannot be effective without overarching system support to guarantee a consistent post-error state. One such system is MPI. In HPC, most parallel distributed applications rely on MPI to exchange data between computation nodes. Without a specific error handler associated with an MPI communicator, when an MPI process fails, the entire application will terminate. Alternatively, the user can replace the default error handler with `MPI_ERRORS_RETURN`, `MPI_Comm_set_errhandler(MPI_COMM_WORLD, MPI_ERRORS_RETURN)`, which reports the error to the user and lets the user make the recovery decision. This is not, however, a complete solution to a crashed process, where—for example—the neighbor processes should be notified, the communicator should be repaired, and ongoing communications should be correctly managed. Although the MPI Forum[1] is actively working to address each of these issues, at the time of this writing, there is no standard specification for fault recovery in MPI applications.

### 2.3.1 Checkpoint on failure protocol

To address the MPI standard's inability to deal with a process failure or a disabled communication system, a new MPI-level checkpointing technique, called the checkpoint-on-failure protocol (CoF), was initially proposed by Wesley Bland et al. in [11] and extended in [12]. Unlike conventional checkpointing techniques that perform checkpoints periodically, the CoF model's checkpointing technique is triggered only when a process failure is detected. The advantage of this technique is that the associated checkpointing overhead is small compared to periodic checkpointing approaches, and it does not induce

[1] http://www.mpi-forum.org

any checkpointing overhead when no faults occur. Assuming the MPI error handler is set to `MPI_ERRORS_RETURN`, the CoF protocol works as follows: when an error occurs and surviving processes have an error return code, they checkpoint their state and exit. The next step is restarting a new MPI instance and loading the checkpoints. However, since the checkpoints are saved after the failure, data from the failed process is still missing. Depending on the application, missing data can be regenerated with information from surviving processes using ABFT techniques or lossy approaches [52]. Therefore, the main limitation of the CoF protocol is that it cannot be implemented using a generic MPI application and is best suited for dense linear algebra kernels.

### 2.3.2 Fault Tolerant MPI

To the best of our knowledge, Faut Tolerant MPI (FT-MPI) [39, 38] is the first attempt at providing full MPI-level support for forward fault recovery. The core idea is to design a fault tolerant version of MPI with the capability , after a failure, to return a fully repaired MPI communicator to the user. Thus, an application based on FT-MPI, by providing lost data recovery algorithms, can survive process crashes and continue running with the same or fewer number of processes. FT-MPI provides four options to handle faults (Table 1), and the user can choose between these options based on the application properties and the desired path forward.

| Mode | Behavior |
|---|---|
| ABORT | All processes exit (Default MPI) |
| BLANK | Surviving processes continue with the same rank in the same communicator |
| SHRINK | A new communicator of small size is created for surviving processes |
| REBUILD | Failed processes are replaced in the same communicator |

Table 1: FT-MPI modes

Unfortunately, because FT-MPI changes some of the MPI semantics, it does not conform to the MPI-2 standard. And although the original FT-MPI project is no longer maintained, the effort was merged with the Open MPI project, and most of the progress continues under the User Level Failure Mitigation (ULFM) project [8, 10].

### 2.3.3 ULFM

The ULFM project started as a proposed solution for error handling within the MPI-3 standard [9]. The main focus of ULFM is to extend the MPI standard with new features to handle process failures without deadlock and sustain MPI communication beyond a failure. The initial proposal focused on three main features: (1) process failure detection, (2) fault notification, and (3) communicator recovery. Unlike FT-MPI, the ULFM project has a substantial standardization effort along with a prototype[2] that has been implemented in Open MPI. Below is a brief description of some new constructs supported by the ULFM effort.

**`MPI_Comm_revoke`** is a collective communication used to notify all surviving MPI processes that a fault occurred and that the communicator is therefore revoked. In addition, it ensures safe completion of all ongoing communications to prevent deadlock.

[2] http://fault-tolerance.org/

`MPI_Comm_shrink` is a collective communication that creates a new communicator that consists of surviving processes.

`MPI_Comm_failure_ack` is a local communication that acknowledges that the user was notified of all local processes that failed.

`MPI_Comm_failure_get_acked` is a local communication that gathers/lists all local processes that failed and were acknowledged.

`MPI_Comm_agree` is a collective communication that agrees on the value and the group of failed processes.

The above list is not exhaustive but is representative of ULFM's features. Note that to exploit all the fault handling provided by ULFM, it is mandatory to change the default error handler to: `MPI_Comm_set_errhandler(comm, MPI_ERRORS_RETURN)`. This enables the user to handle error codes for MPI routines and also call the relevant ULFM functions, based on the recovery policies, without unexpected exit or deadlock. ULFM shows a lot of promise in terms of standardization and is a complete framework to fully explore local recovery algorithms [71].

# 3 ABFT Techniques for Silient Error Detection and Correction

In HPC applications, silent data corruption (SDC) or soft errors [55] may lead to incorrect results and, consequently, influence critical decisions (unbeknownst to the user) with inaccurate data. In spite of the efforts made to design reliable memory using error correcting code (ECC), silent data corruption can still contaminate many applications. For instance, while ECC memory may detect and correct errors during data transmission, ECC cannot protect against data corruptions that occur during the actual floating point computations, making soft errors a significant and insidious hazard at any scale. Fortunately, there are some developments in this area, described below.

## 3.1 Soft error detection

One method for dealing with soft errors is the checksum-based ABFT strategy for silent error detection, which works as follows. For a given vector $x \in \mathbb{C}^n$, a checksum of $x$ (denoted as $x_c$) may be computed as $x_c = \sum_{i=1}^{n} x_i$, where $x_i =$ is the $i^{th}$ entry of $x$. For a $\sum_1^n x_i$, where $x_i$ is the $i^{th}$ entry of $x$. For a better understanding of ABFT schemes, let us consider the example of a matrix-vector multiplication $y = Ax$, where $A \in \mathbb{C}^{n \times n}$ is the coefficient matrix, $x \in \mathbb{C}^n$ is a given vector, and $y \in \mathbb{C}^n$ is the resulting vector. Their respective checksums may be encoded as shown in Figure 2.

During the computation, bit flips may occur in the entries of $A$, in the input vector $x$, or in the result vector $y$. For fault detection, extra information may be encoded in additional checksum rows/columns. The encoded row vector $A_{cc}$ and column vector $A_{rc}$ denote the checksum column of $A$ and the checksum row of $A$, respectively, with $A_{cc}(j) = \sum_{i=1}^{n} a_{ij}$ and $A_{cr}(i) = \sum_{j=1}^{n} a_{ij}$. In addition, a full checksum consisting of the summation of all the entries of $A$ may also be computed. To check the correctness of the matrix-vector multiplication, the checksum of $y$ ($y_c = \sum_1^n y_i$) is compared to $\tilde{y}_c = A_{cc}x$.

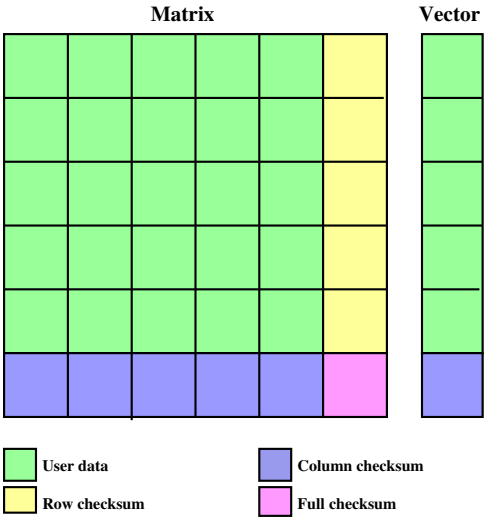



Figure 2: Linear checksum encoding for matrix-vector multiplication.

In exact arithmetic, $y_c$ is equal to $\tilde{y_c}$, and any difference may be reported as a fault. This approach is an efficient scheme for soft error detection. The fault may be accurately located by using the available checksums, because each checksum must satisfy a specific property (sum of the data encoded here). The basic checksum described here may help to correct a single entry corruption, whereas the weighted checksum method developed in [50] is commonly used for detection and correction of multiple faults. Peng Du et al. successfully implemented this approach by adapting a mathematical model that treats soft errors as rank-one perturbations during LU [33] and QR [34] factorizations. More details on the implementation of ABFT techniques for soft error detection in dense linear algebra kernels can be found elsewhere [32].

## 3.2 Soft errors and floating point arithmetic roundoff errors

In floating point arithmetic, roundoff errors of small magnitude may be confused with soft faults, as studied by Bliss et al. [13]. Consequently, ABFT techniques for fault detection may report roundoff errors as faults and waste computational time trying to locate the supposedly corrupt data. Conversely, with roundoff error propagation, a genuine soft error may become difficult to locate because of the numerical inaccuracy in the computation, which makes it very difficult to accurately detect soft errors. For this reason, ABFT techniques for soft error detection must be adapted to tolerate inaccuracies caused by roundoff errors in floating-point arithmetic. Rexford et al. discussed how the upper bound of roundoff errors can be used to provide efficient ABFT for fault detection and how to minimize roundoff errors in checksum encoding [66]. In general, distinguishing faults close to roundoff errors may be very challenging. However, in practice, ABFT techniques for soft error detection turn out to be very efficient, because numerical information from the target applications is often exploited to set an appropriate threshold.

# 4 ABFT Techniques for the Tiled Cholesky, LU, and QR Factorizations

This section describes how ABFT techniques can be adapted for use in a parallel distributed environment in order to recover data after a node crash/failure. Although the scope of our study here focuses on dense matrix factorization algorithms, the underlying ideas can be applied to most linear algebra algorithms. Also, even though the methods developed here could also be used in a shared memory environment, it would not be necessary, because an MPI process could still be recovered from the shared memory; only a mechanism for replacing the faulty process should be investigated for that use case. ABFT techniques for dense matrix one–sided factorization algorithms have been intensively investigated by Peng Du [32]. Here, we revisit the most relevant approaches and explain how they can be combined with other resilience techniques to design a fully featured, fault-tolerant solver.

## 4.1 ABFT for resilient dense matrix factorization

For a given matrix $A$, one-sided matrix factorizations, such as Cholesky factorization $[A = LL^T]$, QR factorization $[A = QR]$, and LU factorization $[A = LU]$, are the compute-intensive operations for solving a system of linear equations: $Ax = b$. These algorithms are time consuming (i.e., longer run time due to cubic time complexity) and are therefore more likely to experience faults. Here, we explain how ABFT techniques can be exploited to design resilient versions of matrix factorization algorithms: for the sake of generality and simplicity let's use $A = ZU$ notation to represent a one-sided matrix factorization, where $Z$ is the left matrix and $U$ is an upper triangular matrix. It also helps to consider a one-sided factorization as recursively applying $Z_i$ to the initial matrix $A$ from the left until $Z_i Z_{i-1} \dots Z_0 A$ becomes upper triangular. The following theorem is the key ingredient behind the success of ABFT techniques for matrix operations.

**Theorem 4.1.** *A checksum relationship established before ZU factorization is maintained during and after factorization.*

*Proof.* For a given matrix, $A \in \mathbb{R}^{n \times n}$, let $A = ZU$ denote its one-sided factorization, where $Z \in \mathbb{R}^{n \times n}$ and $U$ is an upper triangular matrix. Let $A_c = [A, \overline{A}] \in \mathbb{R}^{n \times n+\gamma}$ denote the original matrix augmented by the row-wise checksum matrix, $\overline{A} \in \mathbb{R}^{n \times \gamma}$, where $\gamma$ is the width of the checksum. In the ABFT algorithm, the factorization operations are applied to the matrix $A_c$. If we rewrite $U$ as $Z_n Z_{n-1} \dots Z_0 A = U$, it follows that $\forall i \in [1, n]$:

$$
\begin{aligned}
Z_i Z_{i-1} \dots Z_0 A_c &= Z_i Z_{i-1} \dots Z_0 [A, \overline{A}] && (1)\\
&= [Z_i Z_{i-1} \dots Z_0 A,\ Z_n Z_{n-1} \dots Z_0 \overline{A}] && (2)\\
&= [U^i, \overline{U}^i]. && (3)
\end{aligned}
$$

Since $Z_i Z_{i-1} \dots Z_0$ is a linear operation, the initial relationship between the input matrix $A$ and the corresponding matrix $\overline{A}$ is still preserved in $U$ and $\overline{U}$, which can be considered the checksum matrix of $U$. Consequently, at each iteration of the factorization, $i$, the factor, $U^i$, has its corresponding checksum matrix, $\overline{U}^i$, which can be used to regenerate lost portions of $U^i$. □

Theorem (4.1) remains valid for LU factorization with row pivoting, $PA = LU$, as long as each row permutation operation is applied to both the main matrix and the checksum matrix.

## 4.2 ABFT techniques and light checkpoints

As introduced above, the $A = ZU$ one-sided matrix factorization produces two matrices, the left factor $Z$ and the right factor $U$. The generic ABFT technique, described above, protects only $U$ since $U$ results from linear transformation of the initial matrix, $A$, whereas the left factor $Z$ remains vulnerable to faults. In the special case of Cholesky factorization, $A = LL^T$, where only one factor, $L$, is required, ABFT techniques are sufficient to design a fully resilient factorization algorithm. An implementation of the ABFT Cholesky factorization to recover from hard faults is proposed and evaluated in [45].

In the case of QR factorization, where the left factor $Q$ is required, additional techniques should be investigated to protect the matrix, $Q$. Since the matrix $Q = Q_0 Q_1 \dots Q_n$ grows by one column per iteration, a possible technique may consist of maintaining a column-wise checksum on the $Q_i$ vectors. However, if the left factor matrix, $Q^i$, is extended with a column-wise checksum matrix $\overline{Q^i}$, at the next iteration, the checksum relation will not be satisfied in the resulting matrix, $\left[\frac{Q^{i+1}}{\overline{Q^{i+1}}}\right]$. This is actually because $Q^{i+1}$ does not result from a linear transformation of $Q^i$. Consequently, the column-wise checksum of $Q^i$ should be stored separately and incremented at each iteration. Unlike traditional ABFT, this approach has a light, diskless, compressed checkpoint. Finally a fully resilient QR factorization can be achieved using a clever combination of ABFT techniques to protect the right factor, $R$, and light checkpoint techniques to protect the left factor, $Q$. A variant of this hybrid technique for a fault tolerant, dense QR factorization was successfully implemented in 2015 by Bouteiller et al. in [20] as an extension to the prior work [34].

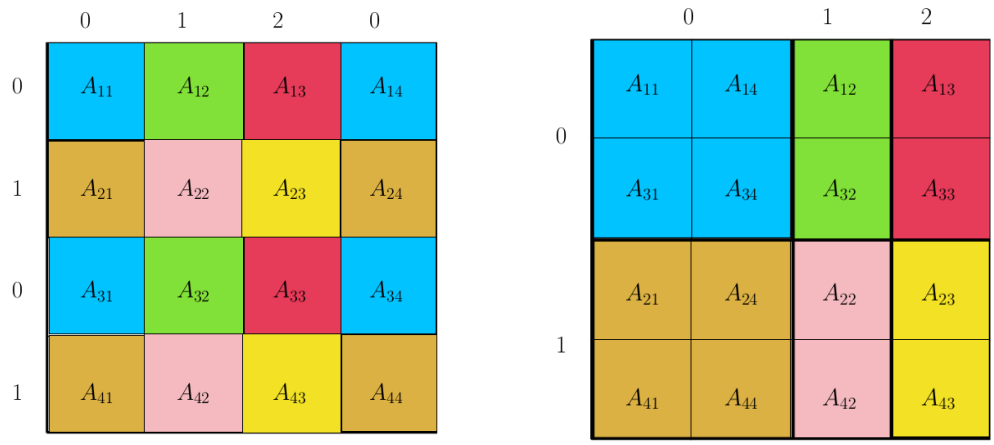
Similarly, protecting the left factor, $L$, in a partial row-pivoting LU factorization with a column-wise checksum, is challenging. Since the column-wise checksum entries will be appended to the matrix rows, the algorithm may be prone to select pivots in the checksum elements. This issue can be solved by revisiting the partial pivoting kernel to exclude checksum rows. Though this solution seems attractive, it may affect the numerical stability of the algorithm. While the checksum rows are considered when searching for a pivot, they contribute to the scaling. In cases where the pivot is significantly smaller when compared to some checksum entries, the algorithm may be prone to excessive pivot growth.

To sum up, while ABFT techniques are effective for protecting right factors in a one-sided matrix factorization, an additional mechanism should be investigated if the left factor is required. Light checkpoint schemes are an affordable option to consider.

## 4.3 ABFT techniques in parallel-distributed environments

To achieve a good performance and design a highly scalable software for parallel distributed architectures, matrices are commonly divided into square ($nb \times nb$) tiles, which are more likely to fit in fast memory like a CPU's L2 cache. To ensure load balancing across the system, these tiles are often distributed over the nodes in a 2-D, block-cyclic,

round-robin fashion as implemented in the Scalable Linear Algebra PACKage (ScaLAPACK) [7].




(a) Matrix view (b) Process view

Figure 3: A 2-D block distribution of a matrix with a $2 \times 3$ process grid.

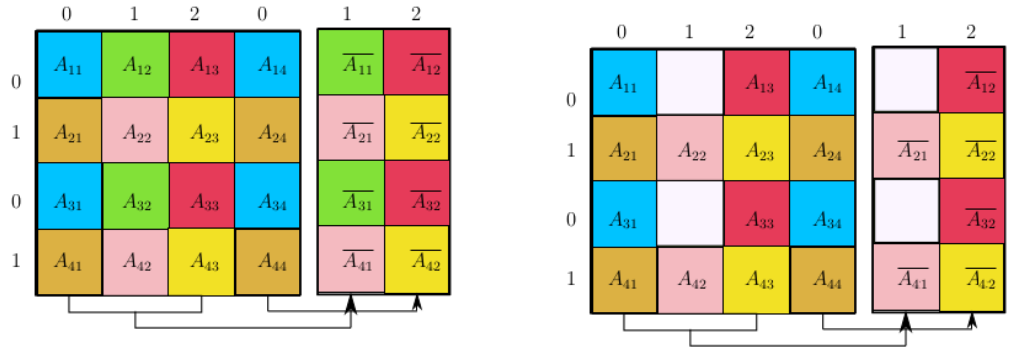
As shown in Figure 3, the input matrix has been divided into a $4 \times 4$ tile matrix and then distributed/mapped onto six processes configured in a $2 \times 3$ rectangular grid. Figure 3a shows the global matrix distribution, where a process is identified by a single color, and tiles of the same color are associated with a unique process. Each process then stores the tiles to which it is mapped, contiguously, as depicted in Figure 3b.

As an example, let's take an $m \times n$ matrix converted to an $mt \times nt$ tile layout, where $mt = \left\lceil \frac{m}{nb} \right\rceil$ is the number of tile rows, and $nt = \left\lceil \frac{n}{nb} \right\rceil$ is the number of tile columns. For each tile row, $it \in [1, mt]$, a row-wise checksum, $\overline{A_{it}} = \sum_{jt=1}^{jt=nt} A_{it,jt}$, can be used to regenerate only one tile per tile row. However, with the 2-D block cyclic data distribution technique, each process is more likely to hold more than one tile per row (Figure 3), where the process at Cartesian coordinate $(0, 0)$ holds the tiles $A_{1,1}$ and $A_{1,4}$, which belong to the same tile row.

To fully recover all data after a process crash, one may provide more than one tile checksum per tile row. Since the tile rows are distributed in a round-robin fashion among the $q$ processes, at the end of the first round each process has only one tile. Consequently, a row checksum that is limited to the tiles distributed in the first round may be successfully used to recover any of the corresponding tiles. A separate checksum must be computed for the tiles distributed in the second round and so on. To generalize, for a $p \times q$ 2-D process grid, $\left\lceil \frac{nt}{q} \right\rceil$ checksums must be computed per tile row to recover from a process crash.

Consider the matrix with four tiles per row distributed among three processes per row (Figure 3). We can see that a first checksum must be computed for the first three tiles, and that a separate checksum must be computed for the fourth tile. A checksum on a single tile is nothing more than a simple copy of the tile. To maintain load balance, the checksum columns are also distributed in a round-robin fashion as illustrated in Figure 4a. With such a distribution, a process may hold both a tile and its associated checksum (Figure 4). As a consequence of this distribution of the checksum, some data may be definitively lost. An affordable trade off consists of replicating (copying) the checksums, and the checksum copies are then also distributed in a round-robin fashion (Figure 5). In the case where the number of checksum columns, $\lceil nt \rceil q$, is a multiple of the number of processes per

column, $q$, each checksum and its replicate will be stored on the same process. However, this is unlikely to happen when $q$ is high enough, and if it does happen, one can shift the replicate storage elsewhere or use reordering schemes to resolve the problem.




(a) Original matrix with the associated checksums

(b) Fault in process (0,1)

Figure 4: Tile row-wise checksum of a matrix in 2-D block cyclic layout.

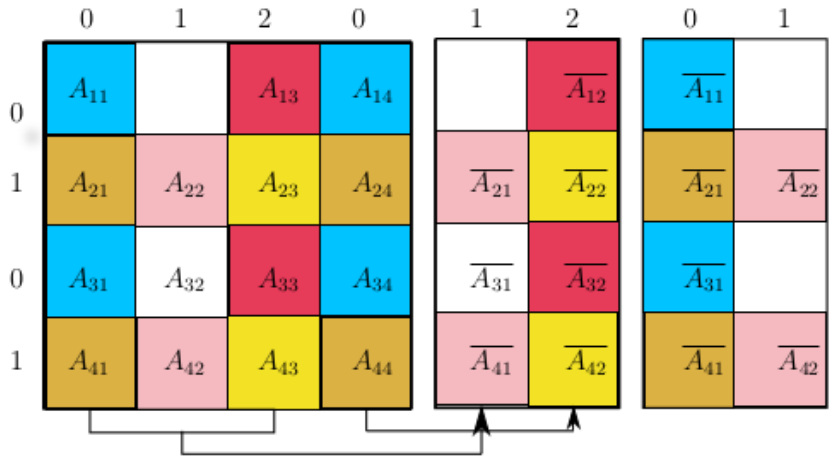



Figure 5: Replication technique for the tile row-wise checksum of a matrix in a 2-D block cyclic layout. The replicate of checksum $\overline{A_{11}}$ is now available on process $(0,0)$ and will help recover both the tile $A_{11}$ and the corresponding checksum on the failed process $(0,1)$.

While, in our example, the checksum requires as much memory as the matrix it protects, the memory requirement is rather affordable in practice. In total, the checksum matrix is of size $mt \times \lceil \frac{nt}{q} \rceil nb^2 = \frac{m}{nb} \times \lceil \frac{n}{nb \times q} \rceil nb^2 = \lceil \frac{m \times n}{q} \rceil$, since each checksum is of size $nb^2$. The ratio of the checksum size over the input matrix size is then $\frac{1}{q}$, which must be doubled because of the data replication. Finally, in terms of memory consumption, the checksum overhead is $\frac{2}{q}$ and decreases as $q$ increases.

## 4.4 ABFT for multiple faults recovery

Owing to increased scales and extended run times, many abnormal behaviors and crashes may occur during an HPC application run. These problems can occur in a variety of different circumstances with a myriad of different impacts/outcomes, including multiple faults. Multiple faults may be classified into three categories. The first scenario consists of

multiple isolated faults, where the time between two consecutive faults is large enough to recover from a fault before the next occurs. Hence, only one fault is mitigated at a time, and these faults are each addressed as an individual, single fault. The second scenario consists of at least two simultaneous faults in different process rows with, at most, one process crash per row. Here, the word "simultaneous" is used broadly as it refers to all situations where a new fault occurs while the previous fault is not yet completely fixed. This case is also easy to handle since each tile row has the associated checksums necessary for data recovery. Therefore, all the missing data can be recovered in parallel. The last scenario, which is also the most challenging, is when more than one process fails simultaneously in the same process row. A row-wise checksum is a linear combination of the row entries, and can therefore be considered as a linear system of equations; for this reason recovering a missing entry involves solving a linear system of one equation. Thus, with a given number, $nf$, of checksums on the same row, one can tolerate $nf$ faults by solving a linear system of $nf$ equations and $nf$ unknowns. The value of $nf$ is set based on a trade off between the desired level of reliability and the affordable overhead. It is important to note that, as $nf$ increases, roundoff errors and numerical stability issues can make this approach unreliable. However, the worst-case scenario is two failures hitting the same row, which makes this scenario less likely.

### 4.5 Two-Sided Decompositions in Distributed Memory and Hardware Accelerated Systems

Two-sided factorizations require application of similarity transformations to both sides of the system matrix in order to preserve the spectrum. They are used in computing eigendecompositions (either symmetric, generalized, and non-symmetric) and Singular Value Decomposition (SVD). Maintaining the checksum relationship becomes an even greater challenge but can be achieved in distributed memory systems for Hessenberg reduction [47] and systems with hardware accelerators for bi-diagonal reduction [48].

## 5 Fault Tolerant Task-Based Tiled Cholesky Factorization

Unlike the block column–oriented algorithms found in the Linear Algebra PACKage (LAPACK), tile algorithms operate at a fine granularity by dividing the whole matrix into small square tiles that are more likely to fit into fast memory (e.g., a CPU's L2 cache). While classical algorithms adapted to operate on tiles may provide a reasonable performance on very simple homogeneous architectures, it is challenging to obtain good performance from complex machines with Non Uniform Memory Access (NUMA), accelerators (e.g., GPUs, Xeon Phi), and other specialized components. The need to exploit complex and heterogeneous architectures at full efficiency led to the development of task-based programming models. The underlying idea of these models is to represent applications as a set of tasks to be executed. Under this model, applications are commonly represented in the form of a Direct Acyclic Graph (DAG) in which each node represents a task, while the edges represent the data dependencies between the tasks.

Once a task-based algorithm is designed, the next step is to provide a runtime system to track data dependencies and manage microtasks on distributed many-core heterogeneous architectures with architecture-aware task scheduling features. To this end, there

are generic runtime systems like StarPU [4], the Parallel Runtime Scheduling and Execution Controller (PaRSEC) [17], and the Symmetric Multiprocessor Superscalar (SMPS) programming envornment [59], to name a few.

To cope with faults in task-based applications, some task-based runtime systems provide resilience-friendly features. For task-based runtime systems that use static scheduling, resilience can be achieved through task duplication as proposed by Fechner et al. in the context of grid applications [40]. The main idea is to exploit the periods of time when processors are idle to execute duplicate tasks. Note that while some applications benefit considerably from such a technique, highly optimized applications that utilize computational resources at or near full efficiency can suffer a performance penalty. In the context of runtime systems based on dynamic scheduling, resilience schemes based on alternative versions and checkpointing were investigated in [56], while mechanisms based on task re-execution and work stealing were considered in [51].

In the rest of this section, we focus primarily on fault tolerant features that can be achieved/implemented using both StarPU and PaRSEC, because they are suitable for distributed applications and meet the requirements for our project. We also rely on the tile Cholesky factorization, designed in 2009 by Buttari et al. [23], to describe the different ways to exploit the DAG to design a resilient task-based algorithm using an example provided by Chongxiao et al. in [24]. Given an $n \times n$ symmetric positive–definite matrix, $A$, a Cholesky algorithm computes a lower triangular factor, $L$, such that $A = LL^T$ or an upper triangular factor such that $A = U^T U$. The building blocks of this algorithm consists of four kernels operating on tiles: Cholesky factorization (POTRF), triangular solve (TRSM), symmetric rank-k update (SYRK), and general matrix-matrix multiplication (GEMM). This algorithm is described in Figure 6 along with a snapshot of a $4 \times 4$ tile matrix at step $k = 2$.

```
1 for k = 1...nt do
2     A[k][k] ← POTRF(A[k][k]);
3     for m = k+1...nt do
4         A[m][k] ←
            TRSM(A[k][k], A[m][k])
5     for n = k+1...nt do
6         A[n][n] ←
            SYRK(A[n][k], A[n][n]);
7         for m = n+1...nt do
8             A[m][n] ←
                GEMM(A[n][k], A[k][m], A[m][n])
```

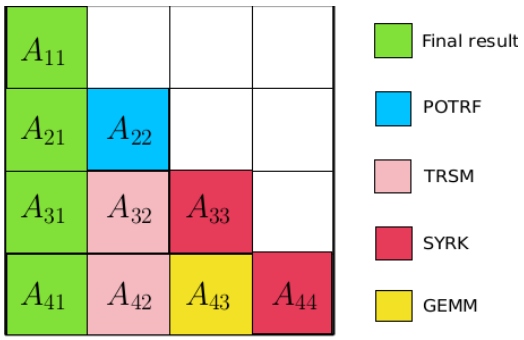
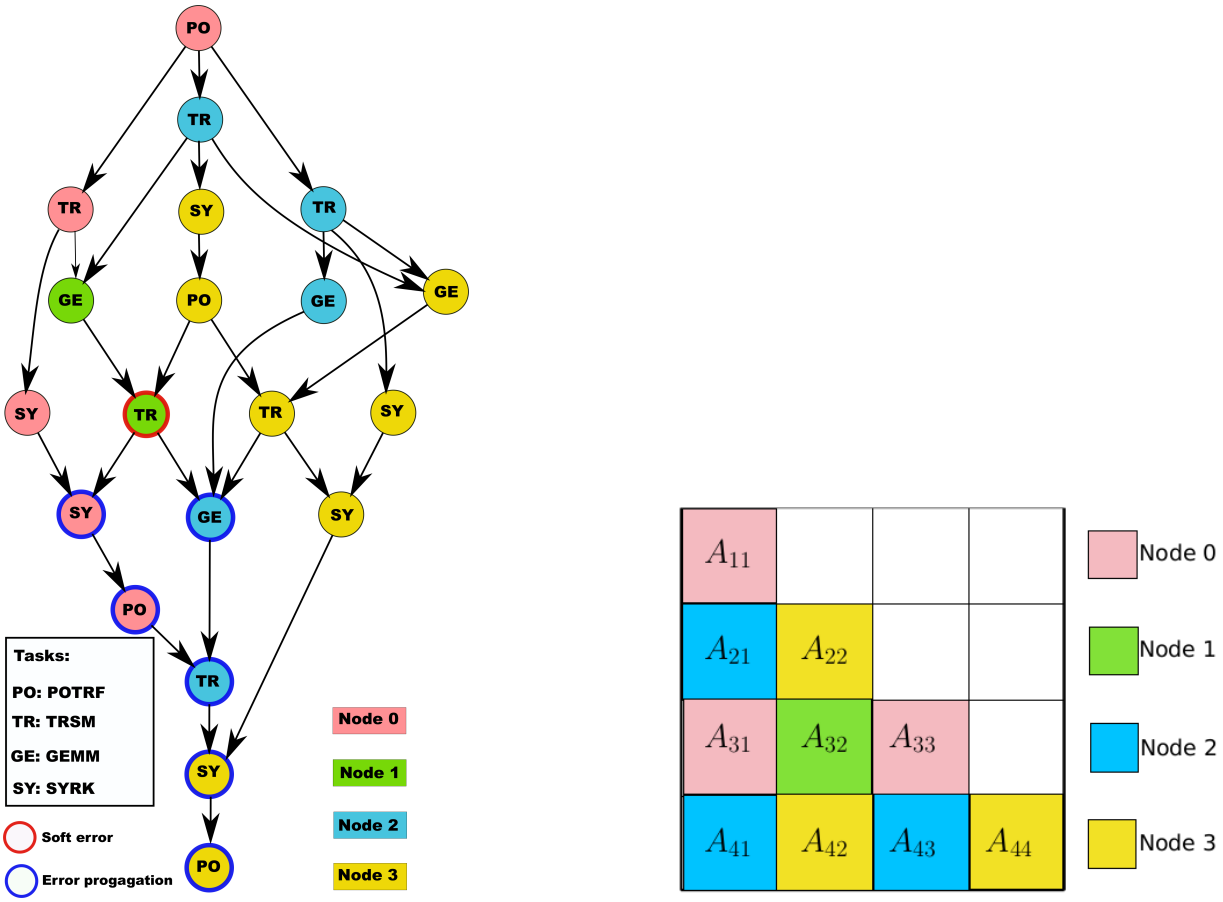



Figure 6: Tile Cholesky factorization algorithm ($A = LL^T$) on left, and the associated matrix view on right, applied to a $4 \times 4$ tile matrix at iteration $k = 2$.

For the sake of simplicity without loss of generality, we consider a task-based Cholesky factorization of a $4 \times 4$ tile matrix using a $2 \times 2$ process grid. When a soft fault occurs on a node, the output of the task in progress will be corrupted, and the wrong output will contaminate all the tasks that depend on data from the faulty task. This problem is illustrated in Figure 7a. A soft error corrupted the output of TRSM (the task circled in red) on node number 1, and the error has been propagated to all descendant tasks (tasks

circled in blue) up to the final result, thereby invalidating the whole computation. For data recovery, we assume that there is a separate mechanism in the system to detect and broadcast an error report to all the active nodes.




(a) DAG of Cholesky factorization of a $4 \times 4$ tile matrix.

(b) 2D block cyclic data distribution of a $4 \times 4$ tile matrix using a $2 \times 2$ process grid.

Figure 7: Illustration the impact of a soft error on a task-based algorithm using the tile Cholesky algorithm as an example. The soft error occurs on the node 1 during the triangular solve, then propagated in all the tasks that dependent on the data that is corrupted.

## 5.1 Data recovery using task re-execution

Here we describe and a fault-tolerant technique based on task re-execution initially introduced by Chongxiao et al. in [24]. Since task-based programming is an asynchronous model, the execution of tasks that are not directly affected by an error may progress, while the faulty task and its descendants will wait for the recovery before they resume. One way to recover from a soft error is to re-execute the failed task. However, the re-execution will require input data from predecessor tasks in the DAG, and without an explicit checkpoint of the data moving between tasks, the required input wont be available. Alternatively, all of the predecessors may be re-executed—and their predecessors in turn—up to the initial task, which has access to the initial input. The backward-task re-execution is possible if each node has access to the entire (or the relevant) portion of the DAG.

Task-based programming models can be classified into two main classes: (1) the sequential task flow (STF) model and (2) the parametrized task graph (PTG) model. In the STF model, task are inserted sequentially, data dependencies are detected using data access information, and each node unrolls the DAG. Consequently, each node has access to the entire DAG. This is the model implemented by StarPU. In the PTG model, tasks and their associated dependencies are expressed symbolically. Each node exploits this symbolic representation to extract the portion of the DAG relevant to the tasks it has to execute. This model is implemented by PaRSEC using a customized programming language called Job Data Flow (JDF) to let the user express data dependencies explicitly. To sum up, with either STF or PTG models, each node has enough information for successful task re-execution. So, once an error is detected, the creation of the sub-DAG required for the re-execution of the corrupted data (TRSM output in this case) will be initiated by the node that experienced the failure. The faulty task will then be replaced by the resulting sub-DAG as illustrated in Figure 8.

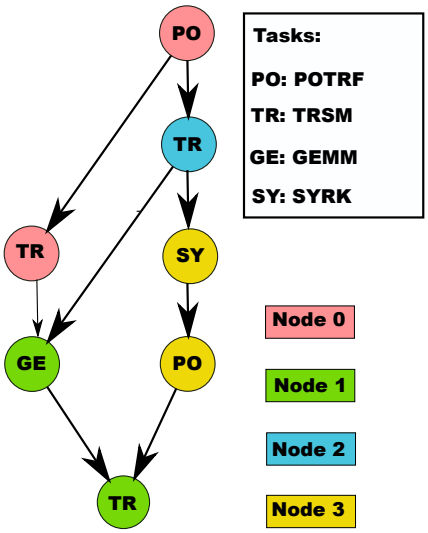



Figure 8: View of the sub-DAG used to re-execute and recover corrupted TRSM output.

In describing the recovery technique using task re-execution, we assume that any node can traverse the DAG backwards. However in practice, runtime systems do not keep a record of completed tasks because that overhead would not scale. Consequently, accessing information about predecessor tasks may be tricky and costly depending on the runtime system.

With runtime systems based on the STF model (e.g., StarPU), it is up to the application to mine its task submission loops for any relevant information on the faulty task's predecessors. Once all the tasks required for the recovery are identified, the next step is to include them in the existing DAG. While appending tasks is not a problem in StarPU, adding tasks in the middle of a string of tasks that, for instance, is modifying a piece of data, is not a trivial undertaking. To achieve this, one must use StarPU "tags," which act as waypoints points. These tags are also necessary because the new tasks will depend on the last submitted task that touched the data, which is probably much later in the DAG. Alternatively, one can also use tags to detach a faulty task from the DAG and plug in the recovery sub-DAG there, independently from the rest of the DAG.

As far as DAG handling is concerned, PaRSEC is very efficient. To recover corrupted data, one can take advantage of the PaRSEC's PTG representation to dynamically retrieve

all the predecessors of a failed task. The node hosting the failed task can then globally initiate the creation of the recovery sub-DAG by broadcasting the event to the other nodes. Since each node runs the scheduling engine, the node(s) can parse the PTG for relevant information about the recovery tasks.

The main advantage of this sub-DAG re-execution technique for fault recovery is that it does not induce any unnecessary overhead when no faults occur. On the other hand, while the recovery overhead may be affordable at the very beginning of the execution, it can increase the computation time by up to 100% if a fault occurs in the very last task, because this scenario requires re-executing the whole DAG from scratch. The full overhead analysis of this technique can be found in [24]. While we did not discuss a scenario with multiple faults here, the technique remains applicable in that case. Instead of recovering only one task, one may have to recover a few or many depending on the workload of the crashed process. This implies the re-execution of a relatively large sub-DAG, which can have an expensive performance penalty.

### 5.2 Recovery using a DAG snapshot

With a relatively high overhead, the task re-execution technique may seem like an unattractive option. However, by saving the intermediary data, the re-execution of the failed task only requires the "save" input from the failed task's immediate predecessors. All things being equal, this will drastically lower the recovery overhead regardless of the position in the DAG where the fault occurs. That said, to access the intermediary data, new tasks should be created to save the data that flows between tasks, and this step comes with additional overhead. To implement this algorithm, neither an external disk nor a checkpoint per every task is required. Instead, a diskless periodic checkpoint is an affordable trade off. To simplify the recovery of failed tasks, the checkpointing can be done at a task granularity (i.e., saving each task's output separately). When a task fails just after a checkpoint by its predecessors, the inputs required for the re-execution of the failed task can be retrieved directly. Otherwise, the recovery will require the re-execution of at most the $\beta$ previous tasks, where $\beta$ is the checkpoint interval. The overall recovery cost is strongly connected to the checkpoint interval, and rigorous study of optimal checkpoint intervals is available in [16].

While exploiting the resilient-friendly features offered by DAGs enables a task re-execution to recover from a fault, ABFT techniques are still applicable. A clever implementation could—depending on the application—combine each of the analyzed methods to design an efficient fault–tolerant solver.

## 6 Conclusions

Many scientific and engineering applications require the solution of linear systems of equations. In this work, we studied state-of-the-art, fault-tolerant techniques suitable for HPC applications and discussed how they can be adapted and combined to design fault-tolerant numerical linear algebra solvers. We demonstrated that, while ABFT techniques enable implementing resilient matrix factorization at an affordable cost, they can only protect the right factor. For this reason, we then studied a hybrid technique for exploiting ABFT and light checkpoint techniques to design a fully featured fault-tolerant matrix-factorization kernel. Both soft error and hard error implementation details were also discussed before we considered a task-based application and described how DAG capabilities can be

exploited to recover from faults using task re-execution.

The fault-tolerant techniques investigated in this work are attractive and have considerable potential from a theoretical point of view. In future work, we will provide experimental material and assess the effectiveness of the most promising techniques.

# References

[1] Emmanuel Agullo, Luc Giraud, Pablo Salas, and Mawussi Zounon. Interpolation-restart strategies for resilient eigensolvers. *SIAM J. Scientific Computing*, 38(5), 2016.

[2] Emmanuel Agullo, Luc Giraud, and Mawussi Zounon. On the resilience of parallel sparse hybrid solvers. In *22nd IEEE International Conference on High Performance Computing, HiPC 2015, Bengaluru, India, December 16-19, 2015*, pages 75–84, 2015.

[3] Lorenzo Alvisi and Keith Marzullo. Message logging: Pessimistic, optimistic, causal, and optimal. *IEEE Trans. Softw. Eng.*, 24(2):149–159, February 1998.

[4] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. *Euro-Par 2009 Parallel Processing*, pages 863–874, 2009.

[5] Anne Benoit, Aurélien Cavelan, Valentin Le Fèvre, Yves Robert, and Hongyang Sun. Towards optimal multi-level checkpointing. *IEEE Transactions on Computers*, 66(7):1212–1226, 2017.

[6] Anne Benoit, Aurélien Cavelan, Yves Robert, and Hongyang Sun. Multi-level checkpointing and silent error detection for linear workflows. *Journal of Computational Science*, 2017.

[7] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[8] W. Bland, A. Bouteiller, T. Herault, J. Hursey, G. Bosilca, and J.J. Dongarra. An evaluation of User-Level failure mitigation support in MPI. *Computing*, 95(12):1171–1184, December 2013. DOI: 10.1007/s00607-013-0331-3, `https://link.springer.com/article/10.1007/s00607-013-0331-3`.

[9] Wesley Bland, George Bosilca, Aurelien Bouteiller, Thomas Herault, and Jack Dongarra. A proposal for User-Level Failure Mitigation in the MPI-3 Standard. Technical report, Tech. rep., Department of Electrical Engineering and Computer Science, University of Tennessee, 2012.

[10] Wesley Bland, Aurelien Bouteiller, Thomas Herault, George Bosilca, and Jack Dongarra. Post-failure recovery of mpi communication capability: Design and rationale. *The International Journal of High Performance Computing Applications*, 27(3):244–254, 2013.

[11] Wesley Bland, Peng Du, Aurelien Bouteiller, Thomas Herault, George Bosilca, and Jack Dongarra. A checkpoint-on-failure protocol for algorithm-based recovery in standard mpi. *Euro-Par 2012 Parallel Processing*, pages 477–488, 2012.

[12] Wesley Bland, Peng Du, Aurelien Bouteiller, Thomas Herault, George Bosilca, and Jack J Dongarra. Extending the scope of the checkpoint-on-failure protocol for forward recovery in standard mpi. *Concurrency and computation: Practice and experience*, 25(17):2381–2393, 2013.

[13] W.G. Bliss, M.R. Lightner, and B. Friedlander. Numerical properties of algorithm-based fault-tolerance for high reliability array processors. In *Signals, Systems and Computers, 1988. Twenty-Second Asilomar Conference on*, volume 2, pages 631–635, 1988.

[14] Anita Borg, Jim Baumbach, and Sam Glazer. A message system supporting fault tolerance. *SIGOPS Oper. Syst. Rev.*, 17(5):90–99, October 1983.

[15] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov. MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. In *Supercomputing, ACM/IEEE 2002 Conference*, pages 29–29, Nov 2002.

[16] George Bosilca, Aurélien Bouteiller, Elisabeth Brunet, Franck Cappello, Jack Dongarra, Amina Guermouche, Thomas Herault, Yves Robert, Frédéric Vivien, and Dounia Zaidouni. Unified model for assessing checkpointing protocols at extreme-scale. *Concurrency and Computation: Practice and Experience*, 26(17):2772–2791, 2014.

[17] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack J Dongarra. Parsec: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering*, 15(6):36–45, 2013.

[18] Aurelien Bouteiller, George Bosilca, and Jack Dongarra. Redesigning the message logging model for high performance. *Concurrency and Computation: Practice and Experience*, 22(16):2196–2211, 2010.

[19] Aurélien Bouteiller, Franck Cappello, Thomas Herault, Géraud Krawezik, Pierre Lemarinier, and Frédéric Magniette. MPICH-V2: A fault tolerant MPI for volatile nodes based on pessimistic sender based message logging. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, SC '03, pages 25–, New York, NY, USA, 2003. ACM.

[20] Aurelien Bouteiller, Thomas Herault, George Bosilca, Peng Du, and Jack Dongarra. Algorithm-based fault tolerance for dense matrix factorizations, multiple failures and accuracy. *ACM Transactions on Parallel Computing*, 1(2):10, 2015.

[21] Aurelien Bouteiller, Thomas Hérault, Géraud Krawezik, Pierre Lemarinier, and Franck Cappello. MPICH-V Project: A multiprotocol automatic fault-tolerant MPI. *IJHPCA*, pages 319–333, 2006.

[22] Patrick G. Bridges, Kurt B. Ferreira, Michael A. Heroux, and Mark Hoemmen. Fault-tolerant linear solvers via selective reliability. *CoRR*, abs/1206.1390, 2012.

[23] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38–53, 2009.

[24] Chongxiao Cao, Thomas Herault, George Bosilca, and Jack Dongarra. Design for a soft error resilient dynamic task-based runtime. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 765–774. IEEE, 2015.

[25] Franck Cappello, Henri Casanova, and Yves Robert. Preventive migration vs. preventive checkpointing for extreme scale supercomputers. *Parallel Processing Letters*, pages 111–132, 2011.

[26] Peter M Chen, Edward K Lee, Garth A Gibson, Randy H Katz, and David A Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys (CSUR)*, 26(2):145–185, 1994.

[27] Zizhong Chen. Online-ABFT: an online algorithm based fault tolerance scheme for soft error detection in iterative methods. *ACM SIGPLAN Notices*, 48(8):167–176, 2013.

[28] Zizhong Chen and Jack Dongarra. Algorithm-based checkpoint-free fault tolerance for parallel matrix computations on volatile resources. In *Proceedings of the 20th international conference on Parallel and distributed processing*, IPDPS'06, pages 97–97, Washington, DC, USA, 2006. IEEE Computer Society.

[29] Zizhong Chen, Graham E Fagg, Edgar Gabriel, Julien Langou, Thara Angskun, George Bosilca, and Jack 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*, pages 213–223. ACM, 2005.

[30] Camille Coti, Thomas Herault, Pierre Lemarinier, Laurence Pilard, Ala Rezmerita, Eric Rodriguez, and Franck Cappello. Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[31] Om P. Damani and Vijay Garg. How to recover efficiently and asynchronously when optimism fails. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 108–115. IEEE, 1996.

[32] Peng Du. Hard and soft error resilience for one-sided dense linear algebra algorithms, 2012.

[33] Peng Du, Piotr Luszczek, and Jack Dongarra. High performance dense linear system solver with soft error resilience. In *2011 IEEE International Conference on Cluster Computing*, pages 272–280, Los Alamitos, CA, USA, 2011. IEEE Computer Society.

[34] Peng Du, Piotr Luszczek, Stan Tomov, and Jack Dongarra. Soft error resilient QR factorization for hybrid system with GPGPU. In *Proceedings of the second workshop on Scalable algorithms for large-scale systems*, ScalA '11, pages 11–14, New York, NY, USA, 2011. ACM.

[35] Jason Duell. The design and implementation of Berkeley Lab's linux checkpoint/restart. Technical report, LBNL, 2003.

[36] G.A El-Sayed and K.A Hossny. A distributed counter-based non-blocking coordinated checkpoint algorithm for grid computing applications. In *Advances in Computational Tools for Engineering Applications (ACTEA), 2012 2nd International Conference on*, pages 80–85, Dec 2012.

[37] E.N. Elnozahy, D.B. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. In *11th Symposium on Reliable Distributed Systems, 1992. Proceedings.*, pages 39–47, Oct 1992.

[38] Graham E. Fagg and Jack Dongarra. FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. In *Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 346–353, London, UK, 2000. Springer-Verlag.

[39] Graham E Fagg, Edgar Gabriel, Zizhon Chen, Thara Angskun, George Bosilca, Antonin Bukovsky, and Jack J Dongarra. Fault tolerant communication library and applications for high performance computing. In *LACSI Symposium*, pages 27–29, 2003.

[40] Bernhard Fechner, Udo Honig, Jorg Keller, and Wolfram Schiffmann. Fault-tolerant static scheduling for grids. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–6. IEEE, 2008.

[41] Ana Gainaru, Franck Cappello, Marc Snir, and William Kramer. Fault prediction under the microscope: A closer look into HPC systems. *SC Conference*, 0:1–11, 2012.

[42] Al Geist and Christian Engelmann. Development of naturally fault tolerant algorithms for computing on 100,000 processors, 2002.

[43] Jim Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481, London, UK, UK, 1978. Springer-Verlag.

[44] A. Guermouche, T. Ropars, E. Brunet, M. Snir, and F. Cappello. Uncoordinated checkpointing without domino effect for send-deterministic MPI applications. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 989–1000, May 2011.

[45] Doug Hakkarinen and Zizhong Chen. Algorithmic Cholesky factorization fault recovery. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–10. IEEE, 2010.

[46] Kuang-Hua Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Trans. Comput.*, 33:518–528, June 1984.

[47] Yulu Jia, George Bosilca, Piotr Luszczek, and Jack J. Dongarra. Parallel Reduction to Hessenberg Form with Algorithm-based Fault Tolerance. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 88:1–88:11, New York, NY, USA, 2013. ACM.

[48] Yulu Jia, Piotr Luszczek, George Bosilca, and Jack J. Dongarra. CPU-GPU hybrid bidiagonal reduction with soft error resilience. In *Proceedings of ScalA13*, Denver, CO, USA, November 2013.

[49] David B Johnson and Willy Zwaenepoel. Sender-based message logging, 1987.

[50] J.-Y. Jou and J.A. Abraham. Fault-tolerant matrix arithmetic and signal processing on highly concurrent computing structures. *Proceedings of the IEEE*, 74(5):732–741, May 1986.

[51] Mehmet Can Kurt, Sriram Krishnamoorthy, Kunal Agrawal, and Gagan Agrawal. Fault-tolerant dynamic task graph scheduling. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 719–730. IEEE Press, 2014.

[52] Julien Langou, Zizhong Chen, George Bosilca, and Jack Dongarra. Recovery Patterns for Iterative Methods in a Parallel Unstable Environment. *SIAM J. Sci. Comput.*, 30:102–116, November 2007.

[53] C.-C .J. Li and W.K. Fuchs. Catch-compiler-assisted techniques for checkpointing. In *Fault-Tolerant Computing, 1990. FTCS-20. Digest of Papers., 20th International Symposium*, pages 74–81, June 1990.

[54] Yudan Liu, R. Nassar, C.B. Leangsuksun, N. Naksinehaboon, M. Paun, and S.L. Scott. An optimal checkpoint/restart model for a large scale high performance computing system. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2008)*, pages 1–9, April 2008.

[55] Franklin T. Luk and Haesun Park. An analysis of algorithm-based fault tolerance techniques. *J. Parallel Distrib. Comput.*, 5(2):172–184, April 1988.

[56] Pierre Mouallem, Daniel Crawl, Ilkay Altintas, Mladen A. Vouk, and Ustun Yildiz. A fault-tolerance architecture for Kepler-Based distributed scientific workflows., 2010.

[57] Xiang Ni, Esteban Meneses, and Laxmikant V Kalé. Hiding checkpoint overhead in hpc applications with a semi-blocking algorithm. In *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*, pages 364–372, 2012.

[58] Xiangyong Ouyang, Sonya Marcarelli, and Dhabaleswar K. Panda. Enhancing checkpoint performance with staging io and ssd. In *Storage Network Architecture and Parallel I/Os (SNAPI), 2010 International Workshop on*, pages 13–20. IEEE, 2010.

[59] Josep M Perez, Rosa M Badia, and Jesus Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Cluster Computing, 2008 IEEE International Conference on*, pages 142–151. IEEE, 2008.

[60] J. S. Plank, K. Li, and M. A. Puening. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):972–986, October 1998.

[61] J. S. Plank, J. Xu, and R. H. B. Netzer. Compressed differences: An algorithm for fast incremental checkpointing. Technical Report CS-95-302, University of Tennessee, August 1995.

[62] James S. Plank, Youngbae Kim, and Jack J. Dongarra. Fault-tolerant matrix operations for networks of workstations using diskless checkpointing. *J. Parallel Distrib. Comput.*, 43:125–138, June 1997.

[63] J.S. Plank and K. Li. ICKP: a consistent checkpointer for multicomputers. *Parallel Distributed Technology: Systems Applications, IEEE*, 2(2):62–67, Summer 1994.

[64] J.S. Plank and Kai Li. Faster checkpointing with N+1 parity. In *Fault-Tolerant Computing, 1994. FTCS-24. Digest of Papers., Twenty-Fourth International Symposium on*, pages 288–297, June 1994.

[65] S. Rao, L. Alvisi, and H.M. Vin. The cost of recovery in message logging protocols. *Knowledge and Data Engineering, IEEE Transactions on*, 12(2):160–173, Mar 2000.

[66] J. Rexford and N.K. Jha. Algorithm-based fault tolerance for floating-point operations in massively parallel systems. In *Circuits and Systems, 1992. ISCAS '92. Proceedings., 1992 IEEE International Symposium on*, volume 2, pages 649–652 vol.2, May 1992.

[67] Sriram Sankaran, Jeffrey M Squyres, Brian Barrett, Vishal Sahay, Andrew Lumsdaine, Jason Duell, Paul Hargrove, and Eric Roman. The lam/mpi checkpoint/restart framework: System-initiated checkpointing. *The International Journal of High Performance Computing Applications*, 19(4):479–493, 2005.

[68] Bianca Schroeder and Garth A. Gibson. A Large-scale study of failures in high-performance computing systems. *IEEE Transactions on Dependable and Secure Computing*, 7(4):337–351, 2010.

[69] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM errors in the wild: A large-scale field study. *SIGMETRICS Perform. Eval. Rev.*, 37(1):193–204, June 2009.

[70] Rob Strom and Shaula Yemini. Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.*, 3(3):204–226, August 1985.

[71] Keita Teranishi and Michael A. Heroux. Toward local failure local recovery resilience model using MPI-ULFM. In *Proceedings of the 21st European MPI Users' Group Meeting*, EuroMPI/ASIA '14, pages 51:51–51:56, New York, NY, USA, 2014. ACM.

[72] Chao Wang, Frank Mueller, Christian Engelmann, and Stephen L. Scott. Hybrid full/incremental checkpoint/restart for MPI jobs in HPC environments. In *Dept. of Computer Science, North Carolina State University*, 2009.