Design for a Soft Error Resilient Dynamic Task-based Runtime

Chongxiao Cao*, Thomas Herault*, George Bosilca*, Jack Dongarra*†‡
*University of Tennessee, Knoxville, USA
†Oak Ridge National Laboratory, Oak Ridge, USA
‡University of Manchester, Manchester, UK

***Abstract*—As the scale of modern computing systems grows, failures will happen more frequently. On the way to Exascale a generic, low-overhead, resilient extension becomes a desired aptitude of any programming paradigm. In this paper we explore three additions to a dynamic task-based runtime to build a generic framework providing soft error resilience to task-based programming paradigms. The first recovers the application by re-executing the minimum required sub-DAG, the second takes critical checkpoints of the data flowing between tasks to minimize the necessary re-execution, while the last one takes advantage of algorithmic properties to recover the data without re-execution. These mechanisms have been implemented in the PaRSEC task-based runtime framework. Experimental results validate our approach and quantify the overhead introduced by such mechanisms.**

I. Introduction

Over the past few years, fault tolerance has become a major concern in High Performance Computing (HPC) area. According to [1], the broad consensus in the community is that Exascale systems, which are expected by 2018-2022, will be subject to errors or faults much more frequently than current Petascale systems. The two main reasons behind this theory are the increase of number of components necessary to reach the scale, coupled with an increase of Mean Time To Failure (MTTF) of each components that will unlikely be high enough to compensate for the first reason.

In the past, most of the development efforts on fault tolerance in HPC community has focused on the *fail-stop* model, where the failed processes stop working and the corresponding data is lost. The large scale, and the continuous increase of memory used by applications, have highlighted another type of faults, known as *soft errors* or *silent data corruption (SDC)*. Soft errors usually manifests as bit-flips in disk, memory or processor registers and are due to any of the following causes: temperature and fluctuations, cosmic particles, electrostatic discharge, etc… [2]. [3] reports that double bit flips, which cannot be corrected by Error Correction Control (ECC), occurs daily in Oak Ridge National Lab's Cray XT5, and [4] states that significant soft error rates were observed on BG/L's unprotected L1 cache.

In critical operations, *SDC*s are sometimes masked using voting techniques that require triplicating (or more) the computations, reducing even more the overall efficiency [5]. A traditional strategy to tolerate such failures together with fail-stop failures involves a combination of replication to detect the occurrence of *SDC*s, and checkpoint/restart to recover from a valid state if such fault is detected [6], [7]. This technique is highly automatic but suffers relatively high checkpointing overheads when saving data to stable storage [8], and induces a low resource efficiency as all computations must be duplicated. Duplication can be avoided if application-specific data validation mechanisms exist to guarantee the validity of a checkpoint [9]. However, the costs incurred with storing the checkpoint data and restarting the whole application once a failure is detected remain.

Today's large scale systems, as well as the next generations, feature other sources of complexity: multicore architectures with Non Uniform Memory Access (NUMA), use of accelerators (GPU, manycore computing), complex network hierarchies. They represent challenges to efficiently execute applications at large scale. Recently, task-based programming frameworks have emerged as a solution to address this challenge. A task-based application represents the algorithm as a set of tasks and data dependencies between these tasks. A runtime system is then used to schedule the tasks and manipulate the data based on the data dependencies, and adapt the execution to the platform characteristics as the algorithm unfolds.

This work proposes a fault tolerant design that provides *SDC* resilience for a dynamic task-based runtime. Our goal is to provide generic and low-overhead strategies to recover from *SDC* during the execution. We focus on recovery from application data corruption, which represents the largest part of the data, and thus the most probable source of *SDC*, and assume that information of application's task graph, or other critical parts are made resilient through other generic methods (e.g. triple modular redundancy as in [10]). We study three possible mechanisms at two levels of granularity: at the coarse level, tasks are re-executed if needed, and at the fine level, tasks are augmented to introduce the necessary protection against *SDC*. These mechanisms support shared-memory and distributed-memory platforms seamlessly. We develop and implement them in the PaRSEC [11] framework, which employs a data flow programming and execution model to efficiently schedule and manage micro-tasks on distributed many-core heterogeneous architectures. We illustrate our design on the Cholesky factorization, which is widely used in linear least squares regression and Monte Carlo simulations in large scale scientific applications. Experimental results demonstrate that our design introduces small overheads, close to the theoretical overheads, to recover from soft errors.

II. Background

This section briefly introduces the dynamic task-based scheduling framework in PaRSEC and our strategy to implement soft error resilience in PaRSEC.

A. Task-Based Scheduling using PaRSEC

Task Graphs have a long history of being used to express task dependencies and to schedule tasks on computing resources. In this paper, a task graph is defined as a Directed Acyclic Graph (DAG) $D = (V, E)$, where every vertex $v \in V$ represents a task (a set of sequential computations), and every edge $(v_1, v_2) \in E$ represents a data dependency between an output of task v_1 and an input of task v_2. If an edge (v_1, v_2) exists in E, then task v_1 must complete its execution and its output data transferred where task v_2 executes before task v_2 starts.

The Parallel Runtime Scheduling and Execution Controller (PaRSEC) is a generic framework for architecture-aware scheduling and management of micro-tasks on distributed many-core heterogeneous architectures. PaRSEC consists of a runtime environment that comprises a distributed multi-level dynamic scheduler, an asynchronous communication engine and a data dependencies engine [12]. The runtime will map the tasks on the data distribution, detect local and remote data dependencies and use dynamic asynchronous distributed scheduling to enable a task-based application to reach the maximum amount of parallelism. PaRSEC uses task graphs to represent the data flow of the algorithm.

While most of the techniques introduced are generic, in this paper, we will illustrate the design of fault tolerance using the tiled Cholesky factorization [13]. This algorithm factors an $N \times N$, symmetric, positive-definite matrix A into the product of a lower triangular matrix L and its transpose, i.e., $A = LL^T$ (or $A = U^T U$, where U is upper triangular). We implement it using a tiled linear algebra algorithm in which linear algebra operations are represented as a set of tasks that operate on square blocks of data (the tiles), and are dynamically scheduled based on the dependencies among them and on the availability of computational resources.

Algorithm 1 describes the tiled Cholesky factorization algorithm and Figure 1 shows the snapshot of a 4×4 matrix at step $k = 1$. The algorithm is based on four computational kernels: **POTRF** (Cholesky factorization), **TRSM** (triangular solver), **SYRK** (symmetric rank-k update) and **GEMM** (general matrix-matrix multiplication) that each operate on a tile (the matrix A is tiled in $NT \times NT$ tiles of size $nb \times nb$, and $A[m][n]$ represents a whole tile of A). Note that as the Cholesky factorization operates on a symmetric matrix, only the lower triangular part of the input matrix is stored, as shown in Figure 1.

Algorithm 1: Tiled Cholesky Factorization Algorithm

```
for $k = 0...NT-1$ do
    $A[k][k] \leftarrow POTRF(A[k][k])$
    for $m = k+1...NT-1$ do
        $A[m][k] \leftarrow TRSM(A[k][k], A[m][k])$
    for $n = k+1...NT-1$ do
        $A[n][n] \leftarrow SYRK(A[n][k], A[n][n])$
        for $m = n+1...NT-1$ do
            $A[m][n] \leftarrow GEMM(A[m][k], A[n][k], A[m][n])$
```

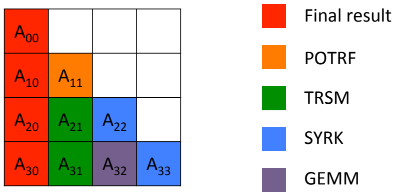

Figure 1. Step $k = 1$ of a Cholesky factorization of 4x4 tile matrix.

The tiles of the matrix are distributed between the nodes following the traditional 2D block cyclic distribution that provides good scalability properties and satisfactory load balancing with the "owner computes" strategy. Figure 2 illustrates an example of distributing a 4×4 tile symmetric matrix on a 2×2 grid of processes.

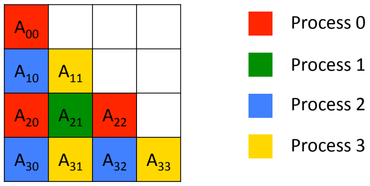

Figure 2. Example of a tile 2D block cyclic distribution.

Implemented in PaRSEC, the Cholesky factorization can be modeled as a DAG of tasks. Figure 3 shows the DAG for a 4×4 tile matrix on a 2×2 process grid. Each of the four basic linear algebra operations have corresponding types of tasks. Depending on the location of the input data, data dependencies among tasks can be local or remote. Note that the output of a task might overwrite its input. For example, in Figure 3, a **TRSM** task overwrites the input from its predecessor task **GEMM**, and a **POTRF** task overwrites the input from its predecessor task **SYRK**.

B. Problem Statement

SDC strikes at random moment and memory location, normally in the form of a bit flip. Figure 3 underline how data corruption is propagated when a *SDC* occurs. If this failure happens during the execution of a **TRSM** task on Node 1 (marked using a red cycle in the figure), by the end of the application the corrupted result would have propagated to the following 6 tasks (marked using blue cycles in the figure), ruining in total 35% of the computations in this example.

As stated earlier our goal was to design a solution at the runtime level that has low overhead and is applicable to a wide range of algorithms. One possible technique to offer low overhead is to prevent the corrupted data from propagating to the task's successors, therefore the failed task should be recovered right after the *SDC* happened before it got a chance to spread the corrupted result. Due to the use of the DAG, the recovery of a task can happen concurrently with the execution of unaffected tasks from the original execution, providing a maximal overlap between the correct original execution and the ongoing recovery. This also minimize the extra synchronizations in distributed-memory platforms, keeping the failure free execution almost unaltered from the original, non resilient, execution.

Algorithmic methods to mitigate the impact of *SDC*s on specific applications are known. For example, in dense linear algebra, Sherman-Morrison formula is applied to recover from one soft error during an LU factorization, only adding very small overhead [2]. However, these application-specific approaches are not considered in our design because they require the modification of each algorithm. We were tackling the challenge of designing an independent generic strategy that can be easily applicable to any task-based algorithm.

III. Design of Fault Tolerance in PaRSEC

In this section, we describe and quantify in terms of storage and memory overhead three mechanisms augmenting PaRSEC with soft error resilience. First, we introduce two coarse level mechanisms: a sub-DAG strategy and a sub-DAG & periodic checkpoint composite strategy; then, we introduce the Algorithm-Based Fault Tolerance (ABFT) strategy, a mechanism that operates at the task level. We use N to refer the matrix size and nb to refer the tile size.

We assume that failures can be either detected by the hardware or by internal checks of the task. Section III-C will discuss and evaluate the impact of algorithmic techniques to provide such detection.

A. Application Level Mechanism I: Correcting Sub-DAG Strategy

In PaRSEC, a DAG is represented by a concise format called Parameterized Task Graph (PTG), which expresses the tasks and their data dependencies in an symbolic way, independent of the problem size [12]. The runtime takes advantage of this representation to discover and schedule tasks without unfolding the entire DAG in memory, reducing the memory requirement for storing the DAG and exchanging the computation cycles to traverse the DAG with cycles to compute the successors of a task.

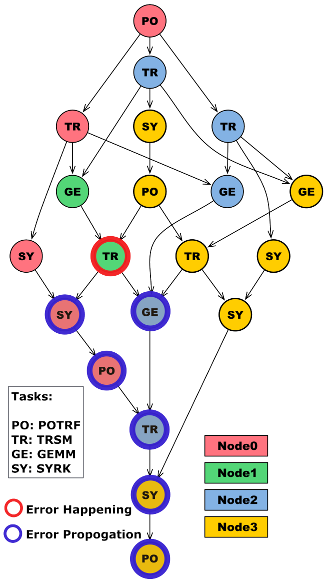

Figure 3. DAG of the Cholesky factorization of a 4x4 tile matrix on a 2x2 process grid, and a possible scenario of a soft error propagation (starting from the task surrounded by a red line).

When a soft error happens, the output, and potentially some of the input, of the failed task is corrupted. The re-execution of the failed task requires its predecessors to provide the input again. As the data flowing between tasks is not saved, the predecessors need to be re-executed as well. This backward traverse will go along the opposite direction of the original data flow until it reaches the source task of each of the necessary data.

Considering that the input comes from a read-only stable storage and is not affected by soft errors, as long as the runtime does not lose any information of the DAG, the correct result of any task in the DAG can be recomputed. Thus, a straightforward idea to recover from soft errors is to reuse the original data and DAG to recompute the missing data. Based on this idea, we exploit the capability of PaRSEC's PTG representation to dynamically retrieve all the predecessors of a failed task. In this mechanism, a failed task is replaced by re-executing a *correcting* sub-DAG consisting of this task and all its predecessors. Such sub-DAG is generated and scheduled by the runtime as soon as a failure is detected.

Figure 4 demonstrates an example of the correcting sub-DAG for the failed **TRSM** task in Figure 3. Compared with the original DAG, this sub-DAG has been trimmed, only keeping tasks related to re-executing the failed task. Re-executing the minimum required sub-DAG ensures that the failed task and its predecessors are recomputed only once from the original input data. Note that the recovery here is not sequential, it will happen concurrently with the execution of remaining tasks from the original DAG.

From an implementation perspective, when a PaRSEC application is executed on a distributed-memory platform, all computing nodes run the scheduling engine and each process can parse the concise PTG representation to find information about any tasks, including remote ones. When a failure is discovered, the node owning the failed task globally triggers the creation of the correcting sub-DAG by broadcasting an recovery event to the other nodes.

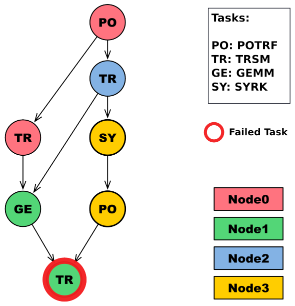

Figure 4. Correcting sub-DAG for the failed **TRSM** task.

Although this mechanism operates at the application level, it can be integrated into any task-based application. In the following, we analyze the overheads of this mechanism:

The **Computing Overhead** is proportional to the position of the failed task in the DAG. If the failure happens in the early stage of the execution, that means the size of the correcting sub-DAG is small and the computing overhead will be relatively low. On the other hand, if the failure happens in the late stage of the execution, the size of the sub-DAG will be large and the computing overhead will be relatively high. We analyze the computing overhead by considering the algorithm.

Figure 5 shows an example when failures happen in the middle of the Cholesky factorization. Failure can happen in four types of tasks, and the recovery cost of the **POTRF** task is minimum, as it is the predecessor of all the other three types of tasks. We compute the overhead as the number of additional floating point operations (FLOPs) to re-execute. The recovery of a **POTRF** task takes the same amount of FLOPs as a Cholesky factorization on the top left submatrix that encompasses the failed **POTRF** task. On our example, this is a half-size submatrix $A[[0, N/2], [0, N/2]]$, as marked by dark blue line in Figure 5. We use the cost of recovering failed **POTRF** in Kth column as the theoretical computing overhead. It is computed as:

$$FLOP_{Orig} = \frac{1}{3}N^3 \qquad FLOP_{Extra} = \frac{1}{3}K^3$$
$$Overhead_{Comp} = \frac{FLOP_{Extra}}{FLOP_{Orig}} = (\frac{K}{N})^3$$

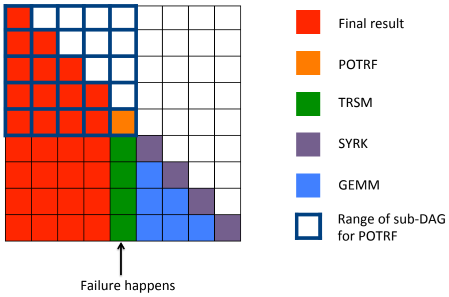

Figure 5. Example when a failure happens in the middle of a factorization.

Table I summarizes the computing overhead for Cholesky when failures happen at different stages of execution. In the failure-free case, the correcting sub-DAG is never triggered and there is no performance penalty.

Table I
COMPUTING OVERHEAD OF SUB-DAG MECHANISM FOR CHOLESKY FACTORIZATION.

Failure Position	Beginning	Middle	End	No Failure
$Overhead_{Comp}$	$(\frac{nb}{N})^3$	12.5%	100%	0

Storage Overhead: This sub-DAG mechanism requires extra memory as backup for the input. Thus in the worst case, when data is not available on a stable storage initially, another $N \times N$ symmetric, positive-definite matrix is allocated and the storage overhead is 100%.

B. Application Level Mechanism II: Sub-DAG & Periodic Checkpoint Composite Strategy

As analyzed above for the correcting sub-DAG mechanism, the re-execution always starts from the source task of the DAG because the intermediary data is not saved during execution. The computing overhead explodes when a failure happens in the late stages of the execution, up to 100% to recover the final task of the factorization, meaning that the whole application needs to be recomputed.

In this strategy, we augment the previous approach, by adding a diskless periodic checkpoint to limit the necessary rollback and therefore reduce the number of recomputed tasks. To recover a failed task, only the predecessors after the last saved intermediary data are required to be re-executed.

Every tile is treated as a checkpointing unit. We define a checkpoint interval β, such that a process will save a copy of each data every β updates. Checkpoint interval β can be modeled as a function of failure rate, task execution time and checkpoint time [14]. The optimal value of β is not discussed in this paper and we set it to a constant. Every node saves its own data locally providing a fully-fledged snapshot of the application. Checkpoint is saved to local memory and we assume that the probability that both the data and its checkpoint are corrupted by correlated failures is negligible. Figure 6 shows an example when $\beta = 2$.

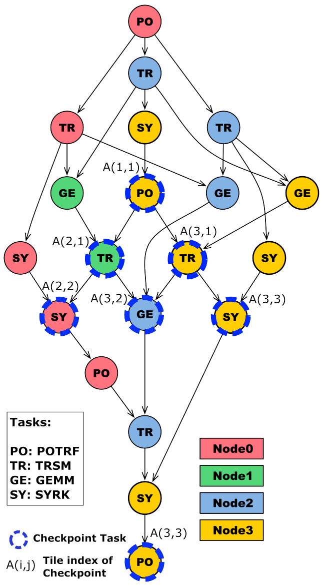

Figure 6. An example of combining correcting sub-DAG with periodic checkpoint when checkpoint interval $\beta = 2$.

Computing Overhead: In this mechanism, the bound on the computing overhead does not depend on a failure position. Indeed, in the Cholesky factorization, the input of any task is either the owner tile for which there was a checkpoint at most β operations ago, or a final output of another task, that was validated before the re-executed task could start. Thus, any failed task output can be recovered by re-executing at most β previous tasks on the same tile. The number of FLOPs of a task is $C \cdot nb^3$, where C is $1/3$ for **POTRF**, 1 for **TRSM**, 1 for **SYRK** and 2 for **GEMM**. We set C to 2 to provide a conservative bound when estimating computing overhead for any task. The theoretical computing overhead can be computed as:

$$\begin{aligned} FLOP_{Extra} &= \beta 2nb^3 \\ Overhead_{Comp} &= \frac{FLOP_{Extra}}{FLOP_{Orig}} = \frac{\beta 6nb^3}{N^3} \end{aligned}$$

Table II summarizes the computing overhead of the Cholesky factorization using the Sub-DAG & Periodic Checkpoint Composite strategy. The overhead in failure free execution is close to 0 because the cost of local memory checkpoint is negligible.

Storage Overhead: This mechanism needs to allocate the same size of matrix as input to store data flowing snapshot periodically, this even if the initial data is available on a stable storage. Thus, the storage overhead is 100%.

Table II
COMPUTING OVERHEAD OF SUB-DAG & PERIODIC CHECKPOINT MECHANISM FOR CHOLESKY FACTORIZATION.

Failure Position	Beginning	Middle	End	No Failure
$Overhead_{Comp}$	$\frac{nb^3}{N^3}$	$\frac{\beta 6nb^3}{N^3}$	$\frac{\beta 6nb^3}{N^3}$	≈ 0

C. Task Level Mechanism: Algorithm-Based Fault Tolerance

The two application level mechanisms described above take advantage of the task graph of the application to recover from failures by re-executing the minimum necessary tasks. A different mechanism, potentially less generic, is to use a known task invariant to completely avoid re-execution. This approach is based on Algorithm Based Fault Tolerance (ABFT) techniques, with well known solutions for most of the dense and sparse linear algebra kernels. In order to recover from data corruption inside a task, additional information would be attached to the data to allow error correction if necessary.

ABFT was firstly introduced by Huang and Abraham to detect and correct soft errors in systolic arrays [15]. ABFT methods are based on the idea of maintaining consistency of the computing data and recovery data, by applying appropriate mathematical operations on both original data and recovery data [16]. Typically, for linear algebra operations, additional rows and/or columns are attached to input matrix to maintain checksums.

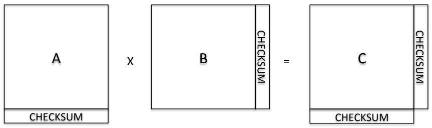

Figure 7. An ABFT matrix multiplication example.

An example of ABFT enabled matrix multiplication is shown in Figure 7. The corresponding matrices A, B, C have the following relationship:

$$A * B = C$$

A is appended with a column checksum vector $e^T A$ while B is appended with a row checksum vector Be. It is shown as follows that the checksum relationship for matrix C is maintained after computation:

$$\begin{bmatrix} A \\ e^T A \end{bmatrix} \begin{bmatrix} B & Be \end{bmatrix} = \begin{bmatrix} AB & ABe \\ e^T AB & e^T ABe \end{bmatrix} = \begin{bmatrix} C & Ce \\ e^T C & e^T Ce \end{bmatrix}$$

In our task level mechanism, we attach column checksum vectors to every tile in the input matrix. Note that we do not modify the factorization itself: the checksum applies only to each tile. Figure 8 shows the matrix snapshot after attaching column checksum vectors to a 4x4 tile matrix. As a sidenote it is interesting to mention that for some tasks this approach also provides an efficient *SDC* detector. In order to detect errors in one task, one checksum vector is sufficient. Furthermore, detecting and correcting n errors require at least $n{+}1$ checksum vectors [17]. For example, let's consider an $n \times n$ matrix $A = [a_1, a_2, ..., a_n]$ and two n vectors

$$e_1 = (1, 1, ..., 1)^T \qquad e_2 = (1, 2, ..., n)^T$$

Two column checksum vectors are defined as:

$$c_1 = e_1 A \qquad c_2 = e_2 A$$

Assume that an error happens at the (i, j) element of A:

$$a'_{i,j} = a_{i,j} + \gamma$$

Now we can first decide the jth column of A is inconsistent with checksums:

$$\alpha_1 = \sum_{k=1}^{n} a_{k,j} - (c_1)_j = \gamma \neq 0$$

This provides a detector for the occurrence of a silent data corruption. Such approach, based on a single checksum, can be used to implement a detector for the two application level mechanisms described above. Then we can determine the ith element of this column causes the inconsistence:

$$\alpha_2 = \sum_{k=1}^{n} k a_{k,j} - (c_2)_j = i\gamma$$

$$\alpha_2 / \alpha_1 = i$$

The value of $a_{i,j}$ is corrected by simply subtracting α_1.

For the remainder of this paper, we consider the case of a single soft error per execution. We simulate it by introducing a significant bit-flip into the exponent of a floating point data and let the runtime detect and recover the state of the computation. It is to be noted that for some tasks, a single soft-error when propagated through the algorithm translates into more than one corrupted result. While the ABFT protection scheme presented in Section III-C is unable to correct the data in this particular case, it can be used as a detection mechanism. In this case, the two mechanisms from Section III-A and III-B can successfully complement the proposed approach. In the following analysis of overhead, we focus on the case where failures can be recovered using an ABFT mechanism.

Computing Overhead: Using ABFT techniques, extra FLOPs are introduced *for each task* for maintaining the checksums. After attaching checksum vectors, the matrix size becomes $(1 + 2/nb)N \times (1 + 2/nb)N$. The number of FLOPs of the Cholesky factorization on this new matrix is:

$$FLOP_{New} = \frac{1}{3}((1 + \frac{2}{nb})N)^3$$

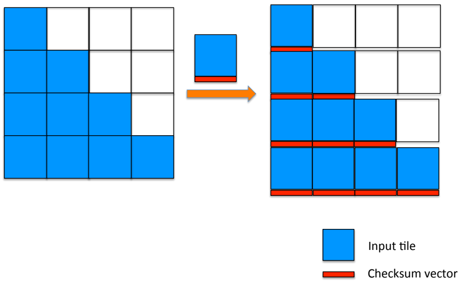

Figure 8. Attaching checksum vectors to a 4x4 tile matrix.

The cost of maintaining checksum is:

$$FLOP_{Chk} = \frac{1}{3}((1 + \frac{2}{nb})N)^3 - \frac{1}{3}N^3$$

The computing overhead of maintaining checksums is:

$$Overhead_{Chk} = \frac{FLOP_{Chk}}{FLOP_{Orig}} = (1 + \frac{2}{nb})^3 - 1$$

The extra FLOPs of correcting error in one task come mostly from using matrix vector multiplication to add elements in the same column together to detect failures, which is $2nb^2$. We plug this matrix vector multiplication into routines of **POTRF**, **TRSM**, **SYRK** and **GEMM** to detect failures from any possible task during execution. If one failure is detected, only nb floating point operations are required to locate the error position and only one FLOPs is required to add the error back to the corrupted matrix element. These $nb + 1$ operations are negligible comparing with the large amount of operations in maintaining checksums and detecting errors and are discarded in overhead estimation. There are approximately $(N/nb)^3/6$ tasks in Cholesky factorization, thus we estimate the total cost of correcting error as:

$$FLOP_{Corr} = \frac{N^3}{3nb}$$

The computing overhead of correcting error is:

$$Overhead_{Corr} = \frac{FLOP_{Corr}}{FLOP_{Orig}} = \frac{1}{nb}$$

The total computing overhead of this mechanism is:

$$\begin{aligned} Overhead_{Comp} &= Overhead_{Chk} + Overhead_{Corr} \\ &= (1 + \frac{2}{nb})^3 - 1 + \frac{1}{nb} \end{aligned}$$

The task level mechanism recovers the data without re-executing any task in the DAG, thus the recovery overhead does not depend on the failure position in the DAG. As shown in Table III, the performance penalty remains the same for failure-free execution as checksum vectors have been encoded into the original matrix.

Table III
COMPUTING OVERHEAD OF ABFT MECHANISM FOR CHOLESKY FACTORIZATION.

	One Failure	No Failure
$Overhead_{Comp}$	$(1+\frac{2}{nb})^3 - 1 + \frac{1}{nb}$	$(1+\frac{2}{nb})^3 - 1 + \frac{1}{nb}$

Storage Overhead: The ABFT mechanism requires allocating extra memory to store checksum vectors. For every $nb \times nb$ tile, the size of checksum vectors is $nb \times 2$, thus the total storage overhead is $2/nb$. In tiled dense linear applications, the tile size is tuned to optimize the efficiency of the operation and the parallelism of the application. This often translates in nb in hundreds, which make the extra memory requirement of storing checksum vectors negligible.

IV. EXPERIMENTAL RESULTS

A. *Experiment Setup*

We use the Titan supercomputer at Oak Ridge National Laboratory as our testing platform. Titan has $18,688$ nodes with Cray custom high-speed interconnect, each node contains a 16-core AMD Opteron 6274 CPU with 32 GiB of DDR3 ECC memory and an Nvidia Tesla K20X GPU with 6 GiB GDDR5 ECC memory. We only use the CPU section of Titan and 8 CPU cores per node since each node possesses 8 floating point units.

At the software level, we use GCC 4.8.2 as compiler and Cray LibSci 12.2.0 to provide basic linear algebra subroutines (BLAS). The Cholesky factorization is implemented in double precision with tile size $nb = 200$ that was tuned to reach the highest performance of the **GEMM** operation, while still allowing a large amount of parallelism at reasonable matrix sizes. To serve as a comparison base, we use the standard Cholesky factorization implemented in PaRSEC without any soft error resilient mechanisms.

We pursue weak scalability experiments to evaluate the capability of the proposed fault tolerant strategies to handle potentially larger problems when more computing resources are available. For these experiments, we fix the memory used on each node and increase the matrix size accordingly when we increase the number of nodes. We set the matrix input size for single-node experiments to 6000, and scale it with $6000\sqrt{P}$ where P is the number of nodes.

Failures are injected as single bit-flip inside one task during the execution. The failure triggers when the Cholesky factorization reaches the middle column of the matrix (as indicated in the Figure 5). In all experiments reported in this section, we take 5 runs and report the average (arithmetic mean) number. Standard deviations are shown using error bars.

B. *Performance of Correcting Sub-DAG Mechanism*

Figure 9 shows the performance and overhead of the Cholesky factorization with the correcting sub-DAG mechanism on Titan when one failure happens in the execution. We inject the failure in four types of task separately. For now, we assume that the failure was detected with an abstract mechanism of zero cost. The red curve is the performance of non fault tolerant Cholesky using standard PaRSEC. It has the same performance as fault tolerant version in failure-free execution since there is no sub-DAG created as analyzed in Section III-A. The theoretical overhead is 12.5%, as computed from Table I (as explained, this is the cost of computing a Cholesky factorization on a matrix of half size). We can see that the overheads of all four one-failure cases are around 15% and close to theoretical overhead.

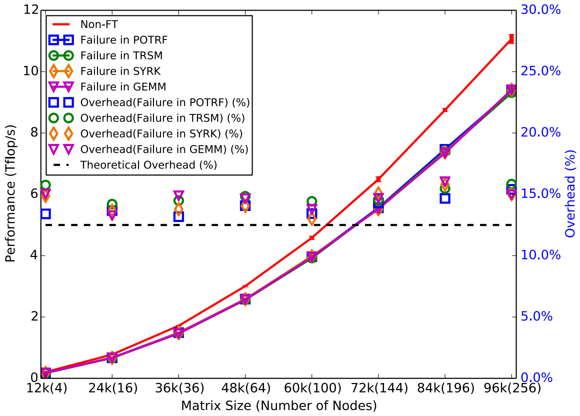

Figure 9. Weak scalability of correcting sub-DAG mechanism compared to non fault tolerant Cholesky.

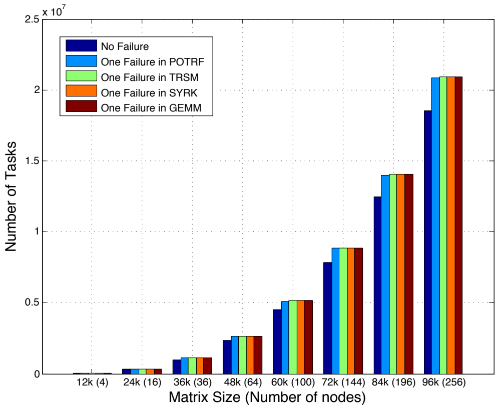

Figure 10. Number of total tasks of the correcting sub-DAG mechanism in failure-free and one-failure cases.

Figure 10 shows the number of total tasks of the correcting sub-DAG mechanism in different cases. It indicates that the number of re-executed tasks for four types of tasks is very close. Recovering from failures in **POTRF** requires the fewest tasks since **POTRF** is on the critical path of DAG and is the predecessors of the other three types of tasks.

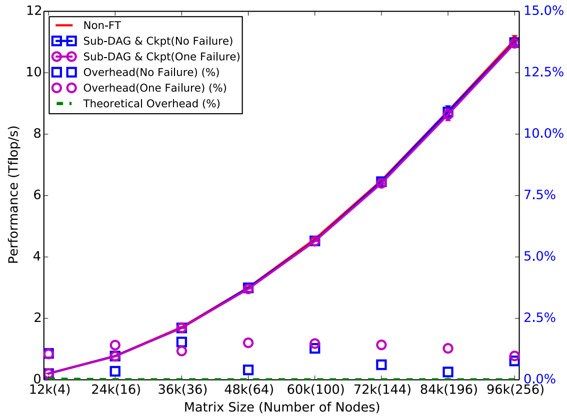

Figure 11. Weak scalability of correcting sub-DAG & periodic checkpoint composite mechanism compared to non fault tolerant Cholesky.

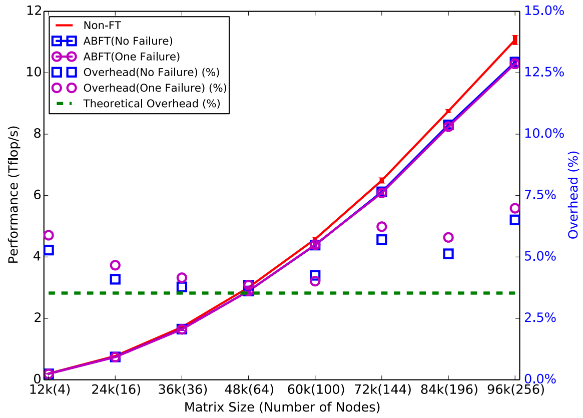

Figure 12. Weak scalability of ABFT mechanism compared to non fault tolerant Cholesky.

C. Performance of Correcting Sub-DAG & Periodic Checkpoint Composite Mechanism

Figure 11 shows the performance and overhead of the Cholesky factorization with the correcting sub-DAG & periodic checkpoint composite mechanism on Titan when one failure happens during the execution. As before, we do not consider yet the mechanism used to detect the failure in this experiment. The checkpoint interval β is set to 10, i.e. the state of a tile is saved to memory locally after 10 tasks update the tile. Since periodic checkpoint protects the data on every tile and the upper bound of recovering from corruption is re-executing 10 tasks for any type of tasks, we inject the failure in one **GEMM** task. Recovering from corruption in the other three tasks have similar overheads. Based on the discussion in Section III-B, the theoretical overhead of one-failure case is close to 0. We can see that the overhead of the failure-free case has a variability of about 1%, and the one-failure case fluctuates around 2%. Both numbers are in the noise of the measurement. These results validate our analysis that diskless checkpoint reduces the number of re-executing tasks drastically and the time spent to save critical snapshots of data flowing remains negligible.

D. Performance of ABFT Mechanism

Figure 12 presents the performance and overhead of Cholesky factorization with task level fault tolerant support using ABFT technique on Titan. ABFT implements at the same time a detection mechanism and the correction method. Each task of the entire DAG is thus validated at completion, and corrective actions are initiated only of this validation fails. For the one-failure case, we inject the failure in one **GEMM** task, and the erroneous matrix element will not be propagated inside the task thus it can be recovered using checksums. The theoretical overhead is obtained from Table III. The results show that the overhead of failure-free case and one-failure case fluctuates around 5%, and does not increase when application size and number of nodes increase, and remains close to the theoretical overhead. Also, it is important to note that the difference between failure-free overhead and one-failure overhead is negligible. Compared with failure-free case, only nb more FLOPs are required to locate the error position and only one FLOP is required to add the error back to the corrupted matrix element. These extra $nb + 1$ operations are negligible considering the total number of FLOPs is $(1/3)N^3$ in the Cholesky factorization.

Compared with the previous two application level mechanisms, this task level mechanism has higher overheads in fault-free case because of the cost of maintaining checksums and failure detection. At the contrary, the additional cost to recover from failures is very small in task level mechanism since it does not require task re-execution.

E. Overhead of Detection Mechanism

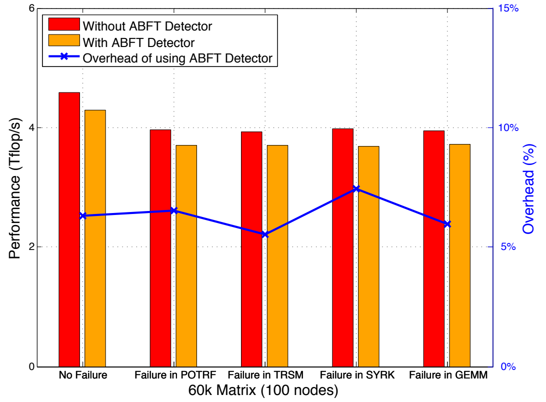

Figure 13. Performance and overhead of using ABFT as a Detection Mechanism for the correcting sub-DAG approach without failures and with one failure.

For the first two application level mechanisms we assumed that a *SDC* detector was available for a 0 cost. In normal conditions this assumption is overly optimistic. Thus, for algorithms exhibiting ABFT properties, the ABFT techniques can be used to provide accurate and effective *SDC* detectors. Figure 13 presents the performance of a Cholesky Factorization using the correcting sub-DAG technique on 60k matrix using 100 nodes, and highlights the cost and overhead of using ABFT on a single checksum for each tile to implement the *SDC* detection mechanism. Detection cost comes from FLOPs in maintaining the checksum and validating results. This cost is paid on each task, regardless if it is a task of the original DAG, or a task of the correcting sub-DAG. The results validate that if ABFT detector is enabled, the overhead cost can increase up to 6%.

V. Related Work

Soft errors become more prevalent with new technologies that feature higher clock frequencies, increased transistor density and lower voltage [5]. While large scale machines are often equipped with hardware mechanisms for resilience, they do not provide an integrated approach in system level to ensure fault tolerance for applications. Pioneering works addressing soft errors in large scale applications can be classified into the following four categories.

Checkpoint/Restart: The major advantage of the Checkpoint/Restart technique is its generality, as it can be applied to all applications. As a representative system, Charm++ employs an automatic Checkpoint/Restart framework that performs application replication to provide fault tolerance [18]. The checkpoint period is decided by online information about current failure rate. Our approach avoids using coordinated checkpoint thus provides much less overhead.

MPI Level Fault Tolerance: In MPI applications, soft errors can be detected by comparing MPI messages between process replicas. The RedMPI library creates "replica" MPI tasks for each "primary" task and performs online MPI message verification intrinsic to existing MPI communication [7]. Our approach does not require process replication.

Algorithm Based Fault Tolerance: Algorithm Based Fault Tolerance does not require disk accesses thus is low-overhead in nature. This technique has been widely adapted in dense linear algebra, for example, Sherman-Morrison formula is applied after LU factorization to recover soft errors for dense linear system solver [2]. FT-ScaLAPACK library modifies LU, QR and Cholesky factorizations in ScaLAPACK with blocked algorithms to detect and correct soft errors during computation [17]. Our method is more generic. At the application level, the user is not required to change the application to support fault tolerance, while at the task level, as a decomposed unit from the application, the additional effort and computational and memory overheads become significantly more reasonable.

Fault-tolerant Task-based System: Many task-based runtimes have integrated fault tolerant capabilities to support their applications. Task scheduling can be static or dynamic, depending on whether an application's task graph is known before computation starts [19]. In static scheduling, tasks are allocated to processors or computing nodes ahead of time. In such systems, task duplication has been introduced to protect applications from failures in grids [20] and in real-time systems [21]. The requirement of doubling the number of protected tasks impacts the application performance in absence of failure. In dynamic scheduling, on the other side, tasks are allocated to processors during the computation. This requires the runtime to schedule the recovery efficiently to reach optimal performance when a failure happens.

Tremendous progress has been made to add resilience to dynamic task-based systems. Mouallem et al. proposed three fault tolerant mechanisms in Kepler scientific workflow system, including forward recovery using retries and alternative versions, checkpointing and adding a Error Handling Layer [22]. The goal of fault tolerance framework in workflow system is to provide an appropriate end-to-end support for detecting and recovering from failures during execution, while the goal of our design is targeted to a low-overhead solution. A fault-tolerant dynamic task graph scheduling algorithm is implemented in the NABBIT system, which recovers from soft errors via task re-execution and work-stealing [23]. Scalable and low-overhead Checkpoint/Restart and task replication schemes are integrated in the Nanos asynchronous data-flow runtime which supports OpenMP programming model [24]. However, these two runtime systems only provide reliability for applications on shared-memory platforms, while our design supports shared-memory and distributed-memory platforms seamlessly. An improved coordinated checkpoint protocol is implemented in KAAPI framework to reduce the number of processes that are required to redo computation by utilizing the communication dependencies [25]. While this scheme requires global synchronization for all processes in checkpointing, our design avoids global synchronization.

VI. Conclusion and Future Work

This paper describes three possible mechanisms designed for a dynamic task-based runtime, for handling soft-errors. The proposed extensions ensure resilience at two different levels of granularity: coarse granularity, automatic solutions at the application level and fine granularity, algorithm-based solutions at the task level. At the application level, a correcting sub-DAG mechanism is used to recover from failures by re-executing only required tasks to retrieve the lost data. A composite mechanism combining sub-DAG with periodic diskless checkpoint saves critical snapshots of the data flowing between tasks during the execution to reduce the amount of necessary re-executions, addressing one of the major issues with traditional application-level checkpoint/restart strategies. These two application-level mechanisms are generic and can support any task-based application, providing an automatic application-independent failure management mechanism. As task-based approaches decompose the application into smaller and less complex tasks, it become possible to take advantage of the intrinsic algorithm properties of these tasks to provide validators allowing to detect, and possibly recover, from soft-errors. Additionally, we presented a *SDC* detector applicable to any algorithms exhibiting similar ABFT properties (as described by Huang and Abraham [15]), that can successfully replace a hardware-level fault detector while providing means to quickly and accurately recover the data. Our experimental evaluation on a large scale platform, Titan, corroborate the feasibility of the proposed approaches and highlight the low overhead of the current implementation in PaRSEC. In the future work we will explore methods to provide transparent, or algorithm-based solutions to the *fail-stop* model in task-based runtimes.

Acknowledgments

This material is based upon work supported by the Department of Energy under Award Number DE-SC0010682 and by the National Science Foundation under Grant Number CCF-1244905. It is also funded in part by the Russian Scientific Fund, Agreement N14-11-00190. The authors thank Wei Wu from University of Tennessee, who shared his experience of developing applications using PaRSEC.

References

[1] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir, "Toward exascale resilience," *Int. J. High Perform. Comput. Appl.*, vol. 23, no. 4, pp. 374–388, Nov. 2009.

[2] P. Du, P. Luszczek, and J. Dongarra, "High performance dense linear system solver with soft error resilience," in *IEEE Intl. Conf. on Cluster Computing*, 2011, pp. 272–280.

[3] A. Geist, "What is the monster in the closet?" Invited Talk at Workshop on Architectures I: Exascale and Beyond: Gaps in Research, Gaps in our Thinking, aug 2011.

[4] A. Moody and G. Bronevetsky, "Scalable I/O systems via node-local storage: Approaching 1 TB/sec file I/O," LLNL Technical Report LLNL-TR-415791, 2009.

[5] R. Baumann, "Soft errors in advanced computer systems," *Design Test of Computers, IEEE*, vol. 22, no. 3, pp. 258–266, May 2005.

[6] K. Ferreira, J. Stearley, J. H. Laros, III, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold, "Evaluating the viability of process replication reliability for exascale systems," in *SC'11*. ACM, 2011, pp. 44:1–44:12.

[7] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell, "Detection and correction of silent data corruption for large-scale high-performance computing," in *SC'12*. IEEE Computer Society Press, 2012, pp. 78:1–78:12.

[8] J. S. Plank, K. Li, and M. A. Puening, "Diskless checkpointing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 9, no. 10, pp. 972–986, Oct. 1998.

[9] G. Aupy, A. Benoit, T. Herault, Y. Robert, F. Vivien, and D. Zaidouni, "On the combination of silent error detection and checkpointing," in *PRDC*, 2013, pp. 11–20.

[10] J. Lidman, D. J. Quinlan, C. Liao, and S. A. McKee, "ROSE: fttransform - A source-to-source translation framework for exascale fault-tolerance research," in *IEEE/IFIP DSN'12, 2012*, 2012, pp. 1–6.

[11] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. Dongarra, "PaRSEC: Exploiting heterogeneity to enhance scalability," *Computing in Science Engineering*, vol. 15, no. 6, pp. 36–45, Nov 2013.

[12] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, "DAGuE: A Generic Distributed DAG Engine for High Performance Computing," in *IPDPS Workshop*, May 2011, pp. 1151–1158.

[13] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures," *Parallel Comp.*, vol. 35, no. 1, pp. 38–53, 2009.

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

[15] K.-H. Huang and J. A. Abraham, "Algorithm-Based Fault Tolerance for matrix operations," *IEEE Trans. Comput.*, vol. 33, no. 6, pp. 518–528, Jun. 1984.

[16] P. Du, A. Bouteiller, G. Bosilca, T. Herault, and J. Dongarra, "Algorithm-Based Fault Tolerance for dense matrix factorizations," in *PPoPP'12*. ACM, 2012, pp. 225–234.

[17] P. Wu and Z. Chen, "FT-ScaLAPACK: Correcting soft errors on-line for ScaLAPACK Cholesky, QR, and LU factorization routines," in *HPDC'14*. ACM, 2014, pp. 49–60.

[18] X. Ni, E. Meneses, N. Jain, and L. V. Kalé, "ACR: Automatic checkpoint/restart for soft and hard error protection," in *SC'13*. ACM, 2013, pp. 7:1–7:12.

[19] T. Johnson, "A concurrent dynamic task graph," in *Parallel Computing, Vol 22, No 2, Ferbruary*, 1993, pp. 327–333.

[20] B. Fechner, U. Honig, J. Keller, and W. Schiffmann, "Fault-tolerant static scheduling for grids," in *IPDPS*, 2008, pp. 1–6.

[21] X. Qin and H. Jiang, "A novel fault-tolerant scheduling algorithm for precedence constrained tasks in real-time heterogeneous systems." *Parallel Computing*, vol. 32, no. 5-6, pp. 331–356, 2006.

[22] P. Mouallem, D. Crawl, I. Altintas, M. Vouk, and U. Yildiz, "A fault-tolerance architecture for kepler-based distributed scientific workflows," in *SSDBM'10*, 2010, pp. 452–460.

[23] M. C. Kurt, S. Krishnamoorthy, K. Agrawal, and G. Agrawal, "Fault-tolerant dynamic task graph scheduling," in *SC'14*, November 2014.

[24] O. Subasi, J. Arias, J. Labarta, O. Unsal, and A. Cristal, "Leveraging a task-based asynchronous dataflow substrate for efficient and scalable resiliency," in *DMTM*, 2014.

[25] X. Besseron, S. Jafar, T. Gautier, and J.-L. Roch, "CCK: An Improved Coordinated Checkpoint/Rollback Protocol for Dataflow Applications in KAAPI," in *ICTTA'06*, Apr. 2006.