31 May 2016

Towards Practical Algorithm based Fault Tolerance in Dense Linear Algebra

Panruo Wu

Qiang Guan

Nathan DeBardeleben

Sean Blanchard

et. al. For a complete list of authors, see https://scholarsmine.mst.edu/comsci_facwork/1104

Recommended Citation

P. Wu et al., "Towards Practical Algorithm based Fault Tolerance in Dense Linear Algebra," *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing (2016, Kyoto, Japan)*, pp. 31-42, Association for Computing Machinery (ACM), May 2016.
The definitive version is available at https://doi.org/10.1145/2907294.2907315

Towards Practical Algorithm Based Fault Tolerance in Dense Linear Algebra

Panruo Wu
Department of Computer Science and Engineering
University of California, Riverside
pwu011@cs.ucr.edu

Qiang Guan, Nathan DeBardeleben and Sean Blanchard
Ultrascale Systems Research Center
Los Alamos National Laboratory*
{qguan,ndebard,seanb}@lanl.gov

Dingwen Tao, Xin Liang, Jieyang Chen and Zizhong Chen
Department of Computer Science and Engineering
University of California, Riverside
{dtao001,xlian007,jchen098,chen}@cs.ucr.edu

ABSTRACT

Algorithm based fault tolerance (ABFT) attracts renewed interest for its extremely low overhead and good scalability. However the fault model used to design ABFT has been either abstract, simplistic, or both, leaving a gap between what occurs at the architecture level and what the algorithm expects. As the fault model is the deciding factor in choosing an effective checksum scheme, the resulting ABFT techniques have seen limited impact in practice. In this paper we seek to close the gap by directly using a comprehensive architectural fault model and devise a comprehensive ABFT scheme that can tolerate multiple architectural faults of various kinds. We implement the new ABFT scheme into high performance linpack (HPL) to demonstrate the feasibility in large scale high performance benchmark. We conduct architectural fault injection experiments and large scale experiments to empirically validate its fault tolerance and demonstrate the overhead of error handling, respectively.

1. INTRODUCTION

The extreme scale high performance computing (HPC) systems that are expected by the end of this decade poses several challenges including performance, power efficiency, and reliability. Due to the large amount of components in these systems and the shrinking feature size, the probability that an extreme scale application experiences faults during its execution is projected to be non negligible. Resilience to faults have been widely accepted as critical for exascale HPC applications[21, 6, 3].

*This work was performed at the Ultrascale Systems Research Center (USRC) at Los Alamos National Laboratory. The publication has been assigned the LANL identifier LA-UR-16-20226.

HPDC'16, May 31-June 04, 2016, Kyoto, Japan

DOI: http://dx.doi.org/10.1145/2907294.2907315

Faults are malfunctions of the hardware or software, and are the underlying causes for observable errors. When the fault does not interrupt the execution of a process the program can continue execution normally, but the results may be corrupted. Such silent data corruptions cannot be tolerated by checkpoint/restart (C/R) alone unless they can be frequently detected. Silent data corruptions may be the consequence of soft faults caused by cosmic rays and radiation from packaging materials, and are usually one time events that corrupt the state of the machine but not its overall functionality. We restrict our scope to silent data corruptions (SDC) in this work. Note that since soft errors which are caused by single event upset frequently corrupt data silently, SDC handling is also often discussed in context of soft errors.

Faults in storage and communication systems are often effectively tolerated by error correction codes (ECC) because the data stored or communicated are not changing. However, faults in logic units that transform the data are harder to detect and tolerate. Typically some kind of double modular redundancy (DMR) is needed to detect soft faults in logic units and triple modular redundancy (TMR) is needed to tolerate SDCs. Although modular redundancy requires at least 100% resource overhead and often incurs significant execution time overhead, it is sometimes the only general system level solution to tolerate SDCs [9, 22].

System level SDC solutions can be prohibitively expensive for HPC systems. An alternative solution is to implement fault tolerance in applications, which can take advantage of the semantics and structure of a specific application resulting in much lower cost. Algorithm based fault tolerance (ABFT) represents a middle ground between application specific fault tolerance and architecture fault tolerance. At one end application specific fault tolerance is highly diverse that often require ad-hoc solutions, at the other end system fault tolerance is general but too costly and unscalable. Algorithms thus presents just enough semantics to take advantage and structure to be generally useful.

ABFT has first been proposed in a seminal work by Huang and Abraham [13] for matrix-matrix multiplication on systolic arrays. The idea of ABFT can be seen as an adaption of ECC to numeric structures like matrices or vectors. The significant difference is that for ECC the data is static but for ABFT the data is under transformation. In ABFT

the central problem is that the codes must maintain after transformation in order to be able to detect errors using the codes. The fault model is a deciding factor in the design of ABFT codes and adaption to the associated algorithm. However the fault models used in existing ABFT research are either too abstract [8, 14, 10] or too simplistic [5, 24, 25] limiting their use where the architectural fault models do not fit. In this work we rethink the fault model and explore the challenges if we use a comprehensive architectural fault model that allows both logic/arithmetic faults and storage faults in main memory, on-chip memory, and other datapaths. We demonstrate that with this fault model we still can design highly efficient and resilient ABFT techniques for dense linear algebra and use high performance linpack (HPL) to show that the new techniques can be implemented efficiently in complex real world high performance and highly scalable applications. The design is validated empirically by a QEMU [2] based architectural fault injector, F-SEFI [1], which implements the comprehensive fault model. We incorporate the new ABFT techniques into the latest Netlib HPL-2.1 and empirically show that the resulting FT-HPL incurs low overhead and maintains high scalability of the original HPL.

The contributions of this paper are:

New fault model We use a fault model that allows logic faults and memory system faults that are comprehensive temporally and spatially and design ABFT schemes that can effectively detect and correct errors caused by these faults.

New checksum scheme We propose a novel process local checksum scheme, multiple checksums for error detection and correction by studying the syndrome (error patterns) caused by the faults.

Validation and software implementation We test and validate the resilience using an architectural fault injector. We implement the new ABFT schemes in the latest Netlib HPL-2.1.

The rest of the paper is organized as follows. In section 2 we survey the techniques to handle SDCs in computing systems especially the algorithm based approach. In section 3 we propose the architectural fault model and the errors it causes in the eyes of application. In section 4, we present our new designs to handle the proposed fault model. In section 5, fault tolerance capability, various sources of overheads, and optimization methods are discussed. In section 6, we present empirical study of the fault tolerance of the proposed design and implementation through error injection, and the overheads in large scale runs. Section 7 concludes the paper.

2. RELATED WORK

The first report on soft errors due to alpha particles in computer chips was from Intel in 1978 [15]. The first report on soft errors due to cosmic radiations in computer chips was in 1984 [27]. In 1996, Norman [18] studied error logs of several large computer systems and reported a number of incidents of cosmic ray strikes. In 2005, Hewlett-Packard admitted that the ASC Q supercomputer located in Los Alamos National Laboratory experienced frequent crashes because of cosmic ray strikes to its parity protected cache tag arrays. The machine is particularly susceptible because of the 7000ft altitude of the installation location [16]. The book by Mukherjee [17] surveys extensively the architectural techniques to design architectures for soft errors.

In the HPC context much effort has been spent on techniques to detect and tolerate soft errors. System level approaches usually involves some kind of modular redundancy. RedMPI [9] is a general MPI level solution that replicates each MPI rank to form double modular redundancy (DMR) for soft error detection or triple modular redundancy (TMR) for error correction. The difficulty is the silent nature of soft errors; error detection must be active and in a timely manner. RedMPI does the error detection when MPI ranks communicate: the replicas should send out the same message otherwise a soft error is detected. According to the paper, MPI rank level replication incurs 20% to 60% execution time overhead in addition to 100% to 200% computing resource overheads. Another approach is algorithmic error detection coupled with checkpointing for recovery. In [4], the intrinsic orthogonality of some Krylov linear solvers is used for error detection. A study by [20] proposes to turn many interesting problems into optimization problems that can be solved iteratively which is naturally resilient to soft errors.

In the following texts the most relevant related works are discussed and special attention is paid to the fault models and the influence fault model on the design of algorithm based fault tolerant schemes. ABFT has been researched extensively for many algorithms but we will narrow our scope to those that are checksum based and applicable on dense matrix multiplication and triangularization.

Algorithm based fault tolerance was first proposed by Abraham and Huang [13]. The original ABFT was proposed for matrix multiplication and LU on systolic arrays for real time signal processing. The fault model used is logic faults that produces erroneous results. Storage cell faults such as in memory, latch, and registers are assumed to be handled by traditional error correction codes. In matrix multiplication, as a single arithmetic fault causes only a single error in the result matrix, this ABFT scheme can effectively detect and correct it. In LU decomposition, because of error propagation, a single fault will cause an overwhelmingly large amount of errors in the results, thus making this ABFT scheme unable to tolerate a single fault algorithmically. The limited correction capability is due to three factors: 1) inability to tolerate multiple errors in the checksum scheme, 2) massive error propagation in matrix triangularization, and 3) offline error correction. These three factors conspire to make algorithmic error correction difficult in matrix triangularization.

Later Luk and Park [14] described an elegant analytical model for ABFT in matrix triangularization. The analytical model assumes an abstract fault model that a transient error occurs at some intermediate iteration in the triangularization. Even though the single error will propagate in later stages and become uncorrectable at the end, it can be shown that the error can be cast back as a single rank perturbation to the original input matrix, much like the widely used backward error analysis [23]. Then assuming two row checksums the correct result can be derived based on the backward fault model. This is a powerful technique that avoids the error propagation problem but it has three limitations: 1) the fault model assumes single error not necessarily single fault, as we have seen that single fault may cause multiple

errors; 2) this checksum scheme has no column checksums thus may fail to even detect certain faults as pointed out by a recent work by Yao [26]; and 3) the method can only tolerate at most one fault during the decomposition. As the scale of supercomputing marches towards exascale, fault tolerance is becoming a key aspect in achieving the required performance at reasonable cost [21, 6, 3]. And assuming only one fault during the application run seems not appropriate in future large scale systems any more. To address more than one error in matrix triangularization, Du [8, 7] proposed a technique to tolerate two errors in solving linear system using partial pivoting LU decomposition. In this case, the decomposition cannot be corrected, but the result to the linear system can be recovered using the Sherman-Morrison-Woodbury formula. Handling beyond two errors would be more expensive than the LU decomposition itself. The fault model used is the same as in Luk and Park [14] thus suffers from the same problem.

Some researchers went in another direction in order to tolerate more faults effectively. Realizing that the offline approach taken by the traditional ABFT techniques have to face catastrophic error propagation at the end, researchers attempted to adapt checksum schemes for online error detection and correction [5, 25, 24]. The idea is that online ABFT catches errors early on when they are not propagated far away, therefore making it easier to correct. Online ABFT also can tolerate more errors that spread in time by avoiding errors compounding each other. The fault model used however is still arithmetic faults, and there still is no column checksums due to the difficulty in row pivoting.

A recent study [26] discovers that the fault models used in the previous ABFT works are not adequate even in detecting faults (Section 3 in [26]). This work proposes a global row and column checksums that can effectively detect errors and it is also an online approach. However error correction is not considered.

In this work we do not use an abstract fault model; rather we assume an architectural fault model and aim to detect and correct multiple errors. The architectural fault model is closer to what happens in real world and not only include all the fault models discussed above but also more improvements.

3. FAULT MODEL

The fault model for silent soft errors includes arithmetic faults that result in a wrong answer, for example 1+1=3. The other important fault is the memory system fault, manifesting as corrupted bits in storage cells. Memory faults could happen in main memory, in caches, registers, and other datapaths. We suppose one memory fault only affects one memory word; it can be multiple bits or single bit corruption.

It is useful to see how the architectural level faults manifest themselves in the algorithm level. Typically numerical algorithms deal with scalar numbers, vectors, and matrices. A variable may be mapped to multiple memory devices. For example the variable may be mapped to main memory, and cached in on-chip cache. It may also live in a register temporarily. The fault that affects the variable may be caused by corruptions in one of the mapped physical devices, and manifest themselves differently. For example if the main memory is corrupted, the mapped variable may read the corrupted value continuously until the memory is overwritten. If the corruption happens in cache, the variable may read incorrect value until the cache line is flushed. Therefore, a corrupted data element in program may sometimes read correct value but at other times read corrupted value.

4. THE CHECKSUM SCHEME

It is important to make a distinction between fault and error. For our purpose, a fault is a malfunction in the architecture, such as a bit flip in memory, cache, or registers. An error is the symptom due to the fault. Thus faults are the cause and errors are what we observe that are not correct. In designing numerical algorithms, errors are erroneous floating point variables. A single bit fault may lead to multiple errors, depending on how the faulty value is used. For algorithm designers and implementers, the problem to design fault tolerant algorithms is to find ways to detect and tolerate errors. In online ABFT framework, the problem can be further specified as to detect and tolerate errors resulting from one for *every error handling interval*. In this section, we will first study the error patterns of a single fault and how to tolerate them; then we will discuss how to design checksum schemes in LU decomposition; we will discuss how to put this technique in use in the very high performance LU decomposition package HPL; last we drop the assumption of precise arithmetic and deal with finite precision floating point arithmetic.

4.1 Error patterns and correction

We begin by studying the error patterns caused by a single fault in matrix multiplication, as matrix multiplication is the simplest dense matrix operation and it is an important part of the LU decomposition. We will see that memory faults may lead to multiple errors, while in contrast one arithmetic fault will only lead to one error in matrix multiplication.

Figure 1 shows four cases when one fault strikes. The fault could be an arithmetic fault or a silent data corruption (SDC). The red elements indicate errors. In subfigure (a), a single arithmetic error can only corrupt one element in the result, because the intermediate value produced by the faulty arithmetic operation is only used to calculate one element. In subfigure (b), a SDC in matrix A corrupts the whole row in the result C, because the corrupted element in A is used to calculate the whole row. In subfigure (c), the SDC occurs not in memory but in for example cache, or occurs later during the matrix multiplication. In this case a single SDC in matrix A causes partial row corruptions in C. In subfigure (d) a single SDC in matrix B causes partial column corruption in C. The important observation here is that *a single fault cannot cause errors in more than one row or column*. This observation enables us to design checksums that can correct all the error patterns caused by a single fault.

Next we discuss how to design checksum schemes to detect and correct up to one fault based on the fault patterns in figure 1. A matrix can have two types of checkums along its two dimensions: the checksum at the bottom of a matrix is called column checksum and the checksum to the right of a matrix is called row checksum. The column checksum encoded matrix is often denoted by a superscript A^c and the row checksum encoded matrix by superscript A^r. If a matrix has both then it is called fully checksummed and denoted by A^f. Mathematically, let e be the weight vector (or matrix

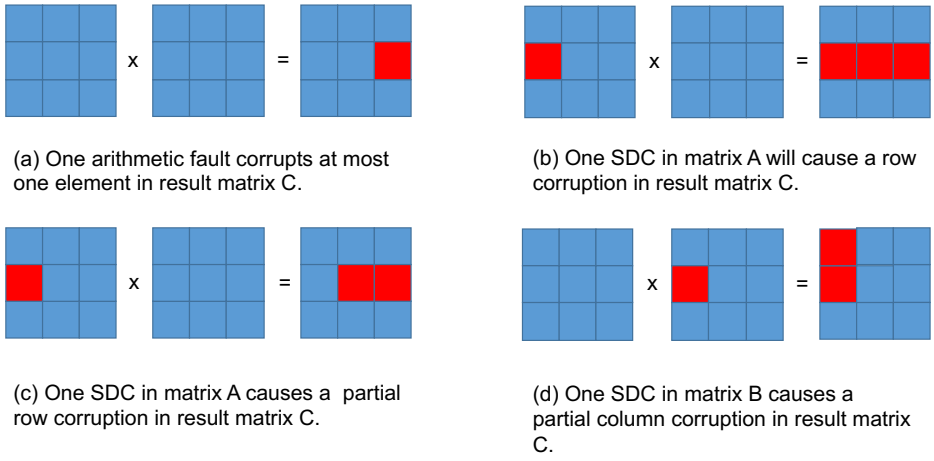


Figure 1: Error patterns for a single fault in matrix multiplication

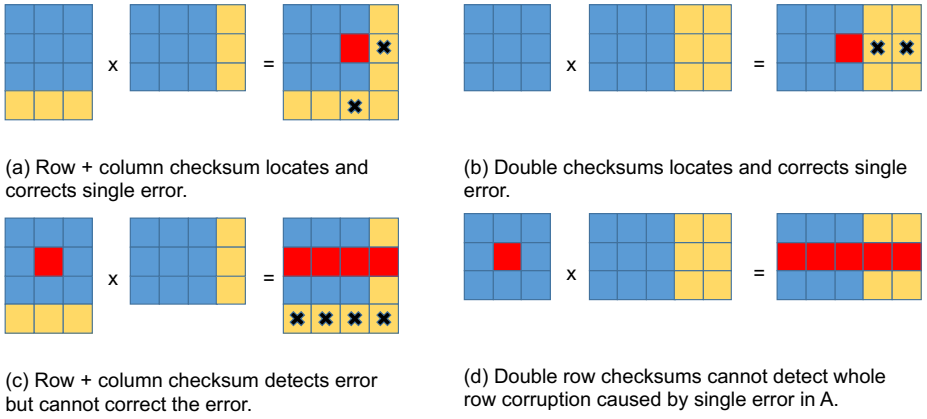


Figure 2: Checksums for matrix multiplication

in the case of multiple checksums) then:

$$A^c = \begin{bmatrix} A \\ e^T A \end{bmatrix}, \quad B^r = \begin{bmatrix} B & Be \end{bmatrix}, \quad C^f = \begin{bmatrix} C & Ce \\ e^T C & \end{bmatrix}$$

As shown in figure 2, we have multiple configurations of checksums. The yellow blocks are row or column checksums associated with the matrix. The red block indicates an incorrect element, and a black cross on a row/column checksum indicates that the row/column checksum is inconsistent with the respective row/column in matrix. We need at least two checksums to correct up to one error because the location and the magnitude of the error are two unknowns. For a single error in a matrix, either two row checksum, two column checksum, or one row plus one column checksums can detect and correct one error in matrix C. In subfigure (a), the error can be located at the intersection of the inconsistent row and column. The error can be recovered using either the row or column checksum [13], because the *checksums are correct*. In subfigure (b), a single error in matrix C can be detected and corrected using two row (weighted) checksums with different weights [24]. The location of the error and the magnitude of the error can be solved from the two checksums. In subfigure (c), a single SDC in matrix A causes a whole row corruption that result in an incorrect but consistent row. Because the row checksum is corrupted, it leaves us with only one column checksum which is inadequate to correct the errors. In subfigure (d), a single SDC in matrix A causes a whole row corruption with incorrect but consistent checksums. In this case the checksum scheme cannot detect the errors.

It is now clear that we need both row and column checksums to avoid the error detection failure. And to correct row/column corruptions we need two row checksums and column checksums, as shown in figure 3. In figure 3, a SDC in matrix A causes a whole row corruption in C detectable by the column checksums. The errors can be located and corrected on per column basis using the two correct column checksums. The row checksums are neither able to locate the errors nor correct them.

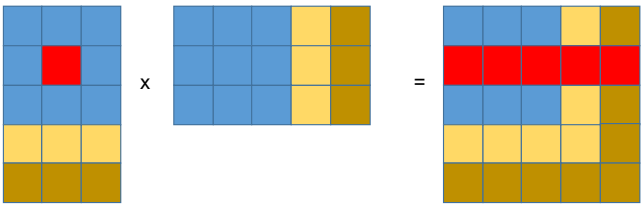


Figure 3: The checksum scheme that can tolerate single arithmetic fault or memory fault

Specifically, how do we locate and correct one erroneous element using two checksums? There is an easy to use encoding method. Suppose we encode a vector using two different weights $e_1 = [1, 1, \ldots, 1]^T, e_2 = [1, 2, \ldots, n]^T$. The vector is $a = [a_1, \ldots, a_n]$ and we have two correct encoded checksums of a:

$$r_1 = ae_1 = \sum_{i=1}^{n} a_i, \quad r_2 = ae_2 = \sum_{i=1}^{n} ia_i$$

Now suppose the computed $a' = [a'_1, \ldots, a'_n]$ has up to one erroneous element $a'_j \neq a_j$, where the location j is unknown to us. However when we verify the checksums:

$$\delta_1 = \sum_{i=1}^{n} a'_i - r_1 = a'_j - a_j \neq 0$$

$$\delta_2 = \sum_{i=1}^{n} ia_i - r_2 = j(a'_j - a_j) \neq 0$$

Then a simple division δ_2/δ_1 gives us the location j. The correct value of a_j can then be recovered using the correct checksum and the other correct elements of a: $a_j = a'_j - \sum_{i=1, i\neq j}^{n} a'_i$.

In this subsection the error patterns in matrix multiplication are discussed and checksums are devised to detect and correct errors, given that we have the desired checksums available. In the following subsection, how to maintain the checksums online is discussed in LU decomposition. Note that in LU decomposition the matrix multiplication is actually $C \leftarrow C - A \times B$ instead of $C \leftarrow A \times B$ so *correction through re-computation cannot be used* because the original C is overwritten.

4.2 Checksum scheme in LU decomposition

In this subsection the right-looking LU decomposition is briefly introduced. We first show that LU decomposition maintains global row and column checksums. Then we discuss the two adaptions to the LU decomposition that are essential in achieving good performance on modern cache based system and parallel computing.

LU decomposition factors a matrix A into the product of two triangular matrices (lower) L and (upper) U: $A \rightarrow L \times U$. The tiled right-looking variant of the LU algorithm works as shown in figure 4.

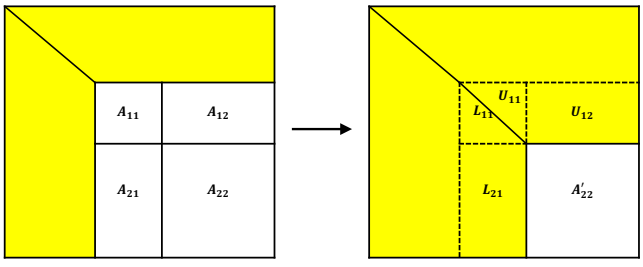


Figure 4: Tiled right-looking LU algorithm, one iteration

Figure 4 shows the state before and after an iteration in the algorithms. The algorithm is a series of iterations that keeps shrinking the trailing matrix until done. The yellow parts of the matrix indicate areas that have been factored and not active. For a certain iteration, the algorithms follows three steps: left panel factorization, top panel update, and trailing matrix update, described by the following equations:

$$\begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix} \rightarrow \begin{bmatrix} L_{11} \\ L_{21} \end{bmatrix} \times U_{11} \tag{1}$$

$$A_{12} \rightarrow L_{11} \times U_{12} \tag{2}$$

$$A'_{22} \leftarrow A_{22} - L_{21} \times U_{12} \tag{3}$$

The maintenance of checksums offline: In the original Huang and Abraham ABFT paper [13] it has been shown that if we LU decompose a full checksummed matrix A^f, we will end up with column checksummed L^c and row checksummed U^r:

$$\begin{bmatrix} A & Ae \\ e^T A & \end{bmatrix} \rightarrow \begin{bmatrix} L \\ e^T L \end{bmatrix} \times \begin{bmatrix} U & Ue \end{bmatrix} \tag{4}$$

where vector e is the checksum weights vector. This relationship can only be used to detect errors but not correct errors because in LU the errors will propagate to checksums too.

The maintenance of checksums online: If LU decompose a full checksum matrix, we will end up with a column checksummed L and row checksummed U. However multiple errors compound each other resulting in algorithmically uncorrectable errors. It would be desirable to detect and correct errors frequently during the factorizations to handle errors in a timely manner. In fact, we will show that at the end (or beginning) of each iteration, the factored left panel and top panel will be column checksummed and row checksummed, and the trailing matrix will be fully checksummed. We will show this claim inductively by first assuming the condition holds at the beginning of a certain iteration and prove that the condition holds at the end of the iteration. The initial condition clearly holds as we have a fully checksummed initial matrix.

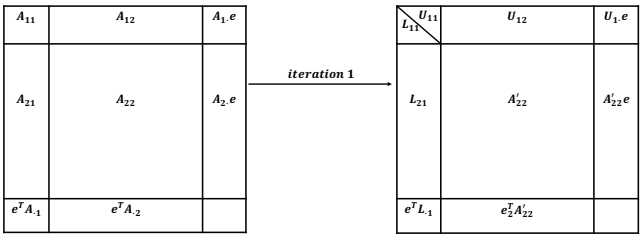


Figure 5: Tiled right-looking LU algorithm with checksums, one iteration

For simplicity we only examine the first iteration. As shown in figure 5, before the iteration we have the full checksums. After the left panel has been factorized according to equation (1), the column checksum associated with the left panel turns into the checksum of the factorized panel: $e^T A_{\cdot 1} \rightarrow e^T L_{\cdot 1}$. To see why this is true one only has to observe: 1) the factorized left panel will not be updated again therefore will stay unchanged through the end; 2) from equation (4) we know that at the end the left panel will be column checksummed. Thus we proved that the left panel factorization maintains column checksum. Similarly, the second step according to equation (2) maintains the row checksum of the top panel. Next we need to prove that after the trailing

matrix update according to equation (5), the trailing matrix will be fully checksummed. To see that we only have to apply the matrix multiplication to the checksums. Take the column checksums for example. The transformation done to the column checksums is depicted by:

$$\begin{aligned} &= e^T \begin{bmatrix} A_{12} \\ A_{22} \end{bmatrix} - e^T \begin{bmatrix} L_{11} \\ L_{21} \end{bmatrix} \times U_{12} \\ &= e_1^T (A_{12} - L_{11}U_{12}) + e_2^T (A_{22} - L_{21}U_{12}) \\ &= e_2^T (A'_{22}) \end{aligned} \tag{5}$$

which proves that the trailing matrix is fully checksummed by the second part of the checksum weights vector e_2.

4.3 The complete picture as in HPL

The previous subsection discusses the algorithmic structure of tiled right-looking LU decomposition, and the maintenance of checksums at each iteration in fault free execution. In this subsection we discuss what happens when faults strikes, namely the error patterns. Once we know the error patterns we can describe correction procedures. We will also deal with two more complications in HPL: partial row pivoting for numerical stability and 2d cyclic block distributions of matrix for load balance in distributed computing.

Error patterns: We examine the error patterns in the three steps during one iteration, and discuss detection and correction procedures. First, we look at the first step and the second step according to equations (1) and (2), namely the left and top panel factorization. Our first claim is that *any single fault that occur during the left and top panel factorization will lead to inconsistent checksums*, provided that the arithmetic are precise, i.e. no round-off errors. In other words, the error detection by checksums is precise. The reason that the error detection is precise is because we have both row and column checksums. If for example only row checksums are used, as pointed out by figure 5 in [26], certain faults strike in lower triangular L will not be detected. In our case the fault will be detected by the inconsistent column checksums. Depending on the location and timing of the fault, the error pattern could be very complex and both the row and column checksums will be contaminated and there is no easy algorithmic corrections, as shown in figure 6 (a). For this case we can use in-memory checkpointing and rollback specifically for the left and top panels. Once the checksum inconsistency is detected the computation can be rolled back to the beginning of the iteration. In HPL the in-memory checkpoint can be stored in the communication buffer for broadcasting L thus do not consume extra memory space. The overhead of memory copy of two panels is not significant.

For the trailing matrix update, as discussed earlier a single arithmetic fault only affects one element in the result thus easily correctable. More interesting cases are memory faults within L_{21} or U_{12}. For a single SDC in L_{21} or U_{12}, *the errors cannot be in more than one row or one column.* Assuming precise arithmetic, a single fault will trigger at least one row checksum inconsistency and one column checksum inconsistency. Therefore the error detection in trailing matrix update is precise, and furthermore the error patterns are within our capability to correct. For example in the case shown in figure 6 (b) a memory fault associated with an element in L_{21} causes partial row corruptions. In this

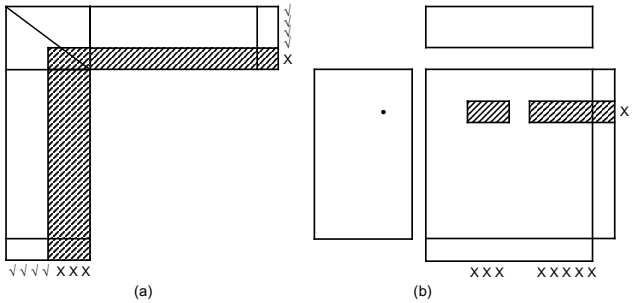


Figure 6: Tiled right-looking LU algorithm with checksums, one iteration. Shaded area are incorrect due to error propagation. Note that the affected checksums are also incorrect but the checksums are inconsistent therefore can be used to detect errors.

case the errors are easily located by the intersection of the inconsistent row and column checksums and corrected by the correct column checksums. It seems that one row checksum and one column checksum is sufficient to locate and correct any single fault in the trailing matrix update. However this is not true and will be explained next.

Parallel LU decomposition and 2d cyclic block distribution: On a multiprocessor machine a matrix is usually distributed onto a PxQ grid of processes according to 2d block cyclic scheme for load balance and scalability. As shown in figure 7, a 4x4 block matrix is distributed onto four processes. In the previous discussion we only look at the logical (global) view of the matrix and the checksum scheme is applied to the whole matrix. This view has some drawbacks. First, the fault tolerance capability is not scalable with the size of the matrix. Second, as the checksums are associated with the global matrix that are distributed, the error detection and correction requires inter process communication. To avoid these two drawbacks, we instead apply checksums to the process local matrix rather than the global matrix. In this way, the fault tolerance capability is fixed per process, and increases proportionally with the number of processes or the size of the matrix. Error detection and correction only involve local information.

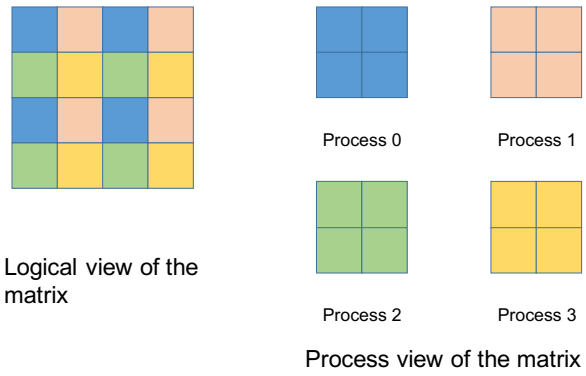


Figure 7: 2D block cyclic matrix distribution.

The online maintenance of the process local checksums are very similar to the global checksums. The error patterns can exhibit more patterns than that of global checksums. For

example, consider the first iteration and the matrix distribution in figure 7. In the trailing matrix update, for process 0 and 2, a memory fault in left panel will always produce one inconsistent row checksum but that is not the case for process 1 and 3. For process 1 and 3, a persistent memory corruption in L causes the trailing matrix update to exhibit the error pattern shown in figure 2 (d) where all row checksums are incorrect but consistent. In this case a single column checksum can only detect error; two column checksums are required to correct the errors. For process 0 and 2 we show that even a persistent memory fault in L can produce one inconsistent incorrect checksums. Similar to equation (5) and figure 5, suppose after the left panel is factorized it is corrupted in one element $L_{21} \rightarrow \widehat{L_{21}} := L_{21} + \alpha e_i e_j^T$. Then the trailing matrix A_{22} and its row checksums will be updated by the corrupted $\widehat{L_{21}}$ in the following way (the symbol with a hat indicates a corruption):

$$
\begin{aligned}
\widehat{A'_{22}} &\leftarrow A_{22} - \widehat{L_{21}}U_{12} \\
\widehat{CS(A'_{22})} &\leftarrow \begin{bmatrix} A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} e_1 \\ e_2 \end{bmatrix} \\
&= (A_{21} - \widehat{L_{21}}U_{11})e_1 + (A_{22} - \widehat{L_{21}}U_{12})e_2 \\
CS(\widehat{A'_{22}}) &= \widehat{A'_{22}}e_2 = (A_{22} - \widehat{L_{21}}U_{12})e_2 \\
CS(\widehat{A'_{22}}) - \widehat{CS(A'_{22})} &= (A_{21} - \widehat{L_{21}}U_{11})e_1 \\
&= \alpha e_i e_j^T U_{11} e_1
\end{aligned}
\tag{6}
$$

with the last equation indicating one inconsistent row checksum. Note that the equations confirm that only one row in the trailing matrix will be affected; the whole row is corrupted and so is the associated row checksum, but they are corrupted in a way that makes them inconsistent. The single row corruption can be handled by the double column checksum effectively. The above analysis also shows that the Example 3 in [26] is incorrect.

Partial row pivoting in LU: In practice unpivoted LU can easily break down due to numerical instability. To reduce the instability while not incur prohibitively high overhead, partial row pivoting is commonly used. However the row swapping in the pivoting disrupts the maintenance of the checksums. If the row checksums are swapped together with their respective rows the row checksums still maintain. Column checksums need to be fixed and not swapped. In process local checksums however, maintaining column checksums require more care. One row may be swapped with a row from another process, thus invalidating both checksums. Therefore the checksums must be updated when inter process swapping happens.

Putting them together The pseudo code algorithm 1 summarizes the error detection and correction logic. For brevity it is in the point of view of global matrix.

4.4 Round-off error bounds

In the last section we have shown that if we limit the faults to one per error handling interval and assume precise arithmetic, the error detection is both sound and precise. In practice the floating point arithmetic are not precise, soundness and precision cannot be attained simultaneously. As lack of soundness is not acceptable in fault tolerance, we thus strive to maintain soundness at some expense of precision. To do that, we derive *a priori* norm based error bounds for the round-off error, and use the upper bound as the threshold to distinguish architectural faults from floating point round-off errors. If the architectural faults alters the less significant bits in a floating point number and the result is still within round-off error bounds, no errors will be detected and the fault is deemed indistinguishable from round-off errors.

Algorithm 1 The fault tolerant HPL algorithm, global view.

Require: Fully checksummed matrix A^f and right hand side b
Ensure: $x = A^{-1}b$ in the presence of floating point soft errors, or signal errors
n is the size of A, B the blocking factor
for $i = 0$ to n step B **do**
$A(i:n, i:n) =: \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$
Factorize left panel $\begin{bmatrix} A_{11} \\ A_{21} \\ CS(A_{\cdot 1}) \end{bmatrix} \rightarrow \begin{bmatrix} L_{11} \backslash U_{11} \\ L_{21} \\ CS(L_{\cdot 1}) \end{bmatrix}$
Factorize top panel $\begin{bmatrix} A_{12} & CS(A_{12}) \end{bmatrix} \rightarrow \begin{bmatrix} U_{12} & CS(U_{12}) \end{bmatrix}$
Check column checksums for L and row checksums U
if Errors not algorithmically correctable **then**
Rollback to the start of this iteration
end if
Update the trailing matrix $A^f_{22} \leftarrow A^f_{22} - L^c_{21}U^r_{12}$
Check and correct full checksum matrix A^f_{22}
end for

Specifically, when we are verifying the checksums we need to compare the calculated sums to the checksums. Because floating point arithmetic has finite precision, those two may differ even in fault free execution. Our problem is now to bound the difference that round-off errors such that round-off errors alone would not violate the bound. Consider the matrix multiplication $C = AB$. A well known norm bound of the round-off errors in matrix multiplication is as follows [11].

$$
||fl(AB) - AB||_\infty \leq \gamma_n ||A||_\infty ||B||_\infty \tag{7}
$$

Assuming that the encoded matrix multiplication $C^f = A^c B^r$ is carried correctly, and the variable with a hat represents its floating point representation, we have the following result:

$$
\begin{aligned}
\left| \sum_{j=1}^{n} \widehat{c_{ij}} - \widehat{c_{i,n+1}} \right| &= \left| \sum_{j=1}^{n} (\widehat{c_{ij}} - c_{ij}) - (\widehat{c_{i,n+1}} - c_{i,n+1}) \right| \\
&\leq \left| \sum_{j=1}^{n} (\widehat{c_{ij}} - c_{ij}) \right| + |(\widehat{c_{i,n+1}} - c_{i,n+1})| \\
&\leq ||fl(C^f) - C^f||_\infty \\
&\leq \gamma_n ||A^c||_\infty ||B^r||_\infty
\end{aligned}
\tag{8}
$$

where $\gamma_n = nu/(1-nu)$ and u is the unit round-off error of the machine. For IEEE 754 64bit floating point number $u = 10^{-16}$. We thus obtained a bound of round-off errors that can be used as a threshold to distinguish architectural faults from floating point round-off errors. There is a similar bound to verify the row checksums.

5. OVERHEAD, PERFORMANCE, SCALABILITY, AND FAULT TOLERANCE CAPABILITY

In this section we model the fault tolerance capability, the execution time overhead, the scalability, and optimization of the proposed fault tolerant HPL.

5.1 Fault tolerance capability

For the error correction capability provided that errors can be detected, a natural question is how many errors or faults can be corrected? For each process in each error handling interval, any number of errors during the left and top panel factorization can be tolerated by the rollback. Multiple errors or one fault can be tolerated during the trailing matrix update, provided that the errors are within one row or one column. Note that the number of faults that can be tolerated is scalable with the number of processes and problem size, so at large scale enormous number of errors or faults can be tolerated as long as the faults do not burst into one error handling interval.

Compared to online ABFT (FT-ScaLAPACK) [5, 24]: Online ABFT may fail to detect memory error in the trailing matrix update where the process is not engaging in the left panel factorization. FT-ScaLAPACK cannot correct the errors caused by faults in the left panel during the matrix multiplication.

Compared to offline ABFT (Du, Luk) [7, 8, 14] : Our FT-HPL is resilient to much more faults. For non permanently sticky memory fault, for example faults in cache or registers, offline ABFT correction based on casting the fault back to low rank perturbations to the initial matrix no longer work. In fact, any fault that do not corrupt a variable for its entire lifespan will fail in offline ABFT fault tolerance scheme, as the fault do not fit in the abstract fault model. Thus the tolerable faults in offline ABFT schemes is a small subset of the more comprehensive fault models considered in this paper.

5.2 Execution time overhead

The fault tolerant LU decomposition introduces overheads in maintaining checksums, checking checksums periodically, and correcting errors if detected. As the analysis here only serves as a first order approximation of the performance, we use a widely used simple machine model. The communication time is modeled as $T = \alpha + \beta L$ where α is network latency and β is the reciprocal of network bandwidth. The computation of matrices and vectors can be modeled by the product of compute rate γ and number of floating point operations (FLOPs). The compute rate of BLAS3 operation such as matrix multiplication is γ_3 and the compute rate of BLAS2 operation such as matrix vector multiplication is γ_2. On modern architectures γ_2 is much lower than γ_3 so it is important to make the distinction. Let N be the size of the matrix A, B be the blocking factor, $P \times Q$ be the dimension of the process grid, then the run time of HPL LU decomposition is as follows [19]:

$$T_{\text{hpl}} = 2\gamma_3 \frac{N^3}{3PQ} + \beta N^2 \frac{3P+Q}{2PQ} + \alpha N \frac{(B+1)\log P + P}{B} \tag{9}$$

Checksum maintenance overhead: The overhead of checksum maintenance can be considered as the effectively increased matrix size. Adding two row checksums and two column checksums to the process local matrix, the global checksum matrix is bounded by $\max(N(1+2P/N), N(1+2Q/N))$. In a reasonable configuration of HPL, N/P and N/Q are the local dimensions of process local matrix that are around 10,000 therefore the enlargement of the global matrix size is around 0.02%. The resulting relative increase run time in equation 9 will be less than 0.1%, thus not a significant contribution to the run time overheads.

Checksum verification overhead: The periodical verification of the checksums is one major contribution to the run time overhead. The verification of checksums is a BLAS2 operation. The overhead of the verifications are:

$$\begin{aligned} T_{\text{check}} &= \frac{4\gamma_2}{PQ}(N^2 + (N-B)^2 + (N-2B)^2 + \cdots + B^2) \\ &= 4\gamma_2 \frac{N^3}{3BPQ} \end{aligned} \tag{10}$$

Compared to equation 9 the relative overhead is

$$\frac{T_{\text{check}}}{T_{\text{hpl}}} < \frac{2\gamma_2}{B\gamma_3} \tag{11}$$

Assuming a blocking factor B around 200 and BLAS2 operation is 5x slower than BLAS3 operations, the overhead is less than 5%. Different machines will have different ratio and different relative overhead.

5.3 Error correction overhead

This overhead is only present when errors are detected and correctable. The algorithmic error correction using checksums are non-significant. For the errors that are not algorithmically correctable by the checksums, the overhead is the lost work and rollback and recompute of the left panel factorization, which is empirically a small relative to the whole factorization.

5.4 Memory overhead

The fault tolerance needs extra memory space to store the checksums and the left panels. The extra space to store the checksums are less than 0.1% so not a significant overhead. The memory overhead of storing the left panel is more significant at $\frac{B}{N/Q}$. Again assuming a typical HPL configuration $B = 200, N/Q = 10,000$ the overhead is at 2%.

5.5 Impact on scalability

If we measure scalability by the parallel efficiency $\frac{T_{\text{ser}}}{PQT_{\text{hpl}}}$ which indicates how close it is to ideal parallel speedup, because the execution time overhead is bounded if memory usage per process is fixed and regardless of P, Q, the scalability of the fault tolerant HPL will remain the same as the original HPL which is excellent.

5.6 Tradeoffs between resilience and overhead

According to the overhead analysis and the detailed timing result from the experiments we found that the verification of the trailing matrix is one major overhead to the execution time. In fact when the trailing matrix verification is disabled the fault free execution time overhead dropped by half. In this section we discuss the tradeoff between fault

tolerance and overhead, and the insights to allow such tradeoffs to happen.

Let us take the point of view of one particular process. Suppose there is a grid of $P \times P$ processes and the matrix is distributed in 2d cyclic blocked manner. Since the LU decomposition works factorizes left and top panel sequentially from left to right and from top to bottom, the particular process engages into panel factorization every P iterations (in figure 8 $P = 4$). As we have discussed in the error patterns in matrix-matrix multiplication (TU), the errors propagate in a controlled way. In fact, if we skip the trailing matrix verification procedure at the end of iteration 1 and 2, we still can correct up to 1 fault happening during iteration 1,2, and 3 at the end of iteration 3. In this way we trade fault tolerance for reduced error checking overhead. The observation that allows us to make this tradeoff is that faults during iteration 1 will not propagate during iteration 2 and 3. However this is not true for PF as one fault in PF will propagate and cause massive errors in subsequent TU making the single fault uncorrectable. Thus the error handling procedure after each PF cannot be skipped to reduce fault tolerance.

The overhead analysis in the last section takes a global workload approach and assumes perfect load balance between the processes. But in parallel LU there is load imbalance during the panel factorizations where only a column of processes engage and other processes are waiting for the factorization result. It can be seen that PF is likely to be on the critical path. As PF depends on the TU immediately before, that TU is also likely to be on the critical path. The TU verification before PF thus is likely to be on the critical path. In fact from the experiments we found that by disabling only the TU verification immediately before a PF (shown in figure 8 OPT) the overall execution time drops almost as much as by disabling *all* TU verifications altogether. This significant reduction in overhead is therefore highly desirable, however it seems to break the promise that single fault during one error handling interval is tolerable. To remedy this problem, we only need to observe that, one fault in the last TU will cause the immediate subsequent PF verification to fail. The PF can be made non-destructive and once the PF fails the checksum verification, the error handling procedure for the previous TU is automatically invoked and the PF will restart. Therefore, the best tradeoff

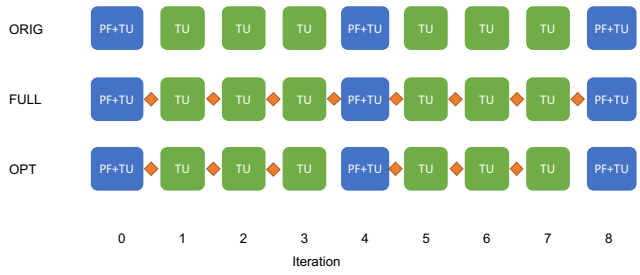


Figure 8: One process view in a 4x4 process grid: PF stands for (left and top) panel factorization and TU for trailing matrix update. The red diamond represents checksum verification point.

between fault tolerance and overhead is to disable only the TU verification immediately before a PF for every process. In this way the error handling interval remains short and the critical TU verification overhead is reduced significantly.

6. EXPERIMENTAL STUDY

In this section we empirically evaluate: 1) the fault coverage of the proposed FT-HPL in comparison to the state-of-the-art ABFT techniques by targeted fault injection; 2) the resilience of the FT-HPL scheme and implementation by randomly injecting various faults; 3) the cost of introducing such fault tolerance by measuring large scale executions.

6.1 Fault injection for fault coverage

In this subsection we experimentally compare the fault coverage of the state-of-the-art ABFT techniques that can apply to LU decomposition and HPL. We inject both arithmetic faults and memory faults to various locations in code and data at various times during the execution. We select several representative stages in one iteration to inject faults. Specifically, during the first iteration of LU algorithm, we inject faults right before the iteration and in the middle of the iteration (at iteration 2 of trailing matrix update). The arithmetic fault is simulated by modifying the output of a floating point multiplication. The memory fault is injected to matrix element (2,1) by modifying the data value. To precisely control where and how to inject a fault, we use the debugger GDB to stop the program and modify the program and data. During each run, we only inject one fault.

The fault coverage are summarized in table 1. As can be seen in table 1, no previous ABFT techniques provide as complete coverage to both arithmetic faults and memory faults happening at any time.

Table 1: Fault coverage for different ABFT techniques. "Before" means the fault affects data that is produced but not yet used. "Middle" means the fault affects data that is undergoing repeated use.

Fault category	Arithmetic	Memory	
Fault timing		Before	Middle
FT-HPL (this paper)	✓	✓	✓
FT-ScaLAPACK[24] /FTLU[5]	✓	✗	✗
FT-DGESV[8, 7]	✗	✓	✗

6.2 Fault injection experiments

We use a architectural fault injector F-SEFI [1] to implement the fault model and reveal the resilience of the FT-HPL implementation. Faults are injected at random time to a random instruction or memory locations that is to be used. Note that we inject faults into active memory to avoid masked faults that are never used. We model both floating point arithmetic faults and memory system faults. F-SEFI is based on QEMU, an architecture emulator. It works by intercepting the instructions of the application and alter the effect of the instructions to simulate arithmetic faults and memory faults. The application runs unmodified in the virtual machine and F-SEFI effectively simulate architecture correct execution (ACE) faults [17]. Memory system faults are modeled in detail: different level of stickiness associated with a memory address is used. In a cache based architecture, a variable in the program is mapped to multiple physical spaces in the memory hierarchy. When the image of the variable in different physical spaces is corrupted, the program perceives a certain stickiness of the error. For example, a corrupted main memory word is very sticky as it

Table 2: Fault tolerant for dense linear algebra: costs and fault tolerance capability. "Yes" means the faults can be tolerated; "No" means otherwise. The percentage indicates the execution time overhead against non fault tolerant LU implementation (PDGESV/PDGESV in ScaLAPACK, HPL_pdgesv in HPL).

Fault category	No Error	Arithmetic Faults		Memory Faults	
Number of faults	0	≤ 2	many	≤ 2	many
FT-HPL	5%	Yes, 5%	Yes, 5-35%[a,b]	Yes, 5%	Yes, 5-35%[a,b]
FT-ScaLAPACK[24]/FTLU[5]	8%	Yes, 8%	Yes, >8%[b]	No	No
FT-DGESV[8, 7]	1%	Partial[c], 1%	No	Partial[c], 1%	No
RedMPI[9][d]	$\geq$ 20%	Yes, $\geq$ 20%	Yes, $\geq$ 20%	Yes $\geq$ 20%	Yes,$\geq$ 20%

[a] Overhead depends on the impacted phase in HPL.
[b] To tolerate multiple faults they must be spaced out in time thus not overwhelming one error handling interval.
[c] The fault must happen in specific time and location to fit the algebraic model in [8, 7]. See table 1.
[d] To tolerate faults RedMPI need 200% more processors to form TMR at MPI rank level.

will read corrupted value until overwritten. On contrast, a corrupted cache word may only read corrupted value temporarily until it is flushed out, and subsequent read to the variable will read from main memory or lower level cache which has the correct value.

The configurations of the fault injection experiments are as follows. Four virtual machines are used with one MPI rank in each virtual machine. The problem size is 200x200 with blocking factor $B = 5$, which means that there are 40 intervals. During each run of the experiment, 5 faults are injected at random times to a random memory locations that are active. We take care not to inject two faults into one error handling interval which our FT-HPL cannot handle. Note that this setting injects a considerable amount of faults into a small problem size to stress the fault tolerance mechanism.

In total 300 repetitions of the experiment are performed. Among them, 252 cases (84%) successfully tolerated the injected faults and passed the residual check of the HPL application. In all passed cases, the injected faults are detected and corrected by our algorithms. Another 21 cases (7%) run to completion but failed to pass the residual check because in HPL application not all data structures and operations can be protected by our algorithm. The remaining 27 (9%) cases crashed or hung. In contrast, when subject to 5 random memory faults both FT-ScaLAPACK/FTLU and FT-DGESV would have success rate of 0%.

6.3 Overheads of fault free execution and error correction

In this section we evaluate how much execution time overhead is during fault free execution, and the cost of error correction in the presence of faults. The experiments are conducted on two clusters: 1) a small cluster TARDIS (up to 512 cores) for detailed overhead reduction experiments, and 2) TACC Stampede for large scale (up to 4096 cores) scalability and overhead experiments. The TARDIS is a 16 node cluster; each node is equipped with two sockets AMD 6272 processors (32 cores) clocked at 2.1GHz. Each node has 64 GB memory. The interconnect is Mellanox QDR InfiniBand. The TACC Stampede is currently the #10 on Top500.org November 2015 list. Each node has two Intel E5 8-core (Sandy Bridge) processors with core frequency 2.7GHz and 32 GB memory. Each core is capable to deliver 21.6GFLOP/s at maximum. The interconnect is FDR 56Gbps InfiniBand Mellanox switches using the 2-level Clos fat tree topology. Table 2 provides summarized comparison to state-of-the-art ABFT techniques in terms of overhead and fault coverage.

6.3.1 Overhead reduction and correction overhead

This set of experiments are done on TARDIS to investigate the overhead reduction effect discussed in subsection 5.6. In the fault free execution mode, four variants of implementations are measured: ORIG is the original unmodified Netlib HPL-2.1[19]; FULL implements the fault tolerance described in the last section; OPT implements an optimization technique that partially removes the trailing matrix checksum verification from the critical path; and FAULT is essentially FULL plus injected error that triggers all error correction procedures. In the non fault free execution, faults are injected via source code instrumentation to trigger all error checking and correction, thus demonstrating the maximum overhead of error correction. The process local matrix size is fixed at around 3000x3000, lower than a typical 10000x10000 configuration which will take much longer to complete. We use process grids $N \times 32$, with the number of nodes N being from 2 to 10, and the matrix size N being from 24000 to 51000 The block size is fixed at $B = 200$.

Figure 9 thus shows the execution time in weak scaling experiments. It can be seen that with fault free execution, the execution time overhead can be as low as 6% compared to the non fault tolerant original HPL implementation. This is the cost paid to be able to tolerate faults that can occur during the execution. Also the error correction procedures are very cheap and cost between 25% to 35% execution overhead at the maximum of its fault tolerance capability. Note that this is the time it takes to handle hundreds of faults or thousands of errors caused by the faults.

It is also worth noting that the OPT configuration has almost 50% overhead reduction over the FULL configuration which confirms the analysis that the trailing matrix verification immediately before panel factorization is on the critical path.

6.3.2 Scalability experiments

In the following texts we adopt the OPT strategy and look at the fault free overhead at large scale on TACC Stampede using up to 4096 cores (256 nodes, the maximum allowed scale without special request). For HPL the efficiency in terms of floating point operations per second (FLOP/s) per core increases when memory usage per process increases. In the first set of experiments we use only a small fraction of memory available to avoid exceedingly long experiment

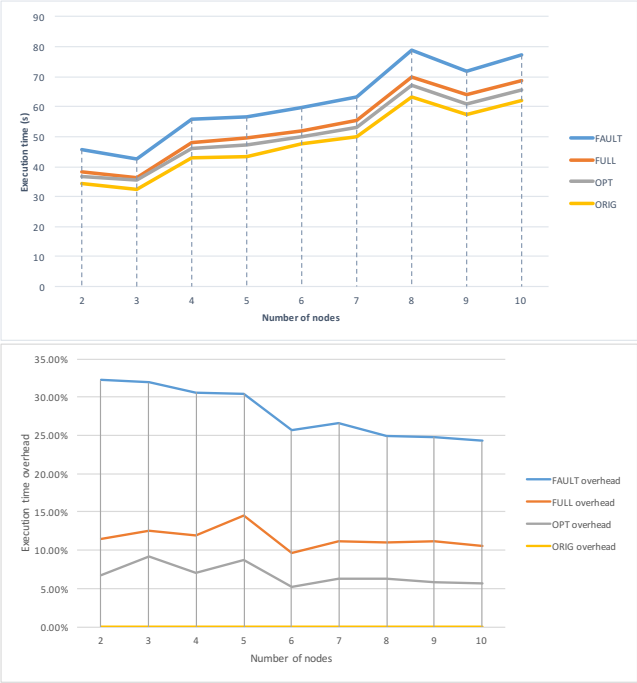
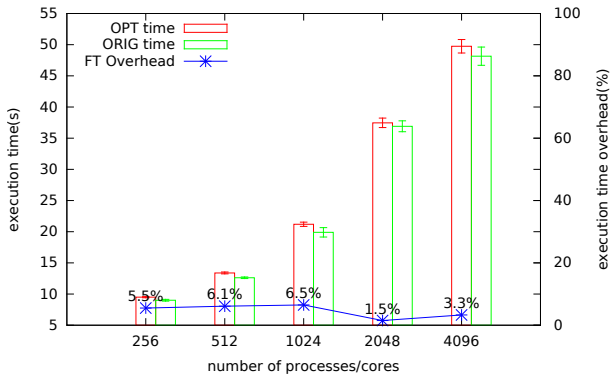
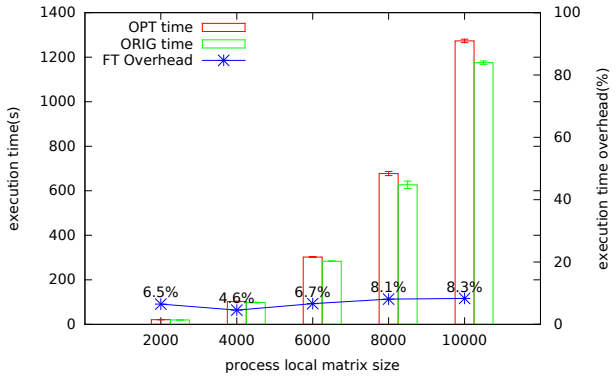


Figure 9: The execution time of FAULT, FULL, OPT, and ORIG HPL with varying number of nodes as X-axis. Each node comes with 32 computing cores.



(a) Fault free execution time for optimized fault-tolerant HPL (OPT) and the original HPL (ORIG). The process local matrix size is fixed at 2000 × 2000 while the number of processes/cores is scaling from 256 to 4096.



(b) Fault free execution time for optimized fault-tolerant HPL (OPT) and the original HPL (ORIG). The number of processes/cores is fixed at 1024 while the process local matrix size scales from 2000 × 2000 to 10000 × 10000.

Figure 10: Fault free execution time for optimized fault-tolerant HPL.

execution time (a single HPL run at its maximum problem size could take hours for 4096 cores). In the second set of experiments we fix the number of computing elements at 1024 cores and increase the problem size to observe the trend of overhead. From these two sets of experiments we can get an empirical idea of the overhead in introducing the resilience into HPL. The results are shown in figure 10

Reproducing large scale parallel experiments is difficult; so we strive to improve the interpretability [12] by providing more contexts and data. Since the execution time of HPL on a typical computing cluster is slightly indeterministic on Stampede, we collected enough measurements until the 99% confidence interval is around 5% of the reported mean measurements, following the recommendations from [12]. Also for this particular experiments on TACC Stampede we strongly suspect that there was an abnormal node with significantly slower network interface. If such node is included in the resource allocation the job will be significantly delayed by at least 20%. We base this conclusion on the following two reasons: 1) the measurements strongly exhibit two clusters around two modes. Any one measurement belonging to one cluster will appear as outlier for the other cluster using Tukey's outlier classification method. 2) jobs involving more nodes have a higher portion of such abnormally slow measurements: for 1024 cores we got 1 every 20 measurements; for 2048 cores we got 1 every 10 measurements; for 4096 cores 1 every 2 measurements. To eliminate the interference of such slow node we remove the measurements that are abnormally slow.

7. CONCLUSION

Fault model is the deciding factor on design of ABFT algorithms. In this work we seek to close the gap between what occurs at the architecture level and what the algorithm expects. We explore the challenges in designing ABFT algorithms under a general architectural fault model that allows both arithmetic and memory system faults comprehensive both temporally and spatially. By dividing the execution into many error handling intervals and aim at tolerating single fault in each error handling interval, we build a process local checksum scheme that achieves scalable fault tolerance (one fault per iteration per process) at around 5% fault free execution time overhead and less than 35% execution time overhead when facing maximum number of faults. Targeted fault injection shows that the comprehensive fault model cannot be handled by existing state-of-the-art ABFT techniques but will be effectively tolerated by FT-HPL scheme. Random fault injection shows that our FT-HPL implementation can tolerate 84% of the cases where 5 faults occur within less than 1 second. Such low overhead and high fault tolerance under comprehensive fault model makes the new ABFT in dense linear algebra practical and attractive in extreme scale systems, on unreliable commodity hardwares, or in hostile environments.

Acknowledgments

The authors would like to thank the anonymous reviewers for their insightful comments and valuable suggestions. This work is partially supported by the NSF grants CCF-1305622, ACI-1305624, CCF-1513201, the SZSTI basic research program JCYJ20150630114942313, and the Special Program for Applied Research on Super Computation of the NSFC-Guangdong Joint Fund (the second phase).

8. REFERENCES

[1] F-SEFI: A Fine-Grained Soft Error Fault Injection Tool for Profiling Application Vulnerability. In *IPDPS'14*.

[2] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[3] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir. Toward Exascale Resilience: 2014 update. *Supercomputing frontiers and innovations*, 1(1):5–28, June 2014.

[4] Z. Chen. Online-ABFT: An Online Algorithm Based Fault Tolerance Scheme for Soft Error Detection in Iterative Methods. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 167–176, New York, NY, USA, 2013. ACM.

[5] T. Davies and Z. Chen. Correcting soft errors online in LU factorization. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, pages 167–178. ACM, 2013.

[6] N. DeBardeleben, J. Laros, J. T. Daly, S. L. Scott, C. Engelmann, and B. Harrod. High-end computing resilience: Analysis of issues facing the HEC community and path-forward for research and development. *Whitepaper*, 2009.

[7] P. Du, P. Luszczek, and J. Dongarra. High Performance Dense Linear System Solver with Soft Error Resilience. In *2011 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 272–280, Sept. 2011.

[8] P. Du, P. Luszczek, and J. Dongarra. High Performance Dense Linear System Solver with Resilience to Multiple Soft Errors. *Procedia Computer Science*, 9:216–225, 2012.

[9] 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. SC '12, pages 78:1–78:12, Los Alamitos, CA, USA, 2012.

[10] P. Fitzpatrick and C. Murphy. Fault tolerant matrix triangularization and solution of linear systems of equations. In *Proceedings of the International Conference on Application Specific Array Processors, 1992*, pages 469–480, Aug. 1992.

[11] G. H. Golub and C. F. V. Loan. *Matrix Computations*. JHU Press, Dec. 2012.

[12] T. Hoefler and R. Belli. Scientific Benchmarking of Parallel Computing Systems: Twelve Ways to Tell the Masses when Reporting Performance Results. SC '15, pages 73:1–73:12, New York, NY, USA, 2015.

[13] K.-H. Huang and J. Abraham. Algorithm-Based Fault Tolerance for Matrix Operations. *IEEE Transactions on Computers*, C-33(6):518–528, June 1984.

[14] F. T. Luk and H. Park. An Analysis of Algorithm-based Fault Tolerance Techniques. *J. Parallel Distrib. Comput.*, 5(2):172–184, Apr. 1988.

[15] T. May and M. H. Woods. Alpha-particle-induced soft errors in dynamic memories. *IEEE Transactions on Electron Devices*, 26(1):2–9, Jan. 1979.

[16] S. E. Michalak, K. W. Harris, N. W. Hengartner, B. E. Takala, S. Wender, and others. Predicting the number of fatal soft errors in Los Alamos National Laboratory's ASC Q supercomputer. *Device and Materials Reliability, IEEE Transactions on*, 5(3):329–335, 2005.

[17] S. Mukherjee. *Architecture design for soft errors*. Morgan Kaufmann, 2011.

[18] E. Normand. Single event upset at ground level. *IEEE transactions on Nuclear Science*, 43(6):2742–2750, 1996.

[19] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers.

[20] J. Sloan, D. Kesler, R. Kumar, and A. Rahimi. A numerical optimization-based methodology for application robustification: Transforming applications for error tolerance. In *2010 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 161–170, June 2010.

[21] M. Snir and et. al. Addressing failures in exascale computing. *International Journal of High Performance Computing Applications*, 28(2):129–173, May 2014.

[22] J. Von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. *Automata studies*, 34:43–98, 1956.

[23] J. H. Wilkinson, J. H. Wilkinson, and J. H. Wilkinson. *The algebraic eigenvalue problem*, volume 87. Clarendon Press Oxford, 1965.

[24] P. Wu and Z. Chen. FT-ScaLAPACK: Correcting Soft Errors On-line for ScaLAPACK Cholesky, QR, and LU Factorization Routines. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, pages 49–60, New York, NY, USA, 2014. ACM.

[25] P. Wu, C. Ding, L. Chen, T. Davies, C. Karlsson, and Z. Chen. On-line soft error correction in matrix–matrix multiplication. *Journal of Computational Science*, 4(6):465–472, Nov. 2013.

[26] E. Yao, J. Zhang, M. Chen, G. Tan, and N. Sun. Detection of soft errors in LU decomposition with partial pivoting using algorithm-based fault tolerance. *International Journal of High Performance Computing Applications*, page 1094342015578487, Apr. 2015.

[27] J. F. Ziegler and H. Puchner. *SER–history, Trends and Challenges: A Guide for Designing with Memory ICs*. Cypress, 2004.