# Coded QR Decomposition for Solving System of Linear Equations

Quang Minh Nguyen[1], Haewon Jeong[2] and Pulkit Grover[2]
[1] Department of Computer Science, National University of Singapore
[2] Department of Electrical and Computer Engineering, Carnegie Mellon University

***Abstract*—QR decomposition of a matrix is one of the essential operations that is used for solving linear equations and finding least-squares solutions. We propose a coded computing strategy for parallel QR decomposition with applications to solving a full-rank square system of linear equations in a high-performance computing system. Our strategy is applied to the parallel Gram-Schmidt algorithm, which is one of the three commonly used algorithms for QR decomposition. Conventional coding strategies cannot preserve the orthogonality of $Q$. We prove a condition for a checksum-generator matrix to restore the degraded orthogonality of the decoded $Q$ through low-cost post-processing, and construct a checksum-generator matrix for single-node failures. We obtain the minimal number of checksums required for single-node failures under the *"in-node checksum storage setting"*, where checksums are stored in original nodes, and further adapt the coded QR decomposition to this setting.**

## I. Introduction

Motivated by the widespread use of large-scale machine learning computations, *coded computing* has been an active area of research [1]–[5], where the aim is to efficiently add redundancies to make computing more robust to uncertainties and accelerate computing. In this work, we consider protecting compute nodes from failures in high-performance computing (HPC) systems. Building a reliable supercomputer has been a long-standing problem, but the emergence of exascale computing poses new challenges that require a new and innovative solution that goes beyond traditional reliability techniques. More than 20% of the computing capacity in today's HPC systems is wasted due to failures and ensuing recovery [6], and this wastage is only expected to grow as the system size grows. To reduce the overhead of fault tolerance in upcoming HPC systems, *algorithm-based fault-tolerance (ABFT)* for HPC has been suggested [7]–[17], the core idea of which is essentially the same as coded computing: adding encoded redundancy tailored to a given numerical algorithm.

In this work, we study coded computing strategy for QR decomposition. QR decomposition factors a matrix into a product of an orthogonal matrix ($Q$) and an upper triangular matrix ($R$). It is an essential building block of linear algebraic computations as it provides a numerically stable method for solving linear equations, and is extensively used in the linear least squares problem (e.g., linear regression). As it is an important computation primitive, ABFT for parallel QR decomposition has been studied in the HPC literature [10], [12], [17]. In this paper, we not only introduce a new computation primitive that has not been considered in the coded computing literature, but also incorporate practical system assumptions that are used in HPC algorithms. Our main contributions are summarized below:

- Previous works in ABFT for QR decomposition studied applying coding on the *Householder algorithm* or the *Givens Rotation algorithm*. Our work is the first to consider applying coding on the *Gram-Schmidt algorithm* (and its variants). We show that using our strategy throughout the Gram-Schmidt algorithm[1], vertical/horizontal checksum structures are preserved (Section III).
- Simply applying linear coding cannot protect the $Q$-factor as the orthogonality is not preserved after linear transforms. To circumvent this issue, we propose an innovative post-processing technique to restore the orthogonality of the Q-factor after coding. We show that if the checksum-generator matrix satisfies certain conditions, with low-cost post-processing, we can perform QR factorization to solve a full-rank square system of linear equations. We propose a construction of such checksum-generator matrix for single node failures (Section IV).
- We consider practical data distribution. Throughout the paper, we assume that the matrices are stored block-cyclically. In HPC applications, matrices are almost always distributed block-cyclically for load-balancing. Furthermore, in Section VI, we consider in-node checksum storage setting where we store the coded data (checksums) in original processors instead of adding extra processors for fault tolerance.

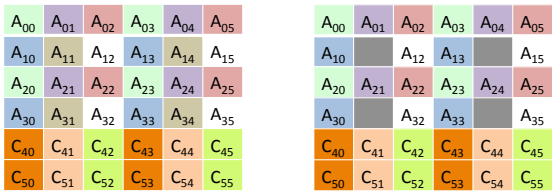
This work focuses on the single-node failure case as it is the most common scenario in HPC. Generalizing our code construction to multiple-node failure scenarios and beyond QR decomposition would be interesting to many intriguing future work. Considering a realistic model of data distribution and communication used in HPC, as is done here, not only brings coded computing closer to practice, but also opens up intriguing theoretical questions.

## II. Background and System Model

### A. QR Decomposition

QR decomposition decomposes a matrix $A$ into a product $A = QR$ of an orthogonal matrix $Q$ (i.e. $Q^TQ = I$) and an upper triangular matrix $R$. There are three classes of commonly-used algorithms to compute QR factorization

[1]Householder, Givens Rotation, Gram-Schmidt are the three most well-known algorithms for QR decomposition




(a) Failure-free state (b) Single-node failure

Figure 1: Out-of-node checksum storage for vertical checksums with $p_r = 2, p_c = 3$. (a) Six systematic processors (green, purple, red, blue, brown and white) own the original data blocks and three extra vertical checksum-processors (orange, light orange, light green) own the checksum-blocks $C_{ij}$. (b) Lost data-blocks are grayed out for the case where the brown node fails.

in HPC: Gram-Schmidt (GS) and Modified Gram-Schmidt (MGS) [18]–[20], Householder Transformation [21], [22] and Givens Rotation [23]. In this work, we consider MGS [19], which is an improved version of the GS algorithm.

We specifically consider solving a system of linear equations $A\boldsymbol{x} = \boldsymbol{b}$, where $A$ is square and full-rank, as the end application of QR decomposition in this work. Solving square and non-singular system of linear equations is a fundamental building block for many applications in HPC [24]–[26], and uses QR decomposition in practice due to its guaranteed stability and computational efficiency [27].

### B. System Model

We assume *2D block cyclic distribution* of a matrix where a non-singular $n \times n$ matrix $A$ is distributed block-cyclically on $P = p_r \times p_c$ processors with block size $b$, and each $b \times b$ block $A_{ij}$ is owned by processor $\Pi(i,j) = (i \bmod p_r) + p_r(j \bmod p_c)$. $N = \frac{n}{b}$ denote the number of data blocks of a block-column or a block-row. This is how matrix algorithms are implemented in practice for load balancing.

Vertical checksums $G_vA$ (resp. horizontal checksums $AG_h$) are checksum rows (resp. columns) which extend the input matrix $A$ vertically (resp. horizontally) and are controlled by the checksum-generator matrix $G_v$ (resp. $G_h$). For fault tolerance, we encode the matrix $A$ with both vertical and horizontal checksums as follows: $\widetilde{A} = \begin{bmatrix} A & AG_h \\ G_vA & G_vAG_h \end{bmatrix}$.

We consider the *out-of-node checksum storage:* (Figure 1a) The vertical (resp. horizontal) checksums are distributed over the new set of $p_c$ vertical (resp. $p_r$ horizontal) checksum processors. Each checksum processor is protected and stores the same amount of data as a systematic processor to ensure load balancing and efficient parallelism.

### C. Failure Model and Real-time Recovery

We focus on recovering from *"fail-stop errors"*, which is a realistic failure model in HPC [28], and is similar to common assumptions in coded computing. In this model, a failure corresponds to a systematic processor that completely stops responding, and loses its part of the global data. We assume that the identity of the processor that fails is provided by some external source (e.g. Message Passing Interface (MPI) library [29]). We focus on the *single-node failure* scenario, i.e., at any step of the QR decomposition at most one failure can occur. We assume that all the data owned by the failed processor is lost when it fails. The failure can strike at any point during the execution of QR decomposition, immediately triggering the recovery process. Computation continues once the system has recovered from its latest failure. Figure 1b illustrates an example of lost data-blocks.

### D. Related Work

State-of-the-art ABFT techniques utilizing coded computation have considered Householder [10], [12] and Givens Rotation [17], all of which only protect $R$ via coding and rely on replication for $Q$ protection. While no work on coded MGS exists, its fault-tolerance via replication was proposed in [30]. As MGS algorithms directly compute columns (or rows) of the $Q$ matrix subsequently used for $R$ computation, coding strategy which can protect $Q$ is the main challenge and the pivotal building block in the design of coded MGS. $Q$-factor protection is challenging: it was shown in [10] that conventional coding approach, which linearly encodes $A$, is not possible for $Q$ protection because the modified $Q$ factor retrieved from the coded computation is not orthogonal.

### E. Problem Definition

We specifically consider MGS algorithm [19] for QR decomposition in our coding design, where $A, Q$ and $R$ are distributed over processors using the 2D block-cyclic distribution and every node is vulnerable to fail-stop failures (Section II-C). As encoding input matrix $A$ naturally enforces fault-tolerant computation, we attribute the main limitation to the unsophisticated decoding of $Q$-factor. We thus allow post-processing after the decoding phase to restore the degraded orthogonality of the decoded $Q$-factor. Post-processing can transform $A$ into an alternative form to be used in place of $A$ in the end application- solving the square and non-singular system of linear equations $A\boldsymbol{x} = \boldsymbol{b}$.

Our goal is to design a novel coding strategy for MGS with full protection, *i.e.,* comprised of $Q$-factor and $R$-factor protection, to tolerate any single-node failure during the the computation. As an extension, for the *in-node checksum storage* recently proposed by [10], we aim to address the optimality on the number of checksums required for single-node failure, which has remained an open question.

## III. Horizontal/Vertical checksums for MGS

Checksum-preservation is the key idea of all the previous work on coded QR decomposition including Householder [10], [12] and Givens Rotations [17]. For MGS, we hereby show how horizontal and vertical checksums added to the input matrix are respectively preserved as checksums for $R$-factor and $Q$-factor throughout the computation. As the QR factorization of a rank-deficient matrix (encoded $\widetilde{A}$) is not unique, this property is not trivial and depends on specific algorithmic structure.

### A. MGS algorithm

For extreme-scale QR factorization, MGS is the favored choice thanks to its low computational cost and ease of implementation. The pseudo-code of sequential MGS is given in

Algorithm 1, where at the $i^{th}$ iteration, the algorithm computes column $\boldsymbol{q_i}$ of the $Q$-factor and row $\boldsymbol{r_i}$ of the $R$-factor. For parallel implementation in practice, block MGS (BMGS) [31] is widely used as its parallel version (PBMGS) can better utilize the Level-3 operations in HPC. We refer to parallel MGS as PBMGS.

**Input:** $\widetilde{A} = \begin{bmatrix}\widetilde{\boldsymbol{a}}_1 & \widetilde{\boldsymbol{a}}_2 & \cdots & \widetilde{\boldsymbol{a}}_{n+d}\end{bmatrix}$ is $(n+c)\times(n+d)$ matrix
**Output:** $Q = \begin{pmatrix}\boldsymbol{q_1} & \boldsymbol{q_2} & \cdots & \boldsymbol{q_{n+d}}\end{pmatrix}$ is $(n+c)\times(n+d)$ matrix , and $R = (r_{ij})$ is $(n+d)\times(n+d)$ matrix
**Result:** $\widetilde{A} = QR$, where $Q^TQ = I$

**for** $i = 1$ **to** $n$ **do**
  $\boldsymbol{u_i} = \widetilde{\boldsymbol{a}}_{\boldsymbol{i}}$
**end**
**for** $i = 1$ **to** $n+d$ **do**
  $r_{ii} = ||\boldsymbol{u_i}||_2$
  $\boldsymbol{q_i} = \boldsymbol{u_i}/r_{ii}$
  **for** $j = i+1$ **to** $n+d$ **do**
    $r_{ij} = \boldsymbol{q_i}^T\boldsymbol{u_j}$
    $\boldsymbol{u_j} = \boldsymbol{u_j} - r_{ij}\boldsymbol{q_i}$
  **end**
**end**

**Algorithm 1:** Modified Gram-Schmidt

### *B. Checksum-preservation for MGS*

In this section, we illustrate how checksums added to the input matrix are preserved throughout the computation.

The $n \times n$ input matrix $A$ is encoded as (Section II-B):

$$\widetilde{A} = \begin{bmatrix} A & AG_h \\ G_vA & G_vAG_h \end{bmatrix} \tag{1}$$

of size $(n+c)\times(n+d)$, where $G_h$ and $G_v$ are respectively checksum-generator matrices of size $n\times d$ and $c\times n$. We assume that $c$ and $d$ are small to keep the overhead negligible. QR decomposition via MGS is further performed on the encoded matrix $\widetilde{A}$. The algorithm executes $T$ iterations $t = 1,\ldots,T$ where at the end of each iteration the algorithm maintains the update of the $Q$-factor $Q^{(t)}$ of size $(n+c)\times(n+d)$ and the $R$-factor $R^{(t)}$ of size $(n+d)\times(n+d)$. The initial values of the $Q$-factor and $R$-factor are $Q^{(0)} = \widetilde{A}$ and $R^{(0)} = 0_{(n+d)\times(n+d)}$ respectively. At the end of the last iteration, we retrieve the orthogonal $Q = Q^{(T)}$ and the upper triangular $R = R^{(T)}$ as the output for $\widetilde{A} = QR$.

We further consider the submatrices of $Q^{(t)}$ and $R^{(t)}$:

$$Q^{(t)} = \begin{bmatrix} Q_1^{(t)} \\ Q_2^{(t)} \end{bmatrix}, R^{(t)} = \begin{bmatrix} R_1^{(t)} & R_2^{(t)} \end{bmatrix}$$

where $Q_1^{(t)}$: $n\times(n+d)$, $Q_2^{(t)}$: $c\times(n+d)$, $R_1^{(t)}$: $(n+d)\times n$, and $R_2^{(t)}$: $(n+d)\times d$. Lemma 1 shows that the original vertical checksums $G_vA$ and horizontal checksums $AG_h$ are translated to checksums $G_vQ_1^{(t)}$ for $Q$-protection and $R_1^{(t)}G_h$ for $R$-protection during the computation. The proofs for Lemma 1 and Corollary 1.1 can be found in [32, Appendix A].

**Lemma 1.** *For both sequential and parallel MGS, the following checksum relations hold for $t \in [0,T]$:*

$$Q_2^{(t)} = G_vQ_1^{(t)} \tag{2}$$

$$R_2^{(t)} = R_1^{(t)}G_h \tag{3}$$

**Corollary 1.1.** *At the end of the QR decomposition, the factorization of $\widetilde{A}$ has the final form:*

$$\widetilde{A} = \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix}\begin{bmatrix} R_1 & R_2 \end{bmatrix} = \begin{bmatrix} Q_1 \\ G_vQ_1 \end{bmatrix}\begin{bmatrix} R_1 & R_1G_h \end{bmatrix} \tag{4}$$

*where $Q_i = Q_i^{(T)}$ and $R_i = R_i^{(T)}$.*

## IV. $Q$-Factor Protection

In this section, we discuss how coding can be applied to the parallel Gram-Schmidt algorithm to protect the left $Q$ factor (an orthogonal matrix). From (20), we have both horizontal and vertical checksums, but in this section we will only consider the vertical checksum for simpler presentation. When we use $A$, one can regard it as $\begin{bmatrix}A & AG_h\end{bmatrix}$.

In [10, Theorem 5.1], it was concluded: *"Q in Householder QR factorization cannot be protected by performing factorization along with the vertical checksum."* The basis of this claim was that in the output retrieval $A = Q_1R$ after the QR factorization of the vertically encoded matrix $\widetilde{A}$:

$$\widetilde{A} = \begin{bmatrix} A \\ GA \end{bmatrix} = \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix} R, \tag{5}$$

$Q_1$ is not orthogonal, i.e. $Q_1^TQ_1 \neq I$. Thus, $Q_1R$ is not the correct QR factorization of $A$. While the theorem statement is limited to the Householder, this reasoning is generally beyond such specific algorithm.

An important contribution of our work is that we can convert $Q_1$ into an orthogonal matrix with a very small amount of computation. In Section IV-A, we prove that if the checksum-generator matrix satisfies certain conditions (given in Theorem 2), there exists a low-cost linear transform that orthogonalizes $Q_1$. In Section IV-B, we propose a checksum-generator matrix construction that satisfies the conditions given in Theorem 2 while providing resilience to any single node failure. Finally, we show through careful analysis that the overhead of fault tolerance including encoding, failure recovery, and low-cost post-orthogonalization is negligible.

### *A. Low-cost post-orthogonalization*

How "non-orthogonal" is $Q_1$? Can we still utilize $Q_1$ to recover the original QR factorization? We show that with a low-cost linear transform, $Q_1$ can be transformed into an orthogonal matrix, if the checksum-generator matrix $G$ satisfies certain conditions.

**Theorem 2.** *Let $G_1, V$ be submatrices of the vertical checksum-generator matrix $G$ as follows:*

$$G = \begin{bmatrix} G_1 & V \end{bmatrix}, \tag{6}$$

*where $G_1$ and $V$ have dimensions $c\times c$ and $c\times(n-c)$, respectively. Let $G_0$ be an $n\times n$ by matrix as follows:*

$$G_0 = \begin{bmatrix} I_c + G_1 & V \\ V^T & -I_{n-c} \end{bmatrix}. \tag{7}$$

*If $G$ satisfies the following condition:*

$$G_1 = -\frac{1}{2}VV^T, \tag{8}$$

*we can prove the following:*

*Claim 1: $G_0Q_1$ is orthogonal, i.e. $(G_0Q_1)^T(G_0Q_1) = I$.*

*Claim 2: $G_0$ is invertible.*

The proof for Theorem 2 is in [32, Appendix B-A].

First, notice that the matrix $G_0$ is very sparse as the bottom-right submatrix is simply an $(n-c)\times(n-c)$ identity matrix, and $c$ is negligible. Claim 1 in Theorem 2 suggests that by multiplying this sparse matrix $G_0$, we can convert $Q_1$ into an orthogonal matrix. We now demonstrate how we can use $G_0Q_1$ in place of the original $Q$ in solving a full-rank square system of linear equations $A\boldsymbol{x} = \boldsymbol{b}$.

Let $A' = G_0A$. Then the QR factorization of $A'$ will be:

$$A' = (G_0Q_1)R \tag{9}$$

with the left factor $(G_0Q_1)$ and the right factor $R$. As $G_0$ is invertible by Claim 2 in Theorem 2,

$$A\boldsymbol{x} = \boldsymbol{b} \iff (G_0A)\boldsymbol{x} = G_0\boldsymbol{b} \tag{10}$$

The linear system on the right side can be solved using the QR factorization of $A'$ given in (9). i.e.,

$$(G_0Q_1)R\boldsymbol{x} = G_0\boldsymbol{b}, \tag{11}$$

$$(G_0Q_1)^T(G_0Q_1)R\boldsymbol{x} = (G_0Q_1)^T(G_0\boldsymbol{b}), \tag{12}$$

$$R\boldsymbol{x} = (G_0Q_1)^T(G_0\boldsymbol{b}). \tag{13}$$

Then, we can perform triangular solve to get the final answer $\boldsymbol{x}$. Remember that we already have $Q_1$ and $R$ from the QR factorization of the encoded matrix $\widetilde{A}$. Hence, all we need to perform in the above steps is computing post-orthogonaliza tion, $G_0Q_1$ and $G_0\boldsymbol{b}$. We show in Theorem 3 that the overhead of post-orthogonalization is negligible.

### B. Checksum-Generator Matrices for Single-Node Failures

The low-cost post-orthogonalization scheme exists under the constraint (8) on the checksum-generator matrix $G$. One crucial question is whether we can construct $G$ that has good error correction/detection capability while satisfying (8). In this subsection, we present one such construction of $G$ for single-node failure recovery. Constructing such checksum-generator matrix for multiple-node failures is an intriguing open question. Throughout this section, we assume $p_r$ divides $n$ for simplicity, but results generalize to any $p_r$ and $n$.

**Construction 1** ($Q$-factor Checksum-Generator Matrix for Single Node Failure Recovery)**.** *If $gcd(\frac{n}{p_r}, p_r) = 1$, the following $\frac{n}{p_r} \times n$ checksum-generator matrix $G$ satisfies the restriction* (8)*, and guarantees single-node fault tolerance for out-of-node checksum storage:*

$$G = \begin{bmatrix} \lambda I_{\frac{n}{p_r}} & I_{\frac{n}{p_r}} & I_{\frac{n}{p_r}} & \cdots & I_{\frac{n}{p_r}} \end{bmatrix} \tag{14}$$

*where $\lambda = -\frac{1}{2}(p_r - 1)$*

It can be easily verified that the generator matrix in Construction 1 satisfies the restriction (8) and tolerates any single-node failure. Detailed proofs are given in [32, Appendix B-B]. This construction also satisfies the maximum-distance separable (MDS) condition, *i.e.,* it has the optimal number of checksums for single-node failures: $c = \frac{n}{p_r}$.

### C. Overhead Analysis

Finally, we analyze the overhead of the proposed coding strategy for $Q$-factor protection. We consider communication and computation cost formulated as an $\alpha$-$\beta$-$\gamma$ model [33], [34]:

$$T = \alpha C_1 + \beta C_2 + \gamma C_3 \tag{15}$$

where $C_1$ is the number of communication rounds, $C_2$ is the number of bytes communicated on the critical path, and $C_3$ is the number of floating point operations (flops). The $\alpha$ term models the communication latency, the $\beta$ term models the per-byte bandwidth, and $\gamma$ term is the cost per flop. Two types of overhead are considered: the total overhead of coding $T_{coding}$ and the overhead for recovering any single-node failure $T_f$. The coding overhead is modeled as:

$$T_{coding} = T_{enc} + T_{post} + T_{comp} \tag{16}$$

where $T_{enc}$, $T_{post}$ and $T_{comp}$ are the overhead for encoding, post-orthogonalization, and increased computation cost for QR factorization of the encoded matrix. We compare $T_{coding}$ with the cost $T_{QR}$ of QR factorization without coding for an $n \times n$ matrix using $p_r \times p_c$ grid of nodes.

**Theorem 3.** *For the MGS algorithm [19], applying a code given in Construction 1 for Q-factor protection has the following overhead:*

$$T_{coding} = O\left(\frac{1}{p_r} + \frac{1}{p_c}\right)\cdot T_{QR}, \quad T^{\alpha}_{coding} = O\left(\frac{1}{n}\right)\cdot T^{\alpha}_{QR},$$
$$T_f = O\left(\frac{1}{p_c}\right)\cdot T_{QR}.$$

Notice that while the total overhead scales as $O(1/p_r + 1/p_c)$ of the QR decomposition cost, the overhead in terms of communication latency (the $\alpha$ term) is much smaller, scaling as $O(1/n)$. Since $\alpha$ is often the dominating term in HPC systems, our proposed coding scheme could have very small overhead in real-world systems. Showing this through experiments would be an interesting future direction. The full proof of Theorem 3 is given in [32, Appendix D].

## V. Coded QR Decomposition – An Example

To illustrate our coding strategy for Q-factor and R-factor protection, we provide a small example of performing coded QR factorization on a $3 \times 3$ input matrix $A$ by using the checksum-generator matrices below:

$$G_v = \begin{bmatrix} -1 & 1 & 1 \end{bmatrix}, \; G_h = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}^T,$$

Note that $G_v$ satisfies the restriction (8). Now, consider:

$$A = \begin{bmatrix} 1 & -1 & 4 \\ 1 & 4 & -2 \\ 1 & 4 & 2 \end{bmatrix} \xrightarrow{\text{Encode}} \widetilde{A} = \begin{bmatrix} 1 & -1 & 4 & 4 \\ 1 & 4 & -2 & 3 \\ 1 & 4 & 2 & 7 \\ 1 & 9 & -4 & 6 \end{bmatrix} \tag{17}$$

The MGS algorithm is executed on $\widetilde{A}$ where $Q$-factor and $R$-factor are first initialized to $Q = \widetilde{A}$ and $R = 0$. The intermediate $Q^{(t)}$ and $R^{(t)}$ matrices are shown in Figure 2.

It is easy for a reader to check that at any iteration the last row of $Q$ (resp. last column of $R$) protects all other rows (resp. columns) via the the checksum relation by $G_v$ (resp. $G_h$):

$$Q = \begin{bmatrix} 1/2 & -1 & 4 & 4 \\ 1/2 & 4 & -2 & 3 \\ 1/2 & 4 & 2 & 7 \\ 1/2 & 9 & -4 & 6 \end{bmatrix} \quad R = \begin{bmatrix} 2 & 8 & 0 & 10 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

**Iteration 1**

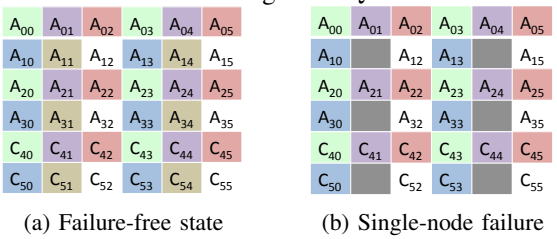
$$Q = \begin{bmatrix} 1/2 & -1/\sqrt{2} & 4 & 4 \\ 1/2 & 0 & -2 & 3 \\ 1/2 & 0 & 2 & 7 \\ 1/2 & 1/\sqrt{2} & -4 & 6 \end{bmatrix} \quad R = \begin{bmatrix} 2 & 8 & 0 & 10 \\ 0 & 5\sqrt{2} & -4\sqrt{2} & \sqrt{2} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

**Iteration 2**

$$Q = \begin{bmatrix} 1/2 & -1/\sqrt{2} & 0 & 4 \\ 1/2 & 0 & -1/\sqrt{2} & 3 \\ 1/2 & 0 & 1/\sqrt{2} & 7 \\ 1/2 & 1/\sqrt{2} & 0 & 6 \end{bmatrix} \quad R = \begin{bmatrix} 2 & 8 & 0 & 10 \\ 0 & 5\sqrt{2} & -4\sqrt{2} & \sqrt{2} \\ 0 & 0 & 2\sqrt{2} & 2\sqrt{2} \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

**Iteration 3**

Figure 2: $Q$ and $R$ matrices over the iterations for the example given in (17).




(a) Failure-free state (b) Single-node failure

Figure 3: In-node checksum storage for $p_r = 2, p_c = 3$. (a) The distribution of the original data is the same as Figure 1a, but checksum-blocks $C_{ij}$'s are also distributed over the original systematic processors. (b) Lost data-blocks are grayed out for the case where the brown node fails. Notice that some checksum blocks are also lost in this case.

- $Q$-factor protection: The last row is the sum of the $2^{nd}$ and $3^{rd}$ rows subtracting the $1^{st}$ row.
- $R$-factor protection: The last column is the sum of all the previous columns.

We proceed to show how the post-orthogonalization step works. At the end of iteration 3, we retrieve $A = Q_1 R$ where:

$$Q_1 = \begin{bmatrix} 1/2 & -1/\sqrt{2} & 0 \\ 1/2 & 0 & -1/\sqrt{2} \\ 1/2 & 0 & 1/\sqrt{2} \end{bmatrix} \text{ and } R = \begin{bmatrix} 2 & 8 & 0 \\ 0 & 5\sqrt{2} & -4\sqrt{2} \\ 0 & 0 & 2\sqrt{2} \end{bmatrix}$$

Following Theorem 2, we can compute $G_0$ and check the orthogonality of $(G_0 Q_1)$:

$$G_0 = \begin{bmatrix} 0 & 1 & 1 \\ 1 & -1 & 0 \\ 1 & 0 & -1 \end{bmatrix} \text{ and } G_0 Q_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1/\sqrt{2} & 1/\sqrt{2} \\ 0 & -1/\sqrt{2} & -1/\sqrt{2} \end{bmatrix}.$$

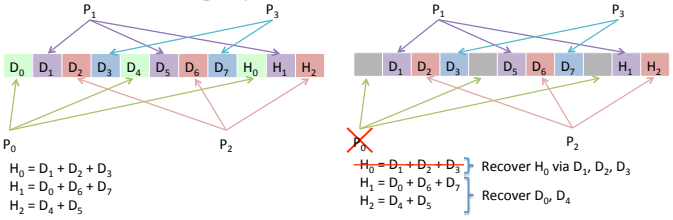
It is easy to see that $(G_0 Q_1)^T (G_0 Q_1) = I$.

## VI. Optimal In-node checksum Storage for Single-node Failure

So far in this paper, we examined coding strategies for the out-of-node checksum storage setting where we add additional nodes to store the coded data (checksums). This is a commonly used assumption in the coded computing literature. However, in the HPC literature, *in-node-checksum storage* was also considered [10], [35]. where the coded data is distributed to the existing nodes instead of introducing additional processors. This new setting could be more appealing in practice as it does not require additional processors [2], and alleviates the expense to make them protected [36], [37]. An example of in-node-checksum storage is depicted in Figure 3.

In this section, we prove the lower bound on the number of checksums required for in-node checksum storage setting for recovering from *any single failure*, and provide a checksum scheme that meets the lower bound. This improves the existing strategy for in-node checksum storage [10] by $\sim 2$x.

[2]In computations for HPC, algorithms are often designed for powers-of-two number of nodes for efficiency reasons. A coded computing strategy would be less desirable to practitioners if they have to change their algorithm to accommodate for additional checksum nodes.




(a) Failure-free state (b) Single-node failure (at $P_0$)

Figure 4: An example code for in-node checksum storage given in Theorem 5 for the case $L = 8, \rho = 4$. Here, $K = \left\lceil \frac{L}{\rho - 1} \right\rceil = 3$.

As checksums of each block-row are encoded with the same linear relation dictated by the checksum-generator matrix, it suffices to consider the checksum computation for one block-row. For simplicity, we consider a block-row of $L$ data blocks $D_0, D_1, ..., D_{L-1}$ distributed block-cyclically onto $\rho$ processers $P_0, P_1, ..., P_{\rho-1}$, and $K$ checksums $H_0, H_1, ..., H_{K-1}$ for recovery. If we let $f(i,j) =$ the index of the $i^{th}$ data block of processor $P_j$, then $f(i,j) = j + i\rho$. For convenience, we define $D_{f(i,j)} = 0$ if $f(i,j) \geq L$, i.e. $i$ exceeds the index of the last data point of processor $j$. We first prove a lower bound on the number of checksums $K$ for single failure recovery.

**Theorem 4.** *Under the in-node checksum storage setting, the minimum value of $K$ to tolerate a single node failure is:*

$$K \geq \left\lceil \frac{L}{\rho - 1} \right\rceil. \tag{18}$$

In the previous work by Bouteiller et al. [10], the $R$-protection under in-node checksum storage required $2\left\lceil \frac{L}{\rho} \right\rceil$ checksums, which is $\sim 2$x than the lower bound.

**Theorem 5.** *Under the in-node checksum storage setting, if $\rho$ divides $L$, the following checksum construction guarantees single-node failure tolerance and achieves optimal size of checksums $K = \left\lceil \frac{L}{\rho - 1} \right\rceil$:*

$$H_{l+v\rho} = \sum_{i=0}^{l-1} D_{f(l-1+v(\rho-1),i)} + \sum_{j=l+1}^{\rho-1} D_{f(l+v(\rho-1),j)} \tag{19}$$

*for $l \in [0, \rho)$ and $v \in [0, \left\lceil \frac{K}{\rho} \right\rceil)$ such that $l + v\rho < \left\lceil \frac{L}{\rho - 1} \right\rceil$.*

The proofs of Theorem 4 and Theorem 5 are given in [32, Appendix C-A]. Figure 4 illustrates the checksum scheme in Theorem 5 for $L = 8, \rho = 4, K = \left\lceil \frac{8}{4-1} \right\rceil = 3$.

We also propose a checksum-generator matrix for the $Q$-factor protection under the in-node checksum storage setting in [32, Appendix C-B][3].

[3]However, this does not meet the lower bound given in Theorem 4. We also include our thoughts on the gap from the optimality in the discussion.

## REFERENCES


[1] K. Lee, M. Lam, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran, "Speeding up distributed machine learning using codes," *IEEE Transactions on Information Theory*, 2017.

[2] S. Dutta, V. Cadambe, and P. Grover, "Short-dot: Computing large linear transforms distributedly using coded short dot products," in *Advances In Neural Information Processing Systems*, 2016.

[3] R. Tandon, Q. Lei, A. G. Dimakis, and N. Karampatziakis, "Gradient coding," *arXiv preprint arXiv:1612.03301*, 2016.

[4] Q. Yu, M. A. Maddah-Ali, and A. S. Avestimehr, "Polynomial codes: an optimal design for high-dimensional coded matrix multiplication," *arXiv preprint arXiv:1705.10464*, 2017.

[5] S. Dutta, M. Fahim, F. Haddadpour, H. Jeong, V. Cadambe, and P. Grover, "On the optimal recovery threshold of coded matrix multiplication," *IEEE Transactions on Information Theory*, 2019.

[6] E. Elnozahy, R. Bianchini, T. El-Ghazawi, A. Fox, F. Godfrey, A. Hoisie, K. McKinley, R. Melhem, J. Plank, and P. Ranganathan, "System resilience at extreme scale. Defense Advanced Research Project Agency (DARPA)," 2008.

[7] Z. Chen and J. Dongarra, "Algorithm-based checkpoint-free fault tolerance for parallel matrix computations on volatile resources," *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, 2006.

[8] G. Bosilca, R. Delmas, J. Dongarra, and J. Langou, "Algorithmic Based Fault Tolerance Applied to High Performance Computing," 2008.

[9] 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," *The International Journal of High Performance Computing Applications*, vol. 29, no. 4, 2015.

[10] P. Du, A. Bouteiller, G. Bosilca, T. Herault, and J. Dongarra, "Algorithm-based fault tolerance for dense matrix factorizations," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 225–234. [Online]. Available: https://doi.org/10.1145/2145816.2145845

[11] P. Du, P. Luszczek, S. Tomov, and J. Dongarra, "Soft error resilient qr factorization for hybrid system with gpgpu," in *Proceedings of the Second Workshop on Scalable Algorithms for Large-scale Systems*, ser. ScalA '11. New York, NY, USA: ACM, 2011.

[12] P. Wu and Z. Chen, "Ft-scalapack: correcting soft errors on-line for scalapack cholesky, qr, and lu factorization routines," in *HPDC*, 2014.

[13] F. T. Luk and H. Park, "An analysis of algorithm-based fault tolerance techniques," *Journal of Parallel and Distributed Computing*, vol. 5, no. 2, 1988.

[14] H. Park, "On multiple error detection in matrix triangularizations using checksum methods," *Journal of Parallel and Distributed Computing*, vol. 14, no. 1, 1992.

[15] P. Fitzpatrick and C. C. Murphy, "Fault tolerant matrix triangularization and solution of linear systems of equations," in *[1992] Proceedings of the International Conference on Application Specific Array Processors*, Aug 1992.

[16] T. Davies, C. Karlsson, H. Liu, C. Ding, and Z. Chen, "High performance linpack benchmark: A fault tolerant implementation without checkpointing," in *Proceedings of the International Conference on Supercomputing*, ser. ICS '11. New York, NY, USA: ACM, 2011.

[17] O. Maslennikow, J. Kaniewski, and R. Wyrzykowski, "Fault tolerant qr-decomposition algorithm and its parallel implementation," in *Euro-Par'98 Parallel Processing*, D. Pritchard and J. Reeve, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998.

[18] K. Swirydowicz, J. Langou, S. Ananthan, U. Yang, and S. Thomas, "Low synchronization gmres algorithms," 2018.

[19] G. Rünger and M. Schwind, "Comparison of different parallel modified gram-schmidt algorithms," vol. 3648, 08 2005.

[20] S. Oliveira, L. Borges, M. Holzrichter, and T. Soma, "Analysis of different partitioning schemes for parallel gram-schmidt algorithms," *Parallel Algorithms Appl.*, vol. 14, 04 2000.

[21] F. Rotella and I. Zambettakis, "Block householder transformation for parallel qr factorization," *Applied Mathematics Letters*, vol. 12, no. 4, pp. 29 – 34, 1999. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0893965999000282

[22] G. Ballard, J. Demmel, L. Grigori, M. Jacquelin, H. D. Nguyen, and E. Solomonik, "Reconstructing householder vectors from tall-skinny qr," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, May 2014.

[23] O. Egecioglu, "Givens and householder reductions for linear least squares on a cluster of workstations," Santa Barbara, CA, USA, Tech. Rep., 1995.

[24] J. J. Dongarra, P. Luszczek, and A. Petitet, "The linpack benchmark: Past, present, and future," 2002.

[25] C. Ashcraft and R. G. Grimes, "Spooles: An object-oriented sparse matrix library," in *PPSC*, 1999.

[26] T. Sterling, M. Anderson, and M. Brodowicz, *High Performance Computing: Modern Systems and Practices*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017.

[27] G. H. Golub and C. F. van Loan, *Matrix Computations*, 4th ed. JHU Press, 2013. [Online]. Available: http://www.cs.cornell.edu/cv/GVL4/golubandvanloan.htm

[28] A. Benoit, A. Cavelan, Y. Robert, and H. Sun, "Assessing general-purpose algorithms to cope with fail-stop and silent errors," *ACM Transactions on Parallel Computing*, vol. 3, pp. 1–36, 07 2016.

[29] T. Herault and Y. Robert, *Fault-Tolerance Techniques for High-Performance Computing*, 01 2015.

[30] W. N. Gansterer, G. Niederbrucker, H. Straková, and S. S. Grotthoff, "Scalable and fault tolerant orthogonalization based on randomized distributed data aggregation," *Journal of Computational Science*, vol. 4, no. 6, 2013, scalable Algorithms for Large-Scale Systems Workshop (ScalA2011), Supercomputing 2011.

[31] G. Rünger and M. Schwind, "Comparison of different parallel modified gram-schmidt algorithms," vol. 3648, 08 2005.

[32] "Full version of the paper." [Online]. Available: http://www.andrew.cmu.edu/user/haewonj/documents/coded_qr.pdf

[33] E. Chan, M. Heimlich, A. Purkayastha, and R. van de Geijn, "Collective communication: Theory, practice, and experience: Research articles," *Concurr. Comput. : Pract. Exper.*, vol. 19, no. 13, Sep. 2007.

[34] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in mpich," *Int. J. High Perform. Comput. Appl.*, vol. 19, no. 1, Feb. 2005.

[35] V. Fèvre, T. Herault, Y. Robert, A. Bouteiller, A. Hori, G. Bosilca, and J. Dongarra, "Comparing the performance of rigid, moldable and grid-shaped applications on failure-prone hpc platforms," *Parallel Computing*, vol. 85, 02 2019.

[36] 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 *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Washington, DC, USA: IEEE Computer Society Press, 2012.

[37] A. Rezaei, H. Khetawat, O. Patil, F. Mueller, P. Hargrove, and E. Roman, *End-to-End Resilience for HPC Applications*, 05 2019, pp. 271–290.