Coded QR Decomposition

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.

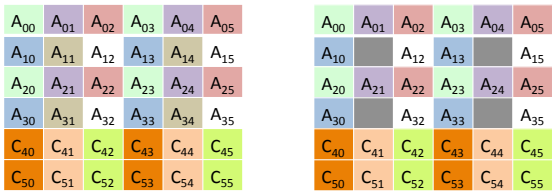
This work focuses on the single-node failure case as it is the most common scenario in HPC. Generalizing our code construction to multiple-node failure scenarios and beyond QR decomposition would be interesting to many intriguing future work. Considering a realistic model of data distribution and communication 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 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.

[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.

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: $Q = \begin{bmatrix}\boldsymbol{q}_1 & \boldsymbol{q}_2 & \cdots & \boldsymbol{q}_{n+d}\end{bmatrix}$ 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**
  $\boldsymbol{u_i} = \widetilde{\boldsymbol{a}}_i$
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 (1), 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 α-β-γ 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 α term models the communication latency, the β term models the per-byte bandwidth, and γ 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 α term) is much smaller, scaling as $O(1/n)$. Since α 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 3×3 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

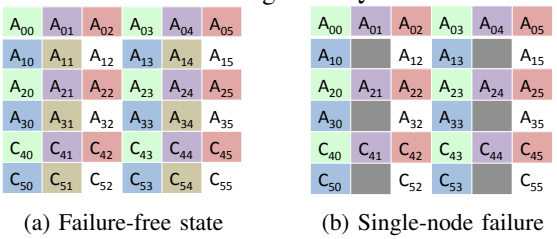
$$Q = \begin{bmatrix} 1/2 & -1/\sqrt{2} & 4 & 4 \\ 1/2 & 0 & -2 & 3 \\ 1/2 & 0 & 2 & 7 \\ 1/2 & 1/\sqrt{2} & -4 & 6 \end{bmatrix} \quad R = \begin{bmatrix} 2 & 8 & 0 & 10 \\ 0 & 5\sqrt{2} & -4\sqrt{2} & \sqrt{2} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

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 $(G_0 Q_1)$:

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

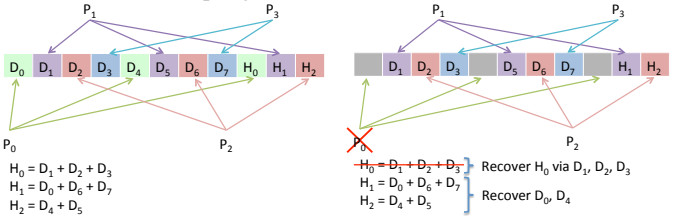
It is easy to see that $(G_0 Q_1)^T (G_0 Q_1) = I$.

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 ~ 2x.

[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 ρ 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 ~ 2x than the lower bound.

Theorem 5. *Under the in-node checksum storage setting, if ρ 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.

APPENDIX A
CHECKSUM-PRESERVATION

We now proceed to prove the checksum-preservation, equations (2) and (3) in Lemma 1, for sequential MGS (Appendix A-A) and parallel MGS (Appendix A-A) respectively.

A. Checksum-preservation for sequential MGS

We consider the columns of the encoded matrix $\widetilde{A}$:

$$\widetilde{A} = \begin{bmatrix} \widetilde{\boldsymbol{a}}_1 & \widetilde{\boldsymbol{a}}_2 & \cdots & \widetilde{\boldsymbol{a}}_{n+d} \end{bmatrix} \text{ where } \widetilde{\boldsymbol{a}}_i = \begin{bmatrix} \boldsymbol{a_i} \\ G_v \boldsymbol{a_i} \end{bmatrix}$$

and the columns of the final Q-factor $Q^{(t)}$ at the t^{th} iteration for $t = 1 \to T$:

$$Q^{(t)} = \begin{bmatrix} \boldsymbol{q}_1^{(t)} & \boldsymbol{q}_2^{(t)} & \cdots & \boldsymbol{q}_{n+d}^{(t)} \end{bmatrix}$$
$$Q_1^{(t)} = \begin{bmatrix} \boldsymbol{q}_{11}^{(t)} & \boldsymbol{q}_{12}^{(t)} & \cdots & \boldsymbol{q}_{1(n+d)}^{(t)} \end{bmatrix}$$
$$Q_2^{(t)} = \begin{bmatrix} \boldsymbol{q}_{21}^{(t)} & \boldsymbol{q}_{22}^{(t)} & \cdots & \boldsymbol{q}_{2(n+d)}^{(t)} \end{bmatrix}$$
$$\text{where } \boldsymbol{q}_i^{(t)} = \begin{bmatrix} \boldsymbol{q}_{1i}^{(t)} \\ \boldsymbol{q}_{2i}^{(t)} \end{bmatrix}.$$

We now present the MGS algorithm on $\widetilde{A}$:

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: $Q = \begin{bmatrix} \boldsymbol{q}_1 & \boldsymbol{q}_2 & \dots & \boldsymbol{q}_{n+d} \end{bmatrix}$ is $(n+c) \times (n+d)$ matrix , and $R = (r_{ij})$ is $(n+d) \times (n+d)$ matrix
Result: $\widetilde{A} = QR$, where $Q^T Q = I$

1 **for** $i = 1$ **to** n **do**
2 | $\boldsymbol{u_i} = \widetilde{\boldsymbol{a}}_i$
3 **end**
4 **for** $i = 1$ **to** $n + d$ **do**
5 | $r_{ii} = ||\boldsymbol{u_i}||_2$
6 | $\boldsymbol{q_i} = \boldsymbol{u_i} / r_{ii}$
7 | **for** $j = i + 1$ **to** $n + d$ **do**
8 | | $r_{ij} = \boldsymbol{q_i}^T \boldsymbol{u_j}$
9 | | $\boldsymbol{u_j} = \boldsymbol{u_j} - r_{ij} \boldsymbol{q_i}$
10 | **end**
11 **end**

Algorithm 2: Sequential MGS

In the algorithm, we use the temporary $(n+c) \times (n+d)$ matrix $U = \begin{bmatrix} \boldsymbol{u_1} & \boldsymbol{u_2} & \cdots & \boldsymbol{u_{n+d}} \end{bmatrix}$. For analysis, we consider its value $U^{(t)}$ at the end of each iteration $t = 1 \to T$. U is initialized to $U^{(0)} = \widetilde{A}$ by the loop from line 1 to line 3. We further consider:

$$U^{(t)} = \begin{bmatrix} U_1^{(t)} \\ U_2^{(t)} \end{bmatrix} = \begin{bmatrix} \boldsymbol{u_1} & \boldsymbol{u_2} & \cdots & \boldsymbol{u_{n+d}} \end{bmatrix}$$
(where $U_1^{(t)} : n \times (n+d)$, $U_2^{(t)} : c \times (n+d)$,
$$U_1^{(t)} = \begin{bmatrix} \boldsymbol{u}_{11}^{(t)} & \boldsymbol{u}_{12}^{(t)} & \cdots & \boldsymbol{u}_{1(n+d)}^{(t)} \end{bmatrix}$$
$$U_2^{(t)} = \begin{bmatrix} \boldsymbol{u}_{21}^{(t)} & \boldsymbol{u}_{22}^{(t)} & \cdots & \boldsymbol{u}_{2(n+d)}^{(t)} \end{bmatrix}$$
$$\text{where } \boldsymbol{u}_i^{(t)} = \begin{bmatrix} \boldsymbol{u}_{1i}^{(t)} \\ \boldsymbol{u}_{2i}^{(t)} \end{bmatrix}$$

$$R^{(t)} = \begin{bmatrix} R_1^{(t)} & R_2^{(t)} \end{bmatrix} = (r_{ij}^{(t)}) = \begin{bmatrix} \boldsymbol{r}_1^{(t)} \\ \vdots \\ \boldsymbol{r}_{n+c}^{(t)} \end{bmatrix}$$
$$R_1^{(t)} = \begin{bmatrix} \bar{\boldsymbol{r}}_{11}^{(t)} \\ \vdots \\ \bar{\boldsymbol{r}}_{(n+c)1}^{(t)} \end{bmatrix}, R_2^{(t)} = \begin{bmatrix} \bar{\boldsymbol{r}}_{12}^{(t)} \\ \vdots \\ \bar{\boldsymbol{r}}_{(n+c)2}^{(t)} \end{bmatrix}$$
$$\text{where } \boldsymbol{r}_i^{(t)} = \begin{bmatrix} \boldsymbol{r}_{i1}^{(t)} & \boldsymbol{r}_{i2}^{(t)} \end{bmatrix}$$

For the serial MGS (Algorithm **??**), there are $T = n + d$ iterations in the main loop from line 4 to line 11.

We prove by induction that for $t = 0 \to T$, we have $U_2^{(t)} = G_v U_1^{(t)}$, $Q_2^{(t)} = G_v Q_1^{(t)}$ and $R_2^{(t)} = R_1^{(t)} G_h$.

Base case: For $t = 0$:
As Q and U are initialized to $Q^{(0)} = U^{(0)} = \widetilde{A} = \begin{bmatrix} A & AG_h \\ G_v A & G_v A G_h \end{bmatrix}$, we have $Q_2^{(0)} = G_v Q_1^{(0)}$ and $U_2^{(0)} = G_v U_1^{(0)}$. As R is initialized to $R^{(0)} = 0$, we have $R_2^{(0)} = R_1^{(0)} G_h = 0$.

Inductive step: For the inductive step from $t-1$ to t ($t \geq 1$): Only the column $\boldsymbol{q}_t^{(t)}$ of $Q^{(t)}$ is updated on line 6:

$$\begin{aligned} \boldsymbol{q}_t^{(t)} = \boldsymbol{u}_t^{(t-1)} / r_{tt} &= \frac{1}{r_{tt}} \begin{bmatrix} \boldsymbol{u}_{1t}^{(t-1)} \\ \boldsymbol{u}_{2t}^{(t-1)} \end{bmatrix} \\ &= \frac{1}{r_{tt}} \begin{bmatrix} \boldsymbol{u}_{1t}^{(t-1)} \\ G_v \boldsymbol{u}_{1t}^{(t-1)} \end{bmatrix} \text{ (as } U_2^{(t-1)} = G_v U_1^{(t-1)}) \\ &= \begin{bmatrix} \boldsymbol{u}_{1t}^{(t-1)} / r_{tt} \\ G_v (\boldsymbol{u}_{1t}^{(t-1)} / r_{tt}) \end{bmatrix} = \begin{bmatrix} \boldsymbol{q}_{1t}^{(t)} \\ \boldsymbol{q}_{2t}^{(t)} \end{bmatrix} \end{aligned}$$

We thus obtain: $\boldsymbol{q}_{2t}^{(t)} = G_v \boldsymbol{q}_{1t}^{(t)}$. Other columns of Q remain the same: $\boldsymbol{q}_j^{(t)} = \boldsymbol{q}_j^{(t-1)}$, so $\boldsymbol{q}_{2j}^{(t)} = G \boldsymbol{q}_{1j}^{(t)}$ for all other $j \neq t$. We conclude that $Q_2^{(t)} = G_v Q_1^{(t)}$.

For the matrix U, there are $n - t$ columns $\boldsymbol{u_j}$ updated ($j = t + 1 \to n$) from line 7 to line 9:

$$\begin{aligned} \boldsymbol{u}_j^{(t)} = \boldsymbol{u}_j^{(t-1)} - r_{tj} \boldsymbol{q}_j^{(t)} &= \begin{bmatrix} \boldsymbol{u}_{1j}^{(t-1)} \\ \boldsymbol{u}_{2j}^{(t-1)} \end{bmatrix} - r_{tj} \begin{bmatrix} \boldsymbol{q}_{1j}^{(t)} \\ \boldsymbol{q}_{2j}^{(t)} \end{bmatrix} \\ &= \begin{bmatrix} \boldsymbol{u}_{1j}^{(t-1)} \\ G_v \boldsymbol{u}_{1j}^{(t-1)} \end{bmatrix} - r_{tj} \begin{bmatrix} \boldsymbol{q}_{1j}^{(t)} \\ G_v \boldsymbol{q}_{1j}^{(t)} \end{bmatrix} \\ &\text{(as } U_2^{(t-1)} = G_v U_1^{(t-1)} \text{ and } Q_2^{(t)} = G_v Q_1^{(t)}) \\ &= \begin{bmatrix} \boldsymbol{u}_{1j}^{(t-1)} - r_{tj} \boldsymbol{q}_{1j}^{(t)} \\ G_v (\boldsymbol{u}_{1j}^{(t-1)} - r_{tj} \boldsymbol{q}_{1j}^{(t)}) \end{bmatrix} = \begin{bmatrix} \boldsymbol{u}_{1j}^{(t)} \\ \boldsymbol{u}_{2j}^{(t)} \end{bmatrix} \end{aligned}$$

We thus obtain: $\boldsymbol{u}_{2t}^{(t)} = G_v \boldsymbol{u}_{1t}^{(t)}$. Other columns of U remain the same: $\boldsymbol{u}_j^{(t)} = \boldsymbol{u}_j^{(t-1)}$, so $\boldsymbol{u}_{2j}^{(t)} = G_v \boldsymbol{u}_{1j}^{(t)}$ for all other $j \leq t$. We conclude that $U_2^{(t)} = G_v U_1^{(t)}$.

Only the row $\boldsymbol{r}_t^{(t)}$ of $R^{(t)}$ is updated on line 8.

Observation 1. *At the end of iteration t, we have:*

$$\boldsymbol{r}_t^{(t)} = \boldsymbol{q}_t^T U^{(t)} = \boldsymbol{q}_t^T \widetilde{A} \tag{20}$$

Note that here $\boldsymbol{q_t}$ refers to $\boldsymbol{q_t}^{(t)}$. However, as the value of $\boldsymbol{q_t}$ is last updated in the t^{th} iteration and becomes the final value afterward, we neglect the superscript (t) for simplicity.

Proof. We first prove that $\boldsymbol{r_t}^{(t)} = \boldsymbol{q_t}^T U^{(t)}$. By line 5 and 8, $r_{tj}^{(t)} = \boldsymbol{q_t}^T \boldsymbol{u_j}^{(t)}$ for $j \geq t$, so it is left to show that $r_{tj}^{(t)} = \boldsymbol{q_t}^T \boldsymbol{u_j}^{(t)}$ also for $j < t$.

We note that for $j < t$:

- $r_{tj}^{(t)} = 0$ (as $r_{tj}^{(t)}$ is not updated by the algorithm and thus maintains the initial value 0).
- $\boldsymbol{q_j}^{(j)} = \boldsymbol{u_j}^{(j)}/r_{jj}$, updated on line 8 in the previous j^{th} iteration.
- $\boldsymbol{q_j}^{(t)} = \boldsymbol{q_j}^{(j)} = \boldsymbol{q_j}$, as $\boldsymbol{q_j}^{(j)}$ is last updated in the j^{th} iteration, and becomes the final value of $\boldsymbol{q_j}$.
- $\boldsymbol{u_j}^{(t)} = \boldsymbol{u_j}^{(j)} = \boldsymbol{u_j}$, as $\boldsymbol{u_j}^{(j)}$ is last updated in the j^{th} iteration, and becomes the final value of $\boldsymbol{u_j}$.

For $j < t$, we thus obtain that $\boldsymbol{q_t}^T \boldsymbol{u_j}^{(t)} = r_{jj} \boldsymbol{q_t}^T \boldsymbol{q_j}$. As $Q = \begin{bmatrix} \boldsymbol{q_1} & \boldsymbol{q_2} & \cdots & \boldsymbol{q_{n+d}} \end{bmatrix}$ is orthogonal, we have $I = Q^T Q = (\boldsymbol{q_i}^T \boldsymbol{q_j})_{ij}$, implying $\boldsymbol{q_t}^T \boldsymbol{q_j} = 0$. Therefore, $\boldsymbol{q_t}^T \boldsymbol{u_j}^{(t)} = 0 = r_{tj}^{(t)}$.

To complete the proof of (20), we proceed to prove that $\boldsymbol{q_t}^T U^{(t)} = \boldsymbol{q_t}^T \widetilde{A}$. This is equivalent to proving that $\boldsymbol{q_t}^T \boldsymbol{u_i}^{(t)} = \boldsymbol{q_t}^T \widetilde{\boldsymbol{a}}_{\boldsymbol{i}}$ for any $i \in [1, n+d]$. Each $\boldsymbol{u_i}^{(t)}$ is only updated on line 9 in the first i iterations, and thus formulated as:

$$\begin{aligned} \boldsymbol{u_i}^{(t)} &= \widetilde{\boldsymbol{a}}_{\boldsymbol{i}} - \sum_{l=1}^{\min(i,t)-1} r_{lj} \boldsymbol{q_l}^{(l)} \\ &= \widetilde{\boldsymbol{a}}_{\boldsymbol{i}} - \sum_{l=1}^{\min(i,t)-1} r_{lj} \boldsymbol{q_l} \\ &\quad \text{(as } \boldsymbol{q_l}^{(l)} \text{ is last updated in the } l^{th} \text{ iteration.)} \end{aligned}$$

Left multiplying $\boldsymbol{q_t}^T$, we obtain:

$$\begin{aligned} \boldsymbol{q_t}^T \boldsymbol{u_i}^{(t)} &= \boldsymbol{q_t}^T \widetilde{\boldsymbol{a}}_{\boldsymbol{i}} - \sum_{l=1}^{\min(i,t)-1} r_{lj} \boldsymbol{q_t}^T \boldsymbol{q_l} \\ &= \boldsymbol{q_t}^T \widetilde{\boldsymbol{a}}_{\boldsymbol{i}}, \end{aligned}$$

where the last line holds because $I = Q^T Q = (\boldsymbol{q_i}^T \boldsymbol{q_j})_{ij}$, implying $\boldsymbol{q_t}^T \boldsymbol{q_j} = 0$ for any $j \neq t$, and $l < \min(i,t) \leq t$. □

Back to our main proof, we now have:

$$\begin{aligned} \boldsymbol{r_t}^{(t)} &= \boldsymbol{q_t}^T \widetilde{A} = \boldsymbol{q_t}^T \begin{bmatrix} A & AG_h \\ G_v A & G_v A G_h \end{bmatrix} \\ &= \begin{bmatrix} \boldsymbol{q_t}^T \begin{bmatrix} A \\ G_v A \end{bmatrix} & \boldsymbol{q_t}^T \begin{bmatrix} A \\ G_v A \end{bmatrix} G_h \end{bmatrix} \\ &= \begin{bmatrix} \boldsymbol{r_{t1}}^{(t)} & \boldsymbol{r_{t2}}^{(t)} \end{bmatrix} \\ \Longrightarrow & \begin{cases} \boldsymbol{r_{t1}}^{(t)} = \boldsymbol{q_t}^T \begin{bmatrix} A \\ G_v A \end{bmatrix} \\ \boldsymbol{r_{t2}}^{(t)} = \boldsymbol{q_t}^T \begin{bmatrix} A \\ G_v A \end{bmatrix} G_h \end{cases} \end{aligned}$$

We thus obtain: $\boldsymbol{r_{t2}}^{(t)} = \boldsymbol{r_{t1}}^{(t)} G_h$. Other rows of R remain the same: $\boldsymbol{r_j}^{(t)} = \boldsymbol{r_j}^{(t-1)}$, so $\boldsymbol{r_{j2}}^{(t)} = \boldsymbol{r_{j1}}^{(t)} G_h$ for all other $j \neq t$. We conclude that $R_2^{(t)} = R_1^{(t)} G_h$.

B. Checksum-preservation for parallel MGS

We consider the columns of the encoded matrix $\widetilde{A}$:

$$\widetilde{A} = \begin{bmatrix} \widetilde{\boldsymbol{a}}_{\boldsymbol{1}} & \widetilde{\boldsymbol{a}}_{\boldsymbol{2}} & \cdots & \widetilde{\boldsymbol{a}}_{\boldsymbol{n+d}} \end{bmatrix} \text{ where } \widetilde{\boldsymbol{a}}_{\boldsymbol{i}} = \begin{bmatrix} \boldsymbol{a_i} \\ G_v \boldsymbol{a_i} \end{bmatrix}$$

and the columns of the final Q-factor $Q^{(t)}$ at the t^{th} iteration for $t = 1 \to T$:

$$\begin{aligned} Q^{(t)} &= \begin{bmatrix} \boldsymbol{q_1}^{(t)} & \boldsymbol{q_2}^{(t)} & \cdots & \boldsymbol{q_{n+d}}^{(t)} \end{bmatrix} \\ Q_1^{(t)} &= \begin{bmatrix} \boldsymbol{q_{11}}^{(t)} & \boldsymbol{q_{12}}^{(t)} & \cdots & \boldsymbol{q_{1(n+d)}}^{(t)} \end{bmatrix} \\ Q_2^{(t)} &= \begin{bmatrix} \boldsymbol{q_{21}}^{(t)} & \boldsymbol{q_{22}}^{(t)} & \cdots & \boldsymbol{q_{2(n+d)}}^{(t)} \end{bmatrix} \\ &\text{where } \boldsymbol{q_i}^{(t)} = \begin{bmatrix} \boldsymbol{q_{1i}}^{(t)} \\ \boldsymbol{q_{2i}}^{(t)} \end{bmatrix}. \end{aligned}$$

We now present the BMGS algorithm on $\widetilde{A}$:

Input: $\widetilde{A} = \begin{bmatrix} \widetilde{a}_1 & \widetilde{a}_2 & \cdots & \widetilde{a}_n \end{bmatrix}$ is $(m+c) \times n$ matrix
Output: $Q = \begin{bmatrix} q_1 & q_2 & \ldots & q_n \end{bmatrix}$ is $(m+c) \times n$ matrix , and $R = (r_{ij})$ is $n \times n$ matrix
Result: $\widetilde{A} = QR$, where $Q^T Q = I$

```
1  Q = Ã
2  for i = 1 to n Step b do
3      Q̄ = Q[:, i : i + b − 1]
4      MGS(Q̄, R[i : i + b − 1, i : i + b − 1])
5      Q[:, i : i + b − 1] = Q̄
6      if i < n − b then
7          R̄ = Q̄^T Q[:, i + b : n]
8          R[i : i + b − 1, i + b : n] = R̄
9          Q[:, i + b : n] − = Q̄R̄
10     end
11 end
```

Algorithm 3: BMGS

The BMGS is essentially the reformulation of MGS beneficial for parallelization. The parallel MGS, called PBMGS, is obtained by parallelizing certain steps in BMGS to adapt to the 2D block-cyclic distribution, where step b is set to the block size. As the computation and thus numerical output at each iteration remain the same through the parallelization, it suffices to prove the checksum-preservation for BMGS. Figure 5 illustrates the checksum-preservation for PBMGS.

In the algorithm, we use the temporary $(n+c) \times b$ matrix $\bar{Q}$. We further consider:

$$\bar{Q} = \begin{bmatrix} \bar{Q}_1 \\ \bar{Q}_2 \end{bmatrix} \text{ (where } \bar{Q}_1 : n \times b \text{ and } \bar{Q}_2 : c \times b)$$

$$R^{(t)} = \begin{bmatrix} R_1^{(t)} & R_2^{(t)} \end{bmatrix} = (r_{ij}^{(t)}) = \begin{bmatrix} \boldsymbol{r_1}^{(t)} \\ \vdots \\ \boldsymbol{r_{n+c}}^{(t)} \end{bmatrix}$$

$$R_1^{(t)} = \begin{bmatrix} \bar{\boldsymbol{r}}_{\mathbf{11}}^{(t)} \\ \vdots \\ \bar{\boldsymbol{r}}_{(\boldsymbol{n+c})\mathbf{1}}^{(t)} \end{bmatrix}, R_2^{(t)} = \begin{bmatrix} \bar{\boldsymbol{r}}_{\mathbf{12}}^{(t)} \\ \vdots \\ \bar{\boldsymbol{r}}_{(\boldsymbol{n+c})\mathbf{2}}^{(t)} \end{bmatrix}$$
where $\boldsymbol{r}_{\boldsymbol{i}}^{(t)} = \begin{bmatrix} \boldsymbol{r}_{\boldsymbol{i1}}^{(t)} & \boldsymbol{r}_{\boldsymbol{i2}}^{(t)} \end{bmatrix}$

We prove by induction that for $t = 0 \to T$, we have $Q_2^{(t)} = G_v Q_1^{(t)}$ and $R_2^{(t)} = R_1^{(t)} G_h$.

Base case: For $t = 0$:
As Q is initialized to $Q^{(0)} = \widetilde{A} = \begin{bmatrix} A & AG_h \\ G_v A & G_v A G_h \end{bmatrix}$, we have $Q_2^{(0)} = G_v Q_1^{(0)}$.

Inductive step: For the inductive step from $t-1$ to t ($t \geq 1$):

At the t^{th} iteration, we have $i = 1 + tb$ in the for-loop of BMGS.

The Q-factor is updated on line 5 for the b columns $Q[:, i : i+b-1]$ and on line 9 for the $n-i-b+1$ columns $Q[:, i+b : n]$.

The matrix $\bar{Q}$ is initialized to $\bar{Q} = Q^{(t-1)}[:, i : i+b-1]$ on line 3 and then serves as the data input for the MGS algorithm. As $Q_2^{(t-1)} = G_v Q_1^{(t-1)}$, we obtain that $\bar{Q}_2 = G_v \bar{Q}_1$ on line 3. By lemma **??**, the checksum relation $\bar{Q}_2 = G_v \bar{Q}_1$ of $\bar{Q}$ is maintained throughout and at the end of the computation of MGS on line 4.

For the b columns $Q[:, i : i+b-1]$ updated on line 5:

$$\begin{aligned} Q^{(t)}[:, i : i+b-1] = \bar{Q} = \begin{bmatrix} \bar{Q}_1 \\ \bar{Q}_2 \end{bmatrix} &= \begin{bmatrix} \bar{Q}_1 \\ G_v \bar{Q}_1 \end{bmatrix} \\ &= \begin{bmatrix} Q_1^{(t)}[:, i : i+b-1] \\ Q_2^{(t)}[:, i : i+b-1] \end{bmatrix} \end{aligned}$$

We thus obtain: $Q_2^{(t)}[:, i : i+b-1] = G_v Q_1^{(t)}[:, i : i+b-1]$.

For the $n-i-b+1$ columns $Q[:, i+b : n]$ updated on line 9:

$$\begin{aligned} Q^{(t)}[:, i+b : n] &= Q^{(t-1)}[:, i+b : n] - \bar{Q}\bar{R} \\ &= \begin{bmatrix} Q_1^{(t-1)}[:, i+b : n] \\ Q_2^{(t-1)}[:, i+b : n] \end{bmatrix} - \begin{bmatrix} \bar{Q}_1 \\ \bar{Q}_2 \end{bmatrix} \bar{R} \\ &= \begin{bmatrix} Q_1^{(t-1)}[:, i+b : n] \\ G Q_1^{(t-1)}[:, i+b : n] \end{bmatrix} - \begin{bmatrix} \bar{Q}_1 \\ G\bar{Q}_1 \end{bmatrix} \bar{R} \\ &\quad \text{(as } Q_2^{(t-1)} = G Q_1^{(t-1)} \text{ and } \bar{Q}_2 = G\bar{Q}_1\text{)} \\ &= \begin{bmatrix} Q_1^{(t-1)}[:, i+b : n] - \bar{Q}_1 \bar{R} \\ G(Q_1^{(t-1)}[:, i+b : n] - \bar{Q}_1 \bar{R}) \end{bmatrix} \\ &= \begin{bmatrix} Q_1^{(t)}[:, i+b : n] \\ Q_2^{(t)}[:, i+b : n] \end{bmatrix} \end{aligned}$$

We thus obtain: $Q_2^{(t)}[:, i+b : n] = G Q_1^{(t)}[:, i+b : n]$.

We have just proved that $q_{2j}^{(t)} = G q_{1j}^{(t)}$ for $j \in [i, n]$, where each of these columns are updated on either line 5 or line 9 of the BMGS algorithm. Other columns of Q remain the same: $q_j^{(t)} = q_j^{(t-1)}$, so $q_{2j}^{(t)} = G q_{1j}^{(t)}$ for all other $j < i = 1 + tb$. We conclude that $Q_2^{(t)} = G Q_1^{(t)}$.

The R-factor is updated on line 8 for the b rows $R[i : i+b-1, :]$.

Observation 2. *At the end of iteration t, we have:*

$$R^{(t)}[i : i+b-1, :] = \bar{Q}^T \widetilde{A} \tag{21}$$

Proof. The same reasoning as Observation 1's proof can be adapted. Here we would like to present an intuitive proof to show the validity of (21).

From the algorithm, we know that $R^{(t)}[i : i+b-1, :]$, and $Q^{(t)}[:, i : i+b-1] = \bar{Q}$ are last updated in this t^{th} iteration and thus also the final values:

$$\begin{aligned} R[i : i+b-1, :] &= R^{(t)}[i : i+b-1, :] \\ Q[:, i : i+b-1] &= Q^{(t)}[:, i : i+b-1] = \bar{Q} \end{aligned}$$

If the algorithm correctly computes the QR decomposition of $\widetilde{A}$, at the end we would have $\widetilde{A} = QR$, implying:

$$\begin{aligned} R &= Q^T \widetilde{A} \\ R[i : i+b-1, :] &= Q[:, i : i+b-1]^T \widetilde{A} \\ R^{(t)}[i : i+b-1, :] &= Q^{(t)}[:, i : i+b-1]^T \widetilde{A} \\ R^{(t)}[i : i+b-1, :] &= \bar{Q}^T \widetilde{A} \end{aligned}$$

□

Back to the main proof, we have:

$$\begin{aligned} R^{(t)}[i : i+b-1, :] = \bar{Q}^T \widetilde{A} &= \bar{Q}^T \begin{bmatrix} A & AG_h \\ G_v A & G_v A G_h \end{bmatrix} \\ &= \begin{bmatrix} \bar{Q}^T \begin{bmatrix} A \\ G_v A \end{bmatrix} & \bar{Q}^T \begin{bmatrix} A \\ G_v A \end{bmatrix} G_h \end{bmatrix} \\ &= \begin{bmatrix} R_1^{(t)}[i : i+b-1, :] & R_2^{(t)}[i : i+b-1, :] \end{bmatrix} \\ \implies &\begin{cases} R_1^{(t)}[i : i+b-1, :] = \bar{Q}^T \begin{bmatrix} A \\ G_v A \end{bmatrix} \\ R_2^{(t)}[i : i+b-1, :] = \bar{Q}^T \begin{bmatrix} A \\ G_v A \end{bmatrix} G_h \end{cases} \end{aligned}$$

We thus obtain: $\boldsymbol{r}_{\boldsymbol{j2}}^{(t)} = \boldsymbol{r}_{\boldsymbol{j1}}^{(t)} G_h$ for $j \in [i, i+b-1]$. Other rows of R remain the same: $\boldsymbol{r}_{\boldsymbol{j}}^{(t)} = \boldsymbol{r}_{\boldsymbol{j}}^{(t-1)}$, so $\boldsymbol{r}_{\boldsymbol{j2}}^{(t)} = \boldsymbol{r}_{\boldsymbol{j1}}^{(t)} G_h$ for all other $j \notin [i, i+b-1]$. We conclude that $R_2^{(t)} = R_1^{(t)} G_h$.

C. Corollary 1.1: final form of the QR decomposition on $\widetilde{A}$

By Lemma 1, we know that at the end of the algorithm, $Q_2^{(T)} = G_v Q_1^{(T)}$ and $R_2^{(T)} = R_1^{(T)} G_h$.

As the QR decomposition is retrieved as $\widetilde{A} = QR$, where $Q = Q^{(T)} = \begin{bmatrix} Q_1^{(T)} \\ Q_2^{(T)} \end{bmatrix} = \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix}$ and $R = R^{(T)} = \begin{bmatrix} R_1^{(T)} & R_2^{(T)} \end{bmatrix} = \begin{bmatrix} R_1 & R_2 \end{bmatrix}$, equation (4) directly holds:

$$\widetilde{A} = \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix} \begin{bmatrix} R_1 & R_2 \end{bmatrix} = \begin{bmatrix} Q_1 \\ G_v Q_1 \end{bmatrix} \begin{bmatrix} R_1 & R_1 G_h \end{bmatrix}$$

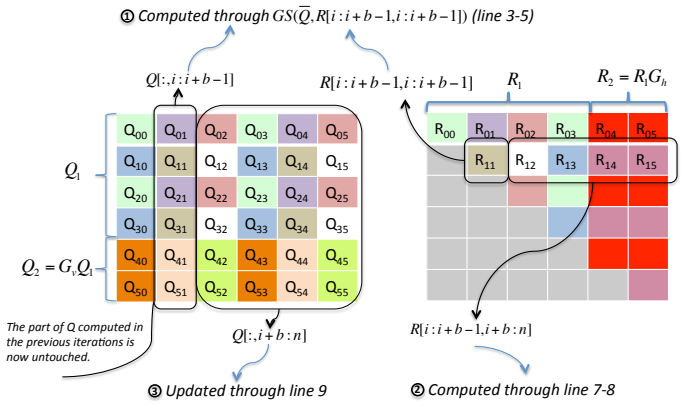


Figure 5: 2^{nd} iteration of PBMGS, i.e. parallelized BMGS (Algorithm 3), for $p_r = 2, p_c = 3$. Different colors represent different processors. (a) Six systematic processors (green, purple, red, blue, brown and white) own the original data blocks; three extra checksum-processors (orange, light orange, light green) own the vertical checksum-blocks Q_{ij}; and two extra checksum-processors (red, pink) own the horizontal checksum-blocks R_{ij} (b) The left matrix illustrates Q , and the right matrix illustrates R.

APPENDIX B
Q-FACTOR PROTECTION

A. Proof of Theorem 2

Proof of Theorem 2. Recall that given $G = \begin{bmatrix} G_1 & V \end{bmatrix}$, the $n \times n$ matrix G_0 is computed by:

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

The following Observation 3 is used in our proof later:

Observation 3. *Under the restriction* (8) *that* $G_1 = -\frac{1}{2}VV^T$, *the following equation holds:* $G_0^T G_0 = I + G^T G$.

Proof of Observation 3. From the restriction (8), we have $G_1^T = G_1 = \frac{-1}{2}VV^T$, and obtain that:

$$\begin{aligned} G_0^T G_0 &= \begin{bmatrix} I + G_1 & V \\ V^T & -I \end{bmatrix}^T \begin{bmatrix} I + G_1 & V \\ V^T & -I \end{bmatrix} \\ &= \begin{bmatrix} (I+G_1)^2 + VV^T & (I+G_1)V - V \\ V^T(I+G_1) - V^T & V^TV + I \end{bmatrix} \\ &= \begin{bmatrix} I + G_1^2 & G_1 V \\ V^T G_1 & V^T V + I \end{bmatrix} \\ &= I + \begin{bmatrix} G_1^2 & G_1 V \\ V^T G_1 & V^T V \end{bmatrix} \\ &= I + G^T G \end{aligned}$$

□

At the end of QR decomposition of encoded $\widetilde{A}$ in equation (4), the left factor $\begin{bmatrix} Q_1 \\ GQ_1 \end{bmatrix}$ is orthogonal :

$$I = \begin{bmatrix} Q_1 \\ GQ_1 \end{bmatrix}^T \begin{bmatrix} Q_1 \\ GQ_1 \end{bmatrix}$$

$$\begin{aligned} &= Q_1^T Q_1 + Q_1^T G^T G Q_1 \\ &= Q_1^T (I + G^T G) Q_1 \\ &= Q_1^T G_0^T G_0 Q_1 \text{ (by Observation 3)} \\ &= (G_0 Q_1)^T (G_0 Q_1) \end{aligned}$$

Therefore,$G_0 Q_1$ is orthogonal .

The following Observation 4 will be useful for computing the determinant of G_0.

Observation 4. *Consider matrices* A, B, C, D *of size* $n \times n, n \times m, m \times n, m \times m$ *respectively. If* D *is invertible then:*

$$det \begin{bmatrix} A & B \\ C & D \end{bmatrix} = det(D) det(A - BD^{-1}C)$$

We compute the determinant of G_0:

$$\begin{aligned} det(G_0) &= det \begin{bmatrix} I_c + G_1 & V \\ V^T & -I_{m-c} \end{bmatrix} \\ &= det(-I_{m-c}) det((I + G_1) - V(-I)^{-1}V^T) \\ &= (-1)^{m-c} det(I + G_1 + VV^T) \end{aligned}$$

where the second line follows Observation 4.

Under our setting $G_1 = \frac{-1}{2}VV^T$, the determinant of G_0 becomes:

$$det(G_0) = (-1)^{m-c} det(I + \frac{1}{2}VV^T)$$

Taking any non-zero column vector $\boldsymbol{z}$, we have $\boldsymbol{z}^T(I + \frac{1}{2}VV^T)\boldsymbol{z} = \boldsymbol{z}^T\boldsymbol{z} + \frac{1}{2}(V^T\boldsymbol{z})^T V^T\boldsymbol{z} \geq \boldsymbol{z}^T\boldsymbol{z} > 0$. This means that $I + \frac{1}{2}VV^T$ is positive definite, and $det(I + \frac{1}{2}VV^T) \neq 0$. Therefore, $det(G_0) \neq 0$ and G_0 is invertible.

□

B. Proof of Construction 1

Proof of Construction 1 . We first prove that G satisfies the restriction (8). Breaking down $G = \begin{bmatrix} G_1 & V \end{bmatrix}$, where G_1 is $\frac{n}{p_r} \times \frac{n}{p_r}$ and V is $\frac{n}{p_r} \times (n - \frac{n}{p_r})$, we obtain that:

$$\begin{aligned} G_1 &= \lambda I_{\frac{n}{p_r}} \\ V &= [\underbrace{I_{\frac{n}{p_r}} \quad I_{\frac{n}{p_r}} \quad \cdots \quad I_{\frac{n}{p_r}}}_{p_r - 1 \text{ identities } I_{\frac{n}{p_r}}}] \end{aligned}$$

We further check that:

$$-\frac{1}{2}VV^T = -\frac{1}{2}\begin{bmatrix} I_{\frac{n}{p_r}} & I_{\frac{n}{p_r}} & \cdots & I_{\frac{n}{p_r}} \end{bmatrix} \begin{bmatrix} I_{\frac{n}{p_r}} \\ I_{\frac{n}{p_r}} \\ \vdots \\ I_{\frac{n}{p_r}} \end{bmatrix}$$

$$\begin{aligned} &= -\frac{1}{2}\sum^{p_r - 1} I_{\frac{n}{p_r}} = -\frac{1}{2}(p_r - 1) I_{\frac{n}{p_r}} \\ &= \lambda I_{\frac{n}{p_r}} = G_1 \end{aligned}$$

Hence, G satisfies the restriction (8).

We are left to prove that G guarantees single-node failure tolerance.

In our system model (Section II-B), the input $n \times n$ matrix A is divided into $N \times N$ blocks A_{ij} $(i, j \in [0, N-1])$, each of which is of size $b \times b$ and owned by processor $\Pi(i, j)$. Under the out-of-node checksum storage, the Figures 1a and

1b respectively illustrate the distribution of processors in the case of the failure-free state and the single-node failure. The p_c checksum processors are $V_0, V_1, ..., V_{p_c-1}$ where V_j owns the $j^{th}, (j+p_c)^{th}, (j+2p_c)^{th}, ...$ block-columns (i.e. column of $b \times b$ blocks).

As the checksum-generator matrix G is left-multiplied with each block-column to generate checksums, it suffices to consider and prove the single-node fault tolerance for one block-column without loss of generality. In particular, for any j^{th} block-column:

$$\begin{bmatrix} C_{0,j} \\ C_{1,j} \\ \vdots \\ C_{\frac{n}{p_r}-1,j} \end{bmatrix} = G \begin{bmatrix} A_{0,j} \\ A_{1,j} \\ \vdots \\ A_{N-1,j} \end{bmatrix} \tag{22}$$

where each A_{ij} is owned by processor $\Pi(i,j)$ and C_{ij} is owned by the checksum processor $V_{(j \mod p_c)}$. Note that $\Pi(i,j) = \Pi(i \mod p_r, j)$, so there are p_r processors vertically covering all the data blocks.

To simplify the notations, we will omit the column index j and use $\begin{bmatrix} A_0 & A_1 & \cdots & A_{N-1} \end{bmatrix}^T$ to represent the block-column, and $\begin{bmatrix} C_0 & C_1 & \cdots & C_{\frac{n}{p_r}-1} \end{bmatrix}^T$ to represent the checksum matrix. The new A_i is owned by processor $P_{(i \mod p_r)}$, while all the checksums are owned by one checksum processor. Now, (22) is written as:

$$\begin{bmatrix} C_0 \\ C_1 \\ \vdots \\ C_{\frac{n}{p_r}-1} \end{bmatrix} = G \begin{bmatrix} A_0 \\ A_1 \\ \vdots \\ A_{N-1} \end{bmatrix} =$$

$$\begin{bmatrix} \lambda I_b & & & I_b & & & \cdots & I_b & \\ & \ddots & & & \ddots & & \cdots & & \ddots \\ & & \lambda I_b & & & I_b & \cdots & & I_b \end{bmatrix} \begin{bmatrix} A_0 \\ A_1 \\ \vdots \\ A_{N-1} \end{bmatrix}$$

$$\underbrace{\qquad}_{\frac{n}{p_r} \text{ identities } \lambda I_b} \quad \underbrace{\qquad}_{\frac{n}{p_r} \text{ identities } I_b} \quad \underbrace{\qquad}_{\frac{n}{p_r} \text{ identities } I_b}$$

$$= \begin{bmatrix} \lambda A_0 + A_{\frac{n}{p_r}} + A_{\frac{2M}{p_r}} + ... + A_{\frac{(p_r-1)M}{p_r}} \\ \vdots \\ \lambda A_i + A_{i+\frac{n}{p_r}} + A_{i+\frac{2M}{p_r}} + ... + A_{i+\frac{(p_r-1)M}{p_r}} \\ \vdots \end{bmatrix}$$

For the representation of G in the second line, we note that $n = bN$. The third line follows as A_i is of size $b \times b$.

Thus C_i ($i \in [0, \frac{n}{p_r} - 1]$) can be written as:

$$C_i = \lambda A_i + \sum_{j=1}^{p_r-1} A_{i+j\frac{n}{p_r}}$$

Observation 5. *For any $i \in [0, \frac{n}{p_r} - 1]$, $t \in [0, p_r - 1]$, processor P_t contributes exactly one data block to the checksum C_i. Moreover, the $\frac{n}{p_r}$ checksums cover all the N data blocks.*

Proof of Observation 5. Firstly, we show that any two data blocks in the computation of C_i are held by two different processors. In particular, we consider any two data blocks $A_{i+x\frac{n}{p_r}}$ and $A_{i+y\frac{n}{p_r}}$ where $x \neq y$ and $x, y \in [0, p_r - 1]$.

$A_{i+x\frac{n}{p_r}}$ and $A_{i+y\frac{n}{p_r}}$ are respectively owned by processor $P_{(i+x\frac{n}{p_r}) \mod p_r}$ and $P_{(i+y\frac{n}{p_r}) \mod p_r}$. It is left to check that $P_{(i+x\frac{n}{p_r}) \mod p_r} \neq P_{(i+y\frac{n}{p_r}) \mod p_r}$:

$$\begin{aligned} & P_{(i+x\frac{n}{p_r}) \mod p_r} \neq P_{(i+y\frac{n}{p_r}) \mod p_r} \\ \iff & i + x\frac{n}{p_r} \not\equiv i + y\frac{n}{p_r} \mod p_r \\ \iff & (x-y)\frac{n}{p_r} \not\equiv 0 \mod p_r \end{aligned}$$

The last line is true as $gcd(\frac{n}{p_r}, p_r) = 1$ and $|x - y| \in (0, p_r)$. Therefore, any two data blocks in the computation of C_i are held by two different processors.

Since there are exactly p_r data blocks $A_{i+j\frac{n}{p_r}}$, for $j \in [0, p_r - 1]$, in the computation of C_i, each of the p_r processors contributes one data block to C_i.

Lastly, the $\frac{n}{p_r}$ checksums cover all the M data blocks as any data block A_x ($x \in [0, N-1]$) is protected by checksum $C_{x \mod \frac{n}{p_r}}$. □

Recovery Scheme: Upon the failure of processor P_t ($t \in [0, p_r - 1]$), we lose data blocks $A_t, A_{t+p_r}, ..., A_{t+(\frac{n}{p_r}-1)p_r}$. By Observation 5, each lost data block A_{t+jp_r} ($j \in [0, \frac{n}{p_r} - 1]$) can be recovered from checksum $C_{(t+jp_r) \mod \frac{n}{p_r}}$ and other constituting $p_r - 1$ data blocks, given that the checksum is unaffected by the failure of P_t under out-of-node checksum storage.

□

APPENDIX C
IN-NODE CHECKSUM STORAGE

A. Proofs of Theorem 4 and Theorem 5

Proof of Theorem 4. The following Observation 6 is used in our proof later:

Observation 6. *For integers L, K and ρ such that $K - \left\lceil \frac{K}{\rho} \right\rceil \geq \left\lceil \frac{L}{\rho} \right\rceil$, we have: $K \geq \left\lceil \frac{L}{\rho - 1} \right\rceil$.*

Proof of Observation 6. Suppose $K < \frac{L}{\rho-1} \implies K(\rho-1) < L \implies K < \frac{L+K}{\rho} \leq \left\lceil \frac{L}{\rho} \right\rceil + \left\lceil \frac{K}{\rho} \right\rceil$. Then $K - \left\lceil \frac{K}{\rho} \right\rceil < \left\lceil \frac{L}{\rho} \right\rceil$ (Contradiction).

Therefore, $K \geq \frac{L}{\rho-1}$. As K is an integer, $K \geq \left\lceil \frac{L}{\rho-1} \right\rceil$.

□

Back to our proof, under the block cyclic distribution, any processor P_t ($t \in [0, \rho - 1]$) :

- holds either $\left\lceil \frac{L}{\rho} \right\rceil$ data points or $\left\lceil \frac{L}{\rho} \right\rceil - 1$ data points
- holds either $\left\lceil \frac{K}{\rho} \right\rceil$ checksums or $\left\lceil \frac{K}{\rho} \right\rceil - 1$ checksums

We consider two cases:

Case 1. There exists a processor P_s such that P_s holds exactly $\left\lceil \frac{L}{\rho} \right\rceil$ data points and $\left\lceil \frac{K}{\rho} \right\rceil$ checksums.

If P_s fails, we will be left with $K - \left\lceil \frac{K}{\rho} \right\rceil$ available checksums to recover all the $\left\lceil \frac{L}{\rho} \right\rceil$ lost data points. From [Theorem 5.1, [10]], to tolerate the node failure of P_s we must have: $K - \left\lceil \frac{K}{\rho} \right\rceil \geq \left\lceil \frac{L}{\rho} \right\rceil$. Using Observation 6, we obtain $K \geq \left\lceil \frac{L}{\rho-1} \right\rceil$.

Case 2. There is no processor that holds at the same time $\left\lceil \frac{L}{\rho} \right\rceil$ data points and $\left\lceil \frac{K}{\rho} \right\rceil$ checksums.

We can easily check that under block cyclic distribution, processor P_0 holds $\left\lceil \frac{L}{\rho} \right\rceil$ data points. Then it must hold $\left\lceil \frac{K}{\rho} \right\rceil - 1$ checksums according to this case 2. If P_0 fails, we will be left with $K - (\left\lceil \frac{K}{\rho} \right\rceil - 1)$ available checksums to recover all the $\left\lceil \frac{L}{\rho} \right\rceil$ lost data points. From [Theorem 5.1, [10]], to tolerate the node failure of P_0 we must have: $K - (\left\lceil \frac{K}{\rho} \right\rceil - 1) \geq \left\lceil \frac{L}{\rho} \right\rceil$, or :

$$\left\lceil \frac{L}{\rho} \right\rceil + \left\lceil \frac{K}{\rho} \right\rceil - 1 \leq K \tag{23}$$

As no processor holds at the same time $\left\lceil \frac{L}{\rho} \right\rceil$ data points and $\left\lceil \frac{K}{\rho} \right\rceil$ checksums, any processor P_t holds $\leq \left\lceil \frac{L}{\rho} \right\rceil + \left\lceil \frac{K}{\rho} \right\rceil - 1$ values (here value refers to either data point or checksum).

Summing up over ρ processors and noticing that the ρ processors hold the total of $L + K$ values (L data points and K checksums):

$$\begin{aligned} L + K &\leq \rho(\left\lceil \frac{L}{\rho} \right\rceil + \left\lceil \frac{K}{\rho} \right\rceil - 1) \\ &\leq \rho \cdot K \text{ (by Equation (23))} \\ K &\geq \frac{L}{\rho - 1} \end{aligned}$$

As K is an integer, this means $K \geq \left\lceil \frac{L}{\rho-1} \right\rceil$.

□

Proof of Theorem 5. Under the setting of in-node checksum storage and the condition $\rho | L$, upon the failure of some processor P_t, we lose data points $D_{f(0,t)}, D_{f(1,t)}, D_{f(2,t)}, ..., D_{f(\frac{L}{\rho}-1,t)}$ and checksums $H_t, H_{t+\rho}, H_{t+2\rho}, ..., H_{t+v_0\rho}$, where v_0 is the largest integer such that $t + v_0\rho < \left\lceil \frac{L}{\rho-1} \right\rceil$. We now further describe the recovery scheme for all these lost checksums and data points.

Recovery of Checksums: By the checksum formula (19), any checksum $H_{t+v\rho}$ (with $v \in [0, v_0]$) is the sum of data points , each of which has the form $D_{f(x,j)}$ with $j \neq t$, and stored by the processor $P_j \neq P_t$. Therefore, $H_{t+v\rho}$ can be recovered.

Recovery of Data Points: We consider any lost data point $D_{f(i,t)}$ (with $i \in [0, \frac{L}{\rho} - 1]$) and write $i = i_1 + i_2(\rho - 1)$ ($i_1 \in [0, \rho - 2]$).

Before proceeding, we consider the following Observation 7 that would be helpful for our proof:

Observation 7. $1 + i_1 + i_2\rho < \left\lceil \frac{L}{\rho-1} \right\rceil$.

Proof of Observation 7.

$$\begin{aligned} 1 + i_1 + i_2\rho &= 1 + i + i_2 = 1 + i + \frac{i - i_1}{\rho - 1} \\ &< i + 1 + \frac{i+1}{\rho - 1} = (i+1)\frac{\rho}{\rho - 1} \\ &\leq \frac{L}{\rho}\frac{\rho}{\rho - 1} \leq \left\lceil \frac{L}{\rho - 1} \right\rceil \end{aligned}$$

□

Obseveration 7 facilitates the existence of the two checksums $H_{i_1+i_2\rho}$ and $H_{1+i_1+i_2\rho}$, used in our recovery schemes later, as their indices lie in the range $[0, \left\lceil \frac{L}{\rho-1} \right\rceil - 1]$.

There are two cases leading to two different recovery schemes.

Case 1: $i_1 < t$

By the checksum formula (19), we have:

$$H_{i_1+i_2\rho} = \sum_{j=0}^{i_1-1} D_{f(i-1,j)} + \sum_{j=i_1+1}^{\rho-1} D_{f(i,j)}$$

The checksum $H_{i_1+i_2\rho}$ is owned by processor P_{i_1} ($\neq P_t$) and thus available for usage. The term $D_{f(i,t)}$ appears in the sum $\sum_{j=i_1+1}^{\rho-1} D_{f(i,j)}$ (as $t \geq i_1 + 1$) and is the only data of processor P_t in the computation of checksum $H_{i_1+i_2\rho}$.

Then $D_{f(i,t)}$ can be recovered by:

$$D_{f(i,t)} = H_{i_1+i_2\rho} - \sum_{j=0}^{i_1-1} D_{f(i-1,j)} - \sum_{\substack{j=i_1+1 \\ j\neq t}}^{\rho-1} D_{f(i,j)}$$

Case 2: $i_1 \geq t$

By the checksum formula (19), we have:

$$H_{1+i_1+i_2\rho} = \sum_{j=0}^{i_1} D_{f(i,j)} + \sum_{j=i_1+2}^{\rho-1} D_{f(i+1,j)}$$

The checksum $H_{1+i_1+i_2\rho}$ is owned by processor P_{1+i_1} ($\neq P_t$) and thus available for usage. The term $D_{f(i,t)}$ appears in the sum $\sum_{j=0}^{i_1} D_{f(i,j)}$ (as $t \leq i_1$) and is the only data of processor P_t in the computation of checksum $H_{1+i_1+i_2\rho}$.

Then $D_{f(i,t)}$ can be recovered by:

$$D_{f(i,t)} = H_{1+i_1+i_2\rho} - \sum_{\substack{j=0 \\ j\neq t}}^{i_1} D_{f(i,j)} + \sum_{j=i_1+2}^{\rho-1} D_{f(i+1,j)}$$

Remark 1. *The recovery for data point $A_{f(i,t)}$ (of failed processor P_t) can be summarized as the following:*
If $(i \bmod (\rho - 1)) < t$, $D_{f(i,t)}$ can be recovered from the checksum $H_{i+\lfloor \frac{i}{\rho-1} \rfloor}$. Otherwise, $D_{f(i,t)}$ can be recovered from the checksum $H_{i+1+\lfloor \frac{i}{\rho-1} \rfloor}$.

□

B. Coded QR decomposition for in-node checksum storage

Construction 2 (Code Construction for In-node Checksum Storage)**.** *If $gcd(\frac{n}{p_r}, p_r) = 1$, the following $\frac{2n}{p_r} \times m$ generator matrix G satisfies the restriction* (8)*, and guarantees single-node fault tolerance for in-node checksum storage:*

$$G = \begin{bmatrix} \lambda I_{\frac{n}{p_r}} & \lambda I_{\frac{n}{p_r}} & I_{\frac{n}{p_r}} & \cdots & I_{\frac{n}{p_r}} \\ \lambda I_{\frac{n}{p_r}} & \lambda I_{\frac{n}{p_r}} & I_{\frac{n}{p_r}} & \cdots & I_{\frac{n}{p_r}} \end{bmatrix} \tag{24}$$

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

Remark 2 (Conjecture on the optimality of in-node checksum storage with restriction (8))**.** *We prove in Theorem 4 that the optimal size of checksums is $c = \left\lceil \frac{n}{p_r - 1} \right\rceil$. However, when the additional restriction* (8) *is required, the optimality remains an open question. The generator matrix in Construction 2 achieves the size of checksums $c = \frac{2n}{p_r}$, which has $\sim 2x$ gap from the lower bound. We conjecture that the optimality is still $c = \left\lceil \frac{n}{p_r - 1} \right\rceil$, motivated by an example given in Appendix C-C.*

Proof of Construction 2 . We first prove that G satisfies the restriction (8). Breaking down $G = \begin{bmatrix} G_1 & V \end{bmatrix}$, where G_1 is $\frac{2m}{p_r} \times \frac{2m}{p_r}$ and V is $\frac{2m}{p_r} \times (m - \frac{2m}{p_r})$, we obtain that:

$$G_1 = \begin{bmatrix} \lambda I_{\frac{n}{p_r}} & \lambda I_{\frac{n}{p_r}} \\ \lambda I_{\frac{n}{p_r}} & \lambda I_{\frac{n}{p_r}} \end{bmatrix}$$

$$V = \underbrace{\begin{bmatrix} I_{\frac{n}{p_r}} & I_{\frac{n}{p_r}} & \cdots & I_{\frac{n}{p_r}} \\ I_{\frac{n}{p_r}} & I_{\frac{n}{p_r}} & \cdots & I_{\frac{n}{p_r}} \end{bmatrix}}_{p_r - 2 \text{ identities } \begin{bmatrix} I_{\frac{n}{p_r}} \\ I_{\frac{n}{p_r}} \end{bmatrix}}$$

We further check that:

$$\begin{aligned} -\frac{1}{2} V V^T &= -\frac{1}{2} \begin{bmatrix} I_{\frac{n}{p_r}} & I_{\frac{n}{p_r}} & \cdots & I_{\frac{n}{p_r}} \\ I_{\frac{n}{p_r}} & I_{\frac{n}{p_r}} & \cdots & I_{\frac{n}{p_r}} \end{bmatrix} \begin{bmatrix} I_{\frac{n}{p_r}} & I_{\frac{n}{p_r}} \\ I_{\frac{n}{p_r}} & I_{\frac{n}{p_r}} \\ \vdots & \\ I_{\frac{n}{p_r}} & I_{\frac{n}{p_r}} \end{bmatrix} \\ &= \begin{bmatrix} -\frac{1}{2}\sum_{p_r-2} I_{\frac{n}{p_r}} & -\frac{1}{2}\sum_{p_r-2} I_{\frac{n}{p_r}} \\ -\frac{1}{2}\sum_{p_r-2} I_{\frac{n}{p_r}} & -\frac{1}{2}\sum_{p_r-2} I_{\frac{n}{p_r}} \end{bmatrix} \\ &= \begin{bmatrix} -\frac{1}{2}(p_r-2) I_{\frac{n}{p_r}} & -\frac{1}{2}(p_r-2) I_{\frac{n}{p_r}} \\ -\frac{1}{2}(p_r-2) I_{\frac{n}{p_r}} & -\frac{1}{2}(p_r-2) I_{\frac{n}{p_r}} \end{bmatrix} \\ &= G_1 \end{aligned}$$

Hence, G satisfies the restriction (8).

We are left to prove that G guarantees single-node failure tolerance.

Similar to Appendix B-B, we simplify the notations by using $\begin{bmatrix} A_0 & A_1 & \cdots & A_{N-1} \end{bmatrix}^T$ to represent a block-column, and $\begin{bmatrix} C_0 & C_1 & \cdots & C_{\frac{2M}{p_r}-1} \end{bmatrix}^T$ to represent a checksum matrix, all of which are distributed cyclically over p_r processors $P_0, P_1, ..., P_{p_r-1}$. A_i and C_i are owned by processor $P_{i \mod p_r}$. Under our simplifed notations, the checksum relation of one block-column is written as:

$$\begin{bmatrix} C_0 \\ C_1 \\ \vdots \\ C_{\frac{2M}{p_r}-1} \end{bmatrix} = \begin{bmatrix} \lambda I_{\frac{n}{p_r}} & \lambda I_{\frac{n}{p_r}} & I_{\frac{n}{p_r}} & \cdots & I_{\frac{n}{p_r}} \\ \lambda I_{\frac{n}{p_r}} & \lambda I_{\frac{n}{p_r}} & I_{\frac{n}{p_r}} & \cdots & I_{\frac{n}{p_r}} \end{bmatrix} \begin{bmatrix} A_0 \\ A_1 \\ \vdots \\ A_{N-1} \end{bmatrix} \tag{25}$$

Note that we can express $G = \begin{bmatrix} G_1 \\ G_1 \end{bmatrix}$ where $G_1 = \begin{bmatrix} \lambda I_{\frac{n}{p_r}} & \lambda I_{\frac{n}{p_r}} & I_{\frac{n}{p_r}} & \cdots & I_{\frac{n}{p_r}} \end{bmatrix}$.
From (25), we observe that:

$$\begin{bmatrix} C_0 \\ C_1 \\ \vdots \\ C_{\frac{n}{p_r}-1} \end{bmatrix} = \begin{bmatrix} C_{\frac{n}{p_r}} \\ C_{1+\frac{n}{p_r}} \\ \vdots \\ C_{\frac{2M}{p_r}-1} \end{bmatrix} = G_1 \begin{bmatrix} A_0 \\ A_1 \\ \vdots \\ A_{N-1} \end{bmatrix} \tag{26}$$

Remark 3. *The λ term in the checksum-generator matrix of either Construction 1 or Construction 2 only serves as some 'modifier' enforcing the checksum-generator matrix to satisfy the restriction* (8)*. If we ignore such λ, the checksum-generator matrix in Construction 1 and G_1 above share the same structure, and the coding strategy for this Construction 2 can be thought of as the duplication of checksums formulated in the previous Construction 1. Such duplication is illustrated in equation (26).*

By Remark 3, the derivation of the checksum computation in this Construction 2 is similar to that of Construction 1 (Appendix B-B). We thus omit the intermediate steps and just state the checksum formulation.
For any $i \in [0, \frac{n}{p_r} - 1]$, we have:

$$C_i = C_{i+\frac{n}{p_r}} = \lambda A_i + \lambda A_{i+\frac{n}{p_r}} + \sum_{j=2}^{p_r-1} A_{i+j\frac{n}{p_r}} \tag{27}$$

Observation 8. *The two checksums C_i and $C_{i+\frac{n}{p_r}}$ in the equation (27) are held by two different processors.*

Proof of Observation 8. Checksums C_i and $C_{i+\frac{n}{p_r}}$ are respectively owned by processor $P_{i \mod p_r}$ and $P_{(i+\frac{n}{p_r}) \mod p_r}$. It is left to verify that $P_{i \mod p_r} \neq P_{(i+\frac{n}{p_r}) \mod p_r}$.

$$\begin{aligned} & P_{i \mod p_r} \neq P_{(i+\frac{n}{p_r}) \mod p_r} \\ \iff & i \not\equiv i + \frac{n}{p_r} \mod p_r \\ \iff & \frac{n}{p_r} \not\equiv 0 \mod p_r \end{aligned}$$

The last line is true as $gcd(\frac{n}{p_r}, p_r) = 1$. □

Observation 9. *For any $i \in [0, \frac{n}{p_r} - 1]$, $t \in [0, p_r - 1]$, processor P_t contributes exactly one data block to the checksum C_i. Moreover, these former $\frac{n}{p_r}$ checksums cover all the M data blocks.*

Proof of Observation 9. By Remark 3, the proof is essentially simlar to that of Observation 5. □

Recovery Scheme: Upon the failure of processor P_t ($t \in [0, p_r - 1]$), we lose data blocks $A_t, A_{t+p_r}, ..., A_{t+(\frac{n}{p_r}-1)p_r}$. Let $x_{t,j} = (t + jp_r) \mod \frac{n}{p_r}$. By Observation 9 and equation (27), each lost data block A_{t+jp_r} ($j \in [0, \frac{n}{p_r} - 1]$) can be recovered from either $C_{x_{t,j}}$ or $C_{x_{t,j}+\frac{n}{p_r}}$. As $C_{x_{t,j}}$ and $C_{x_{t,j}+\frac{n}{p_r}}$ are stored by two different processors by Observation 8, the failure of processor P_t can only trigger the loss of at most one of these two checksums. Therefore, A_{t+jp_r} can be recovered from a non-failed checksum among $C_{x_{t,j}}$ and $C_{x_{t,j}+\frac{n}{p_r}}$.

□

C. Conjecture of Remark 2

Under the in-node checksum storage with restriction (8), we conjecture that the optimal size of checksums is $c = \left\lceil \frac{n}{p_r - 1} \right\rceil$. We consider the case $m = 6, p_r = 3$, and propose the example checksum-generator matrix with $c = \left\lceil \frac{6}{3-1} \right\rceil = 3$ checksum-blocks tolerant to single-node failure.

We denote the 6 data-blocks as $A_0, A_1, ..., A_5$ and the 3 checksum-blocks as C_0, C_1, C_2, all of which are distributed block-cyclically over 3 processors P_0, P_1, P_2. In particular, we have:

- P_0 owns A_0, A_3 and C_0.
- P_1 owns A_1, A_4 and C_1.
- P_2 owns A_2, A_5 and C_2.

Consider the checksum-generator matrix:

$$G = \begin{bmatrix} -\frac{1}{2}I & 0 & -\frac{1}{2}I & 0 & I & 0 \\ 0 & -I & -I & I & 0 & I \\ -\frac{1}{2}I & -I & -\frac{3}{2}I & I & I & I \end{bmatrix}$$

We first prove that G satisfies the restriction (8). Breaking down $G = \begin{bmatrix} G_1 & V \end{bmatrix}$, we obtain that:

$$G_1 = \begin{bmatrix} -\frac{1}{2}I & 0 & -\frac{1}{2}I \\ 0 & -I & -I \\ -\frac{1}{2}I & -I & -\frac{3}{2}I \end{bmatrix}$$

$$V = \begin{bmatrix} 0 & I & 0 \\ I & 0 & I \\ I & I & I \end{bmatrix}$$

It can be verified that:

$$-\frac{1}{2}VV^T = -\frac{1}{2}\begin{bmatrix} 0 & I & 0 \\ I & 0 & I \\ I & I & I \end{bmatrix}\begin{bmatrix} 0 & I & I \\ I & 0 & I \\ 0 & I & I \end{bmatrix} = G_1$$

We now check the single node fault-tolerance of G.

$$\begin{bmatrix} C_0 \\ C_1 \\ C_2 \end{bmatrix} = G \begin{bmatrix} A_0 \\ A_1 \\ \vdots \\ A_5 \end{bmatrix} = \begin{bmatrix} -\frac{1}{2}A_0 - \frac{1}{2}A_2 + A_4 \\ -A_1 - A_2 + A3 + A5 \\ -\frac{1}{2}A_0 - A_1 - \frac{3}{2}A_2 + A3 + A_4 + A5 \end{bmatrix}$$

Direct substitution gives $C_2 = C_0 + C_1$. Regardless of whichever node failure, we thus can always recover all the checksum-blocks C_0, C_1 and C_2 by using the two remaining non-failed processors.

Then:

- If P_0 fails, A_0 and A_3 can be recovered respectively from C_0 and C_1.
- If P_1 fails, A_1 and A_4 can be recovered respectively from C_1 and C_0.
- If P_2 fails, we first recover A_2 from C_0, and then recover A_5 from $A_5 = C_1 - (-A_1 - A_2 + A_3)$.

APPENDIX D
OVERHEAD ANALYSIS

A. MPI collective communication operations

The implementations in practice of parallel QR decomposition and our coding strategy depend on MPI collective operations, including broadcasts, reductions, and all-reductions. We respectively denote $T_{all-to-all}, T_{broadcast}(p, w), T_{reduce}(p, w)$ and $T_{allreduce}(p, w)$ as respectively the cost of broadcast, reduction, and all-reduction for p processors to transfer an array of w words [4]. We consider the algorithms given in [34] for these collective operations, which are optimized for communication of long messages:

$$T_{all-to-all}(p, w) = \alpha \log p + \beta \frac{w}{2} \log p \tag{28}$$

$$T_{broadcast}(p, w) = \alpha \log p + \beta \frac{p-1}{p} w \tag{29}$$

$$T_{reduce}(p, w) = 2\alpha \log p + 2\beta \frac{p-1}{p} w + \gamma \frac{p-1}{p} w \tag{30}$$

$$T_{allreduce}(p, w) = 2\alpha \log p + 2\beta \frac{p-1}{p} w + \gamma \frac{p-1}{p} w \tag{31}$$

B. Detailed overhead analysis for out-of-node and in-node checksum storage

As the encoding and the retrieval of the uncoded output matrix respectively are preprocess and postprocess of the QR decomposition, and the recovery depends mainly on the generator matrix, the following overheads T_{enc}, T_{post} and T_f are independent of the algorithm used. They only depend on the pattern of 2D block-cyclic distribution and the generator matrix, which is illustrated by the following Theorem 3 and Theorem 4.

Theorem 6. *Under the out-of-node checksum storage, the coding strategy in Construction 1 can be designed to achieve:*

$$T_{post} \leq 3\alpha \log p_r + 3\beta \frac{p_r - 1}{p_r} \frac{n(n+1)}{P} + \gamma \frac{2n(n+1)}{P} \tag{32}$$

$$T_{enc} = T_f = \alpha \log(p_r + 1) + 2\beta \frac{p_r}{p_r + 1} \cdot \frac{n^2}{P} + \gamma \frac{n^2}{P} \tag{33}$$

Theorem 7. *Under the in-node checksum storage, the coding strategy in Construction 2 can be designed to achieve:*

$$T_{post} \leq 2\alpha \log p_r + 4\beta \frac{p_r - 1}{p_r} \frac{n(n+1)}{P} + \gamma \frac{2p_r + 3}{p_r} \frac{n(n+1)}{P} \tag{34}$$

$$T_{enc} = T_f = \alpha \log p_r + 2\beta \frac{p_r - 1}{p_r} \cdot \frac{n^2}{P} + \gamma \frac{n^2}{P} \tag{35}$$

The proofs of Theorem Theorem 6 and Theorem 7 are given respectively in Appendix D-C and D-D.

The proofs of Theorem 3 and relative bounds in Table I are given in Appendix D-E.

[4] It can be other units, such as bytes, but then the term β is the bandwidth per bytes.

Table I: Summary of overhead analysis for MGS [19]

Type of Overhead	Out-of-node Checksum Storage (Construction 1)	In-node Checksum Storage (Construction 2)
T_{post}	$\leq \frac{6}{p_c} \cdot T_{QR}$	$\leq \frac{8}{p_c} \cdot T_{QR}$
$T_{enc} = T_f$	$\leq \frac{4}{p_c} \cdot T_{QR}$	$\leq \frac{4}{p_c} \cdot T_{QR}$
T_{comp}	$\leq \frac{1}{p_r} \cdot T_{QR}$	$\leq \frac{2}{p_r} \cdot T_{QR}$
T_{coding}	$\leq (\frac{10}{p_c} + \frac{1}{p_r}) \cdot T_{QR}$	$\leq (\frac{12}{p_c} + \frac{2}{p_r}) \cdot T_{QR}$

C. Proof Theorem 6

We now prove the formulation of T_{enc}, T_f and T_{post} for the out-of-node checksum storage.

Encoding Overhead

Now we proceed to present the encoding strategy to achieve the encoding overhead T_{enc}, which is essentially the cost for computing the checksum $C = GA$. The final encoding strategy and its overhead are summarized in Lemma 8.

Lemma 8. *Our encoding strategy is composed of the following steps :*

1) *Every processor $\Pi(t, j)$ performs in parallel the transformation (43) of its local data $\bar{A}_{\Pi(t,j)} \to \hat{A}_{\Pi(t,j)}$.*
2) *On any j^{th} block-column in parallel, all p_r processors $\Pi(0, j), \Pi(1, j), ..., \Pi(p_r - 1, j)$ perform MPI_Reduce described in Observation 12. The checksum processor V_j is the destination of this MPI_Reduce call.*

The overhead of this encoding strategy is:

$$\begin{aligned} T_{enc} &= T_{reduce}(p_r, \frac{n^2}{P}) + \gamma \frac{1}{p_r} \frac{n^2}{P} \\ &= \alpha \log p_r + 2\beta \frac{p_r - 1}{p_r} \cdot \frac{n^2}{P} + \gamma \frac{n^2}{P} \end{aligned}$$

Proof of Lemma 8. The two steps of the encoding strategy will be further explained in details. The overhead for step 1 is $\gamma \frac{1}{p_r} \frac{n^2}{P}$ by Observation 11. The overhead of step 2 is $T_{reduce}(p_r, \frac{n^2}{P})$, given by Observation 12. The total overheads of these two steps form our encoding overhead T_{enc}. □

In our system model (Section II-B), the input $n \times n$ matrix A is divided into $N \times N$ blocks A_{ij} ($i \in [0, N-1]$, $j \in [0, N-1]$), each of which is of size $b \times b$ and owned by processor $\Pi(i, j)$. Under out-of-node checksum storage, Figures 1a and 1b respectively illustrate the distribution of processors in the case of failure-free state and single-node failure. The p_c checksum processors are $V_0, V_1, ..., V_{p_c-1}$ where V_j owns the $j^{th}, (j + p_c)^{th}, (j + 2p_c)^{th}, ...$ block-columns (i.e. column of $b \times b$ blocks). In particular, for any j^{th} block-column:

$$\begin{bmatrix} C_{0,j} \\ C_{1,j} \\ \vdots \\ C_{\frac{N}{p_r}-1,j} \end{bmatrix} = G \begin{bmatrix} A_{0,j} \\ A_{1,j} \\ \vdots \\ A_{N-1,j} \end{bmatrix} \tag{36}$$

where each A_{ij} is owned by processor $\Pi(i, j)$ and, under our restriction $p_r | N$ (to avoid unecessary complication), C_{ij} is owned by the checksum processor $V_{(j \mod p_c)}$. It is notable that $\Pi(i, j) = \Pi(i \mod p_r, j)$, so there are p_r processors vertically covering all the data blocks.

Similar to Appendix B-B, we can obtain the checksum C_{ij} ($i \in [0, \frac{N}{p_r} - 1]$) by:

$$C_{i,j} = \lambda A_{i,j} + \sum_{l=1}^{p_r - 1} A_{i + l\frac{N}{p_r}, j} \tag{37}$$

We now present the strategy to compute (37).

Observation 10. *The sub-index j in (37) is fixed, so we only need vertical communication of data. We can design the communication of data among all the vertical processors $\Pi(0, j), \Pi(1, j), ..., \Pi(p_r - 1, j)$ and the checksum processors V_j, all of which occupy data blocks on the same block-column. Each column-wise communication can run in parallel for all j.*

Figure 1a illustrates the algorithmic point of view. However, on the local view of each processor, processor $\Pi(t, j)$ (where $t \in [0, p_r - 1], j \in [0, p_c - 1]$) holds the following matrix of concatenated data blocks:

$$\begin{bmatrix} A_{t,j} & A_{t,j+p_c} & \cdots & A_{t,j+(\frac{N}{p_c}-1)p_c} \\ A_{t+p_r,j} & A_{t+p_r,j+p_c} & \cdots & A_{t+p_r,j+(\frac{N}{p_c}-1)p_c} \\ \vdots & \vdots & \ddots & \vdots \\ A_{t+(\frac{N}{p_r}-1)p_r,j} & A_{t+(\frac{N}{p_r}-1)p_r,j+p_c} & \cdots & A_{t+(\frac{N}{p_r}-1)p_r,j+(\frac{N}{p_c}-1)p_c} \end{bmatrix} \tag{38}$$

And the checksum processors V_j holds the following matrix of concatenated checksums:

$$\begin{bmatrix} C_{0,j} & C_{0,j+p_c} & \cdots & C_{0,j+(\frac{N}{p_c}-1)p_c} \\ C_{1,j} & C_{1,j+p_c} & \cdots & C_{1,j+(\frac{N}{p_c}-1)p_c} \\ \vdots & \vdots & \ddots & \vdots \\ C_{\frac{N}{p_r}-1,j} & C_{\frac{N}{p_r}-1,j+p_c} & \cdots & C_{\frac{N}{p_r}-1,j+(\frac{N}{p_c}-1)p_c} \end{bmatrix} \tag{39}$$

We denote $\bar{A}_x^j = \begin{bmatrix} A_{xj} & A_{x,j+p_c} & \cdots & A_{x,j+(\frac{N}{p_c}-1)p_c} \end{bmatrix}$ and $\bar{C}_x^j = \begin{bmatrix} C_{xj} & C_{x,j+p_c} & \cdots & C_{x,j+(\frac{N}{p_c}-1)p_c} \end{bmatrix}$. The data matrix (38) stored locally on $\Pi(t, j)$ can be written as:

$$\bar{A}_{\Pi(t,j)} = \begin{bmatrix} \bar{A}_t^j \\ \bar{A}_{t+p_r}^j \\ \vdots \\ \bar{A}_{t+(\frac{N}{p_r}-1)p_r}^j \end{bmatrix} \tag{40}$$

And the matrix of checksums stored locally on checksum processor V_j can be written as:

$$\bar{C}_j = \begin{bmatrix} \bar{C}_0^j \\ \bar{C}_1^j \\ \vdots \\ \bar{C}_{\frac{N}{p_r}-1}^j \end{bmatrix} \tag{41}$$

Using the equation (37), we obtain that:

$$\bar{C}_i^j = \lambda \bar{A}_i^j + \sum_{l=1}^{p_r - 1} \bar{A}_{i + l\frac{N}{p_r}}^j \tag{42}$$

Following Observation 10, for a fixed j, we now proceed to describe the communication pattern to compute all $\bar{C}_i^j$ ($i \in [0, \frac{N}{p_r} - 1]$), which ultimately completes the computation of all checksums, stored as $\bar{C}_j$, of checksum processors V_j.

As $gcd(\frac{N}{p_r}, p_r) = 1$, in Equation (40) the set of indices $\{t, t + p_r, t + 2p_r, ..., t + (\frac{N}{p_r} - 1)p_r\}$ forms a least residue system modulo $\frac{N}{p_r}$. Therefore, if we let:

$$f(t, v) = t + xp_r \text{ for } x \in [0, \frac{N}{p_r} - 1]$$
$$\text{such that } (t + xp_r \mod \frac{N}{p_r} = v)$$

then the set $\{f(t,0), f(t,1), ..., f(t, \frac{N}{p_r} - 1)\}$ also forms a least residue system modulo $\frac{N}{p_r}$, and is thus just a reordering. On each processor $\Pi(t, j)$, We further reorder the $\bar{A}_x^j$'s in Equation (40) to pepare data for checksum computation:

$$\bar{A}_{\Pi(t,j)} \to \hat{A}_{\Pi(t,j)} = \begin{bmatrix} \lambda_{t,0} \cdot \bar{A}^j_{f(t,0)} \\ \lambda_{t,1} \cdot \bar{A}^j_{f(t,1)} \\ \vdots \\ \lambda_{t,\frac{N}{p_r}-1} \cdot \bar{A}^j_{f(t,\frac{N}{p_r}-1)} \end{bmatrix} \quad (43)$$

Where $\lambda_{t,v} = \begin{cases} 1, & \text{if } f(t,v) \geq \frac{N}{p_r} \\ \lambda, & \text{if } f(t,v) < \frac{N}{p_r} \end{cases}$

The transformation (43) is essentially the reordering of data and the multiplications of λ with the matrices $\bar{A}^j_{f(t,v)}$'s ,

Observation 11. *The trasformation (43) incurs $\gamma \frac{1}{p_r} \frac{n^2}{P}$ overhead.*

Proof of Observation 11. The reordering of data is only logical, and we do not perform any real computation of data. On the other hand, each multiplication of λ with the matrix $\bar{A}^j_{f(t,v)}$ of size $b \times \frac{n}{p_c}$ incurs $\gamma \frac{bn}{p_c}$ overhead. It is left to count how many such multiplications are performed.

The definition of $\lambda_{t,v}$ shows that multiplication occurs when the index $f(t,v) < \frac{N}{p_r}$. By Equation (40), each processor $\Pi(t,j)$ has at most $\frac{N}{p_r^2}$ indices that are $< \frac{N}{p_r}$, where those $\Pi(0,j)$ have exactly $\frac{N}{p_r^2}$ such indices.

The transformation (43) is executed in parallel among all processors, so the overhead is equal to that of the processor with highest overhead. Therefore, the total overhead is $\frac{N}{p_r^2} \cdot \gamma \frac{bn}{p_c} = \gamma \frac{1}{p_r} \frac{n^2}{P}$. □

Observation 12. *The checksum matrix $\bar{C}_j$ of the checksum processors V_j can be computed by:*

$$\bar{C}_j = \sum_{t=0}^{p_r - 1} \hat{A}_{\Pi(t,j)} \quad (44)$$

Furthermore, the computation (44) can be performed by one MPI_Reduce incurring $T_{reduce}(p_r + 1, \frac{n^2}{P})$.

Proof of Observation 12. We first prove the equation (44).

In the checksum formulation (42), the set of indices $W_1 = \{i, i + \frac{N}{p_r}, i + 2\frac{N}{p_r}, ..., i + (p_r - 1)\frac{N}{p_r}\}$ forms a least residue system modulo p_r as $gcd(\frac{N}{p_r}, p_r) = 1$. We consier the set of indices $W_2 = \{f(0,i), f(1,i), f(2,i), ..., f(p_r - 1, i)\}$ and prove that $W_2 \equiv W_1$.

For any $f(l,i) \in W_2$, by definition there exists some $x \in [0, \frac{N}{p_r} - 1]$ such that $f(l,i) = l + xp_r$. This implies $f(l,i) \equiv l(\mod p_r)$, and thus W_2 forms a least residue system modulo p_r. $W_2 \equiv W_1$ if we can prove that any such W_2's element $f(l,i) \in W_1$. By definition, $f(l,i) \equiv i(\mod \frac{N}{p_r})$ so we can write $f(l,i) = i + q\frac{N}{p_r}$ (where $q \geq 0$). We have:

$$q\frac{N}{p_r} \leq i + q\frac{N}{p_r} = l + xp_r (= f(l,i))$$
$$\leq (p_r - 1) + (\frac{N}{p_r} - 1)p_r < M$$
$$q < p_r$$

The last line implies $q \in [0, p_r - 1]$, so $f(l,i) = i + q\frac{N}{p_r} \in W_1$. Therefore, $W_2 \equiv W_1$.

Based on the definition of $\lambda_{t,v}$ (in (43)) and the fact that $W_2 \equiv W_1$, we can refine the checksum formulation (42) as:

$$\bar{C}_i^j = \sum_{l=0}^{p_r - 1} \lambda_{l,i} \cdot \bar{A}^j_{f(l,i)} \quad (45)$$

The equation (44) is directly derived from (45).

Now to computer $\bar{C}_j$, each of the p_r processors $\Pi(t,j)$ ($t = 0 \to p_r - 1$) performs the MPI_Reduce by sending its $\hat{A}_{\Pi(t,j)}$ to checksum processor V_j. As the matrix $\hat{A}_{\Pi(t,j)}$ is of size $\frac{n}{p_r} \frac{n}{p_c} = \frac{n^2}{P}$, this invokes the overhead $T_{reduce}(p_r + 1, \frac{n^2}{P})$.

□

Overhead for Single-node Failure Recovery

Now we present the recovery process in the presence of failure and analyze the overhead T_f for recovering from one failure.

Remark 4. *In the actual implementation, the algorithm iteratively replaces (from left to right) columns of A by columns of Q. In the analysis below, we still keep A without introducing additional notations for simplicity. However, the recovery does include Q-factor protection.*

When a processor $\Pi(s,j)$ fails, we lose all of its data $\bar{A}_{\Pi(s,j)}$. To recover $\bar{A}_{\Pi(s,j)}$, we first recover its transformed data $\hat{A}_{\Pi(s,j)}$ given in (43). By (44), $\hat{A}_{\Pi(s,j)}$ can be recovered from $\bar{C}_j$ held by checksum processors V_j:

$$\hat{A}_{\Pi(s,j)} = \bar{C}_j - \sum_{\substack{t=0 \\ t \neq s}}^{p_r - 1} \hat{A}_{\Pi(t,j)} \quad (46)$$

The computation (46) can be performed by one MPI_Reduce with V_j sending $\bar{C}_j$ and each $\Pi(t,j)$ ($t \neq s$) sending its $-\hat{A}_{\Pi(t,j)}$ to $\Pi(s,j)$ incurring $T_{reduce}(p_r + 1, \frac{n^2}{P})$ (as there are 1 checksum processor and p_r systematic processors, each of which holds $\frac{n}{p_r} \frac{n}{p_c} = \frac{n^2}{P}$ data).

Then we reverse-transform $\hat{A}_{\Pi(t,j)} \to \bar{A}_{\Pi(t,j)}$. Under similar reasonings to Observation 11's, this reverse-transformation incurs $\gamma \frac{1}{p_r} \frac{n^2}{P}$ overhead.

Therefore, the overhead T_f for recovering from one failure is:

$$T_f = T_{reduce}(p_r + 1, \frac{n^2}{P}) + \gamma \frac{1}{p_r} \frac{n^2}{P} (= T_{enc})$$
$$= \alpha \log(p_r + 1) + 2\beta \frac{p_r}{p_r + 1} \cdot \frac{n^2}{P} + \gamma \frac{n^2}{P}$$

Postprocessing Overhead

The overhead for postprocessing is attributed to the cost for performing matrix multiplications $(G_0 Q_1)$ and $(G_0 \boldsymbol{b})$ (refer to Section IV-A). For convenience, we denote the $n \times (n+1)$ matrix $Z = \begin{bmatrix} Q_1 & \boldsymbol{b} \end{bmatrix} = \begin{bmatrix} Z_0 \\ Z_1 \\ \vdots \\ Z_{p_r - 1} \end{bmatrix}$, where Z_i is $\frac{n}{p_r} \times (n+1)$ and Z is $n \times (n+1)$. Then the postprocessing overhead is the cost for performing matrix multiplication $G_0 Z$, which is further expressed as the followings to utilize the sparsity of G_0:

$$G_0 Z = \begin{bmatrix} I_{\frac{n}{p_r}} + G_1 & V \\ V^T & -I_{m - \frac{n}{p_r}} \end{bmatrix} Z$$
$$= \begin{bmatrix} (\lambda+1) I_{\frac{n}{p_r}} & I_{\frac{n}{p_r}} & I_{\frac{n}{p_r}} & \cdots & I_{\frac{n}{p_r}} \\ I_{\frac{n}{p_r}} & & & & \\ I_{\frac{n}{p_r}} & & & -I_{m - \frac{n}{p_r}} & \\ \vdots & & & & \\ I_{\frac{n}{p_r}} & & & & \end{bmatrix} \begin{bmatrix} Z_0 \\ Z_1 \\ \vdots \\ Z_{p_r - 1} \end{bmatrix}$$
$$= \begin{bmatrix} \lambda Z_0 + \sum_{l=0}^{p_r - 1} Z_l \\ Z_0 - Z_1 \\ Z_0 - Z_2 \\ \vdots \\ Z_0 - Z_{p_r - 1} \end{bmatrix} = \begin{bmatrix} W_0 \\ W_1 \\ \vdots \\ W_{p_r - 1} \end{bmatrix} \quad (47)$$

Our goal is to compute $G_0 Z = \begin{bmatrix} W_0 & W_1 & \cdots & W_{\frac{n}{c} - 1} \end{bmatrix}^T$ distributed 2D-block cyclically over the P systematic processors.

The final postprocessing strategy and its overhead are summarized in Lemma 9.

Lemma 9. *Our postprocessing strategy is composed of the following steps:*

1) *Compute $\sum_{l=0}^{p_r - 1} Z_l$ through MPI_Allreduce call incurring $T_{allreduce}(p_r, \frac{n(n+1)}{P})$ overhead.*
2) *Distribute Z_0 through MPI_Broadcast call incurring $T_{broadcast}(p_r, \frac{n(n+1)}{P})$ overhead.*
3) *Compute all W_i's in parallel, incurring $\gamma \frac{p_r + 1}{p_r} \frac{n(n+1)}{P}$ overhead.*

The overhead of this strategy is bounded by:

$$T_{post} \leq 3\alpha \log p_r + 3\beta \frac{p_r - 1}{p_r} \frac{n(n+1)}{P} + \gamma \frac{2n(n+1)}{P}$$

Proof of Lemma 9. We further break down the three steps in details.

Step 1.

Under the 2D-block cyclic distribution, the $n \times (n+1)$ matrix Z is distributed over $\frac{n}{b} \times \frac{n+1}{b}$ data-blocks of size $b \times b$, where data-block $Z[i,j]$ is owned by processor $\Pi(i,j)$.

Similarly, the $\frac{n}{p_r} \times (n+1)$ matrix Z_t is distributed over $\frac{N}{p_r} \times \frac{n+1}{b}$ data-blocks of size $b \times b$. We further denote $Z_t[i,j]$ as the i^{th} from up to down and j^{th} from left to right data-block of Z_t. (just like how $Z[i,j]$ is the data-block of Z).

Observation 13. *The following relation holds:*

$$Z_t[i,j] = Z[i + t\frac{N}{p_r}, j] \quad (48)$$

Proof of Observation 13. There are t such $Z_0, Z_1, ..., Z_{t-1}$ above Z_t, each of which has $\frac{N}{p_r}$ data-blocks. □

The computation of $\sum_{l=0}^{\frac{n}{c} - 1} Z_l$ is distributed as the computation of local data-blocks as:

$$\sum_{l=0}^{\frac{n}{c} - 1} Z_l[i,j] = \sum_{l=0}^{p_r - 1} Z[i + l\frac{N}{p_r}, j] \quad (49)$$

Fix i and j. Each $Z[i + l\frac{N}{p_r}, j]$ is held by processor $\Pi(i + l\frac{N}{p_r}, j) \equiv \Pi(i + l\frac{N}{p_r} \mod p_r, j)$. As $gcd(\frac{N}{p_r}, p_r) = 1$, the set $\{i, i + \frac{N}{p_r}, i + 2\frac{N}{p_r}, ..., i + (p_r - 1)\frac{N}{p_r}\}$ forms a least residue system modulo p_r. Therefore, the sum (49) can be computed through one MPI_Reduce call over p_r processors vertically.

Varying i and j, the sum $\sum_{l=0}^{\frac{n}{c} - 1} Z_l$ can be computed through $\frac{n+1}{bp_c}$ MPI_Allreduce call in parallel, where the j^{th} call is over p_r processors $\Pi(0,j), \Pi(1,j), \Pi(2,j), ..., \Pi(p_r - 1, j)$.

Therefore, the overhead incurred is $T_{allreduce}(p_r, \frac{n(n+1)}{P})$, as each processor holds $\frac{n}{p_r} \frac{n+1}{p_c} = \frac{n(n+1)}{P}$ data points.

Step 2.

The reasonings are similar to step 1's, based on the local data-blocks $Z_t[i,j]$.

The overhead incurred is at most $T_{broadcast}(p_r, \frac{n(n+1)}{P})$, as any processor holds at most $\frac{n}{p_r} \frac{n}{p_c} = \frac{n(n+1)}{P}$ data-points in Z_0 to be broadcasted.

Step 3.

After the first two steps, we are now ready to compute all the $W_0, W_1, ..., W_{p_r - 1}$. Each processor needs to compute $\frac{n(n+1)}{p_r P}$ data-points in W_0, and $(1 - \frac{1}{p_r}) \frac{n(n+1)}{P}$ data-points in the remaining W_l's. Any computation in W_0 incurs 2 flops (one for multiplication with λ and one for addition), while any computation in other W_l's incurs 1 flop (for subtraction). Therefore, the computation incurred by this step 3 is:

$$\gamma(2 \cdot \frac{n(n+1)}{p_r P} + 1 \cdot (1 - \frac{1}{p_r}) \frac{n(n+1)}{P}) = \gamma \frac{p_r + 1}{p_r} \frac{n(n+1)}{P}$$

Summing up the overhead of all three steps, we derive the postprocessing overhead:

$$\begin{aligned}T_{post} \leq & T_{allreduce}(p_r, \frac{n(n+1)}{P}) + T_{broadcast}(p_r, \frac{n(n+1)}{P}) \\ & + \gamma \frac{p_r+1}{p_r}\frac{n(n+1)}{P} \\ = & 2\alpha \log p_r + 2\beta \frac{p_r-1}{p_r}\frac{n(n+1)}{P} + \gamma \frac{p_r-1}{p_r}\frac{n(n+1)}{P} \\ & + \alpha \log p_r + \beta \frac{p_r-1}{p_r}\frac{n(n+1)}{P} + \gamma \frac{p_r+1}{p_r}\frac{n(n+1)}{P} \\ = & 3\alpha \log p_r + 3\beta \frac{p_r-1}{p_r}\frac{n(n+1)}{P} + \gamma \frac{2n(n+1)}{P}\end{aligned}$$

□

D. Proof of Theorem 7

We now prove the formulation of T_{enc}, T_f and T_{post} for the in-node checksum storage.

Encoding Overhead

The reasonings can be adopted from the case of out-of-node checksum storage (Appendix D-C), with the two differences:

1) We change the MPI_Reduce call, which is meant to aggregate data to the checksum processor, to MPI_Allreduce, which distributes the aggregated data to all the processors.
2) The number of processors participating in the MPI call is p_r instead of p_r+1, as we now do not have the extra checksum processor.

The overhead is:

$$\begin{aligned}T_{enc} &= T_{allreduce}(p_r, \frac{n^2}{P}) + \gamma \frac{1}{p_r}\frac{n^2}{P} \\ &= \alpha \log(p_r) + 2\beta \frac{p_r-1}{p_r} \cdot \frac{n^2}{P} + \gamma \frac{n^2}{P}\end{aligned}$$

Overhead for Single-node Failure Recovery

With reasonings similar to Appendix D-C, e obtain the overhead as:

$$T_f = T_{enc} = \alpha \log(p_r) + 2\beta \frac{p_r-1}{p_r} \cdot \frac{n^2}{P} + \gamma \frac{n^2}{P}$$

Postprocessing Overhead

The overhead for postprocessing is attributed to the cost for performing matrix multiplications (G_0Q_1) and $(G_0\boldsymbol{b})$ (refer to Section IV-A). For convenience, we denote the $n \times (n+1)$ matrix $Z = \begin{bmatrix} Q_1 & \boldsymbol{b} \end{bmatrix} = \begin{bmatrix} Z_0 \\ Z_1 \\ \vdots \\ Z_{p_r-1} \end{bmatrix}$, where Z_i is $\frac{n}{p_r} \times (n+1)$ and Z is $n \times (n+1)$. Then the postprocessing overhead is the cost for performing matrix multiplication G_0Z, which is further expressed as the followings to utilize the sparsity of G_0:

$$G_0 Z = \begin{bmatrix} I_{\frac{2n}{p_r}} + G_1 & V \\ V^T & -I_{m-\frac{2n}{p_r}} \end{bmatrix} Z \tag{50}$$

$$= \begin{bmatrix} (\lambda+1)I_{\frac{n}{p_r}} & I_{\frac{n}{p_r}} & I_{\frac{n}{p_r}} & \cdots & I_{\frac{n}{p_r}} \\ \lambda I_{\frac{n}{p_r}} & (\lambda+1)I_{\frac{n}{p_r}} & I_{\frac{n}{p_r}} & \cdots & I_{\frac{n}{p_r}} \\ I_{\frac{n}{p_r}} & I_{\frac{n}{p_r}} & & & \\ I_{\frac{n}{p_r}} & I_{\frac{n}{p_r}} & & & \\ \vdots & & & -I_{m-\frac{2n}{p_r}} & \\ I_{\frac{n}{p_r}} & I_{\frac{n}{p_r}} & & & \end{bmatrix} \begin{bmatrix} Z_0 \\ Z_1 \\ \vdots \\ Z_{p_r-1} \end{bmatrix} \tag{51}$$

$$= \begin{bmatrix} (\lambda+1)Z_0 + \lambda Z_1 + \sum_{l=2}^{p_r-1} Z_l \\ \lambda Z_0 + (\lambda+1)Z_1 + \sum_{l=2}^{p_r-1} Z_l \\ (Z_0+Z_1) - Z_2 \\ (Z_0+Z_1) - Z_3 \\ \vdots \\ (Z_0+Z_1) - Z_{p_r-1} \end{bmatrix} = \begin{bmatrix} W_0 \\ W_1 \\ \vdots \\ W_{p_r-1} \end{bmatrix} \tag{52}$$

Our goal is to compute $G_0Z = \begin{bmatrix} W_0 & W_1 & \cdots & W_{\frac{n}{c}-1} \end{bmatrix}^T$ distributed 2D-block cyclically over the P systematic processors.

The final postprocessing strategy and its overhead are summarized in Lemma 10.

Lemma 10. *Our postprocessing strategy is composed of the following steps:*

1) *Compute $\sum_{l=2}^{p_r-1} Z_l$ and Z_0+Z_1 through MPI_Allreduce call incurring $T_{allreduce}(p_r, \frac{2n(n+1)}{P})$ overhead.*
2) *Compute all W_i's in parallel, incurring $\gamma \frac{p_r+1}{p_r}\frac{n(n+1)}{P}$ overhead.*

The overhead of this strategy is bounded by:

$$\begin{aligned}T_{post} \leq & 2\alpha \log p_r + 4\beta \frac{p_r-1}{p_r}\frac{n(n+1)}{P} \\ & + \gamma \frac{2p_r+3}{p_r}\frac{n(n+1)}{P}\end{aligned}$$

Proof of Lemma 10. We further break down the two steps in details.

Step 1.

The computation of $\sum_{l=2}^{p_r-1} Z_l$ is similar to Step 1 in Lemma 9, where one MPI_Allreduce call is invoked. In addition, we also utilize this MPI_Allreduce to compute Z_0+Z_1 which doubles the the amount of transferred data to $\frac{2n(n+1)}{P}$). The cost of the MPI_Allreduce call in Step 1 is thus at most $T_{allreduce}(p_r, \frac{2n(n+1)}{P})$.

Step 2.

After the first step, $\sum_{l=2}^{p_r-1} Z_l$ and Z_0+Z_1 are distributed over all P processors, so we are now ready to compute all the $W_0, W_1, ..., W_{p_r-1}$. Each processor needs to compute $\frac{2n(n+1)}{p_r P}$ data-points in W_0 and W_1 (combined), and $(1-\frac{2}{p_r})\frac{n(n+1)}{P}$ data-points in the remaining W_l's. Any computation in W_0 or W_1 incurs 3 flops (one for multiplication between λ and (Z_0+Z_1) and two for the addition $\lambda(Z_0+Z_1)+Z_i+(\sum_{l=2}^{p_r-1} Z_l)$ of W_i), while any computation in other W_l's incurs 1 flop

(for subtraction). Therefore, the computation incurred by this step 3 is:

$$\gamma(3\cdot\frac{2n(n+1)}{p_rP}+1\cdot(1-\frac{2}{p_r})\frac{n(n+1)}{P})=\gamma\frac{p_r+4}{p_r}\frac{n(n+1)}{P}$$

Summing up the overhead of the two steps, we derive the postprocessing overhead:

$$\begin{aligned}T_{post}\leq&T_{allreduce}(p_r,\frac{2n(n+1)}{P})+\gamma\frac{p_r+4}{p_r}\frac{n(n+1)}{P}\\=&2\alpha\log p_r+2\beta\frac{p_r-1}{p_r}\frac{2n(n+1)}{P}\\&+\gamma\frac{p_r-1}{p_r}\frac{2n(n+1)}{P}+\gamma\frac{p_r+4}{p_r}\frac{n(n+1)}{P}\\=&2\alpha\log p_r+4\beta\frac{p_r-1}{p_r}\frac{n(n+1)}{P}+\gamma\frac{2p_r+3}{p_r}\frac{n(n+1)}{P}\end{aligned}$$

□

E. Proof of Theorem 3 and Table I

Input: $n\times b$ matrix A distributed 1D-block cyclically over p_r processors.
Output: Q, R
Result: $A=QR$, where $Q^TQ=I$

```
Q=A
for i = 1 to b do
    q = Q[:, i]
    All-reduce: vector v = Q[:, i:b]^T · Q[:, i]
    q = q/√v[1]
    Q[:, i] = q
    R[i, i:b] = v^T
    if i < b then
        Q[:, i+1:b] -= q · v[i+1:b]^T
    end
end
```

Algorithm 4: CGS(A, n, b, p_r)

Input: $n\times n$ matrix A distributed 2D-block cyclically over $P=p_r\times p_c$ processors.
Output: Q, R
Result: $A=QR$, where $Q^TQ=I$

```
Q=A
for i = 1 to n Step b do
    Q̄ = Q[:, i:i+b-1]
    ICGS(Q̄, R[i:i+b-1, i:i+b-1])
    Q[:, i:i+b-1] = Q̄
    if i < n − b then
        All-reduce: R̄ = Q̄^T · Q[:, i+b:n]
        R[i:i+b-1, i:i+b-1] = R̄
        Q[:, i+b:n] -= Q̄ · R̄
    end
end
```

Algorithm 5: PBMGS(A, n, n, p_r, p_c)

We consider the parallel blocked MGS algorithm (PBMGS) in [19], which is specifically designed for 2D-block cyclic distribution. However, our analysis can be adapted to other algorithms in [18], [23].

At high level, the algorithm PBMGS (Algorithm 11) iterates through $N=\frac{n}{b}$ block-column. At the i^{th} iterations it performs the iterated classical Gram-Schmidt algorithm (ICGS) on the whole block-column, and updates the remaining part (on the right side of the block-column) of the matrix.

The ICGS algorithm is essentially the classical Gram-Schmidt (CGS), except that in each iteration ICGS further reorthogonalizes for one more time, depending on an easy-to-compute criterion [31]. The cost of ICGS algorithm is thus lower bounded by the cost of CGS algorithm (Algorithm 11).

$T_{CGS}(A,n,b,p_r)$ denotes the cost of performing CGS algorithm on the $n\times b$ matrix A distributed 1D-block cyclically over p_r processors.
$T_{PMGS}(A,n,n,p_r,p_c)$ denotes the cost of performing PMGSGS algorithm on the $n\times n$ matrix A distributed 2D-block cyclically over $P=p_r\times p_c$ processors.

In the i^{th} iteration of algorithm $CGS(A,n,b,p_r)$, the computation of the $(b-i+1)\times 1$ vector $\boldsymbol{v}$ requires the local matrix-vector multiplication on each of the p_r processors incurring $\gamma 2\frac{n}{p_r}\cdot(b-i+1)$, and one MPI_Reduce call with overhead $T_{reduce}(p_r,b-i+1)$. Summing up over b iterations, the cost of the CGS algorithm can be lower bounded by:

$$\begin{aligned}T_{CGS}(A,n,b,p_r)\geq&\sum_{i=1}^{b}[T_{reduce}(p_r,b-i+1)+\gamma 2\frac{n}{p_r}(b-i+1)]\\=&\sum_{i=1}^{b}[2\alpha\log p_r+\beta\frac{p_r-1}{p_r}(b-i+1)\\&+\gamma\frac{p_r-1}{p_r}(b-i+1)]+\gamma\frac{nb(b+1)}{p_r}\end{aligned}$$

Hence,

$$T_{CGS}(A,n,b,p_r)\geq 2\alpha b\log p_r+\gamma\frac{nb(b+1)}{p_r}\qquad(53)$$

In the i^{th} iteration of algorithm $PBMGS(A,n,n,p_r,p_c)$, the computation of the $b\times(n-i-b+1)$ matrix $\bar{R}$ requires the following main steps (we do not list all steps) and communication patterns:

1) ICGS algorithm to update $\bar{Q}$, incurring $\geq T_{CGS}(A,n,b,p_r)$ overhead.
2) Horizontal broadcast of $\bar{Q}$ via MPI_Broadcast (in preparation for next step), incurring $T_{broadcast}(p_c,\frac{n}{p_r}b)$ overhead.
3) Local matrix-matrix multiplication: each of the p_r processors prepares its local inner-products through a matrix-matrix multiplication of the local parts of $\bar{Q}$ (transposed) with its local parts of Q. This incurs $\gamma 2b\frac{n}{p_r}\cdot\frac{n-i-b+1}{p_c}=\gamma 2\frac{n}{P}b(n-i-b+1)$ overhead.
4) MPI_Reduce call with overhead $T_{reduce}(p_r,b(n-i-b+1))$ over p_r vertical processors on each block-column.

(Note: The local matrix-matrix multiplication in step 3 results in a $b \times (n-i-b+1)$ submatrix.)

Summing up over all $\frac{n}{b}$ iterations (with step b), the cost of the PBMGS algorithm can be lower bounded by:

$$\begin{aligned}
&T_{PBMGS}(A,n,n,p_r,p_c)\\
&\geq \frac{n}{b}T_{CGS}(A,n,b,p_r)+\frac{n}{b}T_{broadcast}(p_c,\frac{n}{p_r}b)\\
&+\sum_{\substack{j=0\\ i=1+bj}}^{\frac{n}{b}-1}[T_{reduce}(p_r,b(n-i-b+1))+\gamma 2\frac{n}{P}b(n-i-b+1)]\\
&\geq \frac{n}{b}[2\alpha b\log p_r+\gamma\frac{nb(b+1)}{p_r}]+\frac{n}{b}(\alpha\log p_c+\beta\frac{p_c-1}{p_c}\frac{n}{p_r}b)\\
&+\sum_{\substack{j=0\\ i=1+bj}}^{\frac{n}{b}-1}[\gamma 2\frac{n}{P}b(n-i-b+1)] \text{ (by (53))}\\
&\geq [2\alpha n\log p_r+\gamma\frac{n^2(b+1)}{p_r}]+[\beta\frac{(p_c-1)n^2}{P}]\\
&+\gamma(\frac{n^3}{P}-\frac{n^2b}{P})\\
&=2\alpha n\log p_r+\beta\frac{(p_c-1)n^2}{P}+\gamma(\frac{n^3}{P}+\frac{n^2(b+1)p_c}{P}-\frac{n^2b}{P})\\
&\geq 2\alpha n\log p_r+\beta\frac{1}{2}\frac{p_cn(n+1)}{P}+\gamma\frac{n^2(n+1)}{P}
\end{aligned}$$

Therefore,

$$T_{QR}\geq 2\alpha n\log p_r+\beta\frac{1}{2}\frac{p_cn(n+1)}{P}+\gamma\frac{n^2(n+1)}{P} \quad (54)$$

Observation 14. *For the MGS algorithm [19], the overhead T_{comp} for running QR decomposition on the larger $(m+c)\times n$ encoded matrix $\widetilde{A}$ instead of $m\times n$ matrix A is bounded by:*

$$T_{comp}\leq\frac{c}{m}\cdot T_{QR} \quad (55)$$

Note: We hereby derive the generalized result for $m\times n$ matrix A, and later set $m=n$ in our overhead analysis.

Proof of Observation 14. As any algorithm in [18], [19], [23] iterates from left to right over the width of the matrix, there are $O(n)$ such iterations. In each iteration, all the mentioned algorithms use some form of MPI call (in Appendix D-A) excluding, MPI_Alltoall. In particular, MPI_Broadcast is required to communicate the data blocks, or MPI_Reduce/MPI_Allreduce is used to compute inner-products or matrix-products. As each processor owns $\frac{mn}{P}$ data, the data transferred in any such MPI call does not exceed $O(\frac{mn}{P})$. Any such MPI call (excluding MPI_Alltoall) by Appendix D-A would incur $O(\alpha\log P)+O(\beta\frac{mn}{P})$ communication. In terms of computation, all these efficient algorithms do not exceed $\gamma O(\frac{mn}{P})$ overhead. Therefore, summing up over $O(n)$ iterations, we can get the asymptotical cost for T_{QR} as:

$$T_{QR}=O(\alpha n\log P+\beta\frac{mn^2}{P}+\gamma\frac{mn^2}{P}) \quad (56)$$

As m increases at most linearly with the asymptotical cost of T_{QR}, the overhead T_{comp} for running QR decomposition on some $(m+c)\times n$ matrix against some $m\times n$ matrix can be bounded by:

$$T_{comp}\leq\frac{c}{m}T_{QR}$$

Remark 5. *In (56), we also note that $T^{\alpha}_{comp}=O(1)$, as the increase of m does not affect the α term.*

□

Proof of results in Table I. Here, we consider the out-of-node checksum storage setting. The proof for in-node checksum storage setting can be done similarly.

By Theorem 6 , we have:

$$\begin{aligned}
T_{post}&\leq 3\alpha\log p_r+3\beta\frac{p_r-1}{p_r}\frac{n(n+1)}{P}+\gamma\frac{2n(n+1)}{P}\\
&\leq\frac{6}{p_c}(\alpha\frac{p_c}{2}\log p_r+\beta\frac{p_c}{2}\frac{n(n+1)}{P}+\gamma\frac{p_c}{3}\frac{n(n+1)}{P})\\
&\leq\frac{6}{p_c}(2\alpha n\log p_r+\beta\frac{1}{2}\frac{p_cn(n+1)}{P}+\gamma\frac{n^2(n+1)}{P})\\
&\text{(As } p_c\leq n)\\
&\leq\frac{6}{p_c}\cdot T_{QR}\text{ (by the inequality (54))}\\
T_{enc}&=T_f=\alpha\log(p_r+1)+2\beta\frac{p_r}{p_r+1}\cdot\frac{n^2}{P}+\gamma\frac{n^2}{P}\\
&\leq\frac{4}{p_c}(\alpha\frac{p_c}{4}\log(p_r+1)+\beta\frac{1}{2}\cdot\frac{p_cn^2}{P}+\gamma\frac{p_cn^2}{4P})\\
&\leq\frac{4}{p_c}(2\alpha n\log p_r+\beta\frac{1}{2}\frac{p_cn(n+1)}{P}+\gamma\frac{n^2(n+1)}{P})\\
&\text{(As } p_c\leq n)\\
&\leq\frac{4}{p_c}\cdot T_{QR}\text{ (by the inequality (54))}\\
\therefore T_{coding}&=T_{enc}+T_{post}+T_{comp}\leq\frac{10}{p_c}\cdot T_{QR}+\frac{c}{n}\cdot T_{QR}\\
&=(\frac{10}{p_c}+\frac{c}{n})T_{QR}
\end{aligned}$$

For the coding strategy in Construction 1 (for out-of-node checksum storage), we have $c=\frac{n}{p_r}$ and thus obtain:

$$T_{coding}\leq(\frac{10}{p_c}+\frac{1}{p_r})T_{QR}$$

Proof of Theorem 3. From the proof of Table I, we have:

$$T_{coding}\leq(\frac{10}{p_c}+\frac{2}{p_r})T_{QR}$$

$$T_f\leq\frac{4}{p_c}\cdot T_{QR}$$

So $T_{coding}=O(\frac{1}{p_r}+\frac{1}{p_c})\cdot T_{QR}$, and $T_f=O(\frac{1}{p_c})\cdot T_{QR}$.

By (54) and Theorem 6, we have:

$$T^{\alpha}_{QR} \geq 2n \log p_r$$
$$T^{\alpha}_{post} \leq 3 \log p_r \leq \frac{3}{2n} T^{\alpha}_{QR}$$
$$T^{\alpha}_{enc} = T^{\alpha}_{f} = \log p_r \leq \frac{1}{2n} T^{\alpha}_{QR}$$

By Remark 5, we know that:

$$T^{\alpha}_{comp} = O(\frac{1}{n}) T^{\alpha}_{QR}$$

Therefore, $T^{\alpha}_{coding} = T^{\alpha}_{enc} + T^{\alpha}_{post} + T^{\alpha}_{c} = O(\frac{1}{n}) T^{\alpha}_{QR}$. □

□