3D Coded SUMMA: Communication-Efficient and Robust Parallel Matrix Multiplication

Haewon Jeong[1], Yaoqing Yang[2], Vipul Gupta[2], Christian Engelmann[3], Tze Meng Low[1], Viveck Cadambe[4], Kannan Ramchandran[2], and Pulkit Grover[1]

[1] Carnegie Mellon University
[2] UC Berkeley
[3] Oak Ridge National Laboratory*
[4] Penn State University

Abstract. In this paper, we propose a novel fault-tolerant parallel matrix multiplication algorithm called *3D Coded SUMMA* that is communication efficient and achieves higher failure-tolerance than replication-based schemes for the same amount of redundancy. This work bridges the gap between recent developments in *coded computing* and fault-tolerance in high-performance computing (HPC). The core idea of coded computing is the same as algorithm-based fault-tolerance (ABFT), which is weaving redundancy in the computation using error-correcting codes. In particular, we show that *MatDot codes*, an innovative code construction for parallel matrix multiplications, can be integrated into three-dimensional SUMMA (Scalable Universal Matrix Multiplication Algorithm [29]) in a communication-avoiding manner. To tolerate any two node failures, the proposed 3D Coded SUMMA requires ∼50 % less redundancy than replication, while the overhead in execution time is only about 5-10 %.

Keywords: Parallel Matrix Multiplication · Fault-tolerant algorithms · Algorithm-based fault tolerance · Coded computing · Communication-efficient algorithms · Error detection and correction

1 Introduction

Upcoming exascale computing systems are expected to bring about new challenges in building resilience against failures and faults [25,4,15,3]. To see how the scale

* This work was sponsored by the U.S. Department of Energy's Office of Advanced Scientific Computing Research. This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (`http://energy.gov/downloads/doe-public-access-plan`).

affects reliability, let us consider the Japanese supercomputer, the Fugaku system that is now being built to be available in 2021. The Fugaku system will have 150,000 physical nodes with a total of 8 million cores [21]. To build a system with mean-time-between-failure (MTBF) of 24-48 hours, the MTBF of each node must be 411-822 years. This can create a huge burden on component manufacturers and the system vendor and provides little-to-no room for unexpected reliability issues that have been experienced in the past, such as bad solder, dirty power, unexpected early wear-out, and so on [16].

The most widely used method for fault tolerance in high-performance computing (HPC) is checkpoint-restart, which saves the state of computation at specific intervals and can recover from detected faults by rolling back to a check-pointed state. While the checkpoint-restart approach is universal, it generates a significant amount of I/O overhead and its efficiency decreases with the increasing system size. The deployment of node-local nonvolatile memory, such as solid state disks, has eased the I/O pressure for checkpoint/restart, but it will not be sufficient in the long run. Another method considered is replication, where the application is executed either in parallel or sequentially multiple times such as triple modular redundancy (TMR) [15,22,14]. Despite the high resource overhead of replication, it has been shown that process replication strategies can outperform traditional checkpoint-restart approaches for a certain range of system parameters [3].

In this paper, we study a different approach called *coded computing* [20,12,32], more widely known as algorithm-based fault-tolerance (ABFT) [10,11] in the HPC community. This approach reduces the overhead of checkpointing or replication by sacrificing universality and designing the redundancy tailored to a specific numerical algorithm. For designing low-overhead redundancy, both ABFT and coded computing utilize error-correcting codes (in short, coding or codes), a tool extensively used in communication or storage systems. While ABFT uses off-the-shelf classical codes and adapts them to practical problems in HPC, coded computing literature studies devising a new code tailor-made for computation by assuming a simple theoretical computing model. These endeavors in coded computing have shown remarkable improvements in the failure tolerance versus memory/computation trade-off, improving over classical codes designed for communication/storage systems. However, due to the simplified models in coded computing that can be unrealistic in practical HPC systems, it is unclear if the new code constructions can be applied in the HPC context. This paper bridges this gap and demonstrates that the new advances in coded computing can be mapped to HPC systems with a careful choice of architecture.

We propose a novel algorithm for robust and communication-efficient parallel matrix multiplication called *3D Coded SUMMA*. In 3D Coded SUMMA, we incorporate MatDot codes (storage-optimal matrix-multiplication codes) [12] with 3D SUMMA (communication-efficient matrix multiplication algorithm) [26]. Applying ABFT to a three-dimensional matrix multiplication algorithm was studied before [23]. Their goal was to apply ABFT within each node to detect/correct soft errors locally. On the other hand, our aim is to construct a coding strategy that can be applied across distributed nodes to recover from node failures, where we cannot recover any partial result from the failed node. We show that MatDot codes can be integrated into 3D SUMMA seamlessly with small communication overhead. The amount of redundancy required in 3D Coded SUMMA is considerably smaller than replication for cases where more than one failure, or where node corruptions (nodes affected by soft errors) are to be tolerated. For instance, to provide resilience against any two node failures, or against a single corruption, 3D Coded SUMMA requires ∼50 % fewer nodes than the baseline replication strategy. To provide resilience against any two node corruptions, 3D Coded SUMMA requires ∼100 % fewer nodes compared to replication. Finally, we show through theoretical and experimental analysis that 3D Coded SUMMA achieves higher failure resilience with small overhead in execution time: 5-7 % more execution time compared to replication on an $8 \times 8 \times 4$ grid of nodes.

2 Background

2.1 3D SUMMA

We introduce 3-dimensional matrix multiplication algorithm, 3D SUMMA. Three-dimensional algorithms for matrix multiplication in which nodes are placed on a 3D grid were proposed [1,24,26] and proved to achieve the optimal communication time in scaling sense [26] under some constraints. 3D SUMMA we present here is an adaptation of 2.5D matrix multiplication algorithm [26]: instead of using Cannon's algorithm on each layer, we use SUMMA on each layer. In this work, for simplicity, we assume that nodes are placed on layers of square grids, *i.e.,* on a $n \times n \times m$ grid where m is the number of layers and n is the layer size. The goal is to compute matrix product:

$$\mathbf{C} = \mathbf{A}\mathbf{B}. \tag{1}$$

We assume matrices $\mathbf{A}, \mathbf{B}, \mathbf{C}$ all have dimension $N \times N$.[5] We use $P(i, j, l)$ to denote the node on the (i, j, l)-th coordinate on the 3D grid.

We summarize 3D SUMMA algorithm below.

1. Matrix product in (1) is split into outer-products as follows:

$$\mathbf{A} = \begin{bmatrix}\mathbf{A}_1 \cdots \mathbf{A}_m\end{bmatrix}, \mathbf{B} = \begin{bmatrix}\mathbf{B}_1 \\ \vdots \\ \mathbf{B}_m\end{bmatrix}, \mathbf{C} = \mathbf{A}_1\mathbf{B}_1 + \cdots + \mathbf{A}_m\mathbf{B}_m, \tag{2}$$

   where $\mathbf{A}_i, \mathbf{B}_i$ $(i = 1, \ldots, m)$ are $N \times N/m$ and $N/m \times N$ dimensional submatrices, respectively.
2. Initially, all $\mathbf{A}_i$'s and $\mathbf{B}_i$'s are stored at the nodes on the first layer of the 3D grid. The first layer scatters $\mathbf{A}_i$ and $\mathbf{B}_i$ to the i-th layer.
3. Each layer performs 2D SUMMA[6] to compute $\mathbf{C}_i = \mathbf{A}_i\mathbf{B}_i$ in parallel.

[5] Throughout the paper, we will assume that m and n divide N for simplicity. In practice, when N is not divisible by m, n, the matrix can be zero-padded to make N divisible by m and n. Also, the assumption that they are square matrices is only for simplicity, and the algorithm can be used for rectangular matrices as well.

[6] For more details on 2D SUMMA, please see [29].

4. All layers reduce to the first layer and the first layer obtains:

$$\mathbf{C} = \mathbf{C}_1 + \cdots + \mathbf{C}_m.$$

2.2 MatDot Codes

MatDot codes [12] are one of the latest advances in coded computing and proven to be optimal in terms of *recovery threshold*[7] for parallel matrix multiplication under certain constraints [32]. Classical error-correcting codes such as Reed-Solomon codes encode data through polynomial evaluations where the coefficients of the polynomial are the raw data. These algorithms use polynomial interpolation for decoding to recover the polynomial coefficients, *i.e.,* the raw data, when the number of evaluations that survive after failures is larger than the degree of the polynomial. The construction of MatDot codes is inspired by this approach, but the polynomials are carefully constructed so that the matrix product can be extracted from the polynomial coefficients at the end of computation. A main innovation is the construction of encoding polynomials $p_{\mathbf{A}}(x)$ and $p_{\mathbf{B}}(x)$ that exploit the sum of outer-product structure in (2):

$$p_{\mathbf{A}}(x) = \sum_{i=1}^{m} \mathbf{A}_i x^{i-1}, \quad p_{\mathbf{B}}(x) = \sum_{j=1}^{m} \mathbf{B}_j x^{m-j}. \tag{3}$$

Note that the co-efficients are placed in reverse order in $p_{\mathbf{B}}(x)$. Then, in 3D SUMMA, the i-th layer will receive encoded versions of matrices:

$$\begin{aligned} \widetilde{\mathbf{A}}_i &= p_{\mathbf{A}}(\alpha_i) = \mathbf{A}_1 + \alpha_i \mathbf{A}_2 + \cdots + \alpha_i^{m-1}\mathbf{A}_m, \\ \widetilde{\mathbf{B}}_i &= p_{\mathbf{B}}(\alpha_i) = \mathbf{B}_m + \alpha_i \mathbf{B}_{m-1} + \cdots + \alpha_i^{m-1}\mathbf{B}_1, \end{aligned}$$

and then compute matrix multiplication on the encoded matrices:

$$\widetilde{\mathbf{C}}_i = \widetilde{\mathbf{A}}_i \widetilde{\mathbf{B}}_i = p_{\mathbf{A}}(\alpha_i) p_{\mathbf{B}}(\alpha_i) = p_{\mathbf{C}}(\alpha_i).$$

The polynomial $p_{\mathbf{C}}(x)$ has degree $2m-2$ and has the following form:

$$p_{\mathbf{C}}(x) = \sum_{i=1}^{m}\sum_{j=1}^{m} \mathbf{A}_i \mathbf{B}_j x^{m-1+(i-j)}. \tag{4}$$

Because of our careful choice of $p_{\mathbf{A}}(x)$ and $p_{\mathbf{B}}(x)$, the coefficient of x^{m-1} in $p_{\mathbf{C}}(x)$ is $\mathbf{C} = \sum_{i=1}^{m} A_i B_i$. Since $p_{\mathbf{C}}(x)$ is a polynomial of degree $2m-2$, its coefficients can be recovered as long as we have evaluations of $p_{\mathbf{C}}(x)$ at any $2m-1$ distinct points. Hence the recovery threshold is $K = 2m-1$. In the context of 3D SUMMA, we need m layers for the uncoded strategy. The recovery threshold $K = 2m-1$ implies that when we have $M = 2m-1+r$ layers, it is

[7] Recovery threshold is one metric to measure the performance of a code, which is the minimum number of workers required to recover the computation output.

guaranteed to tolerate any r failed layers. On the other hand, to tolerate any r failures with replication, we need $M = rm$ layers. This will be further discussed in Section 4.1.

Systematic MatDot codes: A code is called *systematic* if, for the first m layers, the output of the r-th layer is the product $\mathbf{A}_r\mathbf{B}_r$. We refer to the first m layers as *systematic layers*. Having systematic layers is useful because if all the systematic layers complete their computation successfully, there is no need for decoding. Systematic MatDot codes are achieved by using Lagrange polynomials for encoding. Let

$$p_{\mathbf{A}}(x) = \sum_{i=1}^{m} \mathbf{A}_i L_i(x),\ p_{\mathbf{B}}(x) = \sum_{i=1}^{m} \mathbf{B}_i L_i(x), \tag{5}$$

where $L_i(x)$ is defined as: $L_i(x) = \prod_{j\in\{1,\ldots,m\}\setminus\{i\}} \frac{x - x_j}{x_i - x_j}$ for $i \in \{1,\ldots,m\}$. Using these polynomials, the worst-case recovery threshold remains the same as non-systematic MatDot codes [12].

2.3 Related Work in ABFT

Algorithm-based fault tolerance (ABFT) was first proposed by Huang and Abraham to detect and correct errors on circuits during linear algebra operations. Recently, Chen and Dongarra discovered that a similar technique could be used for parallel matrix algorithms for HPC systems [10]. A follow-up work [6] experimentally showed that the overhead of ABFT is less than 12% with respect to the fastest failure-free implementation of PDGEMM (Parallel General Matrix Multiplication). Numerical stability of the ABFT technique was also examined in [8] and applied to soft error detection [9]. The ABFT technique is extended to matrix factorization algorithms such as Cholesky factorization [17] and LU factorization [11,31].

Our work goes beyond existing works in ABFT for HPC as we employ the novel MatDot codes which go beyond traditional error-correcting codes. MatDot codes are designed specifically for distributed matrix multiplication where the matrix product is split into the sum of outer products.

3 3D Coded SUMMA

We propose a failure-resilient and communication-efficient parallel-matrix multiplication algorithm, 3D Coded SUMMA, by integrating MatDot codes into 3D SUMMA. Since 3D SUMMA partitions matrix multiplication into outer products across layers, we can weave MatDot codes into the third dimension (l-axis) of the algorithm.

Recall that the recovery threshold of MatDot codes is $K = 2m-1$. This means that if we have any K successful (non-failed) nodes, we can recover the matrix product $\mathbf{C}$, and thus to tolerate one failure, we need $K + 1 = 2m$ nodes. For failure resilience, we need at least m redundant layers and use a total of $M \geq 2m$ layers. This redundancy is the same as replication for a single failure. A thorough comparison between 3D Coded SUMMA and replication for an arbitrary number of failures will be provided in the next section. In this section, we focus on the algorithm design of 3D Coded SUMMA and demonstrate a simple example of $(n = 2, m = 2, M = 4)$. The full algorithm is given in Algorithm 1.

Example 1 (3D Coded SUMMA for $(n = 4, m = 2, M = 4)$).

Initial Data Distribution: The node $P(i, j, 1)$ initially has $\mathbf{A}_{i,j}$ and $\mathbf{B}_{i,j}$ for $i, j = 1 \cdots 4$ where $\mathbf{A}_{i,j}$'S and $\mathbf{B}_{i,j}$'s are $N/m \times N/m$ sub-blocks as follows:

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{1,1} & \cdots & \mathbf{A}_{1,4} \\ \vdots & \ddots & \vdots \\ \mathbf{A}_{4,1} & \cdots & \mathbf{A}_{4,4} \end{bmatrix}, \mathbf{B} = \begin{bmatrix} \mathbf{B}_{1,1} & \cdots & \mathbf{B}_{1,4} \\ \vdots & \ddots & \vdots \\ \mathbf{B}_{4,1} & \cdots & \mathbf{B}_{4,4} \end{bmatrix} \tag{6}$$

Encoding: To encode MatDot codes, we begin with splitting $\mathbf{A}_{i,j}$ into two equal-sized column blocks and $\mathbf{B}_{i,j}$ into two equal-sized row blocks as follows:

$$\mathbf{A}_{i,j} = \begin{bmatrix} \mathbf{A}_{i,j}^{(1)} & \mathbf{A}_{i,j}^{(2)} \end{bmatrix}, \mathbf{B}_{i,j} = \begin{bmatrix} \mathbf{B}_{i,j}^{(1)} \\ \mathbf{B}_{i,j}^{(2)} \end{bmatrix}. \tag{7}$$

Then, the node $P(i, j, 1)$ locally computes four encoded column-blocks and row-blocks as follows:

$$\begin{aligned}
\widetilde{\mathbf{A}}_{i,j,1} &= \mathbf{A}_{i,j}^{(1)} + \alpha_1 \mathbf{A}_{i,j}^{(2)}, \quad \widetilde{\mathbf{B}}_{i,j,1} = \alpha_1 \mathbf{B}_{i,j}^{(1)} + \mathbf{B}_{i,j}^{(2)}, \\
\widetilde{\mathbf{A}}_{i,j,2} &= \mathbf{A}_{i,j}^{(1)} + \alpha_2 \mathbf{A}_{i,j}^{(2)}, \quad \widetilde{\mathbf{B}}_{i,j,2} = \alpha_2 \mathbf{B}_{i,j}^{(1)} + \mathbf{B}_{i,j}^{(2)}, \\
\widetilde{\mathbf{A}}_{i,j,3} &= \mathbf{A}_{i,j}^{(1)} + \alpha_3 \mathbf{A}_{i,j}^{(2)}, \quad \widetilde{\mathbf{B}}_{i,j,3} = \alpha_3 \mathbf{B}_{i,j}^{(1)} + \mathbf{B}_{i,j}^{(2)}, \\
\widetilde{\mathbf{A}}_{i,j,4} &= \mathbf{A}_{i,j}^{(1)} + \alpha_4 \mathbf{A}_{i,j}^{(2)}, \quad \widetilde{\mathbf{B}}_{i,j,4} = \alpha_4 \mathbf{B}_{i,j}^{(1)} + \mathbf{B}_{i,j}^{(2)},
\end{aligned}$$

where $\alpha_1, \cdots, \alpha_4$ are four distinct real numbers.[8] Then $P(i, j, 1)$ sends $\mathbf{A}_{i,j,k}$ to $P(i, j, k)$ for $k = 2, 3, 4$ using MPI Scatter operation.

After MatDot encoding step, the node $P(i, j, k)$ will have $\mathbf{A}_{i,j,k}$ and $\mathbf{B}_{i,j,k}$ for all $i, j, k = 1, \ldots, 4$.

Computation: Perform 2D SUMMA [29] on each layer in parallel.

Decoding: Any $K = 2m - 1 = 3$ layers out of $M = 4$ layers are sufficient to decode the final output. Instead of performing MPI Reduce on the raw output, each node will scale their output with the decoding coefficients and then perform MPI Reduce. E.g., if $P(i, j, 4)$ fails, $P(i, j, 1), P(i, j, 2), P(i, j, 3)$ will send $d_1 \widetilde{\mathbf{C}}_{i,j,1}, d_2 \widetilde{\mathbf{C}}_{i,j,2}$, and $d_3 \widetilde{\mathbf{C}}_{i,j,3}$, then the first layer will have the final output $\mathbf{C}_{i,j} = d_1 \widetilde{\mathbf{C}}_{i,j,1} + d_2 \widetilde{\mathbf{C}}_{i,j,2} + d_3 \widetilde{\mathbf{C}}_{i,j,3}$.[9] ■

[8] We can also use systematic MatDot codes where $\mathbf{A}_{i,j,1} = \mathbf{A}_{i,j}^{(1)}$ and $\mathbf{A}_{i,j,2} = \mathbf{A}_{i,j}^{(2)}$ by using the polynomials given in (5). However, for simplicity, we only discuss the non-systematic formulation.

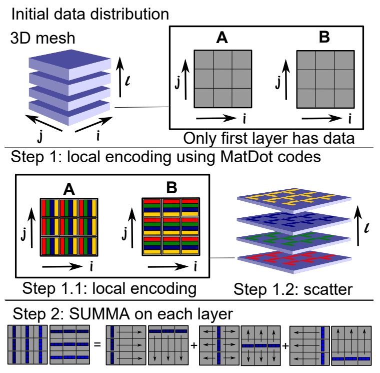
[9] The decoding coefficients, $d_1, \ldots, d_4$ are determined by the choice of $\alpha_1, \ldots, \alpha_4$. For more information on how to compute $d_1, \ldots, d_4$, see [12].

Algorithm 1 3D Coded SUMMA

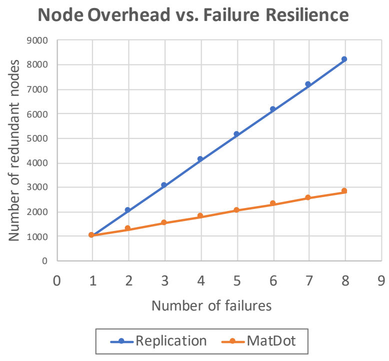
```
1: Initial Data Distribution: $P(i,j,1)$ has $\mathbf{A}_{i,j}$ and $\mathbf{B}_{i,j}$.
2: /* Encoding $\mathbf{A}$, $\mathbf{B}$ and Scattering encoded data */
3: for $i = 1$ to $n$ do
4:     for $j = 1$ to $n$ do
5:         for $l = 1$ to $M$ do
6:             $P(i,j,1)$ computes:   /* All $P(i,j,1)$ in parallel */
                   $\widetilde{\mathbf{A}}_{i,j,l} = \mathbf{A}_{i,j}^{(1)} + \alpha_l \mathbf{A}_{i,j}^{(2)} + \cdots + \alpha_l^{m-1} \mathbf{A}_{i,j}^{(m)}$   (8)
                   $\widetilde{\mathbf{B}}_{i,j,l} = \alpha_l^{m-1} \mathbf{B}_{i,j}^{(1)} + \alpha_l^{m-2} \mathbf{B}_{i,j}^{(2)} + \cdots + \mathbf{B}_{i,j}^{(m)}$   (9)
7:         end for
8:         $P(i,j,1)$ scatters $\widetilde{\mathbf{A}}_{i,j,l}$ and $\widetilde{\mathbf{B}}_{i,j,l}$ to $P(i,j,l)$'s ($l = 1, \ldots, M$)
9:     end for
10: end for
11: /* 2D SUMMA Computation */
12: for $l = 1$ to $m$ do
13:     All $l$-th layers in parallel, perform 2D SUMMA to compute: $\widetilde{\mathbf{A}}_l^{\text{col}} \cdot \widetilde{\mathbf{B}}_l^{\text{row}}$.
14: end for
15: /* Decoding and Reduce to recover $\mathbf{C}$ */
16: for $i = 1$ to $n$ do
17:     for $j = 1$ to $n$ do
18:         for $l = 1$ to $M$ do
19:             /* All $i, j, l$ in parallel */
20:             $P(i,j,l)$ knows which nodes failed among $P(i,j,k)$'s ($k = 1, \ldots, M$).
21:             $P(i,j,l)$ computes $d_l \widetilde{\mathbf{C}}_{i,j,l}$ and reduce to $P(i,j,1)$
22:         end for
23:     end for
24: end for
```

Notice that the encoding of MatDot codes does not require any communication as encoding computation is performed at each local node. There is no additional communication required for MatDot decoding either as the decoding process is embedded in the final reduce step. The only communication cost increase comes from the initial MPI Scatter and the final MPI Reduce with the bigger size, *i.e.,* scatter/reduce over 4 layers instead of 2.

We want to make a remark that we can apply the ABFT technique [18,10] (rediscovered as Product codes in [20]) at each layer of 2D SUMMA for fault tolerance. Although in terms of additional nodes required, ABFT can be more efficient than MatDot codes, for higher failure tolerance, MatDot codes are a more communication-efficient solution. In the encoding of the ABFT strategy, one has to compute linear combinations of the column (row) blocks of $\mathbf{A}$ ($\mathbf{B}$), which requires column (row) shuffling, or multiple reduce operations to parity nodes. Furthermore, for decoding, for recovering from more than one failure, nodes have to perform peeling decoding (see [20]) which can potentially require many rounds


Initial data distribution
3D mesh
A
B
Only first layer has data
Step 1: local encoding using MatDot codes
Step 1.1: local encoding
Step 1.2: scatter
Step 2: SUMMA on each layer

(a)



(b)

Fig. 1: (a) Summary of 3D Coded SUMMA algorithm. (b) Number of redundant nodes required to be resilient to f failures in the node/layer failure scenario for $n = 16, m = 4$.

of communication. However, depending on which resource (communication delay or the number of compute nodes) is more expensive in the system, one can choose between ABFT on each layer and MatDot codes across layers as proposed in the first version of this work [19].

4 Performance Analysis

In this section, we will show how 3D Coded SUMMA can provide higher resilience for the same number of nodes compared to replication. Then, we analyze the overhead of the MatDot-coded strategy in terms of communication and computation time, and prove that the total overhead is negligible when $m = o(n)$. Finally, we demonstrate through experimental evaluations that the total execution time of 3D Coded SUMMA is only about 5-7% more compared to replication.

4.1 Node Overhead vs. Failure Resilience

To analyze the failure resilience, we will consider three different failure scenarios:

1. Node failure: This corresponds to *a fail-stop error* where a node fails and the entire data or intermediate result on the failed node is lost.
2. Layer failure: All nodes on one layer fail at once. This can be relevant when one layer is placed under the same rack and a rack failure occurs.
3. Node corruption: A node is corrupted by a soft error (a bit flip), and an arbitrary amount of data is affected beyond the capability of correction/detection at the local node. This can be due to error propagation during computation.

We say that a strategy is resilient to f failures in a certain failure scenario if we can recover the entire output $\mathbf{C}$ as long as the number of failures is at most

f. We now compare replication and 3D Coded SUMMA for each failure scenario. To be resilient to any f failures in the node failure or the layer failure scenario, the total number of nodes required are the following:

- Replication: $p = (f+1) \cdot mn^2$.
- 3D Coded SUMMA: $p = (2m - 1 + f) \cdot n^2$.

To be resilient to any s failures (*i.e.,* corrupted nodes) in the node corruption scenario, the total number of nodes required are[10]:

- Replication: $p = (2s+1) \cdot mn^2$.
- 3D Coded SUMMA: $p = (2m - 1 + 2s) \cdot n^2$.

Let us make this more concrete by considering an example of $n = 16$ and $m = 4$. To be resilient to any single failure, both replication and 3D Coded SUMMA require 2048 nodes, which is twice more than the uncoded algorithm without any resilience. To be resilient to any two node failures (or any one node corruption), replication requires 3072 nodes while 3D Coded SUMMA requires 2304 nodes. To be resilient to any two node corruptions, replication requires 5120 nodes while 3D Coded SUMMA requires 2816 nodes. Because the recovery threshold of MatDot codes is $K = 2m-1$, there is an upfront cost of 2x node redundancy in 3D Coded SUMMA. However, increasing resilience from one failure to more failures only requires incremental overhead compared to the replication strategy (Fig. 1b).

4.2 Execution Time Analysis

We now analyze the overhead of MatDot coding in terms of its execution time: communication + computation. For communication time, we use the simple α-β model [7]:

$$T_{\text{comm}} = C_1\alpha + C_2\beta, \tag{10}$$

where C_1 is the number of communication rounds and C_2 is the number of bytes communicated on the critical path. The α term is latency cost and the β term is per-byte bandwidth cost. For computation time, we count number of floating-point operations (flops). For 3D Coded SUMMA that encodes an $n \times n \times m$ grid into an $n \times n \times M$ grid using MatDot codes and computes a matrix product of dimension $N \times N$, the communication overhead of MatDot coding is summarized in the following theorem.

Theorem 1. *Suppose we use a MatDot code with a constant rate, i.e., $M = \Theta(m)$. Then, the total communication time of 3D Coded SUMMA is:*

$$T_{comm}^{total} = \left[\alpha\Theta\left(\log n\right) + \beta\Theta\left(N^2/n^2\right)\right] \cdot \frac{n}{m}, \tag{11}$$

and the communication time overhead of MatDot encoding and decoding is:

$$T_{comm}^{MatDot} = \alpha\Theta(\log n) + \beta\Theta(N^2/n^2). \tag{12}$$

[10] Using the recently proposed collaborative decoding [27] might further reduce the number of nodes required for 3D Coded SUMMA, but we use a conservative estimate.

The theorem implies that both the latency and the bandwidth of MatDot encoding/decoding is negligible if $m = o(n)$. Note that this is the same condition for the 3D SUMMA to outperform the 2D version of SUMMA [26].

Proof of Theorem 1. We will analyze the time complexity of each step.

Encoding MatDot codes and scattering the encoded matrices: The first layer has $n \times n$ nodes. Each node has a square matrix of size N^2/n^2. Each local square matrix is partitioned into m small blocks and encoded into M blocks. The M encoded blocks are scattered to M layers (across the l-axis). Both $\mathbf{A}$ and $\mathbf{B}$ need encoding and scattering.

- Local encoding cost: $C_{\text{enc}} = 2N^2/n^2 \cdot M$.
- Communication cost (scatter using recursive-halving [28]): $T_{\text{scatter}} = 2\alpha \log M + 2\beta \frac{N^2}{n^2} \cdot \frac{M}{m}$.

Matrix multiplication with 2D SUMMA: The data on each layer is gathered into n^2/m nodes, i.e., the nodes in each row and column are partitioned into groups of size m and a local data gathering is carried out. Then, SUMMA proceeds in n/m rounds. In each round, one node in each row broadcasts data of size $\frac{N^2}{n^2 m} \cdot m = \frac{N^2}{n^2}$ to the entire row, and similarly for each column. Then, local computation is carried out, which multiplies two matrices of size $N/n \times N/n$.

- Local gathering using recursive-doubling [28]: $T_{\text{gather}} = 2\alpha \log m + \frac{2N^2}{n^2}\beta$.
- Broadcast in SUMMA (scatter using recursive-halving followed by all-gather using recursive-doubling): $T_{\text{bcast}} = (4\alpha \log n + \frac{4N^2}{n^2}\beta) \cdot (n/m)$.
- Local matrix-matrix multiplication: $C_{\text{MxM}} = (N^3/n^3) \cdot (n/m) = \frac{N^3}{n^2 m}$.

Decoding and reduction: The decoding of MatDot codes only requires a reduce across layers. The data size at each node in the reduction phase is still N^2/n^2, and the number of layers required in the reduce is $2m-1$ (for MatDot codes).

- Decoding MatDot codes (reduce using recursive-halving followed by tree-gather [28]): $T_{\text{reduce}} = 2\alpha \log(2m-1) + (2N^2/n^2)\beta$.

Note that this communication cost analysis is the worst-case analysis because if we use *systematic* codes, we only need to reduce the first m systematic layers.

Putting this altogether, we obtain the total communication time as follows:

$$\begin{aligned}
T_{\text{comm}}^{\text{total}} =& T_{\text{scatter}} + T_{\text{gather}} + T_{\text{bcast}} + T_{\text{reduce}} \\
=& 2\alpha \log M + 2\beta \frac{N^2}{n^2} \cdot \frac{M}{m} + 2\alpha \log m + \frac{2N^2}{n^2}\beta \\
& + (4\alpha \log n + \frac{4N^2}{n^2}\beta) \cdot (n/m) + 2\alpha \log(2m-1) + (2N^2/n^2)\beta \\
\overset{(a)}{=}& \left[\alpha \Theta(\log n) + \beta \Theta\left(N^2/n^2\right)\right] \cdot \frac{n}{M},
\end{aligned}$$

where in step (a), we use the fact that $M = \Theta(m)$, *i.e.,* the code has a constant rate. Finally, total communication overhead of using MatDot codes only come from the increased size of gather and reduce operations:

$$\begin{aligned}T_{\text{comm}}^{\text{MatDot}} =& T_{\text{scatter}} + T_{\text{reduce}} \\ =& 2\alpha \log m + \frac{2N^2}{n^2}\beta + 2\alpha \log(2m-1) + (2N^2/n^2)\beta \\ =& \alpha\Theta(\log M) + \beta\Theta(N^2/n^2).\end{aligned}$$

□

Computation time overhead of MatDot coding is summarized below.

Theorem 2. *Suppose we use a MatDot code with a constant rate, i.e.,* $M = \Theta(m)$*. Then,*

$$T_{comp}^{total} = \Theta\left(\frac{N^3}{n^2 m}\right) + \Theta\left(\frac{mN^2}{n^2}\right) + \Theta\left(m^2\right), \tag{13}$$

$$T_{comp}^{MatDot} = \Theta\left(\frac{mN^2}{n^2}\right) + \Theta\left(m^2\right). \tag{14}$$

Notice that the computation time overhead of MatDot coding is negligible when $m = o(\sqrt{N})$, which is often the case since the matrix dimension N is orders of magnitude bigger than the number of layers m.

Proof of Theorem 2. The number of flops required at each local node for each step is given below:

- MatDot encoding: Each node generates M encoded blocks of dimension $N/n \times N/mn$ (or $N/mn \times N/n$), each of which is a linear combinations of m small sub-blocks of the same dimension. Hence,

$$T_{\text{enc}} = 2M \cdot m \cdot \frac{N^2}{mn^2} = \Theta\left(\frac{mN^2}{n^2}\right).$$

- Matrix multiplication: $T_{\text{MxM}} = \Theta\left(\frac{N^3}{n^2 m}\right)$
- MatDot decoding: Each node has to obtain decoding coefficient depending on which nodes have failed through polynomial interpolation, which has computation complexity of at most $\Theta(m^2)$. Then, it scales its output matrix by the decoding coefficient. Thus,

$$T_{\text{dec}} = \Theta(m^2) + \Theta(N^2/n^2).$$

□

4.3 Experimental Evaluation

In this section, we evaluate the performance of 3D Coded SUMMA through experiments. In our experimental setup, we used a cluster with 40 compute nodes, each of which has two 12-Core AMD Opteron (tm) Processor 6164 HE, 64 GB DRAM, and 500 GB hard disk. Nodes are connected through Gigabit Ethernet under a single switch. We used each core as one MPI process, *i.e.,* one core was one logical node $P(i,j,l)$. To ensure that there is no MPI communication within the same compute node, we used cyclic distribution of compute nodes. We injected a layer failure by artificially ignoring the result from one layer in the reduce phase. We assumed that the information about the failed node will be made available at all surviving nodes. We recorded execution time of: memory allocation, MatDot Encoding (line 6 in Algorithm 1), MPI Scatter (line 8 in Algorithm 1), 2D SUMMA (line 12-14 in Algorithm 1), and Decoding + MPI Reduce (line 16-24 in Algorithm 1).

Table 1: Execution time comparison of $(n = 8, m = 2, M = 4)$ 3D Coded SUMMA and replication. We used systematic MatDot codes and 8 cores per node.

N	Strategy	Memory Allocation (s)	Encoding (s)	Scatter (s)	2D SUMMA (s)	Decoding + Reduce (s)	Total (s)
10000	Replication	0.1	0	1.505	19.583	0.926	22.245
	MatDot	0.105	0.124	2.25	18.621	1.384	22.486
20000	Replication	0.369	0	6.574	87.792	3.626	98.681
	MatDot	0.362	0.402	9.075	88.371	5.502	103.357
30000	Replication	0.75	0	14.993	214.798	7.859	239.035
	MatDot	0.752	0.864	19.773	224.232	12.316	257.883
40000	Replication	1.317	0	25.613	438.356	13.941	480.464
	MatDot	1.325	1.418	39.496	440.872	21.853	505.41

Since the cluster we used for experiments had total of 960 cores, the most extensive experiments were run on an $8 \times 8 \times 4$ grid with total of 256 cores[11]. We first compare our proposed MatDot-coded approach and replication. Execution time comparison of the two is summarized in Table 1. First notice that almost 90 % of the total execution time is used in 2D SUMMA operations. Then, the next significant portion of the execution time is MPI Scatter and Reduce. Computation time for MatDot encoding and decoding makes up less than 1% of the total time. When we compare the total execution time, the overhead of MatDot coding is about 5-7 % compared to replication. This is mainly due to the increased communication cost in the scatter and reduce communication as predicted in the previous section. We further compare the total execution time of replication and

[11] Bigger grids with the dimensions of non-power-of-two numbers are not included as they showed worse performance.

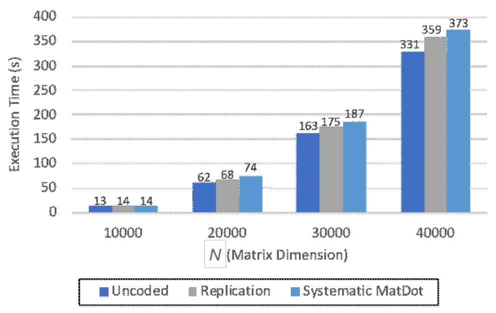


Fig. 2: Comparison of the total execution time between uncoded 3D SUMMA (no resilience), replication, and 3D Coded SUMMA for $(n = 8, m = 2, M = 2)$. We used 16 cores per node.

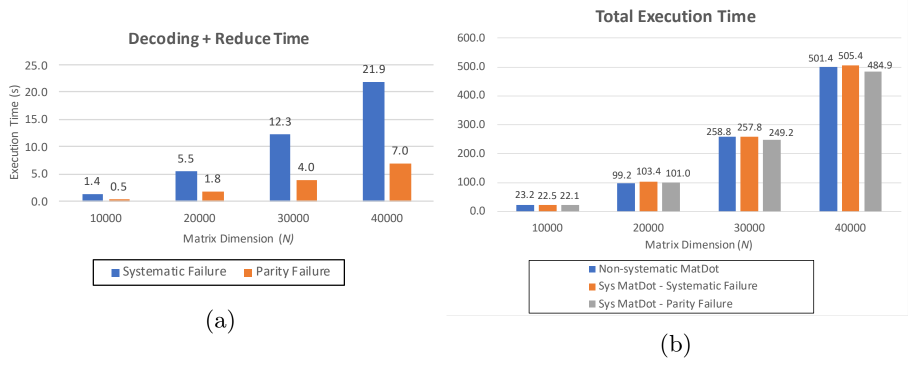


Fig. 3: (a) Comparison of decoding+reduce time using Systematic MatDot codes. When the failed node is a parity node, systematic code is ∼3x faster. (b) Comparison of total execution time for using non-systematic MatDot codes and systematic MatDot codes. For systematic failures, non-systematic and systematic codes share similar performance. For parity failures, systematic codes show a clear advantage.

MatDot against the uncoded counterpart that does not provide any resilience (See Fig. 2). Compared to the uncoded strategy, the execution time of replication is 5-9% higher and 3D Coded SUMMA is about 10-18% higher.

Fig. 3 shows the difference between using systematic and non-systematic codes. In Fig. 3a, systematic failure means a node failure in a systematic layer (the first m layers with the original data) and parity failure means a node failure in a parity layer (the last m layers with encoded data). The biggest benefit of using systematic codes is that when there is no failure in systematic nodes, there is no need for decoding, and the final steps would be no different from the uncoded strategy. The results in Fig. 3a show that this is indeed true in experiments and the last reduce step (including decoding) is about 3x times faster when we have only parity failures, and no systematic failure. Because of this effect, we can see that using systematic codes is about 3-5 % faster than non-systematic codes when there is no systematic failure in Fig. 3b.

5 Discussion and Future Work

In this paper, we examined a new fault-tolerant parallel matrix multiplication algorithm that integrates MatDot codes and 3D SUMMA. In our experiments, we assumed that failure information would be provided to every node. Although the current MPI implementation does not provide such functionality, there have been various research works to incorporate fault mitigation into MPI library [5,2] which include failure reporting and rearranging MPI communicator after the failure. Implementing 3D Coded SUMMA on these prototype fault-tolerant MPI libraries would be interesting future work.

Our work is a first step towards introducing coded computing to HPC applications and showing the feasibility through experiments. We believe that there is an abundance of possibilities in developing practical fault-tolerant algorithms by marrying new developments in coding theory and systems research (see [13] for the recent review in this direction). For instance, our work focuses only on dense matrix multiplication. Extending it to sparse matrix multiplication (e.g., sparse SUMMA) is not a straightforward question since the encoding process would reduce the sparsity of matrices. For linear system solving or eigendecomposition problems, one can consider using the substitute decoding technique for sparse matrices [30].

Acknowledgements

This project was supported partially by NSF grant CCF 1763657 and supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, program managers Robinson Pino and Lucy Nowell. This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy.

References

1. Agarwal, R.C., Balle, S.M., Gustavson, F.G., Joshi, M., Palkar, P.: A three-dimensional approach to parallel matrix multiplication. IBM Journal of Research and Development **39**(5), 575–582 (1995)
2. Ashraf, R.A., Hukerikar, S., Engelmann, C.: Shrink or substitute: Handling process failures in hpc systems using in-situ recovery. In: 2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP). pp. 178–185. IEEE (2018)
3. Benoit, A., Herault, T., Fèvre, V.L., Robert, Y.: Replication is more efficient than you think. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC '19, New York, NY, USA (2019)

4. Bergman, K., Borkar, S., Campbell, D., Carlson, W., Dally, W., Denneau, M., Franzon, P., Harrod, W., Hill, K., Hiller, J., et al.: Exascale computing study: Technology challenges in achieving exascale systems. DARPA Tech. Rep (2008)
5. Bland, W., Bouteiller, A., Herault, T., Bosilca, G., Dongarra, J.: Post-failure recovery of mpi communication capability: Design and rationale. The International Journal of High Performance Computing Applications **27**(3), 244–254 (2013)
6. Bosilca, G., Delmas, R., Dongarra, J., Langou, J.: Algorithmic Based Fault Tolerance Applied to High Performance Computing (2008)
7. Chan, E., Heimlich, M., Purkayastha, A., Van De Geijn, R.: Collective communication: theory, practice, and experience. Concurrency and Computation: Practice and Experience **19**(13), 1749–1783 (2007)
8. Chen, Z.: Optimal real number codes for fault tolerant matrix operations. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. p. 29. ACM (2009)
9. Chen, Z.: Online-ABFT: an online algorithm based fault tolerance scheme for soft error detection in iterative methods. ACM SIGPLAN Notices **48**(8), 167–176 (2013). https://doi.org/10.1145/2442516.2442533
10. Chen, Z., Dongarra, J.: Algorithm-based checkpoint-free fault tolerance for parallel matrix computations on volatile resources. Proceedings 20th IEEE International Parallel & Distributed Processing Symposium (2006)
11. Davies, T., Karlsson, C., Liu, H., Ding, C., Chen, Z.: High performance linpack benchmark: a fault tolerant implementation without checkpointing p. 162–171 (2011). https://doi.org/10.1145/1995896.1995923
12. Dutta, S., Fahim, M., Haddadpour, F., Jeong, H., Cadambe, V., Grover, P.: On the optimal recovery threshold of coded matrix multiplication. IEEE Transactions on Information Theory **66**(1), 278–301 (2019)
13. Dutta, S., Jeong, H., Yang, Y., Cadambe, V., Low, T.M., Grover, P.: Addressing unreliability in emerging devices and non-von neumann architectures using coded computing. Proceedings of the IEEE (2020)
14. Engelmann, C., Ong, H.H., Scott, S.L.: The case for modular redundancy in large-scale high performance computing systems. In: Proceedings of the 8th IASTED international conference on parallel and distributed computing and networks (2009)
15. Ferreira, K., Stearley, J., Laros III, J.H., Oldfield, R., Pedretti, K., Brightwell, R., Riesen, R., Bridges, P.G., Arnold, D.: Evaluating the viability of process replication reliability for exascale systems. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (2011)
16. Geist, A.: How to kill a supercomputer: Dirty power, cosmic rays, and bad solder. IEEE Spectrum **10**, 2–3 (2016)
17. Hakkarinen, D., Chen, Z.: Algorithmic cholesky factorization fault recovery. In: 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS). pp. 1–10. IEEE (2010)
18. Huang, K.H., Abraham, J.A.: Algorithm-Based Fault Tolerance for Matrix Operations. IEEE Transactions on Computers **C-33**(6), 518–528 (1984)
19. Jeong, H., Yang, Y., Gupta, V., Grover, P., Ramchandran, K.: Coded SUMMA: Fully-decentralized coded matrix multiplication for high performance computing. `http://www.andrew.cmu.edu/user/haewonj/documents/codml19_full_summa.pdf` (2019)
20. Lee, K., Lam, M., Pedarsani, R., Papailiopoulos, D., Ramchandran, K.: Speeding up distributed machine learning using codes. IEEE Transactions on Information Theory **64**(3), 1514–1529 (2017)

21. Limited, F.: Fujitsu begins shipping supercomputer fugaku. press release (2019)
22. Lyons, R.E., Vanderkulk, W.: The use of triple-modular redundancy to improve computer reliability. IBM journal of research and development **6**(2), 200–209 (1962)
23. Moldaschl, M., Prikopa, K.E., Gansterer, W.N.: Fault tolerant communication-optimal 2.5 d matrix multiplication. Journal of Parallel and Distributed Computing **104**, 179–190 (2017)
24. Schatz, M.D., Van de Geijn, R.A., Poulson, J.: Parallel matrix multiplication: A systematic journey. SIAM Journal on Scientific Computing **38**(6) (2016)
25. Shalf, J., Dosanjh, S., Morrison, J.: Exascale computing technology challenges. In: International Conference on High Performance Computing for Computational Science. pp. 1–25. Springer (2010)
26. Solomonik, E., Demmel, J.: Communication-optimal parallel 2.5 d matrix multiplication and lu factorization algorithms. In: European Conference on Parallel Processing. pp. 90–109. Springer (2011)
27. Subramaniam, A.M., Heiderzadeh, A., Narayanan, K.R.: Collaborative decoding of polynomial codes for distributed computation. In: 2019 IEEE Information Theory Workshop (ITW). pp. 1–5 (2019)
28. Thakur, R., Rabenseifner, R., Gropp, W.: Optimization of collective communication operations in mpich. The International Journal of High Performance Computing Applications **19**(1), 49–66 (2005)
29. Van De Geijn, R.A., Watts, J.: Summa: Scalable universal matrix multiplication algorithm. Concurrency: Practice and Experience **9**(4), 255–274 (1997)
30. Yang, Y., Grover, P., Kar, S.: Coding for a single sparse inverse problem. In: 2018 IEEE International Symposium on Information Theory (ISIT). pp. 1575–1579 (2018)
31. Yao, E., Zhang, J., Chen, M., Tan, G., Sun, N.: Detection of soft errors in LU decomposition with partial pivoting using algorithm-based fault tolerance. The International Journal of High Performance Computing Applications **29**(4), 422–436 (2015). https://doi.org/10.1177/1094342015578487
32. Yu, Q., Maddah-Ali, M.A., Avestimehr, A.S.: Straggler mitigation in distributed matrix multiplication: Fundamental limits and optimal coding. In: IEEE International Symposium on Information Theory (ISIT). pp. 2022–2026 (2018)