Fault-Tolerant Parallel Integer Multiplication

Roy Nissim
roynissim@cs.huji.ac.il
The Hebrew University of Jerusalem
Department of Computer Science
Jerusalem, Israel

Oded Schwartz
odedsc@cs.huji.ac.il
The Hebrew University of Jerusalem
Department of Computer Science
Jerusalem, Israel

Yuval Spiizer
lahav.yuval@gmail.com
The Hebrew University of Jerusalem
Department of Computer Science
Jerusalem, Israel

ABSTRACT

Exascale machines have a small mean time between failures, necessitating fault tolerance. Out-of-the-box fault-tolerant solutions, such as checkpoint-restart and replication, apply to any algorithm but incur significant overhead costs. Long integer multiplication is a fundamental kernel in numerical linear algebra and cryptography. The naïve, schoolbook multiplication algorithm runs in $\Theta\left(n^2\right)$ while Toom-Cook algorithms runs in $\Theta\left(n^{\log_k(2k-1)}\right)$ for $2 \leq k$. We obtain the first efficient fault-tolerant parallel Toom-Cook algorithm. While asymptotically faster FFT-based algorithms exist, Toom-Cook algorithms are often favored in practice on small scale and on supercomputers. Our algorithm enables fault tolerance with negligible overhead costs. Compared to existing, general-purpose, fault-tolerant solutions, our algorithm reduces the arithmetic and communication (bandwidth) overhead costs by a factor of $\Theta\left(\frac{P}{(2k-1)}\right)$ (where P is the number of processors). To this end, we adapt the fault-tolerant BFS-DFS method of Birnbaum et al. (2020) for fast matrix multiplication and combine it with a coding strategy tailored for Toom-Cook. This eliminates the need for recomputations, resulting in a much faster algorithm.

CCS CONCEPTS

• **Theory of computation** → **Parallel algorithms**; *Communication complexity*; • **Mathematics of computing** → *Coding theory*.

KEYWORDS

Fault Tolerance, Long Integer Multiplication, Toom-Cook, Parallel Computing, I/O Complexity

ACM Reference Format:
Roy Nissim, Oded Schwartz, and Yuval Spiizer. 2024. Fault-Tolerant Parallel Integer Multiplication. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2024), June 17–21, 2024, Nantes, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/XXXXXX.XXXXXX

SPAA 2024, June 17–21, 2024, Nantes, France.

ACM ISBN 979-8-4007-0416-1/24/06
https://doi.org/10.1145/XXXXXX.XXXXXX

1 INTRODUCTION

Long integer multiplication algorithms are fundamental computational kernels in numerous applications, ranging from cryptographic systems to neural networks. They serve as primitives for many elementary functions, including power, square root, and greatest common divisor. Toom-Cook algorithms are often used in polynomial multiplication as well. The naïve, schoolbook multiplication algorithm has a complexity of $\Theta\left(n^2\right)$, where $n \in \mathbb{N}$ is the size of the input. Faster algorithms, such as the Toom-Cook family, are based on polynomial convolution and scales as $O\left(n^{\log_k(2k-1)}\right)$ [16, 40, 72], where $2 \leq k \in \mathbb{N}$ is a parameter of the algorithm. Asymptotically faster FFT-based methods (cf. [22, 27, 33]) exist. However, they often suffer from large hidden constants and limited applicability. Thus, Toom-Cook algorithms are often favored for a large range of inputs (cf. [11, 29–31, 41, 44, 48, 58, 85]), and are widely used by major libraries (cf. [29]).

A major scalability hurdle is faults. The evolving hardware landscape, characterized by machine up-scaling and reduced operating voltage, has led to an increased susceptibility to errors. Exascale machines have small mean time between failures [3, 12, 68], emphasizing the urgent need for fault-tolerant solutions. Faults are categorized into two: i) hard faults - when a processor stops working, and ii) soft faults - when a processor miscalculates. Delay faults, in which the processor's average time per arithmetic operation increases, are sometimes addressed as a third category. We focus here on the first category, hard faults. Our algorithm can easily be adapted for soft faults (see Section 7 for more details).

The standard solutions for dealing with hard faults are checkpoint-restart (cf. [25, 42, 59, 62, 79]), which periodically saves data and state and reverts to the last checkpoint upon error detection, and replication (cf. [1, 2, 13, 20, 28, 38, 75–77]), which divides computations into tasks and run them simultaneously on multiple processors. These solutions are general purpose and easy to employ but entail significant overheads and exhibit poor resource utilization, even in the absence of errors. More efficient, algorithm-specific solutions have emerged, known as algorithm-based fault-tolerant techniques (cf. [8–10, 23, 26, 32, 35, 36, 39, 45–47, 49–51, 54, 55, 57, 64, 67, 69, 71]). These specialized approaches leverage specific knowledge of the algorithm's structure to achieve substantial speedups compared to general-purpose solutions. This paper delves into constructing an algorithm-based, high-performance, fault-tolerant algorithm for parallel fast integer multiplication.

1.1 Previous Work

Long Integer Multiplication: A Toom-Cook algorithm is defined by a split number $k \in \mathbb{N}$, a set of evaluation points S, and a sequence of elementary operations called inversion sequence, which specify

Table 1: Fault-tolerant solutions for Toom-Cook algorithm in the unlimited memory case $\left(M = \Omega\left(\frac{n}{P^{\log_{(2k-1)} k}}\right)\right)$.

Algorithm	Arithmetic cost	Bandwidth cost	Latency cost	Fault Tolerance	Additional Processors
Parallel Toom-Cook [[18], extended here]	$F = \Theta\left(\frac{n^{\log_k (2k-1)}}{P}\right)$	$BW = \Theta\left(\frac{n}{P^{\log_{(2k-1)} k}}\right)$	$L = \Theta(\log P)$	–	–
Toom-Cook with Replication [analyzed here]	F	$(1 + o(1)) \cdot BW$	$(1 + o(1)) \cdot L$	f	$f \cdot P$
Fault-Tolerant Toom-Cook [here]	$(1 + o(1)) \cdot F$	$(1 + o(1)) \cdot BW$	$(1 + o(1)) \cdot L$	f	$f \cdot (2k - 1)$

F, BW, and L denote the arithmetic bandwidth and latency costs of the parallel Toom-Cook algorithm with unlimited memory. P denotes the number of processors, n denotes the input size, and f denotes the number of faults the algorithm can tolerate. The first row is for our parallel Toom-Cook algorithm, which is not fault-tolerant. The second row presents the parallel Toom-Cook with the Replication solution. The third row presents our fault-tolerant parallel Toom-Cook algorithm. See Section 5 for the full analysis.

Table 2: Fault-tolerant solutions for Toom-Cook algorithm in the limited memory case $\left(M = O\left(\frac{n}{P^{\log_{(2k-1)} k}}\right)\right)$.

Algorithm	Arithmetic cost	Bandwidth cost	Latency cost	Fault Tolerance	Additional Processors
Parallel Toom-Cook [[18], extended here]	$F = \Theta\left(\frac{n^{\log_k (2k-1)}}{P}\right)$	$BW = \Theta\left(\left(\frac{n}{M}\right)^{\log_k (2k-1)} \cdot \frac{M}{P}\right)$	$L = \Theta\left(\left(\frac{n}{M}\right)^{\log_k (2k-1)} \cdot \frac{\log P}{P}\right)$	–	–
Toom-Cook with Replication [analyzed here]	F	$(1 + o(1)) \cdot BW$	$(1 + o(1)) \cdot L$	f	$f \cdot P$
Fault-Tolerant Toom-Cook [here]	$(1 + o(1)) \cdot F$	$(1 + o(1)) \cdot BW$	$(1 + o(1)) \cdot L$	f	$f \cdot (2k - 1)$

F, BW, and L denote the arithmetic bandwidth and latency costs of the parallel Toom-Cook algorithm with limited memory M. The parameters P, n, and f are defined as in Table 1. The rows of this table are similar to Table 1. See Section 5 for the full analysis. See Section 5 for the full analysis.

how to invert the interpolation matrix. Extended versions of the Toom-Cook family include algorithms with an unbalanced splitting of the input integers, such as Toom-Cook-$(3, 2)$ (also known as Toom-Cook-2.5) [85] or in general, Toom-Cook-(k_1, k_2). Several sets of evaluation points have been proposed for Toom-Cook algorithms (cf. [16, 72, 84, 86]), where the most commonly used set for Toom-Cook-3 is $\{0, 1, -1, 2, \infty\}$ (cf. [11, 22, 31, 63, 84, 86]).

Many studies have focused on optimizing the Toom-Cook algorithm. Bodrato and Zanoni [11] proposed the Toom-Graph technique, a heuristic to find a fast inversion sequence relative to the cost of different elementary linear operations. Zanoni [85] proposed a technique to compute the evaluation stage faster by reusing the outcome of repeated computations. Bermudo et al. Karmakar, and Verbauwhede [6] proposed the "Lazy Interpolation" method, which uses pre-computations to reduce repeated computations. Other studies addressed performance enhancement in diverse platforms (cf. [11, 60, 84]).

Bilardi and De Stefani [7] introduced lower bounds on the I/O complexity of Toom-Cook algorithms in sequential and parallel settings. De Stefani [19] later extended these bounds to hybrid algorithms, combining standard and fast integer multiplication algorithms. Moreover, De Stefani [18] presented near tight upper bounds for parallel schoolbook multiplication and parallel Karatsuba's (Toom-Cook-2) integer multiplication. His parallel algorithm is based on the BFS-DFS parallelization method of [4, 21, 52, 56]

Algorithm-based fault-tolerance: Huang and Abraham [35] introduced an algorithm-based fault-tolerant solution for classical matrix multiplication. Further refinements include [9, 14, 15, 57]. Erasure-based coding strategies have been applied to matrix-vector and classical matrix-matrix multiplication tasks (cf. [23, 45–47, 64, 67]). Polynomial codes are often favored in coded matrix-matrix multiplication (cf. [24, 34, 61, 70, 78, 82, 83]). Despite their effectiveness in these areas, such methodologies typically do not apply to recursive algorithms. Birnbaum et al. [8] proposed a fault-tolerant algorithm for fast matrix multiplication based on the BFS-DFS parallelization technique (cf. [4, 21, 52, 56]). Their coding strategy enables efficient fault-tolerance in the encoding and decoding phases of fast matrix multiplication, but requires expensive recomputations for faults in the multiplication phase.

1.2 Our Contribution

We propose high-performance fault-tolerant parallel Toom-Cook algorithms. Our algorithms reduce the arithmetic and bandwidth overhead costs by a factor of $\Theta(P)$ compared to existing general-purpose alternatives (where P is the number of processors). We summarize our results in Tables 1,2. We start by extending the parallel Toom-Cook-2 of [18] to the general case (Toom-Cook-k). We use the BFS-DFS parallelization technique (cf. [4, 21, 52, 56]) to deal with the recursive nature of Toom-Cook. We apply a mixed coding strategy that combines the linear coding technique of [8] with a polynomial coding technique tailored to the unique structure of Toom-Cook. More precisely, we apply a linear erasure code to deal with faults in the evaluation and interpolation phases (similar to [8]). Linear codes are not generally preserved through the multiplication phase and hence, require expensive recomputations

(cf. [8]). Instead, we propose a polynomial code that integrates with the Toom-Cook's structure in the form of redundant interpolation points.

1.3 Paper Organization

Section 2 provides preliminaries regarding our machine model and Toom-Cook algorithm. In Section 3, we present the parallel Toom-Cook algorithm. In Section 4, we propose a fault-tolerant parallel Toom-Cook algorithm. Section 5 analyzes and compares our algorithm with existing ones. Finally, Section 7 discusses our results and future work.

2 PRELIMINARIES

2.1 Model and architecture

We consider a parallel model (cf. [5, 37, 74]), where a peer-to-peer communication network connects P identical processors. Each processor holds a local memory of size M (words). We denote by F, BW, and L the number of arithmetic operations, words, and messages, counted along the *critical path* as defined in [81]. We model the total run-time by $C = \alpha \cdot L + \beta \cdot BW + \gamma \cdot F$, where α, β, and γ are the latency for a single message, the bandwidth cost for a single word, and the time of a single arithmetic operation, respectively. We denote by f the maximum number of faults the algorithm can tolerate. We address hard errors wherein, upon a fault occurrence, the affected processor ceases operation, loses its data, and is subsequently replaced by an alternative processor.

2.2 Toom-Cook algorithms

Toom-Cook [16, 40, 72] is a family of fast integer multiplication algorithms based on polynomial multiplication. It is often applied recursively and on long integers. Toom-Cook involve three main stages: evaluation, multiplication and interpolation. Toom-Cook-k works as follows: Let $a, b \in \mathbb{N}$ denote two $n \in \mathbb{N}$ bit integers. The two inputs are split into k-digit numbers $a(B) = a_0, \cdots, a_{k-1}$, $b(B) = b_0, \cdots, b_{k-1}$ using a shared base B where

$$B = 2^{\max\left(\left\lfloor \frac{\log_2 a}{k} \right\rfloor, \left\lfloor \frac{\log_2 b}{k} \right\rfloor\right)+1}$$

Let p_a,p_b denote the $k-1$ degrees homogeneous polynomials with $a(B), b(B)$ as coefficients:

$$p_a(x,h) = a_0 h^{k-1} + a_1 x h^{k-2} + \cdots + a_{k-1} x^{k-1}$$
$$q_b(x,h) = b_0 h^{k-1} + b_1 x h^{k-2} + \cdots + b_{k-1} x^{k-1}$$

Notice that $p_a(B,1) = a_0 + a_1 B + \cdots + a_{k-1} B^{k-1} = a$, and similarly $p_b(B,1) = b$. Thus, $a \cdot b = p_a(B,1) \cdot p_b(B,1) = (p_a \cdot p_b)(B,1)$. The polynomial $r_{a,b}(\cdot) = p_a(\cdot) \cdot q_b(\cdot)$ is of degree $2k-2$ and hence, one can compute its coefficients using $2k-1$ evaluation points. The two polynomials, p_a and p_b, are therefore evaluated at $2k-1$ points, and the two evaluations at each point are multiplied by each other. We then use interpolation to obtain the coefficients of $(p_a \cdot p_b)$. Finally, we evaluate $p_a \cdot p_b$ at the point $(B,1)$, to obtain the product $a \cdot b$.

Theorem 2.1 (Interpolation Theorem [16, 72]). *Let $k \in \mathbb{N}$ and S be a set of k distinct evaluation points. The k-evaluation matrix of S is invertible.*

The bilinear form of Toom-Cook-k with the evaluating points $\{(x_i, h_i)\}_{i=1}^{2k-1}$ is defined as follows. Let $U, V \in \mathbb{Q}^{(2k-1)\times k}$ and $W \in \mathbb{Q}^{(2k-1)\times(2k-1)}$ be the following matrices:

$$U = V = \begin{pmatrix} h_0^{k-1}x_0^0 & h_0^{k-2}x_0^1 & \cdots & h_0^0 x_0^{k-1} \\ h_1^{k-1}x_1^0 & h_1^{k-2}x_1^1 & \cdots & h_1^0 x_1^{k-1} \\ \vdots & \vdots & \ddots & \vdots \\ h_{2k-2}^{k-1}x_{2k-2}^0 & h_{2k-2}^{k-2}x_{2k-2}^1 & \cdots & h_{2k-2}^0 x_{2k-2}^{k-1} \end{pmatrix}$$

$$(W^T)^{-1} = \begin{pmatrix} h_1^{2k-2}x_1^0 & h_1^{2k-3}x_1^1 & \cdots & h_1^0 x_1^{2k-2} \\ h_2^{2k-2}x_2^0 & h_2^{2k-3}x_2^1 & \cdots & h_2^0 x_2^{2k-2} \\ \vdots & \vdots & \ddots & \vdots \\ h_{2k-1}^{2k-2}x_{2k-1}^0 & h_{2k-1}^{2k-3}x_{2k-1}^1 & \cdots & h_{2k-1}^0 x_{2k-1}^{2k-2} \end{pmatrix}$$

Then, the bilinear form of Toom-Cook-k is defined by Algorithm 1.

Algorithm 1 Recursive Toom-Cook-k Algorithm $\langle U, V, W \rangle$

Input: $a, b \in \mathbb{Z}$
Output: $c \in \mathbb{Z}$ where $c = a \cdot b$
Parameters: $s \in \mathbb{N}$ is the hardware's max integer operation size, namely $a \cdot b$ can be computed using one operation if $|a|, |b| \leq s$.
Split a and b into k digits numbers using a shared base B.
Let $\overline{a}, \overline{b} \in \mathbb{Z}^k$ be the k digits of a and b, respectively.
$a' = U \cdot \overline{a}$ *// Evaluating p_a*
$b' = V \cdot \overline{b}$ *// Evaluating p_b*
for $i = 1$ to $2k-1$ **do**
  if $[a']_i > s$ or $[b']_i > s$ **then**
    Recursively compute $[c']_i = [a']_i \cdot [b']_i$
  else
    $[c']_i = [a']_i \cdot [b']_i$
  end if
end for
$c' = W^T \cdot c'$
$c = \sum_{i=1}^{2k-1} ([c']_i B^{i-1})$ *// Compute the carry*
return c

Remark 2.2. *We use the homogeneous notation proposed by Zanoni [84] which bypasses the declaration of an ∞ evaluation point, commonly used in the non-homogeneous notation. Both notations describe the same algorithm.*

Bodrato and Zanoni [11] introduced the Toom-Graph technique, which optimizes the interpolation phase by finding the fastest sequence of linear operations that is equivalent to multiplying by W^T. The method utilizes the property that W^T is an invertible matrix and thus finds a short sequence of row operations that maps $(W^T)^{-1}$ to the identity matrix.

Definition 2.3 (Toom-Graph [11]). *Let $W \in F^{k\times k}$ denote the interpolation matrix of Toom-Cook-k over some field F. The Toom-Graph G of W is a weighted graph where: i) each vertex is a matrix in $F^{k\times k}$, and ii) there is an edge from a vertex u to a vertex v if and only if there is a row operation e such that $eu = v$. The weight of each edge is the cost of the row operation. Let I denote an inversion sequence of W, namely, a sequence of elementary operations specifying how to invert the interpolation matrix. Then the operations in I are equivalent to a path in the G from the vertex corresponding to W^{-1} to the vertex corresponding to the identity matrix. The algorithm finds the sequence with the minimal aggregated cost.*

2.3 Toom-Cook with Lazy Interpolation

Bermudo et al. [6] proposed the Lazy Interpolation technique, a variation of the Toom-Cook algorithm that enables predetermination of the mid-computations (see Algorithm 2 for a pseudo-code). In the standard implementation, the values of Toom-Cook's sub-problems cannot be predetermined due to carry computations. In the Lazy Interpolation technique, the carry computations are postponed to the end of the recursion by splitting the input for all recursive steps in advance. As Bermudo et al. showed, this does not affect the algorithm's correctness or change its arithmetic complexity.

Algorithm 2 Recursive Toom-Cook-k Algorithm $\langle U, V, W\rangle$ With Lazy Interpolation

Input: $a, b \in \mathbb{Z}$
Output: $c \in \mathbb{Z}$ where $c = a \cdot b$
Parameters: $s \in \mathbb{N}$ is the hardware's max integer operation size, namely $a \cdot b$ can be computed using one operation if $|a|, |b| \leq s$. The parameters $n, l \in \mathbb{N}$ are $n = \max\left(\log_s a, \log_s b\right)$, and $l = \log_k n$.
Split a and b into k^l digits numbers using a shared base s.
Let $\overline{a}, \overline{b} \in \mathbb{Z}^{k^l}$ be the k^l digits of a and b, respectively.
$a' = U \cdot \overline{a}$ *// For every $j = 1, \ldots, k$, the sequence of $(\overline{a}[k^{l-1} \cdot (j-1)+1], \ldots, k^{l-1}j$ is a block and the multiplication is between a matrix and a block vector.*
$b' = V \cdot \overline{b}$ *// Same as above.*
for $i = 1$ to $2k-1$ **do**
  if $l > 1$ **then**
    Recursively compute $[c']_i = [a']_i \cdot [b']_i$
  else
    $[c']_i = [a']_i \cdot [b']_i$
  end if
end for
$c' = W^T \cdot c'$
$c = \sum_{i=1}^{2k-1} ([c']_i B^{i-1})$ *// Compute the carry*
return c

We next show that recursive Toom-Cook-k with Lazy Interpolation with recursive depth l is equivalent to multivariate polynomial multiplication.

Claim 2.1. *Let A be a Toom-Cook-k algorithm with lazy interpolation and the evaluation points $S = \{(x_i, h_i)\}_{i\leq 2k-1}$. Denote by A^l the algorithm with l recursive depth. Then, A^l computes a multiplication between two polynomials with l variables such that each variable has a power of at most $k-1$. In addition, the algorithm A^l uses S^l as evaluation points.*

Proof. Let $i \leq l$. Denote by y_I the variable used for the split in the $i'th$ depth of the recursion in A^l. Then, each coefficient of y_I is split into k coefficients for the variable y_{i+1}. Thus, A^l splits each input into a multivariate polynomial such that the power of every variable is at most $k-1$. The evaluation points are S^l since in every recursive depth i, for each assignment of $y_1, \ldots, y_{i-1}$, the variable y_i is assigned all the elements of S. □

Multivariate polynomial multiplication has evaluation points in F^l. In the following claim, we describe a sufficient condition for constructing evaluation points in F^l.

Definition 2.4. *Let $l, r \in \mathbb{N}$. Denote by $Poly_{r,l}$ be the set of polynomials with l variables such that the power of the i'th variable in each monomial is no more than $r-1$.*

Claim 2.2. *Let $l, k, n \in \mathbb{N}$. Let S be a set of n evaluation points for multivariate polynomials with l variables. The set S produces a bilinear multivariate polynomial multiplication algorithm between elements in $Poly_{k,l}$ if and only if the evaluation matrix for the product polynomial is injective.*

To prove Claim 2.2 we need the following claim:

Claim 2.3. *Let Let $l, k, n \in \mathbb{N}$. Let S be a set of n evaluation points for multivariate polynomials with l variables. Let E be the evaluation map of S for polynomials in $Poly_{2k-1,l}$. Let*

$$Poly'_{2k-1,l} = \{p \in Poly_{2k-1,l} | \exists a, b \in Poly_{k,l} \textbf{ s.t. } p = a \cdot b\}$$

Let $E|_{Poly'_{2k-1,l}}$ be the map E reduced to $Poly'_{2k-1,l}$. If there is $\langle U, V, W\rangle$ bilinear multivariate polynomial multiplication algorithm such that U and V are the evaluation matrix of S for elements in $Poly_{k,l}$, then $W^T \circ E|_{Poly'_{2k-1,l}}$ is the identity map.

Proof. Let $p \in Poly'_{2k-1,l}$. Let $a, b \in Poly_{k,l}$ be polynomials such that $a \cdot b = p$. Let $\overline{a}, \overline{b}$, and $\overline{p}$ be the corresponding coefficient vectors of a, b, and p, respectively. Since U and V are the evaluation matrix of S and $p = a \cdot b$, then $(U\overline{a}) \odot (V\overline{b})$ is the evaluation of p at the points S, where $\odot$ is the Hadamard product. Thus,

$$\begin{aligned} W^T(E|_{Poly'_{2k-1,l}}(p)) &= W^T((U\overline{a}) \odot (V\overline{b})) \\ &= ALG_{\langle U,V,W\rangle}(a, b) \\ &= a \cdot b = p \end{aligned}$$

□

Next we prove Claim 2.2.

Proof of Claim 2.2. Assume that S produces a bilinear multivariate polynomial multiplication algorithm $\langle U, V, W\rangle$. Let E be the evaluation matrix of S for the product polynomial. Namely, the matrix E is the evaluation map of S for polynomials in $Poly_{2k-1,l}$. Assume by contradiction that E is not injective. Then there is a vector $\overline{p}$ such that $E \cdot \overline{p} = 0$. Let p be the polynomial that its coefficient vector is $\overline{p}$. Let I be the set of all the subsets of $\{1, \ldots, l\}$. For each

$I \in A_l$ let $X_I = \prod_{i \in I} x_i^{k-1}$ where $x_1, \ldots, x_l$ are the variables. For each $I \in A_l$ let $a_I \in Poly(k, l)$ be such that

$$p = \sum_{I \in A_l} (a_I \cdot X_I)$$

Since for each $I \in A_l$, a_I, X_I are in $Poly(k, l)$:

$$\begin{aligned} 0 = E(p) &= E(\sum_{I \in A_l} (a_I \cdot X_I)) \\ &= \sum_{I \in A_l} E(a_I \cdot X_I) \\ &= \sum_{I \in A_l} E|_{Poly'_{2k-1,l}}(a_I \cdot X_I) \end{aligned}$$

According to Claim 2.3,

$$\begin{aligned} 0 = W^T \cdot 0 &= W^T \sum_{I \in A_l} E|_{Poly'_{2k-1,l}}(a_I \cdot X_I) \\ &= \sum_{I \in A_l} W^T E|_{Poly'_{2k-1,l}}(a_I \cdot X_I) \\ &= \sum_{I \in A_l} (a_I \cdot X_I) = p \end{aligned}$$

Thus, E is injective.

For the other direction, let E be the evaluation map of S for polynomials in $Poly_{2k-1,l}$. Assume that E is injective. Thus, there is a map W^T such that $W^T \circ E$ is the identity map. Let U, V be the evaluation matrix of S for polynomials in $Poly_{k,l}$. Let $a, b \in Poly_{k,l}$. The values $U(a)$ and $V(b)$ are the evaluations of a and b at the points in S, respectively. Thus, $U(a) \odot V(b)$ is the evaluation of $a \cdot b$ at points in S, where $\odot$ is the Hadamard product. Namely, $U(a) \odot V(b)$ equals $E(a \cdot b)$. Therefore,

$$W^T(U(a) \odot V(b)) = W^T \circ E(a \cdot b) = a \cdot b$$

Therefore, $\langle U, V, W \rangle$ is a multivariate polynomial multiplication algorithm produced by S. □

2.4 Collective communication operations

Collective communication operations, such as broadcast and reduce, are frequently used in parallel algorithms. Sanders and Sibeyn [66] introduced an efficient way to perform collective communicators. Birnbaum and Schwartz [9] extended their results to multiple simultaneous reduce operations (referred to as t-reduce or all-reduce when $t = P$). Their technique can easily be generalized to broadcast operation (we similarly denote t-broadcast and all-broadcast). We summarize the results in the following lemmas:

Lemma 2.5 (t-reduce ([9])). *t simultaneous reduce operations on data of size W between P processors cost: $F = t \cdot W$, $BW = t \cdot W$, and $L = O(\log P + t)$.*

Corollary 2.6 (t-broadcast). *t simultaneous broadcast operations on data of size W between P processors cost: $F = 0$, $BW = t \cdot W$, and $L = O(\log P)$.*

2.5 Linear erasure code

Definition 2.7. *An (n, k, d)-code is a linear transformation $T : \mathbb{R}^k \to \mathbb{R}^n$ with distance d, where distance d means that for every $x \neq y$ in $\mathbb{R}^k$, $T(x)$ and $T(y)$ have at least d coordinates with different values in $\mathbb{R}^n$. The generator matrix of T is an $n \times k$ matrix G such that $T(x) = G \cdot x$.*

A systematic code preserves the original word and adds redundant letters. Formally it codes a word x of length k to a word y of length n using $n - k$ additional letters such that $y_{k+i} = \sum_{j=1}^{n} E_{i,j} \cdot x_j$ for some $(n - k) \times k$ matrix E. That is, its generating matrix is of the form:

$$G = \begin{pmatrix} I_k \\ E_{n-k,k} \end{pmatrix}$$

and every minor of E must be invertible. A common choice of E is the Vandermonde matrix (cf. [17, 23, 43, 65]). In this setting, matrix E look as follow:

$$E_{f,\frac{P}{k}} = \begin{pmatrix} 1 & \eta_0 & \eta_0^2 & \cdots & \eta_0^{\frac{P}{k}-1} \\ 1 & \eta_1 & \eta_1^2 & \cdots & \eta_1^{\frac{P}{k}-1} \\ \vdots & \vdots & \ddots & \vdots & \\ 1 & \eta_{f-1} & \eta_{f-1}^2 & \cdots & \eta_{f-1}^{\frac{P}{k}-1} \end{pmatrix}$$

where $\{\eta_i \in \mathbb{N}\}_{i=0}^{f-1}$ are a distinct set of integers.

3 PARALLEL TOOM-COOK

Toom-Cook-k is a recursive long integer multiplication algorithm. At each recursive step, it splits two integers into k parts, constructs $2k - 1$ sub-multiplications, and uses polynomial interpolation to assemble the output. We present a parallel Toom-Cook algorithm based on the BFS-DFS parallelization technique (cf. [4, 21, 52, 56]). Our algorithm generalizes the parallel Karatsuba's algorithm (Toom-Cook-2) of [18]. We follow the sequential algorithm with some adjustments to the parallel setting (recall Toom-Cook's algorithm in Section 2.2 and Algorithm 1). Consider the recursive tree of Toom-Cook-k. The root corresponds to the task of multiplying a by b. The $2k - 1$ children of the root stand for the $2k - 1$ sub-multiplications, and so on. The parallel algorithm traverses the recursive tree in parallel as follows: At each tree level, the algorithm proceeds in either a BFS step or a DFS step. A BFS step divides the $2k - 1$ sub-problems among the processors, such that $\frac{1}{2k-1}$ of the available processors at that point work on each sub-problem independently and in parallel. A DFS step uses all available processors at that point to solve each sub-problem sequentially. As shown in [4], a DFS step requires less memory but eventually increases the communication cost compared to a BFS step. We provided a cost analysis in Section 5. The recursion stops when a single processor is assigned to each sub-problem. Thus, the algorithm performs a total of $\log_{(2k-1)} P$ BFS steps.

Lemma 3.1. *The number of DFS steps a parallel Toom-Cook algorithm must perform is at least*

$$l_{DFS} = \log_k \left(\frac{n}{P^{\log_{(2k-1)} k}} \cdot \frac{1}{M} \right)$$

Proof. Each recursive step reduces the problem size by a factor of k, which reduces the memory footprint of each sub-problem by a factor k. In a DFS step, the sub-problems are computed sequentially, while in a BFS step, the $2k-1$ sub-problems are computed in parallel. Hence, a DFS step reduces the memory footprint for a single processor by a factor of k, while a BFS step increases it by a factor of $\frac{2k-1}{k}$. Recall that the algorithm performs exactly $\log_{(2k-1)} P$ BFS steps, which increases the memory footprint by a factor of

$$\left(\frac{2k-1}{k}\right)^{\log_{(2k-1)} P} = \frac{P}{k^{\log_{(2k-1)} P}} = \frac{P}{k^{\frac{\log_k P}{\log_k (2k-1)}}}$$
$$= \frac{P}{P^{\frac{1}{\log_k (2k-1)}}} = \frac{P}{P^{\log_{(2k-1)} k}} = P^{1-\log_{(2k-1)} k}$$

Let l denote the minimal DFS steps required to satisfy the memory constraints. This means that,

$$M \geq \frac{n}{P} \cdot \left(\frac{1}{k}\right)^l \cdot \left(\frac{2k-1}{k}\right)^{\log_{(2k-1)} P}$$
$$= \frac{n}{P} \cdot P^{1-\log_{(2k-1)} k} \cdot \frac{1}{k^l} = \frac{n}{P^{\log_{(2k-1)} k}} \cdot \frac{1}{k^l}$$

Resulting with

$$l = \left\lceil \log_k \left(\frac{n}{P^{\log_{(2k-1)} k}} \cdot \frac{1}{M} \right) \right\rceil$$

□

A similar analysis is presented in [52] for fast matrix multiplication. Ballard et al. [4] showed that performing sufficient DFS steps (to fit memory limitations) followed by all BFS steps is optimal. Notice that when $M = \Omega\left(\frac{n}{P^{\log_{(2k-1)} k}}\right)$ the algorithm can execute the multiplication with only BFS steps. This is often called the unlimited memory case, as additional memory does not benefit the algorithm. We assume that n is a power of k and P is a power of $2k-1$ (otherwise, we can add padding or use fewer processors). We follow a similar data partitioning as in [4, 53] for parallel fast matrix multiplication, with the necessary adaptions to the Toom-Cook algorithm. We label the processors using $\log_{(2k-1)} P$-digit strings, signifying their index in base $(2k-1)$. We arrange the processors in a two-dimensional grid of dimensions $\frac{P}{2k-1} \times (2k-1)$. At each BFS step, we reposition the processors in the grid such that at the $i'th$ BFS step, processors in the same row align in all coordinates except the i^{th}, and the i^{th} digit points to the column. At the beginning (resp. end) of the run, the input (resp. output) is distributed on all processors. Let l_{total} denote the algorithm's total number of recursive steps. We use a one-dimensional block-cyclic data layout on blocks of size $\frac{n}{P} \cdot \frac{1}{k^{l_{total}}}$. This setting minimizes communication cost as shown in [53]. A BFS step involves communication only within rows of the grid ($2k-1$ processors), and a DFS step does not involve communication at all. The rest of the algorithm is similar to the standard Toom-Cook algorithm.

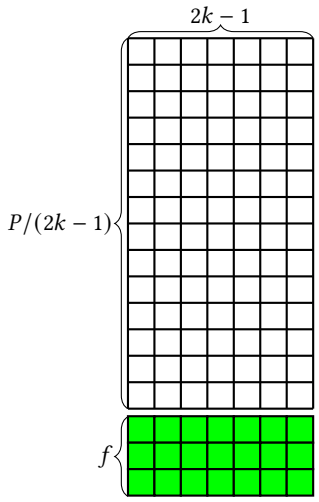

Figure 1: Fault-Tolerant Toom-Cook algorithm with $f \cdot (2k-1)$ additional code processors (highlighted in green), encoded using a linear code. Each code processor encodes a single column. Communication occurs only within the rows.

4 FAULT-TOLERANT TOOM-COOK

In this section, we propose a fault-tolerant Toom-Cook algorithm based on the parallel version of Toom-Cook (Section 3). We propose a tailored code for each stage of the algorithm. We follow [8] and apply a linear erasure code that enables fault-tolerance throughout the evaluation and interpolation stages. We then use a polynomial code to enable fault-tolerance in the multiplication stage, leveraging the unique structure of Toom-Cook. In contrast, in fast matrix multiplication, Birnbaum et al. [8] cannot use a code for the multiplication stage and require recomputations to recover from faults at that stage. Integrating our code with Toom-Cook's multiplication stage reduces the overhead costs of our fault-tolerant algorithm. We next present the application of the codes, starting from our linear coding. We follow the notations in Section 3.

4.1 Linear coding

We follow the parallelization and data partitioning of Parallel Toom-Cook. We arrange the processors in a similar two-dimensional grid of size $\frac{P}{2k-1} \times (2k-1)$ and add f additional rows of code processors (a total of $f \cdot (2k-1)$ code processors) at the bottom of the grid, as shown in Figure 1. We denote by $P^s_{i,j}$ the processor in the $i'th$ row and the $j'th$ column of the two-dimensional grid at the $s'th$ BFS step.

Code creation: We encode the data at each column of standard processors on the f code processors in their column, using $(2k-1)$ identical codes of size $\left(\frac{P}{(2k-1)} + f, \frac{P}{(2k-1)}, f+1\right)$, as shown in Section 2.5. Namely, each code processor holds a weighted sum of the data in its column processors. Let $A^s_{i,j}, B^s_{i,j}$ denote the parts of a, b stored at $P^s_{i,j}$. Following the notations of Section 2.5, we get that

$$\forall i \in [0, \cdots, f-1], j \in [0, \cdots, 2k-2],$$
$$l \in [0, \cdots, \log_{(2k-1)} P - 1] :$$
$$A^s_{P+i,j} = \sum_{l \in [0,\cdots,\frac{P}{k}-1]} \eta_i^l \cdot A^s_{l,j}, \text{ and}$$
$$B^s_{P+i,j} = \sum_{l \in [0,\cdots,\frac{P}{k}-1]} \eta_i^l \cdot B^s_{l,j}$$

Each BFS step initiates a new code creation process (after processors' repositioning). The rest of the algorithm is similar to Parallel Toom-Cook. The code processors mimic the behavior of the processors in their columns.

Fault recovery: When a fault occurs, the data in that processor is lost, and the processor is replaced by a new processor (recall Section 2.1). When the remaining processors in the column finish their computations, they reconstruct its output using the code processors in their column. This is done using a reduce operation to the newly allocated processor.

Correctness: We next show why the code is preserved throughout the run. After the code creation process, each code processor holds a linear sum of the data at the processors in its column. Due to our data partitioning strategy, a DFS step involves only local computations. Since processors in the same column, code processors included, perform the same local operations, the code is preserved. A BFS step involves communication between processors in the same row. However, since all processors, code processors included, perform the same communication operations between processors in their row, the code is preserved. This code is not preserved during the multiplication stage, as the local computations involve inner-data multiplications. To recover from faults in the multiplication phase, one must recompute the output of the faulty processor. This recovery is very costly compared to the on-the-fly recovery of faults in other stages.

4.2 Polynomial coding

To maintain efficient recovery throughout the entire algorithm, one must combine a different coding strategy for the multiplication stage. We propose a polynomial coding strategy that utilizes the algorithm's structure.

We arrange the processors in a two-dimensional grid, similar to Parallel Toom-Cook, and add f additional columns of code processors (a total of $f \cdot \frac{P}{(2k-1)}$ code processors) at the right-hand side of the grid, as shown in Figure 2.

Code creation: Let $S = \{(x_i, h_i)\}_{i=0}^{2k-2}$ denote the set of evaluation points of Toom-Cook. We add f redundant evaluation points, i.e., a total of $2k-1+f$ evaluation points, denoted by $S' = \{P_i = (x_i, h_i)\}_{i=0}^{2k-2+f}$. The evaluation matrices U', V' are defined as following:

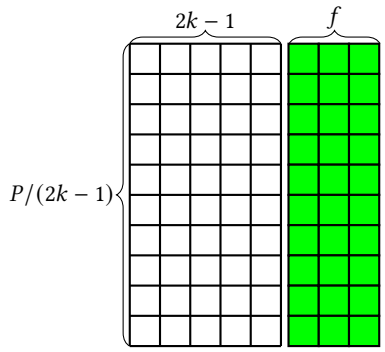

Figure 2: Fault-Tolerant Toom-Cook algorithm with $f \cdot \frac{P}{(2k-1)}$ additional code processors (highlighted in green), encoded using a polynomial code.

$$U' = V' = \begin{pmatrix} h_0^{k-1}x_0^0 & h_0^{k-2}x_0^1 & \cdots & h_0^0 x_0^{k-1} \\ h_1^{k-1}x_1^0 & h_1^{k-2}x_1^1 & \cdots & h_1^0 x_1^{k-1} \\ \vdots & \vdots & \ddots & \vdots \\ h_{2k-2+f}^{k-1}x_{2k-2+f}^0 & h_{2k-2+f}^{k-2}x_{2k-2+f}^1 & \cdots & h_{2k-2+f}^0 x_{2k-2+f}^{k-1} \end{pmatrix}$$

The rest of the algorithm is similar to Parallel Toom-Cook but with a new set of evaluation points S'. After the first recursive step, the algorithm proceeds with the standard Parallel Toom-Cook algorithm.

Fault recovery: There is no fault recovery mechanism. When a fault occurs, we halt the execution of the remaining processors of its column.

Correctness: Notice that the interpolation stage only requires the output of $2k-1$ sub-problems. Since there are at most f faults, at least $2k-1$ sub-problems finish, and the interpolation stage proceeds as planned. The only difference from Parallel Toom-Cook is that the interpolation matrix is calculated on the fly according to the evaluation points of the finished sub-problems.

4.3 Multi-step traversal

The polynomial coding strategy often requires more code processors to withstand each fault ($\frac{P}{(2k-1)}$ compared to k). To improve that, we combine multiple BFS steps into a single large one. Let $l \in [1, \cdots, \log_{(2k-1)} P]$ denote the number of combined steps. We divide the P original processors into a two dimensional grid of size $\frac{P}{(2k-1)^l} \times (2k-1)^l$. We add f additional columns of processors (a total of $f \cdot \frac{P}{(2k-1)^l}$ additional processors) at the right-hand side of the two-dimensional grid as shown in Figure 3. The rest of the algorithm is similar to Fault-tolerant Toom-Cook-k^l with polynomial code.

Allowing multi-step traversal in Toom-Cook with fault tolerance requires finding redundant evaluation points. We leave that for future work.

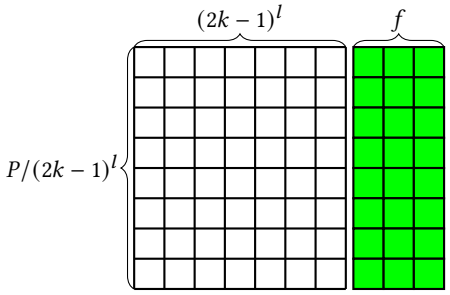

Figure 3: Fault-Tolerant Toom-Cook algorithm with $f \cdot \frac{P}{(2k-1)^l}$ additional code processors (highlighted in green), encoded using a polynomial code.

Remark 4.1. *The Toom-Graph technique [11] (for optimization of the interpolation stage) is applicable to our algorithm.*

5 COST ANALYSIS AND COMPARISON

This section analyzes the costs of Parallel Toom-Cook, Fault-tolerant Toom-Cook, and Toom-Cook with replication.

5.1 Parallel Toom-Cook

Theorem 5.1. *Let P, M, and n denote the number of processors, memory size, and input size. Denote by F, BW, and L the arithmetic, bandwidth, and latency costs of Parallel Toom-Cook, respectively. Then,*

$$F = \Theta\left(\frac{n^{\log_k(2k-1)}}{P}\right)$$
$$BW = \Theta\left(\max\left(\left(\frac{n}{M}\right)^{\log_k(2k-1)} \cdot \frac{M}{P}, \frac{n}{P^{\log_{(2k-1)}k}}\right)\right)$$
$$L = \Theta\left(\max\left(\left(\frac{n}{M}\right)^{\log_k(2k-1)} \cdot \frac{\log P}{P}, \log P\right)\right)$$

Proof. **Unlimited-memory case:** In this case, the algorithm only performs BFS steps. In a BFS step, the evaluation and interpolation stages involve local computations and data exchanges between processors in the same row. The data exchanges are performed using an all-reduce operation. For a data of size $\frac{n}{P}$, this costs $(F_{BFS}, BW_{BFS}, L_{BFS}) = \left(\Theta\left(\frac{n}{P}\right), \Theta\left(\frac{n}{P}\right), \Theta(1)\right)$ (Lemma 2.5). Each BFS step increases the data size by a factor of $\frac{2k-1}{k}$. Summing the costs of all BFS steps, we get that:

$$F_{BFS} = \sum_{i=1}^{\log_{(2k-1)}P} \Theta\left(\frac{n}{P} \cdot \left(\frac{2k-1}{k}\right)^i\right) = \Theta\left(\frac{n}{P^{\log_{(2k-1)}k}}\right)$$
$$BW_{BFS} = \sum_{i=1}^{\log_{(2k-1)}P} \Theta\left(\frac{n}{P} \cdot \left(\frac{2k-1}{k}\right)^i\right) = \Theta\left(\frac{n}{P^{\log_{(2k-1)}k}}\right)$$
$$L_{BFS} = \sum_{i=1}^{\log_{(2k-1)}P} \Theta(1) = \Theta(\log P)$$

The multiplication phase involves $n^{\log_k(2k-1)}$ scalar multiplications, performed by P processors. Hence,

$$F_{mul} = \Theta\left(\frac{n^{\log_k(2k-1)}}{P}\right),\ BW_{mul} = L_{mul} = 0$$

Overall, we get that

$$F = \Theta\left(\frac{n^{\log_k(2k-1)}}{P}\right)$$
$$BW = \Theta\left(\frac{n}{P^{\log_{(2k-1)}k}}\right) \quad (1)$$
$$L = \Theta(\log P)$$

Limited-memory case: In this case, the algorithm performs $l_{DFS} = \log_k\left(\frac{n}{P^{\log_{(2k-1)}k}} \cdot \frac{1}{M}\right)$ DFS steps before switching to BFS steps (Lemma 3.1). Each DFS step reduces the problem size by a factor of k but increases the number of problems by a factor of $2k-1$. Thus, the limited-memory case is similar to running the unlimited-memory case t_{um} times for inputs of size n_{um}, where

$$t_{um} = (2k-1)^{\log_k\left(\frac{n}{P^{\log_{(2k-1)}k}} \cdot \frac{1}{M}\right)}$$
$$= k^{\log_k(2k-1)\cdot\log_k\left(\frac{n}{P^{\log_{(2k-1)}k}} \cdot \frac{1}{M}\right)}$$
$$= \left(\frac{n}{P^{\log_{(2k-1)}k}} \cdot \frac{1}{M}\right)^{\log_k(2k-1)} = \left(\frac{n}{M}\right)^{\log_k(2k-1)} \cdot \frac{1}{P}$$

and

$$n_{um} = \frac{n}{k^{l_{DFS}}} = \frac{n}{\frac{n}{P^{\log_{(2k-1)}k}} \cdot \frac{1}{M}} = P^{\log_{(2k-1)}k} \cdot M$$

From Equation (1), we have

$$F_{um} = \Theta\left(\frac{n_{um}^{\log_k(2k-1)}}{P}\right) = \Theta\left(\frac{\left(P^{\log_{(2k-1)}k} \cdot M\right)^{\log_k(2k-1)}}{P}\right)$$
$$= \Theta\left(\frac{P \cdot M^{\log_k(2k-1)}}{P}\right) = \Theta\left(M^{\log_k(2k-1)}\right)$$
$$BW_{um} = \Theta\left(\frac{n_{um}}{P^{\log_{(2k-1)}k}}\right) = \Theta\left(\frac{P^{\log_{(2k-1)}k} \cdot M}{P^{\log_{(2k-1)}k}}\right) = \Theta(M)$$
$$L_{um} = \Theta(\log P)$$

Overall, we get that

$$F = t_{um} \cdot F_{um} = \Theta\left(\frac{n^{\log_k(2k-1)}}{P}\right)$$
$$BW = t_{um} \cdot BW_{um} = \Theta\left(\left(\frac{n}{M}\right)^{\log_k(2k-1)} \cdot \frac{M}{P}\right)$$
$$L = t_{um} \cdot L_{um} = \Theta\left(\left(\frac{n}{M}\right)^{\log_k(2k-1)} \cdot \frac{\log P}{P}\right)$$

□

5.2 Fault-tolerant Toom-Cook

Theorem 5.2. *Let P, M, n, and f denote the number of processors, memory size, input size, and number of faults. Let F, BW, L, and P denote the arithmetic cost, bandwidth cost, latency cost, and processor count of Parallel Toom-Cook, respectively. Denote by $F^{'}$, $BW^{'}$, $L^{'}$, and $P^{'}$ the arithmetic cost, bandwidth cost, latency cost, and processor count of Fault-tolerant Toom-Cook, respectively. Then,*

$$\begin{aligned} F^{'} &= (1+o(1)) \cdot F \\ BW^{'} &= (1+o(1)) \cdot BW \\ L^{'} &= (1+o(1)) \cdot L \\ P^{'} &= \begin{cases} f \cdot (2k-1) & M = O\left(\frac{n}{P^{\log_{(2k-1)} k}}\right) \\ f & otherwise \end{cases} \end{aligned}$$

Proof. Our fault-tolerance version combines two types of codes. We start with linear coding for the evaluation and interpolation phases, and then we switch to polynomial coding for the multiplication phase. We arrange the processors in a $\frac{P}{2k-1} \times (2k-1)$ grid and add $f \cdot (2k-1)$ code processors (as shown in Figure 1). We then encode the original processors data on the code processors (using an f-reduce operation), which costs (Lemma 2.5)

$$(F_{cc},\ BW_{cc},\ L_{cc}) = \left(O(f \cdot M),\ O(f \cdot M),\ O\left(\log\left(\frac{P}{2k-1}+f\right)\right)\right)$$

When a fault occurs, the processors in the faulty processor's column recover its lost data using a reduce operation to the newly allocated processor. Recovering from f faults costs is done using an f-reduce operation, which costs (Lemma 2.5)

$$\left(F_{fr},\ BW_{fr},\ L_{fr}\right) = \left(O(f \cdot M),\ O(f \cdot M),\ O\left(\log\left(\frac{P}{2k-1}+f\right)\right)\right)$$

For the multiplication phase, we arrange the processors in a $\frac{P}{(2k-1)^r} \times (2k-1)^r$ grid and add $f \cdot \frac{P}{(2k-1)^r}$ code processors (as shown in Figure 2). We then perform a single step of Parallel Toom-Cook with f additional evaluation points and continue recursively with the standard Parallel Toom-Cook. Recall the cost analysis of Parallel Toom-Cook. The cost of the first step in the fault tolerant algorithm (5.1), increasing the cost of the first step by a factor of $\frac{2k-1+f}{2k-1}$, compared to the same step (for using $2k-1+f$ sets instead of $2k-1$) does not affect asymptotically. Moreover, as we disregard the work of faulty processors, there are no overhead costs for fault recovery. Overall, we get that:

$$\begin{aligned} F^{'} &= (1+o(1)) \cdot F \\ BW^{'} &= (1+o(1)) \cdot BW \\ L^{'} &= (1+o(1)) \cdot L \\ P^{'} &= f \cdot (2k-1) \end{aligned}$$

Notice that in the unlimited-memory case ($M = \Omega\left(\frac{n}{P^{\log_{(2k-1)} k}}\right)$), one can avoid using linear coding. In this case, using Fault-Tolerant Toom-Cook-k with l multi-step traversal (for $l = \log_{2k-1} P$) reduces the number of additional processors to f (instead of $f \cdot (2k-1)$). □

5.3 Toom-Cook with Replication

Theorem 5.3. *Let P, M, n, and f denote the number of processors, memory size, input size, and number of faults. Let F, BW, L, and P denote the arithmetic cost, bandwidth cost, latency cost, and processor count of Parallel Toom-Cook, respectively. Denote by $F^{'}$, $BW^{'}$, $L^{'}$, and $P^{'}$ the arithmetic cost, bandwidth cost, latency cost, and processor count of Toom-Cook with Replication, respectively. Then,*

$$\begin{aligned} F^{'} &= F \\ BW^{'} &= (1+o(1)) \cdot BW \\ L^{'} &= (1+o(1)) \cdot L \\ P^{'} &= f \cdot P \end{aligned}$$

Proof. In the replication strategy, we replicate the processors into f (by adding $f-1$ sets of P processors) and run Parallel Toom-Cook on each set independently and in parallel. Thus, the costs of this algorithm are similar to Parallel Toom-Cook with a negligible communication overhead on data replicating. □

6 GENERALIZATION AND SPECIAL CASES

6.1 Multi-step Toom-Cook

This section provides the full details on Multi-step Toom-Cook with fault tolerance. Toom-Cook with l-step traversal generates k^l sub-problems at each recursive step. In our fault-tolerant version, we add f new sub-problems, namely, f new evaluation points. To do so, we introduce (r,l)-general position, a generalization of linear general position ([80]), and show that choosing a set of points with this property is a sufficient condition for valid evaluation points. We then propose a heuristic for finding the set of points.

Let $k, l, f \in \mathbb{N}$ be the parameters of a fault-tolerant l-step Toom-Cook-k algorithm. A set S of $(2k-1)^l + f$ evaluation points over some field F is valid if and only if the evaluation matrix of every subset of $(2k-1)^l$ points is injective.

Definition 6.1 ((r,l)-General Position). *Let F^l be an l dimensional Euclidean space. Let S be a set of points in F^l. The points in S are in (r,l)-general position if and only if for every $S' \subset S$ such that $|S'| = r^l$, the only polynomial in $Poly_{r,l}$ that vanishes on S' is the zero polynomial.*

Claim 6.1. *Let $N = r^l$. Let S be a set of n evaluation points, where $N \leq n \in \mathbb{N}$. Let $A \in F^{n \times N}$ be an evaluation matrix of S. Every sub-matrix of size $N \times N$ is injective if and only if S is in (r,l)-general position.*

Proof. Assume that S is in (r,l)-general position. Assume by contradiction that a sub-matrix $A^{'} \in F^{N \times N}$, corresponding to the evaluation matrix of $S^{'} \subset S$, is not injective, namely its N rows are linearly dependent. Since A' is not invertible, there is $p \in \ker A' \backslash \{0\}$, which means that p is the coefficient vector of a non-zero polynomial in $Poly_{r,l}$ that vanishes on $S^{'}$. Hence, S is not in (r,l)-general position (see Definition 6.1). The proof for the other direction is similar. If S is in (r,l)-general position, then the evaluation matrix for every subset $S^{'} \subset S$ of size N is invertible, and hence, injective. □

Hence, if a set S of $(2k-1)^l + f$ points is in $(2k-1,l)$-general position, then it is a valid set of evaluation points for fault-tolerant Toom-Cook-k with l-step traversal.

6.2 Finding points in $(2k-1,l)$-general position

We provide a heuristic for finding f redundant evaluation points for Fault-Tolerant Toom-Cook-k. Our heuristic is recursive, adding one point at a time.

We follow the notations in Section 6.1. Let S be a set of evaluation points in $(2k-1,l)$-general position. Denote by $T_S = \{L \subset S : |L| = (2k-1)^l - 1\}$, and let $P \in T_S$ be an arbitrary subset of S. Let $x \in F^l$ be a vector. We denote by $A_P(x) \in F^{(2k-1)^l \times (2k-1)^l}$ the evaluation matrix of $P \cup \{x\}$, and by $q_P(x)$ the multivariate polynomial $\det(A_P(x))$ where $x = (x_1, ..., x_l)$ are the variables.

Claim 6.2. *Let $k, l \in \mathbb{N}$. Let S be a set of evaluation points in $(2k-1,l)$-general position. Let $x \in F^l$ be a point such that $q_P(x) \neq 0$ for every $P \in T_S$. Then, $S \cup \{x\}$ is in $(2k-1,l)$-general position.*

Proof. Let $S' \subset S \cup \{x\}$ such that $|S'| = (2k-1)^l$. In case $x \notin S'$ then $S' \subset S$ and thus the only polynomial that vanishes on S' is the zero polynomial since S is in $(2k-1,l)$-general position.

In case $x \in S'$, then let $P = S' \backslash \{x\}$. The matrix $A_P(x)$ is the evaluation matrix of S'. The matrix $A_P(x)$ is invertible since $\det(A_P(x)) = q_P(x) \neq 0$. Therefore, $\ker(A_P(x)) = \{0\}$ and thus the only polynomial that vanishes on S' is the zero polynomial. Thus, $S \cup \{x\}$ is in $(2k-1,l)$-general position. □

Let U_S denote the set:

$$U_S = \{x \in F^l : \exists P \in T_S \textbf{ s.t. } q_P(x) = 0\}$$

From Claim 6.2, every $x \notin U_S$ can be used as a redundant evaluation point. We show in the following claims that there such point x always exists.

Claim 6.3. *Let $k, l \in \mathbb{N}$. Let S be a finite set in $(2k-1,l)$-general position. Let $P \in T_S$. Then, q_P is not the zero polynomial.*

Proof. Assume by contradiction that q_P is the zero polynomial. Let $E_P \in F^{((2k-1)^l-1)\times(2k-1)^l}$ be the evaluation matrix of P. Let y be a row vector in $(F^{(2k-1)^l})^T$ that is not in the span of the rows of E_P. Let $E_{P,y} \in F^{(2k-1)^l \times (2k-1)^l}$ be the matrix that its first $(2k-1)^l - 1$ rows are E_P and the last row is y. The rows of E_P are linearly independent since S is in $(2k-1,l)$-general position. Thus the rows of $E_{P,y}$ are linearly independent and $\det(E_{P,y}) \neq 0$.

Let $B_{P,i} \in F^{((2k-1)^l-1)\times((2k-1)^l-1)}$ be the evaluation matrix of P without the i'th column. The coefficient of the i'th monomial of q_P is $(-1)^{i+1}\det(B_{P,i})$. Since q_P is the zero polynomial, $\det(B_{P,i}) = 0$ and we get the contradiction:

$$0 \neq \det(E_{P,y}) = \sum_i^{t^l} [y]_i (-1)^{i+1} \det(B_{P,i}) = 0$$

. □

Claim 6.4. *Let S be a finite set in $(2k-1,l)$-general position. The set U_S is a null set, and there is $x \in F^l \backslash U_S$.*

Proof. Let S be a finite set in $(2k-1,l)$-general position. Let $P \in T_S$. The polynomial q_P is not the zero polynomial (Claim 6.3). Thus, q_P vanishes on a finite union of curves (Harnack's Inequality [73]). Since each curve is a null set and a finite union of null sets is a null set, q_P vanishes on a null set. Hence, U is a finite union of null sets and thus a null set. Since F^l is not a null set, there is $x \in F^l \backslash U_S$. □

Hence, we can always find more evaluation points using our proposed heuristic.

An alternative proof is that we can always find more evaluation points in $\mathbb{Z}^l$.

Claim 6.5. *Let R be a ring with characteristic 0. Let $l, k \in \mathbb{N}$ and let S be a finite set of evaluation points from R^l that are in (k,l)-general position. There is an element $x \in \mathbb{Z}^l \subset R^l$ such that $S \cup \{x\}$ is in general position.*

Proof. It is enough to show that there is $x \in \mathbb{Z}$ such that $q_P(x) \neq 0$ for every $P \in T_S$ (Claim 6.2). Let $q_S = \prod_{P \in T_S} q_P$ be a polynomial. We can see that there is $P \in T_S$ such that $q_P(x) = 0$ if and only if $q_S(x) = 0$. Since the power of each variable of each monomial in each q_P is at most $k-1$, then the power of each variable in each monomial of q_S is at most $|T_S|(k-1)$. Therefore, $q_S \in Poly_{|T_S|(k-1)+1,l}$. Since $q_P \neq 0$ for every $P \in T_S$, we conclude that $q_S \neq 0$.

Let $A = \{0, \ldots, |T_S|(k-1)\}$ a set of points in $\mathbb{Z}$. The points A can produce a Toom-Cook-$\frac{|T_S|(k-1)+2}{2}$ algorithm since those are $|T_S|(k-1)+1$ different evaluation points in one dimension. Thus, A^l can produce an l-steps Toom-Cook-$\frac{|T_S|(k-1)+2}{2}$ algorithm. So, A^l are in $(|T_S|(k-1)+1,l)$-general position (Claim 2.2). Since $q_S \in Poly_{|T_S|(k-1)+1,l}$ and q_S is not the zero polynomial, then q_S does not vanish on A^l. Thus, there is $x \in A^l$ that $q_S(x) \neq 0$. Then, $q_P(x) \neq 0$ for every $P \in T_S$ and thus $S \cup \{x\}$ is in (k,l)-general position. Since $A^l \subset \mathbb{Z}^l$, there is $x \in \mathbb{Z}^l$ such that $S \cup \{x\}$ is in (k,l)-general position. □

7 DISCUSSION

This paper addresses the problem of faults in long integer multiplication and proposes a new coded computation approach that is significantly faster than existing solutions. Traditional general-purpose methods, while broadly applicable, incur large overheads. Future directions include empirical research on real-world settings and exploring the applicability of our coding strategy to other algorithms. The code of Birnbaum et al. [8] requires recomputations to recover from faults in the multiplication stage. Our code avoids this cost by introducing a code tailored to the algorithm. This approach may apply to improve fault-tolerant of other recursive linear algebra algorithms, including FFT-based algorithms. Optimizing the choice of redundant evaluation points may lead to speedup in practice by decreasing the arithmetic cost by a constant factor. In addition, finding redundant evaluation points for Fault Tolerant Toom-Cook-k with multi-step traversal will reduce the additional processor count (in the unlimited-memory case) by a significant constant factor of $2k-1$. Finally, the evaluation and interpolation stages are linear transformations. These can be viewed as codes

that may hold natural fault tolerance, similar to the ones in fast matrix multiplication [8].

8 ACKNOWLEDGMENTS

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 818252). This project has received funding from the European Research Council under the European Union's Horizon 2020 research and innovation programme (grant agreement No 101113120, 101138056). Research was supported by grant No. 1354/23 of the Israel Science Foundation (founded by the Israel Academy of Sciences and Humanities). This research has been supported by the Science Accelerator and by the Frontiers in Science initiative of the Ministry of Innovation, Science and Technology.

REFERENCES

[1] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. 2013. Effective straggler mitigation: Attack of the clones. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. 185–198.

[2] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. 2010. Reining in the outliers in {Map-Reduce} clusters using mantri. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*.

[3] Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. 2004. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing* 1, 1 (2004), 11–33.

[4] Grey Ballard, James Demmel, Olga Holtz, Benjamin Lipshitz, and Oded Schwartz. 2012. Communication-optimal parallel algorithm for Strassen's matrix multiplication. In *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 193–204.

[5] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. 2011. Minimizing communication in numerical linear algebra. *SIAM J. Matrix Anal. Appl.* 32, 3 (2011), 866–901.

[6] Jose Maria Bermudo Mera, Angshuman Karmakar, and Ingrid Verbauwhede. 2020. Time-memory trade-off in Toom-Cook multiplication: an application to module-lattice based cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2020, 2 (2020), 222–244.

[7] Gianfranco Bilardi and Lorenzo De Stefani. 2019. The I/O complexity of Toom-Cook integer multiplication. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 2034–2052.

[8] Noam Birnbaum, Roy Nissim, and Oded Schwartz. 2020. Fault Tolerance with High Performance for Fast Matrix Multiplication. In *2020 Proceedings of the SIAM Workshop on Combinatorial Scientific Computing*. SIAM, 106–117.

[9] Noam Birnbaum and Oded Schwartz. 2018. Fault Tolerant Resource Efficient Matrix Multiplication. In *Proceedings of the Eighth SIAM Workshop on Combinatorial Scientific Computing*. SIAM, 23–34.

[10] Noam Birnbaum and Oded Schwartz. 2018. Fault tolerant resource efficient matrix multiplication. In *2018 Proceedings of the Seventh SIAM Workshop on Combinatorial Scientific Computing*. SIAM, 23–34.

[11] Marco Bodrato and Alberto Zanoni. 2006. What about Toom-Cook matrices optimality. *Centro "Vito Volterra" Università di Roma Tor Vergata* (2006).

[12] Franck Cappello, Al Geist, William Gropp, Sanjay Kale, Bill Kramer, and Marc Snir. 2014. Toward exascale resilience: 2014 update. *Supercomputing frontiers and innovations* 1, 1 (2014), 5–28.

[13] Henri Casanova. 2007. Benefits and drawbacks of redundant batch requests. *Journal of Grid Computing* 5, 2 (2007), 235–250.

[14] Zizhong Chen and Jack Dongarra. 2006. Algorithm-based checkpoint-free fault tolerance for parallel matrix computations on volatile resources. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*. IEEE, 10–pp.

[15] Zizhong Chen and Jack Dongarra. 2008. Algorithm-based fault tolerance for fail-stop failures. *IEEE Transactions on Parallel and Distributed Systems* 19, 12 (2008), 1628–1641.

[16] Stephen A Cook and StalO Aanderaa. 1969. On the minimum computation time of functions. *Trans. Amer. Math. Soc.* 142 (1969), 291–314.

[17] Son Hoang Dau, Wentu Song, Alex Sprintson, and Chau Yuen. 2015. Constructions of MDS codes via random Vandermonde and cauchy matrices over small fields. In *2015 53rd Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. IEEE, 949–955.

[18] Lorenzo De Stefani. 2020. Communication-optimal parallel standard and karatsuba integer multiplication in the distributed memory model. *arXiv preprint arXiv:2009.14590* (2020).

[19] Lorenzo De Stefani. 2022. Brief Announcement: On the I/O Complexity of Sequential and Parallel Hybrid Integer Multiplication Algorithms. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '22)*. Association for Computing Machinery, New York, NY, USA.

[20] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[21] James Demmel, David Eliahu, Armando Fox, Shoaib Kamil, Benjamin Lipshitz, Oded Schwartz, and Omer Spillinger. 2013. Communication-optimal parallel recursive rectangular matrix multiplication. (2013), 261–272.

[22] Jinnan Ding, Shuguo Li, and Zhen Gu. 2018. High-speed ECC processor over NIST prime fields applied with Toom–Cook multiplication. *IEEE Transactions on Circuits and Systems I: Regular Papers* 66, 3 (2018), 1003–1016.

[23] Sanghamitra Dutta, Viveck Cadambe, and Pulkit Grover. 2016. Short-dot: Computing large linear transforms distributedly using coded short dot products. *Advances In Neural Information Processing Systems* 29 (2016).

[24] Dutta, Sanghamitra and Fahim, Mohammad and Haddadpour, Farzin and Jeong, Haewon and Cadambe, Viveck and Grover, Pulkit. 2020. On the Optimal Recovery Threshold of Coded Matrix Multiplication. *IEEE Transactions on Information Theory* 66, 1 (2020), 278–301.

[25] Elmootazbellah N Elnozahy and James S Plank. 2004. Checkpointing for peta-scale systems: A look into the future of practical rollback-recovery. *IEEE Transactions on Dependable and Secure Computing* 1, 2 (2004), 97–108.

[26] Rūsiņš Freivalds. 1979. Fast probabilistic algorithms. In *International Symposium on Mathematical Foundations of Computer Science*. Springer, 57–69.

[27] Martin Fürer. 2009. Faster integer multiplication. *SIAM J. Comput.* 39, 3 (2009), 979–1005.

[28] Kristen Gardner, Samuel Zbarsky, Sherwin Doroudi, Mor Harchol-Balter, and Esa Hyytia. 2015. Reducing latency via redundant requests: Exact analysis. *ACM SIGMETRICS Performance Evaluation Review* 43, 1 (2015), 347–360.

[29] Torbjörn Granlund. 2004. GNU MP: The GNU multiple precision arithmetic library. *http://gmplib.org/* (2004).

[30] Torbjörn Granlund and Peter L Montgomery. 1994. Division by invariant integers using multiplication. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*. 61–72.

[31] Zhen Gu and Shuguo Li. 2018. A division-free Toom–Cook multiplication-based montgomery modular multiplication. *IEEE Transactions on Circuits and Systems II: Express Briefs* 66, 8 (2018), 1401–1405.

[32] John A Gunnels, Daniel S Katz, Enrique S Quintana-Orti, and RA Van de Gejin. 2001. Fault-tolerant high-performance matrix multiplication: Theory and practice. In *Dependable Systems and Networks, 2001. DSN 2001. International Conference on*. IEEE, 47–56.

[33] David Harvey and Joris Van Der Hoeven. 2021. Integer multiplication in time O(n log n). *Annals of Mathematics* 193, 2 (2021), 563–617.

[34] Sangwoo Hong, Heecheol Yang, Youngseok Yoon, Taehyun Cho, and Jungwoo Lee. 2021. Chebyshev Polynomial Codes: Task Entanglement-based Coding for Distributed Matrix Multiplication. In *Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 4319–4327.

[35] Kuang-Hua Huang and Jacob A Abraham. 1984. Algorithm-based fault tolerance for matrix operations. *IEEE transactions on computers* 100, 6 (1984), 518–528.

[36] Longbo Huang, Sameer Pawar, Hao Zhang, and Kannan Ramchandran. 2012. Codes can reduce queueing delay in data centers. In *2012 IEEE International Symposium on Information Theory Proceedings*. IEEE, 2766–2770.

[37] Dror Irony, Sivan Toledo, and Alexander Tiskin. 2004. Communication lower bounds for distributed-memory matrix multiplication. *J. Parallel and Distrib. Comput.* 64, 9 (2004), 1017–1026.

[38] Gauri Joshi, Yanpei Liu, and Emina Soljanin. 2014. On the delay-storage trade-off in content download from coded distributed storage systems. *IEEE Journal on Selected Areas in Communications* 32, 5 (2014), 989–997.

[39] Gauri Joshi, Emina Soljanin, and Gregory Wornell. 2017. Efficient redundancy techniques for latency reduction in cloud systems. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)* 2, 2 (2017), 1–30.

[40] Anatoly Karatsuba and Yu Ofman. 1963. Multiplication of MultiDigit Numbers on Automata. *Soviet Physics-Doklady* 7 (1963), 595–596.

[41] Myungsun Kim and Benjamin Z Kim. 2017. An experimental study of encrypted polynomial arithmetics for private set operations. *Journal of Communications and Networks* 19, 5 (2017), 431–441.

[42] Richard Koo and Sam Toueg. 1987. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on software Engineering* 1 (1987), 23–31.

[43] Jérome Lacan and Jérome Fimes. 2004. Systematic MDS erasure codes based on Vandermonde matrices. *IEEE Communications Letters* 8, 9 (2004), 570–572.

[44] Harashta Tatimma Larasati, Asep Muhamad Awaludin, Janghyun Ji, and Howon Kim. 2021. Quantum Circuit Design of Toom 3-Way Multiplication. *Applied Sciences* 11, 9 (2021), 3752.

[45] Kangwook Lee, Maximilian Lam, Ramtin Pedarsani, Dimitris Papailiopoulos, and Kannan Ramchandran. 2017. Speeding up distributed machine learning using codes. *IEEE Transactions on Information Theory* 64, 3 (2017), 1514–1529.

[46] Kangwook Lee, Ramtin Pedarsani, Dimitris Papailiopoulos, and Kannan Ramchandran. 2017. Coded computation for multicore setups. In *2017 IEEE International Symposium on Information Theory (ISIT)*. IEEE, 2413–2417.

[47] Kangwook Lee, Changho Suh, and Kannan Ramchandran. 2017. High-dimensional coded matrix multiplication. In *2017 IEEE International Symposium on Information Theory (ISIT)*. IEEE, 2418–2422.

[48] Ku Leuven, Reviewers Marco Lewandowsky, and Miha Stopar. 2020. D5. 3 Final Report on Hardware-Optimized Schemes. *FENTEC* (2020).

[49] S. Li, M. A. Maddah-Ali, and A. S. Avestimehr. 2015. Coded MapReduce. In *2015 53rd Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. 964–971.

[50] Songze Li, Mohammad Ali Maddah-Ali, Qian Yu, and A Salman Avestimehr. 2017. A fundamental tradeoff between computation and communication in distributed computing. *IEEE Transactions on Information Theory* 64, 1 (2017), 109–128.

[51] Songze Li, Sucha Supittayapornpong, Mohammad Ali Maddah-Ali, and Salman Avestimehr. 2017. Coded terasort. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 389–398.

[52] Benjamin Lipshitz, Grey Ballard, James Demmel, and Oded Schwartz. 2012. Communication-avoiding parallel strassen: Implementation and performance. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 101.

[53] Qingshan Luo and John B Drake. 1995. A scalable parallel Strassen's matrix multiplication algorithm for distributed-memory computers. In *Proceedings of the 1995 ACM symposium on Applied computing*. ACM, 221–226.

[54] Ankur Mallick, Malhar Chaudhari, Utsav Sheth, Ganesh Palanikumar, and Gauri Joshi. 2019. Rateless codes for near-perfect load balancing in distributed matrix-vector multiplication. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 3, 3 (2019), 1–40.

[55] Ankur Mallick, Malhar Chaudhari, Utsav Sheth, Ganesh Palanikumar, and Gauri Joshi. 2022. Rateless codes for near-perfect load balancing in distributed matrix-vector multiplication. *Commun. ACM* 65, 5 (2022), 111–118.

[56] William F McColl and Alexandre Tiskin. 1999. Memory-efficient matrix multiplication in the BSP model. *Algorithmica* 24, 3 (1999), 287–297.

[57] Michael Moldaschl, Karl E Prikopa, and Wilfried N Gansterer. 2017. Fault tolerant communication-optimal 2.5D matrix multiplication. *J. Parallel and Distrib. Comput.* 104 (2017), 179–190.

[58] Peter L Montgomery. 2005. Five, six, and seven-term Karatsuba-like formulae. *IEEE Trans. Comput.* 54, 3 (2005), 362–369.

[59] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R De Supinski. 2010. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–11.

[60] Duc Tri Nguyen and Kris Gaj. 2021. Fast NEON-based multiplication for lattice-based NIST Post-Quantum Cryptography finalists. In *International Conference on Post-Quantum Cryptography*. Springer, 234–254.

[61] Roy Nissim and Oded Schwartz. 2023. Accelerating Distributed Matrix Multiplication with 4-Dimensional Polynomial Codes. In *SIAM Conference on Applied and Computational Discrete Algorithms (ACDA23)*. SIAM, 134–146.

[62] James S Plank, Kai Li, and Michael A Puening. 1998. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems* 9, 10 (1998), 972–986.

[63] Dedy Septono Catur Putranto, Rini Wisnu Wardhani, Harashta Tatimma Larasati, and Howon Kim. 2023. Space and Time-Efficient Quantum Multiplier in Post Quantum Cryptography Era. *IEEE Access* 11 (2023), 21848–21862.

[64] Amirhossein Reisizadeh, Saurav Prakash, Ramtin Pedarsani, and Amir Salman Avestimehr. 2019. Coded computation over heterogeneous clusters. *IEEE Transactions on Information Theory* 65, 7 (2019), 4227–4242.

[65] Mahdi Sajadieh, Mohammad Dakhilalian, Hamid Mala, and Behnaz Omoomi. 2012. On construction of involutory MDS matrices from Vandermonde Matrices in GF (2 q). *Designs, Codes and Cryptography* 64 (2012), 287–308.

[66] Peter Sanders and Jop F Sibeyn. 2003. A bandwidth latency tradeoff for broadcast and reduction. *Inform. Process. Lett.* 86, 1 (2003), 33–38.

[67] Albin Severinson, Alexandre Graell i Amat, and Eirik Rosnes. 2018. Block-diagonal and LT codes for distributed computing with straggling servers. *IEEE Transactions on Communications* 67, 3 (2018), 1739–1753.

[68] Marc Snir, Robert W Wisniewski, Jacob A Abraham, Sarita V Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, Andrew A Chien, Paul Coteus, Nathan A DeBardeleben, Pedro C Diniz, Christian Engelmanm, Mattan Erez, Saverio Fazzari, Al Geist, Rinku Gupta, Fred Johnson, Sriram Krishnamoorthy, Sven Leyffer, Dean Liberty, Subhasish Mitra, Todd Munson, Rob Schreiber, Jon Stearley, and Eric Van Hensbergen. 2014. Addressing failures in exascale computing. *The International Journal of High Performance Computing Applications* 28, 2 (2014), 129–173.

[69] Kyungrak Son and Wan Choi. 2022. Distributed Matrix Multiplication Based on Frame Quantization for Straggler Mitigation. *IEEE Transactions on Signal Processing* 70 (2022).

[70] Pedro Soto, Jun Li, and Xiaodi Fan. 2019. Dual Entangled Polynomial Code: Three-Dimensional Coding for Distributed Matrix Multiplication. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 5937–5945.

[71] Rashish Tandon, Qi Lei, Alexandros G Dimakis, and Nikos Karampatziakis. 2017. Gradient coding: Avoiding stragglers in distributed learning. In *International Conference on Machine Learning*. PMLR, 3368–3376.

[72] Andrei L Toom. 1963. The complexity of a scheme of functional elements realizing the multiplication of integers. In *Soviet Mathematics Doklady*, Vol. 3. 714–716.

[73] Shih-hsiung Tung. 1964. Harnack's inequality and theorems on matrix spaces. *Proc. Amer. Math. Soc.* 15, 3 (1964), 375–381.

[74] Robert A Van de Geijn and Jerrell Watts. 1997. SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency-Practice and Experience* 9, 4 (1997), 255–274.

[75] Ashish Vulimiri, Philip Brighten Godfrey, Radhika Mittal, Justine Sherry, Sylvia Ratnasamy, and Scott Shenker. 2013. Low latency via redundancy. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*. 283–294.

[76] Da Wang, Gauri Joshi, and Gregory Wornell. 2014. Efficient task replication for fast response times in parallel computation. In *The 2014 ACM international conference on Measurement and modeling of computer systems*. 599–600.

[77] Da Wang, Gauri Joshi, and Gregory Wornell. 2015. Using straggler replication to reduce latency in large-scale parallel computing. *ACM SIGMETRICS Performance Evaluation Review* 43, 3 (2015), 7–11.

[78] Sinong Wang, Jiashang Liu, and Ness Shroff. 2018. Coded sparse matrix multiplication. In *International Conference on Machine Learning*. PMLR, 5152–5160.

[79] Yi-Min Wang, Yennun Huang, Kiem-Phong Vo, Pe-Yu Chung, and Chandra Kintala. 1995. Checkpointing and its applications. In *Twenty-fifth International Symposium on fault-tolerant Computing. Digest of papers*. IEEE, 22–31.

[80] Paul B Yale. 2014. *Geometry and symmetry*. Courier Corporation.

[81] C-Q Yang and Barton P Miller. 1988. Critical path analysis for the execution of parallel and distributed programs. In *The 8th International Conference on Distributed*. IEEE Computer Society, 366–367.

[82] Qian Yu, Mohammad Maddah-Ali, and Salman Avestimehr. 2017. Polynomial Codes: an Optimal Design for High-Dimensional Coded Matrix Multiplication. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc.

[83] Qian Yu, Mohammad Ali Maddah-Ali, and A. Salman Avestimehr. 2020. Straggler Mitigation in Distributed Matrix Multiplication: Fundamental Limits and Optimal Coding. *IEEE Transactions on Information Theory* 66, 3 (2020), 1920–1933.

[84] Alberto Zanoni. 2009. Toom-Cook 8-way for long integers multiplication. In *2009 11th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. IEEE, 54–57.

[85] Alberto Zanoni. 2010. Iterative Toom-Cook methods for very unbalanced long integer multiplication. In *Proceedings of the 2010 International Symposium on Symbolic and Algebraic Computation*. 319–323.

[86] Dan Zuras. 1993. On squaring and multiplying large integers. In *Proceedings of IEEE 11th Symposium on Computer Arithmetic*. IEEE, 260–271.