Fault Tolerant Resource Efficient Matrix Multiplication

Noam Birnbaum
The Hebrew University of Jerusalem
bnoam@cs.huji.ac.il

Oded Schwartz
The Hebrew University of Jerusalem
odedsc@cs.huji.ac.il

Abstract

General-purpose hard-error resiliency solutions such as checkpoint-restart severely degrade performance. For numerical linear algebra, more efficient solutions incur lower overhead. Current solutions require a significant increase in the number of processors. Further, they are based on distributed algorithms that guarantee good performance only when the matrices are large enough to fill all the local memories. Otherwise, their inter-processor communication costs are asymptotically larger than the lower bounds dictate.

We obtain fault tolerant parallel matrix multiplication algorithms that reduce the resource overhead by minimizing both the number of additional processors and the communication costs. In particular, we reduce the number of additional processors from $\Theta(\sqrt{P})$ to 1 (or from $\Theta(h\sqrt{P})$ to h, where h is the maximum number of simultaneous faults), and we save a $\Theta(\log P)$ factor of the latency costs. Further, for local memories larger then the minimum required to store the input and output, we obtain fault tolerant adaptations of the 2.5D algorithm that significantly reduce the communication costs, with very few additional processors.

1 Introduction

Errors are a serious concern in high performance computing. Given the increase in machine size and decrease in operating voltage, hard errors (component failure) and soft errors (bit flip) are likely to become more frequent. Hardware trends predict two errors per minute up to two per second on exascale machines [2, 24, 11]. Here we address resiliency for hard errors. General-purpose hard-error resiliency solutions such as checkpoint-restart [22], and diskless-checkpoints [22] successfully meet this challenge but are costly and severely degrade performance. Our methods enable efficient resource utilization and high performance for error-resilient algorithms that are close to the efficiency and performance of non-resilient algorithms.

For numerical linear algebra computations, certain efficient solutions incur considerably lower overhead by combining error correcting codes with matrix computations. These solutions are based on distributed 2D algorithms, and can guarantee high performance only when matrices fill all local memories. Otherwise, their inter-processor communication costs become asymptotically larger than the lower bounds, which degrades performance. Moreover, current solutions require a significant increase in the number of the processors.

Here we present new fault tolerant algorithms for matrix multiplication that reduce the number of additional processors and guarantee good inter processors communication costs. Specifically, for the 2D case, we decrease the number of additional processors from $\Theta(\sqrt{P})$ to 1 (or from $\Theta(h\sqrt{P})$ to h, where h is the maximum number of simultaneous faults), and we save a $\log P$ factor of the latency cost. We attain the bandwidth lower bound when $f = O\left(\sqrt{P}\right)$, where f is the total number of faults. When local memories are larger than the minimum needed to store inputs and outputs, we reduce the communication costs using 2.5D technique, with no (or very few) additional processors, attaining the bandwidth lower bound for $f = O\left(\sqrt{P/c}\right)$.

The model: computation, communication, and faults. Our computation model is a distributed machine with P processors, each having a local memory of size M words. The processors communicate via message passing. We assume that the cost of sending a message is proportional to its length and does not depend on the identity of the sender or receiver as in [19], and in the context of fault tolerance [12, 13]. This assumption can be alleviated with predictable impact on communication cost, cf. [3]. The number of arithmetic operations is denoted by F. The bandwidth cost of the algorithm is given by the words count and is denoted by BW. The latency cost is given by the message count and is denoted by L. We count the number of words, messages and arithmetic operations along the *critical path* as defined in [30]. The total runtime is modeled by $\gamma \cdot F + \beta \cdot BW + \alpha \cdot L$, where α, β, γ are machine-dependent parameters.

We denote by h the maximum number of faults that

can occur simultaneously; i.e., the maximum number of faults in one step or iteration of the algorithm, and by f the total number of faults throughout the execution. When comparing an algorithm to a fault tolerant adaptation, we use (P, M, F, BW, L) to denote the resources used by the original algorithm and (P', M', F', BW', L') to denote the resources used by the fault tolerant adaptation. We express the latter as a function of the former, of h, f, and the input size n. When a fault occurs, the faulty processor loses all its data, and the machine allocates a new processor to replace the faulty one. For simplicity, we assume no faults occur during recovery phases. Note that faults during the recovery phase of any of the algorithms we present may introduce at most a constant factor overhead to the recovery phase, and thus do not affect our analysis.

1.1 Previous work

Communication costs of (non-resilient) matrix multiplication. Cannon [10], Van De Geijn and Watts [27], and Fox et al. [15] proposed matrix multiplication algorithms that minimize communication when the memory is the minimum needed to store input and output, namely $M = \Theta\left(\frac{n^2}{P}\right)$. The communication costs of these algorithms (also known as 2D algorithms) are $(BW, L) = \left(O\left(\frac{n^2}{\sqrt{P}}\right), O\left(\sqrt{P}\right)\right)$. Agarwal et al. [1] put forward a 3D algorithm that uses less communication when additional memory is available: For $M = \Theta\left(\frac{n^2}{p^{2/3}}\right)$, they obtained $(BW, L) = \left(O\left(\frac{n^2}{P^{3/2}}\right), O\left(\log P\right)\right)$. McColl and Tiskin [20], and Solomonik and Demmel [25] generalized these algorithms to cover the entire relevant range of memory size, namely $\Theta\left(\frac{3n^2}{P}\right) \leq M \leq \Theta\left(\frac{n^2}{p^{2/3}}\right)$. The communication costs of their 2.5D algorithm are $(BW, L) = \left(O\left(\frac{n^3}{P\cdot\sqrt{M}}\right), O\left(\frac{n^3}{P\cdot M^{3/2}} + \log c\right)\right)$ where c is the memory redundant factor, namely $c = \Theta\left(\frac{P\cdot M}{n^2}\right)$. For $M > \Theta\left(\frac{n^2}{P^{2/3}}\right)$, the communication costs are bounded below by the memory independent lower bound of $\Omega\left(\frac{n^2}{P^{2/3}}\right)$ [4], therefore increasing c beyond $\sqrt[3]{P}$ cannot help reduce communication costs. McColl and Tiskin [20], Ballard et al. [5], and Demmel et al. [14] used an alternative parallelization technique for recursive algorithms, such as classical and fast matrix multiplication. The communication costs of the 2.5D algorithm and the BFS-DFS scheme applied to classical matrix multiplication are both optimal (up to an $O\left(\log P\right)$ factor), since they attain both the lower bound in Irony, Toledo and Tiskin [19], in the range $\Theta\left(\frac{n^2}{P}\right) \leq M \leq \Theta\left(\frac{n^2}{P^{2/3}}\right)$, and the lower bound in Ballard et al. [4] for larger M values.

Checkpoints-restart. One general approach to handling faults is checkpoint-restart; all the data and states of the processors are periodically saved to disk. Upon a fault, the machine loads the most recent checkpoint. This solution requires disks, which are expensive, for storing the checkpoints and incurs many I/O operations, which are time consuming.

Plank, Li, and Puening [22] suggested using a local memory for checkpoints instead of disks. This solution does not require additional hardware, and the writing and reading of checkpoints are faster. Still, the periodic write operations, as well as the restart operations significantly slow down the algorithms. Furthermore, this solution takes up some of the available memory from the algorithm. For many algorithms, matrix multiplication included, less memory implies a significant increase in communication cost, hence a slowdown.

Algorithm-based fault tolerance. Huang and Abraham [18] suggested algorithm-based fault tolerancy for classic matrix multiplication. The main idea was to add a row to A which is the sum of rows, and a column to B which is the sum of columns. The product of the two resulting matrices is the matrix $A \cdot B$ with an additional row containing the sum of its rows and an additional column containing the sum of its columns. They addresses soft errors, and showed that by using the sum of rows and columns it is possible to locate and fix one faulty element of C.

Huang and Abraham [18] used a technique that allows recovery from a single fault throughout the entire execution. Gunnels et al. [16] presented fault tolerant matrix multiplication that can detect errors in the input, and distinct between soft errors and round-off errors. Chen and Dongarra showed that by using the technique of [18], combined with matrix multiplication as in Cannon [10] and Fox [15] does not allow for fault recovery in the middle of the run, but only at the end. This severely restricts the number of faults an algorithm can withstand. Chen and Dongarra [12, 13] adapted the approach described by Huang and Abraham for hard error resiliency, using the *outer-product* [28] multiplication as a building block. Their algorithm keeps the partially computed matrix C encoded correctly, in the inner steps of the algorithm, not only at the end. By so doing, they were able to recover from faults occurring in the middle of a run without recomputing all the lost data. In [13] they analyzed the overhead when at most one fault occurs at any given time. In [12] they suggested an elegant multiple faults recovery generaliza-

tion of this algorithm. For this purpose they introduced a new class of useful erasure correcting codes. There algorithm requires $2h \cdot \sqrt{P}$ additional processors to be able to deal with up to h simultaneous faults. Further, they analyzed its numerical stability. Wu [29] et al. used outer product for soft error resiliency.

Hakkarinen and Chen [17] presented a fault tolerant algorithm for 2D Cholesky factorization. Bouteiller et al. [9] expanded this approach and obtained hard error fault tolerant 2D algorithms for matrix factorization computations. Moldaschl et al. [21] extended the Huang and Abraham scheme to the case of soft errors with memory redundancy. They considered bit flips in arbitrary segments of the mantissa and the exponent, and showed how to tolerate such errors with small overhead.

1.2 Our contribution. We introduce a new coding technique as well as ways to apply it to both 2D and 2.5D matrix multiplication algorithms. By doing so we obtain fault tolerant algorithms for matrix multiplication. Specifically in the 2D case we use only h additional processors, the minimum possible, and we use even fewer processors for the 2.5D algorithm. The runtime overhead is low, and the algorithms can utilize additional memory for communication minimizing. These algorithms can also handle multiple simultaneous faults. We also obtain a new efficient way for pipelining broadcast and reduce operations.

1.3 Paper organization. In section 2 we provide preliminaries including a new efficient pipelined reduce operation. In Section 3 we focus on the minimum memory case, with resiliency for a single fault (see Table 1). In Section 4 we show how to extend this approach to tolerate multiple faults (see Table 2). In Section 5 we show how to combine our algorithms with methods that utilize additional memory (see Table 3). In Section 6 we compare the algorithms and discuss some open questions.

2 Preliminaries

2.1 Pipeline reduce operations. We use broadcast and reduce operations in our algorithms. Sanders and Sibeyn [23] showed an efficient algorithm for performing broadcast and reduce.

LEMMA 2.1. ([23]) *Let P be the number of processors, and W the data size of each processor. It is possible to compute a weighed sums of the data of the P processors, using:* $(F, BW, L) = (O(W), O(W), O(\log P))$

We introduce an efficient way to perform l reduce operation in a row. The naïve implementation uses the algorithm above l times and requires $(F, BW, L) = (O(l \cdot W), O(l \cdot W), O(l \cdot \log P))$. We pipeline the reduce operations and save latency.

LEMMA 2.2. (EFFICIENT MULTIPLE WEIGHED SUM) *Let $P + l$ be the number of processors, and W the data size on P of them. It is possible to compute l weighed sums of the data of the P processors on the l other processors with resources:* $(F, BW, L) = (O(l \cdot W), O(l \cdot W), O(\log P + l))$

Proof. We first describe an alternative algorithm for a single weighted sum, then explain how it pipelines efficiently. The algorithm for one weighted sum has two phases. For ease of presentation, assume P is an integer power of 2 (the generalization is straightforward). The first phase reduces the weighed sum but the data remains distributed. The second phase gathers the data to the destination processor. The reduce works as follows. It divides the processors into two sets. Each set performs half of the task. The division involves communicating half of the data. Each set recursively calls the reduce. The base case is when each set contains only one processor. Then each processor holds $\frac{1}{P}$ fraction of the results. Next we gather the data to the additional processor. The reduction phase costs $F = \sum_{i=0}^{\log 2P} \frac{W}{2^i} = O(W)$, $BW = \sum_{i=1}^{\log 2P} \frac{W}{2^i} = O(W)$, and $L = \log_2 P$. The gathering costs $BW = \sum_{i=1}^{\log_2 P} \frac{W}{2^i} = O(W)$, and $L = \log_2 P$. Thus the total cost of the single wighted sum algorithm is:
$(F, BW, L) = (O(W), O(W), O(\log P))$

This algorithm can be efficiently pipelined since the messages size decreases exponentially. Let the names of the processors be a binary string of length $\log_2 P$. In the first phase the communication is between pairs of processors that agree on all the digits aside from the first digit. They communicate the first weighted sum. In the second step the communication is between processors that agree on all digits aside from the second and they send the second step of the first reduce the first step of the second reduce, and so on. Each weighted sum takes at most $O(\log P)$ steps and then the data are sent to one of the l new processors. Therefore at any time at most $O(\log P)$ weighted sums are being computed. The memory required for all the reduces that can occur in parallel is at most $\sum_{i=1}^{\log P} \frac{W}{2^i} \leq 2W$, and the memory required for all the gathering is at most $\sum_{i=1}^{\log P} \frac{W}{2^i} \leq 2W$. Therefore the memory footprint of this algorithm is $M \leq 4W$. In summary, performing l reduce operations in a row with this algorithm uses local memories of size $4W$ costs: $(F, BW, L) = (O(l \cdot W), O(l \cdot W), O(\log P + l))$

2.2 Linear erasure code We use linear erasure code for recovering faults.

DEFINITION 2.1. *(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 $x \neq y \in \mathbb{R}^k$, $T(x), T(y)$ have at least d coordinates with different values. The generator matrix of T is an $n \times k$ matrix G such that $T(x) = G \cdot x$.*

The erasures code we use preserve the original word and add redundant letters. Formally we code 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, the code generating matrix is of the form $G = \begin{pmatrix} I_k \\ E_{n-k,k} \end{pmatrix}$.

3 Minimum memory, single fault

In this section we discuss previous and new fault tolerant algorithms, for $M = \Theta\left(n^2/P\right)$.

3.1 Previous algorithms. Chen and Dongarra [12, 13] used the Huang and Abraham scheme [18] to tolerate hard errors. Specifically, they added one row of processors that store the sum of the rows[1] of A and similarly for C , and one column of processors that store the sum of the columns of B and similarly for C. They called these rows and columns the *check-sum*; a matrix that has both is called a *fully check-sum* matrix.

Chen and Dongarra showed that this approach, applied to 2D algorithms (e.g., Cannon [10] and Fox [15]), allows for the recovery of C at the end of the matrix multiplication. However these 2D algorithms do not preserve the check-sum during the inner steps of the algorithm. To deal with higher fault rate, that requires recovery of faults during the run of the algorithm, Chen and Dongarra used the *outer product* as the building block of their algorithm. Thus their algorithm can recover faults throughout the run of the algorithm at the end of each outer product iteration. Lost data of A and C of a faulty processor can be recovered at the end of every outer product step, from the processors of the same column from the processor in the check-sum row. Similarly, the data from B and C can be recovered using the check-sum column.

THEOREM 3.1. ([12, 13]) *Consider a 2D communication optimal matrix multiplication algorithm with resources (P, F, BW, L). Let (P', F', BW', L') be the resources required for the fault tolerant 2D matrix multiplication algorithm of Chen and Dongarra that can withstand a single fault at any given time. Let n be the matrix dimension and let f be the total number of faults. Then:*

$$(P', F', BW', L') = \left(P + 2\sqrt{P} + 1, F \cdot \left(1 + O\left(\frac{f}{n}\right)\right), BW \cdot O\left(\log P + \frac{f}{\sqrt{P}}\right), L \cdot O\left(\log P\right) + O\left(f \cdot \log P\right)\right).$$

A proof can be found in [8].

Our algorithms. Since matrices A and B are not modified by the algorithm, lost input data can be easily handled using an erasure code of length $P + 1$. The main challenge involves recovering C. To this end, we introduce two alternatives: the first algorithm uses the outer product and encoding of the blocks of C with additional processors. This is similar to the approach by Chen and Donagarra. However we use a new coding scheme that decreases the additional processor count from $\Theta\left(\sqrt{P}\right)$ to one. We denote this algorithm by slice-coded algorithm. The second algorithm recovers the lost data of C by recomputing at the end of the run. We denote this the posterior-recovery algorithm.

THEOREM 3.2. (SLICE-CODED) *Consider a 2D communication cost optimal matrix multiplication algorithm with resources (P, F, BW, L). Then there exists a fault tolerant 2D matrix multiplication algorithm that can withstand a single fault at any given time, where n is the matrix dimension and f is the total number of faults, with resources: $(P', F', BW', L') = \left(P + 1, F \cdot \left(1 + O\left(\frac{f}{n}\right)\right), BW \cdot O\left(1 + \frac{f}{\sqrt{P}}\right), O\left(L \cdot \log P\right)\right)$.*

THEOREM 3.3. (POSTERIOR-RECOVERY) *Consider a 2D communication cost optimal matrix multiplication algorithm with resources (P, F, BW, L). Then there exists a fault tolerant 2D matrix multiplication algorithm that can withstand a single fault at any given time, where n is the matrix dimension and f is the total number of faults, with resources $(P', F', BW', L') = \left(P + 1, F \cdot \left(1 + O\left(\frac{1}{\sqrt{P}}\right)\right), BW \cdot \left(1 + O\left(\frac{f}{\sqrt{P}}\right)\right), L + O\left(f \cdot \log P\right)\right)$.*

3.2 Slice-coded algorithm. We follow the approach in [12, 13], and use the outer product matrix multiplication as the basis for the algorithm. However, where they used $2 \cdot \sqrt{P} + 1$ additional processors, for the coded data, we only use one. The additional processor contains the sum of the others. This processor acts similarly to the corner processor in Chen and Dongarra's algorithm (corresponding to the red processor in Figure 1). It contains the sum of the blocks of A,B, and C. In the s iteration of the algorithm, it multiplies

[1] From here on we write rows to refer to block rows, and columns to refer to block columns.

Table 1: Fault tolerant algorithms for 2D algorithms, namely $M = \theta\left(\frac{n^2}{P}\right)$ with at most one simultaneous fault. n is the matrix dimension, P is the number of processors, and f is the total number of faults occurring throughout the run of the algorithm.

Algorithm	F' (flops per processor)	BW' (bandwidth per processor)	L' (latency per processors)	Additional processors
Cannon [10], SUMMA [27] (No fault)	$F = \frac{2n^3}{P}$	$BW = O\left(\frac{n^2}{\sqrt{P}}\right)$	$L = O\left(\sqrt{P}\right)$	-
Previous algorithm [12, 13]	$F \cdot (1 + o(f))$	$BW \cdot O(\log P + o(f))$	$O(L \cdot \log P)$	$2\sqrt{P} + 1$
Slice-coded [here, Theorems 3.2]	$F \cdot (1 + o(f))$	$BW \cdot O(1 + o(f))$	$O(L \cdot \log P)$	1
Posterior-recovery [here, Theorems 3.3]	$F \cdot (1 + o(f))$	$BW \cdot (1 + o(f))$	$L + O(f \cdot \log P)$	1

the sum of the current row with the sum of the current column. Thus it keeps the sum of the blocks of C updated. In each outer-product iteration the algorithm computes the sum of the current outer product. We show in Equation 3.1 that the sum of the blocks of C can be computed by multiplying two sums of blocks.

(3.1)

$$\begin{aligned}\sum_{i,j=1}^{n} \left(A(:,s) \cdot B(s,:)\right)_{i,j} &= \sum_{i,j=1}^{n} A_{i,s} \cdot B_{s,j} \\ &= \left(\sum_{i=1}^{n} A_{i,s}\right) \cdot \left(\sum_{j=1}^{n} B_{s,j}\right)\end{aligned}$$

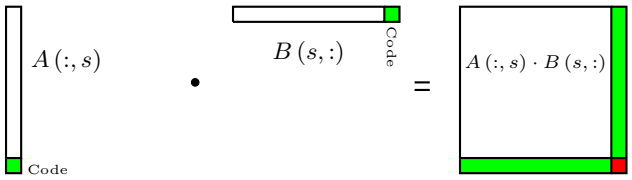

Figure 1: An iteration of Chen and Dongarra's algorithm. Each column of A and row of B contains a check-sum processor.

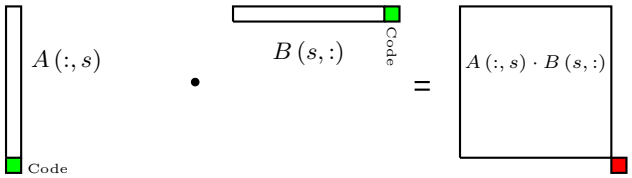

Figure 2: An iteration of our slice-coded algorithm. The algorithm computes the green parts and sends them to the additional processors (red).

Proof. [Proof of Theorem 3.2] The algorithm allocates one additional processor for the code, thus $P' = P + 1$. The algorithm is composed of three steps. In the first step, code creation *(CC)* the algorithm creates codes for A and B and stores them in the additional processor. The second step is the matrix multiplication *(MM)*. Upon a fault, a recovery *(Re)* step is performed. Therefore F' is composed of three components, namely, $F' = F_{\text{CC}} + F_{\text{MM}} + F_{\text{Re}}$. Similarly $BW' = BW_{\text{CC}} + BW_{\text{MM}} + BW_{\text{Re}}$, and $L' = L_{\text{CC}} + L_{\text{MM}} + L_{\text{Re}}$.

Code creation. In this step, the algorithm computes the sum of the blocks of A and of B, and stores them in the additional processor,using a reduce operation. By Lemma 2.1 this takes:

(3.2)

$$(F_{\text{CC}}, BW_{\text{CC}}, L_{\text{CC}}) = \left(O\left(\frac{n^2}{P}\right), O\left(\frac{n^2}{P}\right), O(\log P)\right)$$

Matrix multiplication. The matrix multiplication phase is performed as in an outer-product algorithm with a small change: every processor computes its share of the code. To be more precise, in the s^{th} iteration (of $\sqrt{P}$ iterations) the processors compute the outer product $A(:,s) \cdot B(s,:)$. The processors of the current block column of A and the processors of the current block row of B broadcast them. The processors compute the sum of the current block column of A; specifically each column of processors computes $1/\sqrt{P}$ of this sum. Similarly, the processors compute the sum of the current block row of B. The processors send these two sums to the additional processor. Then each processor multiplies the two blocks.

By Theorem 2.2 the broadcasting *(B)* takes $(F_{\text{B}}, BW_{\text{B}}, L_{\text{B}}) = \left(0, O\left(\frac{n^2}{P}\right), O(\log P)\right)$. The reduce operation is distributed among the rows and the columns, where each row and column of processors performs a reduce operation with an $\frac{n^2}{P^{3/2}}$ block size. There-

fore this reduce operation *(R)* takes:

$$(F_{\mathrm{R}}, BW_{\mathrm{R}}, L_{\mathrm{R}}) = \left(O\left(\frac{n^2}{P^{3/2}}\right), O\left(\frac{n^2}{P^{3/2}}\right), O\left(\log P\right)\right).$$

The multiplication of two blocks in time is $\frac{2n^3}{P^{3/2}}$. There are $\sqrt{P}$ iterations; thus the multiplications takes:

(3.3)

$$\begin{aligned}(F_{\mathrm{MM}}, BW_{\mathrm{MM}}, L_{\mathrm{MM}}) &= (\frac{2n^3}{P} + O\left(\frac{n^2}{P^{3/2}} \cdot \sqrt{P}\right), \\ &\quad O\left(\frac{n^2}{\sqrt{P}}\right), O\left(\sqrt{P}\log P\right))\end{aligned}$$

Recovery. Each recovery is a reduce operation. By Lemma 2.1 f recoveries take:

(3.4)

$$\begin{aligned}&(F_{\mathrm{Re}}, BW_{\mathrm{Re}}, L_{\mathrm{Re}}) \\ &= \left(f \cdot O\left(\frac{n^2}{P}\right), f \cdot O\left(\frac{n^2}{P}\right), f \cdot O\left(\log P\right)\right)\end{aligned}$$

Total costs. Summing up Equations 3.2, 3.3, and 3.4 we have

$$\begin{aligned}F' &= F_{\mathrm{CC}} + F_{\mathrm{MM}} + F_{\mathrm{Re}} \\ &= \frac{2n^3}{P} + O\left(\frac{n^2}{P} + \frac{n^2}{P} + \frac{f \cdot n^2}{P}\right) \\ &= F \cdot \left(1 + O\left(\frac{f}{n}\right)\right) \\ BW' &= BW_{\mathrm{CC}} + BW_{\mathrm{MM}} + BW_{\mathrm{Re}} \\ &= BW \cdot O\left(1 + \frac{f}{\sqrt{P}}\right) \\ L' &= L_{\mathrm{CC}} + L_{\mathrm{MM}} + L_{\mathrm{Re}} \\ &= L \cdot \log P\end{aligned}$$

3.3 Posterior-recovery. In this algorithm we recover output by re-computation. That is, A and B input matrices are coded, but C is not. A faulty processor incurs the restoration of its share of A and B. But recomputing its lost share of the workload is performed at the end of the algorithm, using all processors. When a fault occurs, the algorithm recovers the lost data of A and B using their code, initializes the lost block of C to zeros, and resumes computations.

DEFINITION 3.1. *We denote by a* cube *the set of scalar multiplications defined by the two blocks (sub-matrices) multiplication.*

Proof. [Proof of Theorem 3.3] We assume that at each iteration, at most one fault occurs. Therefore the algorithm needs only one additional processor to encode A and B, namely, $P' = P + 1$.

$F' = F_{\mathrm{CC}} + F_{\mathrm{MM}} + F_{\mathrm{ReIn}} + F_{\mathrm{ReOut}}$, where CC stands for code creation, MM for the matrix multiplication, ReIn for the recovery of the input A and B, and ReOut for the recomputation. Similarly $BW' = BW_{\mathrm{CC}} + BW_{\mathrm{MM}} + BW_{\mathrm{ReIn}} + BW_{\mathrm{ReOut}}$, and $L' = L_{\mathrm{CC}} + L_{\mathrm{MM}} + L_{\mathrm{ReIn}} + L_{\mathrm{ReOut}}$.

Code creation. The costs of this phase are as in the Slice-coded algorithm see Section 3.2.

Matrix multiplication. The algorithm performs 2D matrix multiplication (e.g., Cannon's [10]), thus

(3.5) $$(F_{\mathrm{MM}}, BW_{\mathrm{MM}}, L_{\mathrm{MM}}) = (F, BW, L).$$

Input recovery. By Lemma 2.1 the costs of f recoveries are:

(3.6)

$$\begin{aligned}&(F_{\mathrm{ReIn}}, BW_{\mathrm{ReIn}}, L_{\mathrm{ReIn}}) = \\ &\left(O\left(f \cdot \frac{n^2}{P}\right), O\left(f \cdot \frac{n^2}{P}\right), O\left(f \cdot \log P\right)\right)\end{aligned}$$

Output recovery. This stage involves communication, multiplication, and reducing the data. We assume that the maximum number of faults in an iteration is 1. Each processor computes $\sqrt{P}$ cubes. Therefore there are at most P cubes to compute again, as there are $\sqrt{P}$ iterations. The algorithm distributes the workload of the lost cubes. Each processor gets at most one cube. Since computing a cube is multiplying two blocks of size $\frac{n^2}{P}$ it takes $F_{\mathrm{ReOut}} = O\left(\frac{n^3}{P^{3/2}}\right)$ flops. The communication cost is due to moving two input blocks and the reduce of C. Thus it takes $BW_{\mathrm{ReOut}} = O\left(\frac{n^2}{P}\right)$, and $L_{\mathrm{ReOut}} = O\left(\log P\right)$. We have

(3.7)

$$\begin{aligned}&(F_{\mathrm{ReOut}}, BW_{\mathrm{ReOut}}, L_{\mathrm{ReOut}}) = \\ &\left(O\left(\frac{n^3}{P^{3/2}}\right), O\left(\frac{n^2}{P}\right), O\left(\log P\right)\right)\end{aligned}$$

Total costs. Summing up Equations 3.2, 3.5, 3.6, and 3.7 we have

$$\begin{aligned}F' &= F_{\mathrm{CC}} + F_{\mathrm{MM}} + F_{\mathrm{ReIn}} + F_{\mathrm{ReOut}} \\ &= F + O\left(\frac{n^3}{P^{3/2}}\right) \\ &= F \cdot \left(1 + O\left(\frac{1}{\sqrt{P}}\right)\right) \\ BW' &= BW_{\mathrm{CC}} + BW_{\mathrm{MM}} + BW_{\mathrm{ReIn}} + BW_{\mathrm{ReOut}} \\ &= BW \cdot \left(1 + O\left(\frac{f}{\sqrt{P}}\right)\right) \\ L' &= L_{\mathrm{CC}} + L_{\mathrm{MM}} + L_{\mathrm{ReIn}} + L_{\mathrm{ReOut}} \\ &= O\left(\log P\right) + L + O\left(f \log P\right) + O\left(\log P\right) \\ &= L + O\left(f \log P\right)\end{aligned}$$

Table 2: Fault Tolerant 2D algorithms, namely, $M = \Theta\left(\frac{n^2}{P}\right)$ with at most h simultaneous faults. n is the matrix dimension, P is the number of processors, and f is the total number of faults occurring throughout the run of the algorithm.

Algorithm	F' (flops per processor)	BW' (bandwidth per processor)	L' (latency per processors)	Additional processors
Cannon [10], SUMMA [27] (No fault)	$F = \frac{2n^3}{P}$	$BW = O\left(\frac{n^2}{\sqrt{P}}\right)$	$L = O\left(\sqrt{P}\right)$	–
Previous algorithm [12]	$F \cdot \left(1 + O\left(\frac{h+f}{n}\right)\right)$	$BW \cdot O\left(1 + \frac{h+f}{\sqrt{P}}\right)$	$O\left(L \cdot \log P + f \cdot \log P + h\right)$	$2h\sqrt{P} + h^2$
Slice-coded [here, Theorem 4.2]	$F \cdot \left(1 + O\left(\frac{h+f}{n}\right)\right)$	$BW \cdot O\left(1 + \frac{h+f}{\sqrt{P}}\right)$	$O\left(L \cdot \log P + h + f\right)$	h
Posterior-recovery [here, Theorem 4.3]	$F \cdot \left(1 + O\left(\frac{h}{n} + \frac{f}{P}\right)\right)$	$BW \cdot \left(1 + O\left(\frac{h+f}{\sqrt{P}}\right)\right)$	$L + O\left(f \cdot \log P + h\right)$	h

4 Multiple faults

In this section we extend our algorithms to a number of simultaneous faults.

4.1 Previous algorithm.

THEOREM 4.1. ([12]) *Consider a 2D communication cost optimal matrix multiplication algorithm with resources (P, F, BW, L). Let n be the matrix dimension. Then there exists a fault tolerant 2D matrix multiplication algorithm that can withstand h simultaneous faults at any given time, and f total faults, with resources* $(P, F, BW, L) = \left(P + 2 \cdot h \cdot \sqrt{P} + h^2, F \cdot \left(1 + O\left(\frac{h+f}{n}\right)\right), BW \cdot O\left(\log P + \frac{h+f}{\sqrt{P}}\right), O\left(\left(f + \sqrt{P}\right) \log P + h\right)\right)$.

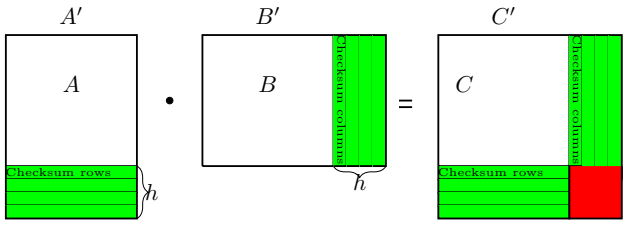

Figure 3: Previous algorithm for dealing with h simultaneous faults[12].

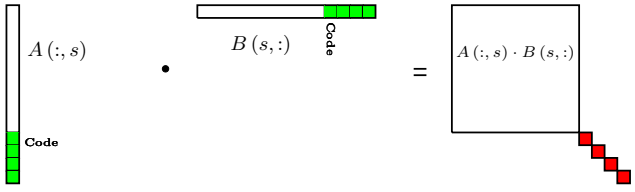

Figure 4: An iteration of the slice-coded algorithm.

The algorithm adds h rows of processors to A and C and h columns of processors to B and C. It stores weighted sums of the original processors in the additional processors. Their algorithm uses $\left(\sqrt{P}, \sqrt{P} + h, h + 1\right)$-code, with a generating matrix $G = \begin{pmatrix} I_k \\ E_{n-k,k} \end{pmatrix}$ such that every minor of E is invertible. It can therefore recover h simultaneous faults, even if they occur in the same row or column of processors. We provide a proof for Theorem 4.1 in [8].

4.2 Erasure correcting code. For multiple faults we use erasure code ()recall the definition in Section 2.2). To withstand h simultaneous faults we require a $(P, P + h, h + 1)$-code. In other words, any P letters suffice to recover the data. This is possible if and only if every minor of size P of the generating matrix $G = \begin{pmatrix} I_P \\ E_{h \times P} \end{pmatrix}$ is invertible. In other words, every minor of E is invertible.

Similar to the single fault case, each code processor multiplies a weighed sum of the current block column of A with a weighed sum of the current block row of B, and adds it to the accumulated sum. Thus the weighed sum is of the form:

$$\begin{aligned} (4.8) \qquad & \left(\sum_{i=1}^{n} u_i \cdot A_{i,k}\right) \cdot \left(\sum_{j=1}^{n} v_j \cdot B_{k,j}\right) \\ = & \sum_{i,j=1}^{n} v_i \cdot u_j \cdot A_{i,k} \cdot B_{k,j} \\ = & \sum_{i,j=1}^{n} \left(w_{i,j} \cdot A\left(:, k\right) \cdot B\left(k, :\right)\right)_{i,j} \end{aligned}$$

where $w_{i,j} = v_i \cdot u_j$ for some vectors v and u. The code used in [12] does not have the above property, and therefore cannot be used for our purpose. We show that there exists a code with the required properties.

LEMMA 4.1. *There exists* $(P+h, P, h+1)$ *code, such that the generating matrix* $G = \begin{pmatrix} I \\ E \end{pmatrix}$ *has the following property. For every row* i *of* E *there exists two vectors* $v^i, u^i \in \mathbb{R}^{\sqrt{P}}$ *such that* $E_i = v^i \otimes u^i$. *Namely* $E_{i,a+(\sqrt{P}-1)b} = v^i_a \cdot u^i_b$.

Proof. Consider an erasure code with generating matrix $\begin{pmatrix} I \\ E \end{pmatrix}$, where $I = I_P$, and E is an $h \times P$ Vandermonde matrix. Every minor of the Vandermonde matrix is invertible. The i^{th} row of the Vandermonde matrix is of the form $r^i = (\alpha_i^0, \alpha_i^1, \ldots, \alpha_i^{P-1})$. By taking $v^i = \left(\alpha_i^0, \ldots, \alpha_i^{\sqrt{P}-1}\right)$ and $u^i = \left(\alpha_i^0, \alpha_i^n, \ldots, \alpha_i^{P-\sqrt{P}}\right)$, we obtain that $E_{i,a+(\sqrt{P}-1)b}$ is $v_i \cdot u_j = r^i_{a+(\sqrt{P}-1)b}$.

4.3 Slice-coded.

THEOREM 4.2. (SLICE-CODED ALGORITHM) *Consider a 2D communication cost optimal matrix multiplication algorithm with resources* (P, F, BW, L). *Let* n *be the matrix dimension. Then there exists a fault tolerant 2D matrix multiplication algorithm that can withstand* h *simultaneous faults at any given time, and* f *total faults, with resources* $(P', F', BW', L') = (P+h, F \cdot \left(1 + O\left(\frac{h+f}{n}\right)\right),$ $BW \cdot O\left(1 + \frac{h+f}{\sqrt{P}}\right), L \cdot \log P + O\left((h+f) \cdot \log P\right))$

Proof. We showed in Section 4.2 how to use h additional processors to obtain a code with distance $h+1$; thus $P' = P + h$. The rest of the analysis is similar to the single fault case in the proof of Theorem 3.2. We have $F' = F_{\text{CC}} + F_{\text{MM}} + F_{\text{Re}}$, and similarly for BW' and L'.

Code creation. The algorithm first creates an erasure code for A and B. By Theorem 2.2 as $W = \frac{n^2}{P}$ and $l = h$ this takes:

$$(F_{\text{CC}}, BW_{\text{CC}}, L_{\text{CC}}) = \left(O\left(h \cdot \frac{n^2}{P}\right), O\left(h \cdot \frac{n^2}{P}\right), O\left(\log P + h\right)\right) \tag{4.9}$$

Matrix multiplication. The multiplication involves broadcasting and reduction of h weighted sums. Each column of processors computes $\frac{h}{\sqrt{P}}$ weighted sums of the blocks of A and each row of processors computes $\frac{h}{\sqrt{P}}$ weighted sums of the blocks of B. The broadcasting and reduction *(BR)* takes:

$$(F_{\text{BR}}, BW_{\text{BR}}, L_{\text{BR}}) = (O\left(\frac{h}{\sqrt{P}} \cdot \frac{n^2}{P}\right), O\left(\left(1 + \frac{h}{\sqrt{P}}\right) \cdot \frac{n^2}{P}\right), O\left(\log P + \frac{h}{\sqrt{P}}\right))$$

The multiplication of two blocks takes $O\left(\frac{n^3}{p^{3/2}}\right)$ flops. There are $\sqrt{P}$ iterations. Therefore,

$$(F_{\text{MM}}, BW_{\text{MM}}, L_{\text{MM}}) = (O\left(\frac{n^3 + h \cdot n^2}{P^2}\right), O\left(\frac{n^2}{\sqrt{P}}\left(1 + \frac{h}{\sqrt{P}}\right)\right), O\left(\sqrt{P} \cdot \log P + h\right)) \tag{4.10}$$

Recovery. When faults occur, the portion of A,B, and C of the faulty processor are recovered at the end of the iteration, using the erasure code. Assume that at iteration i that the number of faults is f_i. By Theorem 2.2, as $W = n^2/P$ and $l = f_i > 0$ the recovery takes:

$$(F_{\text{Re}_i}, BW_{\text{Re}_i}, L_{\text{Re}_i}) = \left(O\left(f_i \cdot \frac{n^2}{P}\right), O\left(f_i \cdot \frac{n^2}{P}\right), \log P + f_i\right) \tag{4.11}$$

Recall that $f = \sum_{i=1}^{\sqrt{P}} f_i$. Therefore,

$$(F_{\text{Re}}, BW_{\text{Re}}, L_{\text{Re}}) = (O\left(f \cdot \frac{n^2}{P}\right), O\left(f \cdot \frac{n^2}{P}\right), O\left(\min\left(f, \sqrt{P}\right) \log P + f\right)) \tag{4.12}$$

Total costs. Summing up Equations 4.9, 4.10, and 4.12 we have,

$$\begin{aligned} F' &= F_{\text{CC}} + F_{\text{MM}} + F_{\text{Re}} \\ &= F \cdot \left(1 + O\left(\frac{h+f}{n}\right)\right) \\ BW' &= O\left(h \cdot \frac{n^2}{P} + \frac{n^2}{\sqrt{P}}\left(1 + \frac{d}{\sqrt{P}}\right) + f \cdot \frac{n^2}{P}\right) \\ &= BW \cdot O\left(1 + \frac{h+f}{\sqrt{P}}\right) \\ L' &= O\left(\sqrt{P} \cdot \log P + h + f\right) \end{aligned}$$

4.4 Posterior-recovery.

This algorithm allocates h processors for encoding A and B. It runs a 2D matrix multiplication (e.g., Cannon [10] not just outer product ones). When a processor faults, the algorithm recovers A and B and proceeds. After the multiplication the algorithm re-computes the lost portion of C.

THEOREM 4.3. (POSTERIOR-RECOVERY) *Consider a 2D communication cost optimal matrix multiplication algorithm with resources* (P, F, BW, L). *Let* n *be the matrix dimension. Then there exists a fault tolerant 2D matrix multiplication algorithm that can withstand* h *simultaneous faults at*

any given time, and f total faults, with resources $(P', F', BW', L') = (P + h, F + O\left(h\frac{n^2}{P} + f\frac{n^3}{P^2}\right), BW + O\left((h+f)\cdot\frac{n^2}{P}\right), L + O\left(\sqrt{P}\cdot\log P + f + h\right)$

The analysis of this algorithm is very similar to the single fault case. We provide a proof for Theorem 4.3 in [8].

5 Memory redundancy

We next present the extension of our two algorithms to the case where redundant memory is available, namely $M = \Theta\left(\frac{cn^2}{P}\right)$ for some $1 < c < \sqrt[3]{P}$.

Fault distribution. Recall that a 2.5D algorithm splits the processors into c sets, where each set performs $\frac{1}{c}$ of the iteration of 2D algorithm. When h is the maximum number of simultaneous faults, each set of processors has to be able to tolerate h simultaneous faults. For ease of analysis we assume that the faults are distributed uniformly among the c sets. If this is not the case the algorithm can divide the computations differently, and assign more computation to a set that has fewer faults. This is possible since each set of processors has sufficient data to perform all these computations.

5.1 Slice-coded.

THEOREM 5.1. (SLICE-CODED) *Consider a 2.5D communication-cost optimal matrix multiplication algorithm with resources (P, F, BW, L). Let n be the matrix dimension and let M be the local memories size. Let c be the memory redundancy factor, namely $c = \Theta\left(\frac{P\cdot M}{n^2}\right)$. Then there exists a fault tolerant 2.5D matrix multiplication algorithm that can withstand h simultaneous faults at any given time, and f total faults, with resources:* $(P', F', BW', L') = (P + c\cdot h, F + O\left(\frac{h+f}{c}\cdot M\right), O\left(BW + \frac{h+f}{c}\cdot M\right), O\left(L\cdot\log P + \frac{h+f}{c}\right))$.

The proof of Theorem 5.1 is similar to the proof of Theorem 4.2 and is provided in [8]. Roughly speaking the fault tolerant 2.5D algorithm splits the processor into c sets ($c = \Theta\left(\frac{P\cdot M}{n^2}\right)$). Each set allocates h processors for code, and then creates code for A and B. The matrix multiplication is divided equally among the sets, and each set performs $\frac{1}{c}$ iteration of the outer product. During the run, the weighed sums in the code processors are kept updated. When faults occur in processors i^{th} set ($i \in [c]$), all processors in the i^{th} set (and only them) participate in the recovery phase.

5.2 Posterior-recovery. The 2.5D adaptation of the posterior-recovery algorithm is similar to the 2D case, with one main exception: there is an inherent redundancy in the replications of A and B in the 2.5D algorithm that we utilize to decreases the length of the code, hence reduces the number of additional processors required. If $h < c$, the algorithm does not require additional processors at all.

The algorithm splits the processors into c sets where $c = \Theta\left(\frac{P\cdot M}{n^2}\right)$. Each set performs $\frac{1}{c}$ of the iterations of a 2D algorithm (not necessarily the outer product algorithm). When a fault occurs, the processors in the set of the faulty processor wait for the recovery of that processor. The lost data of A and B are recovered from the next set of processors.

THEOREM 5.2. (POSTERIOR-RECOVERY) *Consider a 2.5D algorithm with resources (P, F, BW, L). Let n be the matrix dimension and let M be the local memories size. Let c be the memory redundant factor, namely $c = \Theta\left(\frac{P\cdot M}{n^2}\right)$. Then there exists a fault tolerant 2.5D matrix multiplication algorithm that can withstand h simultaneous faults at any given time, and f total faults, with resources:* $(P', F', BW', L') = (P, F\cdot\left(1 + O\left(\frac{f}{P}\right)\right), O\left(BW\right), O\left(L\cdot\log c + \log P\right))$.

Proof. As explained above $P' = P$. The algorithm does not create code, and similar to the 2D case it recovers the input immediately and re-computes the lost output data after the multiplication ends. Therefore $F' = F_{\text{MM}} + F_{\text{ReIn}} + F_{\text{re\_out}}$. Likewise $BW' = BW_{\text{MM}} + BW_{\text{ReIn}} + BW_{\text{re\_out}}$, and $L' = L_{\text{MM}} + L_{\text{ReIn}} + L_{\text{re\_out}}$.

Matrix multiplication. The algorithm performs a 2.5D matrix multiplication therefore,

$$(F_{\text{MM}}, BW_{\text{MM}}, L_{\text{MM}}) = (F, BW, L) \tag{5.13}$$

Input recovery. The algorithm recovers faults at the end of each iteration. Since $c > h$ there is at least one copy of each block even when h processors fault simultaneously. If k processors that hold the same block of A (or B) fault simultaneously, the algorithm broadcasts this block. Therefore in the worst case, this recovery requires $O\left(\log k\right)$ messages. Recall that in the i^{th} iteration $f_i < c$ processors fault. By Lemma 2.1 it costs:

$$(F_{\text{ReIn}_i}, BW_{\text{ReIn}_i}, L_{\text{ReIn}_i}) = (O\left(M\right), O\left(M\right), O\left(\log c\right))$$

thus the total recovery costs are:

$$\begin{aligned}(F_{\text{ReIn}}, BW_{\text{ReIn}}, L_{\text{ReIn}}) &= (O\left(\sqrt{P/c^3}\cdot M\right), \\ &\quad O\left(\sqrt{P/c^3}\cdot M\right), O\left(\sqrt{p/c^3}\cdot\log c\right)) \\ &= \left(O\left(\frac{n^3}{P\cdot\sqrt{M}}\right), O\left(BW\right), O\left(L\cdot\log c\right)\right)\end{aligned} \tag{5.14}$$

Table 3: Fault tolerant 2.5D algorithms, c copies of the input and the output fit into the memory; namely $c = \Theta\left(\frac{M \cdot P}{n^2}\right)$, where n is the matrix dimension, and P is the number of processors. f is the total number of faults occurring throughout the run of the algorithm. h is the maximum number of simultaneous faults.

Algorithm	F' (flops per processor)	BW' (bandwidth per processor)	L' (latency per processors)	Additional processors
2.5D [20, 25]	$F = \frac{2n^3}{P}$	$BW = O\left(\frac{n^3}{P \cdot \sqrt{M}}\right)$	$L = O\left(\frac{n^3}{P \cdot M^{3/2}}\right) + \log c$	–
Previous algorithm* [12]	$F + O\left(\frac{f \cdot n^2}{P}\right)$	$O\left(BW \cdot \sqrt{c} + \frac{(h+f) \cdot n^2}{P}\right)$	$O\left(L \cdot c^{3/2} \cdot \log P + h\right)$	$2h\sqrt{P} + h^2$
Slice-coded [here, Theorem 5.1]	$F + O\left(\frac{h+f}{c} \cdot M\right)$	$O\left(BW + \frac{h+f}{c} \cdot M\right)$	$O\left(L \cdot \log P + \frac{h+f}{c}\right)$	$c \cdot h$
Posterior-recovery† [here, Theorem 5.2]	$F + O\left(\frac{f \cdot n^3}{P^2}\right)$	$O(BW)$	$O(L \cdot \log c + \log P)$	0

* This algorithm does not utilize additional memory; hence its communication costs are larger. We do not include the algorithm suggested in Moldashcl et al. [21] as they handle soft errors only.

† We analyze this algorithm only when $h < c$.

Output recovery. After the 2.5D matrix multiplication is completed, the algorithm computes the lost cubes (recall Definition 3.1). When a processor faults it loses $O\left(\sqrt{P/c^3}\right)$ such cubes. Each processor gets $O\left(\frac{f \cdot \sqrt{P/c^3}}{P}\right) = O\left(\frac{f}{\sqrt{P \cdot c^3}}\right)$ such cubes for recomputing, and multiplies pairs of them. The block size is $\frac{n}{\sqrt{P/c}} \times \frac{n}{\sqrt{P/c}}$; therefore multiplying two blocks costs $\left(\frac{n}{\sqrt{P}}\right)^3$ flops. Thus the costs are:

$$
\begin{aligned}
F_{\text{ReOut}} &= O\left(\frac{f}{\sqrt{P \cdot c^3}} \cdot \frac{c^{3/2} \cdot n^3}{P^{3/2}}\right) \\
&= O\left(\frac{f \cdot n^3}{P^2}\right) \\
BW_{\text{ReOut}} &= O\left(\frac{f}{\sqrt{P \cdot c^3}} \cdot M\right) \\
L_{\text{ReOut}} &= O\left(\frac{f}{\sqrt{P \cdot c^3}} + \log P\right)
\end{aligned} \tag{5.15}
$$

We add $\log P$ to the latency because the output recovery may include the broadcast operation of the blocks and the reduce operation of the results.

Total costs. Summing up Equations 5.13, 5.14, and 5.15:

$$
\begin{aligned}
F' &= F_{\text{MM}} + F_{\text{ReIn}} + F_{\text{ReOut}} \\
&= F + \frac{n^3}{P \cdot \sqrt{M}} + \frac{f \cdot n^3}{P^2} \\
&= F \cdot \left(1 + O\left(\frac{f}{P}\right)\right) \\
BW' &= BW_{\text{MM}} + BW_{\text{ReIn}} + BW_{\text{ReOut}} \\
&= O(BW) \\
L' &= L_{\text{MM}} + L_{\text{ReIn}} + L_{\text{ReOut}} \\
&= O(L \cdot \log c + \log P)
\end{aligned}
$$

6 Discussion

In this paper we presented two methods for obtaining fault tolerance at lower costs: the slice-coded algorithm and the posterior-recovery algorithm. Both can handle multiple simultaneous faults. When the memory is minimal both algorithms use as few processors as possible; namely h, where h is the maximum number of faults that may occur in one iteration. We showed how to combine these methods with a 2.5D algorithm that utilizes redundant memory, to reduce the communication costs. When the number of fault is not too large our algorithms only marginally increase the number of arithmetic operations and the bandwidth costs. The slice-coded algorithm increases the latency by a factor of $\log P$. If faults occur in every iteration of the posterior recovery algorithm, its latency increases by a factor of $\log P$ as well.

The slice-coded algorithm uses the outer-product in each iteration and keeps the code processors updated. The outer product uses up to a constant factor more words, and up to $O(\log P)$ factor more messages. Therefore, the slice-coded algorithm communicates a little more, but it can recover faults quickly at each iteration. In contrast, the posterior recovery communicates less in this phase, but performs more operations to recover faults. Therefore the slice-coded algorithm is more efficient when many faults occur, and useful when quick recovery is needed. For fewer faults, the posterior recovery is more efficient.

The posterior recovery with redundant memory uses the input replication of the 2.5D algorithm. It utilize the redundant memory to reduce communication

costs and to reduce the number of required additional processors. We analyzed the case of $h < c$, where the maximum number of simultaneous faults is smaller than the number of copies of the input. In this case the algorithm does not need to allocate additional processors but rather recovers the input using the existing replication. We do not analyze here the case of $h \geq c$, where $h - c + 1$ additional processors are required, and the recovery run-time depends on the faults distribution. Briefly, in this case, if a code processor faults, the recovery requires computations, whereas when an original processor faults, the recovery uses the input replication, and is very fast.

For Strassen's [26] and other fast matrix multiplication, Ballard et al. [5] described a communication optimal parallelization that matches the communication costs lower bound [6]. However, this parallelization technique does not allow for a direct application of either methods introduced in this paper. Recently, we designed a new method that enables fault tolerancy fast matrix multiplication algorithms at low overhead [7].

7 Acknowledgments

This work was supported by grants 1878/14, and 1901/14 from the Israel Science Foundation (founded by the Israel Academy of Sciences and Humanities) and grant 3-10891 from the Ministry of Science and Technology, Israel. It was also supported by the Einstein Foundation and the Minerva Foundation, a grant from the United States-Israel Binational Science Foundation (BSF), Jerusalem, Israel, and the HUJI Cyber Security Research Center in conjunction with the Israel National Cyber Bureau of the Prime Minister's Office.

References

[1] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar. A three-dimensional approach to parallel matrix multiplication. *IBM J. Research and Development*, 39(5), 1995.

[2] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1), 2004.

[3] G. Ballard, J. Demmel, A. Gearhart, B. Lipshitz, Y. Oltchik, O. Schwartz, and S. Toledo. Network topologies and inevitable contention. In *Communication Optimizations in HPC (COMHPC), International Workshop on*. IEEE, 2016.

[4] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz. Brief announcement: strong scaling of matrix multiplication algorithms and memory-independent communication lower bounds. In *Proc. of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 2012.

[5] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz. Communication-optimal parallel algorithm for Strassen's matrix multiplication. In *Proc. of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 2012.

[6] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Graph expansion and communication costs of fast matrix multiplication. *J. the ACM (JACM)*, 59(6), 2012.

[7] N. Birnbaum and O. Schwartz. Fault tolerance with high performance for matrix multiplication. Talk presented at the meeting of the Platform for Advanced Scientific Computing Conf. (PASC17), Lugano, Switzerland, 2017.

[8] N. Birnbaum and O. Schwartz. Fault tolerant resource efficient matrix multiplication. http://www.cs.huji.ac.il/∼odedsc/papers/FTMM-full.pdf, 2018.

[9] A. Bouteiller, T. Herault, G. Bosilca, P. Du, and J. Dongarra. Algorithm-based fault tolerance for dense matrix factorizations, multiple failures and accuracy. *ACM Trans on Parallel Computing*, 1(2), 2015.

[10] L. E. Cannon. A cellular computer to implement the kalman filter algorithm. Technical report, DTIC Document, 1969.

[11] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir. Toward exascale resilience: 2014 update. *Supercomputing frontiers and innovations*, 1(1), 2014.

[12] Z. Chen and J. Dongarra. Algorithm-based checkpoint-free fault tolerance for parallel matrix computations on volatile resources. In *Proc. 20th IEEE International Par. & Dist. Processing Symposium*. IEEE, 2006.

[13] Z. Chen and J. Dongarra. Algorithm-based fault tolerance for fail-stop failures. *IEEE Trans on Par. & Dist. Systems*, 19(12), 2008.

[14] J. Demmel, D. Eliahu, A. Fox, S. Kamil, B. Lipshitz, O. Schwartz, and O. Spillinger. Communication-optimal parallel recursive rectangular matrix multiplication. In *Par. & Dist. Processing (IPDPS), 2013 IEEE 27th International Symposium on*. IEEE, 2013.

[15] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving problems on concurrent processors. Vol. 1: General techniques and regular problems*. Prentice-Hall, Inc., 1988.

[16] J. A. Gunnels, D. S. Katz, E. S. Quintana-Orti, and R. Van de Gejin. Fault-tolerant high-performance matrix multiplication: Theory and practice. In *Dependable Systems and Networks, 2001. DSN 2001. International Conf. on*. IEEE, 2001.

[17] D. Hakkarinen, P. Wu, and Z. Chen. Fail-stop failure algorithm-based fault tolerance for cholesky decomposition. *IEEE Trans on Par. & Dist. Systems*, 26(5), 2015.

[18] K. H. Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE transactions on computers*, 100(6), 1984.

[19] D. Irony, S. Toledo, and A. Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *J. Par. & Dist. Computing*, 64(9), 2004.

[20] W. F. McColl and A. Tiskin. Memory-efficient matrix multiplication in the BSP model. *Algorithmica*, 24(3), 1999.

[21] M. Moldaschl, K. E. Prikopa, and W. N. Gansterer. Fault tolerant communication-optimal 2.5D matrix multiplication. *J. Par. & Dist. Computing*, 104, 2017.

[22] J. S. Plank, K. Li, and M. A. Puening. Diskless checkpointing. *IEEE Trans on Par. & Dist. Systems*, 9(10), 1998.

[23] P. Sanders and J. F. Sibeyn. A bandwidth latency tradeoff for broadcast and reduction. *Information Processing Letters*, 86(1), 2003.

[24] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, et al. Addressing failures in exascale computing. *The International J. High Performance Computing Applications*, 28(2), 2014.

[25] E. Solomonik and J. Demmel. Communication-optimal parallel 2.5D matrix multiplication and lu factorization algorithms. In *European Conf. on Parallel Processing*. Springer, 2011.

[26] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4), 1969.

[27] R. A. Van De Geijn and J. Watts. Summa: Scalable universal matrix multiplication algorithm. *Concurrency-Practice and Experience*, 9(4), 1997.

[28] C. F. Van Loan. Matrix computations (Johns Hopkins studies in mathematical sciences), 1996.

[29] P. Wu, C. Ding, L. Chen, F. Gao, T. Davies, C. Karlsson, and Z. Chen. Fault tolerant matrix-matrix multiplication: correcting soft errors on-line. In *Proc. of the second workshop on Scalable algorithms for large-scale systems*. ACM, 2011.

[30] C. Q. Yang and B. P. Miller. Critical path analysis for the execution of parallel and distributed programs. In *Distributed Computing Systems, 1988., 8th International Conf. on*. IEEE, 1988.