Karlsruhe Institute of Technology

Master Thesis

Lightweight Checkpointing and Recovery for Iterative Converging Algorithms

Guido Zeitzmann

Date of Submission: 20 September 2023

First Reviewer:	Prof. Dr. Peter Sanders
Second Reviewer:	Prof. Dr. Thomas Bläsius
Advisor:	M. Sc. Lukas Hübner
	M. Sc. Demian Hespe

Institute of Theoretical Informatics, Algorithmics
Department of Informatics
Karlsruhe Institute of Technology

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Karlsruhe, den 20. September 2023 _____

Contents

Abstract

Fault tolerance is important for High Performance Computing, since the mean time between failures (hardware failures) decreases as the number of compute nodes increases. A system with 100.000 nodes each with a mean time between failures of 10 years, would have an expected failure rate of once every 50 minutes. We test various different methods to increase the fault tolerance of PageRank. We show that checkpointless methods can handle failures in the first 40% and finish in the same amount of iteration as a failure free PageRank execution. By estimating the progress of PageRank based on the convergence we manage to write checkpoints after 40% completion, relying on checkpointless methods beforehand. This allows us to reduce the checkpoint overhead by 40%. We also use different compression methods to compress the checkpoints. We try both lossless and lossy compression methods and show that checkpoints compressed with ZFP (lossy) cause around 40% less overhead than uncompressed checkpoints. We can reduce the checkpoint overhead of ZFP by another 16% by writing differential checkpoints, one ZFP checkpoint of the PageRank values followed by 3 strongly compressed SZ3 (lossy) checkpoints of the difference between the current PageRank values and the PageRank values at the time of the last ZFP checkpoint. In most of our test our adaptive approach, with a 1$\sim$2% checkpoint overhead, manages to handle failures without increasing the iteration time of PageRank.

Kurzfassung

Fehlertoleranz ist wichtig in großen verteilten Systemen, da der Mittelwert für die Zeit zwischen Fehlern (Hardware versagen) abnimmt, je mehr Rechenknoten das System hat. Die erwartete Fehlerrate für ein System mit 100.000 Knoten und einem Mittelwert für die Zeit zwischen Fehlern von 10 Jahren pro Knoten beträgt 50 min. Wir testen verschiedene Methoden um die Fehlertoleranz von PageRank zu erhöhen. Wir zeigen, dass Methoden ohne Checkpoints zu Beginn des Algorithmus in den ersten 40% mit Fehlern umgehen können, ohne die Iterationszeit von PageRank zu erhöhen. Außerdem testen wir verschiedene Kompressionsverfahren um das Erstellen von Checkpoints zu beschleunigen. Dabei testen wir sowohl verlustfreie als auch verlustbehaftete Kompressionsmethoden. Wir zeigen, dass das Komprimieren von Checkpoints mit ZFP (verlustbehaftete) die Kosten zum Erstellen von Checkpoints im Vergleich zu nicht komprimierten Checkpoints um 40% reduziert. Wir können den Fortschritt von PageRank an der Konvergenz messen, was uns erlaubt, checkpointfreie Methoden in den ersten 40% zu benutzen und erst dann zu checkpointen. Dies reduziert die Kosten für die Fehlertoleranz um ca. 40%. Wir reduzieren die Checkpointkosten von ZFP weiter, indem wir differenzielle Checkpoints schreiben. Wir schreiben einen ZFP Checkpoint, gefolgt von drei stark komprimierten SZ3 (verlustbehaftete Kompression) Checkpoints, die die Differenz zum letzten ZFP Checkpoint speichern. Dies senkt die Kosten des checkpointens um weitere 12%. Unsere adaptive Methode benötigte in den meisten von unseren Experimenten keine extra Iterationen wenn ein Fehler aufgetreten ist und erhöht die Gesamtlaufzeit nur um 1∼2%.

1. Introduction

After processor clock speed stopped increasing in 2004, after years of exponential growth, new ways to fulfill, the rising performance demand needed to be found. A viable path to sustained petascale and even exascale performance is massive parallelism [14]. Besides design challenges, massive parallelism also brings with it new challenges regarding resilience. Since more compute nodes also mean more potential for hardware failures. For example a machine with 100.000 nodes, each with a mean time between failures (MTBF) of 10 years, would have an expected failure rate of once every 50 minutes [20]. There are different types of failures that can occur besides fail-stop-errors (hardware failures) like silent data corruption but for our purposes we focus on fail-stop errors. A general approach to handle fail stop failures is creating a snapshot of the distributed system by writing process checkpoints (saving the state of each process). The protocol by Chandy and Lamport [9] is a way to take such a consistent snapshot. A consistent snapshot means that if we restart the application from the snapshot that the application is in a state consistent with its normal behaviour. The problem with these snapshots is that their space requirement is directly proportional to the memory footprint of the processes. Depending on the application we have to save only a portion of the data, thereby reducing the checkpoint overhead. We look for ways to increase the fault tolerance (ability to handle failures) for iterative converging algorithms, specifically Googles PageRank algorithm [6]. We look at ways to reduce the checkpoint overhead by writing compressed checkpoints. We use both lossy and lossless compression methods to reduce checkpoint size. A compression method is lossy if the data after decompression can deviate from the original data. We also test the fault tolerance of various approaches that do not require checkpoints. We compare the recovery of the various approaches. For us recovery entails the amount of redundant work caused by a failure as well as time it takes until we can resume calculation.

Structure

We start by introducing some common definitions used in this thesis, followed by the related works we found. In Chapter 2 we take a look at how the PageRank algorithm works and its different properties. We also look at what adjustment we make to PageRank to turn it into a distributed algorithm. Then in Chapter 3 we describe the different methods we test to make PageRank resilient in case of fail-stop errors, as well as describing the function of the different compression methods we use. Our experimental evaluation of the various approaches is shown in Chapter 4. Then in Chapter 5 we give our conclusions and final remarks.

1.1 Definitions

We use the terms fault, failure and fail-stop error interchangeably. To avoid confusion between graphs and compute nodes, we refer to graph nodes as vertices. Some commonly used symbols are seen in Table 1.1

V	vertex set
E	edge set
$PR(v)$	PageRank of vertex v
$PR^i(v)$	PageRank of vertex v at iteration i
V^p	vertex set of process p
μ	Mean time between failures
P	set of processes

Table 1.1: Definitions

1.1.1 Graph

A directed graph is a pair $G = (V, E)$ made up of a set of vertices V and a set of edges that are ordered pairs of vertices $E \subseteq \{(x, y)|x, y \in V\}$. We call (x, y) an edge from x to y. We say a graph is undirected when for every edge $(x, y) \in E$ with $x \neq y \implies (y, x) \in E$. We call edges originating from x (e.g (x, y)) out-edges of x and edges ending in x in-edges of x.

1.2 Related Work

Checkpoint Restart is a method that deals with fail-stop errors. Fail-stop errors are errors that cause the component experiencing it to stop operating. Different checkpointing approaches have been researched over the years. The book by Herault and Robert [19] gives an overview of different general checkpointing approaches. They differentiate between coordinated checkpoints where all the processes checkpoint together to create a consistent snapshot and uncoordinated checkpoints where each process can make its own checkpoint. The goal of uncoordinated checkpoints is to reload only the failed process. To enable this all messages need to be logged until their receiver checkpoints, so they can be "resend" in case of failure. The combination of both coordinated and uncoordinated checkpoints are hierarchical checkpoints where the processes are divided into groups, processes of a group checkpoint coordinated and groups use uncoordinated checkpoints between each other.
In the 1970s Young [42] proposed a first order approximation for the optimal time between two checkpoints, to minimize the overall expected computation time. This approach was later refined by Daly [11] resulting in a formula for the optimal checkpoint frequency based on the mean time between failures μ and a constant checkpointing cost. The optimal checkpoint frequency according to the Young/Daly formula is $f = \sqrt{2\mu C}$ where C is the time it takes to create a Checkpoint. For iterative methods were the checkpoint size (cost) may vary or checkpoints are taken only at specific times in the execution, e.g after completion of a task or an iteration Du et al. [15] propose a dynamic programming algorithm to calculate the optimal checkpoint pattern, under the assumption that each iteration consist of n tasks of different run times and the cost of a checkpoint differs depending on after which task it is taken. A good way to increase the performance of checkpointing is to decrease the checkpoint size. For iterative graph processing, Yan et al. [41] introduce an application specific checkpointing system, that reduces the amount of information that needs to be checkpointed. For example checkpointing only the ranks of vertices for PageRank but not checkpointing incoming messages or edges. Since the edges can be reloaded from the input and the messages sent can be recomputed.

An approach to further reduce the size of a checkpoint is to compress the data that gets checkpointed. Islam et al. [25] use data aggregation and lossless compression to improve the performance of the checkpoint and recover routines. There has also been research for using error bounded lossy compression to further reduce the size of checkpoints, since lossy compression can achieve a much higher compression rate than lossless compression. Calhoun et al. [8] use lossy compressed checkpoints for partial differential equation (PDE) simulations. They show that an upper bound on the error of lossy compression can be found by masking the error of compression with the numerical error of discretization. They show that if the solution is accurate to only 1e-4 then a solution x is indistinguishable from a perturbed solution $\tilde{x} = x + \epsilon$ when $\epsilon <$ 1e-4. Their results boast an overall increase in performance without loss of accuracy for two specific PDE simulations. Sasaki et al. [36] compress the checkpoints of climate applications using wavelet transformation and calculate that they can reduce the checkpoint time including compression by 83% compared to uncompressed checkpoints under the assumptions that each process possesses 1.5MB of data that it needs to checkpoint and the I/O throughput of the parallel file system is 20GB/s. They show that their strongly compressed checkpoints scale better, leading to a reduced checkpoint time of 83% for a sufficiently large system. With each variable deviating from their original values by approximately 1.2%. Tao et al. [38] build a generic, theoretical performance model that considers the impact of lossy compression for iterative methods and used lossy checkpoints to improve the overall performance for multiple iterative methods (Jacobi, CG, GMRES). They compare the performance of checkpoints with lossy compression to uncompressed checkpoints and checkpoints with lossless compression. By comparing how much time it takes to create and recover one checkpoint, the increase in iterations compared to a failure free execution and the overhead of the different checkpointing methods (increase in execution time when no failure occurs). Qiao et al. [33] proposed a checkpoint method for iterative converging machine learning. They make use of the self correcting nature of iterative converging ML and restore only partial checkpoints (only the failed parameters) to reduce the rework cost (increase in number iterations, batches or epochs until convergence). To further reduce the rework cost they use a different checkpoint pattern. Instead of checkpointing all parameters every N iterations, they checkpoint $\frac{1}{x}$ of the parameters every $\frac{N}{x}$ iterations (checkpointing the same amount of total data in N iterations). The parameters that changed the most since the last time they were checkpointed, are chosen for the next checkpoint. We compare the performance of multiple approaches for PageRank like partial recovery and full recovery from lossless and lossy checkpoints as well as testing different checkpointless approaches for recovery. We also see if we can reduce the checkpoint overhead by using less precise checkpoints or checkpointless methods at the start and adjusting the precision of the methods based on the convergence of the PageRank algorithm. This adaptive checkpointing scheme that switches between checkpointless methods and writing checkpoints based on the convergence of the algorithm is to our knowledge a novel idea.

2. PageRank

The PageRank algorithm [31, 6] is used by Google search to order the results based on importance. These days, PageRank is not the only algorithm Google uses to order the results, but it is the first and most well known algorithm for ordering webpages. The idea behind PageRank is that the importance of a webpage depends on the webpages that link to it. But instead of counting the number of links to a webpage, PageRank takes the importance of the webpage the link originates from into account. One way to think about the PageRank algorithm would be as a model for user behavior. Were a random surfer is given a random web page and continues surfing from there by clicking random links (without hitting back) until they get bored and restarts from another random page.

2.1 Algorithm

Given a Web-Graph $G(V, E)$, with vertices V the webpages and edges E the links between webpages. With $(u, v) \in E$ when webpage u links to webpage v. Let $PR : V \to (0, 1]$ be the PageRank of a page and $L : V \to \mathbb{N}$ the number of outgoing links of a webpage. Let us take a look at the PageRank algorithm 1. Initially (line 3) every page gets the same amount of PageRank. Here PR^k is the PageRank at iteration k. The PageRank of all vertices gets updated in every iteration as seen in lines 7 and 8. We explain the equation in line 8 with the random surfer model. The damping factor $d \in (0, 1)$[1] is the probability that the surfer does not get bored and clicks another link. The first part handles the teleport probability $1 - d$, the probability that the random surfer gets bored and is requesting a new random page (evenly distributed), causing the surfer to "teleport" to a new random webpage. Since there are $|V|$ many pages there is a $\frac{1-d}{|V|}$ probability that he arrives at a specific page. The second part is the probability that the random surfer arrives at the page through a link of another page. $L(u) = \sum_{(u,v)\in E} 1$ are the amount of outgoing links from page u. The PageRank $PR(u)$ is the probability that the random surfer is on page u. Because the surfer clicks random links the probability that he arrives at a page v through u depends on the amount of outgoing links of vertex u and the probability d that he clicks another link, thus the probability is $d \cdot \frac{PR(u)}{L(u)}$. When we add all the parts up we get the PageRank of the current iteration. After calculating the new PageRank values we check for convergence. We take a look at different convergence criteria in section 2.2. It is important to note that when each vertex has at least one outgoing edge the sum of all PageRank is always one (Equation (2.1)). For vertices with no outgoing edges (dangling vertices) the random surfer has no choice but to request a new random web page. This is simulated by assuming that dangling vertices have edges to all vertices otherwise we would loose PageRank [7].

[1] d is set to 0.85 in the original paper [6]

Algorithm 1 PageRank

```
1: procedure PageRank(Graph G, damping factor d)
2:     for v ∈ V do
3:         PR^0(v) ← 1.0/|V|                          ▷ Set initial PageRanks
4:     end for
5:     k ← 1
6:     while not converged do                          ▷ Begin Iterations
7:         for v ∈ V do
8:             PR^k(v) ← (1−d)/|V| + d · Σ_{(u,v)∈E} PR^{k−1}(u)/L(u)   ▷ Update PageRanks
9:         end for
10:        k ← k + 1
11:        check if converged                          ▷ Different convergence criteria
12:    end while
13: end procedure
```

$$
\begin{aligned}
&\text{Show: } \sum_{v_i \in V} PR^k(v_i) = 1, \text{ with } \forall v \in V, L(v) > 0 \\
&\sum_{v_i \in V} PR^0(v_i) = 1, \text{ since } PR^0(v) = 1/|V| \text{ for all } v \in V \\
&\sum_{v_i \in V} PR^n(v_i) = \sum_{v_i \in V} \frac{1-d}{|V|} + d \cdot \sum_{(u,v_i) \in E} \frac{PR^{n-1}(u)}{L(u)} \\
&= 1 - d + d \cdot \sum_{v_i \in V} \sum_{(u,v_i) \in E} \frac{PR^{n-1}(u)}{L(u)} \\
&= 1 - d + d \cdot \sum_{u \in V} PR^{n-1}(u), \text{ since } \sum_{v_i \in V} \sum_{(u,v_i) \in E} 1 = \sum_{u \in V} L(u) \\
&= 1 - d + d \cdot 1 = 1
\end{aligned}
\tag{2.1}
$$

2.2 Convergence Criteria

There are different convergence criteria. One way is to simply stop after x iterations. Another criterion is to check for every vertex if the difference between last iterations PageRanks and this iterations PageRanks falls under a tolerance threshold t ($\forall v \in V, |PR^k(v) - PR^{k-1}|(v) < t$). But if the graph has more vertices, the threshold would also need to be adjusted since the PageRank values are now smaller. We do not need to adjust the tolerance threshold if we sum up all the absolute differences between the PageRank (L1 Norm) as seen in equation (EQ: Sum).

$$\sum_{v_i \in V} (|PR^k(v_i) - PR^{k-1}(v_i)|) < t \tag{EQ: Sum}$$

Another similar way would be to sum up the squares of the differences between the old and new PageRank values, see equation (EQ: SquareSum). Which is basically the L2 Norm except we do not calculate the square root (can be masked in the tolerance t).

$$\sum_{v_i \in V} (PR^k(v_i) - PR^{k-1}(v_i))^2 < t \tag{EQ: SquareSum}$$

Finally we have a more strict convergence criterion that can be thought of like an extreme case of the sum criterion where all differences are equal to the maximum difference between a pair of PageRank values. We call this criterion MaxChange see equation (EQ: MaxChange). This criterion tolerates a lot less distortion of the PageRank values since it amplifies the maximum distortion by the number of vertices (assuming we use the same tolerance t).

$$|V| * \max_{v_i \in V}(|PR^k(v_i) - PR^{k-1}(v_i)|) < t \qquad \text{(EQ: MaxChange)}$$

2.3 Power Iteration

To show that PageRanks solution is unique we look at power iteration the algorithm PageRank is based on. Power iteration is an algorithm that calculates the dominant eigenvalue λ (highest in absolute value) and the corresponding eigenvector v of a diagonalizable matrix M ($Mv = \lambda v$) [29]. In PageRanks case the Power iteration method works by taking a probability distribution vector k_0 (the initial PageRanks) and improving our current approximation k_i by calculating $k_{i+1} = Mk_i$. Here k_i are the PageRanks after the i-th iteration. The iteration matrix M is the matrix that describes the update step in line 8 of the PageRank algorithm. Power Iteration converges if M has a dominant eigenvalue and the search vector k_i has a nonzero component in the direction of the dominant eigenvector[2]. The dominant eigenvalue of M is always one according to the Perron-Frobenius theorem [32]. The Perron-Frobenius theorem says that a square matrix K with only positive real number as elements (greater than 0) has a dominant eigenvalue r (Perron-Frobenius eigenvalue) that is strictly larger in absolute value than any other eigenvalue of K. The dominant eigenvalue of K satisfies the inequalities $\min_i \sum_j k_{ij} \leq r \leq \max_i \sum_j k_{ij}$ where k_{ij} is the entry at column i and row j of K. To understand that the iteration matrix M of PageRank is always a positive matrix, with the sum of each of its columns being one, we show an example of how the iteration matrix M is calculated. In Figure 2.1 we see our example graph G and its corresponding adjacency matrix $\hat{A}$.

$$\hat{A} = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

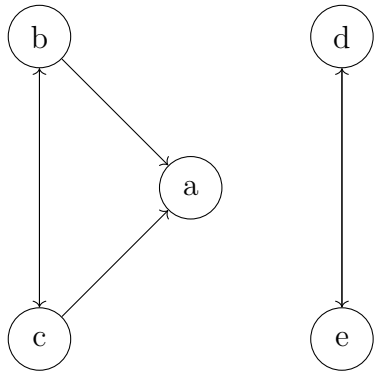


Figure 2.1: Disconnected Graph G of adjacency matrix $\hat{A}$ and a dangling vertex a

G is disconnected and has a dangling vertex a. We know that dangling vertices need to be fixed by adding edges from them to all vertices (columns with all 0 in $\hat{A}$

[2]eigenvector of the dominant eigenvalue

are transformed to all 1s) [7]. After fixing a the graph G and adjacency matrix $\hat{A}$ changes as seen in Figure 2.2.

$$\hat{A} = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{pmatrix}$$

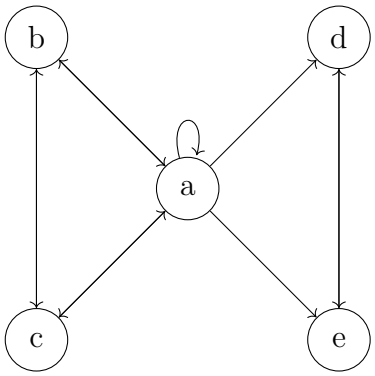


Figure 2.2: Disconnected Graph G and adjacency matrix $\hat{A}$ after "fixing" dangling vertex a

After we "fixed" the dangling vertices we normalise the columns of the adjacency matrix (sum of each column is 1) since the PageRank gets divided evenly amongst all outgoing edges of a vertex. After normalising the columns we get the matrix

$$A = \begin{pmatrix} \frac{1}{5} & \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ \frac{1}{5} & 0 & \frac{1}{2} & 0 & 0 \\ \frac{1}{5} & \frac{1}{2} & 0 & 0 & 0 \\ \frac{1}{5} & 0 & 0 & 0 & 1 \\ \frac{1}{5} & 0 & 0 & 1 & 0 \end{pmatrix}$$

We then calculate the iteration matrix M by introducing the damping factor d and teleport probability $(1-d)$ to the matrix A. Thus $M = d \cdot A + (1-d) \cdot B$ with B a matrix of the same size as A with all entries being 1. Using $d = 0.85$ the standard value for PageRank we get the following iteration matrix.

$$M = \begin{pmatrix} \frac{1}{5} & \frac{91}{200} & \frac{91}{200} & \frac{3}{100} & \frac{3}{100} \\ \frac{1}{5} & \frac{3}{100} & \frac{91}{200} & \frac{3}{100} & \frac{3}{100} \\ \frac{1}{5} & \frac{91}{200} & \frac{3}{100} & \frac{3}{100} & \frac{3}{100} \\ \frac{1}{5} & \frac{3}{100} & \frac{3}{100} & \frac{3}{100} & \frac{22}{25} \\ \frac{1}{5} & \frac{3}{100} & \frac{3}{100} & \frac{22}{25} & \frac{3}{100} \end{pmatrix}$$

Important to note is that the the teleport probability $(d-1)$ always ensures that M is a positive matrix and the fixing of dangling nodes as well as the normalising of columns ensures that the sum of each column of M is one. Therefore the dominant eigenvalue r of M is always one and r has a unique normalised eigenvector. In other words the ranking of the vertices is unique for a fixed dampening factor d. Also since power iteration converges as long as the starting vector has a nonzero component in the direction of the dominant eigenvector, PageRank converges towards the same solution for any arbitrary ranking PR with $\forall v \in V, PR(v) > 0$ and $\sum_{v \in V} PR(v) = 1$ [40].

V^{p_i}	$V^{p_i \to p_0}$	$V^{p_i \to p_1}$	...	$V^{p_i \to p_n}$

Figure 2.3: Structure of newScore, ghost vertices are grouped by process they belong to

2.4 Implementation

To make the PageRank algorithm parallel we divide the vertices amongst the different processes. So that every process p has its own set of vertices V^p with no overlaps between processes (meaning if $p, q \in P$, then $V^p \backslash V^q \neq \emptyset \implies p = q$). Each process also possesses the outgoing edges of its vertices $E^p = \{(u, v) \in E | u \in V^p, v \in V\}$. The main challenges are the outgoing edges that lead to a vertex of a different process. For performance reasons we do not want to sent the messages immediately, but instead bundle the communication data and send it all at once. To bundle the communication data each process has "ghost" vertices $V^{*p} = \{v | (u, v) \in E^p, v \notin V^p\}$, the vertices of other processes whose PageRank values the vertices of process p contribute to. This leads us to the parallel PageRank algorithm seen in algorithm 2. Here we have two arrays *score* with $|V^p|$ entries, the local PageRanks of the last iteration and *newScore* with $|V^p + V^{*p}|$ entries containing the PageRanks of the current iteration and the PageRank updates for the ghost vertices V^{*p}. At the beginning we distribute the PageRank evenly amongst all local vertices (*score*). We set *newScore* to the teleport probability for local vertices and to zero for ghost vertices. In lines 6 to 8 we divide the current PageRank of each local vertex amongst its neighbouring vertices. To allow indexing with the vertices indices, the local indices for vertices are adjusted. In line 11 we handle the communication, sending the PageRank values of the ghost vertices to the corresponding process and receiving the PageRanks values for local vertices. Let $V^{p_i \to p_j} = \{v \in V^{p_j} \backslash V^{p_i} | (u, v) \in E^{p_i}\}$ be the vertices of process p_j that process p_i sends PageRank to, then *newScore* is structured as seen in Figure 2.3. The structure of newScore allows us to handle the communication of PageRank updates in one MPI_Alltoallv call. After distributing the PageRank updates we handle the dangling vertices by summing up their values and distributing them evenly amongst all vertices. After the communication phase *newScore* contains the updated PageRank values of the current iteration, so we can now check for convergence. Afterwards in lines 13 to 15 the new PageRank values are moved into *score* and *newScore* is reset in preparation for the next iteration. Before beginning the next iteration we check if we should checkpoint the current PageRank values.

Algorithm 2 PageRank Parallel

1: **procedure** PageRank(Graph G, damping factor d)
2: $\forall\ v \in V^p$, $score\,[v] \leftarrow \frac{1}{|V|}$
3: $\forall\ v \in V^p$, $newScore\,[v] \leftarrow \frac{1-d}{|V|}$ ▷ account for teleport probability
4: $\forall\ v \in V^{*p}$, $newScore\,[v] \leftarrow 0$ ▷ Ghost vertices
5: **while** not converged **do**
6: **for** $v \in V^p$ **do**
7: **for** $(v, u) \in E^p$ **do**
8: $newScore[u] \leftarrow newScore[u] + d \cdot \frac{score[v]}{L(v)}$
9: **end for**
10: **end for**
11: SendAndReceiveMessages
12: check convergence
13: $\forall\ v \in V^p$, $score\,[v] \leftarrow newScore\,[v]$.
14: $\forall\ v \in V^p$, $newScore\,[v] \leftarrow \frac{1-d}{|V|}$.
15: $\forall\ v \in V^{*p}$, $newScore\,[v] \leftarrow 0$.
16: MakeCheckpoint?
17: **end while**
18: **end procedure**

3. Approaches

In this Chapter we take a look at various approaches we apply to make PageRank fault tolerant for fail-stop faults. When a process p fails, we loose the PageRank for the set of vertices of process p namely V^p. We assume that we are able to reload the lost graph data of p from the input. Therefore our goal is to set/restore the lost PageRank values $PR(v_i) = ?$ for $v_i \in V^p$ in a way that minimizes the amount of extra work done compared to a faultless execution. But we do not loose all information about the PageRanks of V^p, since the sum of all PageRanks is always 1. Thus we can calculate the sum of the lost PageRanks (eq. 3.1). Furthermore every vertex has a minimum PageRank based on the damping factor d and the total number of vertices ($\min(PR(v_i)) = \frac{1-d}{|V|}$) In the random surfer model this minimum value is the chance that the surfer lands on the web page by requesting a new random web page. Therefore we know that all lost PageRanks values must lie in the interval $[\frac{1-d}{|V|}, \frac{1-d}{|V|} + (missing\_sum(p) - |V^p| \cdot \frac{1-d}{|V|}]$. We use this fact in the various approaches we describe in this Chapter. First we take a look at various checkpointless approaches where the sum of the missing PageRanks is all the information available. Later we look at approaches where checkpoints are available and see how we can utilise the information of older PageRank values.

$$missing\_sum(p) = \sum_{v_i \in V^p} PR(v_i) = 1 - \sum_{v_i \in V/V^p} PR(v_i) \tag{3.1}$$

3.1 Checkpointless Approaches

When a process fails for example due to a hardware failure we loose the PageRank values of the process. If we have no checkpoints, we only know the total amount of PageRank the process had. We do not want to restart the algorithm, since we lost only a fraction of our results. But we do not know the distribution of the PageRank of the lost vertices. We have to make an assumption about the distribution. We take a look at three different checkpointless approaches that use different methods to distribute the PageRank.

3.1.1 Divide Even

Divide Even is the naive method. The method works by taking the sum of lost PageRanks and dividing it evenly among the lost vertices V^p as seen in equation (EQ: Divide Even). It resets the PageRank of lost vertices to their starting values adjusted based on the amount of missing PageRank. This approach fails to capture any nuance of the actual distribution of the PageRank among the vertices. For example if we lose a couple of vertices with a large amount of PageRank and the other lost vertices have a small amount of PageRank after restoring they would all have the same PageRank.

$$PR(v_i) = \frac{missing\_sum(p)}{|V^p|} \tag{EQ: Divide Even}$$

3.1.2 Based On In Edges

With this approach we leverage the information of the graphs structure to make an assumption about distribution of the PageRank amongst lost vertices. We assume that vertices with more incoming edges are more likely to have a higher PageRank than vertices with fewer incoming edges, since more sources contribute to their PageRank. We then assign each restored vertex the minimum PageRank value, divide the remaining sum based on the number of in edges of a vertex v_i in proportion to the number of total incoming edges of V^p as seen in equation (EQ: boie).

$$\begin{aligned} m &= missing\_sum(p) - |V^p|\frac{1-d}{|V|} \\ PR(v_i) &= \frac{1-d}{|V|} + m \cdot \frac{|\{e' \mid (u, v_i) \in E\}|}{|\{e \mid (u, v) \in E, v \in V^p\}|} \end{aligned} \tag{EQ: boie}$$

3.1.3 Local Iteration

The last checkpointless approach we look at is to set the PageRanks using one of the other two approaches (Divide Even or Based On In Edges) mentioned above and then running x local iterations of the PageRank algorithm. A local iteration here is defined as cutting out the communication that would usually occur when a vertex receives an amount of PageRank from a vertex $v \notin V^p$ or sends an amount of its own PageRank outside of V^p. We need to account for these outgoing edges of V^p to make sure that the sum of all PageRanks is still one after the local iterations. We try two different methods to handle outgoing edges. The first one is to ignore them by reducing $L(v_i)$ by the amount of outgoing edges of v_i. Another method is to reflect the PageRank on outgoing edges back to the vertex that would send it. This idea is mainly useful for undirected graphs where the vertex would receive some PageRank back and it also ensures that a vertex $v_i \in V^p$ with a lot of edges that point out of V^p and few internal edges does not increase the internal PageRanks too much. The main idea for local iterations is that most of the work of the algorithm is communication between processes so by ignoring communication we might restore some of the finer details of the PageRank distribution without incurring the cost of communication. The probability that the surfer teleport to a local page needs to be adjusted based on the amount of PageRank the process has ($\frac{1-d}{|V^p|} \cdot \sum_{v \in V^p} PR(v)$) otherwise we gain PageRank.

3.2 Checkpoints

The main problem with checkpointless approaches is that towards the end of the calculation we need to restore more precise PageRank values (as we see in section 4.1). To solve this problem we can write checkpoints when the performance of checkpointless approaches deteriorate. When we write checkpoints we have a lot more information to fall back on namely the PageRanks of previous iterations or an approximation of the PageRanks if we used lossy compression to create the checkpoints. In the following we examine the different ways we use this information as well as describing different compression methods for writing checkpoints.

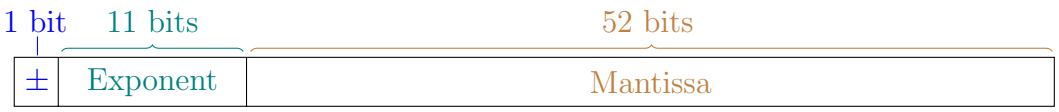


Figure 3.1: IEEE 754 double precision representation.

3.2.1 LoadAll and LoadFailed

For the following section we will assume we have a checkpoint of old PageRanks $PR^{checkpoint}(v)$. The naive approach of using this checkpoint is to set all PageRanks to the checkpointed values and then continue calculation from there (eq. LoadAll).

$$PR^{current}(v) := PR^{checkpoint}(v) \tag{LoadAll}$$

We call this approach LoadAll. If the PageRank values where lossy compressed it can lead to an increase or decrease in the total PageRank, which we repair by calculating the missing/excess PageRank and then distribute it equally amongst all vertices. When the checkpoints are lossless compressed we know that we have to redo the iterations computed since the last checkpoint. But for the processes that did not fail we already have the results after calculating all those extra iterations. That means when we load the checkpoint for all processes, we also discard all that extra information. Which brings us to the LoadFailed approach, here we set only the PageRanks of the failed process all other processes get to keep their ranks. We then scale the loaded PageRank values relative to the missing sum of the current iteration as seen in (eq. LoadFailed).

$$PR^{current}(v) := PR^{checkpoint}(v) * \frac{missing\_sum(p)}{\sum_{u \in V^p} PR^{checkpoint}(u)} \tag{LoadFailed}$$

3.2.2 Compression Methods

We try to reduce the time it takes to create a checkpoint by compressing the data before checkpointing it, since smaller checkpoints are faster to create. In the following we explain eight different compression methods.

No Compression (Lossless)

As a baseline for comparison, we checkpoint the uncompressed PageRank values.

Exponent (Lossy)

Here we truncate the floating point values saving only the first x bits/bytes. We do not truncate the exponent and sign making x at least 12 for double precision. Figure 3.1 shows the representation of an IEEE 754 double precision number.

Bucket (Lossy)

For bucket checkpoints, we group the values in buckets of size x and then checkpoint the sum of each bucket. During decompression we divide the value of each bucket evenly amongst the vertices of the bucket. In Figure 3.2 we show an example for $x = 4$. The PageRank values are divided into buckets of size four. Then we calculate the sum for each bucket. If the number of values ($|V^p|$) is not dividable by x the last bucket has size $|V^p| \mod x$.

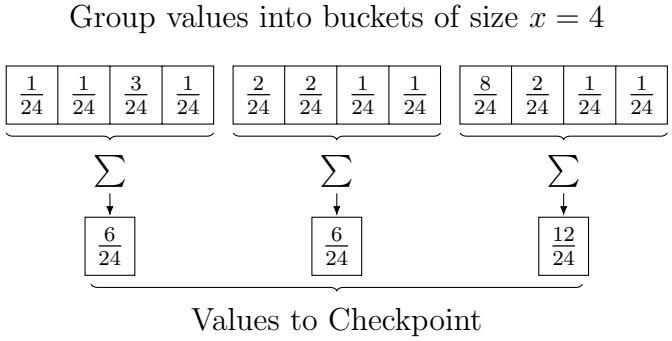


Figure 3.2: The original values are divided into groups of size $x = 4$. Each group is then summed up. The sums of the different groups are then checkpointed.

Cutoff (Lossy)

Under the assumption that the most important vertices, are vertices with a high PageRank and other vertices are less important for convergence we cut off all values below a cutoff threshold η. We can then either perform a run-length encoding by setting cutoff values to 0 or saving only non cutoff values and memorising which values we saved. When we restore the values we can set the cutoff values either to the cutoff threshold η, the minimum PageRank or random values between.

ZFP (Lossy)

ZFP is a lossy compression library for floating-point and integer arrays based on an algorithm by Peter Lindstrom [28]. The current version of the algorithm can be found in the paper by James Diffenderfer et. al. [13]. The algorithm has the following 8 steps.

1. Group the input data into 4^d blocks with d being the dimension of the input data (d = 1 in our case).
2. Transform the blocks into block floating point numbers. This means using a single common exponent for all 4^d values of a block (the highest exponent of the values). In Figure 3.3 we see an example for the transformation of four 32bit floating point numbers into block floating point numbers with a common exponent. This step is lossy because it can cause truncation of numbers with a smaller exponent than the common exponent.
3. Transform the block floating point numbers using a near-orthogonal transform, similar to the discrete cosine transform used in JPEG, to decorrelate the values of a block.
4. Skipped in our application (d = 1).
5. Convert the transformation coefficients from two complement representation into negabinary (binary numbers with base -2). The advantage of negabinary is that they do not have a dedicated sign bit instead the highest non 0 bit encodes the sign. For small coefficients this leads to many leadings zeros.

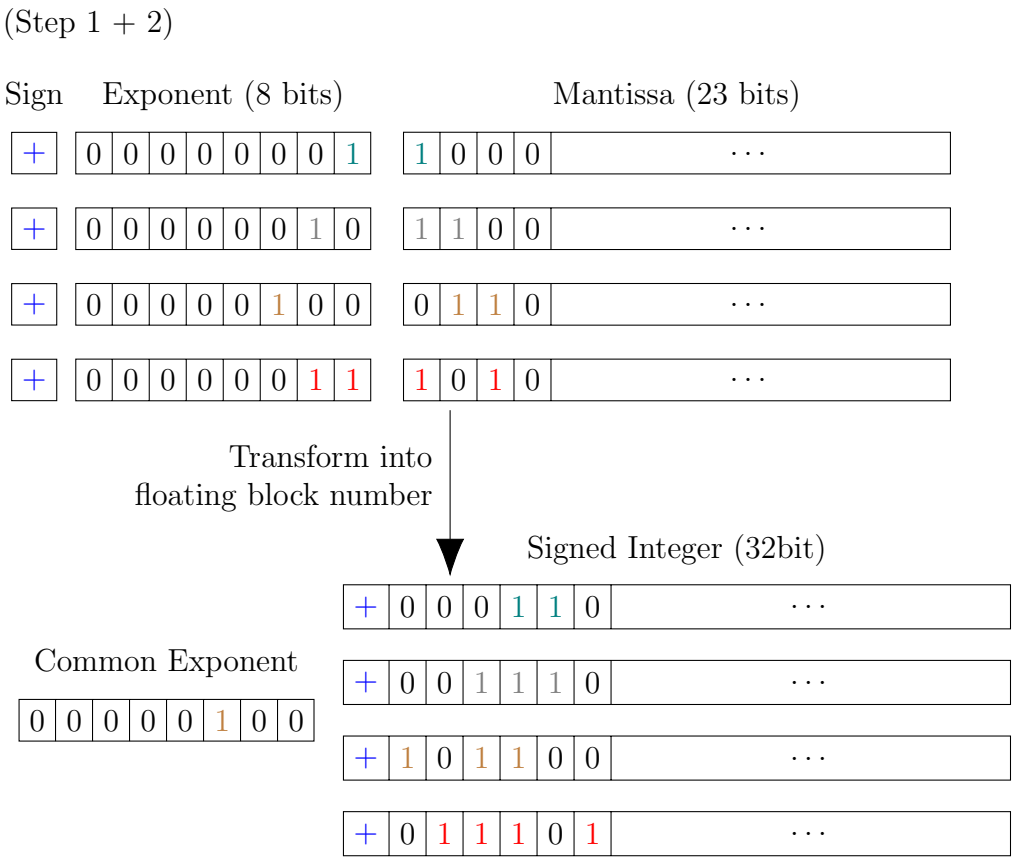


Figure 3.3: Transformation of four 32 bit floating point numbers into block floating point numbers with a common exponent

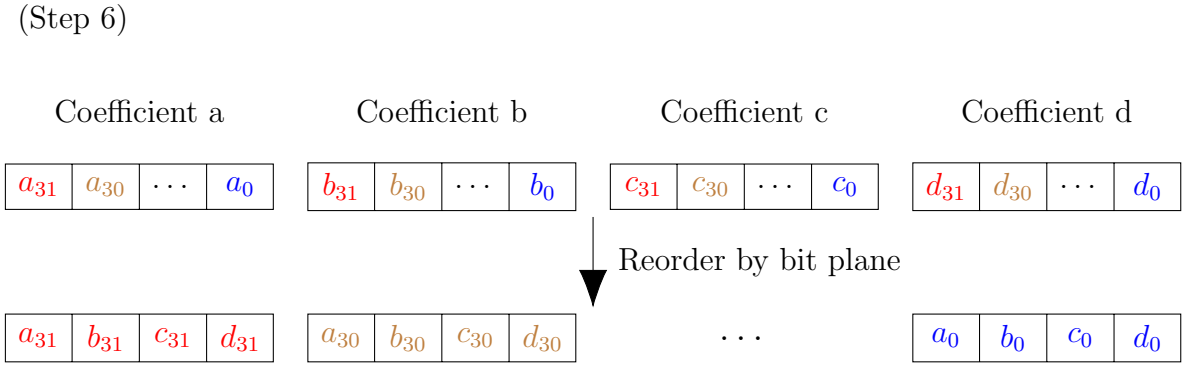


Figure 3.4: Ordering the bits of the coefficients by bit plane (from most significant bit to least significant bit)

6. Order the bits by bit plane instead of by transformation coefficient. Starting with the most significant bits. Meaning we start with the most significant bits of the coefficients followed by the next most significant bits as seen in Figure 3.4.

7. Compress the bit planes (lossless). Using the fact that the coefficients have many leading zeros that do not need to be explicitly encoded. Emit the lowest n bits unchanged. Here n is the highest set bit of all previous encoded bit planes. Then perform a variable length encoding on the remaining $4^d - n$ bits. If there are no more set bits in the bit plane the encoder emits a 0, otherwise it emits a 1 and all the bits up to the next 1 then it test if there are more ones left in the bit plane.

8. In the final step the encoder emits one bit at a time, each bit increasing the accuracy of the approximation. Truncating the emitted stream at any point still allows the reconstruction of the values of the encoded block. ZFP has 3 different modes. The modes decide when to truncate the stream. The mode we use is the Fixed-precision mode.
   Fixed-precision mode: The user sets the number of bit planes to encode. Here the amount of bits the encoder emits for each block can vary.

ZFPs fixed-precision mode is better for relative error. Allowing it to automatically adjust to expected PageRank values based on the number of vertices of the graph.

ZFP (Lossless)

ZFP has another mode called reversible mode that allows lossless compression. Most of the steps are the same steps from lossy zfp with the main differences occurring in steps 2, 3 and 8 the steps that can lead to a loss in precision in the compressed values. In step 2 the algorithm test the block floating point numbers to see if the transformation was lossless if it was lossless it continues as usual otherwise it interprets the floating point numbers as integers by type punning (reinterpret the bits as an integer) and continues from there. The transform in step 3 is also slightly modified and in step 8 the encoder truncates no set bits. The algorithm still truncates the least significant bits planes that are all 0.

SZ3 (Lossy)

SZ3 is a lossy compression library that allows error bounds [27, 12, 39]. The algorithm we use works by first calculating an absolute error bound η based on the input data by finding the middle of the data range and multiplying it with the set relative error bound. In the next step a predictor estimates the value of a data points. The predictor that is used depends on the internal parameter optimization of SZ3. SZ3 decides between using spline-interpolation prediction (linear or cubic) and Lorenzo prediction. To select the best predictor SZ3 uses a uniform sample of the data (3%) and performs trial runs with the different predictors on it. The predicted points undergo linear quantization to transform the floating point variables into integer variables and more importantly to enforce the error bound. Linear quantisation works as follows. Let x be the actual value and p_0 the predicted value the calculated integer value int_x is then $int_x = round\left(\frac{(p_0 - x)}{\eta}\right)$ in other words we have bins of equal

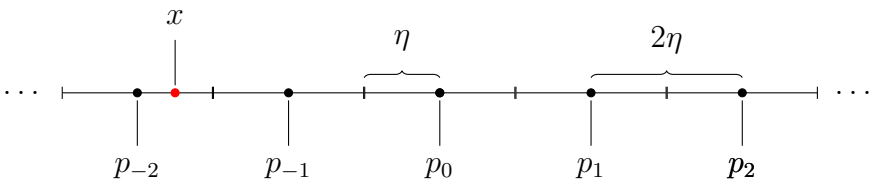


Figure 3.5: Linear Quantisation: Assign the value x to the closest prediction point p with p_0 the actual prediction. All prediction points are exactly 2η apart with absolute error bound η.

intervals of size 2η and we find which bin x belongs in (Figure 3.5). If the number falls out of the range of the bins mark it and save it uncompressed. The resulting integers are then Huffman encoded and finally dictionary encoded.
Lorenzo Predictor: Predicts the value of the current data point based on the already predicted preceding data points. In our case (1-dimension) the lorenzo predictor works as follows, let d_i be the i-th data point and $pred_L$ the Lorenzo predictor. The 1st order prediction for d_i would then be $pred_L(d_i) = d'_{i-1}$, where d'_{i-1} is the reconstructed value after linear quantisation (the decompressed value of d_i). After prediction of a value d_i, it undergoes linear quantisation and the reconstructed value d'_i is used for further predictions. The 2nd order Lorenzo predictor would be $pred_L(d_i) = 2 \cdot d'_{i-1} - d'_{i-2}$. Whether first or second order prediction is used depends on the results of the trial runs.
Spline Interpolation prediction: Spline interpolation prediction works by finding a polynomial that goes through already predicted/known data point to predict the data point that lies in the middle of the points. For linear interpolation two points are needed and the resulting spline is the line that connects them. For cubic spline interpolation SZ3 uses 4 points (Figure 3.6) and the spline is made up of three third degree polynomial functions f_1, f_2, f_3 connecting each neighbouring point. The first and second derivative needs to be the same where the polynomial functions connect (e.g $f'_1(i-1) = f'_2(i-1), \cdots$). SZ3 uses not-a-knot cubic splines which means the third derivative is also the same (when only four points are used). SZ3 does not solve the linear equation but uses a closed-form solution for both linear and cubic spline interpolation.

- Linear $pred_{Lin}(d_i) = \frac{1}{2}d_{i-1} + \frac{1}{2}d_{i+1}$
- cubic $pred_{Cube}(d_i) = -\frac{1}{16}d_{i-3} + \frac{9}{16}d_{i-1} + \frac{9}{16}d_{i+1} - \frac{1}{16}d_{i+3}$

These solutions assume that we know all odd numbered data points. The actual predictor avoids this by predicting the data through multiple steps. First the data is divided into smaller blocks (size 32). The prediction uses multiple levels on every block. We show an example of a multilevel linear prediction for block size 5 in Figure 3.7. Here L1-L5 denote the level. In the first level SZ3 predicts d_0 from 0. Between each level the predicted values undergo linear quantisation we denote the reconstructed values, after linear quantisation as d'_i. In level two the reconstructed data point d'_0 is used to predict d_5. This continues until level five where all data points of the block have been predicted. It works similar for cubic spline interpolation.

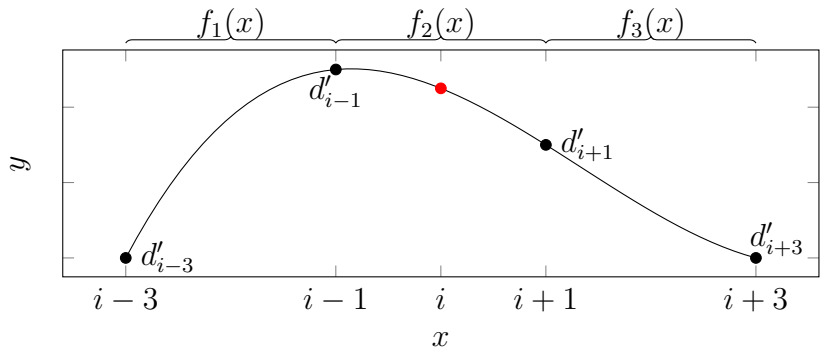


Figure 3.6: Cubic Spline prediction: Cubic Spline with not-a-knot condition (third derivative equal at edges). Red point is the predicted value for d_i

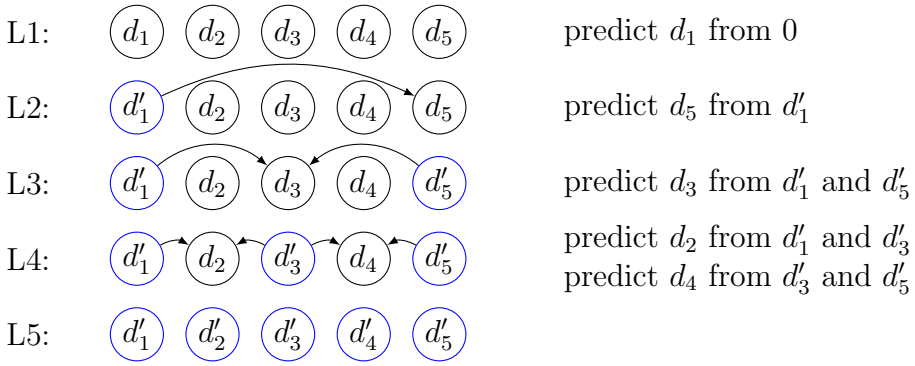


Figure 3.7: SZ3: Multilevel linear spline interpolation prediction

Differential Checkpoints (Lossy)

The idea behind differential checkpoints is that the changes of PageRank values become smaller with each iteration (they converge). To make use of that feature we write a full checkpoint of all PageRank values every x checkpoints and in between the full checkpoints we write $x-1$ checkpoints of the difference between the old PageRank values (at the time of the last full Checkpoint) and the current values. We compress these checkpoints more lossy (less strict error bound). The secondary checkpoints therefore have an "updating" effect for values that have changed the most since the last full checkpoint. The tradeoff here is that we have an increased space requirement since we need to remember the PageRanks of the last full checkpoint.

3.2.3 Calibration

We need to calibrate the different compression methods we use. To this end we simulate faults. To test how precise checkpoints need to be, how often we need to write checkpoints, if we can gain performance increases by creating more precise checkpoints as the algorithm gets closer to convergence and if we can reduce the number of checkpoints we write by using checkpointless approaches (while they are viable).

3.3 Lossy Communication

Finally we want to test if we can speed up PageRank by compressing the communication data inspired by C-Coll [22]. This has less to do with fault tolerance but we want to try a different application for lossy compression in PageRank. One obvious problem of using lossy communication is that we lose the property that all PageRanks sum up to 1. To make sure we keep that property and to guarantee convergence every x iterations there needs to be a repair phase where we add/subtract the missing sum and after a certain amount of iterations communication should not be lossy otherwise we can not guarantee convergence.

4. Experiments

The experiments are run on the SuperMUC-NG which consist of 6,336 thin compute nodes each with 48 cores and 96 GB memory and has an internal network speed of 100 Gbit/s. We run the following experiments on 32 nodes so a total of 1536 cores unless stated otherwise. We use randomly generated BA (Barabási-Albert [1]) graphs with 800Mio vertices and 1.6Bn edges and GNM (Erdos-Renyi Graphs [17]) graphs with 2Bn vertices and 8Bn edges unless stated otherwise. The Graphs are generated with KaGen [18, 35]. In the GNM graph model ($G(n, M)$) a graph is chosen uniformly at random from the set of graphs with n vertices and M edges. The BA graph model works by adding the vertices one after another and using preferred attachment, meaning that the more links a vertex has the more likely it is to get new links. This causes the first vertices added to have the most links. For PageRank we set the damping factor to $d = 0.85$ as described in the original paper [6]. We use two different settings regarding the tolerance t. Setting 1 uses the same tolerance $t = 10^{-9}$ regardless of convergence criterion, making Sum and MaxChange converge slower. For Setting 2 we adjust the tolerance t for both Sum and MaxChange to converge about as fast as SquareSum $t_{Sum} = 10^{-7}, t_{MaxChange} = 10^{-2}$. To avoid confusion between these two settings we say normal tolerance ($t_{SquareSum} = 10^{-9}$), low tolerance ($t_{Sum} = 10^{-9}$) and strict tolerance ($t_{MaxChange} = 10^{-9}$) to describe the different conversion criteria when we use Setting 1. The parameter for tolerance differ between different PageRank implementations. The strict and low tolerance of setting 1 can lead to very slow convergence [34]. We create checkpoints using ReStore [23] unless stated otherwise. ReStore creates checkpoints in memory distributed amongst the different compute nodes. For ReStore we use a replication of 4, meaning 4 copies of the checkpointed data are distributed amongst the different nodes. We simulate faults by assuming that one of the cores looses its PageRank values. We then take the failed core as a replacement and restore the lost PageRank values on the failed core according to our different approaches.

4.1 Checkpointless Approaches

To test how well the checkpointless approaches perform depending on when the fault occurs. We simulate a fault every 10% of a faultless execution, causing one fault per total PageRank calculation. Example given if a faultless execution of PageRank would take 100 iterations, we simulate a fault at the 10th, 20th, ..., and 90th iteration. Figure 4.1 shows the performance of the different checkpointless approaches depending on when the fault occurs for different tolerances. The x-axis plots at which percentile of a faultless execution the failure occurs and the y-axis shows the increase in iterations relative to the number of iterations a faultless execution takes. The red line (restart) shows the performance of restarting PageRank instead. We see that all approaches fall below the restart line and thus outperform the naive approach of restarting from the beginning. For Local Iteration we use Divide Even to restore the PageRank values and perform 10 local iterations. We see that Divide Even outperforms the Local Iterations in our tests meaning that the PageRanks, the local iteration converge to differ from the actual PageRanks which causes a bigger

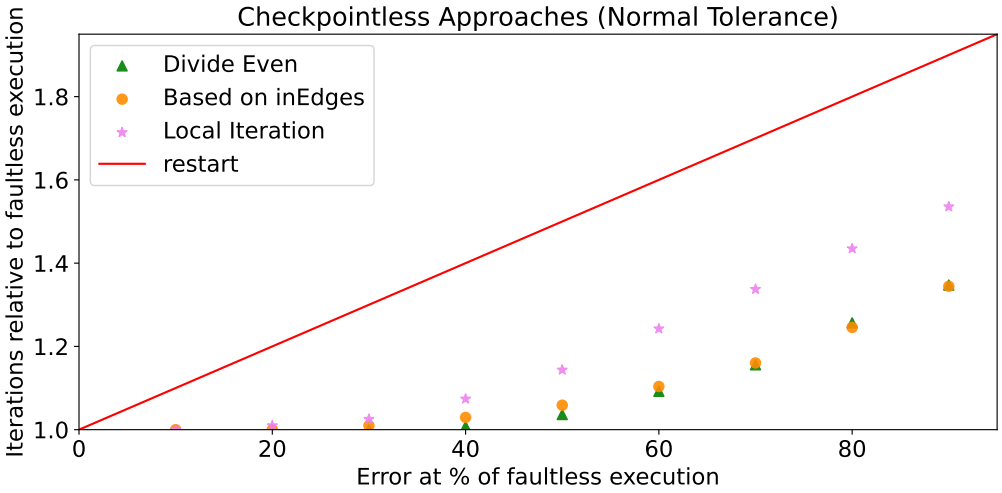
disruption when global calculations resume. We also see that Based on inEdges performs similar to Divide Even. Divide Even tends to perform slighly better for early faults as seen in Figure 4.1a but gets outperformed for later faults or when the tolerance t is strict as in Figure 4.1b. But using graph structure to assume the PageRank distribution as we do in Based on inEdges can also be hurtful as seen in Figure 4.1c which shows the performance of the different approaches for the it-2004 graph (crawl of the .it domain) where the PageRanks restored by Based on inEdges perform worse than those restored by Divide Even. When we adjust the tolerance so that all the convergence criteria converge as fast as with normal tolerance the checkpointless methods also behave similar as they do for normal tolerance giving us an average recomputation cost of 8∼15% for Divide Even when one fault occurs. Furthermore since for the first 40% of the PageRank calculations Divide Even tends to not incur any extra iterations when using normal tolerance (or adjusted convergence criteria). We consider only writing checkpoints after the 40% mark. We can estimate the progress of the PageRank algorithm based on how close PageRank is to convergence.

4.2 Calibration for Checkpoints

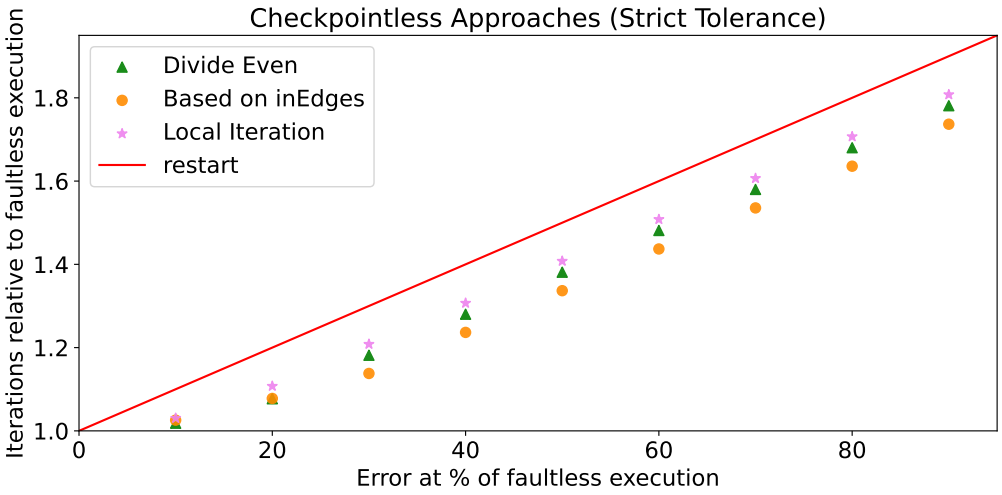
We calibrate the compression methods to minimize the cost of faults while also keeping the cost of checkpointing low. The parameters that we calibrate are the checkpoint frequency f and also the precision of the checkpoints (for lossy compression). Since the checkpointing cost is proportional to the amount of vertices, we set the checkpoint frequency inversely proportional to the amount of vertices (e.g checkpointing every $\frac{|V|}{x}$ seconds). To calibrate we test the different parameters we simulate 1 to 10 faults after 50% completion in each calculation. We begin by searching for a low checkpoint frequency for uncompressed checkpoints that reduces the amount of extra iteration incurred by a fault when we restore only the lost PageRanks (LoadFailed). After finding a good checkpoint frequency we calibrate the precision for lossy compression methods. We also check if lowering the checkpoint frequency allows us to use less precise checkpoints. In Figure 4.2 we can see how the precision of ZFP and the checkpoint frequency impact the increase of iterations caused by a fault, as well as the overhead for writing checkpoints. On the x-axis we have different precision settings for ZFP where ZFP_{i+1} is more precise than ZFP_i. We notice that more precise checkpoints are better for avoiding an increase in iterations than more frequent checkpoints.

4.2.1 Calibration Data

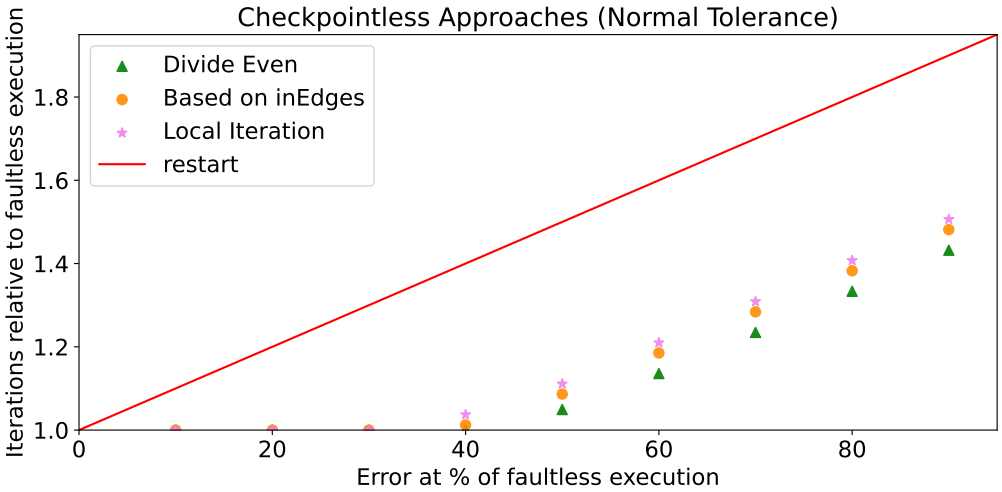
In Figure 4.3 we see how much the different aspect of the recovery affect the increase in execution time. The plots show the data for 2 faults occurring during computation. The y-axis shows the increase in run time in % and the x-axis shows different settings for the compression methods (higher index means more precise). On the left side we plot the results for ZFP and on the right side for SZ3. We see that the biggest cost factor is an increase in iterations which can be reduced by increasing the precision of the compression without increasing the checkpoint overhead too much. Also the cost for restoring data seems negligible. So to get the best performance in case of faults using precise checkpoints to decrease the amount of work lost tends to perform better. This also shows, that there is not much to be gained by making



(a) Performance comparison for Normal Tolerance

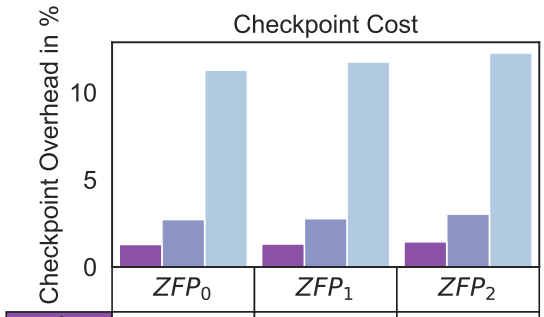


(b) Performance comparison for Strict Tolerance

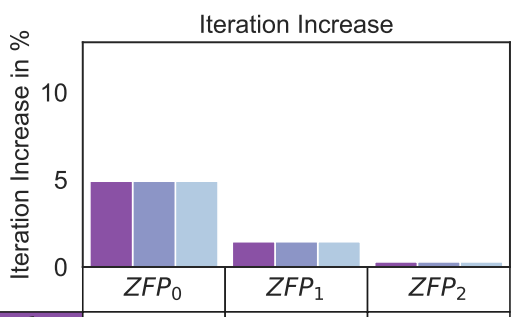


(c) Unfavorable PageRank approximation by Based on inEdges (empirical graph)

Figure 4.1: Performance comparison for Checkpointless Approaches



	ZFP_0	ZFP_1	ZFP_2
$f = \frac{1}{8}Hz$	1.3	1.3	1.4
$f = \frac{1}{4}Hz$	2.7	2.8	3.0
$f = 1Hz$	11.3	11.7	12.3



	ZFP_0	ZFP_1	ZFP_2
$f = \frac{1}{8}Hz$	4.9	1.4	0.3
$f = \frac{1}{4}Hz$	4.9	1.4	0.3
$f = 1Hz$	4.9	1.4	0.3

Figure 4.2: The plots shows the impact higher checkpoint frequencies and more precise checkpoints have on the Checkpoint overhead and the increase in Iterations after a fault for ZFP.

checkpoints more precise the closer we are to converging, since it barely effects the checkpoint overhead while also potentially incurring a larger cost due to additional iterations.

4.3 Calculations

Based on the calibration data we try to estimate a good precision setting for the different compression methods depending on the MTBF (mean time between failures). In equation (4.1) we show how we calculate this for a parameter setting s and a MTBF μ. First we calculate an estimation of the extra time the algorithm needs if a fault occurs. We call this extra time $C_{Fault}(s)$. $C_{Fault}(s)$ consist of the time to restore a checkpoint and the assumed amount of extra iteration time including additional checkpoints for the given setting s. Here we assume that the amount of extra iterations caused by a fault is consistent. We calculate the number of faults we expect in an execution by dividing the run time without faults $T_{noF}(s)$ by the MTBF. The increase in time because of faults is then $\frac{T_{noF}(s)}{\mu} * C_{Fault}(s)$. Since the calculation time is now larger, there is also more time for faults to occur. The expected amount of extra faults caused by the first set of faults would then $\frac{T_{noF}(s)*C_{Fault}(s)}{\mu^2}$ and the extra time spend for this second set of faults $\frac{T_{noF}(s)*C_{Fault}(s)^2}{\mu^2}$. This gives us the equation for $TheoreticalTime(s,\mu)$ in equation (4.1). The theoretical run time for a MTBF μ converges when $C_{Fault}(s) < \mu$ allowing us to simplify the equation as seen in equation (4.1).

$$TheoreticalTime(s,\mu) = T_{noF}(s) \cdot \left(1 + \sum_{i \in \mathbb{N}} \left(\frac{CostFault(s)}{\mu}\right)^i\right)$$

$$TheoreticalTime(s,\mu) = \begin{cases} \infty, & \text{if } \frac{CostFault}{\mu} >= 1. \\ T_{noF}(s) + T_{noF}(s) * \frac{CostFault}{\mu - CostFault}, & \text{otherwise.} \end{cases} \tag{4.1}$$

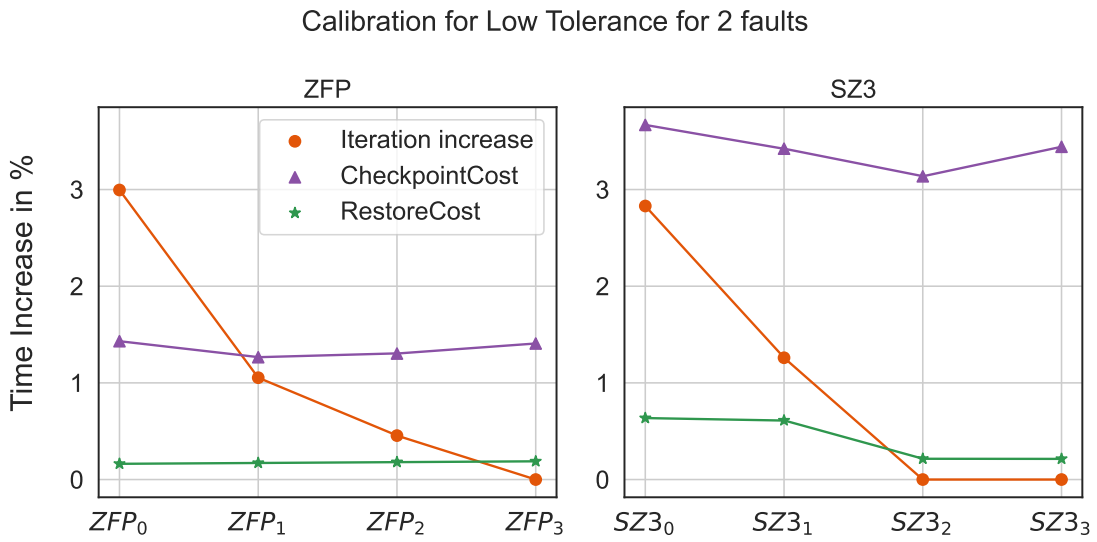


Figure 4.3: How strong the run time is effected based on the precision of the compression method

4.3.1 Theoretical Cost

In Figure 4.4 we show a plot for the Theoretical Cost function for different ZFP precision settings when we use the low tolerance settings. On the x-axis we have the MTBF μ in seconds and on the y-axis we have the expected increase in runtime in % compared to a faultless checkpointless execution. We show 3 ZFP settings with different precision from our calibration data (higher index means more precise). We see that if the MTBF μ is large enough the checkpoint overhead is more significant than the increase in iteration since faults are more unexpected. We see that if faults are rare enough (high MTBF) lower precision are preferable to keep the checkpoint overhead down. But since the execution time for our data is 1∼2 minutes and the cost of a fault may be different for longer running instances of PageRank it is uncertain whether or not they would behave similar.

4.3.2 Expected Runtime (Divide Even)

We calculate how the expected runtime of Divide Even changes based on the MTBF μ and the runtime of PageRank without faults T. We assume the PageRanks are calculated with normal tolerance where Divide Even works best. We have seen in figure 4.1c that the increase in iterations depends on when the fault occurs. For these calculations we are going to assume that every fault is the worst case and causes 50% extra iterations ($0.5 \cdot T$). We can calculate the expected runtime for Divide Even the same way as for checkpoints. The $TheoreticalRuntime/Cost = T \cdot (1 + \frac{0.5 \cdot T}{\mu - 0.5 \cdot T})$ which converges when $\mu > 0.5 \cdot T$. We now calculate when Divide Even outperforms a checkpointing approach. We assume that the checkpointing approach causes no extra iterations and the recovery time can be disregarded ($CostFault = 0$). Let $z = \frac{T_{noF}(s) - T}{T}$ be the relative checkpoint overhead in %. We calculate when the extra cost for faults for divide even intersects with the checkpoint overhead (Equation (4.2)). For example a checkpoint overhead of 1% is expected to be outperformed by Divide Even when the MTBF μ is 50.5 times larger than the runtime of the PageRank instance T.

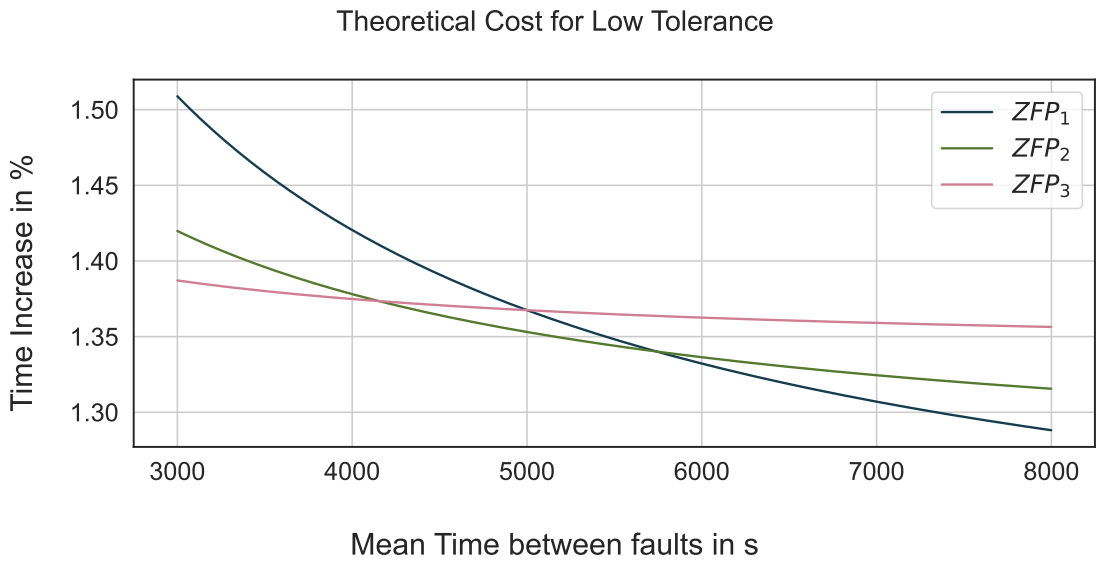


Figure 4.4: Theoretical increase in Time depending on MTBF for 3 different ZFP precision settings

$$\begin{aligned} T \cdot \frac{0.5T}{\mu - 0.5T} - z \cdot T &= 0 \\ \frac{\mu}{T} = \frac{z+1}{2z}&, \text{ With } \mu > 0.5 \cdot T \text{ and } z > 0 \end{aligned} \tag{4.2}$$

4.4 LoadAll vs LoadFailed

We examine the difference between loading all checkpoints and loading just the failed PageRank values. For the following we simulate a fault at 60, 70, 80, 90% of a faultless execution (1 fault per calculation). We test if LoadFailed can outperform LoadAll. In Figure 4.5 we see the results. On the y-axis the relative increase in iterations for LoadAll compared to LoadFailed $\left(\frac{LoadAll}{LoadFailed}\right)$. The results are grouped by the strictness of the tolerance. We see that for normal and low tolerance settings LoadFailed clearly outperforms LoadAll, with LoadAll on average needing 5% more iteration than LoadFailed. An exception here is SZ3 where LoadFailed outperforms LoadAll even more, this is not because LoadFailed works better for SZ3 but instead LoadAll tends to take extra iterations to complete. Since SZ3 works by casting all values as multiples of the error bound (Linear Quantisation) most values loose or gain some PageRank during decompression which when we load all checkpoints leads to small distortions on all vertices which sum up when we calculate convergence. When we have a very strict tolerance setting LoadAll and LoadFailed are more equally matched with LoadFailed still performing slightly better in the median. Because the tolerance is strict the PageRank values need to be very precise so we often incur the iterations we lost by loading a checkpoint as recomputation cost regardless of whether we load all or just the failed checkpoint.

4.5 Compression Comparison

Exponent, Bucket, and Cutoff compression do not perform well, they either cause a significant increase the recomptutation cost (more than the iterations lost) or barely

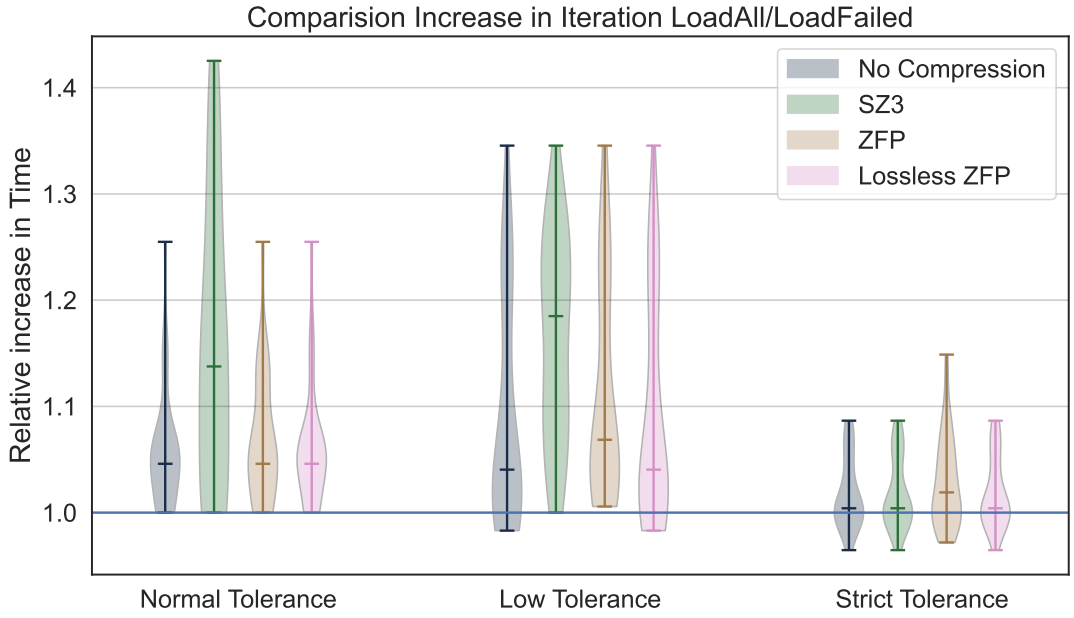


Figure 4.5: Increase in Iteration for LoadAll compared to LoadFailed

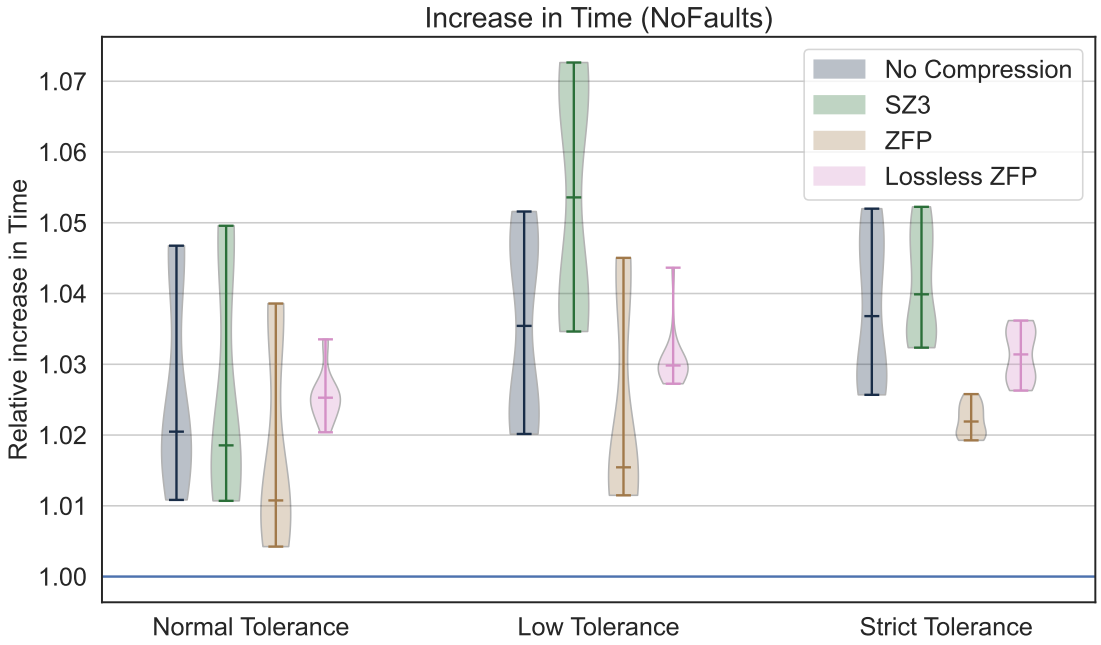
compress the data. In Figure 4.6a we see the increase in execution time when no faults occur while writing checkpoints. On the y-axis we see the increase in time relative to a faultless checkpointless execution and on the x-axis we have different compression methods. The checkpoint overhead can be kept under 5% and for the best performing compression ZFP the median checkpoint overhead lies around 1%. Similar in Figure 4.6b we see the performance of the different compression methods when a fault occurs. Here we simulate a fault at 60, 70, 80 and 90% of a faultless execution. We see that the increase in calculation time tends to be less then 10%. Again ZFP performs slightly better than the rest with a median time increase of 1∼2%. In Table 4.1 we see the compression rate of the different compressors after calibration. The compression rate is the uncompressed size divided by the compressed size. When considering space concerns SZ3 is optimal for writing small checkpoints but it also has a larger compression overhead.

Compressor	Compression rate
SZ3	4.6∼7.6
ZFP	4∼4.4
ZFP_Lossless	1.07∼1.2

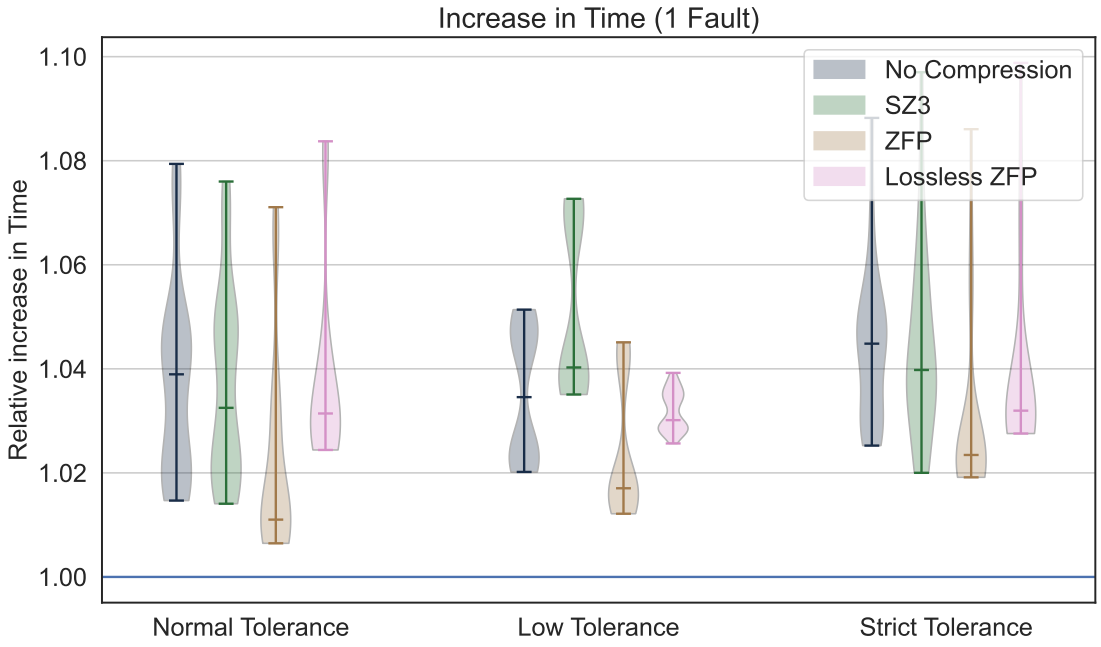
Table 4.1: Compression Rate of the different compressors for normal Tolerance

4.5.1 Differential Checkpoints

For differential checkpoints we write full checkpoints with ZFP, with the same calibration as if we would use only ZFP. In between the full checkpoints we write 3 differential checkpoints compressed with SZ3. The differential checkpoints are a lot stronger compressed compared to the full checkpoints as there purpose is to update values that have changed a lot since the last checkpoint. The differential checkpoints



(a) Increase in Time for no faults (checkpoint overhead)



(b) Increase in time for 1 fault

Figure 4.6: Performance of different checkpoints on synthetic graphs (32 nodes 48 cores each)

are about 25% percent faster than the full checkpoint and have a compression ratio of ca 2000:1. Since the differential checkpoints are so strongly compressed they increase only the total checkpoint size by ca 2%. In total using differential checkpoints lowers the overhead compared to using only ZFP checkpoints by around 16%.

4.5.2 Empirical Graphs

After seeing the performance on generated graphs we test how well the calibrated settings perform on empirical graph instances. We use graphs from the Laboratory of web algorithms [2, 4]. Namely the it-2004 (41Mio vertices, 1.1Bn edges), the sk-2005 (50Mio vertices, 1.9Bn edges), the uk-2005 (39Mio vertices, 900Mio edges) and the arabic-2005 (22Mio vertices, 639Mio edges) graph which are all webcrawls gathered by the UbiCrawler [5].
In Figure 4.8 we see the performance of the different compression methods on empirical Graphs. Like in the last section we have a fault at 60, 70, 80 and 90% completion of a faultless execution. In most of our test the faults caused no extra iterations. But we can notice some outliers for SZ3. The calibration of SZ3 did not as easily transfer to the empirical data sets. The cost and checkpoint overhead for ZFP are around 2∼3%. The checkpoints work well even for empirical graphs but since the empirical graph instances have a lot less vertices and we base the checkpoint frequency on the number of vertices we generally have a higher checkpoint overhead as seen in Figure 4.7. The best performing method here are differential checkpoints with a median checkpoint overhead of 1∼3%. We also see that the checkpoint overhead when using the Sum convergence criterion is higher. This is the case because our calibration to avoid writing checkpoints while checkpointless approaches are still sufficient is not as good adjusted as for the other convergence criteria. Causing Sum to write up to 40% more checkpoints. We also notice that Lossless ZFP performs worse than No Compression since the overall checkpoint including compression time is larger. The checkpoint time is also larger in our synthetic experiment (figure 4.6a) but the overall performance of Lossless ZFP was still better than No Compression possibly due to the different memory access.

Big Graphs

We test how well the checkpoints work on larger graph instances. These graphs are also from the Laboratory of web algorithms [2, 4] and were gathered by BUbiNG [3]. We use the gsh-2015 (988M vertices, 33B edges) and uk-2014 (787M vertices, 47B edges) graph instances. For these graph instances we use 100 compute nodes 48 cores each. We again simulate faults at 60, 70, 80 and 90%. We see the result in Figure 4.9. We see that the checkpointing overhead is a lot lower for these bigger graphs because they have more vertices and thus checkpoint less frequent. But overall the performance of the different checkpointing methods is similar to the experiments on smaller graphs with less compute nodes.

4.5.3 SZ3 predictor

We test how well the predictor works for our PageRank values by testing it against using only linear quantisation (a smallest unit). We use the same error bound as we do for SZ3 and look at the error distribution, the difference between the actual PageRank values and the compressed PageRanks. In Figure 4.10 we show the error

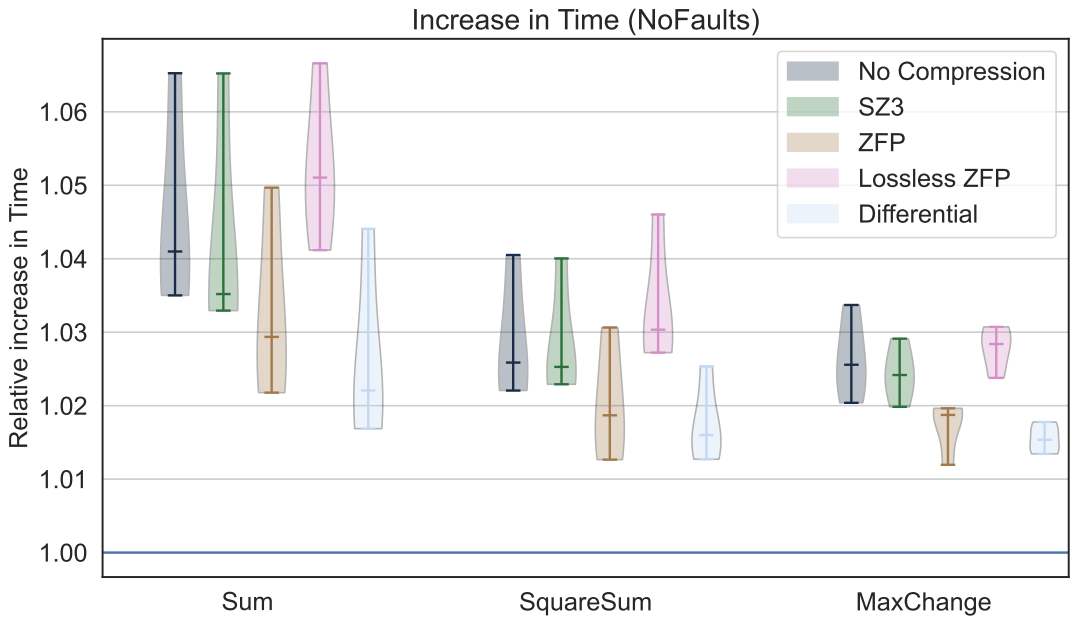


Figure 4.7: Increase in Time NoFaults (CostFaultless) for empirical graphs

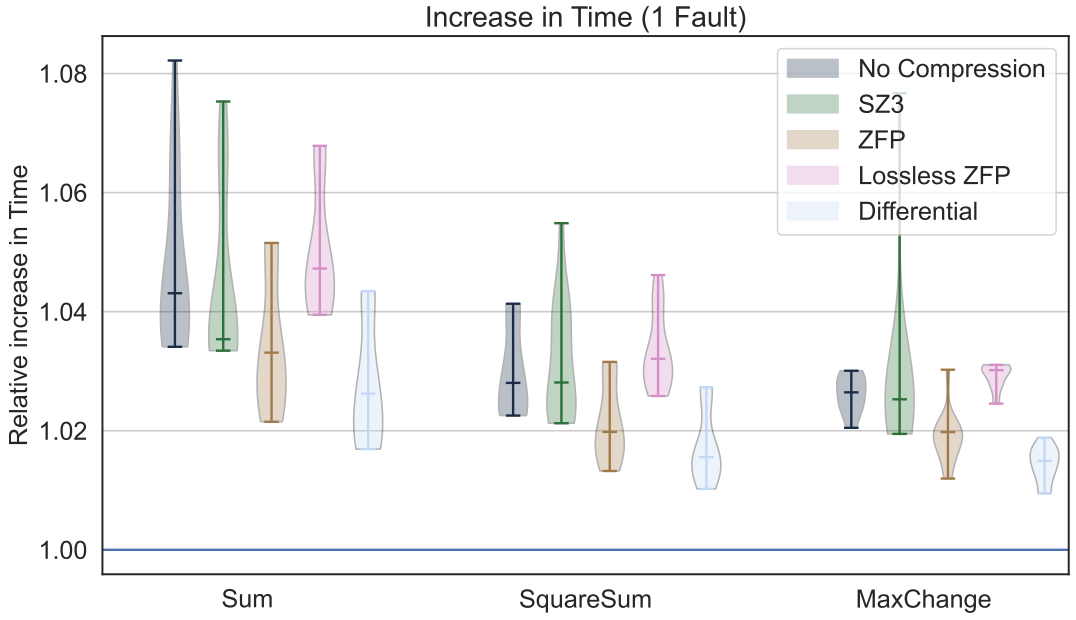


Figure 4.8: Increase in Time with 1 fault (CostFault) for empirical graphs

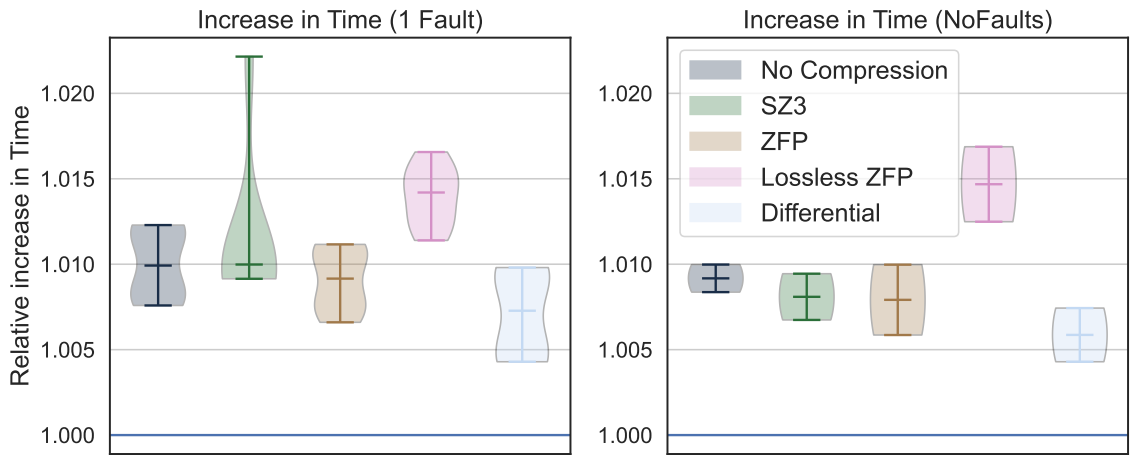


Figure 4.9: 100 Nodes each 48 Tasks, results for no fault and 1 Fault on gsh-2015 and uk-2014 graphs ca 800M∼1B vertices and 30B∼50B edges

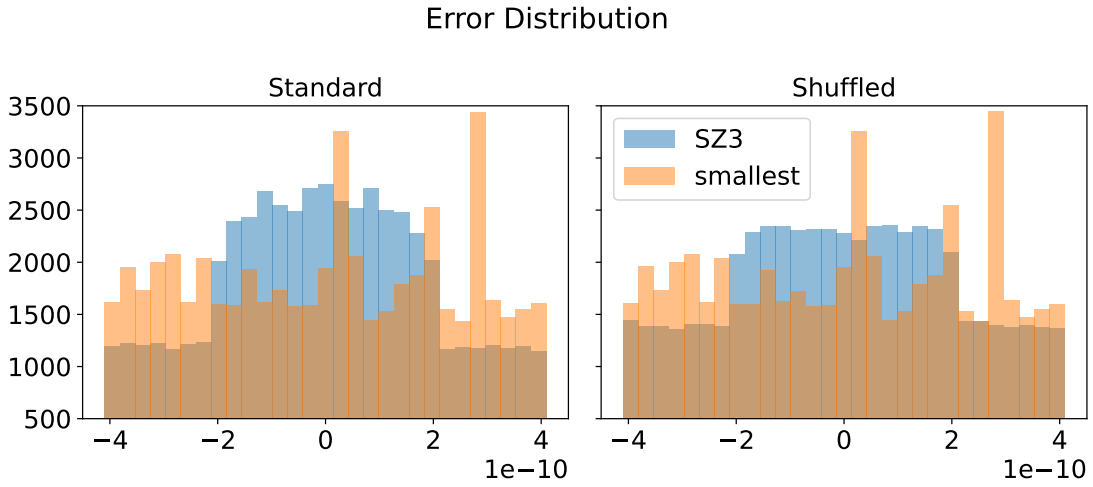
distribution when using the value prediction of SZ3 and just rounding every value to a nearest smallest number as well as the correlation between the PageRanks of vertices with neighbouring indices. On the left side we see the results if the vertex indices a not randomly permuted we see that the correlation between PageRanks of neighbouring indices is quite high (Figure 4.10b). On the right side we see the values after randomly permuting the vertex indices, here we have little correlation between values. When we look at the histogram of the error distribution in Figure 4.10a we see that the predictor of SZ3 produces a much better error spread than simply using linear quantisation. Only linear quantisation causes an even spread of the error but in conjunction with the SZ3 predictor it manages to keep the majority of values beneath half the error bound.

4.6 Lossy Comunication

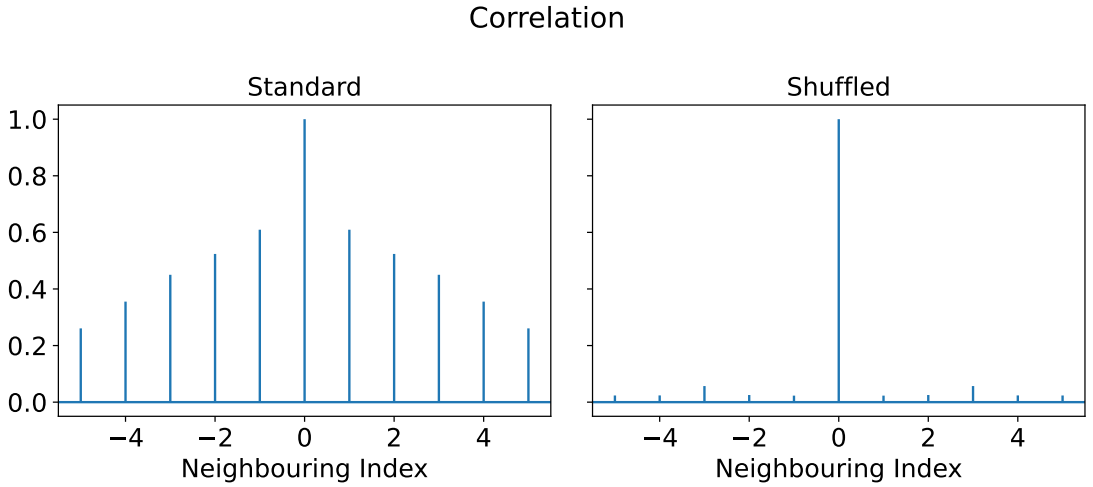
To reduce the amount of data sent in every iteration we compress the communication data (PageRank updates) before sending them. A problem here is that this can break the convergence of PageRank. Since the amount of total amount of PageRank is no longer guaranteed 1. We noticed that simply removing/adding the missing value will not always fix the problem with convergence. To insure convergence we switch back to normal communication after a certain convergence threshold is reached or when we no longer converge. We measure the time needed for communication each iteration to see the effect of lossy communication. In Table 4.2 we see that lossy communication ends up slows down the total communication. The overhead for compression and decompression is larger then the time it takes to send the data. The actual sending of data is substantially faster (around 30%).

	Communication	Lossy Communication	Compression	Decompression
Time BA	0.42s	0.3s	1.7s	1.7s
Time GNM	0.34s	0.22s	0.25s	0.31s

Table 4.2: Average Time for communication each iteration



(a) Error Distribution with and without predictor



(b) Correlation between vertices with neighbouring vertices.

Figure 4.10: Error Distribution and Correlation between neighbouring data points

5. Conclusion

After trying different checkpointless methods (DivideEven, BasedOnInEdges, Local Iterations) as well as writing uncompressed and compressed checkpoints, to increase PageRanks fault tolerance for fail-stop errors. We think that both checkpointless approaches and lossy compressed checkpoints are a good way to make PageRank more fault tolerant. The checkpointless approach Divide Even outperforms restarting by $40 \sim 50\%$. The lack of overhead makes Divide Even useful for systems with high mean time between failures compared to the expected runtime of the PageRank instances. Since for a MTBF μ that is 50.5 times the runtime of the PageRank instance, Divide Even is expected to outperform a 1% checkpoint overhead. Making Divide Even useful in systems that have a high MTBF compared to the expected runtime of the PageRank instances. But for systems with low MTBF compared to the run time of the PageRank instances writing checkpoints is useful. Using checkpointless approaches in the beginning we can reduce the checkpoint overhead by around 40%. The checkpointless approaches performance also depends on how much vertices are lost due to a failure, meaning they could potentially work even better for larger systems, assuming the failures cause us to loose less a smaller fraction of the data. The checkpoints compressed with ZFP have around 40% less overhead compared to uncompressed checkpoints and performed equally in case of a failure. The use of differential checkpoints with strongly compressed secondary checkpoints reduces the checkpoint overhead by another 16% compared to using only ZFP.
Future Work: It remains to be seen if the checkpointless methods can be used in other converging algorithms to decrease the checkpoint overhead. Since the checkpointless methods rely on the properties of PageRank. There is also the question how the performance of these methods scales with the amount of data lost. It would also be interesting to see if the use of differential checkpoint with strongly compressed secondary checkpoints is useful in other converging algorithms since the secondary checkpoints work as an update function on the most changes values.

Bibliography

[1] Réka Albert and Albert-László Barabási. "Statistical mechanics of complex networks". In: *ArXiv* cond-mat/0106096 (2001). URL: https://api.semanticscholar.org/CorpusID:60545.

[2] Paolo Boldi and Sebastiano Vigna. "The WebGraph Framework I: Compression Techniques". In: *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. Manhattan, USA: ACM Press, 2004, pp. 595–601.

[3] Paolo Boldi et al. "BUbiNG: Massive Crawling for the Masses". In: *Proceedings of the Companion Publication of the 23rd International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2014, pp. 227–228.

[4] Paolo Boldi et al. "Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks". In: *Proceedings of the 20th international conference on World Wide Web*. Ed. by Sadagopan Srinivasan et al. ACM Press, 2011, pp. 587–596.

[5] Paolo Boldi et al. "UbiCrawler: A Scalable Fully Distributed Web Crawler". In: *Software: Practice & Experience* 34.8 (2004), pp. 711–726.

[6] Sergey Brin and Lawrence Page. "The anatomy of a large-scale hypertextual Web search engine". In: *Computer Networks and ISDN Systems* 30.1 (1998). Proceedings of the Seventh International World Wide Web Conference, pp. 107–117. ISSN: 0169-7552. DOI: https://doi.org/10.1016/S0169-7552(98)00110-X. URL: https://www.sciencedirect.com/science/article/pii/S016975529800110X.

[7] Michael Brinkmeier and Michael. "PageRank Revisited". In: *ACM Transaction on Internet Technologies* 6 (Aug. 2006), pp. 257–. DOI: 10.1145/1151087.1151090.

[8] Jon Calhoun et al. "Exploring the feasibility of lossy compression for PDE simulations". In: *The International Journal of High Performance Computing Applications* 33.2 (2019), pp. 397–410. DOI: 10.1177/1094342018762036. eprint: https://doi.org/10.1177/1094342018762036. URL: https://doi.org/10.1177/1094342018762036.

[9] K. Mani Chandy and Leslie Lamport. "Distributed Snapshots: Determining Global States of Distributed Systems". In: *ACM Trans. Comput. Syst.* 3.1 (Feb. 1985), pp. 63–75. ISSN: 0734-2071. DOI: 10.1145/214451.214456. URL: https://doi.org/10.1145/214451.214456.

[10] Zizhong Chen. "Online-ABFT: An Online Algorithm Based Fault Tolerance Scheme for Soft Error Detection in Iterative Methods". In: *SIGPLAN Not.* 48.8 (Feb. 2013), pp. 167–176. ISSN: 0362-1340. DOI: 10.1145/2517327.2442533. URL: https://doi.org/10.1145/2517327.2442533.

[11] J.T. Daly. "A higher order estimate of the optimum checkpoint interval for restart dumps". In: *Future Generation Computer Systems* 22.3 (2006), pp. 303–312. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2004.11.016. URL: https://www.sciencedirect.com/science/article/pii/S0167739X04002213.

[12] Sheng Di and Franck Cappello. "Fast Error-Bounded Lossy HPC Data Compression with SZ". In: *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2016, pp. 730–739. DOI: 10.1109/IPDPS.2016.11.

[13] James Diffenderfer et al. "Error Analysis of ZFP Compression for Floating-Point Data". In: *SIAM J. Sci. Comput.* 41 (2018), A1867–A1898. URL: https://api.semanticscholar.org/CorpusID:73708315.

[14] Jack Dongarra et al. "The International Exascale Software Project: A Call to Cooperative Action by the Global High Performance Community". In: *International Journal of High Performance Computing Applications (to appear)* (July 2009).

[15] Yishu Du et al. "Optimal Checkpointing Strategies for Iterative Applications". In: *IEEE Transactions on Parallel and Distributed Systems* 33.3 (2022), pp. 507–522. DOI: 10.1109/TPDS.2021.3099440.

[16] Griffin Dube et al. "SIMD Lossy Compression for Scientific Data". In: *CoRR* abs/2201.04614 (2022). arXiv: 2201.04614. URL: https://arxiv.org/abs/2201.04614.

[17] P. Erdös and A. Rényi. "On Random Graphs I". In: *Publicationes Mathematicae Debrecen* 6 (1959), p. 290.

[18] Daniel Funke et al. "Communication-free Massively Distributed Graph Generation". In: *2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2018, Vancouver, BC, Canada, May 21 – May 25, 2018*. 2018.

[19] Thomas Herault and Yves Robert. *Fault-Tolerance Techniques for High-Performance Computing*. 1st. Springer Publishing Company, Incorporated, 2015. ISBN: 3319209426.

[20] Thomas Herault and Yves Robert. *Fault-Tolerance Techniques for High-Performance Computing*. 1st. Springer Publishing Company, Incorporated, 2015. ISBN: 3319209426.

[21] Mark Frederick Hoemmen and Michael Allen Heroux. *Fault-tolerant iterative methods.* Tech. rep. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2011.

[22] Jiajun Huang et al. *C-Coll: Introducing Error-bounded Lossy Compression into MPI Collectives*. 2023. arXiv: 2304.03890 [cs.DC].

[23] Lukas Hübner et al. "ReStore: In-Memory REplicated STORagE for Rapid Recovery in Fault-Tolerant Algorithms". In: *2022 IEEE/ACM 12th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS)*. 2022, pp. 24–35. DOI: 10.1109/FTXS56515.2022.00008.

[24] Lorenz Hübschle-Schneider and Peter Sanders. "Linear Work Generation of R-MAT Graphs". In: *Network Science* 8.4 (2020), pp. 543–550.

[25] Tanzima Zerin Islam et al. "MCREngine: A scalable checkpointing system using data-aware aggregation and compression". In: *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 2012, pp. 1–11. DOI: 10.1109/SC.2012.77.

[26] J. Langou et al. "Recovery Patterns for Iterative Methods in a Parallel Unstable Environment". In: *SIAM Journal on Scientific Computing* 30.1 (2008), pp. 102–116. DOI: 10.1137/040620394. eprint: https://doi.org/10.1137/040620394. URL: https://doi.org/10.1137/040620394.

[27] Xin Liang et al. "An Efficient Transformation Scheme for Lossy Data Compression with Point-Wise Relative Error Bound". In: *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. 2018, pp. 179–189. DOI: 10.1109/CLUSTER.2018.00036.

[28] Peter Lindstrom. "Fixed-Rate Compressed Floating-Point Arrays". In: *IEEE Transactions on Visualization and Computer Graphics* 20 (Aug. 2014). DOI: 10.1109/TVCG.2014.2346458.

[29] Richard von Mises and Hilda Pollaczek-Geiringer. "Praktische Verfahren der Gleichungsauflösung ." In: *Zamm-zeitschrift Fur Angewandte Mathematik Und Mechanik* 9 (), pp. 152–164. URL: https://api.semanticscholar.org/CorpusID:120916484.

[30] A. Mitra. "On Finite Wordlength Properties of Block-Floating-Point Arithmetic". In: (July 2005).

[31] Lawrence Page et al. "The PageRank Citation Ranking : Bringing Order to the Web". In: *The Web Conference*. 1999.

[32] Oskar Perron. "Zur Theorie der Matrices". In: *Mathematische Annalen* 64 (1907), pp. 248–263. URL: https://api.semanticscholar.org/CorpusID:123460172.

[33] Aurick Qiao et al. "Fault Tolerance in Iterative-Convergent Machine Learning". In: *CoRR* abs/1810.07354 (2018). arXiv: 1810.07354. URL: http://arxiv.org/abs/1810.07354.

[34] Subhajit Sahu, Kishore Kothapalli, and Dip Sankar Banerjee. *Adjusting PageRank parameters and Comparing results*. 2021. arXiv: 2108.02997 [cs.DM].

[35] Peter Sanders and Christian Schulz. "Scalable generation of scale-free graphs". In: *Information Processing Letters* 116.7 (2016), pp. 489–491.

[36] Naoto Sasaki et al. "Exploration of Lossy Compression for Application-Level Checkpoint/Restart". In: May 2015, pp. 914–922. DOI: 10.1109/IPDPS.2015.67.

[37] Yanyan Shen et al. "Fast failure recovery in distributed graph processing systems". In: *Proceedings of the VLDB Endowment* 8 (Dec. 2014), pp. 437–448. DOI: 10.14778/2735496.2735506.

[38] Dingwen Tao et al. "Improving performance of iterative methods by lossy checkponting". In: *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, June 2018. DOI: 10.1145/3208040.3208050. URL: https://doi.org/10.1145%2F3208040.3208050.

[39] Dingwen Tao et al. "Significantly Improving Lossy Compression for Scientific Data Sets Based on Multidimensional Prediction and Error-Controlled Quantization". In: *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2017, pp. 1129–1139. DOI: 10.1109/IPDPS.2017.115.

[40] Brian Vargas. "Exploring PageRank Algorithms: Power Iteration & Monte Carlo Methods". MA thesis. California State University, San Marcos, 2020.

[41] Da Yan et al. "Lightweight Fault Tolerance in Pregel-Like Systems". In: *Proceedings of the 48th International Conference on Parallel Processing*. ICPP '19. Kyoto, Japan: Association for Computing Machinery, 2019. ISBN: 9781450362955. DOI: 10.1145/3337821.3337823. URL: https://doi.org/10.1145/3337821.3337823.

[42] John W. Young. "A First Order Approximation to the Optimum Checkpoint Interval". In: *Commun. ACM* 17.9 (Sept. 1974), pp. 530–531. ISSN: 0001-0782. DOI: 10.1145/361147.361115. URL: https://doi.org/10.1145/361147.361115.