Improving Fault Tolerance and Accuracy of a Distributed Reduction Algorithm

Gerhard Niederbrucker*, Hana Straková*, and Wilfried N. Gansterer*
*University of Vienna, Austria
Research Group Theory and Applications of Algorithms

***Abstract*—Most existing algorithms for parallel or distributed reduction operations are not able to handle temporary or permanent link and node failures. Only recently, methods were proposed which are in principal capable of tolerating link and node failures as well as soft errors like bit flips or message loss. A particularly interesting example is the push-flow algorithm. However, on closer inspection, it turns out that in this method the failure recovery often implies severe performance drawbacks. Existing mechanisms for failure handling may basically lead to a fall-back to an early stage of the computation and consequently slow down convergence or even prevent convergence if failures occur too frequently. Moreover, state-of-the-art fault tolerant distributed reduction algorithms may experience accuracy problems even in failure free systems.**

We present the *push-cancel-flow* (*PCF*) *algorithm*, a novel algorithmic enhancement of the push-flow algorithm. We show that the new push-cancel-flow algorithm exhibits superior accuracy, performance and fault tolerance over all other existing distributed reduction methods. Moreover, we employ the novel PCF algorithm in the context of a fully distributed QR factorization process and illustrate that the improvements achieved at the reduction level directly translate to higher level matrix operations, such as the considered QR factorization.

I. Introduction

Global all-to-all reduction operations, such as summation or averaging, are among the most frequently performed operations on HPC systems. While fast and efficient implementations of all-to-all reductions exist for today's HPC systems [1], they are quite fragile in the sense that a *single* failure leads to a wrong result on *many* nodes. Furthermore, it is commonly expected that the future may bring a slight shift from traditional parallel HPC systems towards less tightly coupled systems which need to be more distributed in nature due to the extreme scale and complexity required to go to Exascale and beyond. Hence, in the future we may face hardware systems where even such simple and atomic operations as global all-to-all reductions have to be robust against various types of system failures. Due to the low complexity of a single reduction/aggregation operation, it is clear that fault tolerance at the algorithmic level is the only relevant approach in this case. In general, fault tolerance at the algorithmic level is anticipated to become more important in future HPC systems, since we have to prepare for a lot more soft errors than today due to the shrinking integrated circuit scale and the increasing overall system complexity.

In this paper, we investigate how the fault tolerance mechanisms incorporated in modern distributed reduction algorithms influence their behavior in practice. Algorithms like the *push-flow algorithm* [2], [3] have been shown to be able to handle link and node failures as well as soft errors like bit flips or message loss. But commonly, fault tolerance questions are treated at a theoretical level where practical aspects like floating-point arithmetic are neglected. We perform a closer inspection which reveals several weaknesses of these fault tolerance concepts not discussed in the literature so far. Subsequently, we illustrate how the discovered (practical) problems can be resolved. Our improvements preserve all existing (theoretical) fault tolerance properties.

While traditional all-to-all reductions for parallel systems require complex structured communication sequences [4], *gossip-based* [5] distributed reduction algorithms [2], [3], [5]–[8] are entirely based on randomized communication schedules and nearest-neighbor communication. Thus, they do not require any kind of synchronization. In contrast to traditional parallel all-to-all reduction algorithms, distributed reduction algorithms produce on each node a sequence of estimates of the target aggregate which converges towards the correct value. Therefore, at each point of time t during the computation, the local error of the estimate of the aggregate at every node is bounded by some $\epsilon(t) > 0$. The convergence speed of the distributed approaches we consider in this paper strongly depends on the topology of the system on which the computation is performed. However, in fact they exhibit the same scaling behavior as existing parallel approaches, since both, the parallel as well as the distributed reduction algorithms, require a short network diameter. More precisely, a system with n nodes which allows for utilizing an efficient traditional parallel reduction method requiring $\mathcal{O}(\log n)$ time steps (either due to a suitable physical network topology or due to some overlay network and very efficient routing technology), will also allow for a fast gossip-based reduction in $\mathcal{O}(\log n + \log \epsilon^{-1})$ time (cf. [5]), where ϵ is the accuracy of the aggregate reached at all nodes. Thus, while gossip-based computation of aggregates up to machine precision introduces a constant overhead compared to state-of-the-art parallel algorithms, higher level matrix operations, such as linear or eigenvalue solvers, can benefit from the iterative nature of gossip-based reduction algorithms for saving communication costs (cf. [9]). More generally, all commonly required functionality in numerical linear algebra

(cf. BLAS [10]) is based on the computation of sums and dot products. In a distributed setting, these building blocks can be computed by the reduction algorithms considered here. Hence, providing fault tolerance already on this lowest level allows for the design of naturally fault tolerant distributed matrix computations [9], [11] (cf. Section IV).

A major advantage of gossip-based all-to-all reduction algorithms is their flexibility which can be exploited for tolerating failures at the algorithmic level. This has been illustrated in [2], [3] where the push-sum algorithm [6] has been enhanced to the more resilient *push-flow algorithm (PF)*.

1) Contributions of this paper: First of all, we identify common problems of existing distributed reduction algorithms, such as numerical inaccuracies which clearly restrict their practical applicability. We also show that these problems get worse with increasing scale, i. e., with an increasing number of nodes. The main contribution of this paper is the *push-cancel-flow algorithm* (*PCF*), an improved version of the recently introduced PF algorithm [2], [3], which (i) improves the numerical accuracy in the final results and (ii) can even recover from permanent failures (such as permanent link failures) with very low overhead. Note that the computational efficiency of the PF algorithm in a failure-free environment is fully preserved in our new PCF algorithm. Last, but not least, we illustrate in a prototypical case study of a fully distributed QR factorization that the achieved improvements can be directly carried over to higher level matrix algorithms.

2) Related Work: Fault tolerant methods for large scale computer systems have a long tradition. Solution approaches span a wide range from entirely hardware based solutions to methods working exclusively at the algorithmic level.

A classical generic, entirely hardware-based approach is TMR (triple modular redundancy, [12]), where three identical systems are run in parallel and the results are determined by a two-out-of-three voting. Due to resource and energy constraints this approach is not suitable nowadays and cannot cope with the challenges raised by future extreme scale systems. Contrary to TMR, more recent approaches incorporate redundancy at the message passing level, i. e., in MPI libraries, resulting in much more efficient solutions [13]. However, such approaches focus only on communication correctness, i. e., the correctness of the messages sent over MPI.

A widely used fault tolerance paradigm is checkpoint/restart (C/R), where the system state is—either automatically or by a user request—written to stable storage, such that the application can be continued from a previously stored checkpoint. The major drawback shared by all flavors of C/R approaches is the overhead introduced by the expensive disk I/O. To avoid this bottleneck and to reduce the overhead of C/R in the absence of system failures, diskless checkpointing was introduced [14]. Another property shared by C/R methods is that they require information about occurring failures such that the recovery procedures can be launched. Consequently, silently occurring soft errors like bit-flips are out of scope of purely C/R-based solutions and require a more sophisticated logic, e. g., at the algorithmic level.

Due to the ever shrinking integrated circuit technology soft errors are expected to become the rule rather than the exception. Even though hardware solutions for soft error resilience (like ECC memory) are available, they are very expensive and limited to a small number of soft errors due to the overhead they introduce. Therefore, fault tolerance already at the algorithmic level is a necessity to meet the fault tolerance challenges which are anticipated for the future.

A classical technique which targets the emerging need of soft error resilience in the context of matrix operations is *ABFT* (*algorithm-based fault tolerance* [15]). In ABFT soft error resilience is achieved by extending the input matrices by checksums and by adapting the algorithms such that these checksums are updated correctly. Consequently, the checksums are used for detecting and recovering from soft errors. The original ABFT work [15] was restricted to a single soft error and matrix-matrix multiplication. Later work extended the principal concept to handle more failures [16] and to integrate it with matrix factorizations [17]. Moreover, in the last years the ABFT concept was extended to fail-stop failures [18], where a failed process cannot be continued and all its data is lost. The utilization of naturally available redundancy in iterative solvers for increasing fault tolerance was investigated in [19]. In contrast to classical ABFT work where failures are detected and corrected in the final result, recent ABFT-based approaches focus on detecting soft errors as early as possible to avoid their propagation [20]. Furthermore, ABFT-based factorization algorithms were shown to perform well on hybrid [21] as well as on large-scale [22] systems.

While ABFT is a promising approach within the context of matrix computations, this concept is not attractive for operations as simple as reductions due to the missing structure, the low complexity and the resulting high relative overhead of checksum strategies in this kind of computation. On the other hand, other known fault tolerance approaches are even less suitable due to the low complexity and atomicity of a reduction operation. Consequently, the fault tolerant computation of reduction operations in a distributed environment requires naturally fault tolerant, self-healing algorithms. As analyzed in [3], the PF algorithm shows superior theoretical properties over other competing approaches.

Since ABFT-based methods operate at the matrix level and are directly encoded into the algorithms, the corresponding algorithmic modifications needed for achieving fault tolerance are very specific for each matrix operation. In contrast to ABFT, by building matrix operations on top of

fault tolerant reduction algorithms, i. e., by performing all summations and dot products using a distributed reduction algorithm as discussed in this paper, the fault tolerance achieved on the reduction level directly translates to the higher (matrix) level. Of course, ABFT-based methods could still be used on top of this approach to provide an additional layer of fault tolerance.

Summarizing, gossip-based distributed reduction algorithms theoretically provide an exciting potential of fault tolerance and a logarithmic scaling behavior with the number of nodes. These attractive properties motivate a deeper investigation of their behavior in practice as building blocks for future reliable and resilient numerical algorithms.

3) Synopsis: Section II is devoted to an in-depth analysis of the shortcomings of existing distributed reduction algorithms. In Section III we develop our main results and present the *push-cancel-flow algorithm* (*PCF*), an improvement and extension of the push-flow algorithm which overcomes the issues identified in Section II. In Section IV we employ the PCF algorithm in the context of a fully distributed QR factorization algorithm and show that the improvements directly translate to higher level matrix operations. Section V concludes the paper.

II. Weaknesses of State-of-the-Art Methods

In this section we want to develop an understanding why substantial algorithmic improvements are needed such that existing distributed reduction algorithms with theoretically high resilience potential become useful in practice. For the sake of clarity we restrict our discussion to the PF algorithm. We want to point out, though, that the issues discussed in this section are common among all existing fault tolerant distributed reduction algorithms (cf. [23]). We start with reviewing the basic structure and functionality of the push-flow algorithm.

A. Review of the Push-Flow Algorithm (PF)

In principle, the PF algorithm can be viewed as a fault tolerant extension of the push-sum algorithm [6]. The push-sum algorithm is a gossip-based reduction algorithm which proceeds as follows. Initially (at time $t = 0$), every node i hosts some arbitrary (scalar or non-scalar) initial data $v_i(0)$. Afterwards, every node i iteratively randomly chooses a node in its neighborhood $\mathcal{N}_i$ (a nonempty fixed set of nodes i can communicate with) and sends half of its data, i. e., $v_i(t)/2$, to the chosen node. Receiving nodes combine the received data with their local data. Additionally, scalar weights are exchanged which determine the type of aggregation (summing, averaging, ...) to be performed. Even though no global control mechanism are assumed, a time complexity of $\mathcal{O}(\log n + \log \epsilon^{-1})$ for computing an ϵ-approximate of the target aggregate at each node can be proved for fully connected networks [6]. Later, this result was extended to all networks (i. e., local neighborhoods $\mathcal{N}_i$)

Input: Local initial data v_i and local weight w_i for each node i
Output: Estimate of global aggregate $(\sum_k x_k)/(\sum_k w_k)$

... initialization ...
1: $v_i \leftarrow (x_i, w_i)$
2: **for each** $j \in \mathcal{N}_i$ **do**
3: $\quad f_{i,j} \leftarrow (0, 0)$
4: **end for**
... on receive ...
5: **for each** received pair $f_{j,i}$ **do**
6: $\quad f_{i,j} \leftarrow -f_{j,i}$
7: **end for**
... on send ...
8: $k \leftarrow$ choose a random neighbor $k \in \mathcal{N}_i$
9: $e_i \leftarrow v_i - \sum_{j \in \mathcal{N}_i} f_{i,j}$
10: $f_{i,k} \leftarrow f_{i,k} + e_i/2$
11: send $f_{i,k}$ to node k

Figure 1. The push-flow algorithm [3] for the local computation of a global aggregate. The current local estimate of a node is computed as $e_i(1)/e_i(2)$, with $e_i = v_i - \sum_{j \in \mathcal{N}_i} f_{i,j}$, where $\mathcal{N}_i$ denotes node i's neighborhood.

which allow for a parallel reduction in $\mathcal{O}(\log n)$ time [5]. This implicit characterization comprises almost all networks of relevance. Practically, systems allow for fast reductions either by their physical topology or by some overlay network and efficient routing technologies. Thus, we can assume that the neighborhoods of the nodes are defined in an arbitrary way which allows for a fast reduction, e. g., $\mathcal{N}_i$ could be defined as the set of nodes with which node i communicates in an efficient parallel reduction process. As discussed in [3], time complexity results for the push-sum algorithm also hold for the PF algorithm.

Classical parallel reduction approaches like recursive doubling [4] require a pre-determined, sophisticated and well synchronized sequence of data movements. Contrary to that, the distributed reduction algorithms considered here only require the initial data to be preserved across the network for correctly performing an all-to-all reduction. More precisely, *mass conservation* needs to be ensured, i. e., $\sum_i v_i(t) = \sum_i v_i(0)$ needs to hold at all times t. Obviously, mass conservation is a *global* property which is violated by any kind of system failure. Consequently, the push-sum algorithm is not suited for the robust computation of all-to-all reductions. Henceforth, we omit explicit time dependencies, i. e., we write v_i instead of $v_i(t)$, etc., for the sake of a concise notation.

To overcome the drawbacks of the push-sum algorithm, the PF algorithm utilizes the graph theoretical flow concept in the following sense. For every neighboring node $j \in \mathcal{N}_i$ node i holds a *flow* variable $f_{i,j}$ representing the data sent to ("flowing from node i to") node j. Hence, the actual local data of node i is computed as $v_i = v_i(0) - \sum_{j \in \mathcal{N}_i} f_{i,j}$.

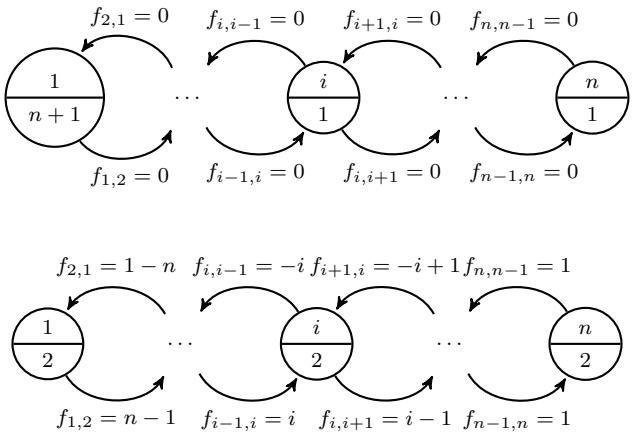

Figure 2. Schematic representation of a bus network before (top) and after (bottom) running the PF algorithm for averaging the local initial data. Nodes are consecutively numbered from 1 to n from left to right and represented by circles. Each circle contains the index (top) and the current local data (bottom) of the corresponding node i which is computed as $v_i - \sum_{j\in\mathcal{N}_i} f_{i,j}$ with node i's neighborhood $\mathcal{N}_i$ and initial data v_i.

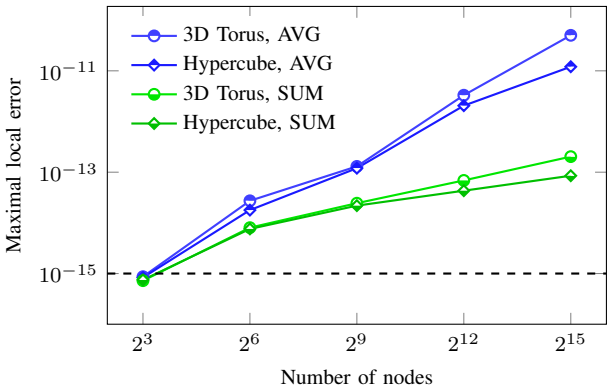

Figure 3. Globally achievable accuracy with the PF algorithm for increasing system size, different topologies and different types of aggregations.

While in the push-sum algorithm data (*mass*) is transferred between nodes, in the PF algorithm the nodes only share flows. More specifically, if node i wants to send $v_i/2$ to node $j \in \mathcal{N}_i$ it first sets $f_{i,j} = f_{i,j}+v_i/2$ ("virtual send") and then sends $f_{i,j}$ to j ("physical send"). The receiver negates the result, i. e., it sets $f_{j,i} = -f_{i,j}$ to ensure *flow conservation*, i. e., $\sum_i \sum_{j\in\mathcal{N}_i} f_{i,j} = 0$. In contrast to mass conservation, flow conservation is a *local* property and can be maintained more easily. Note that flow conservation implies mass conservation because of $\sum_i v_i(0) = \sum_i (v_i(0) - \sum_{j\in\mathcal{N}_i} f_{i,j}) = \sum_i v_i$. This implication highlights the role of the flow concept as major source of fault tolerance. A pseudocode of the PF algorithm is shown in Figure 1. Additionally, a concrete example which will be discussed in Section II-B is illustrated in Figure 2.

By utilizing graph theoretical flows, the PF algorithm naturally recovers from loss or corruption of messages as well as from soft errors like bit flips at the next successful failure-free communication without even detecting or correcting them explicitly (cf. [3]). Additionally, broken system components, i. e., permanently failed links or nodes, can be tolerated by setting the corresponding flow variables to zero, since this algorithmically excludes the failed components from the computation [3].

On the one hand, modern gossip-based distributed reduction algorithms like the PF algorithm have an attractive potential for providing fault tolerance while preserving logarithmic scaling with the number of nodes with a complexity of $\mathcal{O}(\log n + \log \epsilon^{-1})$. On the other hand, the next two sections will reveal substantial weaknesses occurring in practical computations. We will show how to overcome them in Section III.

B. Numerical Inaccuracy

A central need for reliable scientific algorithms is to provide a pre-determined level of accuracy in the results. In particular, resilient distributed reduction algorithms have to be able to compute results accurate to machine precision $\varepsilon_{\text{mach}}$. More precisely, the approximate aggregates $\tilde{r}_i$ at node i of a distributed all-to-all reduction with the exact result r should satisfy $\max_i |(\tilde{r}_i - r)/r| \leq c(n)\varepsilon_{\text{mach}}$ with a constant $c(n)$ which grows only modestly with n.

Existing analyses of distributed reduction algorithms are commonly of a purely theoretical nature and neglect the effects of floating-point arithmetic. Nevertheless, basic algorithms like the push-sum algorithm, which are not fault tolerant, meet the accuracy requirement stated above. In contrast to that, reduction algorithms which integrate mechanisms for achieving fault tolerance at the algorithmic level, commonly exhibit inaccuracies, especially with increasing n (cf. [23]). In the following, we illustrate the fundamental accuracy problems for the case of the PF algorithm with a simple example which can be easily analyzed.

Case Study—Bus Network: Consider a system consisting of n nodes, connected via a bus-style network which only allows for communication with neighboring nodes, i. e., for $1 < i < n$ node i can only communicate with its neighbors $i-1$ and $i+1$. Moreover, node 1 can only communicate with node 2 and node n can only communicate with node $n-1$. Again, $\mathcal{N}_i$ denotes the neighborhood of node i. Assume that every node hosts a single scalar value v_i and that these local values are initialized as $v_1 = n+1$ and $v_i = 1$ for all $1 < i \leq n$. For the sake of simplicity, we omit the weights in the PF algorithm here and assume them to be constantly one due to a regular, synchronous communication schedule. The upper part of Figure 2 illustrates this initial configuration. The flow variables hosted by a node are visually identified as the outgoing arcs.

If the PF algorithm is used to compute the global average (at each node) in this setup and if the effects of floating-

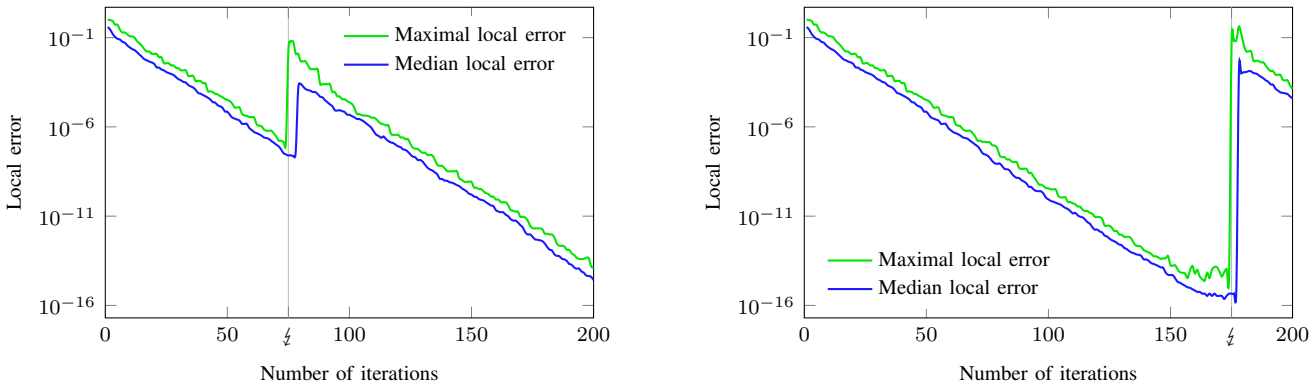

Figure 4. The PF algorithm algorithm is executed on a 6D hypercube and a single system failure is injected per experiment. The failure handling takes place after 75 (left) and 175 (right) iterations, respectively. Although the local accuracies achieved until the time of the failure differ strongly (much better in the right case), the PF algorithm falls back almost to the beginning of the computation in both cases.

point arithmetic are ignored, it reaches the state shown in the lower part of Figure 2. More specifically, convergence is reached if for all $1 < i \leq n$ the flow variables have the values $f_{i-1,i} = n-i+1$ and $f_{i,i-1} = -n+i-1$. Note that for growing n, the target aggregate remains two, whereas the values of the local flows grow linearly with n. Thus, several nodes can be expected to experience cancellation and accuracy problems in floating-point arithmetic since the average—and therefore also the local sum of flows $\sum_{j \in \mathcal{N}_i} f_{i,j}$—is constant, whereas the individual flows grow linearly and cancel each other out. Even if the sum of flows is stored in a single variable (for efficiency reasons) the updates of this variable will still lead to inaccurate results due to the linearly growing flow variables.

This example illustrates that several system parameters influence the final accuracy of the results of the PF algorithm. More specifically, the achievable accuracy depends on (i) the number of nodes, (ii) the network topology, (iii) the initial data distribution, and (iv) the type of target aggregate.

In Figure 3 we present experimental results for more realistic setups which verify the aforementioned concerns about the achievable numerical accuracy. Concretely, we evaluate the PF algorithm on 3D torus ($2^i \times 2^i \times 2^i$) and hypercube ($2^{3i}$ dimensional) topologies for computing sums and averages. We can clearly see how the resulting accuracy is influenced by varying system parameters. The major issue is obviously that the resulting accuracy rapidly gets worse with increasing scale.

C. Restart Implied by Failure Handling

As discussed in Section II-A there are two ways how system failures are handled in the PF algorithm. First, it naturally recovers from soft errors like lost messages or bit flips just by following the steps in the algorithm. Second, permanently failed components are excluded from the computation by setting the corresponding flow variables to zero: e. g., if the link between nodes i and j fails permanently, $f_{i,j}$ and $f_{j,i}$ are set to zero. To provide a concise presentation we restrict our discussion in the following to a single permanent link failure. While the importance of handling a single permanently failed link might not be clear at the first sight, we want to point out that a permanently failed node can be interpreted as a permanent failure of all its connecting communication links.

As we saw in the lower part of Figure 2, when the PF algorithm has converged, an equilibrium state is reached in which the flow variables describe a data flow which guarantees that every node can locally compute the global target aggregate. Moreover, we saw that the flows are not directly connected to the aggregate values and might even differ from those by orders of magnitude. So if we "artificially" set flows to zero, e. g., as a reaction to a failure in the system, the local values of the corresponding nodes may change arbitrarily within the range of the initial data and therefore it may happen that the computation of the aggregate is basically restarted. Concluding, the basic problem in handling permanent failures in the PF algorithm is, that—in contrast to the local values, which converge to the target aggregate—the flow variables converge to arbitrary values which depend on the execution of the PF algorithm. Thus, if we set a flow variable to zero, we have no control over the impact of this action on the local approximates of the target aggregate.

This is illustrated in Figure 4 which shows example runs of the PF algorithm where a single permanent link failure was introduced at different points of time. As we can see, in terms of local accuracy the computation is basically restarted from the beginning no matter how late the failure occurs.

To overcome this major issue, we will develop an improved algorithm in Section III where also the flow variables are forced to converge to the target aggregate.

III. Improving Fault Tolerance and Accuracy

In Section II we identified major issues in the failure handling and accuracy of the PF algorithm. As we have seen, the key issue—responsible for all the problems shown in Section II—is that the flow variables do not converge to the target aggregate but to some value which depends on the execution of the PF algorithm. In the following, we introduce the *push-cancel-flow algorithm (PCF)* which operates similarly to the PF algorithm with the additional benefit that all flow variables converge to the target aggregate. Moreover, the fault tolerance properties of the PF algorithm are preserved.

A. The Push-Cancel-Flow Algorithm (PCF)

To preserve the fault tolerance properties of the PF algorithm we have to stick with the concept of exchanging flows, i. e., whatever data the nodes exchange has to be exchanged via flows. This restriction guarantees that the resulting algorithm has the same fault tolerance properties as the PF algorithm since its fault tolerance is solely based on utilizing the flow concept. In general, the ideas behind the PCF algorithm can be implemented in various ways. The resulting algorithms differ slightly in terms of computational efficiency and fault tolerance. In the following, we will describe step by step how to derive the computationally most efficient variant of the PCF algorithm (see Figure 5) from the PF algorithm (see Figure 1).

First of all, a local flow vector φ_i is introduced to store the sum over all local flows of a node i. Therefore, $\varphi_i = \sum_{j\in\mathcal{N}_i} f_{i,j}$ and the current local data is computed simply as $(s_i, w_i) - \varphi_i$. The basic idea behind the PCF algorithm is that if flow conservation holds, i. e., $f_{i,j} = -f_{j,i}$, the flows should be added to φ_i and φ_j and afterwards they should be set to zero. Proceeding this way, we would ensure that the flow variables converge to the target aggregate because they only get updated by the current estimates of neighboring nodes (cf. Figure 1, line 10 and Figure 5, line 31) which converge to the target aggregate per definition.

To solve this problem the PCF algorithm uses two flow variables $f_{i,j,1}$ and $f_{i,j,2}$ instead of a single flow variable $f_{i,j}$. At each point of time these two flows serve a different purpose. We speak about an *active flow* for exchanging current data, i. e., for imitating the PF algorithm, and a *passive flow* where we want to achieve $f_{i,j} = -f_{j,i} = 0$. Hence, if both passive flows are zero, active and passive flows change their roles. This procedure is continuously repeated. Moreover, control variables $c_{i,j} \in \{1, 2\}$ and $r_{i,j} \in \mathbb{N}$ are introduced. The current active flow between i and j is stored in $c_{i,j}$ and $r_{i,j}$ counts how often active and passive flows changed their roles. We use the notation $c^c_{i,j}$ for denoting the complementary value of $c_{i,j}$ in the sense that $c^c_{i,j} = 2$ for $c_{i,j} = 1$ and vice versa.

The additional logic for achieving the desired convergence properties of the PCF algorithm only affects the handling

Input: Local initial data x_i and local weight w_i for each node i
Output: Estimate of global aggregate $(\sum_k x_k)/(\sum_k w_k)$

```
   ...initialization...
 1: v_i ← (x_i, w_i), φ_i ← (0, 0)
 2: for each j ∈ N_i do
 3:     {φ_i, f_{i,j,1}, f_{i,j,2}} ← (0, 0)
 4:     {c_{i,j}, r_{i,j}} ← 1
 5: end for
   ...on receive...
 6: for all received tuples ⟨f_{j,i,1}, f_{j,i,2}, c_{j,i}, r_{j,i}⟩ do
 7:     if c_{i,j} ≠ c_{j,i} and r_{i,j} = r_{j,i} then
 8:         c_{i,j} ← c_{j,i}
 9:     end if
10:     if c_{i,j} = c_{j,i} then
11:         φ_i ← φ_i − (f_{i,j,c_{i,j}} + f_{j,i,c_{i,j}})
12:         f_{i,j,c_{i,j}} ← −f_{j,i,c_{i,j}}
13:         if (f_{j,i,c^c_{i,j}} = −f_{i,j,c^c_{i,j}} and r_{i,j} = r_{j,i}) then
14:             f_{i,j,c^c_{i,j}} ← 0
15:             r_{i,j} ← r_{i,j} + 1
16:         else
17:             if (f_{j,i,c^c_{i,j}} = 0 and r_{i,j} + 1 = r_{j,i}) then
18:                 c_{i,j} ← c^c_{i,j}
19:                 f_{i,j,c^c_{i,j}} ← 0
20:                 r_{i,j} ← r_{i,j} + 1
21:             else
22:                 if r_{i,j} ≤ r_{j,i} then
23:                     φ_i ← φ_i − (f_{i,j,c^c_{i,j}} + f_{j,i,c^c_{i,j}})
24:                     f_{i,j,c^c_{i,j}} ← −f_{j,i,c^c_{i,j}}
25:                 end if
26:             end if
27:         end if
28:     end if
29: end for
   ...on send...
30: k ← choose a neighbor uniformly at random
31: f_{i,k,c_{i,k}} ← f_{i,k,c_{i,k}} + ((s_i, w_i) − φ_i)/2
32: φ_i ← φ_i + ((s_i, w_i) − φ_i)/2
33: Send ⟨f_{i,k,1}, f_{i,k,2}, c_{i,k}, r_{i,k}⟩ to k
```

Figure 5. The PCF algorithm for the local computation of a global aggregate. The current local estimate of a node is computed as $e_i(1)/e_i(2)$, with $e_i = v_i - \varphi_i$.

of received messages (lines 6–29) whereas the initialization part (lines 1–5) and the part for sending messages (lines 30–32) are completely analogous to the PF algorithm. For a received message we distinguish two cases: In the first case (lines 7–9), the receiving node gets informed about a change of active and passive flow and updates its local values accordingly. In the second case (lines 10-29), first the active flow is included into φ_i analogously to the PF algorithm and afterwards the passive flows are processed. In the handling of the passive flows we have to distinguish the following three cases: (i) lines 13–16: Flow conservation was achieved and the process of setting the passive flows to zero is started, (ii) lines 17–21: The process of setting the passive flows

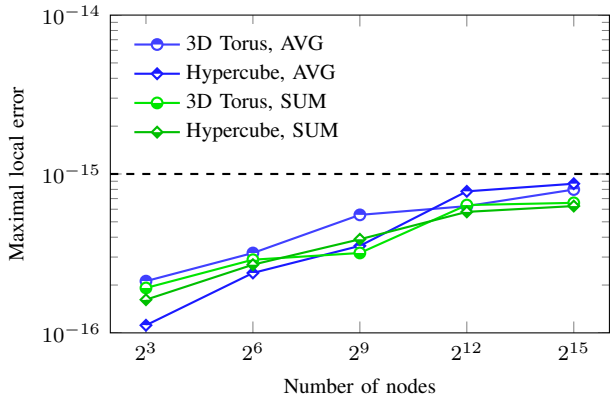

Figure 6. The accuracy experiments carried out for the PF algorithm in Section II-B (see Figure 3) are repeated for the PCF algorithm. In all cases the target accuracy of $\epsilon = 10^{-15}$ is reached in strong contrast to the results achieved by the PF algorithm (cf. Figure 1).

to zero was successful and the active/passive flows change their roles and (*iii*) lines 22–25: Flow conservation does not hold for the passive flows and the passive flows are treated like the active flows.

Strictly speaking, the algorithm shown in Figure 5 cannot tolerate bit flips. In order to preserve the full range of fault tolerance capabilities of the PF algorithm also in the PCF algorithm, updated flows cannot be included directly into the overall sum of flows φ_i (cf. lines 11, 23 and 32). Hence, in this case, the sum of flows gets only updated prior to the "cancellation" of a flow (cf. line 19). Therefore, active and passive flows have to be included into the computation of the local estimate to provide correct results. In contrast to the summation in the PF algorithm this summation is much more robust due to the different behavior of the flow variables as described above.

B. Correctness and Convergence

By the fact that for the active flows nothing else than the PF algorithm is executed it is clear that the PCF algorithm and the PF algorithm are equivalent and produce (theoretically) identical results. Hence, the correctness and convergence of the PCF algorithm follow immediately from the respective properties of the PF algorithm. Moreover, since the aggregate data is only exchanged via flows, the PCF algorithm directly inherits—depending on the concrete implementation—all or almost all fault tolerance properties from the PF algorithm.

Summarizing, the PCF algorithm is equivalent to the PF algorithm and provides the additional property that all flow variables converge to the target aggregate. This is achieved by overlapping the actual computation (active flows) with incorporating conserved flows in a locally stored overall sum of flows (passive flows).

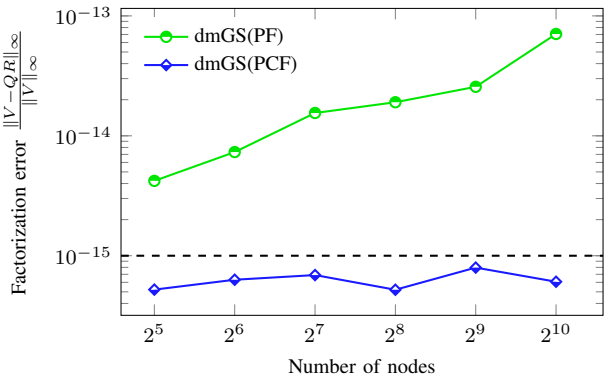

Figure 8. Factorization errors of the dmGS(PF) and the dmGS(PCF) on a failure-free hypercube network.

C. Experimental Results

To illustrate the superior fault tolerance and accuracy properties of the PCF algorithm over the PF algorithm we repeat the experiments carried out in Sections II-B and II-C replacing the PF algorithm with the PCF algorithm.

First, in Figure 6 we study how the techniques introduced in the PCF algorithm influence the resulting accuracy. We see that in all cases the chosen target accuracy of $\epsilon = 10^{-15}$ is reached. Moreover, the error increase with growing network size is much slower than for the PF algorithm (cf. Figure 3).

The second problem which we discussed in Section II-C is an inefficient failure handling of the PF algorithm which leads to major fall-backs in convergence. In Figure 7 the superior fault tolerance properties of the PCF algorithm are depicted by a direct comparison to the results achieved by the PF algorithm (shown in light colors and dashed). For the experiments in Figs. 4 and 7 we initially used exactly the same random seed, i. e., the simulated random communication schedules are the same. Hence, since the PF algorithm and PCF algorithm behave identically for the same communication schedules and initial data (if no failures occur), we see no difference between the two algorithms until the first failure occurs. After the failure handling, we see that the PCF algorithm tolerates the failure without any fall-back in the convergence in strong contrast to the PF algorithm.

IV. A Higher Level Application

We investigate a distributed QR factorization algorithm as a prototypical example of how a distributed reduction algorithm can be used for designing distributed higher level matrix computations. A fully distributed QR factorization (*dmGS*) based on the push-sum algorithm [6] has been introduced in [11]. The dmGS originates from the modified Gram-Schmidt orthogonalization method [24], but summations in each computation of a vector norm or a dot product are replaced by a call of a distributed reduction (originally

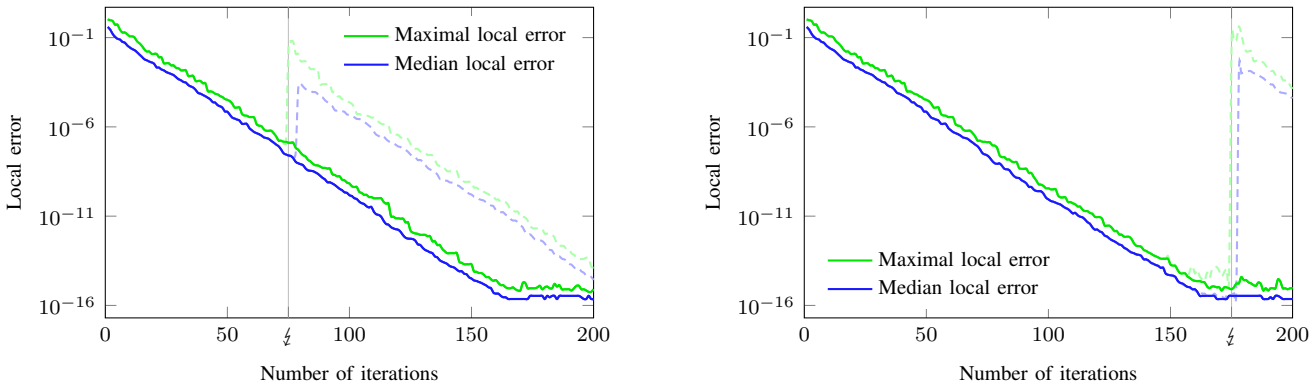

Figure 7. The fault tolerance experiments carried out for the PF algorithm in Section II-C (see Figure 4) are repeated for the PCF algorithm. From the point on where the results differ, i. e., the point where the failure handling takes place, the results for the PF algorithm (depicted in Figure 4) are indicated in light colors.

the push-sum algorithm). Except for the distributed summation via reductions, all computations are done locally in the nodes.

However, beyond *reported* message loss or *known* unavailability of communication links, the push-sum algorithm cannot tolerate any types of failures. In contrast to push-sum, the PF algorithm and the PCF algorithm are much more fault-tolerant in terms of tolerating link and node failures as well as soft errors like bit flips or message loss. Since dmGS uses distributed reductions as a *black box*, it is beneficial to use such resilient algorithms as a building block for designing more robust distributed matrix algorithms. However, it needs to be explored whether the distributed algorithms based on fault tolerant reduction algorithms also preserve known (numerical) properties of the standard algorithms without fault tolerance capabilities. In the following we denote with dmGS(PF) and dmGS(PCF) the dmGS algorithm based on the PF algorithm and PCF algorithm, respectively.

We compared dmGS(PF) and dmGS(PCF) in terms of a relative factorization error $\|V - QR\|_\infty / \|V\|_\infty$ when factorizing random matrices $V \in \mathbb{R}^{n \times 16}$, $n \geq m$ distributed over a hypercube with N nodes. In our experiments, we chose $n = N$, but dmGS works for all $n \geq N$ (see [11]). Since dmGS does not depend on a specific reduction algorithm, we expect that the improvements of the PCF algorithm over the PF algorithm shown in Section III directly translate to dmGS, e. g., dmGS(PCF) should deliver more accurate results than dmGS(PF). Each reduction was given a prescribed target accuracy $\epsilon = 10^{-15}$ in all simulations, and a maximal number of iterations per reduction was set to terminate reductions which did not achieve this target accuracy.

The factorization error of dmGS(PF) and dmGS(PCF) is depicted in Figure 8. All results shown are averaged over 50 runs to account for the variations due the randomized communication schedules of the gossip-based reduction algorithms. We see that dmGS(PF) does not reach the accuracy $\epsilon = 10^{-15}$ prescribed as target accuracy for the reductions. Moreover, with increasing number of nodes the factorization error increases. In contrast, dmGS(PCF) always achieves the target accuracy $\epsilon = 10^{-15}$ set for individual reductions and the factorization error does not increase with the number of nodes. Similarly, the improved accuracy of the PCF algorithm translates to the orthogonalization error of dmGS.

V. Conclusions

State-of-the-art distributed reduction algorithms like the PF algorithm scale equally well with the number of nodes as their deterministic parallel counterparts. Moreover, distributed approaches show an exciting potential for fault tolerance at the algorithmic level and can in principle tolerate link and node failures as well as soft errors like bit flips or message loss.

We analyzed the PF algorithm and identified significant shortcomings in terms of numerical accuracy as well as in terms of performance degradations caused by its failure handling mechanism. Moreover, we presented the novel PCF algorithm which resolves the discovered problems of the PF algorithm. We performed numerical experiments to illustrate the superior properties of the PCF algorithm. While the PCF algorithm overcomes all discovered shortcomings of the PF algorithm, it preserves the attractive properties of the PF algorithm in terms of flexibility, scalability and fault tolerance. Finally, we illustrated for the case of a fully distributed QR factorization algorithm that the improvements achieved with the PCF algorithm can be directly exploited in the context of higher level distributed matrix operations.

Acknowledgment

The research was funded by the Austrian Science Fund (FWF) under project number S10608.

REFERENCES

[1] T. Adachi, N. Shida, K. Miura, S. Sumimoto, A. Uno, M. Kurokawa, F. Shoji, and M. Yokokawa, "The design of ultra scalable MPI collective communication on the K computer," *Computer Science - Research and Development*, pp. 1–9, 2012.

[2] W. N. Gansterer, G. Niederbrucker, H. Straková, and S. Schulze Grotthoff, "Robust Distributed Orthogonalization Based on Randomized Aggregation," in *Proceedings of the Second Workshop on Scalable Algorithms for Large-Scale Systems*, ScalA '11, 2011, pp. 7–10.

[3] ——, "Scalable and Fault Tolerant Orthogonalization Based on Randomized Distributed Data Aggregation," 2012, (*under review*). [Online]. Available: http://eprints.cs.univie.ac.at/3258/

[4] R. Thakur and W. Gropp, "Improving the performance of collective operations in MPICH," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer Verlag, 2003, pp. 257–267.

[5] S. Boyd, A. Ghosh, B. Prabhakar, and D. Shah, "Randomized Gossip Algorithms," *IEEE/ACM Trans. Netw.*, vol. 14, pp. 2508–2530, 2006.

[6] D. Kempe, A. Dobra, and J. Gehrke, "Gossip-Based Computation of Aggregate Information," in *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*, 2003, pp. 482–491.

[7] P. Jesus, C. Baquero, and P. S. Almeida, "Fault-Tolerant Aggregation by Flow Updating," in *Proceedings of the 9th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems*, DAIS '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 73–86.

[8] I. Eyal, I. Keidar, and R. Rom, "LiMoSense – Live Monitoring in Dynamic Sensor Networks," in *7th Int'l Symp. on Algorithms for Sensor Systems, Wireless Ad Hoc Networks and Autonomous Mobile Entities (ALGOSENSORS'11)*, 2011.

[9] H. Straková and W. N. Gansterer, "A Distributed Eigensolver for Loosely Coupled Networks," in *Proceedings of the 21^{st} Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, PDP 2013, 2013, (*to appear*).

[10] L. S. Blackford et al., "An updated set of basic linear algebra subprograms (BLAS)," *ACM Trans. Math. Softw.* 28:2, pp. 135–151, 2002.

[11] H. Straková, W. N. Gansterer, and T. Zemen, "Distributed QR Factorization Based on Randomized Algorithms," in *PPAM (1)*, Lecture Notes in Computer Science, R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, Eds., vol. 7203. Springer, 2011, pp. 235–244.

[12] J. von Neumann, "Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components," *Automata Studies*, pp. 43–98, 1956.

[13] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell, "Detection and correction of silent data corruption for large-scale high-performance computing," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12. IEEE Computer Society Press, 2012, pp. 78:1–78:12.

[14] J. Plank, K. Li, and M. Puening, "Diskless checkpointing," *IEEE Trans. Parallel and Dist. Syst.*, vol. 9, no. 10, pp. 972–986, 1998.

[15] K.-H. Huang and J. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations," *IEEE Transactions on Computers*, vol. C-33, no. 6, pp. 518–528, 1984.

[16] C. Anfinson and F. Luk, "A Linear Algebraic Model of Algorithmic-Based Fault Tolerance," in *Proceedings of the International Conference on Systolic Arrays, 1988*, 1988, pp. 483–493.

[17] F. Luk and H. Park, "Fault-tolerant matrix triangularizations on systolic arrays," *Computers, IEEE Transactions on*, vol. 37, no. 11, pp. 1434–1438, 1988.

[18] Z. Chen and J. J. Dongarra, "Algorithm-Based Fault Tolerance for Fail-Stop Failures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 12, pp. 1628–1641, 2008.

[19] Z. Chen, "Algorithm-Based Recovery for Iterative Methods without Checkpointing," in *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, HPDC '11, 2011, pp. 73–84.

[20] 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 *Proceedings of the Second Workshop on Scalable Algorithms for Large-Scale Systems*, ScalA '11, 2011, pp. 25–28.

[21] P. Du, P. Luszczek, S. Tomov, and J. Dongarra, "Soft error resilient QR factorization for hybrid system with GPGPU," in *Proceedings of the Second Workshop on Scalable Algorithms for Large-Scale Systems*, ScalA '11, 2011, pp. 11–14.

[22] P. Du, P. Luszczek, and J. Dongarra, "High Performance Dense Linear System Solver with Resilience to Multiple Soft Errors," *Procedia Computer Science*, vol. 9, pp. 216–225, 2012.

[23] G. Niederbrucker and W. N. Gansterer, "Robust Gossip-Based Aggregation: A Practical Point of View," in *Proceedings of the 2013 Meeting on Algorithm Engineering & Experiments (ALENEX)*, 2013, (*to appear*).

[24] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 3rd ed. The Johns Hopkins University Press, 1996.