Resilient Gossip-Inspired All-Reduce Algorithms for High Performance Computing – Potential, Limitations and Open Questions

Marc Casas[1] and Wilfried N. Gansterer[2] and Elias Wimmer[2,3]

Abstract

We investigate the usefulness of gossip-based reduction algorithms in a High Performance Computing (HPC) context. We compare them to state-of-the-art deterministic parallel reduction algorithms in terms of fault tolerance and resilience against silent data corruption (SDC) as well as in terms of performance and scalability. New gossip-based reduction algorithms are proposed which significantly improve the state-of-the-art in terms of resilience against SDC. Moreover, a new *gossip-inspired* reduction algorithm is proposed which promises a much more competitive runtime performance in an HPC context than classical gossip-based algorithms, in particular for low accuracy requirements.

Keywords

all-to-all reduction, all-reduce, gossip algorithm, fault tolerance, bit-flip, silent data corruption, recursive doubling, push-flow algorithm

1 Introduction

The end of Dennard scaling (Dennard et al. (1974)) around 2005 created the basis for the current many-core chips prevalence in computing systems. In the context of High Performance Computing (HPC), distributed memory parallel clusters composed of shared-memory many-core sockets became prominent, which forced HPC systems to exploit both distributed and shared memory parallelism. The Message Passing Interface (MPI) standard is the most wide spread parallel programming paradigm for distributed memory systems while shared memory machines are typically programmed with OpenMP (OpenMP Architecture Review Board (2013)) or POSIX threads (Nichols et al. (1996)). Hybrid MPI+OpenMP codes are common since they exploit the different parallel layers of HPC infrastructures.

Since the technological limits of Dennard scaling were hit, performance enhancements of HPC systems have been mainly achieved by increasing their number of components, which necessarily requires the software stack to deploy increasing concurrency levels. This trend towards more hardware components and more parallelism will certainly keep driving the design of future HPC systems, as well as the constraints in terms of power consumption, which will bring more heterogeneity inside the node in terms of GPU accelerators or vector units.

Such increases in terms of hardware components will make the behavior of future HPC systems more unpredictable, either in terms of hard and soft faults, which brings issues in terms of software correctness and performance, or in terms of low performing hardware and software components, which will bring significant performance slowdowns. Current HPC applications, implemented by combining MPI and OpenMP, are very vulnerable to unexpected hardware errors and shutdowns. The software stack must evolve towards a more flexible and asynchronous approach, either when managing on-chip parallelism or when dealing with off-chip concurrency.

In particular, *collective communications*, which are defined as communication operations that involve a group of software components, are a specific kind of communication patterns handled by the MPI programming model that are

[1]Barcelona Supercomputing Center (BSC), Spain
[2]University of Vienna, Faculty of Computer Science, Research Group Theory and Applications of Algorithms, Vienna, Austria
[3]TU Wien, Faculty of Informatics, Research Group Parallel Computing, Vienna, Austria

Corresponding author:
Wilfried N. Gansterer, University of Vienna, Faculty of Computer Science, Währinger Straße 29, 1090 Vienna, Austria

Email: wilfried.gansterer@univie.ac.at

particularly challenging on future systems. Global reductions and barrier synchronizations are two very common types of collective communications. An increase by two or three more orders of magnitude in terms of concurrency compared to today's high-end computer systems will make current collective communication approaches too rigid. Given the way global MPI reductions are implemented today, the impact of having just a single network switch performing abnormally slow, OS noise, or of a single bit-flip within a network packet can strongly undermine the global performance or correctness of a large operation involving hundreds of thousands or MPI processes (Ferreira et al. (2008)).

The main drawback of current MPI collective implementations is that they are based on static (and deterministic) approaches, such as recursive doubling (Thakur and Gropp (2003)), and that they have no chance to react or to dynamically adapt to sudden and unexpected issues. In this paper we investigate alternative algorithmic strategies for collective communications based on gossip algorithms.

Gossip protocols (sometimes called *epidemic* protocols) are decentralized protocols originally intended for loosely coupled distributed systems, such as P2P or ad-hoc networks. Based on periodic information exchange between nodes (or processes), they do not rely on any assumptions about specific hardware connections, and thus they work on arbitrary (connected) topologies. Moreover, they do not assume reliable communication, and the selection of communication partners is randomized in some form. Every node operates only based on local knowledge of its neighborhood, and conceptually no synchronization across the network is assumed. To some extent, gossip-based algorithms can be *self-healing*, i.e., they can potentially deliver a correct result in the presence of failures or faults *without* the need to explicitly detect or report a fault.

Gossip protocols have mainly been used for two different purposes: gossip-based *dissemination* algorithms use gossiping for spreading information (*rumor-mongering*), and gossip-based *aggregation* algorithms compute a system-wide aggregate—an *all-to-all reduction* (e.g., sum, average, min, max)—by sampling information from randomly chosen nodes (or processes) and combining the values. Gossip-based all-to-all reduction algorithms have the capacity to deal with unexpected behavior of the system for three main reasons: (i) Due to their randomized communication schedules they are more flexible and adaptive than classical deterministic algorithms, (ii) they have very strong resilience properties, and (iii) their accuracy can be adapted dynamically to the

requirements with a decrease of the computational cost for lower accuracy.

While the utilization of gossip-based dissemination algorithms in an HPC context has been proposed recently (Katti et al. (2015); Herault et al. (2015)) for resilience purposes, the focus of this paper is on investigating and evaluating the potential of gossip-based *all-to-all reduction algorithms* in an HPC context, taking into account resilience as well as performance aspects. This is a very relevant problem since all-to-all reduction (all-reduce) operations appear at the heart of many important HPC applications.

The ability to trade accuracy for performance is a very attractive aspect of gossip-based all-reduce algorithms. Reduced precision computations have been studied in many HPC application areas as one potential approach for improving computational efficiency, e.g., by Váňa et al. (2017) in weather forecasting. The interaction between arithmetic precision and resulting accuracy has also received interest beyond classical HPC, e.g., in machine learning (see, e.g., Lesser et al. (2011)). In particular, for contemporary deep learning applications, which pose important challenges in high performance computing aspects (such as efficient parallelization), the capability of reducing the computational cost of global all-reduce operations for lower accuracy requirements is of high interest. On the one hand, global reduction operations appear in parallel variants of the stochastic gradient descent algorithm, which is a pivotal component of standard methods for training deep neural networks (Dean et al. (2012); Bottou (2010)). This has already motivated first attempts at exploiting a gossip strategy in an asynchronous stochastic gradient descent method (Jin et al. (2016)). However, several open questions remain, and thus most strategies used in practice still rely on a central server for aggregating gradient information. On the other hand, the effects (and benefits) of reduced precision in data representation and computation in the context of training of and inference with deep neural networks have been investigated extensively. In particular, it has been demonstrated that very low precision is often sufficient in the context of deep learning (see, e.g., Gupta et al. (2015) and references therein).

Not only reduced precision, but also resilience against silent data corruption (SDC) at the algorithmic level is a relevant aspect for deep learning applications. For large models, the training phase is often done on consumer GPUs where mechanisms for detecting and correcting SDC are usually weaker than on today's HPC hardware.

1.1 Related Work

The Message Passing Interface (MPI) standard is the most widespread distributed memory parallel programming model (MPI Forum (2012)). Extensive research efforts have been undertaken in the last years to improve the performance of MPI codes. For example, Hoefler et al. (2007) have extended the non-blocking communication paradigm, which the MPI standard defines for point-to-point communications, to collective communications, showing some potential to improve performance under certain conditions. In terms of increasing MPI fault tolerance, Herault et al. (2015) have developed an agreement algorithm to address permanent crashes. This solution is built on top of the User-Level Failure Mitigation (ULFM) proposal by Bland et al. (2013). Other approaches also rely on finding consensus for handling faults in the context of MPI (e. g., Katti et al. (2015)).

Most existing work on fault tolerant gossip-based aggregation algorithms exclusively focuses on message loss or hardware failures (see Eyal et al. (2014); Jesus et al. (2015); Gansterer et al. (2013)). Only Niederbrucker and Gansterer (2013) have done a limited study on the impact of certain types of bit-flips. In this paper, we extend existing algorithms and thoroughly investigate to which extent the fault tolerance potential of gossip-based aggregation can be exploited when SDC is considered.

While gossip-based aggregation algorithms have shown tolerance to loss of messages or hardware failures, they can also dynamically adapt the accuracy they provide, which brings proportional cost reductions. Other approaches that deploy inexact or irregular computations for achieving performance gains while keeping the algorithms' accuracy within acceptable margins have been considered. For example, the loop perforation approach by Sidiroglou-Douskos et al. (2011) aims to trade accuracy for performance by transforming loops to execute only a subset of their iterations. Such a subset is composed of the most relevant iterations in terms of numerical accuracy.

1.2 Objectives and Contributions

The main motivation for this paper is to summarize the first steps towards a better understanding of the potential and limitations of gossip-based all-to-all reduction algorithms in the context of High Performance Computing.

For this purpose, we start with summarizing the upcoming challenges for HPC on future extreme-scale systems and motivating the investigation of gossip-based all-to-all reduction algorithms for such systems in Section 2. In Section 3, we provide a compact review of representative all-to-all reduction methods, both classical deterministic algorithms as well as state-of-the-art gossip-based algorithms. In Section 4 we focus on resilience aspects. In particular, we compare deterministic algorithms and asynchronous gossip-based approaches in terms of their resilience against silent data corruption. Moreover, we propose new asynchronous gossip-based algorithms which exhibit improved resilience against silent data corruption. Section 5 is dedicated to a discussion of performance aspects of gossip-based all-to-all reduction methods in an HPC context. To the best of our knowledge, this has not been investigated systematically so far. Since gossip-based algorithms are iterative in nature, we first discuss various factors influencing the *number of iterations* required. This is followed by identifying some fundamental open questions with respect to the *cost per iteration*. Then we summarize some experiences with a first naive MPI implementation of a gossip-based reduction algorithm on a few thousand nodes of a HPC cluster. Based on the insight gained from this implementation, we devise an alternative round-based gossip-*inspired* implementation strategy which is better suited for an HPC context. With experiments on more than 100 000 processes on a high-end supercomputer and simulations up to one billion processes we then investigate how the number of iterations required scales for low accuracy requirements.

2 Challenges in HPC

Future HPC machines are expected to use 7 nm process technology, several orders of magnitude more concurrency than contemporary petascale systems and to operate under very restricted power budgets. All these factors, either coming from the hardware or the software side, will certainly make HPC machines more vulnerable to faults (Seifert et al. (2012)). To mitigate errors, several software- and hardware-based techniques can be leveraged (Cappello et al. (2014)), at the cost of decreasing performance and adding extra latencies to detect and correct errors. Also, the chances of a hardware component suddenly delivering unexpectedly low performance just adds more uncertainty to the whole scenario. Without any countermeasures, the reduced reliability and increased irregular behavior of future HPC systems will largely impact performance, making the usage of such large infrastructures a very difficult task.

2.1 The Hardware Perspective

Historically, the performance improvements reported by the Top500 List (2018) were due to CPU clock frequency

increases, among other factors. However, due to technological constraints, the CPU clock frequency has stagnated since 2007. As a consequence, performance enhancements can only be achieved by increasing the number of sockets of supercomputing infrastructures. Therefore, the expected total number of components of an exascale system will be much larger than those of a petascale system. Indeed, exascale systems are expected to have at least hundreds of millions of CPU cores and to handle concurrency levels of similar orders of magnitude.

Individual components of such systems are not expected to be more resilient. It is true that redundancy levels and error checking mechanisms at the hardware level are increasing, but these increases are aimed at dealing with the higher complexity of hardware components, not at making components more resilient. The main reason behind this trend is that hardware vendors calibrate the resilience levels of their products to satisfy the needs of their main markets, and the HPC sector is not one of them. Consequently, the fault rate of exascale systems is expected to grow at the same rate as the total number of components does.

2.2 The Software Perspective

The perspective from the software side is not much better (Ashby et al. (2010)). Indeed, current HPC software stacks are not ready to deal with unexpected and faulty behavior. They are typically very rigid, and they have not been run and tested on faulty platforms. Moreover, significant portions of them contain legacy code that is hardly adaptable to the new context where HPC is going with systems which will have to continuously handle misbehaving hardware components.

In the current situation, once a fault is detected it is either propagated through the different layers of the software stack or some fault tolerance approach is executed to correct it. Since an increase in the frequency and the diversity of faults is expected, overheads due to fault correction using current techniques may become intolerable. The solution is to provide the software stack with new approaches to take action in response to detected errors in a way that the overhead associated with the correction is as low as possible.

Besides the resilience problem, which may lead to execution crashes, incorrect results or significantly reduced application performance due to correction overhead, there is another important factor: performance degradation in parts of the hardware. The chances of particular components of the machine performing well below their expected delivery rate may strongly impact the overall performance of large scale runs. It is thus required to extend the software stack

with mechanisms and paradigms which are able to hide extra latencies, either due to faults or to low performing hardware components. More asynchronous and adaptive approaches are required to tolerate unexpected latencies.

Global reduction operations are among the ones that suffer most from faults or partial performance degradation in the hardware. First, because the entire parallel execution may be impacted by an error or a performance slowdown during a global reduction, since this operation involves all parallel threads. Second, because the rigidness of their current implementations in today's software stacks neither allows the design of resilient software solutions which take action against faults, nor other kinds of solutions to react to performance degradations in hardware components.

3 Parallel All-to-all Reduction

In this section we survey the main available approaches for performing global all-to-all reduction (all-reduce) operations. In particular, we review in detail the classical recursive doubling approach in Section 3.1, and two gossip-based approaches in Section 3.2: the push-sum algorithm and its more robust advancement, the push-flow algorithm.

3.1 Recursive Doubling

Recursive doubling is a standard method for carrying out a parallel all-reduce operation. It is also one of the algorithms used by current MPI implementations. Depending on the message size and on the number of processes involved in the all-to-all reduction, MPI uses different algorithms (Thakur and Gropp (2003); Chan et al. (2007)). Recursive doubling is the optimal all-reduce algorithm in terms of the number of messages needed when operating with small message sizes.

In this paper, we consider the following all-to-all reduction operation over N parallel processes:

$$y = \bigotimes_{k=0:N-1} x_k.$$

The initial data $x_0, x_1, \ldots, x_{N-1}$ (for simplicity, we consider scalar initial data) is distributed over the N parallel processes. The result y of the all-reduce operation (the *aggregate*) is available at all processes. In general, "$\otimes$" denotes an element-wise associative and commutative binary operation. In this paper we focus on summation and averaging.

Recursive doubling computes the all-to-all reduction on $N = 2^d$ processes (numbered $0, 1, \ldots, N-1$ for simplifying the presentation) in $d = \log_2 N$ steps (Line 2 in Alg. 1). In every step k, each process i pairwise exchanges

---
Algorithm 1 Recursive Doubling $[N = 2^d, \text{process } i]$

Input: $x_i \in \mathbb{R}$
Output: $y_i = \bigotimes\limits_{k=0:N-1} x_k \in \mathbb{R}$

1:   $y_i \leftarrow x_i$
2:   **for** $k \leftarrow 0, \ldots, d-1$ **do**
3:      $j \leftarrow i \oplus 2^k$
4:      **send** y_i to process j
5:      **wait** to receive y_j from process j
6:      $y_i \leftarrow y_i \otimes y_j$
7: **end for**

---

its current value with a process j whose process id differs exactly in the kth bit from i (Lines 3-5 in Alg. 1, where "$\oplus$" denotes an XOR operation). This leads to a pairwise exchange of values between processes whose "distance" (in terms of process rank) doubles in every step. After every exchange, the local variable y_i is overwritten by the result of applying the operator $\otimes$ locally to y_i and y_j (Line 6 in Alg. 1). After d steps, every process has received information from every other process and the result of the reduction is available in the local variable y_i at every process i.

3.1.1 Complexity of parallel all-to-all reductions. A lower bound in terms of *number of messages* required for a parallel all-reduce operation can be derived by the fact that each process has some information needed by every other process. Per step each process can send at most one message, thus information can only be doubled in each step. Consequently, the number of messages required is at least $\lceil \log_2 N \rceil$.

Computing the reduction on a single process requires $(N-1)$ operations. Consequently, when perfectly distributed across N processes, the *computational cost* is at least $\frac{N-1}{N}$ times the cost for a single operation $\otimes$.

The cost of transferring the data can be derived from the lower bound on the computational cost. For completing the computation at least $\frac{N-1}{N}$ data items must be sent and $\frac{N-1}{N}$ items must be received by each process. This leads to a minimal cost of $2\frac{N-1}{N}$ times the cost for transferring a single data item (Thakur and Gropp (2003); Chan et al. (2007)).

3.1.2 Resilience of recursive doubling. Classical parallel reduction approaches like recursive doubling are based on a pre-determined and well synchronized sequence of data movements. Therefore, they rely on the system software and the underlying communication stack to provide failure free data transfer. If an unreported message loss occurs, the behavior of algorithms like recursive doubling is undefined. In case of a SDC during the reduction process, it is very likely that the computation will succeed, but since the error may be propagated to many processes, many or all local results may be wrong (see Section 4.4).

3.2 Gossip-Based All-to-All Reduction

Gossip-based all-to-all reduction algorithms (Eyal et al. (2014); Jesus et al. (2015); Gansterer et al. (2013)) are decentralized iterative algorithms with randomized communication schedules. They do not rely on any assumptions about specific hardware connections, thus they work on arbitrary (connected) topologies. Every process operates only based on local knowledge of its neighborhood and conceptually no synchronization across the network is assumed. In each iteration (round), every process i chooses a *random* communication partner from its neighborhood $\mathcal{N}_i$. This type of algorithm was originally intended for loosely coupled distributed systems, such as P2P or ad-hoc networks. In an HPC context, the neighborhood $\mathcal{N}_i$ would usually be defined by a suitable overlay network (the *communication topology*) which allows for optimizing performance.

3.2.1 Push-Sum. The *Push-Sum* algorithm (PS) by Kempe et al. (2003) is a gossip algorithm for summing or averaging N values $x_0^0, x_1^0, \ldots, x_{N-1}^0$ distributed across N processes. Each process i starts with an initial value-weight pair (x_i^0, w_i^0) and at time t computes a local approximation y_i^t of the weighted sum $\sum_{k=0:N-1} x_k^0 / \sum_{k=0:N-1} w_k^0$. Note that this weighted sum is the *average* over all x_i^0 if $w_i^0 = 1 \quad \forall i$, and it is the *sum* of all x_i^0 if $\sum_i w_i^0 = 1$. In order to simplify the notation we will omit the time superscript (0 or t) in the following unless it is needed to avoid ambiguity.

Algorithm 2 Push-Sum (PS) $[\text{process } i]$

Input: $(x_i^0, w_i^0) \in \mathbb{R}, \ \varepsilon \in \mathbb{R}$
Output: $y_i \approx \sum_{k=0:N-1} x_k^0 / \sum_{k=0:N-1} w_k^0 \in \mathbb{R}$

1: **while** not ε-accurate **do**
2:      $j \leftarrow$ choose a neighbor $\in \mathcal{N}_i$ uniformly at random
3:      $(x_i, w_i) \leftarrow (x_i, w_i)/2$
4:      **send** (x_i, w_i) to process j
5:      **for each** received pair (x_j, w_j) **do**
6:         $(x_i, w_i) \leftarrow (x_i, w_i) + (x_j, w_j)$
7:      **end for**
8: **end while**
9: $y_i \leftarrow x_i/w_i$

In each (local) iteration every process i halves its local value-weight pair (x_i, w_i) and sends it to a neighbor $j \in \mathcal{N}_i$ chosen uniformly at random (Lines 2-4 in Algorithm 2). Afterwards all received value-weight pairs (x_j, w_j) are added to the local value-weight pair (x_i, w_i) (Lines 5-7 in Algorithm 2). This process is repeated until convergence. Note that every process i can send messages independently of all other processes. Thus, PS (like other gossip-based

algorithms) conceptually does not require synchronization of sending and receiving messages among processes.

The local approximation y_i^t of the aggregate is computed by dividing x_i^t by w_i^t (Line 9 in Algorithm 2). We refer to ε as "target accuracy" and say that PS computed an ε-accurate estimate of the true aggregate if the maximum relative error over all processes is bounded by ε.

3.2.2 Improving resilience — Push-Flow. For correctness of the PS algorithm *mass conservation* needs to be ensured (Kempe et al. (2003)), i.e., $\sum_i (x_i^t, w_i^t) = \sum_i (x_i^0, w_i^0)$ needs to hold at all times t. If this relation is not preserved (*mass loss*), the iterative procedure will still converge to the same value at every process ("consensus"), but *not* to the correct aggregate of the initial values.

Note that mass conservation is a *global* property which is usually violated by faults such as (silent) data corruption or message loss. Although the flexible communication schedule of PS easily allows for tolerating *reported* faults, any kind of *unreported* fault (such as silent data corruption) violates mass conservation. Thus, the PS algorithm cannot guarantee correct all-to-all reductions in the presence of silent data corruption.

Several ideas have been pursued to overcome this problem, however, so far with a focus on unreported message loss (Eyal et al. (2011); Hadjicostis et al. (2012); Jesus et al. (2009, 2010)) or on node failures and dynamic topology changes (Eyal et al. (2014); Jesus et al. (2015)). We focus here on the *Push-Flow* algorithm (PF), proposed and investigated by Niederbrucker et al. (2012); Gansterer et al. (2013); Niederbrucker and Gansterer (2013) and shown in Algorithm 3. PF can be viewed as an advancement of PS for improving resilience by utilizing a flow concept borrowed from graph theoretical flow networks .

Algorithm 3 Push-Flow (PF)   [process i]

Input: $(x_i, w_i) \in \mathbb{R}^2$
$\forall\, k\ \in \mathcal{N}_i :\ \boldsymbol{f_{i,k}} \leftarrow (0,0)\ \in \mathbb{R}^2,\ \varepsilon \in \mathbb{R}$
Output: $y_i \approx \sum_{k=0:N-1} x_k \,/\, \sum_{k=0:N-1} w_k \in \mathbb{R}$
  1: **while** not ε-accurate **do**
  2:     $\boldsymbol{e_i} \leftarrow (x_i, w_i) + \sum_{k \in \mathcal{N}_i} \boldsymbol{f_{i,k}}\ \in \mathbb{R}^2$
  3:     $j \leftarrow$ choose a neighbor $\in \mathcal{N}_i$ uniformly at random
  4:     $\boldsymbol{f_{i,j}} \leftarrow \boldsymbol{f_{i,j}} - \boldsymbol{e_j}/2$
  5:     **send** $\boldsymbol{f_{i,j}}$ to process j
  6:     **for each** received flow $\boldsymbol{f_{j,i}}$ from process j **do**
  7:         $\boldsymbol{f_{i,j}} \leftarrow -\boldsymbol{f_{j,i}}$
  8:     **end for**
  9: **end while**
10: $y_i \leftarrow \boldsymbol{e_i}(1)/\boldsymbol{e_i}(2)$

For every neighboring process $j \in \mathcal{N}_i$, process i holds a *flow variable* $\boldsymbol{f_{i,j}} \in \mathbb{R}^2$ representing the data flow (the

"mass" sent) to process j. Whereas in PS the sent mass is immediately subtracted from the current value-weight pair, in PF sent mass is only stored in the flow variables. Hence, the initial value-weight pair is never changed. The current local value-weight pair $\boldsymbol{e_i} \in \mathbb{R}^2$ of process i can be computed as $\boldsymbol{e_i} = (x_i, w_i) + \sum_{k \in \mathcal{N}_i} \boldsymbol{f_{i,k}}$ (Line 2 in Algorithm 3).

Like in PS, in each iteration every process i chooses some process $j \in \mathcal{N}_i$ to which it sends half of its current value-weight pair $\boldsymbol{e_i}$. In contrast to PS, only the flow variables are updated and sent (Lines 4-5 in Algorithm 3). All received flows are negated and *overwrite* the corresponding flow variable (Lines 6-8 in Algorithm 3). The local approximation y_i of the aggregate is computed by dividing the value component $\boldsymbol{e_i}(1)$ of the current value-weight pair $\boldsymbol{e_i}$ by the weight component $\boldsymbol{e_i}(2)$ (Line 10 in Algorithm 2).

In the Push-Flow algorithm, *flow conservation* is ensured, i.e., $\sum_i \sum_{j \in \mathcal{N}_i} \boldsymbol{f_{i,j}} = (0,0)$ holds at all times if no faults occur. Note that flow conservation *implies* mass conservation. However, in contrast to mass conservation, flow conservation is a *local* property and can be maintained or restored more easily. In particular, every successful transmission of an uncorrupted flow variable automatically re-establishes flow conservation between two processes, even if it has been violated previously by some fault. The PF algorithm thus very naturally recovers from loss of messages at the next successful fault-free communication without even detecting it explicitly as shown by Gansterer et al. (2013). *Detected* or *reported* broken system components, e. g., permanently failed links or processes, can easily be tolerated by setting the corresponding flow variables to zero, since this algorithmically excludes the failed components from the computation (Gansterer et al. (2013)). In *exact* arithmetic, PF can also recover from SDC in the flow variables and/or messages (with some additional cost). However, we will illustrate in Section 4 that further advancements are needed for tolerating SDC in practice (in *floating-point* arithmetic).

3.2.3 Convergence and complexity. A time complexity of $\mathcal{O}(\log N + \log \varepsilon^{-1})$ rounds for approximating the target aggregate with a relative error at most ε at each process has been proved for Push-Sum on fully connected networks by Kempe et al. (2003). Later, Boyd et al. (2006) extended this result to a time complexity of $\mathcal{O}(\log N/(1 - \lambda_2))$ rounds, where λ_2 is the second largest eigenvalue of the communication matrix. In practice, networks allow for fast gossip-based all-to-all reductions in $\mathcal{O}(\log N)$ rounds either by their physical topology or by some overlay network combined with efficient routing techniques. As discussed

by Gansterer et al. (2013), these time complexity results for the PS algorithm also hold for the PF algorithm. Note that for aggregating scalar data, which is the focus of our paper, asymptotically this is the same time complexity as the one of deterministic recursive doubling (cf. Section 3.1.1)

4 Improving Resilience Against SDC

This section motivates the need for algorithms to deal with SDC that takes place while data is being transferred. It also discusses the limitations of the approaches described in Section 3 and presents an enhancement of the PF algorithm which is able to effectively deal with SDC in the communication process. In Section 4.5 we also briefly outline how to recover from SDC in the initial data.

4.1 Motivation

Typically, MPI communications are done via a network interconnect. When moving data through such devices, there are many scenarios where silent data corruption may occur, as, e.g., discussed by the Commonwealth Scientific and Industrial Research Organization (CSIRO) (2014). For example, electromagnetic interferences from electric cables may interfere electrical signals being transmitted by others. This problem is especially important in computing infrastructures with many cables packed together, like data centers or supercomputers. Also, the arrival time of the different electrical signals may not be exactly the same due to unavoidable differences in cable lenghts. Normally, interconnecting networks can tolerate differences of up to 50 nanoseconds per 100 meters of cable, but late arrivals beyond this treshold may produce silent data corruptions. All these scenarios may end up producing bit-flips in the data being transmitted. So far, concepts for guaranteeing reliable communication are concentrated at the transport layer and layers below in the OSI model. Our approach is to investigate the capability of algorithmic fault tolerance at the application layer in order to complement and extend the range of available strategies for improving resilience. It remains to be investigated whether a combination of concepts at *several* layers can yield even more improvements in relevant metrics.

Although in theory (exact arithmetic) the concept of flows allows PF to recover from any bit-flip in the flow variables and messages, Niederbrucker and Gansterer (2013) and Niederbrucker et al. (2012) illustrated that in practice the limited precision of floating-point arithmetic causes mass loss. This is due to round-off errors in the computation of the estimate when bits with high significance are affected. In the following we present two novel algorithms which overcome

this problem and are therefore capable of handling SDC in messages as well as in flow variables.

4.2 Push-Flow with Local Correction

The basic idea for handling the effects of bit-flips which cannot be tolerated by PF is to integrate explicit fault detection strategies into the PF algorithm. In our context, checksum approaches based on summing the value-weight pair are appropriate. Although more sophisticated methods for error detection exist, for our purposes it is essential to quantify the size of the detected error and to do an explicit correction only if the error is too large.

The novel gossip-based algorithm *Push-Flow with local correction* (*PFLC*, see Algorithm 4) extends the local data and each flow variable by a third component. In the initial local data this third component contains the sum of the value and the weight variable. In each flow variable this third component is the sum of the first two components. After computing the estimate e_i (Line 2 in Algorithm 4), every process checks for a data corruption by comparing the sum of the first two entries of the estimate with the third entry containing the checksum. If the difference is larger than a threshold τ, an error in the flow variables is detected (Line 3 in Algorithm 4). In this case, the process checks every flow variable for an error and repairs the local data by setting flow variables to zero when necessary (Lines 4 to 6 in Algorithm 4). Afterwards the current estimate e_i is recalculated using the new flow variables (Line 9 in Algorithm 4).

The sending part is identical to Push-Flow (Lines 11 to 13 in Algorithm 4). When process i receives a message from process j, it first verifies the checksum for the received flow $f_{j,i}$. Only if the checksum error is below the threshold τ, the negated received flow is set as the new local flow $f_{i,j}$ (Lines 15 to 17 in Algorithm 4), otherwise the received message (flow variable) is discarded. The local approximation y_i of the aggregate is computed like in PF (Line 20 in Algorithm 4).

Improved fault tolerance. In theory (exact arithmetic), PFLC has the same fault tolerance properties as PF. However, SDC in the most significant bits in floating-point arithmetic, which leads to large errors in magnitude, can only be tolerated by PFLC (and not by PF). This is due to the fact that by dropping received faulty flow variables and by resetting the corresponding local flow variables to zero, PFLC prevents the potentially excessive growth of a flow variable caused by some bit-flips and thus avoids the effects of the associated round-off errors. The next communication

---

Algorithm 4 PFLC   [process i]

Input: $(x_i, w_i, c_i) \in \mathbb{R}^3$, $c_i = x_i + w_i$
$\forall\, k \in \mathcal{N}_i: \boldsymbol{f_{i,k}} \leftarrow (0,0,0) \in \mathbb{R}^3$; $\varepsilon, \tau \in \mathbb{R}$
Output: $y_i \approx \sum_{j=0:N-1} x_j / \sum_{j=0:N-1} w_j \in \mathbb{R}$
1: **while** not ε-accurate **do**
2:      $\boldsymbol{e_i} \leftarrow (x_i, w_i, c_i) + \sum_{k \in \mathcal{N}_i} \boldsymbol{f_{i,k}} \in \mathbb{R}^3$
3:      **if** $|\boldsymbol{e_i}(1) + \boldsymbol{e_i}(2) - \boldsymbol{e_i}(3)| > \tau$ **then**
4:          **for each** $k \in \mathcal{N}_i$ **do**
5:              **if** $|\boldsymbol{f_{i,k}}(1) + \boldsymbol{f_{i,k}}(2) - \boldsymbol{f_{i,k}}(3)| > \tau$ **then**
6:                  $\boldsymbol{f_{i,k}} \leftarrow (0,0,0)$
7:              **end if**
8:          **end for**
9:          $\boldsymbol{e_i} \leftarrow (x_i, w_i, c_i) + \sum_{k \in \mathcal{N}_i} \boldsymbol{f_{i,k}}$
10:     **end if**
11:     $j \leftarrow$ choose a neighbor $\in \mathcal{N}_i$ uniformly at random
12:     $\boldsymbol{f_{i,j}} \leftarrow \boldsymbol{f_{i,j}} - \boldsymbol{e_j}/2$
13:     **send** $\boldsymbol{f_{i,j}}$ to process j
14:     **for each** received flow $\boldsymbol{f_{j,i}}$ from process j **do**
15:         **if** $|\boldsymbol{f_{j,i}}(1) + \boldsymbol{f_{j,i}}(2) - \boldsymbol{f_{j,i}}(3)| \le \tau$ **then**
16:             $\boldsymbol{f_{i,j}} \leftarrow -\boldsymbol{f_{j,i}}$
17:         **end if**
18:     **end for**
19: **end while**
20: $y_i \leftarrow \boldsymbol{e_i}(1)/\boldsymbol{e_i}(2)$

---

between two processes after resetting a corrupted flow variable will re-establish flow conservation between them.

Simply resetting a flow variable $\boldsymbol{f_{i,j}}$ to zero corresponds to a loss of the information exchange so far and thus may in general lead to a significant convergence delay. However, not every fault necessarily leads to such a strong setback in convergence. For seeing this, let us assume that only a single process i experiences one bit-flip in the flow variable $\boldsymbol{f_{i,j}}$ and thus resets it to zero. Depending on which process (i or j) initiates the next communication, we can distinguish two cases: If first i sends its flow variable $\boldsymbol{f_{i,j}}$ (which is zero) to j, then $\boldsymbol{f_{i,j}}$ overwrites $\boldsymbol{f_{j,i}}$, and a significant convergence delay is usually experienced. However, if j sends its flow $\boldsymbol{f_{j,i}}$ first, then $\boldsymbol{f_{i,j}}$ gets overwritten by $\boldsymbol{f_{j,i}}$, and the convergence delay is negligible.

4.3 Push-Flow with Cooperative Correction

As mentioned before, every reset of a flow variable can lead to a setback in convergence. However, due to the redundancy in the flow variables it is in many cases possible that a correct flow overwrites a neighbor's faulty one without causing any setback in convergence. Such considerations motivate the idea underlying *Push-Flow with cooperative correction* (*PFCC*, see Algorithm 5): design an algorithm which resets flow variables to zero *only* when it is absolutely necessary. This can be achieved by leaving a faulty flow uncorrected, but sending it to the corresponding neighbor as

soon as possible. Once the neighbor detects that the received flow variable is faulty, it can send back its correct flow variable, thus correcting the fault by overwriting the faulty flow variable. Only if *both* flow variables are corrupted, the processes involved need to reset their flow variables to zero. This way many SDC errors can be corrected without a convergence delay.

Algorithm 5 PFCC   [process i]

Input: $(x_i, w_i, c_i) \in \mathbb{R}^3$, $c_i = x_i + w_i$
$\forall\, k \in \mathcal{N}_i: \boldsymbol{f_{i,k}} \leftarrow (0,0,0) \in \mathbb{R}^3$; $\varepsilon, \tau \in \mathbb{R}$
Output: $y_i \approx \sum_{j=0:N-1} x_j / \sum_{j=0:N-1} w_j \in \mathbb{R}$
1: **while** not ε-accurate **do**
2:      $\boldsymbol{e_i} \leftarrow (x_i, w_i, c_i) + \sum_{k \in \mathcal{N}_i} \boldsymbol{f_{i,k}} \in \mathbb{R}^3$
3:      **if** $|\boldsymbol{e_i}(1) + \boldsymbol{e_i}(2) - \boldsymbol{e_i}(3)| > \tau$ **then**
4:          **for each** $k \in \mathcal{N}_i$ **do**
5:              **if** $|\boldsymbol{f_{i,k}}(1) + \boldsymbol{f_{i,k}}(2) - \boldsymbol{f_{i,k}}(3)| > \tau$ **then**
6:                  **send** $\boldsymbol{f_{i,j}}$ to k
7:              **end if**
8:          **end for**
9:      **end if**
10:     $j \leftarrow$ choose a neighbor $\in \mathcal{N}_i$ uniformly at random
11:     $\boldsymbol{f_{i,j}} \leftarrow \boldsymbol{f_{i,j}} - \boldsymbol{e_j}/2$
12:     **send** $\boldsymbol{f_{i,j}}$ to j
13:     **for each** received flow $\boldsymbol{f_{j,i}}$ from process j **do**
14:         **if** $|\boldsymbol{f_{j,i}}(1) + \boldsymbol{f_{j,i}}(2) - \boldsymbol{f_{j,i}}(3)| \le \tau$ **then**
15:             $\boldsymbol{f_{i,j}} \leftarrow -\boldsymbol{f_{j,i}}$
16:         **else**
17:             **if** $|\boldsymbol{f_{i,j}}(1) + \boldsymbol{f_{i,j}}(2) - \boldsymbol{f_{i,j}}(3)| > \tau$ **then**
18:                 $\boldsymbol{f_{i,j}} \leftarrow (0,0,0)$
19:             **end if**
20:             **send** $\boldsymbol{f_{i,j}}$ to j
21:         **end if**
22:     **end for**
23: **end while**
24: $y_i \leftarrow \boldsymbol{e_i}(1)/\boldsymbol{e_i}(2)$

PFCC uses the same checksums as PFLC, it differs only in the reaction to the detection of corrupted flow variables. When process i detects an error in the estimate $\boldsymbol{e_i}$, it checks every flow variable and sends the damaged flows to the respective neighbors (cf. Lines 3 to 9 in Algorithm 5). When a process receives a faulty flow variable, it additionally checks if its local flow variable is also corrupted (cf. Lines 14 and 17 in Algorithm 5). Only if both flow variables (received and local) are corrupted, the local flow variable is set to zero (cf. Line 18 in Algorithm 5). For all received corrupted flows, a process sends its current local flow variable to the sender (cf. Line 20 in Algorithm 5).

In summary, PFCC has the same fault-tolerance properties as PFLC, but in many situations it will recover much faster from SDC than PFLC, as we will illustrate experimentally in Section 4.4.

4.4   Fault Tolerance in Practice

The following experiments illustrate the effect of SDC on the various all-to-all reduction algorithms. We investigate bit-flips in local data structures and their influence on the achieved accuracy as well as on the communication overhead. In contrast to all results in the literature so far, our resilience results in this section are for *asynchronous* implementations, i.e., implementations that do not require synchronization operations. In Section 5 we will discuss performance aspects of gossip-based all-to-all reduction algorithms, in particular their runtime overhead compared to less resilient recursive doubling. There, we will see that when implementing gossip-based all-reduce in MPI, the standard parallelization paradigm for HPC, *synchronized* round-based implementations have performance advantages over fully asynchronous implementations.

4.4.1   Experimental setup.
We simulated asynchronous versions of the algorithms discussed so far with a discrete event simulator. All algorithms were run with the same sequence of random numbers. Thus, in the fault-free case all PF variants show exactly the same results. The error $e_{\rm rm}$ shown in the figures represents the maximum relative error over all processes i:

$$\text{max relative error } e_{\rm rm} := \arg\max_i \frac{|y_i - y_{\rm true}|}{|y_{\rm true}|}, \quad (1)$$

where $y_{\rm true}$ is the correct aggregate.

All experiments in Sections 4.4.2 and 4.4.3 were performed for $N = 32$ processes in a hypercube communication topology. The algorithms were terminated once the local relative error of all processes was less than 10^{-14}. Alternatively, the algorithms were stopped once the first process has sent out 500 messages. The threshold τ for the checksums in PFLC and PFCC was set to 10^{-11}. In Section 4.4.4 we discuss how the choice of τ influences the performance of PFLC and PFCC.

4.4.2   Influence of SDC on convergence behavior.
The different gossip-based algorithms recover differently from SDC. As mentioned earlier, due to the effects of floating-point arithmetic, PF cannot handle any SDC which causes a huge error in a flow variable.

Fig. 1 compares the convergence history of the three PF variants in case of a single bit-flip at time $t_{\rm bit\text{-}flip}$. In the failure-free case (no bit-flip), all three algorithms show exactly the same convergence history. For each algorithm the $56^{\rm th}$ bit in a randomly chosen flow variable (the same flow variable in all algorithms) was flipped at time $t_{\rm bit\text{-}flip}$. We see that standard PF first falls back in the convergence process and later
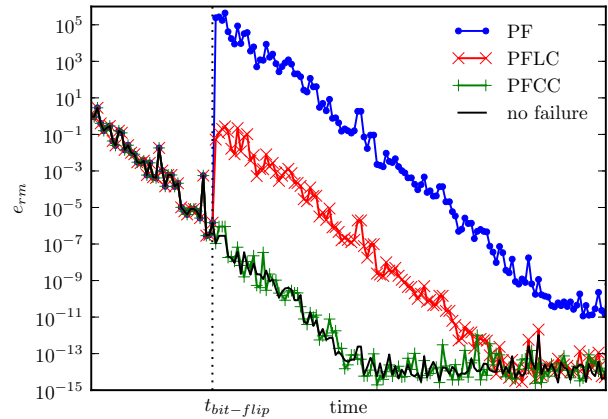

Figure 1. Convergence behavior of different Push-Flow algorithms in case of a single bit-flip

converges to an aggregate which has a relative error of about 10^{-11}, since the round-off error caused by the bit-flip when computing the aggregate is too large. In contrast, both PFLC and PFCC converge much faster, and the aggregate which they produce has a relative error of only about 10^{-14}. PFLC immediately resets every corrupted flow variable to zero and thus also experiences some fall-back in the convergence process. However, in contrast to standard PF, in the worst case this fall-back is only to the initial estimate, and, as the fault is never propagated, it always converges to a more accurate aggregate. PFCC requests a message from the neighbor associated with a detected corrupted flow to overwrite this corrupted flow variable. Thus, PFCC recovers much faster from the fault and experiences hardly any slow-down compared to the failure-free case.

4.4.3   Influence of the position of the fault.
The next experiment illustrates the influence of a bit-flip in a value-weight pair for PS and in a flow variable for all PF variants in all possible positions of an IEEE 754 double precision floating-point variable. For each algorithm a bit is flipped after the first process sent 150 messages.

Fig. 2 shows the maximum of the $e_{\rm rm}$ from Equation (1) over 100 different runs. The first interesting observation is that PS can only tolerate SDC in a few of the least significant bits. Note that in Fig. 2 relative errors greater than 1 are beyond the range of the y-axis. Second, it is interesting to observe that PS and recursive doubling have very similar resilience properties. Third, PF can recover from all bit-flips in the sign and mantissa bits as the resulting error is relatively small, but for bit-flips in the exponent of a floating-point number, the maximum relative error of PF increases rapidly. Fourth, our novel algorithms PFLC and PFCC fully recover from bit-flips at *any* position in the local flow data structures.
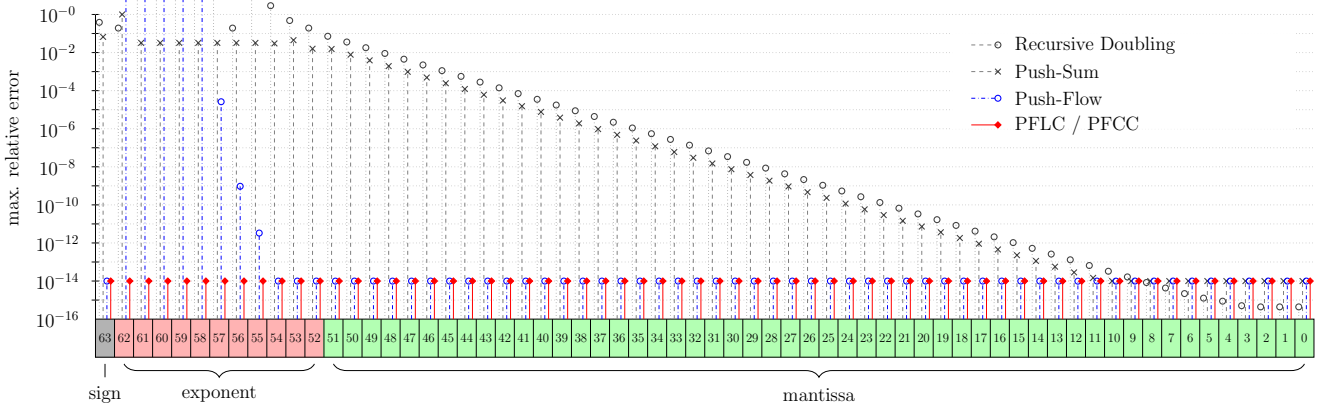
Figure 2. Maximum relative error over 100 individual runs depending on the position of the bit-flip (simulation for $N = 32$, $\varepsilon = 10^{-14}$)

In Fig. 3, we illustrate the "prize" of this increased fault tolerance in terms of its communication overhead. More specifically, we compare the mean (over 100 runs) of all sent messages per process of PS without any bit-flips with the mean of all sent messages of PF, PFLC and PFCC with a single bit-flip on different positions in a flow variable. The overhead of all PF variants compared to PS (slightly more than 50%) is due to a special situation which occurred relatively often in our discrete event simulator: If two neighboring processes send to each other simultaneously, these messages can overwrite each other's flow variable. This has the same effect as if one of the messages is lost and thus causes an overhead each time it happens. In Section 6 we show how to overcome this issue and thereby improve the performance of the PF variants.

We note that up to the 30$^{\text{th}}$ bit there is no difference in the communication overhead of the three resilient algorithms, as the error caused by the flipped bit is below the threshold τ. Beginning with the 31$^{\text{st}}$ bit, PFCC persistently requires fewer messages than PF and PFLC due to its more efficient recovery mechanism. PF and PFLC require roughly the same number of messages up to the 49$^{\text{th}}$ bit. From this point on, PFLC corrects errors and sets corrupted flow variables to zero. Starting with bit number 55, PF does not always converge to an accurate aggregate any more (cf. Fig. 1), and thus the average communication overhead also includes runs which were terminated because one process reached the maximum message count of 500. Only PFCC has an almost constant message count independently of the position of the bit-flip.

4.4.4 Influence of the checksum threshold τ.
For both algorithms PFLC and PFCC the communication overhead in case of bit-flips strongly depends on the checksum threshold τ. In Figures 4 and 5 we illustrate the influence of τ on the

communication overhead of PFLC and PFCC in case of bit-flips by depicting the median and 1.5 times the interquartile range (IQR) of the required number of iterations over all runs. It becomes clear that τ has to be chosen differently for both algorithms, but also for different process counts. In general one can deduce from the two figures, that for PFLC $\tau > 0.1$ is beneficial, whereas for PFCC τ should be chosen very small.

4.4.5 Many bit-flips.
Next we investigated the influence of the number of bit-flips on the overhead of both algorithms. We ran both algorithms 500 times with a fault rate of 0.0001 (before sending a message, some flow variable is corrupted with probability 0.0001) and recorded the number of iterations needed for different numbers of bit-flips. For a single bit-flip, PFLC and PFCC show similar communication cost (see Fig. 6), but starting from two bit-flips per reduction, the overhead of PFLC increases strongly whereas PFCC shows no significant difference in the number of iterations. While it is unclear whether high bit-flip rates appear in practice, for low bit-flip rates both algorithms are comparable.

4.5 SDC Beyond Flow Variables

First, we note that all insights from Figs. 2 and 3 also apply to bit-flips in messages, since such faults have an impact once the contents of the message is written into the flow variable of the receiving process.
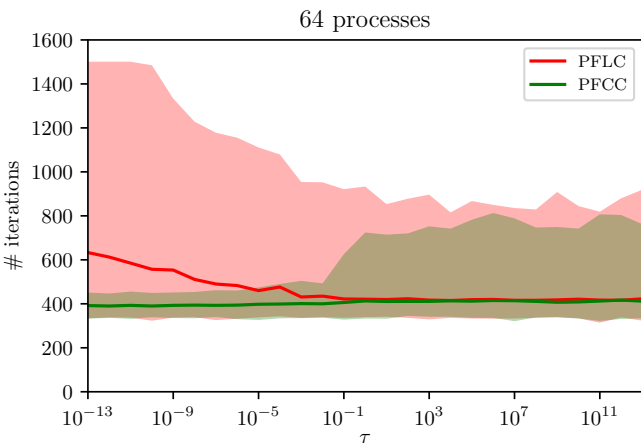
When considering SDC beyond flow variables and messages, things get more complicated. Nevertheless, to some extent it is even possible to repair corrupted initial value-weight pairs. For example, if the current estimate e_i and all flow variables are correct, one can recompute (x_i, w_i, c_i) by subtracting the sum over all flows from the current estimate $(x_i, w_i, c_i) = e_i - \sum_k f_{i,k}$. Moreover, we
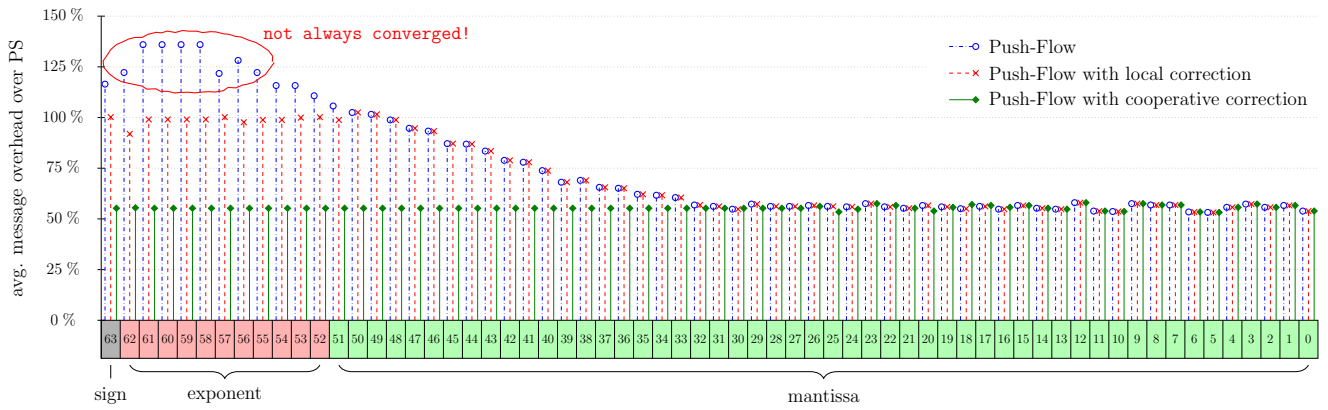
Figure 3. Mean message overhead for PF, PFLC and PFCC compared to Push-Sum (simulation for $N = 32$, $\varepsilon = 10^{-14}$, $\tau_{\mathsf{PFLC}} = 0.5$ and $\tau_{\mathsf{PFCC}} = 10^{-10}$)
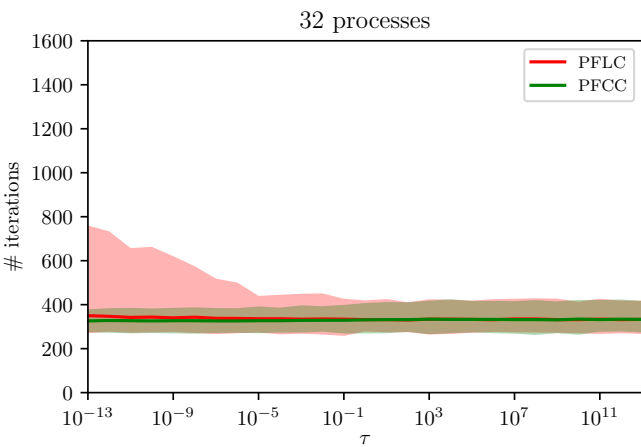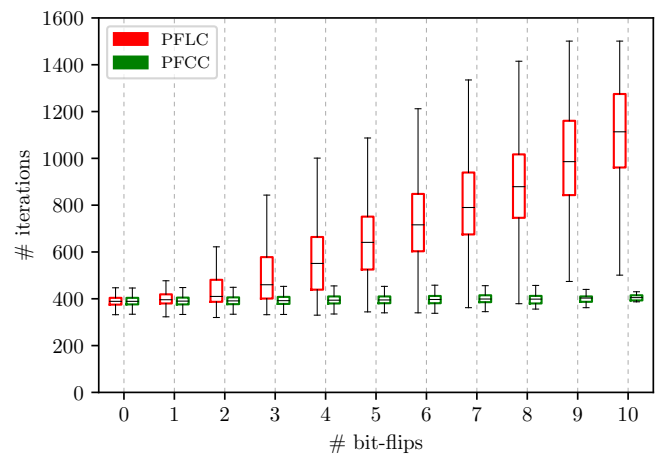

Figure 4. Influence of τ on the number of iterations until convergence for PFLC and PFCC in the presence of several bit-flips (line shows the median and the colored area shows $1.5 \cdot$ IQR, simulation for $N = 32$, $\varepsilon = 10^{-14}$, bit-flip rate 0.0001)

Figure 6. Number of iterations until convergence for PFLC and PFCC for different bit-flip counts (simulation for $N = 64$, $\varepsilon = 10^{-14}$, $\tau_{\mathsf{PFLC}} = 0.5$, $\tau_{\mathsf{PFCC}} = 10^{-10}$, whiskers at $1.5 \cdot$ IQR)

5 Runtime Performance and Scalability

Beyond the strong resilience properties demonstrated in Section 4 it is so far unclear whether gossip-based algorithms are efficient enough in practice to be of interest in an HPC context. Obviously, one has to expect that the greatly increased flexibility and robustness of gossip-based algorithms come with some overhead in terms of execution time compared to routines optimized for classical parallel systems. A thorough quantification of the cost of the improved resilience achieved by gossip-based all-reduce algorithms as discussed in Section 4 is very complex, in particular for asynchronous implementations. In this section, we investigate the runtime performance of gossip-based reduction algorithms. In such an investigation, it is important to distinguish between the number of iterations (cf. Section 5.1) and the cost per iteration (cf. Section 5.2).

The overall execution time to a great extent also depends on an efficient implementation (e. g., to which extent the very low synchronization requirements of PFCC can be exploited). How to implement gossip-based algorithms

Figure 5. Influence of τ on the number of iterations until convergence for PFLC and PFCC in the presence of several bit-flips (line shows the median and the colored area shows $1.5 \cdot$ IQR, simulation for $N = 64$, $\varepsilon = 10^{-14}$, bit-flip rate 0.0001)

are currently investigating whether remote flow variables can be exploited to recover from SDC in the initial value-weight pair. However, this is still work in progress.

efficiently for an HPC context is so far a widely open question. In Sections 5.3 we summarize first experiences when implementing gossip-based reduction algorithms in today's dominant parallel programming model for HPC, MPI, and we present a *gossip-inspired* approach which is much better suited for an efficient implementation in MPI than classical gossip-based algorithms such as Algorithm 2.
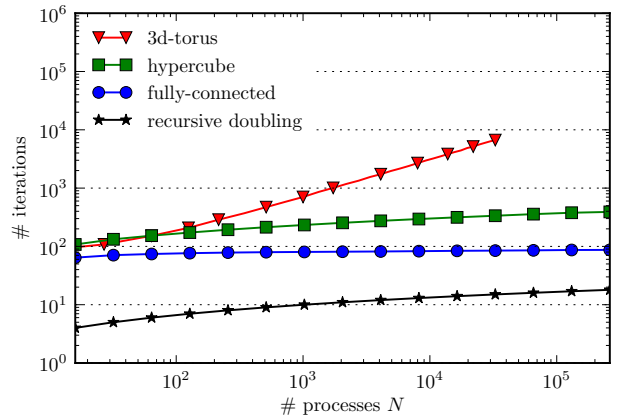
5.1   Number of Iterations

First we note that although recursive doubling is not an iterative algorithm, it proceeds in $\log_2 N$ steps (cf. Algorithm 1). For simplicity, we do not distinguish between a "step" (in the case of recursive doubling) and an "iteration" (in the case of gossip-based reduction algorithms) in the following, but we use the term "iteration" (or "round") for both. Having said this, all algorithms considered in this paper *asymptotically* require $\mathcal{O}(\log N)$ iterations for achieving *full* accuracy (subject to the limitations of floating-point accuracy) on fully-connected topologies.

As indicated, this asymptotic complexity is influenced by two important factors: the underlying communication topology, i.e., the virtual topology superimposed on the physical topology of the computer system, which defines the neighborhood of each node, and—for gossip-based approaches—the desired accuracy of the result in terms of the acceptable maximum relative error (1). Taking these factors into account, the theoretical runtime complexity of PS is given by

$$\mathcal{O}\left(\frac{\log N + \log \varepsilon^{-1}}{1 - \lambda_2}\right) \qquad (2)$$

iterations for achieving an accuracy of ε (Boyd et al. (2006)). Here, $1 - \lambda_2$ denotes the spectral gap of the communication matrix of PS, whose largest eigenvalue by design is always one. The better connected the network, the larger the spectral gap. Thus, the connectivity of the communication network has a strong influence on the number of iterations needed by gossip-based algorithms. The spectral gap of a fully connected network is independent of N, the spectral gap of a hypercube decreases logarithmically with N, whereas the second largest eigenvalue for torus topologies or trees grows much faster with N.

For deterministic reduction algorithms, the communication topology does not influence the number of iterations, but the cost per iteration, which we will discuss in Section 5.2. Moreover, for these algorithms the accuracy of the result usually does not significantly influence the number of iterations. In general, full accuracy will be achieved after $\log_2 N$ iterations. Thus, Expression (2) indicates that gossip-based

Figure 7. Number of iterations until convergence for Push-Sum with different communication topologies (simulation in Matlab, initial data uniformly random $\in [0, 1]$, $e_{\mathrm{rm}} = 10^{-14}$)

approaches can be expected to be most competitive in terms of the number of iterations if the communication topology is well-connected (such as fully-connected or hypercube topologies) and if a low accuracy approximation of the reduction operation is sufficient.

Simulations of synchronized, round-based Push-Sum (see Fig. 7) support these theoretical considerations and show that communication topologies with a small spectral gap need a much larger number of iterations than fully-connected or hypercube toplogies. These simulation results also indicate a constant overhead in terms of the number of iterations for Push-Sum over recursive doubling, independently of the communication topology and the number of nodes or processes.

However, three aspects are worth being pointed out when comparing gossip-based algorithms with recursive doubling: First, due to their iterative nature, gossip-based algorithms allow for achieving a reduced target accuracy at a proportionally reduced cost (cf. expression (2)). This aspect, which can be attractive in certain applications, will be discussed in more detail in Section 6.2. Second, the convergence speed of gossip-based algorithms can be improved using various acceleration strategies, e.g., by adapting ideas presented by Dimakis et al. (2008) or by Janecek and Gansterer (2016). Third, due to their localized communication patterns, gossip-based algorithms may potentially have advantages in terms of the *cost per iteration* on extreme-scale systems, as we will discuss in Section 5.2.

5.2   Cost per Iteration

In both types of algorithms considered in this paper—deterministic parallel reduction algorithms as well as gossip-based reduction algorithms—each process involved sends one message per iteration. However, the distance which such a message has to travel (number of hops) and therefore the communication cost per iteration may differ significantly.

For deterministic parallel reduction algorithms, the communication cost per iteration depends on the physical topology of the computer system. More specifically, the maximum number of hops between any two communicating processes can be as large as the diameter of the physical topology.

In contrast, the communication cost per iteration of a gossip-based algorithm can be made *independent* of the physical topology by restricting all the communication to *single-hop* communication between processes on physically neighboring nodes. However, this extreme case may lead to a relatively large number of iterations for physical topologies with a small spectral gap (as discussed in Section 5.1). Thus, it is usually advisable to superimpose a well-connected virtual communication topology on the physical topology for defining the neighborhood of each process. Although this increases the communication cost per iteration of a gossip-based algorithm by allowing for multi-hop communication, it will keep the total number of iterations low if the communication topology has a large spectral gap.

Consequently, in contrast to deterministic parallel reduction algorithms, gossip-based algorithms support a trade-off between number of iterations and cost per iteration via the choice of the communication topology. Whether this capability can be exploited in terms of the overall execution time, especially for extreme-scale systems, strongly depends on how the communication cost between two nodes grows with the physical distance between these two nodes.

Assuming that the communication cost increases strictly monotonically with the physical distance between two communicating nodes, one would have to solve the following graph theoretical problem: Given a physical topology $\mathfrak{P}$, what is the communication topology $\mathfrak{C}$ defined by the set of nodes of $\mathfrak{P}$ and edges connecting these nodes such that the total number of hops accumulated over all communication operations until convergence of a gossip-based algorithm using this communication topology $\mathfrak{C}$ is minimized? Alternatively, one could consider a slightly different problem: Given a physical topology $\mathfrak{P}$ and a maximum number of hops $h_{\max}$, what is the communication topology $\mathfrak{C}$ defined by the set of nodes of $\mathfrak{P}$ and edges connecting these nodes such that (i) the diameter of $\mathfrak{C}$ is at most $h_{\max}$, and (ii) the spectral gap of $\mathfrak{C}$ is maximized? We are currently not aware of general solutions of such problems, and thus it seems still an open problem whether and under which conditions the localized communication properties of gossip-based reduction algorithms could lead to performance advantages for extreme-scale systems.

5.3   Implementing Gossip All-Reduce in MPI

Since the previous theoretical considerations and sequential simulations do not allow for precise predictions of execution times on HPC-Systems, we implemented Push-Sum using MPI and ran a first set of experiments on the Vienna Scientific Cluster VSC-2. This cluster consists of 1314 nodes, each of which holds two AMD Opteron 6132HE processors with eight cores each and has 32 GB of main memory. The nodes are connected through Infiniband QDR using a fat tree topology.

Our first Push-Sum implementation based on MPI one-sided communication showed disappointing runtime performance. Thus we decided to implement Push-Sum based on MPI point-to-point communication, which subsequently also allowed for extending the code to Push-Flow, PFLC and PFCC. However, the experimental results shown in Fig. 8 only partly reflect the runtime behavior to be expected from the theoretical considerations in Sections 5.1 and 5.2. Whereas the scaling behavior on hypercube and 3d-
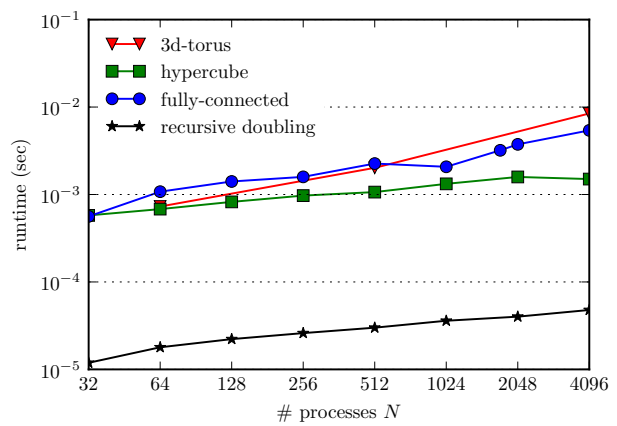
Figure 8. Runtime on VSC-2 for Push-Sum with different communication topologies until convergence (asynchronous MPI implementation, initial data uniformly random $\in [0, 1]$, $\varepsilon = 10^{-14}$)

torus communication topologies is roughly as anticipated, the scaling behavior of this Push-Sum implementation on a fully-connected communication topology was much worse than expected. The reason is that imposing a fully connected communication topology on top of the physical fat-tree

network topology of the VSC-2 causes a lot of network congestion.

These first experiments indicated that a naive MPI-based implementation of Push-Sum for HPC systems can by no means compete in terms of runtime performance with state-of-the-art all-reduce algorithms such as recursive doubling.

6 A Gossip-Inspired Approach

In order to advance the state-of-the-art in this question, we carefully investigate modifications of the original gossip concept which are better suited for the HPC context. More specifically, we develop and implement a gossip-*inspired* all-reduce scheme which we call *HPS* (*high performance push-sum*). It is specifically designed to match the MPI programming model in order to better understand the performance potential of approaches based on local communication. To implement a more competitive gossip-inspired reduction in MPI, we consider a scheme which better matches the MPI programming paradigm than the original asynchronous concept of Algorithm 2.

The key ideas are the following: First, we consider a *synchronized* (global iteration-based) variant, where each MPI process receives exactly one message per iteration. This strategy, where we do not allow iteration $k + 1$ to start before all N communications of iteration k have finished, reduces the communication complexity of the algorithm significantly. In a more general scheme complicated message queues and termination strategies for asynchronous point-to-point communications are needed which limit computational efficiency.

Second, we globally prescribe the sequence of local communications for all processes *a priori*. In other words, the communication pattern in each iteration is defined by a random permutation of the integers $0, 1, \ldots, N - 1$. This restriction to a single communication partner per iteration avoids the need to manage (up to) $N - 1$ active receive operations per MPI process, which would arise in the general setting of Algorithm 2 where the number of communication partners of a process per iteration can potentially be as large as $N - 1$.

The HPS approach has an additional advantage, which is not MPI-specific: The permutations that define the communication patterns of each iteration can be precomputed and stored in the memory of the computing device where each process runs. This eliminates the influence on the parallel execution caused by repeated random number generation.

6.1 Improving Resilience Against SDC

The improvement in resilience which leads from PS to PFLC as described in Section 4.2 involves only modifications in local computations and in book-keeping, but does not change any communication-related aspects. Therefore, it is straightforward to extend the ideas underlying HPS to PFLC. Doing so yields the parallel reduction algorithm *HPFLC* (*high performance push-flow with local correction*) which combines the resilience properties of PFLC with the improved performance of HPS. Moreover, we can solve the problem caused by simultaneous message exchange in Push-Flow-based algorithms which we identified in Section 4.4.3 by generating random permutation *cycles* instead of simple random permutations for each iteration.

Note that extending the ideas underlying HPS to PFCC is *not* straightforward, since the communication patterns of PFCC differ from those of PS in the event of a fault. We plan to investigate this aspect in future work.
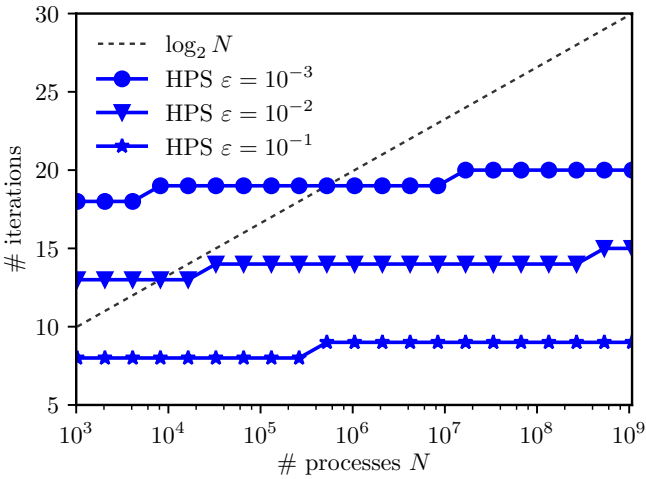
6.2 Accuracy vs. Number of Iterations

Before we show first results of an MPI-based implementation of the HPS algorithm in Section 6.3, we first convince ourselves that HPS exhibits a similar convergence behavior as the resilient gossip-based reduction algorithms proposed in Section 4.
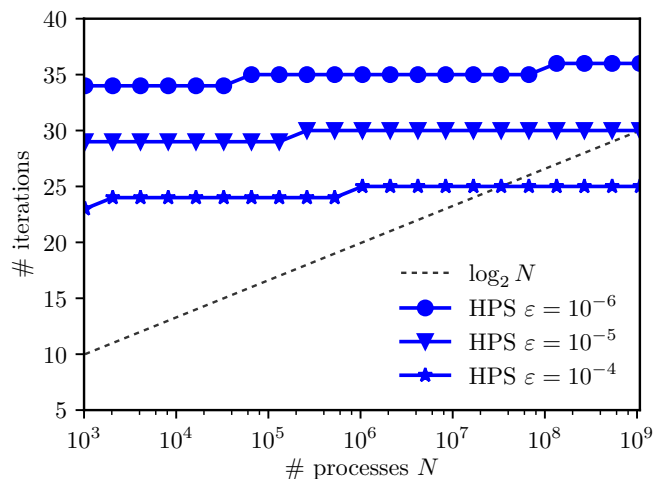
Figure 9. Number of iterations of HPS for higher accuracy levels (median over 100 runs, base topology is fully connected)

Figs. 9 and 10 show the number of iterations required by HPS for various accuracy levels based on a sequential implementation in C. The algorithm was terminated once the maximum relative error e_{rm} over all processes (cf. Equation (1)) fell below a certain threshold ε.

Fig. 9 shows the results for higher accuracy levels ($\varepsilon = 10^{-4}, 10^{-5}, 10^{-6}$), whereas Fig. 10 focusses on lower

Figure 10. Number of iterations of HPS for lower accuracy levels (median over 100 runs, base topology is fully connected)

accuracy levels ($\varepsilon = 10^{-1}, 10^{-2}, 10^{-3}$). For the accuracy range considered we observe that the number of iterations of HPS grows slower with N than $\log_2 N$, which is the number of iterations required by recursive doubling. More generally speaking, *any* method in which every process receives information from every other process requires $\log_2 N$ iterations (cf. Section 5.1). This indicates that if the system is large enough, HPS will require much less than $\log_2 N$ iterations for reduced accuracy requirements. For very low accuracy requirements, this advantage can already be expected for small or medium-sized systems, as Fig. 10 illustrates.

Whenever HPS performs fewer than $\log_2 N$ iterations, it effectively only (randomly) *samples* the initial data. From the previous observations we can conclude that such random sampling is sufficient for meeting reduced accuracy requirements.

Of course, one would also expect that variants of recursive doubling can be developed which have similar properties. More concretely, one could terminate recursive doubling whenever the root node has reached a certain predefined desired accuracy level. In fact, this strategy effectively corresponds to some form of "sampling" the data to be aggregated. We will consider such an approximative "sampling" variant of recursive doubling (*approximate RDB*, abbreviated as *ARDB*) in Section 6.3 and observe that reduced accuracy requirements can also reduce the number of iterations for a deterministic reduction algorithm. However, the extent of this reduction will in general strongly depend on the distribution of the initial data over the participating processes. For uniformly distributed initial data, the possible reductions in HPS and recursive doubling can be expected to be similar (see Section 6.3). Moreover, an

approximate recursive doubling algorithm will not have the same resilience against SDC as the gossip-inspired HPFLC algorithm.

6.3 Experimental Performance Evaluation of HPS and HPFLC

We now experimentally compare MPI implementations of the gossip-inspired all-reduce scheme HPS with recursive doubling on a supercomputer.

In our first set of experiments, a random single precision initial value is independently generated by each MPI process, and then the parallel reduction operation is performed over these randomly generated initial values. More specifically, the initial values are uniformly distributed in an interval between 0 and the largest possible single-precision floating-point number. The experiments are performed on the Vulcan Supercomputer (2015) of the Lawrence Livermore National Laboratory (LLNL), which is an IBM Blue Gene/Q system composed of 24576 16-core PowerPC-A2 computing nodes.

For HPS we consider different levels of accuracy ε ranging from $\varepsilon = 10^{-2}$ to $\varepsilon = 10^{-5}$. The relative error is defined by $\frac{|\tilde{y} - y_{\text{RD}}|}{y_{\text{RD}}}$, where $\tilde{y}$ denotes the approximation computed by the HPS algorithm or by ARDB, and y_{RD} denotes the result of classical recursive doubling, both at the root node. In this context, we consider the HPS approach to converge with M iterations to a particular accuracy ε if the relative error at the root node after M iterations is less than or equal to ε.

Fig. 11 shows the number of iterations required when considering four different accuracy levels ($10^{-2}, 10^{-3}, 10^{-4}$ and 10^{-5}) and four different parallel scenarios from 16384 up to 131072 MPI processes.

For each particular accuracy and each number of MPI processes, the HPS algorithm has been run 50 times using different randomly generated communication patterns (with identical initial data). The error bars displayed in Fig. 11 represent the standard error computed over the 50 repetitions performed per experiment while each point shown for HPS represents its average. Fig. 11 does *not* reflect the influence of variations in the initial data on the number of iterations of the various algorithms.

The experiment validates the results gathered from our sequential simulations of HPS shown in Figs. 9 and 10: For lower MPI process counts, the results clearly show that for reasonable error levels HPS needs many more iterations until convergence than the $\log_2 N$ iterations recursive doubling would need. However, for higher process counts HPS starts having an advantage in terms of number of iterations until convergence, in particular for low accuracy requirements.
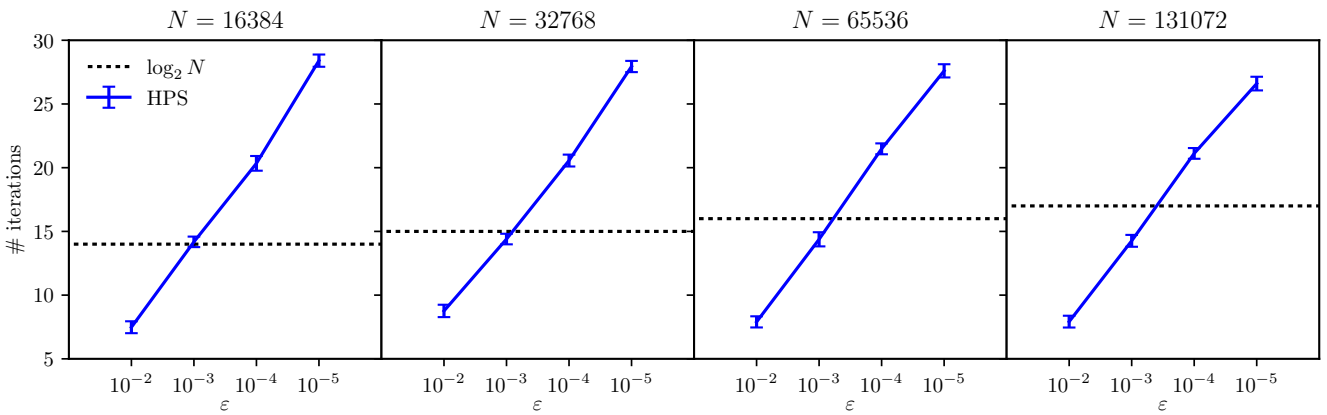
Figure 11. Number of iterations for MPI implementations of HPS on the Vulcan supercomputer (fully connected communication topology, uniformly distributed random initial data, intervals show standard error, 50 runs for each point)

As N grows larger, HPS becomes competitive in terms of number of iterations for higher and higher accuracy.

For Fig. 12 we simulated HPFLC and ARDB on a very large scale. The graphs show the median as well as the minimum and maximum number of iterations over 100 runs for each data point. We note that for low accuracy levels HPFLC clearly outperforms ARDB in all cases and for one billion nodes/processes it is even competitive with ARDB up to an accuracy level of 10^{-5}.

Finally, we also investigate the iteration overhead caused by SDC in HPFLC, i.e., how many extra iterations are needed for a single bit-flip for HPFLC at large scale. Fig. 13 summarizes similar experiments as for PFLC in Fig. 3, showing the average iteration overhead for a single bit-flip at different positions in flow variables of HPFLC. As HPFLC's main strength lies in its fast convergence for low accuracy requirements we internally stored all data in single precision. Fig. 13 shows the iteration overhead for bit-flips in all possible 32 positions. It turns out that HPFLC needs *at most* one additional iteration to compensate for a fault. Most of the time no additional iterations were required, and thus for all positions the average overhead caused by a bit-flip was less than 1%.

7 Conclusions

Gossip-based reduction algorithms as discussed in this paper offer high flexibility and thus have high potential for strong algorithmic resilience and fault tolerance. However, we have shown in this paper that this potential was hardly utilized so far and that the classical gossip-based push-sum algorithm does not have any advantage in terms of resilience over deterministic recursive doubling. Moreover, we have proposed two novel advancements of the push-flow algorithm, PFLC and PFCC, which exploit this fault

tolerance potential better and are much more robust against silent data corruption than any existing reduction algorithm. They can recover from silent data corruption (bit-flips) in all bits of the floating-point representation (including the most significant ones) and still converge to the correct result.

There is currently a trade-off between resilience and performance when comparing deterministic and gossip-based reduction algorithms. We have shown that gossip-based algorithms achieve much better resilience, but deterministic parallel algorithms perform much better. Consequently, we have proposed the novel gossip-*inspired* algorithm HPFLC which inherits the strong resilience properties from PFLC, but achieves much better performance. Although the number of iterations required by HPFLC for achieving full double precision accuracy (within the limitations of floating-point accuracy) is usually significantly higher than the number of steps required by a standard parallel reduction method such as recursive doubling, for *low* accuracy requirements and for *large* process counts the number of iterations required by HPFLC tends to be much lower than for recursive doubling. In these scenarios, HPFLC combines competitive performance with superior resilience.

It is currently still an open question how the runtime performance of gossip-based and gossip-inspired reduction algorithms can be further improved for an HPC context. In addition to efficient implementation, one aspect deserves further exploration: Can the high locality in the communication patterns of gossip-inspired algorithms lead to advantages compared to deterministic parallel reduction algorithms, in particular, in the context of future extreme-scale systems? We expect substantial progress in this question in the near future, which in turn may lead to increased interest in gossip-inspired algorithms for HPC.
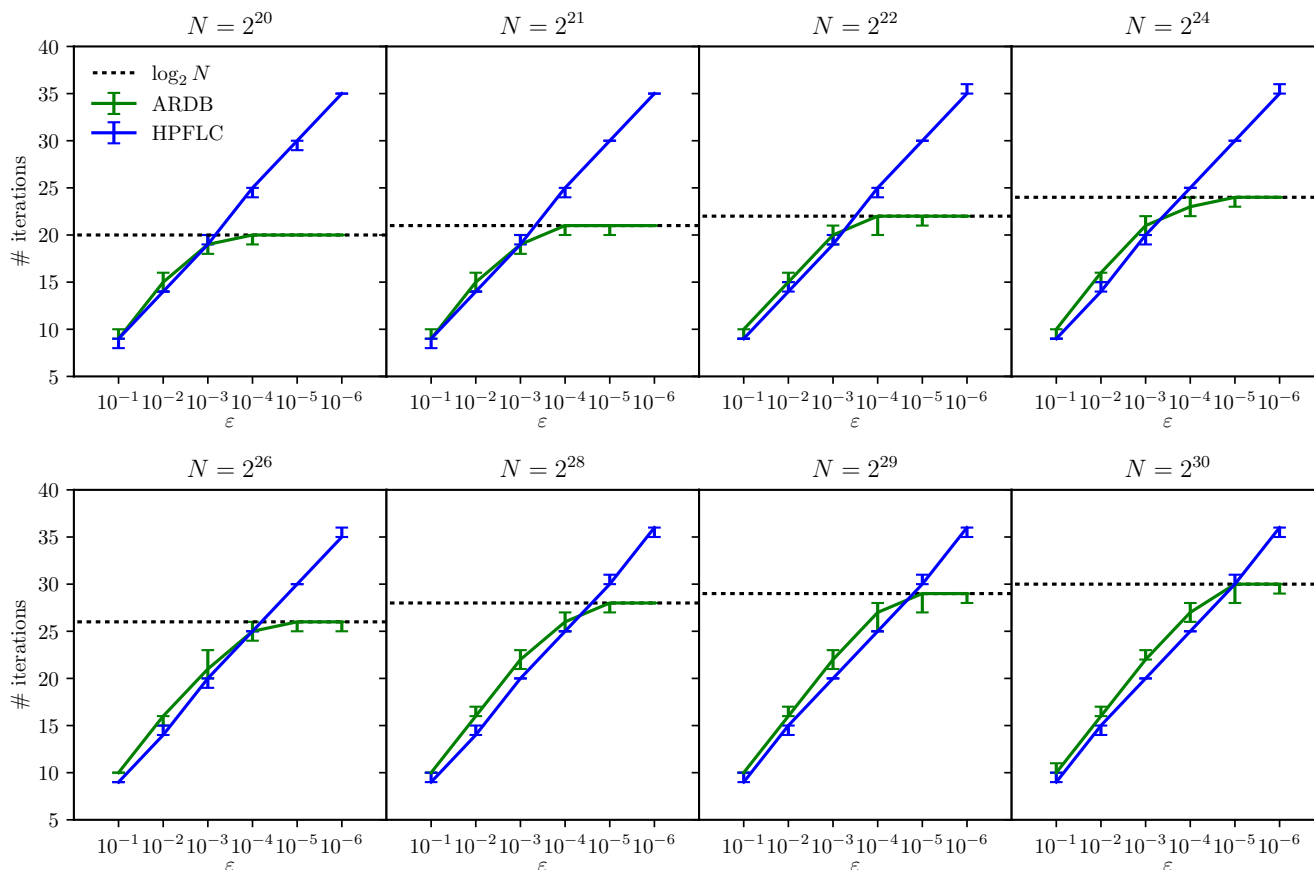
Figure 12. Simulated number of iterations for approximate recursive doubling ARDB and HPFLC (fully connected communication topology, uniformly distributed random initial data, min/max intervals, 100 runs for each point)
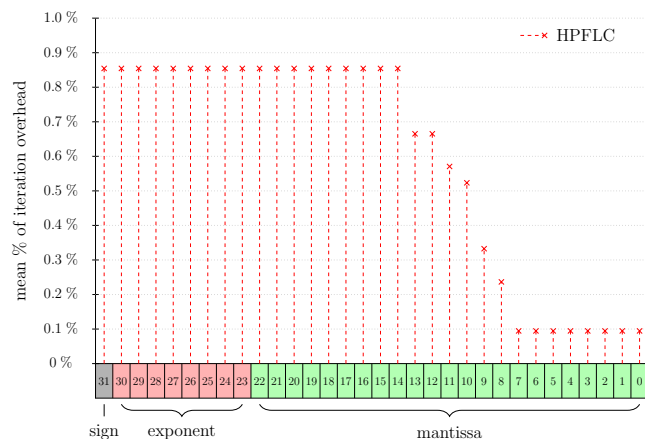
Figure 13. Average iteration overhead caused by a single bit-flip in HPFLC ($N = 2^{20}$, $\varepsilon = 10^{-3}$)

Some of the computational results presented were produced using the Vienna Scientific Cluster (VSC).[*]

Author Biographies

Marc Casas is a senior researcher at the Barcelona Supercomputing Center (BSC). He received a 5-years degree in mathematics in 2004 from the Technical University of Catalonia (UPC) and a PhD degree in Computer Science in 2010 from the Computer Architecture Department of UPC. He was a postdoctoral research scholar at the Lawrence Livermore National Laboratory (LLNL) from 2010 to 2013. His current research interests are high performance computing architectures, runtime systems and parallel algorithms. He is currently involved with the RoMoL and the Montblanc3 projects as well as the IBM-BSC Deep Learning Center.

Wilfried N. Gansterer is full professor and deputy head of the research group Theory and Applications of Algorithms at the Faculty of Computer Science of the University of Vienna. He received a 5-years degree in Mathematics from TU Wien, an M.Sc. degree in Scientific Computing/Computational Mathematics from Stanford University, and a Ph.D. degree in Scientific Computing

Acknowledgements

[*]http://vsc.ac.at/

from TU Wien. His research interests cover various aspects of numerical algorithms, including resilience and fault tolerance, parallel and distributed computing, high performance computing, as well as applications in data mining and Internet security.

Elias Wimmer received Bachelor and Masters degrees in scientific computing at the University of Vienna, where he worked closely together with Prof. Gansterer. Later he joined the research group Parallel Computing at TU Wien as an assistant. Currently he works for IMS, an Intel owned company developing future highly parallel mask writers for VLSI.

References

Ashby S, Beckman P, Chen J, Colella P, Collins B, Crawford D, Dongarra J, Kothe D, Lusk R, Messina P, Mezzacappa T, Moin P, Norman M, Rosner R, Sarkar V, Siegel A, Streitz F, White A and Wright M (2010) The opportunities and challenges of exascale computing. URL https://science.energy.gov/~/MEDIA/ASCR/ASCAC/PDF/REPORTS/EXASCALE_SUBCOMMITTEE_REPORT.PDF.

Bland W, Bouteiller A, Herault T, Bosilca G and Dongarra J (2013) Post-failure recovery of MPI communication capability: Design and rationale. *Int. J. High Perform. Comput. Appl.* 27(3): 244–254.

Bottou L (2010) Large-scale machine learning with stochastic gradient descent. In: Lechevallier Y and Saporta G (eds.) *Proceedings of COMPSTAT'2010: 19th International Conference on Computational Statistics*. Physica-Verlag HD, pp. 177–186.

Boyd S, Ghosh A, Prabhakar B and Shah D (2006) Randomized gossip algorithms. *IEEE Trans. Inf. Theory* 52(6): 2508–2530.

Cappello F, Geist A, Gropp W, Kale S, Kramer B and Snir M (2014) Toward exascale resilience: 2014 update. *Supercomputing Frontiers and Innovations* 1(1): 5–28.

Chan E, Heimlich M, Purkayastha A and van de Geijn R (2007) Collective communication: Theory, practice, and experience. *Concurrency Computat.: Pract. Exper.* 19(13): 1749–1783.

Commonwealth Scientific and Industrial Research Organization (CSIRO) (2014) Scientific computing user manual. URL https://wiki.csiro.au/display/ASC/User+Manual.

Dean J, Corrado GS, Monga R, Chen K, Devin M, Le QV, Mao MZ, Ranzato M, Senior A, Tucker P, Yang K and Ng AY (2012) Large scale distributed deep networks. In: *Proceedings of the 25th International Conference on Neural Information Processing Systems*, NIPS'12. Curran Associates Inc., pp. 1223–1231.

Dennard RH, Gaensslen FH, Yu H, Rideout VL, Bassous E and Leblanc AR (1974) Design of ion-implanted mosfets with very small physical dimensions. *IEEE J. Solid-State Circuits* : 256–268.

Dimakis A, Sarwate A and Wainwright M (2008) Geographic Gossip: Efficient Averaging for Sensor Networks. *IEEE T. Signal Proces.* 56(3): 1205–1216.

Eyal I, Keidar I and Rom R (2011) LiMoSense – Live Monitoring in Dynamic Sensor Networks. In: *Proceedings of the 7th International Symposium on Algorithms for Sensor Systems, Wireless Ad Hoc Networks and Autonomous Mobile Entities*, ALGOSENSORS 2011. pp. 72–85.

Eyal I, Keidar I and Rom R (2014) LiMoSense: live monitoring in dynamic sensor networks. *Distrib. Comput.* 27(5): 313–328.

Ferreira KB, Bridges P and Brightwell R (2008) Characterizing application sensitivity to OS interference using kernel-level noise injection. In: *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08. IEEE Press, pp. 19:1–19:12.

Gansterer WN, Niederbrucker G, Straková H and Schulze Grotthoff S (2013) Scalable and fault tolerant orthogonalization based on randomized distributed data aggregation. *J. Comput. Sci.* 4(6): 480–488.

Gupta S, Agrawal A, Gopalakrishnan K and Narayanan P (2015) Deep learning with limited numerical precision. In: *Proceedings of the 32nd International Conference on Machine Learning*, ICML'15, volume 37. JMLR.org, pp. 1737–1746.

Hadjicostis C, Dominguez-Garcia A and Vaidya N (2012) Resilient average consensus in the presence of heterogeneous packet dropping links. In: *51st Annual Conference on Decision and Control*, CDC 2012. pp. 106–111.

Herault T, Bouteiller A, Bosilca G, Gamell M, Teranishi K, Parashar M and Dongarra J (2015) Practical scalable consensus for pseudo-synchronous distributed systems. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15. ACM, pp. 31:1–31:12.

Hoefler T, Gottschling P, Lumsdaine A and Rehm W (2007) Optimizing a conjugate gradient solver with non-blocking collective operations. *Parallel Comput.* 33(9): 624–633.

Janecek A and Gansterer WN (2016) ACO-inspired acceleration of gossip averaging. In: *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, GECCO '16. ACM, pp. 21–28.

Jesus P, Baquero C and Almeida PS (2009) Fault-Tolerant Aggregation by Flow Updating. In: *Proceedings of the International Conference on Distributed Applications and Interoperable Systems*, DAIS 2009. pp. 73–86.

Jesus P, Baquero C and Almeida PS (2010) Fault-Tolerant Aggregation for Dynamic Networks. In: *Proceedings of the 29th IEEE Symposium on Reliable Distributed Systems*. pp. 37–43.

Jesus P, Baquero C and Almeida PS (2015) Flow updating: Fault-tolerant aggregation for dynamic networks. *J. Parallel Distr. Com.* 78: 53–64.

Jin PH, Yuan Q, Iandola FN and Keutzer K (2016) How to scale distributed deep learning? *CoRR* abs/1611.04581. URL http://arxiv.org/abs/1611.04581.

Katti A, Di Fatta G, Naughton T and Engelmann C (2015) Scalable and fault tolerant failure detection and consensus. In: *Proceedings of the 22nd European MPI Users' Group Meeting*, EuroMPI '15. ACM, pp. 13:1–13:9.

Kempe D, Dobra A and Gehrke J (2003) Gossip-based computation of aggregate information. In: *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*. pp. 482–491.

Lesser B, Mücke M and Gansterer WN (2011) Effects of reduced precision on floating-point SVM classification accuracy. *Procedia Computer Science* 4: 508 – 517.

MPI Forum (2012) MPI: A Message-Passing Interface Standard Version 3.0.

Nichols B, Buttlar D and Farrell JP (1996) *Pthreads Programming*. Sebastopol, CA, USA: O'Reilly & Associates, Inc.

Niederbrucker G and Gansterer WN (2013) Robust gossip-based aggregation: A practical point of view. In: Sanders P and Zeh N (eds.) *Proc. 15th Meeting on Algorithm Engineering & Experiments*. SIAM, pp. 133–147.

Niederbrucker G, Straková H and Gansterer WN (2012) Improving fault tolerance and accuracy of a distributed reduction algorithm. In: *Proc. Third Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*. pp. 643–651.

OpenMP Architecture Review Board (2013) OpenMP application program interface version 4.0.

Seifert N, Gill B, Jahinuzzaman S, Basile J, Ambrose V, Shi Q, Allmon R and Bramnik A (2012) Soft error susceptibilities of 22 nm tri-gate devices. *IEEE T. Nucl. Sci.* 59(6): 2666–2673.

Sidiroglou-Douskos S, Misailovic S, Hoffmann H and Rinard M (2011) Managing performance vs. accuracy trade-offs with loop perforation. In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11. ACM, pp. 124–134.

Thakur R and Gropp WD (2003) Improving the performance of collective operations in MPICH. In: Dongarra J, Laforenza D and Orlando S (eds.) *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, *Lecture Notes in Computer Science*, volume 2840. Springer, pp. 257–267.

Top500 List (2018) TOP500 Supercomputers. URL http://www.top500.org.

Váňa F, Düben P, Lang S, Palmer T, Leutbecher M, Salmond D and Carver G (2017) Single precision in weather forecasting models: An evaluation with the IFS. *Monthly Weather Review* 145(2): 495–502.

Vulcan Supercomputer (2015) Livermore Computing. URL http://computation.llnl.gov/computers/vulcan.