universität wien

MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

Fault-tolerance and Scalability of Gossip-based Reduction Algorithms

verfasst von / submitted by

Elias Wimmer, BSc

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Diplom-Ingenieur (Dipl.-Ing.)

Wien, 2016 / Vienna 2016

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	A 066 940
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Scientific Computing UG2002
Betreut von / Supervisor:	Assoz. Prof. Dipl.-Ing. Dr. Wilfried Gansterer

A publication based on the results of this thesis is currently in preparation

Abstract

It is anticipated that future high performance computing (HPC) systems are much more vulnerable against failures as silent data corruption. We compare the behavior of different types of algorithms for parallel all-to-all aggregation under the influence of silent data corruption (bit-flips). It turns out that gossip-based algorithms have clear advantages over classical deterministic algorithms. But even existing gossip-based algorithms cannot handle all bit-flips in their data structures. Consequently, we also present two novel gossip-based aggregation algorithms which achieve much better resilience against silent data corruption than all existing algorithms. Nevertheless, in HPC, performance is the key to acceptance. Therefore, we investigate the cost of gossip-based aggregation algorithms with different communication strategies on network topologies used in HPC. We verify the derived cost models with results from the Vienna Scientific Cluster[1] and give an outlook on how to further improve the performance of our algorithms.

[1]The computational results presented have been achieved in part using the Vienna Scientific Cluster (VSC).

Zusammenfassung

Es wird angenommen, dass zukünftige „High Performance Computing"-Systeme viel anfälliger für „Silent Data Corruption"-Speicherfehler als derzeitige Systeme sein werden. Wir vergleichen das Fehlerverhalten unterschiedlicher Arten von parallelen Aggregationsalgorithmen unter dem Einfluss von „Bit-Flips". Es stellt sich heraus, dass gossip-basierte Algorithmen einen klaren Vorteil gegenüber klassischen deterministischen Algorithmen haben. Trotzdem können auch existierende gossip-basierte Algorithmen nicht alle „Bit-Flip"-Fehler korrigieren. Deshalb stellen wir zwei neue gossip-basierte Aggregationsalgorithmen vor, welche eine überlegene Fehlertoleranz gegenüber allen bisherigen Algorithmen zeigen. Da Performance die wichtigste Eigenschaft im „High Performance Computing" ist, untersuchen wir die Kosten von gossip-basierten Aggregationsalgorithmen für verschiedene Kommunikationsstrategien in „High Performance Computing"-Netzwerktopologien. Anschließend verifizieren wir die abgeleiteten Kostenmodelle mit Laufzeitergebnissen vom „Vienna Scientific Cluster 2"[2] und geben einen Ausblick wie man die Performance von unserer Algorithmen weiter verbessern kann.

[2]Die präsentierten Rechenergebnisse wurden zum Teil am Vienna Scientific Cluster (VSC) erzielt.

Contents

List of Algorithms

List of Figures

1. Introduction

Resilience against various types of hard- and software faults and their consequences has been investigated in various domains. Classical distributed computing theory has mostly focused on faults in fundamental abstract coordination problems, e. g., consensus and leader election, and in algorithms for combinatorial problems, e. g., finding (minimum) spanning trees, but has much less addressed numerical problems. High Performance Computing (HPC) is a practical example where the relevance of fault-tolerance is expected to increase in the foreseeable future. In particular, future exascale systems are likely to exhibit (much) higher fault rates than current systems [4, 25]. Research and development for increasing fault-tolerance at the software, algorithms and applications level has been recognized as pivotal to achieving the exascale objectives [8].

Since many different types of errors and failures may occur much more often than in current systems the problem in its full generality is an extremely difficult one (see, e. g., [24]). It is to be expected that comprehensive fault-tolerance concepts for large-scale systems require a combination and integration of efforts and techniques at several layers (hardware, system, software and algorithms layer).

In this thesis, we exclusively focus on techniques at the algorithmic layer in an attempt to advance the understanding of the potential as well as the fundamental limitations of purely algorithmic fault-tolerance, which has important consequences for the design of algorithms for future extreme-scale systems. More specifically, we address one prototypical instance of the complex fault-tolerance problem: We focus on silent (unreported) data corruption in the context of parallel all-to-all aggregation operations (such as summation or averaging), a special case of parallel reduction operations. These operations are of fundamental importance in scientific HPC, since they appear as building blocks in important kernels, e.;g., scalar products in iterative linear solvers. Although the likely frequency of silent data corruption in future systems is still unclear, the risk of silent data corruption is predicted to increase [4, 8]. Beyond brute force approaches such as duplication or triplication, so far only few algorithmic approaches are available which have some potential for coping with frequent silent data corruption.

Two basic types of parallel aggregation algorithms can be distinguished: On the

one hand, classical deterministic parallel aggregation algorithms are state-of-the-art in HPC. On the other hand, consensus and randomized gossip-based aggregation algorithms have so far been mostly considered in loosely coupled distributed systems. As we will illustrate, these two types of algorithms exhibit important differences in terms of (potential) resilience against silent data corruption.

1.1 Related Work

Today's standard fault-tolerance paradigm in HPC applications is checkpoint-restart (C/R), which essentially only targets *detected* fail-stop errors. Although many variants of checkpointing have been investigated and important progress has been made (see, e. g., [4] and the references therein), classical C/R tends to become less efficient for large systems [11] and it is still unclear whether it can be made fast and scalable enough for extreme-scale systems.

Other efforts focus on integrating concepts for supporting fault-tolerance in the message passing interface (MPI) layer of HPC applications [14, 3, 1]. For the User-Level Failure Mitigation (ULFM) proposal [1], gossip protocols have been advocated for fault-tolerant failure detection and consensus [16, 13]. However, so far no standardization has been agreed upon.

At the algorithmic level, approaches to algorithmic fault-tolerance for numerical problems can be grouped into three categories: *algorithm-based fault-tolerance* (*ABFT*) methods, *fault-oblivious iterative* (*FOI*) methods, and *randomized* methods. ABFT methods (e. g., [6]) are based on check-sums. They require explicit detection of silent data corruption, and they are most attractive if the number of errors is low. Moreover, they are rather limited in terms of the temporal distribution of the errors they can handle. FOI methods have either considered (reported) node failures (e. g., [18]) or assume a *reliable mode* for parts of the computation without requiring explicit detection of silently corrupted data (e. g., [9]).

Gossip-based aggregation algorithms [10, 15, 12] have randomized communication schedules which makes them much more flexible and potentially more robust than classical parallel algorithms. Every node operates only based on local knowledge of its neighborhood and conceptually no synchronization across the network is assumed. Consequently, gossip-based algorithms potentially have the strongest resilience properties among the approaches at the algorithmic level. To some extent, they can be *self-healing*, i. e., they can potentially deliver a correct result in the presence of errors or failures *without* the need to explicitly detect a failure.

Most existing work on fault-tolerant gossip-based aggregation algorithms exclusively focuses on message loss or node failures [10, 15, 12], and there is only one limited study on the impact of certain types of bit flips [21]. Obviously, it is to be expected that the greatly increased flexibility and robustness of gossip-based algorithms come

with some overhead compared to routines optimized for classical parallel systems. However, it is still widely unclear how well silent data corruption can be handled by gossip-based aggregation algorithms in practice, in particular, in the context of HPC. The main goal of the thesis is to make a first step towards a clearer understanding of the resilience properties of gossip-based aggregation algorithms against silent data corruption and to quantify the costs for their improved resilience.

1.2 Contributions and Synopsis

First, we thoroughly discuss existing parallel all-reduction and gossip-based all-to-all aggregation algorithms in terms of fault-tolerance against silent data corruption and message loss. We describe in detail limitations of so far existing flow-based gossip algorithms and give solutions to overcome most of the issues. For the Push-Cancel-Flow algorithm we give an example that highlights its problems with consecutive messages.

Next, we propose two novel gossip-based aggregation algorithms *Push-Flow with local correction* (*PFLC*) and *Push-Flow with cooperative correction* (*PFCC*) which leverage the concepts of flows to its full extent and therefore significantly extend the resilience against silent data corruption at the algorithmic level.

Then, we investigate for all discussed algorithms the influence of silent data corruption. More specifically, we illustrate that basic gossip aggregation methods – despite their flexibility and potential resilience against other types of faults – do not have any advantages compared to recursive doubling, the standard parallel aggregation algorithm for small messages sizes, when it comes to silent data corruption. We show that even for advanced gossip aggregation methods the location of the corrupted bit is crucial. No aggregation method existing so far can recover from bit-flips in the higher significant bits in the exponent of the floating-point representation. Only the novel algorithms PFLC and PFCC recover from silent data corruption in any bit of relevant data structures.

Finally, we study the scalability and runtime characteristics of gossip-based aggregation algorithms in comparison to parallel reduction algorithms. We discuss the theoretical bounds for all-reductions, as well the asymptotic runtime complexity for gossip-based aggregation. For prototypical communication schedules in gossiping, we give an in detail analysis of the communication costs scaling by problem size. Based on dozens of simulations we build scalability models for all our algorithms and derive runtime cost models for the Vienna Scientific Cluster 2, which we compare with real runtime results from our native implementations for HPC systems. We give results for several accuracy levels of gossip-based aggregation and point out further directions to improve the performance of our algorithms.

In chapter 2 we summarize the state-of-the-art in parallel reduction aswell in gossip-

based aggregation algorithms. In chapter 3 we present the two novel algorithms PFLC and PFCC with improved resilience against silent data corruption. In chapter 4 we discuss experimental evaluations of the various algorithms. In chapter 5 we evaluate the performance of gossip-based all-to-all aggregation in comparison to state-of-the-art parallel all-reduce algorithms. Chapter 6 concludes the thesis and outlines future research directions. In appendix A we give an short overview of the developed simulators and HPC implementations used in the thesis.

2. All-to-all aggregation / reduction

In this section we review existing deterministic and gossip-based algorithms for performing an all-to-all reduction / aggregation operation (*all-reduce*) over N nodes in parallel.

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

In general, $\otimes$ denotes an element-wise associative and commutative binary operation, but in this paper we focus on summation and averaging.

The initial data $x_0, x_1, \ldots, x_{N-1}$ (for simplicity, we consider scalar initial data) is distributed over the N nodes. The result y of the all-reduce operation (the *aggregate*) is available at all nodes.

2.1 Parallel all-reduce

As a reference all-reduce algorithm, we have chosen *recursive doubling* (RDB). In modern HPC systems there are several all-reduce algorithms in use, different algorithms for different combinations of message size and number of nodes [26, 5]. Recursive doubling is not optimal for larger messages, but for small message sizes, as we focus in this work, it is optimal in the number of messages needed to compute a parallel all-reduce operation. Furthermore all currently used all-reduce algorithms in HPC have similar (none) fault-tolerance capabilities.

Recursive doubling

Recursive doubling [26] calculates an all-reduce operation on a power of 2 nodes (for simplicity of the algorithm numbered from 0,...,N-1) in $d = \log_2 N$ iterations (Algorithm 1 line 2). In each iteration k, every node i pairwise exchanges its current value with a node j. Node j is chosen such its node id differs exactly in the kth bit with the node id of i (Algorithm 1 line 3 to 5, $\oplus$ denotes the exclusive or operation). This guarantees a pairwise exchange of values between nodes, where the distance between nodes is doubled in every iteration and in the end every node

Algorithm 1 Recursive doubling for $N = 2^d$ (RDB)

Input: $x_i \in \mathbb{R}$
Output: $y_i = \bigotimes_k x_k \in \mathbb{R}$

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

got information from each other node. Henceforth the operator $\otimes$ is applied on the values y_i and y_j and the result is set as the new value y_i for the next iteration (Algorithm 1 line 6). After d iterations the result of the all-reduction is available in y_i to all nodes.

Fault tolerance properties.

Classical parallel reduction approaches like recursive doubling require a pre-determined, sophisticated 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. In case of unhandled message loss, algorithms like Recursive Doubling have a undefined behavior, whereas in the case of SDC in reduction data, it is very likely that the computation succeeds, but the error is propagated to many nodes, leading to wrong results.

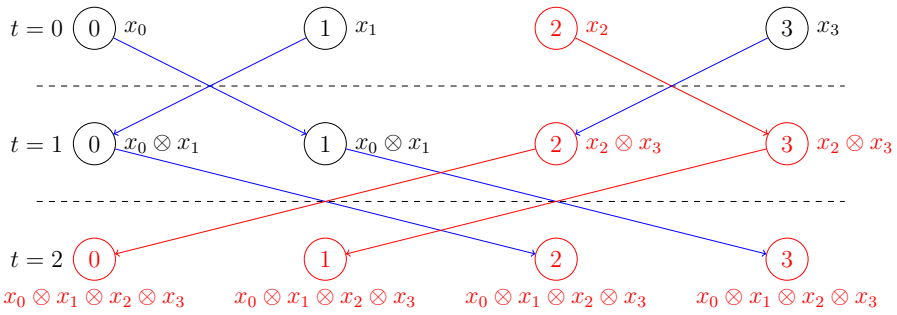


Figure 2.1: Silent data corruption propagation in recursive doubling

In Figure 2.1 it is depicted how an error propagates in recursive doubling with 4 nodes. The error occurs on node 2 before the first message got sent. In each iteration it infects double the nodes as in the iteration before. In this case, at the end, all nodes are affected by the error. Depending on when an error occurs, the failure applies to more or less nodes.

2.2 Gossip-based all-to-all aggregation

Gossip algorithms are decentralized iterative algorithms originally intended for loosely coupled distributed systems, such as P2P or ad-hoc networks. In each iteration every node i chooses a *random* communication partner j from its neighborhood $\mathcal{N}_i$. In an HPC context, this neighborhood would usually be defined by a suitable overlay network which is allows for optimizing performance.

2.2.1 The Push-Sum algorithm

The *Push-Sum* algorithm [17] (PS) is a gossip algorithm for summing N values $x_0^0, x_1^0, \ldots, x_{N-1}^0$ distributed across N nodes. Each node i starts with an initial value-weight pair (x_i^0, w_i^0) and computes an approximation of the weighted sum:

$$y_i \approx \frac{\sum_{k=0:N-1} x_k^0}{\sum_{k=0:N-1} w_k^0}$$

At any time t, every node i has a local approximation y_i^t of this sum. Note that this sum represents the average over all x_i^0 if $w_i^0 = 1 \, \forall i$, and the sum over all x_i^0 if $\sum_i w_i^0 = 1$. For simplicity of notation we will omit the time superscript (0 or t) unless it is needed to avoid ambiguity.

Algorithm 2 Push-Sum Algorithm (PS)

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

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

In each local iteration every node 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 (Algorithm 2, lines 2 to 4). Afterwards all received value-weight pairs (x_j, w_j) are added to the local value-weight pair (Algorithm 2, lines 5 to 6). This is repeated until convergence is reached. Note that every node i can send messages independently of all other nodes. Thus, Push-Sum (like other gossip-based algorithms) does not require synchronization of sending and receiving messages among nodes.

The local approximation y_i of the aggregate is computed by dividing the current x_i by the current w_i (Algorithm 2, line 7). We refer to ε as "target accuracy" and

say that Push-Sum computed an ε-accurate estimate of the true aggregate if the maximum relative error over all nodes is bounded by ε (Equation (4.1)).

Fault tolerance properties

For correctness of the Push-Sum algorithm *mass conservation* needs to be ensured [17], 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 process will still converge at every node, but not to the correct aggregate of the initial values.

Mass conservation is a *global* property which is easily violated by message loss and (silent) data corruption. Although the flexible communication schedule of Push-Sum allows for tolerating *reported* failures, any kind of *unreported* failure, such as silent data corruption, violates mass conservation. Thus, the Push-Sum algorithm cannot guarantee correct all-to-all reductions in the presence of silent data corruption. Consequently, Push-Sum is *not* suited for the robust computation of all-to-all aggregations.

2.2.2 Push-Flow

The *Push-Flow* algorithm [12] (PF) can be viewed as an advancement of Push-Sum which utilizes the graph theoretical flow concept for improving resilience.

Algorithm 3 Push-Flow Algorithm (PF)

Input: (x_i, w_i), ε
Initial values: $\forall\, k \in \mathcal{N}_i : \; f_{i,k} \leftarrow (0, 0)$
Output: $y_i \approx \sum_{k=0:N-1} x_k \,/\, \sum_{k=0:N-1} w_k$

1: **while** not ε-accurate **do**
2: $\quad e_i \leftarrow (x_i, w_i) + \sum_{k \in \mathcal{N}_i} f_{i,k}$
3: $\quad j \leftarrow$ choose a neighbor $\in \mathcal{N}_i$ uniformly at random
4: $\quad f_{i,j} \leftarrow f_{i,j} - e_j/2$
5: $\quad$ **send** $f_{i,j}$ to node j
6: $\quad$ **for each** received flow $f_{j,i}$ from node j **do**
7: $\quad\quad f_{i,j} \leftarrow -f_{j,i}$
8: $y_i \leftarrow e_i(1)/e_i(2)$

For every neighboring node $j \in \mathcal{N}_i$ node i holds a *flow* variable $f_{i,j}$ representing the data flow (the mass sent between two nodes) to node j. While in Push-Sum the sent data ("mass") is immediately added to the current value-weight pair, in Push-Flow sent mass it is only stored in the flow variables. Hence, the initial value-weight pair is never changed. The current estimate e_i of node i can be computed as $e_i = (x_i, w_i) + \sum_{k \in \mathcal{N}_i} f_{i,k}$ (Algorithm 3, line 2).

Like in Push-Sum, in each round every node i chooses some node $j \in \mathcal{N}_i$ to which it sends half of its current estimate e_i. In contrast to Push-Sum, only the flow variables are updated and sent (Algorithm 3, lines 4 and 5). All received flows are negated and overwrite the corresponding flow variable (Algorithm 3, lines 6 to 7). The local approximation y_i of the aggregate is computed by dividing the value component $e_i(1)$ of the current estimate e_i by the weight component $e_i(2)$ (Algorithm 2, line 8).

Fault tolerance properties

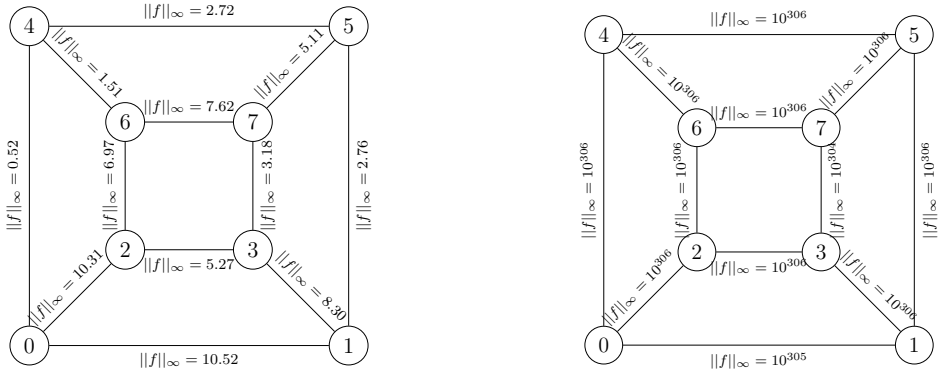
In Push-Flow *flow conservation*, i. e., $\sum_i \sum_{j \in \mathcal{N}_i} f_{i,j} = (0, 0)$, is ensured. Note that flow conservation *implies* mass conservation [12]. In contrast to mass conservation, flow conservation is a *local* property and can be maintained more easily. In particular, every successful transmission of an uncorrupted flow variable re-establishes flow conservation between two nodes. This highlights the role of the flow concept as major source of fault tolerance.

The Push-Flow algorithm naturally recovers from loss of messages at the next successful failure-free communication without even detecting it explicitly [12]. Additionally, *detected* broken system components, e. g., 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 [12]. In *exact* arithmetic, Push-Flow can also recover from SDC in the flow variables and/or messages (with some additional cost).

Numerical inaccuracy of Push-Flow

However, in floating-point arithmetic Push-Flow has numerical accuracy issues on larger node counts due to the growth of the flow variable entries [22]. Further, in case of SDC, when a large error in the flow variables occurs, the flow variable entries grow even faster, preventing Push-Flow from converging to the correct result, as huge round off errors can occur.

The additional mass introduced into flow variables when certain bits get flipped, is equally spread over all nodes flows. Without SDC, the flow variables grow only moderately over time, as Figure 2.2a shows for a 3-cube. On each edge the infinity norm over the flow entries is given at the time when it was converged. In case of a bit-flip in a flow variable, which introduces a huge error the additional mass is spread over all flow variables in all nodes, as shown in Figure 2.2b. The Push-Flow algorithm was alternatively stopped after 500 iterations, as it could not reach the target accuracy.



(a) no bit-flip, converged **(b)** bit-flip in 62th bit, not converged

Figure 2.2: Silent data corruption in Push-Flow can result in extremely large entries in flow variables leading to floating-point round off errors (Python Discrete Event Simulator (Appendix A.3), 8 nodes, hypercube communication topology, $\varepsilon = 10^{-14}$)

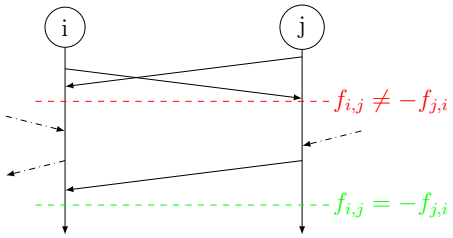


Figure 2.3: Push-Flow: Concurrently sent messages violate flow conservation

Concurrent messages

Another issue that arises when Push-Flow is implemented in parallel, is the problem of concurrently sent messages between nodes. In this case, as Figure 2.3 shows, flow conservation is violated until it gets reestablished by a non-concurrent message. Meanwhile mass conservation is violated and therefore Push-Flow temporally converges to a wrong aggregate.

Table 2.1 gives the results from single runs of Push-Sum and Push-Flow, both with the same initial values and communication schedules. Push-Flow on fully-connected topology is worst effected by concurrent messages. In the case of 512 nodes it could not reach the target accuracy due to the numerical problems of Push-Flow. On a hypercube topology the overhead seams to be a linear factor, whereas for a 3D-Torus the overhead gets less by growing node size. Overall Push-Flow needs significantly more messages to converge than Push-Sum in a parallel environment.

# nodes	PS	PF
32	104	564
64	116	1231
128	120	1871
256	126	4680
512	129	–

(a) Fully-Connected

# nodes	PS	PF
32	163	249
64	176	335
128	210	355
256	253	384
512	272	400

(b) hypercube

# nodes	PS	PF
27	138	272
64	203	328
125	257	351
216	368	387
512	569	432

(c) 3D-Torus

Table 2.1: Push-Flow vs. Push-Sum, maximum messages sent until converged on different network sizes and topologies (Python Discrete Event Simulator (Appendix A.3), $\varepsilon = 10^{-11}$)

Communication on directed graphs

To hinder Push-Flow from sending concurrent messages, we experimented with directed graphs, where Push-Flow can only send in one direction and therefore in the failure free case flow conservation is always guaranteed.

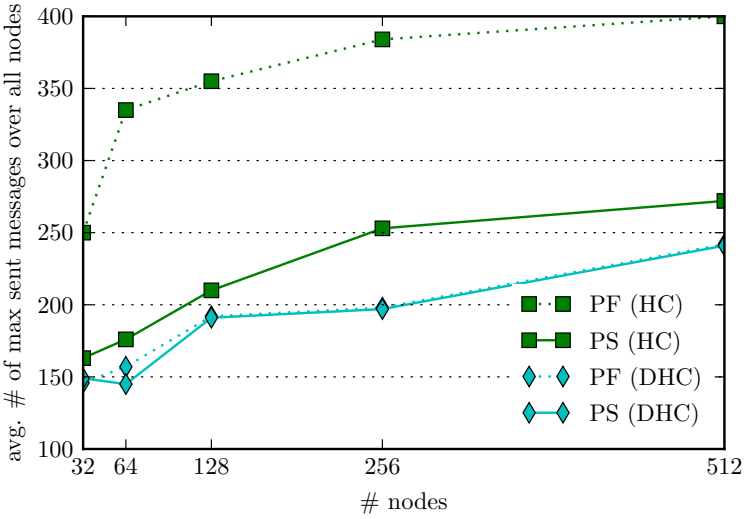


Figure 2.4: Push-Flow (PF) vs. Push-Sum (PS), maximum number of messages over all nodes sent until convergence on a directed hypercube (DHC) compared to a undirected hypercube (HC) (average over 5 runs, Python Discrete Event Simulator (Appendix A.3), $\varepsilon = 10^{-11}$)

On the one hand, using a directed network topology for communication perfectly solves the problem of violated flow conservation, it even showed that on some directed graphs Push-Sum and Push-Flow needs fewer messages than on the undirected underlying graph (Figure 2.4). On the other hand, is it hard to find optimal directed graphs for every network size and topology, therefore we can only give results for a directed hypercube [7]. However, directed communication does not solve the accuracy problems of Push-Flow and additionally prevents further methods to

recover from silent data corruption as presented in Section 3.2.

Mutually exclusive communication

A different approach to solve the concurrent message problem of Push-Flow is to only allow mutual exclusive communication between nodes. Theoretically this should work with any network topology and is therefore more flexible than communication along directed edges, however in practice many questions arise.

- How can the first partner be chosen in a way that no concurrently sent messages appear?
- What to do in case of message loss?
- Is it guaranteed dead locks free?

As this is beyond the scope of this thesis, we have to leave the questions and solutions open for further investigation.

2.2.3 Push-Cancel-Flow

To overcome the accuracy problems of Push-Flow, Push-Cancel-Flow (PCF) [22] resets the flow variables whenever flow conservation is reached. Consequently, as the flow variable entries stay small, PCF achieves the same accuracy as Push-Sum.

Push-Cancel-Flow (Algorithm 4) stores the sum over all flows in φ_i. When flow conservation holds, i.e., $f_{i,j} = -f_{j,i}$, both flows should be added to φ_i and φ_j, then Push-Cancel-Flow tries to set the flows to zero. Therefore, Push-Cancel-Flow uses two flow variables $f_{i,j,0}$ and $f_{i,j,1}$, which serve different roles depending on the value of $c_{i,j}$. One flow is the *active flow* which holds the current flow as in Push-Flow, the other flow variable is called *passive flow* and is used to achieve $f_{i,j} = -f_{j,i} = 0$. When both passive flows are zero, $c_{i,j}$ is toggled and active and passive flow change their role.

The entire handling of the additional logic of Push-Cancel-Flow is located in the receive part of the algorithm. Sending is analogous to Push-Flow.

When a message gets received it is first checked whether active and passive flow have changed their role (Case 0 in Algorithm 4). Afterwards, first the active flow is added to φ_i (lines 10 and 11, Algorithm 4) and then the passive flow gets processed, therefore three cases are distinguished.

Case 2: Flow conservation was reached, the passive flows get zeroed (lines 12-14, Algorithm 4).
Case 3: The passive flows were successfully set to zero, active and passive flow change their role. (lines 15-18, Algorithm 4).

Algorithm 4 Push-Cancel-Flow Algorithm (PCF)

Input: $(x_i, w_i),\ \varepsilon$
Initial values: $\varphi_i \leftarrow (0,0),\ \forall\, k\ \in \mathcal{N}_i :\ \{f_{i,k,0}, f_{i,k,1}\} \leftarrow (0,0),\ \{c_{i,k}, r_{i,k}\} \leftarrow 0$
Output: $y_i \approx \sum_{k=1:N} x_k \,/\, \sum_{k=1:N} w_k \in \mathbb{R}$

1: **while** not ε-accurate **do**
2: $\quad\varphi_i \leftarrow \varphi_i - ((s_i, w_i) + \varphi_i)/2$
3: $\quad j \leftarrow$ choose a neighbor $\in \mathcal{N}_i$ uniformly at random
4: $\quad f_{i,j,c_{i,j}} \leftarrow f_{i,j,c_{i,j}} - ((s_i, w_i) + \varphi_i)/2$
5: $\quad$**send** $(f_{i,j,0}, f_{i,j,1}, c_{i,j}, r_{i,j})$ to node j
6: $\quad$**for each** received message $(f_{j,i,0}, f_{j,i,1}, c_{j,i}, r_{j,i})$ from node j **do**
7: $\quad\quad$**if** $c_{i,j} \neq c_{j,i}$ and $r_{i,j} = r_{j,i}$ **then** ▷ Case 0
8: $\quad\quad\quad c_{i,j} \leftarrow c_{j,i}$
9: $\quad\quad$**if** $c_{i,j} = c_{j,i}$ **then** ▷ Case 1
10: $\quad\quad\quad \varphi_i \leftarrow \varphi_i - (f_{i,j,c_{i,j}} + f_{j,i,c_{j,i}})$
11: $\quad\quad\quad f_{i,j,c_{i,j}} \leftarrow -f_{j,i,c_{j,i}}$
12: $\quad\quad\quad$**if** $f_{j,i,c^c_{j,i}} = -f_{i,j,c^c_{i,j}}$ and $r_{i,j} = r_{j,i}$ **then** ▷ Case 2
13: $\quad\quad\quad\quad f_{i,j,c^c_{i,j}} \leftarrow (0,0)$
14: $\quad\quad\quad\quad r_{i,j} \leftarrow r_{i,j} + 1$
15: $\quad\quad\quad$**else if** $f_{j,i,c^c_{j,i}} = (0,0)$ and $r_{i,j} + 1 = r_{j,i}$ **then** ▷ Case 3
16: $\quad\quad\quad\quad f_{i,j,c^c_{i,j}} \leftarrow (0,0)$
17: $\quad\quad\quad\quad r_{i,j} \leftarrow r_{i,j} + 1$
18: $\quad\quad\quad\quad c_{i,j} \leftarrow c^c_{i,j}$
19: $\quad\quad\quad$**else if** $r_{i,j} \leq r_{j,i}$ **then** ▷ Case 4
20: $\quad\quad\quad\quad \varphi_i \leftarrow \varphi_i - (f_{i,j,c^c_{i,j}} + f_{j,i,c^c_{j,i}})$
21: $\quad\quad\quad\quad f_{i,j,c^c_{i,j}} \leftarrow -f_{j,i,c^c_{j,i}}$
22: $\quad\quad$**else** ▷ Case 5
23: $\quad\quad\quad$**erroneous state**
24: $y_i \leftarrow \varphi_i(1)/\varphi_i(2)$

Case 4: Flow conservation for the passive flows does not hold, passive flows are treated like active flows (lines 19-21, Algorithm 4).

Consecutive messages

Through consecutive messages it can happen that the r and c variables of Push-Cancel-Flow get into an inconsistent state. Then, the condition $c_{i,j} = c_{j,i}$ (Case 1 in Algorithm 4) fails and Push-Cancel-Flow stops making progress.

The example in Figure 2.5 shows how this erroneous state can be reproduced. When the passive flows of two neighboring nodes are the same and both have equal c and r, then Push-Cancel-Flow tries to reset the flows. The next message from j to i

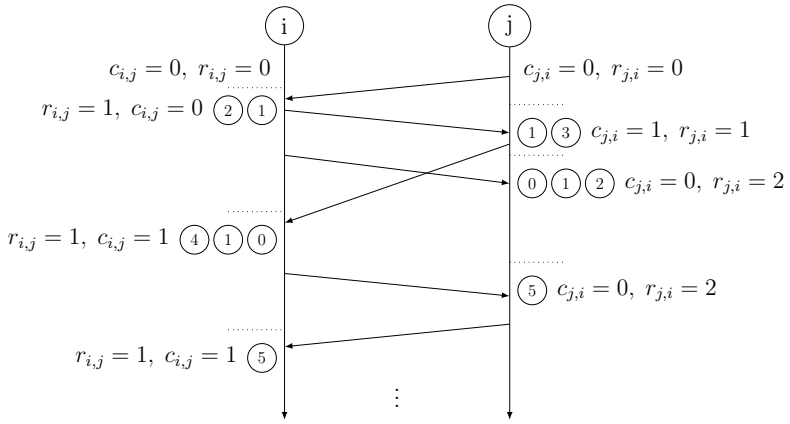


Figure 2.5: Consecutive messages can lead to a combination of r and c where mass gets lost

triggers condition 1 and 2 in Algorithm 4, $r_{j,i}$ gets increased and $f_{i,j,c^c_{i,j}}$ is set to zero. If now i send two consecutive messages to j without receiving any message from j in between and j responds to the first message, then Push-Cancel-Flow sticks to a state where every message between the neighboring nodes is discarded. The first message from i to j satisfies case 1 and 3, $c_{j,i}$ gets flipped and $r_{j,i}$ gets 1. The second message triggers case 0, as $c_{i,j} \neq c_{j,i}$ and $r_{i,j} = r_{j,i}$, then case 1 and case 2 apply, as $f_{j,i,c^c_{j,i}} = -f_{i,j,c^c_{i,j}}$ and $r_{i,j} = r_{j,i}$. Now $c_{i,j}$ is switched again and $r_{i,j}$ gets 2. The message sent from j to i in between the two messages gets received by i after the second consecutive message. For that message conditions 0,1 and 4 hold, as $c_{i,j} \neq c_{j,i}$ and $r_{i,j} = r_{j,i}$, but $f_{j,i,c^c_{j,i}} \neq 0$.

Whenever one of the two node sends now a message to the other node, case 0 in Algorithm 4 does not apply as $r_{i,j} \neq r_{j,i}$ and case 1 in Algorithm 4 does also not apply as $c_{i,j} \neq c_{j,i}$, thus the message gets discarded (Case 5 in Algorithm 4). Push-Cancel-Flow looses mass as this behavior is permanent.

Push-Cancel-Flow, as given in [22], is in practice not functional on realistic systems where consecutive messages will occur!

Avoiding consecutive messages

A simple solution to solve the problem is to track sent messages and only send to a neighbor node after receiving a message from it. This guarantees that no consecutive messages between two neighboring nodes are sent and therefore Push-Cancel-Flows cannot reach the state where mass gets lost.

Fault tolerance properties

Push-Cancel-Flow can handle message loss equally well as Push-Flow as both use the concept of flows. But due to the way flows are added to φ_i, it can not handle silent data corruption in flow variables. In Push-Cancel-Flow the flow $f_{i,j,c_{i,j}}$ gets subtracted from φ_i and then $f_{j,i,c_{j,i}}$ is added to φ_i (line 10 in Algorithm 4). In case a bit in $f_{i,j,c_{i,j}}$ got flipped, then in φ_i the correct $f_{i,j,c_{i,j}}$ is included, but the corrupted $f_{i,j,c_{i,j}}$ gets subtracted, therefore the difference of the correct and corrupted flow will stay as an error in φ_i.

Additionally, the methods described in chapter 3 cannot be applied to Push-Cancel-Flow, as they require that the sum over all flows is always computed all over again, as otherwise silent data corruption leads to the same problems as with φ in Push-Cancel-Flow. Thus, the influence of silent data corruption is the same as in Push-Sum.

3. Improving fault-tolerance

Although in theory the concept of flows allows Push-Flow to recover from any bit-flip in the flow variables and messages, in practice the limited precision of floating-point arithmetic causes mass loss [21, 22]. 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. In Section 3.3 we also briefly outline how to recover from SDC in the initial data.

3.1 Push-Flow with local correction

The basic idea for tolerating the effects of bit-flips which introduce a large error is to integrate explicit fault detection strategies into the Push-Flow algorithm. In our context it suffices to consider very simple checksums, i. e., summing the value-weight pair. Although there are more sophisticated methods for error detection, for us it is essential that the introduced error can be *quantified*.

The novel gossip-based algorithm *Push-Flow with local correction* (*PFLC*) (see Algorithm 5) 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 represents the corresponding flow. After computing the estimate e_i (Algorithm 5, line 2), it tests 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 certain threshold τ, an error in the flow variables is detected (Algorithm 5, line 3). In this case, the algorithm tests every flow variable for an error and repairs the local data by setting flow variables to zero when necessary (Algorithm 5, lines 4 to 6). Afterwards the current estimate e_i is recalculated with the new flow variables (Algorithm 5, line 7).

The sending part is identical to Push-Flow (Algorithm 5, lines 8 to 10). When receiving a message from node j the checksum for the received flow $f_{j,i}$ is verified. Only if the checksum error is below the threshold τ, the flow is set as the new local flow

Algorithm 5 Push-Flow with Local Correction (PFLC)

Input: $(x_i, w_i),\ \varepsilon,\ \tau$
Initial values: $c_i = x_i + w_i\ , \forall\, k\ \in \mathcal{N}_i :\ f_{i,k} \leftarrow (0,0,0)$
Output: $y_i \approx \sum_{j=0:N-1} x_j\ /\ \sum_{j=0:N-1} w_j$

```
1:  while not ε-accurate do
2:      e_i ← (x_i, w_i, c_i) + Σ_{k∈N_i} f_{i,k}
3:      if |e_i(1) + e_i(2) − e_i(3)| > τ then
4:          for each k ∈ N_i do
5:              if |f_{i,k}(1) + f_{i,k}(2) − f_{i,k}(3)| > τ then
6:                  f_{i,k} ← (0, 0, 0)
7:          e_i ← (x_i, w_i, c_i) + Σ_{k∈N_i} f_{i,k}
8:      j ← choose a neighbor ∈ N_i uniformly at random
9:      f_{i,j} ← f_{i,j} − e_j/2
10:     send f_{i,j} to j
11:     for each received flow f_{j,i} from node j do
12:         if |f_{j,i}(1) + f_{j,i}(2) − f_{j,i}(3)| ≤ τ then
13:             f_{i,j} ← −f_{j,i}
14: y_i ← e_i(1)/e_i(2)
```

$f_{i,j}$ (Algorithm 5, lines 11 to 13), otherwise the received message is discarded. The local approximation y_i of the aggregate is computed like in Push-Flow (Algorithm 5, line 14).

Improved fault-tolerance

In theory (exact arithmetic), PFLC has the same fault-tolerance properties as Push-Flow. However, in practice (floating-point arithmetic) it can even tolerate flipping bits with high significance, which lead to large errors in magnitude. 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 between two nodes after resetting a corrupted flow variable will re-establish flow conservation between them.

Note that not every fault necessarily leads to a strong fall-back in convergence. For seeing this, let us assume that only a single node i experiences one bit-flip in the flow variable $f_{i,j}$. Depending on which node initiates the next communication between node i and j, we can distinguish two cases: If the zeroed flow variable $f_{i,j}$ overwrites $f_{j,i}$, strong fall-back in the convergence process is usually experienced. However, if flow $f_{j,i}$ overwrites zeroed flow variable $f_{i,j}$, then the fall-back in convergence depends on how many other nodes have been "infected" with the zeroed flow variable in the

meanwhile. Experience shows that in this case the fall-back in convergence is much milder.

3.2 Push-Flow with cooperative correction

Zeroing flows always leads to some step backward in convergence history. Hence it is useful to wait with setting flows to zero until it is absolutely necessary. Through the redundant character of flow variables it is in many cases possible that a correct flow overwrites a neighbors faulty one.

The idea behind *Push-Flow with cooperative correction* (*PFCC*) is to leave a faulty flow uncorrected and to send it to the according neighbor as early as possible, s.t. the neighbor can send back its correct flow. Only in the case that both flows are damaged, a node resets its flow to the initial value of zero. Thus many SDC errors can be corrected without loss of convergence.

Algorithm 6 Push-Flow with Cooperative Correction (PFCC)

Input: $(x_i, w_i, c_i),\ \varepsilon,\ \tau$
Initial values: $c_i = x_i + w_i,\ \forall\, k\ \in \mathcal{N}_i :\ f_{i,k} \leftarrow (0,0,0)$
Output: $y_i \approx \sum_{j=0:N-1} x_j \,/\, \sum_{j=0:N-1} w_j$

```
1:  while not ε-accurate do
2:      e_i ← (x_i, w_i, c_i) + Σ_{k∈N_i} f_{i,k}
3:      if |e_i(1) + e_i(2) − e_i(3)| > τ then
4:          for each k ∈ N_i do
5:              if |f_{i,k}(1) + f_{i,k}(2) − f_{i,k}(3)| > τ then
6:                  send f_{i,j} to k
7:      j ← choose a neighbor ∈ N_i uniformly at random
8:      f_{i,j} ← f_{i,j} − e_j/2
9:      send f_{i,j} to j
10:     for each received flow f_{j,i} from node j do
11:         if |f_{j,i}(1) + f_{j,i}(2) − f_{j,i}(3)| ≤ τ then
12:             f_{i,j} ← −f_{j,i}
13:         else
14:             if |f_{i,j}(1) + f_{i,j}(2) − f_{i,j}(3)| > τ then
15:                 f_{i,j} ← (0,0,0)
16:             send f_{i,j} to j
17: y_i ← e_i(1)/e_i(2)
```

Push-Flow with cooperative correction uses the same checksums as PFLC. It differs only in the way failures are handled. When an error in the estimate e_i is detected, the

algorithm checks every flow variable and sends the damaged flows to the according neighbors (Algorithm 6 line 3 to 6).

When receiving a faulty flow, it is additionally checked if the local flow is also damaged (Algorithm 6 line 14). Only in the case that both flows are faulty the local flow is set to zero (Algorithm 6 line 15). For all received faulty flows PFCC sends its current flow to the sender (Algorithm 6 line 16).

Improved fault-tolerance and convergence speed

Push-Flow with cooperative correction has the same fault-tolerance properties as *Push-Flow with local correction*. But in almost any case it can recover faster from failures than *Push-Flow with local correction*.

3.3 SDC beyond flow variables

So far we only described methods to recover from SDC in messages and flow variables. To some extent, it is also possible to repair the initial value-weight pair in case of data corruption. In case 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 $(x_i, w_i, c_i) = e_i - \sum_k f_{i,k}$.

4. Evaluation of fault-tolerance

The following experiments show first results on the effects of message loss and silent data corruption on the various all-to-all aggregation algorithms. We investigate in detail bit-flips in local data structures and their influence on the achieved accuracy and on the communication overhead, as well the effects of message loss on the runtime. In contrast to all results in the literature so far, the results are from *asynchronous* implementations, showing a much more realistic picture than strongly synchronized round-based implementations.

4.1 Experimental setup

We used two different types of simulations for our experiments. First, we simulated the given algorithms with a sequential Python-based discrete event simulator explained in detail in appendix A.3 and second, we implemented all algorithms in MPI [19] and run them on a parallel machine called Jupiter. The MPI implementations are described in appendix A.4.

Jupiter is a distributed memory machine with 36 nodes, each node has two 8 core AMD Opteron 6134 processors and 32 GB memory. In total it has 576 cores and 1152 GB memory. The nodes are connected via infiniband in a fully-connected network topology.

All algorithms were run with the same sequence of random numbers. Thus, in the fault-free case all Push-Flow algorithms show exactly the same results for the Python Discrete Event Simulator (Appendix A.3) and similar result for MPI implementations (Appendix A.4), where because of system noise runs are not always exactly reproducible.

The maximum relative error shown in the figures represents the maximum absolute difference between the current approximation of the aggregate y_i at node i and the true aggregate α over all nodes relative to the true result α:

$$\arg\max_i \frac{|y_i - \alpha|}{|\alpha|}, \ \alpha = \frac{\sum_k x_k^0}{\sum_k w_k^0} \tag{4.1}$$

For the Python simulations all algorithms were terminated once the local relative error was less than 10^{-14} for all nodes i:

$$\forall i = 0, \ldots, N-1 : \ \frac{|y_i - \alpha|}{|\alpha|} \leq \varepsilon = 10^{-14}, \ \alpha = \frac{\sum_k x_k^0}{\sum_k w_k^0}$$

Because of reasons described in appendix A.4, the MPI implementations stop when the first node reaches the target accuracy, thus it is not guaranteed that all nodes are reach the same accuracy. Experience showed, that the nodes relative error usually varies in three orders of magnitudes.

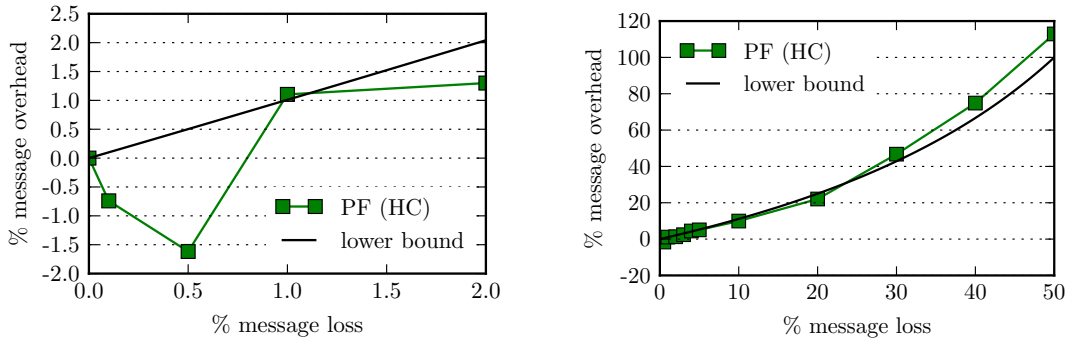
$$\exists i, \ \frac{|y_i - \alpha|}{|\alpha|} \leq \varepsilon = 10^{-14}, \ \alpha = \frac{\sum_k x_k^0}{\sum_k w_k^0}$$

Alternatively, the algorithms were stopped once the first node has sent 500 messages. The threshold τ for the check-sums was set to a value of 10^{-11}.

4.2 Effects of message loss

All Push-Flow based algorithms, including PFLC and PFCC, use the concept of flows, therefore they can handle message loss equally well [21]. Thus, we show only results for Push-Flow, as the runtimes and messages needed until convergence are for all Push-Flow based algorithms nearly the same.

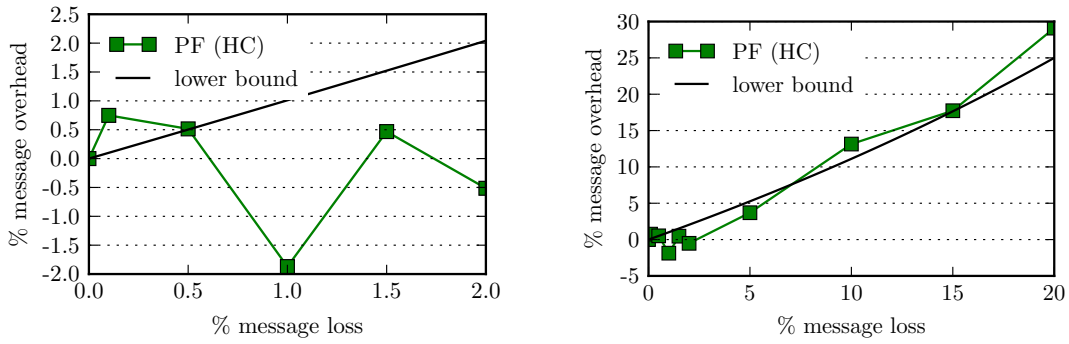
We simulated message loss by choosing a value between 0 and 1 uniformly at random for each received message and dropped the message if the value was below a certain loss rate p. When assumed, that for each lost message at least one successful message must be transmitted, the number of sent messages in case of loss rate p must be at least a factor of $1/(1-p)$ of the number of sent messages in the failure free case. Therefore, the lower bound on message overhead for message loss is $1/(p-1)$.



(a) message loss up to 2% **(b)** message loss up to 50%

Figure 4.1: Push-Flow, message overhead in % induced by message loss (average over 20 runs each point, Python Discrete Event Simulator (Appendix A.3), 32 nodes, hypercube communication topology, $\varepsilon = 10^{-14}$)

When simulated with our Python Discrete Event Simulator for 32 nodes, as given in Figure 4.1, the overhead on messages for message loss was nearly the lower bound on messages. For drop rates up to 2% the natural variation of Push-Flow is even stronger than the effects of message loss. Only for higher loss rates above 20%, the overhead was slightly above the lower bound of additional messages.

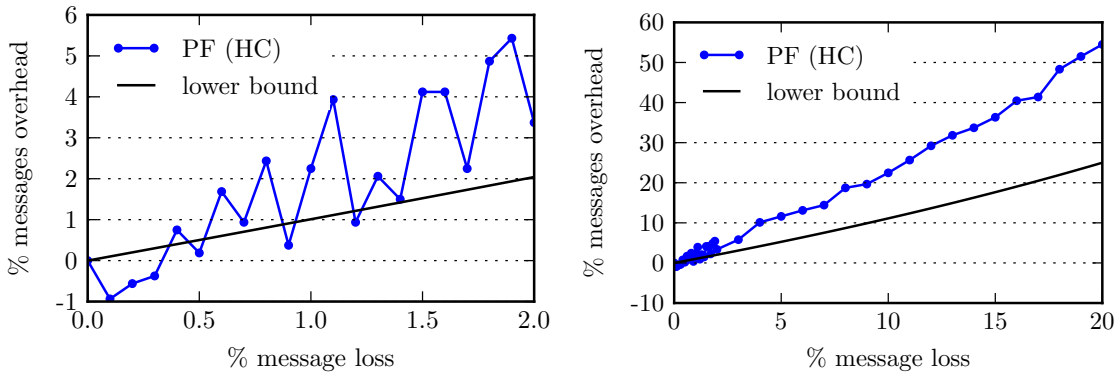


(a) message loss up to 2% **(b)** message loss up to 20%

Figure 4.2: Push-Flow, message overhead in % induced by message loss (average over 5 runs each point, Python Discrete Event Simulator (Appendix A.3), 512 nodes, hypercube communication topology, $\varepsilon = 10^{-11}$)

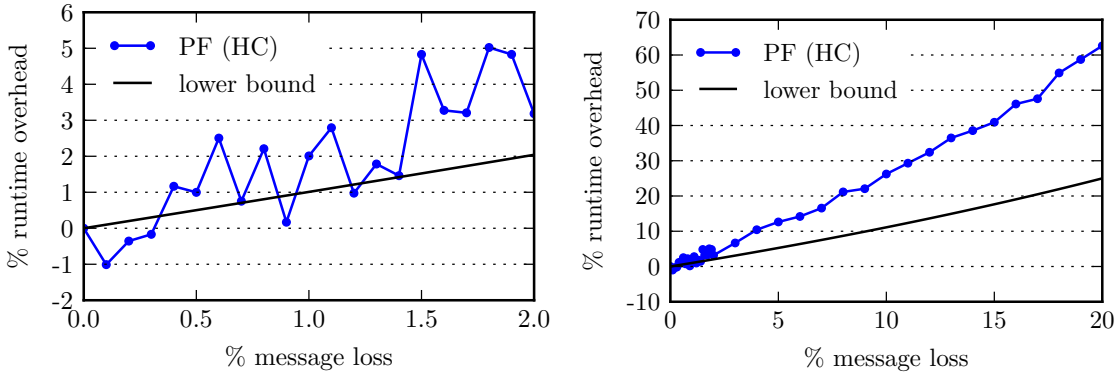
On 512 nodes (Figure 4.2) the results for our Python Discrete Event Simulator are similar to 32 nodes. On Jupiter for 512 cores the overhead grew a lot faster than the theoretical lower bound. Push-Flow on Jupiter with 20% loss rate needed about double the amount of additional messages as the minimum overhead given by the lower bound (Figure 4.3b).

As with our Python Discrete Event Simulator, on Jupiter the overhead for small loss rates varies a lot (Figure 4.3a), but in contrast to the results from our Python Discrete Event Simulator most of the measured points are above the lower bound.



(a) message loss up to 2% **(b)** message loss up to 20%

Figure 4.3: Push-Flow, message overhead in % induced by message loss (median over 100 runs each point, MPI implementation (Appendix A.4), Jupiter 512 processes, hypercube communication topology, $\varepsilon = 10^{-14}$)



(a) message loss up to 2% **(b)** message loss up to 20%

Figure 4.4: Push-Flow, runtime overhead in % induced by message loss (median over 100 runs each point, MPI implementation (Appendix A.4), Jupiter 512 processes, hypercube communication topology, $\varepsilon = 10^{-14}$)

For Jupiter we can additionally give actual results on runtime overhead. As expected the Figures 4.4a and 4.4b show nearly the same results on runtime overhead as for message overhead. Only for higher message loss rates the overhead on runtime was a bit higher as for messages (Figure 4.4b).

Overall, Push-Flow, PFLC and PFCC can handle arbitrary high message loss rates. For realistic loss rates ($\leq 1\%$) the overhead is negligible.

4.3 Error induced by silent data corruption

All discussed algorithms are differently disturbed by silent data corruption. Push-Sum cannot recover from silent data corruption at all. Whereas, Push-Flow converges in general to an equilibrium of the flows and therefore recovers from any disturbance in the flow variables. But, as mentioned earlier, due to floating-point arithmetic, Push-Flow cannot handle silent data corruption that causes a huge error in a flow variable. Push-Cancel-Flow also uses the concept of flows, but cannot tolerate any silent data corruption, because of the usage of an intermediate aggregation variable. To prevent erroneous flows to be added to the estimate, PFLC and PFCC use check-sums and therefore extend the resilience of Push-Flow to any silent data corruption in floating point arithmetic.

4.3.1 Convergence behavior with silent data corruption

Figure 4.5 compares the convergence history of the Push-Flow algorithms in case of a single bit-flip at time $t_{\text{bit-flip}}$. In the failure-free case (no bit-flip), all three

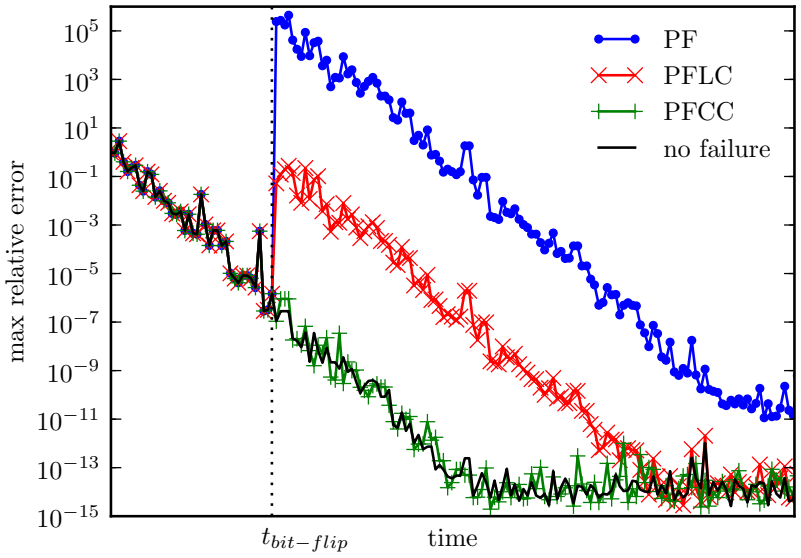


Figure 4.5: Convergence behavior of different Push-Flow algorithms in case of a single bit-flip (Python Discrete Event Simulator (Appendix A.3), 32 nodes, hypercube communication topology)

algorithms show exactly the same convergence behavior. For each algorithm the 56th bit in a randomly chosen flow variable was flipped at time $t_{\text{bit-flip}}$. We see that basic Push-Flow first falls back in convergence and later does not converge to the true aggregate as the round-off error caused by the bit-flip when computing the aggregate was too large. In contrast, both PFLC and PFCC converge to true result,

although at different speeds. PFLC resets the corrupted flow variable to zero and thus also experiences some fall-back in convergence. However, in contrast to basic Push-Flow, in the worst case the fall-back is only to the initial estimate, and, as the failure is never propagated, it always converges to right result. PFCC waits for the next message from the neighbor associated with the erroneous flow to overwrite a corrupted flow variable. Thus, PFCC recovers much faster from the failure and experiences nearly no slow-down over the failure-free case.

4.3.2 Influence of bit position

The next experiment illustrates the influence of a bit-flip on all possible positions of an IEEE 754 floating-point variable in all aggregation algorithms described. For Push-Sum the error is applied to a value-weight pair, for all Push-Flow algorithms a flow variable is disturbed and for recursive doubling a bit in the initial data is flipped.

All simulations show the maximum relative error over all nodes over 100 runs. The results are from our Python Discrete Event Simulator (Appendix A.3) using 32 nodes. For gossip-based algorithms a hypercube communication schedule is used in combination with a target accuracy of $\varepsilon = 10^{-14}$.

Recursive doubling

We injected bit-flips into the data of one node before the first message got sent, so that the error propagated to all other nodes.

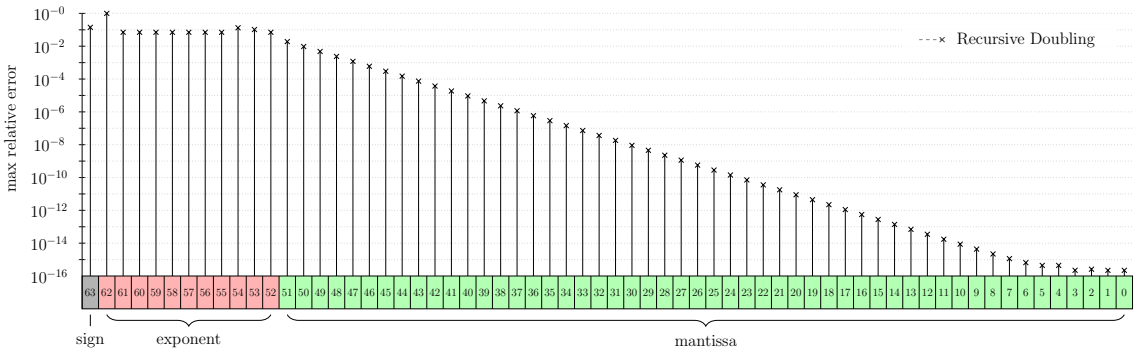


Figure 4.6: Recursive doubling: Maximum relative error over all nodes induced by single bit-flips on different positions in floating-point variable holding the reduction data (maximum over 100 runs each point, Python Discrete Event Simulator (Appendix A.3), 32 nodes)

As anticipated, recursive doubling without any additional measures is vulnerable to silent data corruption (Figure 4.6). For a single bit-flip the overall error mainly

depends on the position of the flipped bit in the data. The impact of bit-flips in the exponent and sign bit of a floating-point variable is always severe, whereas depending on the application of RDB, lower flipped bits in the mantissa may be tolerated.

Push-Sum

The first interesting observation was that Push-Sum cannot handle silent data corruption at all (Figure 4.7), in fact, one gets nearly the same maximum relative errors when flipping a bit in a single floating-point variable of the recursive doubling algorithm (Figure 4.6).

We flipped bits for both, the value and weight variable of the nodes. For each bit-flip we randomly chose a node when the first node has sent more than 150 messages and either flipped a bit on the given position in the value or weight variable for that node. It shows that there is nearly no difference between bit-flips in the value or weight variables of Push-Sum, both affect the final result equally.

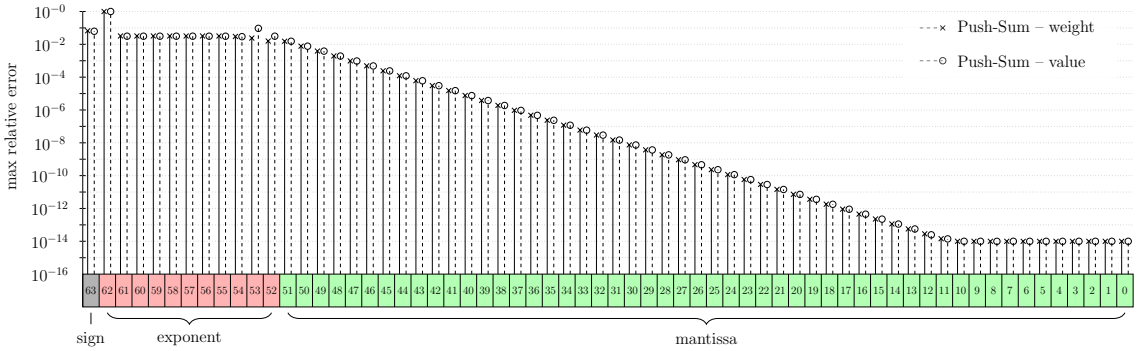


Figure 4.7: Push-Sum: Maximum relative error over all nodes induced by single bit-flips on different positions in floating-point variables holding the value or weight (maximum over 100 runs each point, Python Discrete Event Simulator (Appendix A.3), 32 nodes, hypercube communication topology, $\varepsilon = 10^{-14}$)

Push-Flow

Again, bits are flipped when the first node sent more than 150 messages. But now, as Push-Flow holds a flow variable for each neighbor node, additionally to the random node a random flow variable was chosen to flip bits in.

It shows that Push-Flow can recover from all bit-flips in the sign and mantissa bits as the introduced error is relatively small, but bit-flips in the exponent of a floating-point number can be fatal for Push-Flow (Figure 4.8).

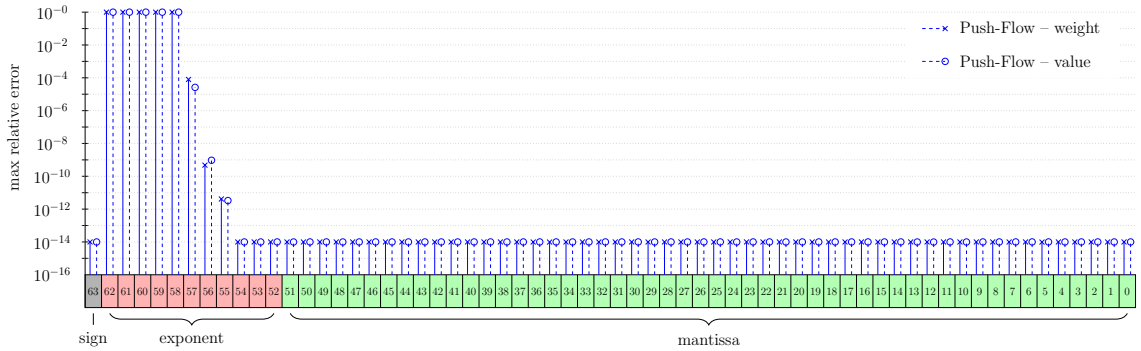


Figure 4.8: Push-Flow: Maximum relative error over all nodes induced by single bit-flips on different positions in floating-point variables holding the value or weight (maximum over 100 runs each point, Python Discrete Event Simulator (Appendix A.3), 32 nodes, hypercube communication topology, $\varepsilon = 10^{-14}$)

Push-Cancel-Flow

The setup for PCF was similar to Push-Flow. Contrary to the results from [22], Push-Cancel-Flow can not handle silent data corruption at all. The results are similar to Push-Sum and recursive doubling. Therefore further measures must be taken to make Push-Cancel-Flow resilient to silent data corruption.

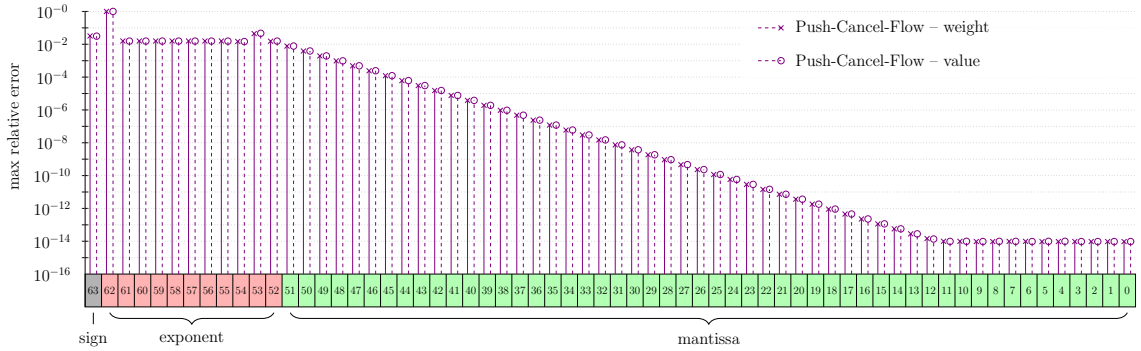


Figure 4.9: Push-Cancel-Flow: Maximum relative error over all nodes induced by single bit-flips on different positions in floating-point variables holding the value or weight (maximum over 100 runs each point, Python Discrete Event Simulator (Appendix A.3), 32 nodes, hypercube communication topology, $\varepsilon = 10^{-14}$)

Push-Flow with check-sums

Our novel algorithms PFLC (Figure 4.10) and PFCC (Figure 4.11) recover from any bit-flip in the local flow data structures. We do not include the error for bit-flips in check-sums, the results from our experiments showed that their influence is equal to the error in the value or weight of flow variables.

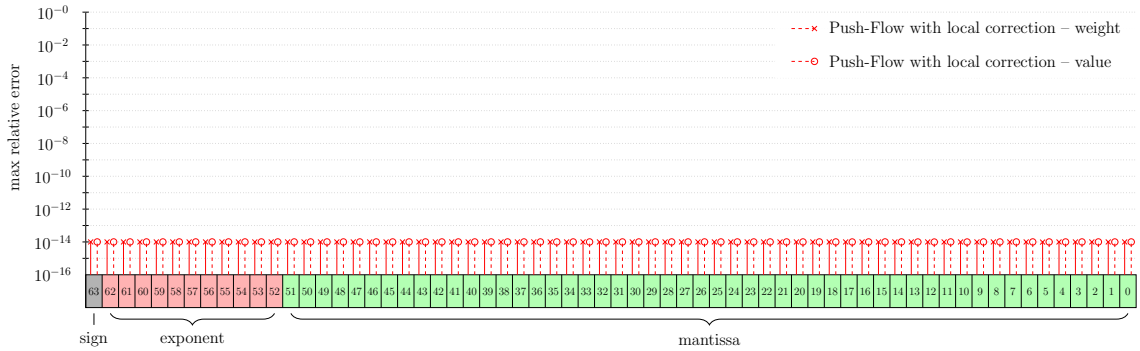


Figure 4.10: PFLC: Maximum relative error over all nodes induced by single bit-flips on different positions in floating-point variables holding the value or weight (maximum over 100 runs each point, Python Discrete Event Simulator (Appendix A.3), 32 nodes, hypercube communication topology, $\varepsilon = 10^{-14}$)

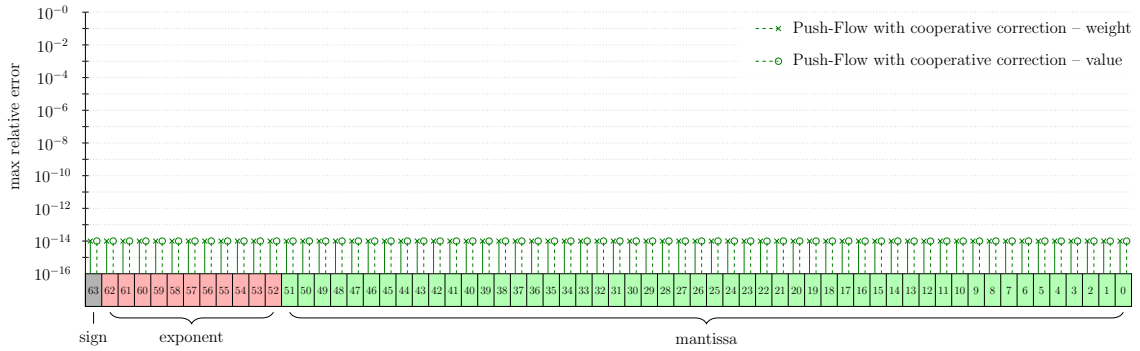


Figure 4.11: PFCC: Maximum relative error over all nodes induced by single bit-flips on different positions in floating-point variables holding the value or weight (maximum over 100 runs each point, Python Discrete Event Simulator (Appendix A.3), 32 nodes, hypercube communication topology, $\varepsilon = 10^{-14}$)

Overall, only our new algorithms PFLC and PFCC are resilient against silent data corruption in local floating-point variables. For all other algorithms described, silent data corruption can have severe impact on the achieved accuracy and are therefore not fault-tolerant against silent data corruption (Figure 4.12 shows results for values only).

4.3.3 Message overhead

In Figure 4.13, we illustrate the communication overhead of the increased fault tolerance by comparing the mean of sent messages per node of Push-Sum without bit-flips with the mean of sent messages of our Push-Flow based algorithms with bit-flips on different positions in a flow variable over 100 runs. Due to the fact that in Push-Flow simultaneous messages between two neighbors can overwrite each others

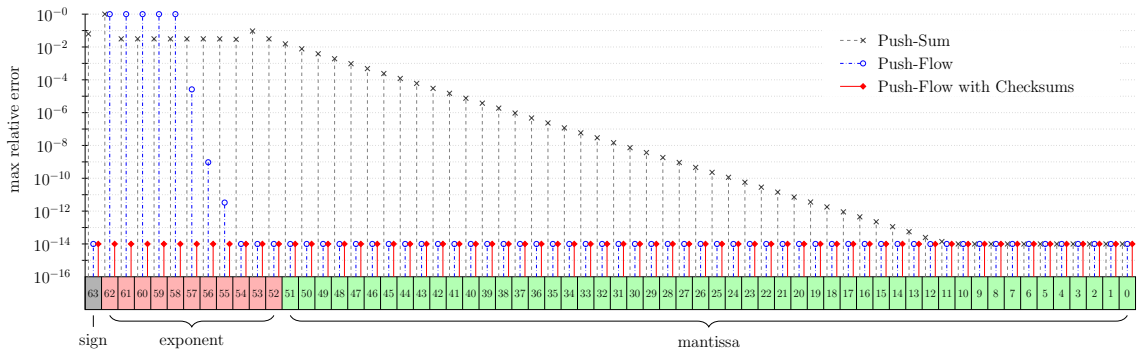


Figure 4.12: Maximum relative error over all nodes depending on the position of the bit-flip (maximum over 100 runs each point, Python Discrete Event Simulator (Appendix A.3), 32 nodes, hypercube communication topology, $\varepsilon = 10^{-14}$)

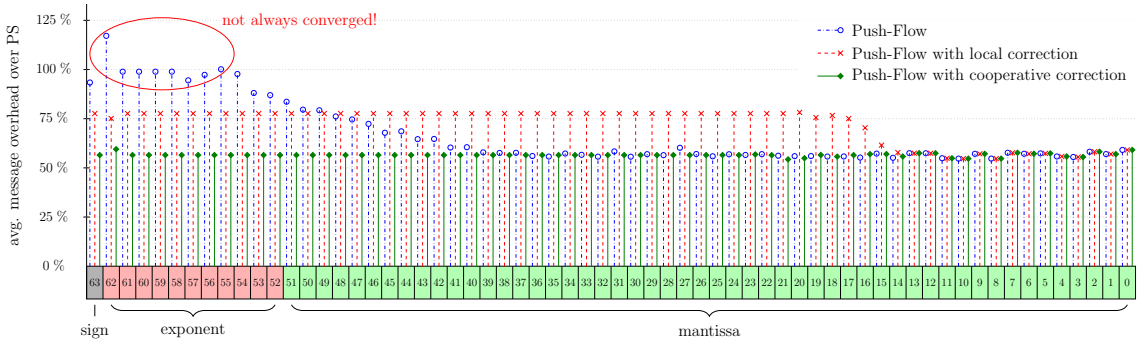


Figure 4.13: Overhead induced by bit-flips in terms of messages compared to Push-Sum (maximum over 100 runs each point, Python Discrete Event Simulator (Appendix A.3), 32 nodes, hypercube communication topology, $\varepsilon = 10^{-14}$)

flow and therefore one of the messages is lost, Push-Flow needs more messages than Push-Sum (see section 2.2.2).

We see that up to the 13th bit there is no difference in the runtime of all Push-Flow based algorithms. Beginning with the 14th bit PFLC needs constantly more messages, as flows are set to zero and therefore PFLC falls back in convergence. Push-Flow and PFCC need roughly the same amount of messages up to the 39th bit, afterwards PFs message count rises. As of bit 55, Push-Flow does not always converge to the true aggregate, thus the mean also includes runs were the maximum message count of 500 was reached. Only PFCC has a nearly constant message count independently of the position of the bit-flip. It is clear that in this example parameter τ was chosen too strictly for PFLC as Push-Flow without correction need less overall messages. The ideal value of τ depends on the range of the input data, and it one of the open questions how to choose τ based on the current estimate.

Note that all results from Figures 4.12 and 4.13 also apply to bit-flips in messages, since then the failure has an impact once the message is written to the flow variable.

4.3.4 Message overhead for more than one bit-flip

To investigate the influence of more than one bit-flip on the message overhead of our fault-tolerant algorithms PFLC and PFCC, we run 500 simulations of each algorithm with our Python Discrete Event Simulator using a fixed flip rate of 0.0001 and maximum number of sent messages set to 1000. The experiments were done for 32 nodes with a hypercube communication topology.

Figure 4.14 shows the result grouped by number of flipped-bits. Note that only successfully converged runs are included, where the number of sent messages was below 1000. It shows that PFCC is in all cases faster than PFLC and PFLC overhead grows significantly by number of bit-flips, whereas for PFCC no difference in messages can be seen.

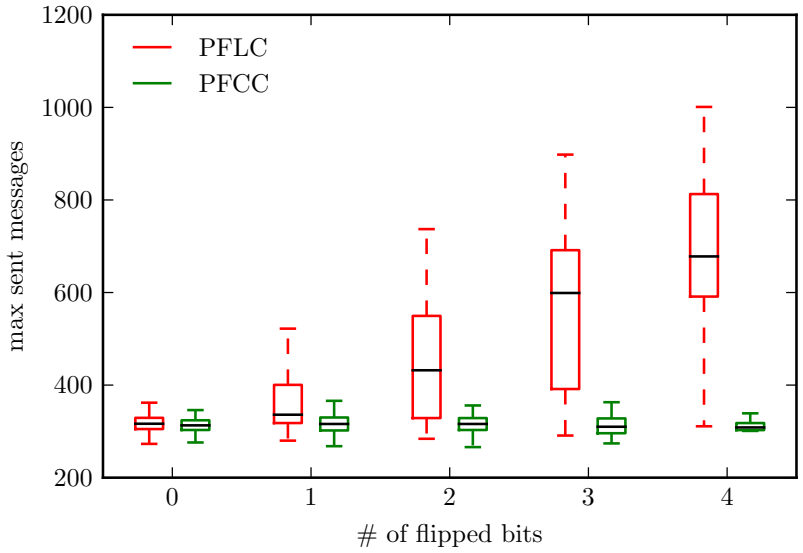


Figure 4.14: Message overhead for number of bit-flips per run (whiskers show 1.5*IQR, 500 runs, Python Discrete Event Simulator (Appendix A.3), 32 nodes, hypercube communication topology, $\varepsilon = 10^{-14}$)

4.4 Open questions

- How can concurrent messages between two nodes in Push-Flow algorithms be avoided?
- Can Push-Cancel-Flow be fixed?
- How must τ be chosen in PFLC and PFCC?
- Can silent data corruption be tolerated in other than flow variables?

5. Scalability of gossip-algorithms

In theory all considered algorithms have the same *asymptotic order* in number of messages on fully-connected topologies. More specifically, Recursive Doubling (section 2.2.1), Push-Sum (section 2.2.1), Push-Flow (section 2.2.2), PFLC (section 3.1) and PFCC (section 3.2) all require $\mathcal{O}(\log n)$ sent messages per node until convergence and yet in practice huge differences in number of messages between deterministic parallel reduction algorithms and gossip-based aggregation algorithms occur. Furthermore, as we focus on the application of gossip algorithms on HPC-Systems, the only criteria for an algorithm is the performance compared to state-of-the-art implementations. One state-of-the-art technique to achieve fault-tolerance against bit flips is triple temporal redundancy. Where a non-fault-tolerant algorithm is run three times in a row and a majority vote on the result is made. Thus, in order to show the feasibility of gossip-based aggregation algorithms in the context of high performance computing, one has to show that the overhead introduced by the improved resilience of gossip algorithms is at most three times the runtime of non-fault-tolerant deterministic algorithms.

On HPC-systems every node can communicate which every other node. Communication is usually realized over switched networks, where a node has no neighborhood as assumed by gossip algorithms. Nevertheless, communication can be more or less expensive depending on the physical distance between two computing nodes in a cluster. Therefore, we distinguish between the physical network topology from a HPC-System and the communication topology induced by the communication patterns of gossip algorithms.

Throughout this chapter we focus on Push-Sum for our analysis, as it is the common ancestor of all our gossip-based aggregation algorithms and in the failure free case all algorithms have the same asymptotic runtime complexity.

First we compare gossip-based aggregation with deterministic parallel all-reduce algorithms in number of iterations needed. Later we investigate the additional cost of communicating on hardware topologies other than fully-connected. Finally, we show results from the Vienna Scientific Cluster 2 and complete this chapter by predicting the runtime of Push-Sum on the Vienna Scientific Cluster 2 with model functions derived from our simulation results.

5.1 Communication cost of all-reduce

Cost for parallel all-to-to-all reductions are well understood for fully-connected systems. Bounds on the cost can be derived by simple thoughts. In the following we summarize the bounds of all-reduce and give the cost for two popular all-reduce algorithms.

5.1.1 Bounds of parallel all-reduce cost

A *lower bound on messages* needed to do an parallel all-reduce operation can be derived by the fact that each node has some information needed by every other node. In each iteration a node can at most send one message, thus information can only be doubled in each iteration, therefore the *lower bound on messages* is $\lceil \log_2 N \rceil$.

Computation on a single process takes $(N-1)$ operations, when perfectly distributed on all processes, *computational cost* are at least $\frac{N-1}{N}$ times the cost for one local operation $\otimes$.

The cost to transfer the data can be derived from the lower bound of computational cost. To complete 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. Giving a minimal cost of $2\frac{N-1}{N}$ times the cost for transferring a single data item [26, 5].

5.1.2 Cost of sending a message

The cost of sending a message of size l can be modeled as $\alpha + l\beta$. There is a constant latency term α for initializing a communication between two nodes, plus a message size dependent term $l\beta$ for transmitting the message. β denotes the time needed to transmit one data unit. Usually for small messages the latency dominates the overall cost of communication, whereas for large messages the latency can be neglected and the bandwidth term $l\beta$ dominates the overall communication time. Thus for parallel reduction algorithms it is important to, depending on the message size, either keep the number of messages small or to send only the minimal amount of data needed, to minimize the overall transmission cost.

5.1.3 Cost of state-of-the-art all-reduce implementations

Currently there exists no algorithm that is optimal in all three types of cost. On the one hand there are algorithms optimal in the number of sent messages, where in each iteration all nodes send their hole local data set to some other node, as recursive doubling, which cost are $\log_2 N \cdot (\alpha + l\beta + l\gamma)$. γ denotes the cost for

one local operation. On the other hand there exist some algorithms optimal in the amount of sent data and number of operations, where the data set is split and operations are only applied to parts of it, accepting an overhead on sent messages, like Rabenseifner's algorithm $(2\log_2 N \cdot \alpha + 2\frac{N-1}{N} l\beta + \frac{N-1}{N} l\gamma)$ [26]. Thus in HPC environments it is common to switch between algorithms depending on message size [26, 5].

5.2 Asymptotic number of iterations for Push-Sum

Obviously, the bounds on parallel all-to-all reduction also holds for stochastic algorithms like Push-Sum. Due to the fact that Push-Sum in every iteration sends all local data to some other node, it cannot reach optimality in terms of overall sent data and number of operations. From this observation it follows that gossip-based all-aggregation is not suitable for large data sets in high-performance computing.

An advantage of gossip-based algorithms over deterministic algorithms is, that they can easily leverage increased performance of local communication without knowing the overall network topology, which makes them simple to adapt to new network architectures. But local only communication comes with some additional cost on number of iterations depending on the connectivity of the communication.

Based on the theoretical runtime complexity of Push-Sum [2]

$$\mathcal{O}\left(\frac{\log N + \log \varepsilon^{-1}}{1 - \lambda_2}\right) \tag{5.1}$$

the number of iterations needed until convergence to an ε-accurate result for different network topologies of size N can be modeled. Note that $1 - \lambda_2$ denotes the *spectral gap*, which is given by the difference between the largest and the second largest eigenvalue of the adjacency matrix of a graph. In the special case of Push-Sums communication matrix in expectation $\mathcal{C}$, also called push-distribution [20], the largest eigenvalue is always 1. The larger the spectral gap, the better connected is a graph. Thus, the connectivity of a graph has a strong influence on the number of iterations needed until convergence of gossip-based algorithms.

5.2.1 Push-distribution

To better understand the iterations needed until convergence of gossip-based aggregation for different communication topologies it is important to know the spectral gap and therefore the second largest eigenvalue of the push-distribution.

The Push-Sum algorithm can also be formulated as linear iterations

$$x(t) = C(t)x(t-1), \quad w(t) = C(t)w(t-1)$$

where x and w are the value and weight of Push-Sum and matrices $C(t)$ are drawn i.i.d. from the push-distribution $\mathcal{C}$. A node i in Push-Sum sends in each iteration t half of its current value-weight pair to a random neighbor in $\mathcal{N}_i$ and the other half to itself, therefore in expectation the communication matrix of Push-Sum is

$$\mathcal{C} = \mathbb{E}[C(t)] = \frac{1}{2}\mathcal{N}A + \frac{1}{2}I$$

$\mathcal{N}$ is a diagonal matrix with the reciprocal number of neighbors for each node $(|\mathcal{N}_0|^{-1}, \ldots, |\mathcal{N}_{N-1}|^{-1})$ as entries. A denotes the adjacency matrix of the graph induced by the communication schedule and I denotes the identity matrix.

5.2.2 Spectral gap and asymptotic cost for several push-distributions

From the push-distribution we can compute eigenvalues and derive the asymptotic cost of Push-Sum for different communication topologies. In the following we focus on three different types of topologies, we analyze the asymptotic cost of Push-Sum on fully-connected, hypercube and different tori graphs.

Fully-Connected

The push-distribution for a fully-connected communication schedule, where each node can communicate with every other node, has a nearly constant spectral gap. The eigenvalues of the adjacency matrix of a fully-connected (complete) graph without loops of size N are $(N-1, -1, \ldots, -1) \in \mathbb{R}^N$ [23], thus the second largest eigenvalue λ_2 is -1. In a fully-connected graph each node has $N-1$ neighbors, the push-distribution is therefore

$$\mathcal{C} = \mathbb{E}[C(t)] = \frac{1}{2(N-1)}A + \frac{1}{2}I$$

The second largest eigenvalue λ_2 of the push-distribution $\mathcal{C}$ is

$$\lambda_2 = \frac{1}{2(N-1)} \cdot -1 + \frac{1}{2} \cdot 1 = \frac{N-2}{2N-2}$$

λ_2 of A gets scaled by a half and the reciprocal number of neighbors for each node and then shifted by a half.

The spectral gap of the communication matrix in expectation for a fully-connected topology is

$$1 - \lambda_2 = 1 - \frac{N-2}{2N-2} = \frac{N}{2N-2}$$

With growing N the spectral gap gets $\approx 1/2$.

$$\lim_{N\to\infty} \frac{N}{2N-2} = \frac{1}{2}$$

Thus, the asymptotic number of iterations needed until convergence are

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

Hypercube

The eigenvalues of the adjacency matrix of a hypercube graph with $N = 2^d$ nodes are $(d, d-2, d-4, ..., -d+4, -d+2, -d)$, every k-th eigenvalue has multiplicity of $\binom{d}{k}$ [23]. The second largest eigenvalue of a hypercube graph is $d-2$. Each node in a hypercube has $d = \log_2 N$ neighbors. The Push-distribution is

$$\mathcal{C} = \mathbb{E}[C(t)] = \frac{1}{2d}A + \frac{1}{2}I$$

and the second largest eigenvalue λ_2 of $\mathcal{C}$ is

$$\lambda_2 = \frac{1}{2d} \cdot (d-2) + \frac{1}{2} \cdot 1 = \frac{d-1}{d}$$

The spectral gap of the hypercube push-distribution is

$$1 - \lambda_2 = 1 - \frac{d-1}{d} = \frac{1}{d} = \frac{1}{\log_2 N}$$

Therefore, the asymptotic cost for Push-Sum on a hypercube are

$$\mathcal{O}\left(\log_2 N \cdot (\log N + \log \varepsilon^{-1})\right) = \mathcal{O}\left(\log^2 N\right)$$

Tori

The second largest eigenvalue of the adjacency matrix of a d-dimensional torus with n nodes in each dimension [23], where each node has $2d$ neighbors, is

$$\lambda_2 = 2(d-1) + 2\cos\frac{2\pi}{n}$$

λ_2 of the push-distribution is

$$\lambda_2 = 1 + \frac{\cos\frac{2\pi}{n} - 1}{2d}$$

the spectral gap for a d-dimensional torus with n nodes in each dimension is

$$1 - \lambda_2 = \frac{1 - \cos\frac{2\pi}{n}}{2d}$$

and the asymptotic iterations needed are

$$\mathcal{O}\left(\frac{2d}{1-\cos\left(\frac{2\pi}{n}\right)} \cdot (\log N + \log \varepsilon^{-1})\right)$$

5.2.3 Asymptotic growth of cost

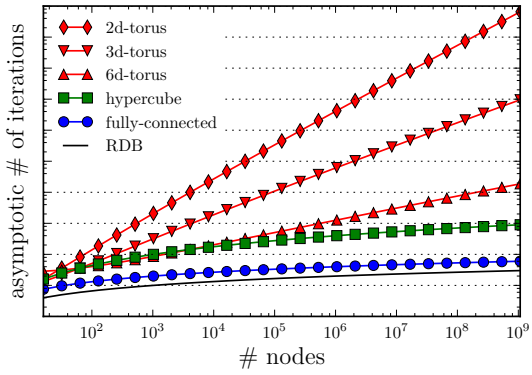


Figure 5.1: Asymptotic number iterations for different communication topologies of Push-Sum compared to recursive doubling

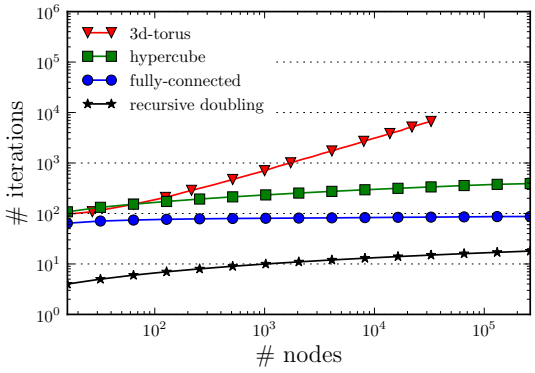


Figure 5.2: Number of iterations for synchronous Push-Sum until convergence (C-based Push-Sum simulator (Appendix A.1), $\varepsilon = 10^{-14}$)

In Figure 5.1 the asymptotic cost on iterations of Push-Sum for the discussed communication topologies are plotted. As mentioned before, Push-Sum on a fully-connected topology scales theoretically equally well as recursive doubling, whereas

the asymptotic growth on all tori topologies is nearly linear and therefore not competitive. Push-Sum on a hypercube asymptotically needs $\mathcal{O}(\log^2(x))$ iterations and scales reasonable well, but worse than recursive doubling.

Solely by the asymptotic runtime complexity of Push-Sum it is not possible to make assumptions about the concrete message overhead in comparison to parallel reduction algorithms. Therefore, results from simulations are necessary.

5.3 Cost models for Push-Sum

There are two possible ways to implement gossip algorithms, synchronized with global iterations and asynchronous with local iterations only. See appendix A for further details. Whether the two versions differ in overall number of sent messages and thus in runtime was not investigated so far. Thus, we compare results from c-based synchronous Push-Sum simulator with asynchronous results from our Python Discrete Event Simulator. As we are limited in resources and we would like to make prognoses for very large systems, we try to fit cost model functions upon the gathered simulation results.

5.3.1 Results for synchronous Push-Sum

Figure 5.2 shows the results from our synchronous implementation of Push-Sum (Appendix A.1), where the target accuracy ε was 10^{-14}. Each point represents the average on iterations needed until convergence over 100 repetitions. Compared to the asymptotic cost in Figure 5.1 the curves mainly differ in an offset on the y axis for the Push-Sum algorithm. As expected, this is due to the additional iterations $\mathcal{O}(\log \varepsilon^{-1})$ Push-Sum needs, independently of the network size, to reach the target accuracy ε.

Before we compare the results of Push-Sum with recursive doubling in details, we fit model functions on the gathered iteration counts. This allows us to infinitely scale the problem size and to compare algorithm cost solely based on functions.

5.3.2 Model on iterations for synchronous Push-Sum

It appears natural, when modeling the cost of Push-Sum to use the asymptotic cost function (5.1) as a foundation. Therefore, we added linear factor and constant variables to the asymptotic cost functions of the different topologies and fitted them with Matlabs curve fitting tool `cftool` on results gathered from our simulations.

In particular our cost model function for Push-Sum was

$$f = a \cdot \frac{\log_2(N+b)}{1-\lambda_2} + c \tag{5.2}$$

In Table 5.1 the fitted coefficients for the functions are given.

a	0.5442
b	-15.9551
c	68.6655

(a) fully-connected

a	0.9064
b	-15.9920
c	133.1687

(b) hypercube

a	0.9327
b	8679000
c	28.88

(c) 3d-torus

Table 5.1: Coefficients for synchronous Push-Sum cost model

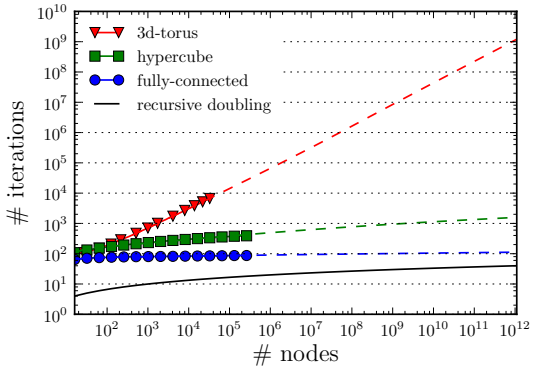


Figure 5.3: Simulation results and fitted model functions for number of iterations on different communication topologies for synchronous Push-Sum (C-based Push-Sum simulator (Appendix A.1), $\varepsilon = 10^{-14}$)

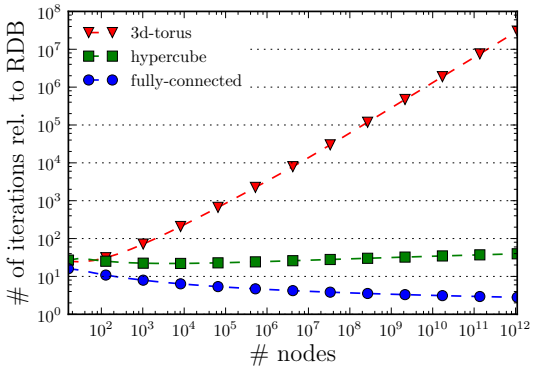


Figure 5.4: Fitted model functions for number of iterations of synchronous Push-Sum relative to iterations of recursive doubling ($\varepsilon = 10^{-14}$)

Figure 5.3, where the fitted model functions in combination with the simulation results for synchronous Push-Sum are plotted, shows that our cost functions fit the simulation data perfectly well. Thus, we assume using the function for predicting cost is a sound method. In particular the r-squared value for fully-connected topology is 0.92, for hypercube 0.98 and for 3d-torus it is 0.99.

The fitted functions relative to recursive doubling in Figure 5.4 show a nearly constant overhead between Push-Sum on a hypercube topology and recursive doubling, whereas for fully-connected topology the overhead decreases. From the theoretical bound of all-reduce we know that recursive doubling is optimal in number of iterations needed to compute an all-to-all reduction. Nevertheless, the fitted model function on iterations of fully-connected Push-Sum crosses the cost function of recursive doubling at a scale fare beyond on what is shown in Figure 5.4. Thus, this phenomenon has to come from the distribution of the input data. Maybe for

extremely huge node sizes it is sufficient to only sample parts of the input data. How exactly the distribution of the input values influences the number of iterations needed until convergence of Push-Sum is still an open question and should be further investigated.

5.3.3 Model on max messages for asynchronous Push-Sum

Next we investigate the communication cost of asynchronous Push-Sum. As with our HPC implementations of Push-Sum there is always a termination overhead included in our measurements, we developed an asynchronous simulator for Push-Sum in python (Python Discrete Event Simulator), where we stop the computation when all nodes reached the target accuracy ε.

With asynchronous Push-Sum each node independently of each other node emits messages. Clearly there is no overall number of iterations. Thus, some metric on sent messages per node has to be used for quantifying the overall communication cost. We use the maximum number of sent messages over all nodes as cost metric, as it gives a lower bound on runtime by the fact that it takes at least number of messages times the time needed to transmit a message to finish.

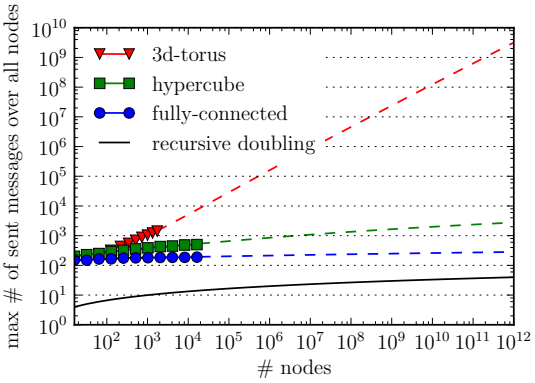


Figure 5.5: Simulation results and fitted model functions for maximum number of sent messages on different communication topologies for asynchronous Push-Sum (Python Discrete Event Simulator (Appendix A.3) , $\varepsilon = 10^{-14}$)

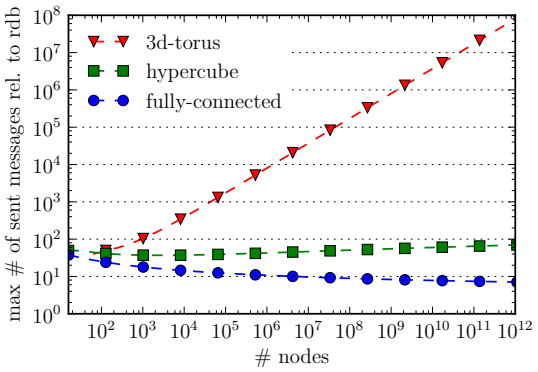


Figure 5.6: Fitted model functions for maximum number of sent messages of asynchronous Push-Sum relative to iterations of recursive doubling ($\varepsilon = 10^{-14}$)

Similar to our performance model for synchronous Push-Sum we used function (5.2) to fit the average on max number of messages from 100 runs for each data point. Again the r-square (fully-connected: 0.84, hypercube: 0.98, 3d-torus: 0.99) values show a high confidence in the fits. On the first glimpse, the graphs for synchronous

and asynchronous Push-Sum look identical. The first thing getting noticed in Figure 5.5 and 5.6 is a constant overhead of maximum sent messages over all node counts compared to synchronous Push-Sum (Figure 5.3). Not as obviously seen from Figure 5.5, but easily from Figure 5.7 and in Table 5.2, is that the linear factors a for all communication schedules in contrast to synchronous Push-Sum are greater than 1. Thus, the model functions for asynchronous Push-Sum cannot cross the cost function of recursive doubling.

a	1.7359
b	-12.7597
c	144.5031

(a) fully-connected

a	1.6106
b	-15.7585
c	211.4465

(b) hypercube

a	2.6268
b	-60.1879
c	207.9838

(c) 3d-torus

Table 5.2: Coefficients for asynchronous Push-Sum cost model

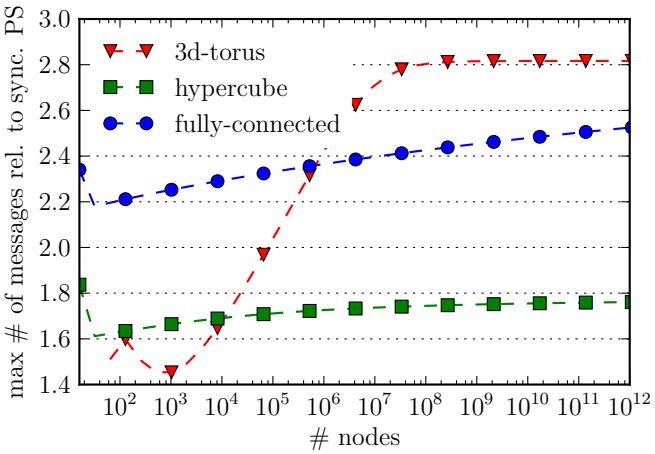


Figure 5.7: Number of messages for asynchronous Push-Sum relative to synchronous Push-Sum (fitted models functions, $\varepsilon = 10^{-14}$)

The results so far showed that Push-Sum is only scaling similarly to parallel all-reduce algorithms, when using highly connected communication topology as fully-connected or hypercube. It also shows that the theoretical asymptotic cost fit the results from our simulations perfectly well, thus when investigating new communication topologies it should be sufficient to look at the asymptotic cost function first to see if the scaling is competitive, before considering spending further effort in simulations.

5.4 Distance dependent cost model

So far we only compared message counts of Push-Sum with recursive doubling. This is sufficient to quantify the performance overhead for gossip-based algorithms when the communications cost between all pairs of nodes in a computing system are the same. As it is the case for fully-connected parallel computing systems. But on current HPC systems where the actual network is seldom fully-connected, network topologies have a huge influence on the communication cost. Therefore, a cost model which takes the additional cost into account is needed.

To show if neighborhood only communication for Push-Sum pays off, we developed a cost model where a 3d-torus network topology is assumed and the actual cost for each iteration are the maximum distances messages got sent. This can be seen as one additional iteration for each hop a message takes along a path in the network graph. The node Ids on which the neighbor selection for the different communication topologies depend, were assigned randomly on the torus graph to prevent that some communication topologies fit the underlying network topology better than others. Except for 3d-torus communication, where we assumed neighborhood only communication.

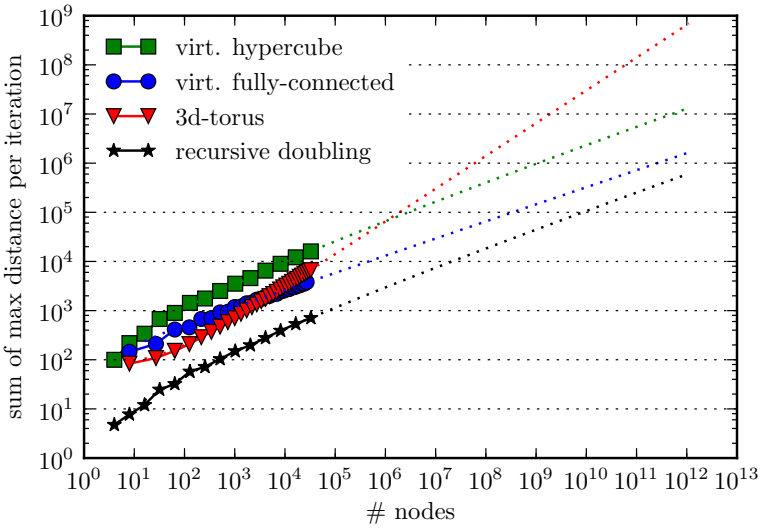


Figure 5.8: Synchronous Push-Sum with different communication schedules on 3d-torus cost topology (C-based Simulator (Appendix A.1), $\varepsilon = 10^{-14}$)

In Figure 5.8 the results of the experiment are shown. For network sizes up to 5000, respectively 1 million nodes, Push-Sum with neighborhood only communication on a 3d-torus had fewer costs than for fully-connected and hypercube communication schedules, but for very large network sizes and for recursive doubling at any range it had always higher costs.

Even though the assumed cost model is very simple and only gives a slight idea on the scalability of Push-Sum with distance dependent communication cost, it shows that the communication costs for all algorithms except Push-Sum with neighborhood only communication scale with the diameter of the network topology. In each iteration it is very likely that for Push-Sum and recursive doubling a node communicates over the maximum distance of the network. The dotted lines in Figure 5.8 show the fitted model functions for synchronous Push-Sum times the diameter of the 3d-torus graph.

Thus, for predicting the costs for Push-Sum on a certain network topology we need to know the second largest eigenvalue of the communication graph and the maximum cost for sending a message in combination with a few small scale results from one of our simulators. Next we show how this information can be used to predict the costs of Push-Sum on the Vienna Scientific Computer 2.

5.5 Models for Vienna Scientific Cluster 2

So far we only compared iteration counts of Push-Sum with recursive doubling, but as we stated in the beginning of this chapter, our actual interest lies in whether the runtime of Push-Sum on HPC systems can compete with deterministic all-reduce algorithms or not. In order to answer this question we developed MPI implementations of all our gossip-based aggregation algorithms [27] and ran them on the Vienna Scientific Cluster 2.

5.5.1 Vienna Scientific Cluster 2

The Vienna Scientific Cluster 2 (http://vsc.ac.at/systems/vsc-2/) is a 1314 nodes machine with a peak performance of 185 TFlops. Each node consists of two AMD Opteron Magny Cours 6132HE (8 Cores, 2.2 GHz) CPUs and 32GiB Memory. The nodes are connected via Infiniband QDR in a fat-tree topology. The fat-tree is divided via 8 root switches into two sub networks, each with 12 spine switches connecting the computer racks. In Figure 5.14 a detailed overview over the network structure is given.

5.5.2 Results from Vienna Scientific Cluster 2

Due to reasons described in detail in appendix A.4 results presented for our MPI Push-Sum implementation show only the number of sent messages and runtime until the first process converged to the target accuracy $\varepsilon = 10^{-14}$. All results from the

Vienna Scientific Cluster 2 show the 10%-quantile over 100 runs each data point, as results were heavily disturbed by system noise for larger process counts.

To make the results comparable, we ran our Python Discrete Event Simulator for Push-Sum with the same termination criteria as our MPI implementations. In Figure 5.9 the results from our Python Discrete Event Simulator for asynchronous Push-Sum where only one node is converged and the according model fits are depicted. Compared to Figure 5.5 where all nodes are converged, the curves differ in a constant offset on the y-axis (Table 5.3 vs. Table 5.2).

a	0.267
b	-16.00
c	109.24

(a) fully-connected

a	0.96
b	-15.94
c	157.64

(b) hypercube

a	1.33
b	-26.99
c	220.39

(c) 3d-torus

Table 5.3: Coefficients for asynchronous Push-Sum where only one node converged cost model

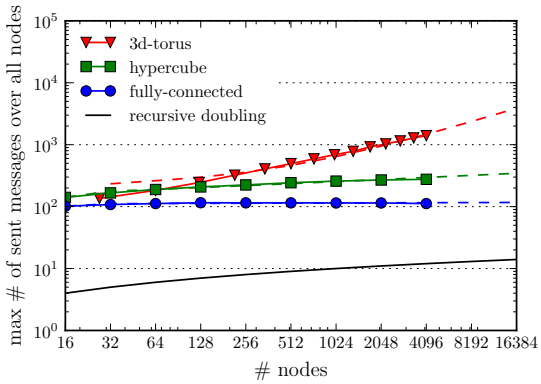


Figure 5.9: Asynchronous Push-Sum where only one node converged and according fitted model functions (Python Discrete Event Simulator (Appendix A.3), $\varepsilon = 10^{-14}$)

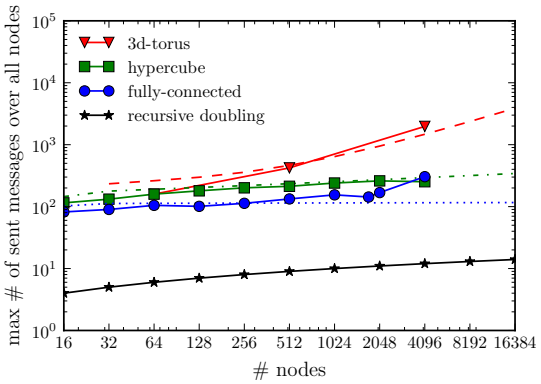


Figure 5.10: Number of messages over all nodes Push-Sum needed until convergence for different communication topologies on Vienna Scientific Cluster 2, dashed curves represent fitted model functions for asynchronous Push-Sum (MPI Push-Sum (Appendix A.4), $\varepsilon = 10^{-14}$)

Figure 5.10 shows the maximum number of messages until convergence of one process for our MPI Push-Sum implementation on the Vienna Scientific Cluster 2. The dashed curves represent the fitted models on max messages for our asynchronous Push-Sum simulation (Figure 5.9). For 3d-torus and hypercube communication schedules the models fit the message counts from Push-Sum on the Vienna Scientific Cluster 2 quite well. But for fully-connected communication schedules Push-Sum on the cluster needs distinctly more messages as expected. Overall and due

reasons described later it shows that the results from our Python Discrete Event Simulator for Push-Sum shows nearly similar results for message counts as our MPI implementations.

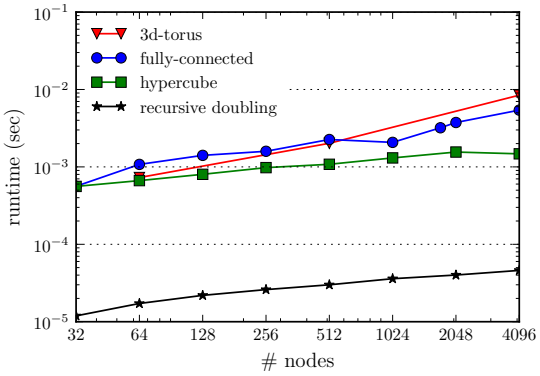


Figure 5.11: Runtime of Push-Sum on Vienna Scientific Cluster 2 for different communication topologies and for recursive doubling. (MPI Push-Sum (Appendix A.4), $\varepsilon = 10^{-14}$)

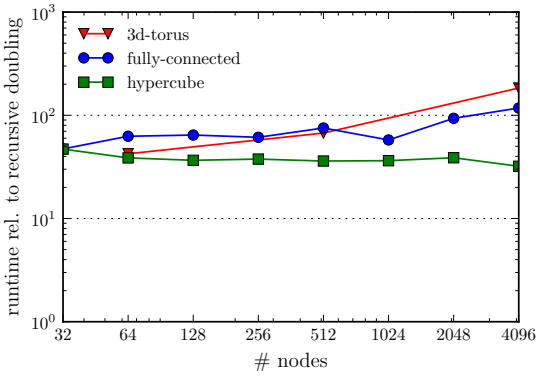


Figure 5.12: Runtime of Push-Sum on Vienna Scientific Cluster 2 for different communication topologies relative to recursive doubling (MPI Push-Sum (Appendix A.4), $\varepsilon = 10^{-14}$)

The runtime results from the Vienna Scientific Cluster 2 for our MPI Push-Sum implementations (Figure 5.11) are on the first glimpse a bit surprising. It shows that other than expected Push-Sum with hypercube schedule is faster than Push-Sum with fully-connected schedule. After some experiments we came to the conclusion that this has to come from overloading the network by the irregular communication patterns of fully-connected Push-Sum. The curves for hypercube and 3d-torus scale as expected, both reflect the differences in message counts seen in the results of our simulations (Figure 5.9).

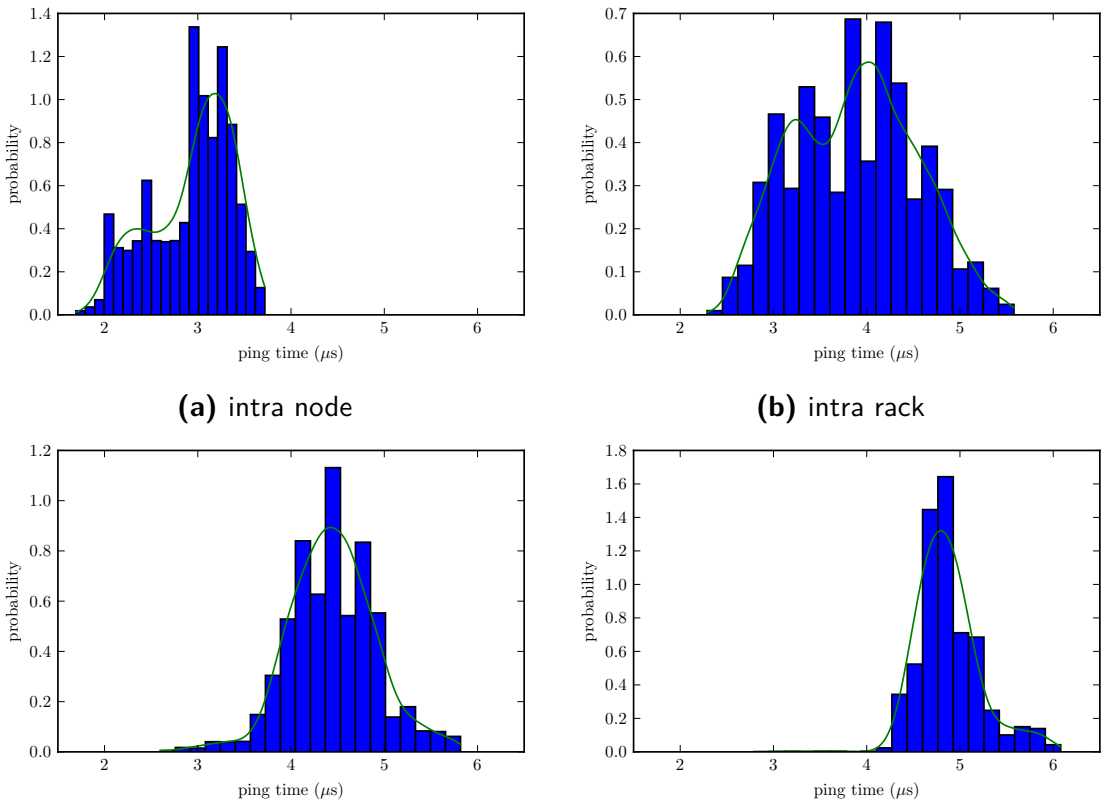
In general, for all communication topologies it shows that the runtime results for Push-Sum on the Vienna Scientific Cluster 2 are far away from being in the range of competitiveness to parallel all-reduce algorithms as recursive doubling. Push-Sum with hypercube communication topology needs roughly 50 times more time than recursive doubling and Push-Sum with fully-connected or 3d-torus schedules is even worse in runtime (Figure 5.12).

Even though the results are not what we hoped for, as the message counts are close to our simulations results, they runtime data allows us to verify our predictive runtime model as sketched in section 5.4. Therefore, the only thing left for predicting the runtime of Push-Sum on the Vienna Scientific Cluster 2 is the maximum time a message needs for a certain process count.

5.5.3 Communication costs of Vienna Scientific Cluster 2

By analyzing the network topology of the Vienna Scientific Cluster 2 (Figure 5.14), the distances between nodes can be divided into 4 classes. The cluster is divided into two parts by root switches, on each side spine switches connect the computer racks. The single nodes in a computer rack are connected via leaf switches. Our first distance class is *intra node communication*, where a message is sent from one process on a node to an other process on the same node. The second class is *intra rack communication*, where a process on one node sends a message to a process on another node on the same rack. Messages from one rack to another on the same side, we call *intra side communication*, whereas messages from one side to the other we gave the name *inter side communication*.

Gathering timing data



(a) intra node **(b)** intra rack

(c) intra side **(d)** inter side

Figure 5.13: Message timings of MPI all-to-all ping on Vienna Scientific Cluster 2

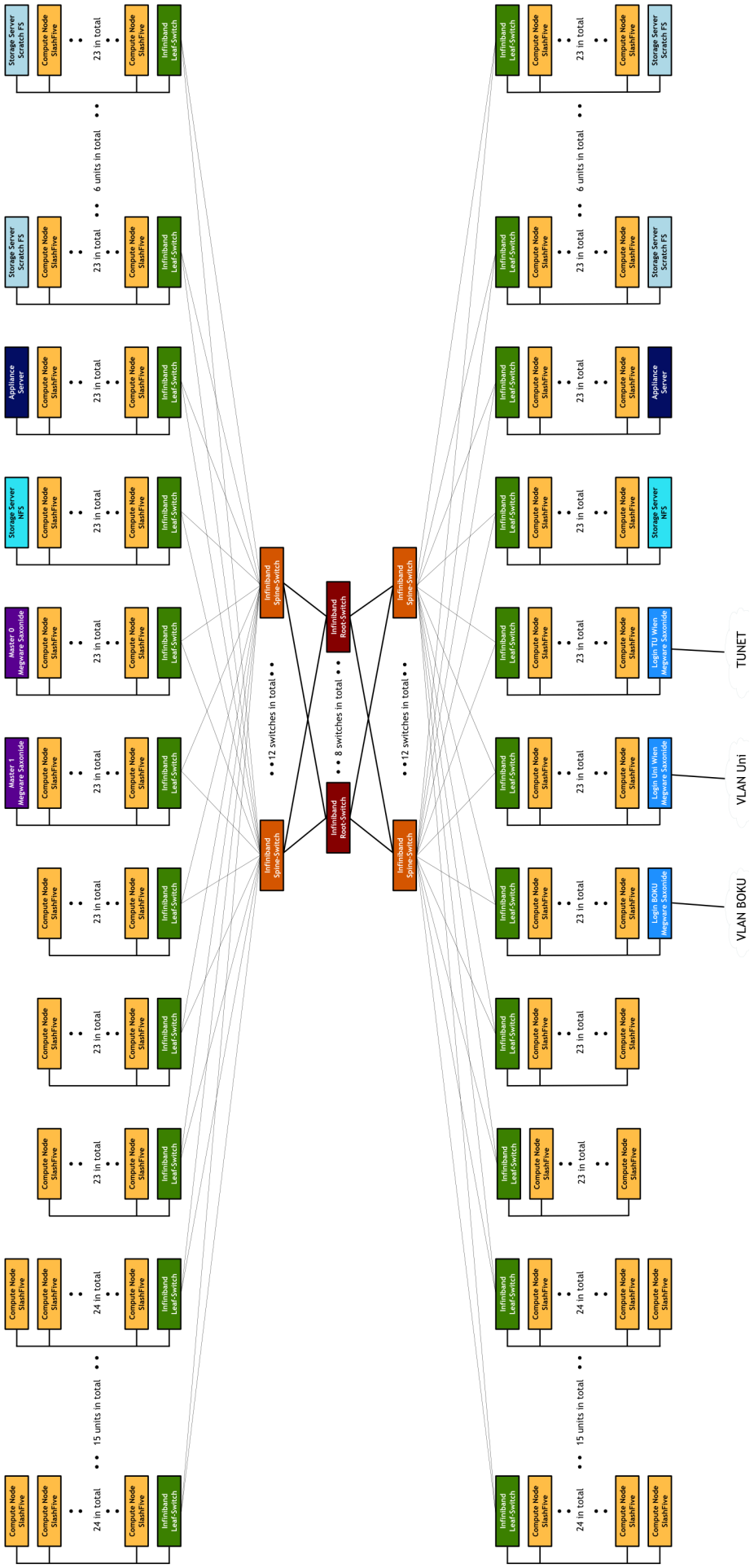


Figure 5.14: Architecture of the Vienna Scientific Cluster 2
(Source: http://vsc.ac.at/systems/vsc-2/)

Having distinguished the 4 different possible distances in the Vienna Scientific Cluster 2 network we wrote a MPI ping program that measures all message round trip timings for all possible pairs of processes. For each process pair the average time for a message sent 5 times between the processes is computed, this is necessary as the time for a single round trip is close to the resolution of the MPI timer function. Each measurement is repeated 50 times.

By the hostnames of the nodes we are capable to distinguish different nodes and racks. It showed that nodes with even Ids are on one side of the network and hosts with odd node Ids are on the other side. Figure 5.13 shows histograms for the measurements separated into the 4 categories. The timing distributions are overlapping but the mean values (Table 5.4) for each distance class are, as a Kruskal-Wallis H-test, shows significantly different. As assumed, intra node communication is the fastest, whereas inter side communication is slowest. The timings for intra rack and intra side communication lie in the middle.

	average time
intra node	$2.9\mu s$
intra rack	$3.9\mu s$
intra side	$4.5\mu s$
inter side	$5.0\mu s$

Table 5.4: Average time for sending a message on Vienna Scientific Cluster 2

5.5.4 Predicting runtimes on Vienna Scientific Cluster 2

With all the information at hand it is possible to predict the runtime of our MPI Push-Sum implementations on the Vienna Scientific Cluster 2. Process counts on the Vienna Scientific Cluster 2 must be multiples of 16. As jobs get only assigned to hole nodes, which have 16 cores each on the Vienna Scientific Cluster 2. Thus, for jobs up to 16 cores the mean time for an iteration is $2.9\mu s$. For larger jobs the maximum communication cost are not as obvious as for 16 processes. Usually the system is highly used and it is not likely that the nodes are assigned communication optimal over the system, this means not all nodes reside in the same rack even though they would fit in. The same holds for the distribution over the two sides of the Vienna Scientific Cluster 2. Therefore, we had to experiment with different threshold values to find a reasonable cost function which represents the cost of the Vienna Scientific Cluster 2 best. The cost function which worked best for us is shown in table 5.5.

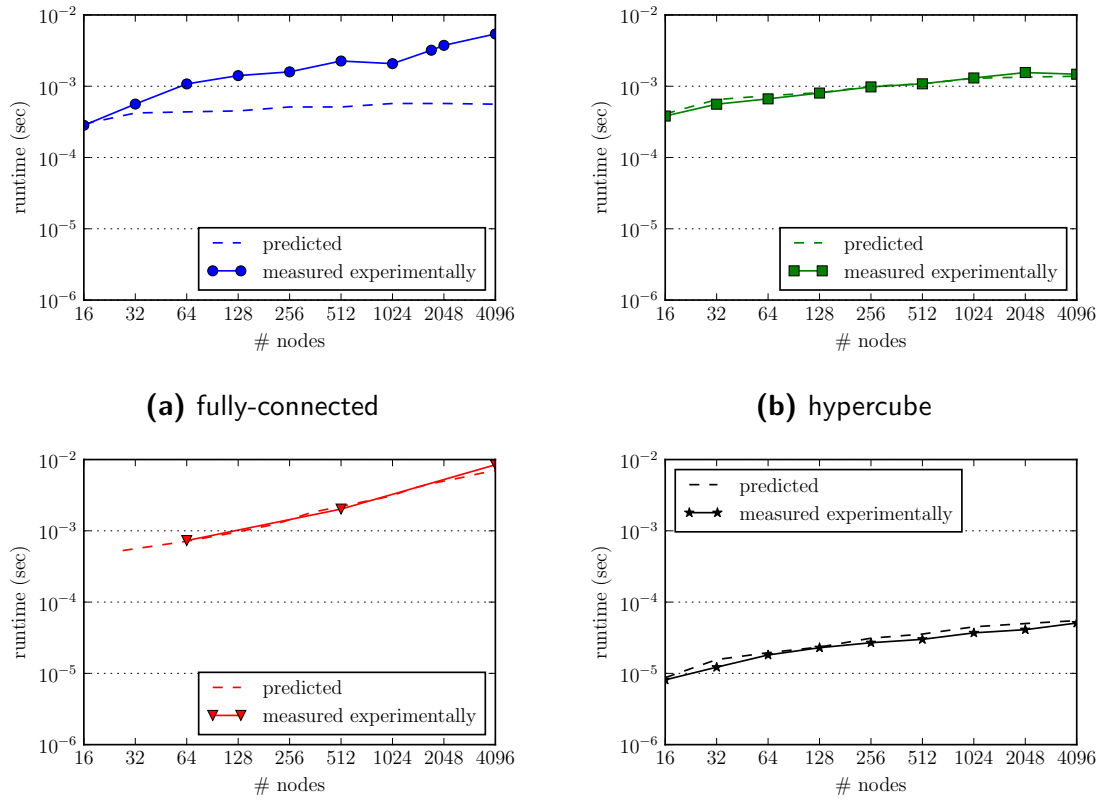
By multiplying the model functions of our asynchronous Push-Sum simulation shown in figure 5.9 with the cost from table 5.5 we can predict the cost of Push-Sum on the Vienna Scientific Cluster 2. Figure 5.15 depicts the resulting cost functions with the

node size	$N \leq 16$	$N < 256$	$N < 1024$	$N \geq 1024$
cost per iteration	$2.9\mu s$	$3.9\mu s$	$4.5\mu s$	$5.0\mu s$

Table 5.5: Estimated cost per iteration for Vienna Scientific Cluster 2 with different process counts N

runtime results from Vienna Scientific Cluster 2. It shows that for Push-Sum with fully-connected communication topology the runtime results are not as expected, as our model does not take congestion into account. For hypercube and 3d-torus communication schedules and recursive doubling the runtime can be predicted very accurately.

Even though our simple model is limited to iteration based algorithms like Push-Sum and recursive doubling and cannot model queuing effects, it was sufficient for our usage. We are now capable to predict the approximate runtime of our algorithm for many HPC systems by only knowing the system specifications.



(a) fully-connected

(b) hypercube

(c) 3d-torus

(d) recursive doubling

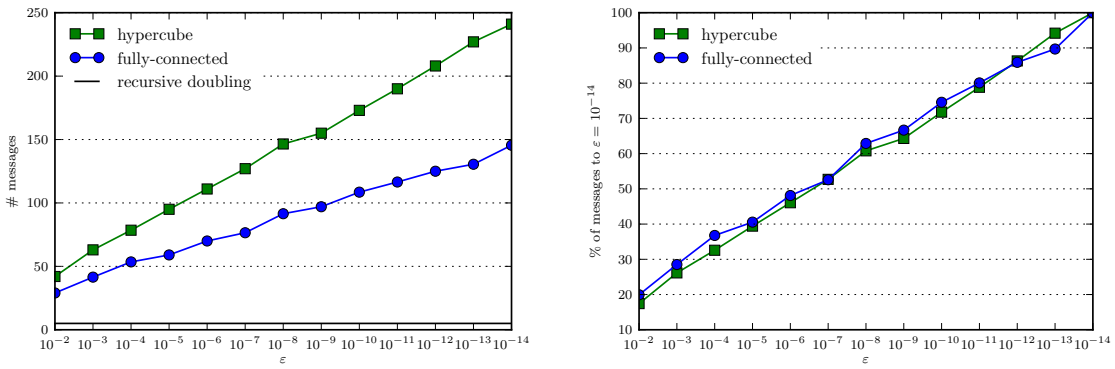
Figure 5.15: Predicted runtimes for Push-Sum on Vienna Scientific Cluster 2

5.6 Can performance be improved?

Push-Sum with a target accuracy close to machine epsilon has a huge overhead on messages compared to parallel all-reduce algorithms as recursive doubling or Rabenseifner's algorithm. One factor on performance we did not investigate so far is the target accuracy ε.

5.6.1 Reduced accuracy

From the asymptotic runtime complexity of Push-Sum (equation 5.1) we know that the cost for reaching a target accuracy ε, are logarithmic to the reciprocal of ε. This means the additional iterations to reach ε, are linear to the magnitude of ε. Thus, Push-Sum with a target accuracy ε_2 of half the magnitude of ε_1, should roughly take half the iterations as for ε_1.



(a) absolute # of messages **(b)** % of messages rel. to $\varepsilon = 10^{-14}$

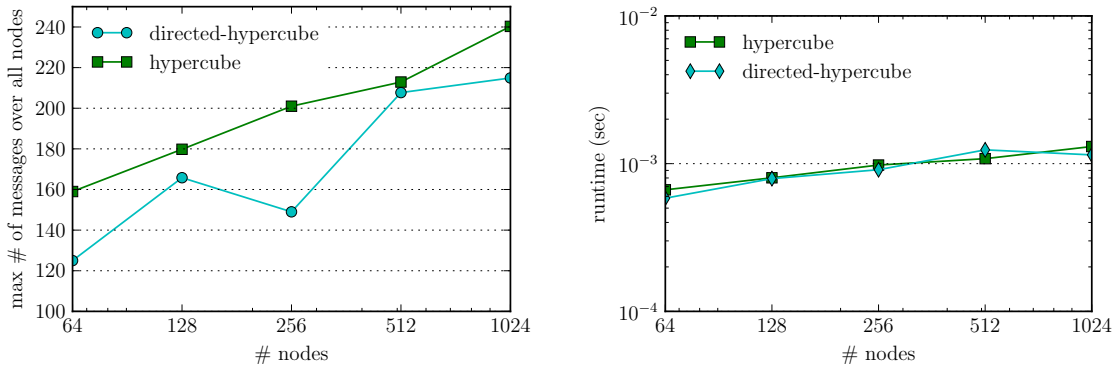
Figure 5.16: Influence of ε on the # of messages until convergence of Push-Sum (Python Discrete Event Simulator (Appendix A.3), 64 nodes, hypercube communication topology, $\varepsilon = 10^{-14}$)

In Figure 5.16a the number of messages until convergence of Push-Sum for different target precision ε is shown. From Figure 5.16b, where the percentage of messages Push-Sum needs for a reduced accuracy relative to a target accuracy of $\varepsilon = 10^{-14}$ is depicted, it can be seen that the cost for reaching a certain magnitude of accuracy are linear.

The dependency of the to achieve accuracy of gossip-based aggregation on the number of sent messages and therefore the reduced cost for less accurate results open the possibility to tune our algorithms exactly to the needed accuracy of certain applications. For less accuracy demanding problems it might even be, that our algorithms can beat current non-fault-tolerant reduction algorithms.

5.6.2 Directed communication graphs

When searching for a solution for the problem of mutual messages in Push-Flow, described in detail in section 2.2.2, we figured out that Push-Sum on some directed communication graphs needs fewer iterations as on the undirected underlying graph (Figure 2.4). As it additionally showed that the directed graphs had smaller second largest eigenvalues as the undirected underlying graph, the results are well supported by the theoretical runtime complexity of Push-Sum described in section 5.2.



(a) number of messages **(b)** runtime

Figure 5.17: Push-Sum with directed hypercube and undirected hypercube communication topology on Vienna Scientific Cluster 2 (MPI Push-Sum (Appendix A.4), $\varepsilon = 10^{-14}$)

In practice, as results for Push-Sum with a directed hypercube [7] communication topology show on the Vienna Scientific Cluster 2, the reduced number of messages (Figure 5.17a) has no significant influence on the runtime (Figure 5.17b).

5.7 Open questions

- How is the influence of the input data on the number of iteration until convergence of our gossip-based all-to-all aggregation algorithms?
- Can the performance be further improved?
- Can permuted communication schedules be used for PFCC?
- Does there exist a combination of network topology and associated communication cost where gossip-based all-to-all aggregation can compete with deterministic all-reduce algorithms?

6. Conclusion

We showed that Push-Sum, even the opposite is often assumed, has the same resilience against message loss and data corruption as non-fault-tolerant deterministic all-reduce algorithms like recursive doubling. The concept of flows allows Push-Flow to recover from message loss and most bit-flips in messages and its local flow variables. But it shows that due to the tremendous growth of the flow variables, when exponent bits get flipped, it can happen that Push-Flow converges to a wrong result. We proved that Push-Cancel-Flow which uses a sophisticated mechanism to hinder the flow variables from growing as incorrect as it has a problem with consecutive messages. Our novel algorithms PFLC and PFCC introduce check-sums into the flow variables and messages of Push-Flow to detect bit-flips. They are capable of handling bit-flips at any position in the flow variables. Through intelligent leverage of the redundancy in the neighbors nodes flows, PFCC recovers almost immediately from data corruption and shows superior performance in case of failures over PFLC.

So far no studies on the performance of gossip-based all-to-all reduction algorithms on high performance computing systems were made. We developed several tools to simulate Push-Sum on a large scale. Additionally, we implemented all our algorithms in MPI and ran them on the Vienna Scientific Cluster 2. Based on the results of our simulators we developed cost models which allow us to predict message counts and runtimes of Push-Sum on fictitious computing systems. We showed the validity of our models by predicting the runtime of Push-Sum for the Vienna Scientific Cluster 2 and comparing it with results from actual runs of our MPI codes.

A question we asked was, whether fault-tolerant gossip-based algorithms can compete with deterministic parallel all-reduce algorithms in terms of runtime. We stated that our gossip-based algorithms runtime has to be in the range of three times the runtime of the recursive doubling algorithm to consider gossiping an alternative to classical fault-tolerance approaches like triple temporal redundancy.

First runtime results from the Vienna Scientific Cluster 2 for Push-Sum showed that the runtime of Push-Sum is approximately 50 times the runtime of recursive doubling and thus far beyond from what would be acceptable. But we also highlighted the potential to improve performance of gossip-based reduction algorithms by reducing the target accuracy and by directed communication schedules. Whether there can

be a combination of network topology and associated communication costs, where gossip-based all-to-all aggregation can compete with deterministic all-reduce algorithms is still an open research question and further effort to finally answer this question is needed.

Concluding the thesis, we would like to point out the great resilience of our novel algorithms PFLC and PFCC against several types of faults, but also that it is very unlikely that PFLC and PFCC can beat the performance of deterministic fault-tolerant all-reduce algorithms in a HPC context.

6.1 Future Work

The next logical step is to finally answer whether gossip-based reduction algorithms can compete with deterministic fault-tolerant algorithms or not. Even if the answer is negative, the following topics are worth further effort as gossip-based algorithms are often the only available algorithms for environments where less on the topology is known and high resilience against several types of failures is necessary.

A pressing problem for gossip-based aggregation is the lack of a practical method to test for global convergence. It is essential that this algorithm has to be fault-tolerant itself, as otherwise all resilience efforts of our algorithms are thwarted. Furthermore, it is important that the methods overhead is minimal and that it scales equally with our algorithms.

The result in figure 5.4 indicate that it might be possible that for extremely large node sizes Push-Sum may need fewer iteration than recursive doubling which is optimal in number of iterations. As then the aggregate cannot include data from all nodes, the effect must be of statistical nature, dependent on the distribution of the input data. A statistical approach to quantify the needed iterations (sample size) for a certain input distribution and problem size may shed light on this phenomenon. Maybe this leads to a hole new type of algorithms for huge problems in HPC.

Less fundamental but still of interest are the questions that arose when developing our novel algorithms PFLC and PFCC. How to find an optimal value for τ, the threshold for resetting flows in PFLC and PFCC and Apart from directed communication topologies, is there a solution to the mutual communication problem in Push-Flow-based algorithms which leads for some communication schedules to a huge message overhead?

A. Implementations

Despite gossip-based algorithms are inherently *asynchronous* communicating algorithms, where each node sends and receives messages independently of all other nodes, they can also be implemented in a *synchronous* way, where iterations are synchronized over all nodes.

The advantage of synchronized iteration based implementations is that they can be easily realized for sequential environments and thus a perfect for simulations on small machines. It also showed that their performance is much higher than for sequential asynchronous implementations of gossip-based algorithms.

Our synchronous C-based Push-Sum simulator allows us to run Push-Sum up to several hundred thousands nodes. Whereas with our python-based discrete event simulator for asynchronous gossip-aggregation it is possible to simulate our sophisticated gossip algorithms PFLC and PFCC which depend on asynchronicity. Additionally, we implemented all algorithms with MPI, the industry standard for communication in HPC.

A.1 C-based Push-Sum simulators

Our C-based Push-Sum simulators are straight forward implementations of Push-Sum as formulated in algorithm 2.

In each iteration consecutively every node i chooses some neighbor node j and adds half of it local value weight pair to the neighbors pair (Algorithm 7 lines 5-7). This is repeated until all nodes are converged to an ε-accurate aggregate or a certain maximum iterations count is reached (Algorithm 7 line 3). Therefore, the true aggregate is calculated in advance (Algorithm 7 line 2) and after each iteration the relative error is calculated (Algorithm 7 line 9).

The source code for our C-based Push-Sum simulators is available under `https://gitlab.cs.univie.ac.at/taa/c-gossip`.

Algorithm 7 C-based Push-Sum simulator

Input: $x, w,\ \varepsilon$, max-iterations
Output: $\forall i : y_i \approx \sum_k x_k \ / \sum_k w_k$

```
1:  iter ← 0
2:  true-result ← Σ_k x_k / Σ_k w_k
3:  while residual > ε and iter < max-iterations do
4:      for i ← 0 : N − 1 do
5:          j ← choose a neighbor ∈ N_i uniformly at random
6:          (x_i, w_i) ← (x_i, w_i)/2
7:          (x_j, w_j) ← (x_j, w_j) + (x_i, w_i)
8:          y_i ← x_i/w_i
9:          iter ← iter + 1
10:     residual ← ||y−true−result||_∞ / ||true−result||_∞
```

A.2 Push-Sum for asynchronous communication

For asynchronous computing models there exist two different types of communication. Synchronous communication send routines block until a message gets received, whereas for asynchronous communication send routines return immediately.

When looking at Algorithm 2 one would assume that after the send routine on process i returns, in average one message arrived, as in expectation each process receives one message per iteration and transmitting a message should take roughly the same time for all processes. This is true for synchronous communication, but with asynchronous communication where the send routine usually returns before the message gets received, Push-Sum as in Algorithm 2 will send a vast amount of *identical* messages before receiving any message from a neighbor (Figure A.1), as then no synchronization trough communication can be achieved. This leads to tremendous amounts of sent messages per nodes without any progress in convergence and thus has to be avoided.

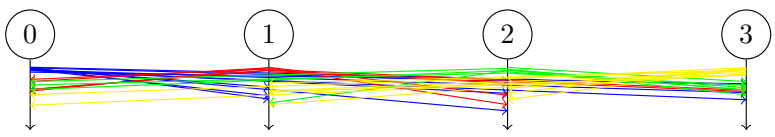


Figure A.1: Standard Push-Sum with asynchronous communication sends lots of messages before receiving messages from neighbors, as the send routine returns immediately

Therefore when using an asynchronous send routine, measures must be taken to avoid sending to many messages without receiving any. One solution can be to mimic the time needed by a synchronous send routine, by introducing an artificial

waiting time. Obviously this parameter has to be set differently for each computing system and thus lacks the generality of the following approach.

By slightly reformulating Algorithm 2, Push-Sum gets well suited for asynchronous communication. The difference is to send one message for each incoming message

Algorithm 8 Push-Sum for asynchronous communication model

Input: $x_i, w_i \in \mathbb{R}$

Output: $y_i \approx \sum_{x_k \in \mathcal{X}} x_k \,/\, \sum_{w_k \in \mathcal{W}} w_k \in \mathbb{R}$

1: **while** not ε-accurate **do**
2: $\quad p_j \leftarrow$ choose a neighbor from $\mathcal{N}_i$ uniformly at random
3: $\quad$ **send** $(x_i, w_i)/2$ to p_j
4: $\quad (x_i, w_i) \leftarrow (x_i, w_i)/2$
5: $\quad$ **wait** until pair (x_j, w_j) received from a neighboring node
6: $\quad (x_i, w_i) \leftarrow (x_i, w_i) + (x_j, w_j)$
7: $y_i \leftarrow \frac{x_i}{w_i}$

(Algorithm 8 line 5). Beside enabling asynchronous communication (Figure A.2), this approach has also the advantage of only sending *new* information, which usually results in sending fewer messages at all until converged.

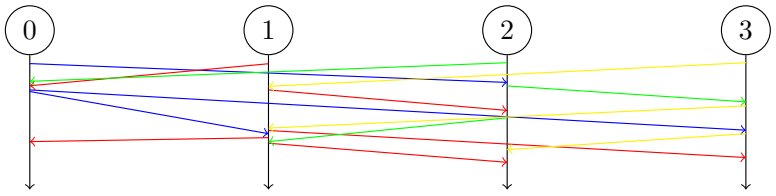


Figure A.2: Push-Sum for asynchronous communication sends for every received message one message to a neighbor

A.3 Python-based discrete event simulator

Our Python-based discrete event simulator (Algorithm 9) uses a priority queue (`heapq`) as container for the messages. In the queue the messages are ordered by receive time and in each time step the first message gets processed (Algorithm 9 lines 3-6).

For each new message the current nodes time plus a transmit from some cost function is added and used as the receive time for the message (Algorithm 9 lines 7-9). After each message got processed the overall accuracy is calculated and the simulator stops when a certain accuracy is achieved or a max message count is exceeded (Algorithm 9 line 10).

Algorithm 9 Python-based discrete event simulator

Input: $x, w,\ \varepsilon$, max-iterations
Output: $\forall i : y_i \approx \sum_k x_k \,/\, \sum_k w_k$

1: true-result $\leftarrow \sum_k x_k \,/\, \sum_k w_k$
2: **while** residual $> \varepsilon$ **do**
3: **get** $(t, i, (x_j, w_j))$ from queue
4: $(x_i, w_i) \leftarrow (x_i, w_i) + (x_j, w_j)$
5: $(x_i, w_i) \leftarrow (x_i, w_i)/2$
6: $y_i \leftarrow x_i/w_i$
7: $j \leftarrow$ choose a neighbor $\in \mathcal{N}_i$ uniformly at random
8: $tt \leftarrow costs(i, j)$
9: **put** $(t + tt, j, (x_i, w_i))$ to queue
10: residual $\leftarrow \frac{||y - true-result||_\infty}{||true-result||_\infty}$

The advantage of our Python-based discrete event simulator over our MPI implementations is that we can stop the algorithm when all nodes are converged without measuring any additional overhead induced by termination algorithms as in MPI.

The source code is available under
https://gitlab.cs.univie.ac.at/taa/python-gossip.

A.4 HPC implementations with MPI

To compare gossip-based aggregation with state-of-the-art parallel reduction algorithms we developed several MPI implementations of Push-Sum [27].

We implemented Push-Sum using MPI synchronous (`ps_ptp_sym_lt`) and asynchronous (`ps_ptp_asy_lt`) point-to-point communication as well with MPI one-sided communication (`ps_win_sym_lt`).

Even tough MPI one-sided communication is meant for asynchronous applications, it showed that Push-Sum with one-sided communication performed worse than with point-to-point communication. Overall the performance of `ps_ptp_asy_lt` using asynchronous MPI point-to-point communication was the best (Figure A.3).

Thus for our fault-tolerant extensions Push-Flow (`pf_ptp_asy_lt`), Push-Cancel-Flow (`pfc_ptp_asy_lt`), PFLC (`pflc_ptp_asy_lt`) and PFCC (`pfcc_ptp_asy_lt`) we solely used asynchronous point-to-point communication in MPI.

So far there exists no satisfying convergence criteria for gossip-based aggregation algorithms, so that every node has knowledge of the convergence of every other node. Thus, similar to our sequential simulators we first had to compute the result in a deterministic way to calculate a nodes accuracy. Still, this gave us only knowledge

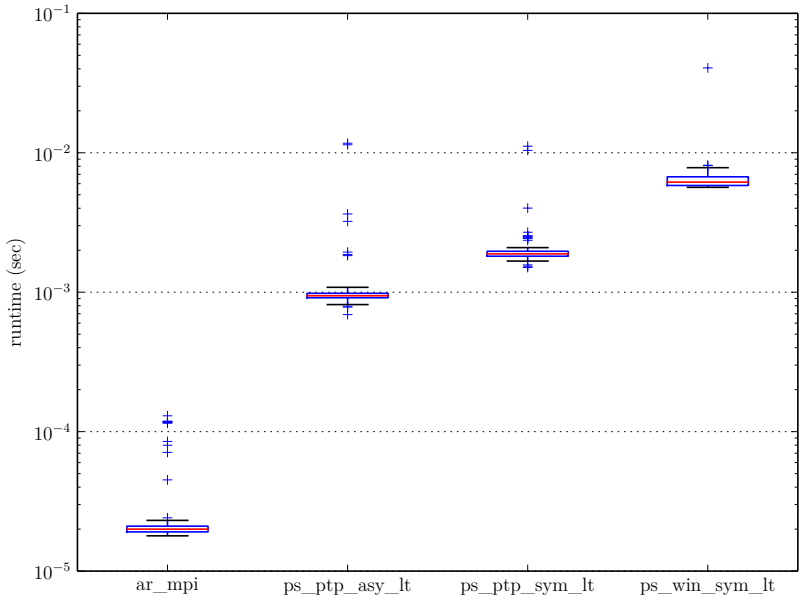


Figure A.3: Runtime comparison of Push-Sum MPI implementations (Vienna Scientific Cluster 2, 128 processes, $\varepsilon = 10^{-14}$)

about the local convergence of each node. Thus, for informing all other nodes about local convergence additional communication was necessary. As this additional communication led to a tremendous overhead in messages and runtime, the results presented show only the runtime and number of message until the first process converged.

Further details of the implementation and termination of our MPI implementations are described in detail in [27].

The source code is available under
https://gitlab.cs.univie.ac.at/taa/mpi-gossip.

A.5 Open questions

- How does a node know when it is ε-accurate.
- How does a node know that all nodes are converged.

Bibliography

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

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

[3] D. Buntinas, C. Coti, T. Hérault, P. Lemarinier, L. Pilard, A. Rezmerita, E. Rodriguez, and F. Cappello. Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI protocols. *Futur. Gener. Comp. Syst.*, 24:73–84, 2008.

[4] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir. Toward exascale resilience: 2014 update. *Supercomput. Frontiers and Innovations*, 1:1–28, 2014.

[5] E. Chan, M. Heimlich, A. Purkayastha, and R. van de Geijn. Collective communication: Theory, practice, and experience: Research articles. *Concurr. Comput. : Pract. Exper.*, 19(13):1749–1783, Sept. 2007.

[6] Z. Chen and J. J. Dongarra. Algorithm-Based Fault Tolerance for Fail-Stop Failures. *IEEE Trans. Parallel Distrib. Syst.*, 19(12):1628–1641, 2008.

[7] C. Domshlak. On recursively directed hypercubes. *Electr. J. Comb.*, 9(1), 2002.

[8] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.-C. Andre, D. Barkai, J.-Y. Berthou, T. Boku, B. Braunschweig, F. Cappello, B. Chapman, X. Chi, A. Choudhary, S. Dosanjh, T. Dunning, S. Fiore, A. Geist, B. Gropp, R. Harrison, M. Hereld, M. Heroux, A. Hoisie, K. Hotta, Z. Jin, Y. Ishikawa, F. Johnson, S. Kale, R. Kenway, D. Keyes, B. Kramer, J. Labarta, A. Lichnewsky, T. Lippert, B. Lucas, B. Maccabe, S. Matsuoka, P. Messina, P. Michielse, B. Mohr, M. S. Mueller, W. E. Nagel, H. Nakashima, M. E. Papka, D. Reed, M. Sato, E. Seidel, J. Shalf, D. Skinner, M. Snir, T. Sterling, R. Stevens, F. Streitz, B. Sugar, S. Sumimoto, W. Tang, J. Taylor, R. Thakur, A. Trefethen, M. Valero, A. Van Der Steen, J. Vetter, P. Williams, R. Wis-

niewski, and K. Yelick. The international exascale software project roadmap. *Int. J. High Perform. Comput. Appl.*, 25(1):3–60, Feb. 2011.

[9] J. Elliott, M. Hoemmen, and F. Mueller. Evaluating the impact of SDC on the GMRES iterative solver. In *Proc. IEEE 28th Int. Parall. & Distr. Processing Symp.*, pages 1193–1202. IEEE, 2014.

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

[11] 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 *Proc. Int. Conf. on High Performance Computing, Networking, Storage and Analysis*, pages 78:1–78:12. IEEE, 2012.

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

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

[14] J. Hursey and R. L. Graham. Analyzing fault aware collective performance in a process fault tolerant MPI. *Parallel Comput.*, 38(1-2):15–25, 2012.

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

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

[17] 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*, pages 482–491, 2003.

[18] J. Langou, Z. Chen, G. Bosilca, and J. Dongarra. Recovery patterns for iterative methods in a parallel unstable environment. *SIAM J. Sci. Comput.*, 30(1):102–116, 2008.

[19] Message Passing Interface Forum. MPI: A message-passing interface standard version 3.0. Specification, September 2012.

[20] G. Niederbrucker. *Towards Truly Distributed Computing: Uniting Theory, Algorithms and Practice*. PhD thesis, University of Vienna, 2014.

[21] G. Niederbrucker and W. N. Gansterer. Robust gossip-based aggregation: A practical point of view. In P. Sanders and N. Zeh, editors, *Proc. 15th Meeting on Algorithm Engineering & Experiments*, pages 133–147. SIAM, 2013.

[22] G. Niederbrucker, H. Straková, and W. N. Gansterer. Improving fault tolerance and accuracy of a distributed reduction algorithm. In *Proc. Third Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, pages 643–651, 2012.

[23] K. Qiu and S. K. Das. Interconnection networks and their eigenvalues. In *Parallel Architectures, Algorithms and Networks, 2002. I-SPAN '02. Proceedings. International Symposium on*, pages 163–168, 2002.

[24] N. S. V. Rao. On undecidability aspects of resilient computations and implications to exascale. In L. Lopes et al., editor, *Euro-Par 2014: Parallel Processing Workshops*, volume 8805 of *Lecture Notes in Computer Science*, pages 511–522. Springer, 2014.

[25] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, A. A. Chien, P. Coteus, N. A. Debardeleben, P. C. Diniz, C. Engelmann, M. Erez, S. Fazzari, A. Geist, R. Gupta, F. Johnson, S. Krishnamoorthy, S. Leyffer, D. Liberty, S. Mitra, T. Munson, R. Schreiber, J. Stearley, and E. V. Hensbergen. Addressing failures in exascale computing. *Int. J. High Perform. Comput. Appl.*, 28(2):129–173, May 2014.

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

[27] E. Wimmer. Implementing push-sum with mpi. Bachelor Thesis, 2015.

Curriculum Vitae

Elias Wimmer, BSc

EMAIL: elias.wimmer@gmail.com

Education

2015 - 2016	Masterstudium *Scientific Computing*, University of Vienna
2008 - 2015	Bachelorstudium *Informatik mit Schwerpunkt Scientific Computing*, University of Vienna Bachelor thesis: „Implementing Push-Sum with MPI” Adviser: Wilfried Gansterer

Work Experience

2012 - 2015	Wissenschaftlicher Projektmitarbeiter, University of Vienna