Scalable and Fault Tolerant Orthogonalization Based on Randomized Distributed Data Aggregation

Wilfried N. Gansterer, Gerhard Niederbrucker, Hana Straková, Stefan Schulze Grotthoff

University of Vienna, Research Group Theory and Applications of Algorithms
Lenaugasse 2/8, A-1080 Vienna, Austria

Abstract

The construction of distributed algorithms for matrix computations built on top of distributed data aggregation algorithms with randomized communication schedules is investigated. For this purpose, a new aggregation algorithm for summing or averaging distributed values, the push-flow algorithm, is developed, which achieves superior resilience properties with respect to failures compared to existing aggregation methods. It is illustrated that on a hypercube topology it asymptotically requires the same number of iterations as the optimal all-to-all reduction operation and that it scales well with the number of nodes. Orthogonalization is studied as a prototypical matrix computation task. A new fault tolerant distributed orthogonalization method rdmGS, which can produce accurate results even in the presence of node failures, is built on top of distributed data aggregation algorithms.

Keywords: distributed reduction operation, push-flow algorithm, distributed orthogonalization, distributed matrix computations, fault tolerant matrix computations

1. Introduction

Algorithms for future large-scale computer systems have to be designed to provide resilience to various types of failures and to require less synchronization between nodes than state-of-the-art parallel algorithms. The basic idea underlying this paper is to investigate the construction of suitable distributed algorithms for matrix computations built on top of *distributed data aggregation algorithms* (*DDAAs*). DDAAs can be seen as distributed versions of all-to-all reduction operations, in particular for summing or for averaging the elements of a long vector distributed over many nodes. Consequently, distributed versions of basically all types of BLAS operations can potentially be constructed based on DDAAs.

We first evaluate the strengths and weaknesses of existing distributed data aggregation algorithms. Then, we present the *push-flow algorithm*, a new DDAA which overcomes some drawbacks of existing methods in terms of resilience. Finally, we show how DDAAs can be used as building blocks for developing a scalable and fault tolerant distributed orthogonalization/QR factorization method as a prototypical matrix computation task.

1.1. Problem Setting

We consider a large-scale computer system with N nodes arranged in a fixed (but otherwise arbitrary) topology as target system for the computations. Every node knows which nodes are its neighbors, but does not need to have any global information about the network. Our focus is on *distributed* algorithms for such large-scale systems. We clearly distinguish distributed from *parallel* algorithms. The latter are usually designed for small to medium-sized static and reliable systems with regular and globally known topology, where synchronized computation across nodes in the network can be guaranteed. However, parallel algorithms have major drawbacks in possibly decentralized large-scale systems with arbitrary topologies and potentially unreliable components (e. g., failing nodes or communication links), where synchronization of the nodes may be difficult to achieve. Distributed algorithms provide much greater flexibility with respect to the hardware infrastructure than classical parallel algorithms. They neither rely on a fixed communication schedule nor on full synchronization across the nodes. Moreover, they have the potential for producing meaningful results even in the presence of link or node failures. More generally speaking, distributed algorithms are attractive in all computations over large-scale computing systems where (*i*) the nodes do not have complete *global* information about the system, but predominantly only local information about their neighborhood, and/or (*ii*) the system may change dynamically (e. g., due to hardware failures).

The algorithms we investigate are based on *gossiping protocols*. Such algorithms are attractive in such situations, because due to their randomized information exchange they do not require static or reliable hardware infrastructure. If communication is restricted to the local neighborhood of each node, the number of iterations required tends to scale logarithmically with the number of nodes in the system, which is asymptotically the same as in optimized all-to-all reduction operations. Moreover, due to their iterative structure they can deliver results at reduced accuracy levels for a communication cost which is proportional to the target accuracy.

1.2. Related Work

Approaches for achieving fault tolerance in parallel and distributed systems have been investigated at various levels. At the MPI level, FT-MPI [1] provides interfaces for improving the fault tolerance of applications. It is designed to recover from link or node failures by continuing from consistent points which have to be defined by the application developer.

A standard approach at the parallel application level is *coordinated checkpointing* followed by *rollback recovery* in case of a failure. However, it has been shown that—depending on the checkpointing interval [2]—the synchronization of the periodic coordinated checkpointing limits application scalability, and for large systems a dominating fraction of the runtime tends to be spent on checkpointing and restarting instead of advancing in the application [3]. Moreover, for large-scale systems it can also become expensive to provide sufficient stable storage. Several improvements of coordinated checkpointing have been proposed. Alternatives focusing on storing the checkpoints more efficiently are diskless checkpointing [4] and RAID-inspired checkpointing [5] which stores checkpoints redundantly across the nodes' memory. *Uncoordinated checkpointing* [6] does not require the synchronization of checkpointing procedures across the nodes and in case of failure only the failed process needs to be restarted, not the entire application. However, the restart tends to be much more complex, since the failed processes need to find a common checkpoint. Also, storage requirements tend to be much higher since it is not clear which checkpoint will be required for restart.

Different approaches have been suggested to increase the interval between failures and thus to decrease the number of restarts. The first approach is called *redundant computing*. Each process is replicated across the system, and thus it can handle multiple failures without recovery overhead. An example is the rMPI library [7], where redundancy is used to increase the interval between failures and thus to reduce the overhead caused by storing checkpoints and restarting the system. In [8] the advantages and limitations of double and triple redundancy are discussed. Another concept called *live migration* [9] is a proactive approach, where processes are migrated away from unhealthy nodes to healthy nodes.

On the lower level of distributed data aggregation algorithms the algorithms discussed in this paper do not require any checkpointing or redundant computing. On the higher level of distributed matrix computations (in this paper, distributed QR factorization) our algorithms differ from the concept of checkpointing because they react to failures whenever they occur. They can be considered a combination of a redundant computing and a live migration approach, without assuming common external storage or extra hardware, though. Failed nodes do not have to be repaired or replaced, but the remaining nodes take over their responsibilities.

A purely algorithmic way of achieving fault tolerance for high level matrix operations is the technique of *algorithm-based fault tolerance* (*ABFT*) [10]. The basic idea of ABFT is to extend the input matrices by checksums and to adapt the algorithms such that these checksums get updated correctly and can consequently be used for detecting and recovering from errors. Classically, ABFT was designed for handling a prescribed amount of miscalculations with a high probability [10], whereas more recently also fail-stop failures, where nodes entirely crash, were considered [11]. Besides the extension to fail-stop failures there are also efforts into methods with ABFT which make use of the inherent redundancy of the data distribution accross the nodes to recover from failures, which results in methods where only failure situations lead to an overhead [12].

The available ABFT literature discusses specific numerical linear algebra tasks, such as matrix-matrix multiplications, LU decompositions or iterative linear solvers, but does not consider the elementary building blocks below the Blas-level. In contrast to ABFT, our approach does not modify the linear algebra algorithms themselves, but instead we focus on new distributed data aggregation building blocks, which also improve the resilience of the algorithms based on them. Our methods do not recover by an explicit (deterministic) recovery step as in ABFT, but they rather enter a "healing" phase once the next successful failure-free iteration can be performed (e. g., after a previously failed link has been re-established). At the distributed data aggregation algorithms-level, the overhead for higher resilience in terms of slower convergence and extra data transmission depends on the failure type and is only incurred when a failure actually happens.

DDAAs and other simple distributed algorithms based on randomized communication schedules have been discussed extensively in the literature. Important examples are algorithms based on *gossiping* (or *epidemic*) protocols [13, 14]. In the basic approach each node communicates with randomly chosen neighboring nodes [13]. In other variants the communication partners of a node are chosen from the entire network regardless of the distance [15] or its local value is broadcasted to all neighbors [16]. Most of the existing work focuses on distributed algorithms for simple network operations, such as information dissemination (rumor mongering) [17], aggregation [18], network organization (routing, load balancing, etc.) [19], or computing separable functions [20].

For our objective, DDAAs for the distributed computation of sums and averages, such as the *push-sum algorithm* [21], are most relevant. The potential of DDAAs for providing a high degree of resilience is mentioned in the literature at various places (see, e. g., [22]), but we are not aware of any work which investigates the challenges arising when trying to tap the full potential. Relevant existing DDAAs are surveyed in Section 2.

Only recently, distributed algorithms for more complex matrix computations based on DDAAs, such as the *dmGS* algorithm for distributed QR factorization [23] or a distributed orthogonal iteration method [24, 25] (both based on the push-sum algorithm) have been designed and compared with state-of-the-art parallel algorithms.

1.3. Synopsis

When trying to construct fault tolerant distributed algorithms for matrix computations based on sequences of DDAAs, resilience aspects first have to be addressed at the level of a single aggregation algorithm. Consequently, in Section 2 we survey relevant existing DDAAs and discuss their strengths and weaknesses in terms of fault tolerance. We then focus on improving fault tolerance in Section 3. We present the new *push-flow algorithm*, which has superior fault tolerance properties compared to existing distributed data aggregation algorithms. As a prototypical example for matrix computations based on DDAAs, we present the new distributed orthogonalization algorithm *rdmGS* which is resilient to node failures. In Section 4, we discuss the scalability of selected DDAAs and of distributed orthogonalization methods. Section 5 concludes the paper.

2. Distributed Data Aggregation Methods

Over the last decade, several distributed data aggregation algorithms based on randomized communication schedules have been proposed. Those methods were traditionally motivated by networks with unreliable communication links, which precludes the usage of classical parallel algorithms with fixed communication schedules. Recently, there are also efforts in introducing self-healing mechanisms into such algorithms. For a structured discussion of the differences between the various methods we distinguish henceforth the following types of *failures* in the system under consideration. Note that we order the failure types according to increasing difficulty for recovering from them at the algorithmic level.

(F1) Reported temporary unavailability of links/nodes
(F2) Unreported loss or corruption of a message
(F3) Reported permanent node or link failures
(F4) Unreported corruption of local data (e. g., bit flip)
(F5) Unreported permanent node failures

While dealing with (F1) is generally easy for any randomized method with flexible communication schedules, the coverage of (F2)–(F5) is a lot harder since those failures usually introduce a (temporary) error from which the system has to recover properly. In the case of (F5) one has to additionally define whether the initial data of a failed node should be included in the target aggregate or not, where the former is usually harder.

2.1. Existing Methods

A basic approach for the distributed computation of aggregates is the *push-sum algorithm* [21] where each node i iteratively updates a local vector $v_i := (s_i, w_i)$. The s_i are initialized with the values x_i to be aggregated, and the w_i are weights. The initial values of the weights w_i determines the type of aggregation operation: For computing $\sum_{i=1}^{N} x_i$, all weights have to be either initialized identically to $w_i = 1/N$, or to $w_i = 0$ for all nodes except one with weight $w_0 = 1$. For computing the average $\sum_{i=1}^{N} x_i/N$, all weights have to be initialized identically to $w_i = 1$. By consecutively sending fractions of the local vector v_i to randomly chosen neighbors (which add the received values to their local values), all local estimates s_i/w_i converge linearly either to the sum or to the average of the distributed values [21].

In [26] a more robust version of the push-sum algorithm, called *LiMoSense*, is derived by keeping a history (i. e., the sum) of sent and received values along each communication link. By always sending the full history the receiver of a message can easily tolerate missed (or wrong) values. To keep the steadily growing histories small, a bidirectional cancellation operation is proposed in [26].

A third approach is *flow updating* [27]. The underlying idea is that the nodes keep their initial values local and only share *flows* with their neighbors. For each communication link, both attached nodes maintain a *flow variable* which represents the overall balance of communicated local values along this link. For communicating local values along a link, the sender does not transmit the local values directly, but instead adds them to the corresponding flow variable and transmits the flow variable. The receiver updates its own flow variable by the negated received flow. Consequently, as in network flow algorithms, the overall flow across each link is zero (*flow conservation*) if no failures occur. This flow conservation is the key idea of this method for recovering from failures, since recovering from a failure corresponds to (re-)establishing flow conservation, which is achieved after each successful communication across a link.

In flow updating, a node locally approximates the aggregate by first subtracting the sum over all flow variables it maintains from its initial value and then averaging this local approximation and the current local estimates of all its neighbors (which are exchanged together with the flows). However, it has some drawbacks in terms of convergence speed and there is no formal analysis given in [27], whereas for the push-sum algorithm [21] and for LiMoSense [26] proofs of correctness and convergence (speed) have been given.

Table 1 summarizes which failure types the existing distributed data aggregation algorithms can handle.

	Push-sum	LiMoSense	Flow updating
(F1)	√	√	√
(F2)	—	√	√
(F3)	—	√	√
(F4)	—	—	√
(F5)	—	—	—

Table 1: Resilience properties of existing DDAAs

2.2. Strengths and Weaknesses

Thanks to their randomized communication schedules, all these methods will *always* produce at least approximate results even if hardware failures occur. The important issue in this context is *mass conservation*, which means that distributed data aggregation algorithms converge to the true aggregate only if all of the initial information is preserved over the whole aggregation process. Since a change of mass (usually mass loss, but also an increase of mass is theoretically possible, e. g. in the case of a bit flip) always results in a corresponding loss of achievable accuracy we are interested in methods which are able to ensure mass conservation or can recover from a change of mass even if failures of types (F1)–(F5) occur.

Note that strict mass conservation in the sense of a network-wide invariant seems to be impossible under our assumptions since the occurrence of (F2)–(F5) will generally result in mass loss. So the methods we aim for are designed for an autonomous full *recovery* from mass loss, which we denote as *weak mass conservation* since it weakens the strict (theoretical) form of mass conservation. Accordingly, those methods ensure strict mass conservation in failure-free scenarios and weak mass conservation if failures occur, i. e., they are able to recover from a change of mass by a proper self-healing mechanism.

Although the push-sum algorithm is independent of the availability of specific communication links, it has no built-in mechanism for mass conservation and therefore it cannot recover from (F2)–(F5). LiMoSense can recover from (F2)–(F3) due to the redundancy in keeping and sending complete histories (cf. [26]). Flow updating does not (directly) rely on the transmission of redundant messages and it can also handle (F4), because a local error like a bit flip in a flow value is simply corrected by (re-)establishing a valid flow (cf. [27, 28]). Since LiMoSense is a direct generalization of the push-sum algorithm, it also delivers the same fast convergence speed, as opposed to flow updating, which showed an uncompetitive convergence speed in our experiments (cf. Section 4.1).

To demonstrate the strength of the flow concept, in Section 3.1 we introduce the *push-flow algorithm*, which fully exploits its advantages while preserving the convergence speed of the push-sum algorithm.

3. Improving Resilience

In the following, we present two new distributed algorithms which are more resilient than existing algorithms in terms of the most challenging failure types (F3)–(F5). In Section 3.1, we concentrate on a single instance of a distributed data aggregation algorithm and introduce the *push-flow algorithm*, which can recover from failure types (F1)–(F4). In Section 3.2 we turn our attention to the level of complete matrix problems and introduce *robust dmGS* for distributed QR factorization or orthogonalization which can recover from failure types (F1)–(F4), and in some scenarios even from failure type (F5).

3.1. The Push-Flow Algorithm

Despite several drawbacks of the flow updating algorithm in terms of uncompetitive scalability and convergence speed (cf. Section 4.1) as well as lack of formal analysis in [27], the principal idea of keeping the initial mass locally and sharing flows (instead of mass) is promising and has several conceptual advantages over keeping histories, as in LiMoSense. First, a valid flow across a link can be established from any direction, while in a history-based approach a specific direction is needed to tolerate failures like (F2). Second, in contrast to the steadily increasing history values, flow variables remain bounded by definition. Third, flow-based approaches achieve higher fault tolerance, since history-based approaches are limited to transmission related failures (F1)–(F3), whereas a flow-based approach can also recover from purely local failures of a node like (F4).

Motivated by this observation, we integrate the flow concept into the push-sum algorithm by translating each transmission of mass, i. e., the direct transmission of (fractions of) local values, into a transmission of flow in a similar way as it is done in the flow updating method. The result is a variant of the push-sum algorithm, which is equivalent to the push-sum algorithm in the absence of failures and exhibits improved resilience if failures occur. The resulting *push-flow algorithm* is shown in Figure 1. Each node i maintains a two dimensional flow vector $f_{i,j}$ for every neighbor j in its neighborhood $\mathcal{N}_i$ whose elements can be interpreted as the balance of mass which was communicated between nodes i and j. Moreover, each node i maintains a two dimentionsl vector $v_i = (x_i, w_i)$ which contains the local initial value x_i and the local weight w_i. The initial values for the local weights w_i are the same as in the push-sum algorithm.

```
Input: Local initial value $x_i$ and local weight $w_i$ for each node $i$
Output: Estimate $e_i(1)/e_i(2)$ of global aggregate $(\sum_{k=1}^N x_k)/(\sum_{k=1}^N w_k)$
 1: ...initialization...
 2: $v_i \leftarrow (x_i, w_i)$
 3: for each $j \in \mathcal{N}_i$ do
 4:     $f_{i,j} \leftarrow (0, 0)$
 5: end for
 6: ...on receive...
 7: for each received pair $f_{j,i}$ do
 8:     $f_{i,j} \leftarrow -f_{j,i}$
 9: end for
10: ...on send...
11: $k \leftarrow$ choose a random neighbor $k \in \mathcal{N}_i$
12: $e_i \leftarrow v_i - \sum_{j \in \mathcal{N}_i} f_{i,j}$
13: $f_{i,k} \leftarrow f_{i,k} + e_i/2$
14: send $f_{i,k}$ to node $k$
```

Figure 1: The push-flow algorithm for the local computation of a global aggregate. $\mathcal{N}_i$ denotes node i's neighborhood.

At every point in time, the current local mass e_i at a node i is (in contrast to flow updating) computed as the difference between the initial vector v_i and the sum over all flows $f_{i,j}$, i. e., $e_i = v_i - \sum_{j \in \mathcal{N}_i} f_{i,j}$. Consequently, analogously to the push-sum algorithm, the local estimate of the global aggregate can be computed by dividing the first component of the vector e_i by its second component, i. e., by forming $e_i(1)/e_i(2)$.

It is easily verified that the push-flow algorithm is essentially equivalent to the push-sum algorithm in failure-free networks, since for identical communication patterns both algorithms produce identical local estimates of the global aggregate The equivalence to push-sum also highlights that the local estimates are not computed as average over the estimates of the neighboring nodes like in flow updating. Consequently, the push-flow algorithm preserves the fast convergence of the push-sum algorithm and does not exhibit the disadvantages of flow updating in terms of convergence speed.

In the presence of failures, the push-flow algorithm benefits from the resilience and self-healing capabilities inherent in the flow concept. In particular, recovery from failures (F3) can be achieved if the neighbors of the failed node set the corresponding flow variables to zero. Therefore, the push-flow algorithm also excludes the local data of a failed node from the final aggregate similar to the resilient methods discussed in Section 2. In the case of failures (F4) where some local flow values are corrupted, e. g., because of a bit flip, the nodes involved will recover the next time a correct communication involving this variable happens, like in the case of transmission related failures (F2). A full proof of correctness and convergence of the push-flow algorithm proceeds analogously to the ones presented in [21, 26] for the push-sum algorithm and LiMoSense, respectively.

In conclusion, the push-flow algorithm achieves the best resilience among all existing DDAAs and preserves the convergence speed of the push-sum algorithm.

3.2. Fault Tolerant Orthogonalization

The resilient DDAAs we discussed so far handle permanently failed nodes by excluding their local data from the final aggregate and thus no redundancy in storing the original data is required. In situations where it is required that the original data of *all* nodes is aggregated, redundancy in the original data has to be introduced.

The distributed orthogonalization of the columns of the matrix $A \in \mathbb{R}^{n\times m}$ over a system with N nodes is a prototypical example where the loss of original data usually resulting from a permanent node failure *cannot* be tolerated. Distributed modified Gram-Schmidt orthogonalization (*dmGS*) for computing the QR decomposition $A = QR$ with $Q \in \mathbb{R}^{n\times m}$ and $R \in \mathbb{R}^{m\times m}$ has been presented in [23]. It assumes that A is distributed rowwise over the N nodes. If a node permanently fails during the execution of dmGS, its local part of A is also permanently lost and as a consequence it becomes impossible to orthogonalize all original vectors. We illustrate in the following how the resilience of this approach can be improved by introducing redundancy in storing the original data. We present *robust dmGS* (*rdmGS*, see Fig. 2), which produces a complete and accurate QR factorization of A even if one or several nodes fail during the computation.

3.2.1. Introducing Redundancy

The pivotal idea of rdmGS is to maintain redundant copies of all nodes' relevant local data at more than one active node at all times. By construction, dmGS automatically computes *all* entries of R at all nodes of the system [23], and thus no specific measures are needed for backing up data of R. Every node k is responsible for a subset of the rows of A and for the parts of the corresponding rows of Q which have been computed so far. We call this data node k's *primary data*. At every point in time, $r-1$ backup copies of node k's primary data are stored on $r-1$ distinct other active nodes which act as backup nodes for node k. The parameter r is a measure for the resilience as well as for the overhead of rdmGS. Larger r allows for tolerating more simultaneous node failures, albeit at higher cost.

Node k may also act as a backup node for one or more other nodes in the system. The corresponding local data at node k is called node k's *backup data*. We call a node k, which backs up node l's data, l's *guardian*, and l in turn k's *protégé*.

If node k fails, its primary data is still available on its $r-1$ guardians. One of these guardians takes over the primary responsibility for this data, and selects another active node (usually in its neighborhood) to replace itself as guardian for the primary data of the failed node k. As a result, again r copies of the data of the failed node k exist in the system.

In the process of local computation in each node not only the primary data is updated, but *also* all local backup data. Since the local results of the DDAA are not necessarily identical over all nodes, the $r-1$ instances of backup data and the corresponding primary data may differ slightly, but no extra data communication is needed for the backup structure as long as no node failure occurs. Upon termination, node k considers only its primary data as part of the final result.

This concept operates successfully under the following assumptions: (*i*) the topology of the system stays connected despite all occurring node failures, (*ii*) a reliable and efficient mechanism is available for determining whether a node (usually in the neighborhood) is alive (active) or not, and (*iii*) in case of permanent failures nodes fail *neatly*, i. e., if a node fails *within* the execution of a DDAA, this failure has to be reported immediately, and the failing node i has to send some of its local values at least to one of its neighbors in order to ensure mass conservation.

3.2.2. Ensuring Mass Conservation

The concrete resilience properties of the distributed orthogonalization method depend on the choice of the DDAA (cf. Table 1). If the push-sum algorithm is used as a building block for rdmGS, reliable communication is required in order to ensure mass conservation. Using the push-flow algorithm as underlying aggregation algorithm for rdmGS instead of the push-sum algorithm allows for recovering from mass loss caused by *temporary* node or link failures and thus increases the resilience of rdmGS to these types of failures.

The weights w_i play an important role, since their initial values determine the type of aggregation operation (cf. Section 2.1). However, in the presence of node failures, both initialization variants for summing the local data across the nodes are unsuitable, because the first initialization variant depends on the system size N, which will not remain constant, and because the second initialization variant introduces a single point of failure (the node with the initial value $w_i = 1$). Consequently, in rdmGS we initialize all $w_i = 1$ such that the DDAA computes the average $\sum_{i=1}^{N} x_i/N$ across the system and we distribute the value N of the initial system size to all nodes at the beginning of the algorithm. After termination of each DDAA, each node scales its local result by N for computing the sum from the average.

Beyond the resilience properties of the DDAA used, we need to ensure that none of the original data gets lost when a node fails. For that purpose, we introduce *virtual nodes*. Initially, the system contains N active physical nodes, and each of them corresponds to exactly one virtual node. Whenever a physical node l fails during the computation, another active physical node k has to take over all virtual nodes which physical node l was responsible for. Thus, if node failures occur in the process of rdmGS, active physical nodes take over responsibility for more than one virtual node. In order to ensure mass conservation, the sum of the weights over all active physical nodes needs to remain equal to the initial system size N. In order to achieve this, the surviving node k from before needs to increase its local weight by the weight of the failing node l ($w_k \leftarrow w_k + w_l$). The resulting mass conservation in the system guarantees that the DDAA can converge to the average of the original values of all N initially active nodes.

The resulting algorithmic structure of rdmGS is outlined in Fig. 2. Each execution of a DDAA is preceded by a fail-checking phase, where every node k checks whether all of its protégés and its guardians are alive. If yes, node k can proceed. If no, the following actions have to be taken: (*i*) If a protégé l of node k has failed, node k has to take over primary responsibility for l's data and l's weight has to be added to k's local weight. Note that l's local weight represents for how many virtual nodes the physical node l has been responsible for. Moreover, all updates of local weight as well as the updated primary data of node k (parts of A and Q) have to be sent to k's guardian. (*ii*) If a guardian l of node k has failed, node k has to select a new guardian and send its local weight and primary data to it for backup. Among other aspects, the selection process of a new guardian should be influenced by the objective to balance the load across the active nodes.

```
Input: $A \in \mathbb{R}^{n \times m}$, node $k$ stores $n/N$ rows of primary data
Output: $Q \in \mathbb{R}^{n \times m}$, $R \in \mathbb{R}^{m \times m}$
 1: for $i = 1$ to $m$ do (in node $k$)
 2:   ...check for node failures, update $P_k$ and $B_k$ ...
 3:   $x(k) \leftarrow \sum_{p \in P_k} A(p,i)^2$
 4:   $s_k \leftarrow \textbf{\textit{DDAA}}(x)$
 5:   $R_k(i,i) \leftarrow \sqrt{s_k}$
 6:   for each $p \in P_k$ do
 7:     $Q(p,i) \leftarrow A(p,i)/R_k(i,i)$
 8:   end for
 9:   for each $b \in B_k$ do
10:     $Q(b,i) \leftarrow A(b,i)/R_k(i,i)$
11:   end for
12:   for $j = i+1$ to $m$ do
13:     ...check for node failures, update $P_k$ and $B_k$ ...
14:     $x(k) \leftarrow \sum_{p \in P_k} Q(p,i)A(p,j)$
15:     $R_k(i,j) \leftarrow \textbf{\textit{DDAA}}(x)$
16:     for each $p \in P_k$ do
17:       $A(p,j) \leftarrow A(p,j) - Q(p,i)R_k(i,j)$
18:     end for
19:     for each $b \in B_k$ do
20:       $A(b,j) \leftarrow A(b,j) - Q(b,i)R_k(i,j)$
21:     end for
22:   end for
23: end for
```

Figure 2: The rdmGS algorithm: P_k denotes the set of indices for rows of A and Q which are primary data on node k. B_k denotes the set of indices for rows of A and Q which are backup data on node k. "DDAA(x)" denotes the execution of a distributed data aggregation algorithm on the distributed vector x.

There are only two specific scenarios of node failures which rdmGS cannot recover from independently of the concrete DDAA used. On the one hand, if a node k and all of its $r-1$ guardians fail permanently before even a single of these failures is detected, then rows of A and Q are lost and cannot be recovered any more. On the other hand, if a node k fails permanently after passing the fail-checks of all of its $r-1$ guardians, but before starting the next aggregation process, mass conservation is violated because k's primary data is not used within that aggregation process and k's guardians are not aware of it. Note that the probability for these two scenarios to happen can be reduced by increasing the overhead in terms of fail-checking frequency.

3.2.3. Simulation Results

In order to illustrate its properties, we developed a simulation model for the rdmGS algorithm with $r = 2$ (each node has one guardian) in the ns-3 network simulator. Simulation results are shown for an asynchronous wired network of 32 nodes arranged in a five-dimensional hypercube. The times until failure of a node are exponentially distributed with mean λ, and nodes fail neatly (cf. Section 3.2.1). As distributed data aggregation algorithm we used the push-sum algorithm. A detailed comparison of push-sum algorithm and push-flow algorithm as building block for rdmGS is work in progress. We varied the maximum numbers $t_{\max}$ of iterations per push-sum algorithm. For a given λ, every value $t_{\max} \in [100 : 50 : 600]$ has been simulated 200 times with different initializations of the random number generator, but with the same underlying topology.

In Fig. 3, the accuracy of rdmGS in terms of the relative factorization error $\|A - QR\|_F / \|A\|_F$ is illustrated for $\lambda = 12$ [s]. With this value of λ, between zero and 17 of the 32 nodes

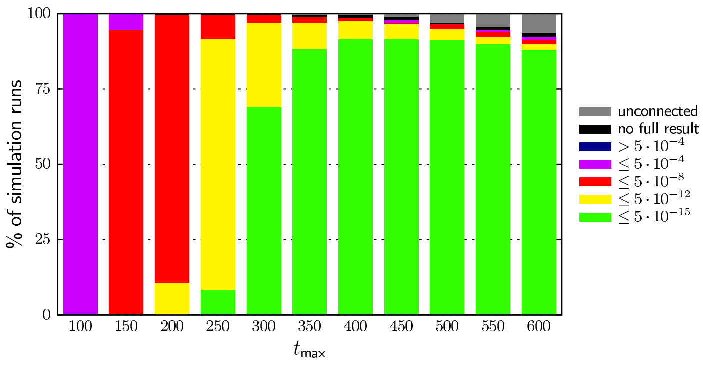

Figure 3: Relative factorization error of rdmGS for $\lambda = 12$ [s]. "unconnected" refers to scenarios where node failures caused the hypercube topology to disconnect into two separate components.

failed per simulation run, on average 5.82 over all values of $t_{\max}$. The average number of failed nodes per simulation run increases with the value of $t_{\max}$. For $t_{\max} = 100$ it is slightly below 2, for $t_{\max} = 400$ around 6.7, and for $t_{\max} = 600$ almost 10 (see Fig. 4).

Fig. 3 illustrates that for small $t_{\max}$ the low accuracy of each push-sum algorithm leads to low accuracy of rdmGS. As $t_{\max}$ increases, the factorization error decreases and machine precision is reached in at least 88% of all runs for $t_{\max} \geq 350$.

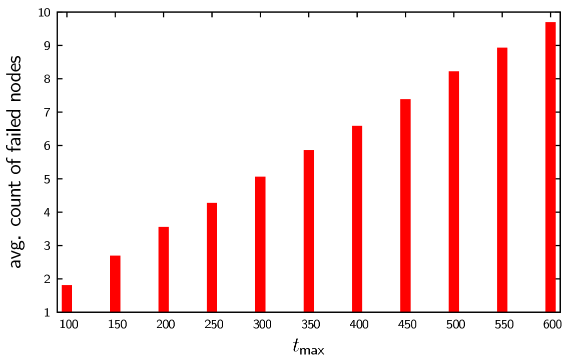

Figure 4: Average number of node failures per simulation for $\lambda = 12$ [s] and different $t_{\max}$. Averaged over all simulation runs and all $t_{\max}$, 5.82 nodes failed per simulation run.

However, larger $t_{\max}$ leads to longer runtimes and thus also to a higher chance of node failure constellations which rdmGS cannot recover from (cf. Section 3.2.2). Consequently, the fraction of simulation runs where rdmGS does not produce the full matrix Q due to node failures slowly grows from 0.5% for $t_{\max} = 200$ to 7.5% for $t_{\max} = 600$.

We observe a similar behavior for the orthogonality of Q, measured in terms of $\|I - Q^\top Q\|_F / \sqrt{m}$ in Fig. 5: it gets better for larger $t_{\max}$, but also the chance increases that rdmGS does not produce the full matrix Q due to node failure constellations which it cannot recover from. Summarizing, Figs. 3 and 5 illustrate that there is a certain range of $t_{\max}$ where factorization accuracy and orthogonality achieved by rdmGS are excellent and in most cases the rdmGS algorithm recovers successfully from the node failures. For the simulation setup considered this range is around $t_{\max} = 400$.

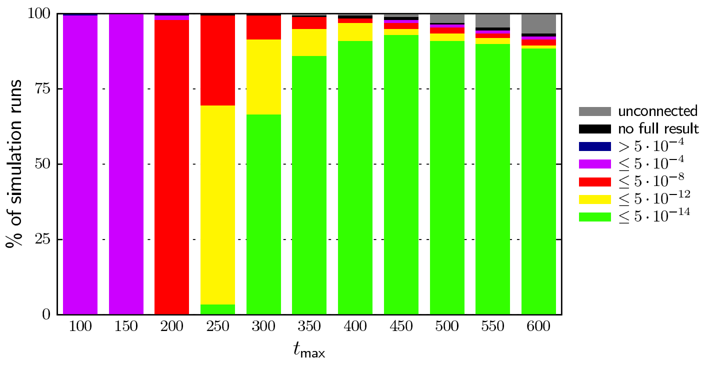

Figure 5: Orthogonality of rdmGS for $\lambda = 12$ [s]. “unconnected” refers to scenarios where node failures caused the hypercube topology to disconnect into two separate components.

4. Scalability

In this section, we discuss the scalability of the methods we developed in this paper. For distributed data aggregation algorithms, scalability in terms of number of nodes corresponds to scalability in terms of problem size, since data from all nodes is aggregated, whereas for the distributed orthogonalization method, scalability in terms of number of nodes and in terms of problem size need to be considered separately.

4.1. Distributed Data Aggregation Algorithms

Generally speaking, the convergence speed of gossip-based algorithms depends on the size N of the system, on properties of the topology of the system, such as diameter or expansion (cf. [29]), and also on the approximation error ϵ. In a failure-free environment, the push-sum and the push-flow algorithm require $O(\log N + \log \epsilon^{-1})$ iterations for approximating the true aggregate with an error below ϵ at each node if every node can communicate with any other node in the system [21].

For more general topologies, a theoretical framework for the analysis of distributed aggregation algorithms is developed in [13]. It is also shown in [13] how to derive algorithms with optimal convergence speed for arbitrary topologies. Although those algorithms are not prepared to deal with failure types (F2)–(F5), it is interesting to observe that for communication graphs with good expansion the number of iterations required for convergence is also $O(\log N + \log \epsilon^{-1})$ [13]. Hypercube topologies are examples for topologies with good expansion properties. Consequently, the results of [13] actually show that for network topologies which allow for fast reduction operations, e. g., k-ary n-cubes [30], randomized gossip algorithms scale asypmptotically equally well as all-to-all reduction operations.

The system topology may have different influences on convergence speed and on resilience properties, though. Since diameter and expansion of a graph are related to the node degrees, a higher average node degree tends to speed up convergence of DDAAs. However, a higher node degree may have drawbacks in terms of recovery from a violation of mass conservation due to a temporary failure: Flow-based DDAAs (see Section 2) in principle have the ability to recover from such a violation of mass conservation at the time of the next failure-free communication along the link which was affected by the failure. Since in gossiping algorithms each node usually selects its communication partners according to a uniform distribution, a higher node degree increases the expected time until recovery from a temporary failure. As a consequence, in order to combine fast convergence with quick recovery from a violation of mass conservation, the network graph should have small node degrees *and* good expansion properties. Examples of topologies with these properties are k-ary n-cubes.

Fig. 6 summarizes simulation results of the scaling behavior for increasing number of nodes on a system *without* failures. It shows the number of iterations required by the distributed data aggregation algorithms discussed in Sections 2 and 3.1 for computing the average of local values with an error at the order of machine precision. Note that without failures, the push-sum algorithm, LiMoSense and our new push-flow algorithm are basically equivalent and thus their scaling behavior is identical. These three algorithms clearly scale better than the flow updating algorithm. Besides the theoretically predicted $O(\log N)$ behavior on topologies where every node can communicate with any other node (e. g., on a fully connected network), we also see that even on the weaker connected hypercube topology the asymptotic behavior of the push-flow algorithm is the same as

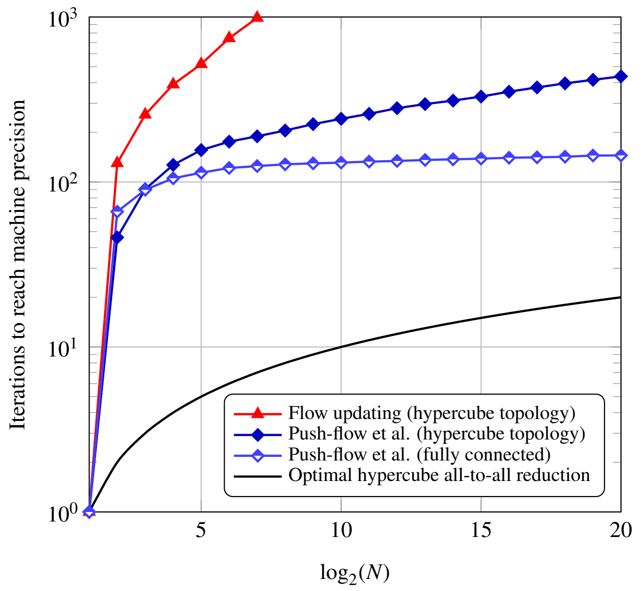

Figure 6: Scalability of DDAAs with the number of nodes on a hypercube topology compared to the optimal hypercube all-to-all reduction [31] (no failures; thus push-flow, push-sum and LiMoSense are equivalent)

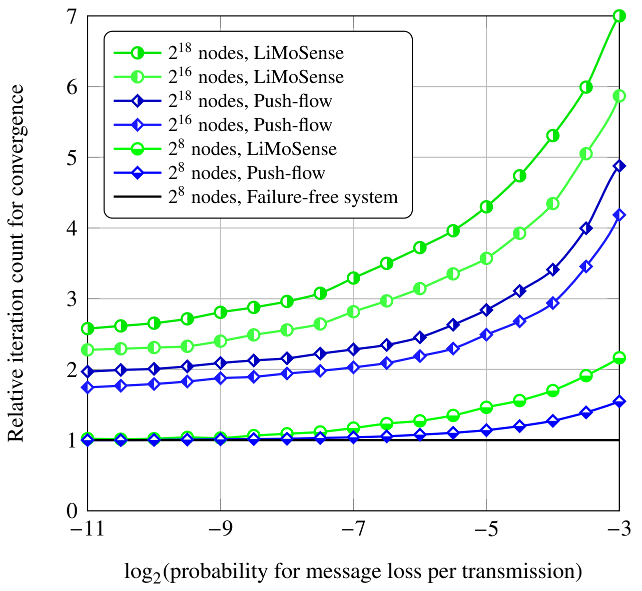

Figure 7: Resilience and scalability of LiMoSense and push-flow for averaging on a hypercube topology relative to failure-free system with 2^8 nodes

for the optimal all-to-all reduction operation (cf. [31]), and the actual number of iterations required differs by about one order of magnitude.

Fig. 7 illustrates the increase in the number of iterations required by LiMoSense and the push-flow algorithm due to increasing node failure rate for different system sizes (numbers of nodes). The values shown in the figure are averages over 100 simulation runs. We see that the push-flow algorithm always requires fewer iterations for convergence than LiMoSense. Compared to a failure free environment, (*i*) for lower numbers of nodes, the push-flow algorithm hardly experiences any increase in the number of iterations, and (*ii*) for higher numbers of nodes, the increase in the number of iterations tends to be very modest for low and medium failure rates and grows up to a factor of five for the push-flow algorithm for high failure rates. The node counts were chosen in order to allow for a rough comparison with an experimental case study for the overhead of checkpointing and restarting given in [3]: In that case study, the overhead of checkpointing and restarting was larger than a factor of two for $2^{16} \approx 65\,000$ nodes, which corresponds to failure probabilities higher than 2^{-7} for the push-flow algorithm. For $2^{18} \approx 260\,000$ nodes, the overhead of checkpointing and restarting in the case study presented in [3] was larger than a factor of eight, which corresponds to very high failure probabilities above 2^{-3} for the push-flow algorithm. This indicates advantages of the gossip-based approaches investigated in this paper compared to classical checkpointing and restarting strategies in terms of lower overhead for fault tolerance.

4.2. *Distributed Orthogonalization*

The scalability of rdmGS with the number of nodes is determined by the scalability of the specific DDAA used, because all communication between nodes is concentrated in the data aggregation processes. As shown before, if they are based on the push-sum algorithm or on the push-flow algorithm, the number of iterations needed will grow like $O(\log N)$ on many topologies, which is asymptotically the same behavior as optimal parallel all-to-all reduction operations.

In terms of scalability with the problem size, we note the following: Increasing the number of rows n for fixed N corresponds to more rows handled per node and thus increases the local computation cost, but does not affect the communication cost. Since local computation is usually much faster than communication, this leads to good scaling behavior and tends to improve the convergence speed due to increased accuracy in the data aggregation processes. Increasing the number of columns m for fixed N is less attractive, since in the version of rdmGS described in this paper the number of DDAAs invoked grows quadratically with m, which can become a limiting factor for large m. However, we are currently developing an improvement of the underlying dmGS algorithm which requires only $O(m)$ DDAAs [25] and thus improves scalability with increasing m.

5. Summary and Conclusions

We have shown that distributed algorithms based on randomized communication schedules can be very attractive for potentially unreliable or unstable large-scale systems, in particular in terms of fault tolerance and resilience. We have presented the new push-flow algorithm for distributed computation of sums or averages, which has better resilience properties than existing distributed data aggregation algorithms. Moreover, we have developed the distributed orthogonalization method rdmGS on top of distributed data aggregation algorithms, which is very resilient to various types of failures and capable of producing fully accurate results even if several nodes fail permanently. Simulation experiments showed that even when 30% of the nodes of the system fail on average, rdmGS produces results accurate to machine precision in at least 88% of the simulation runs.

Investigation of remaining questions in terms of the potential of these new randomized algorithms for high performance requirements as well as a quantitative investigation of the influence of asynchrony on their performance is work in progress.

Acknowledgment

This work was supported by the Austrian Science Fund FWF in project S10608 (NFN SISE).

References

[1] G. E. Fagg, E. Gabriel, Z. Chen, T. Angskun, G. Bosilca, J. Pjesivac-Grbovic, J. J. Dongarra, Process fault tolerance: Semantics, design and applications for high performance computing, International Journal of High Performance Computing Applications 19 (4) (2005) 465–477.

[2] J. T. Daly, A higher order estimate of the optimum checkpoint interval for restart dumps, Future Gener. Comput. Syst. 22 (2006) 303–312.

[3] M. Varela, K. Ferreira, R. Riesen, Fault-tolerance for exascale systems, in: Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS), 2010 IEEE International Conference on, 2010, pp. 1–4.

[4] J. Plank, K. Li, M. Puening, Diskless checkpointing, Parallel and Distributed Systems, IEEE Transactions on 9 (10) (1998) 972 –986.

[5] J. Plank, Improving the performance of coordinated checkpointers on networks of workstations using raid techniques, in: Reliable Distributed Systems, 1996. Proceedings., 15th Symposium on, 1996, pp. 76 –85.

[6] A. Guermouche, T. Ropars, E. Brunet, M. Snir, F. Cappello, Uncoordinated checkpointing without domino effect for send-deterministic MPI applications, in: Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International, 2011, pp. 989 –1000.

[7] K. Ferreira, R. Riesen, R. Oldfield, J. Stearley, J. Laros, K. Pedretti, T. Brightwell, rMPI: increasing fault resiliency in a message-passing environment, Tech. rep., no. SAND2011-2488, Sandia National Laboratories (2011).

[8] C. Engelmann, H. H. Ong, S. L. Scott, The Case for Modular Redundancy in Large-Scale High Performance Computing Systems, in: Proceedings of the 27th IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN) 2009, ACTA Press, Calgary, AB, Canada, 2009, pp. 189–194.

[9] C. Wang, F. Mueller, C. Engelmann, S. L. Scott, Proactive process-level live migration in hpc environments, in: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, SC '08, IEEE Press, Piscataway, NJ, USA, 2008, pp. 43:1–43:12.

[10] K.-H. Huang, J. Abraham, Algorithm-based fault tolerance for matrix operations, Computers, IEEE Transactions on C-33 (6) (1984) 518 –528.

[11] Z. Chen, J. Dongarra, Algorithm-based fault tolerance for fail-stop failures, Parallel and Distributed Systems, IEEE Transactions on 19 (12) (2008) 1628 –1641.

[12] Z. Chen, Algorithm-based recovery for iterative methods without checkpointing, in: Proceedings of the 20th international symposium on High performance distributed computing, HPDC '11, ACM, New York, NY, USA, 2011, pp. 73–84.

[13] S. Boyd, A. Ghosh, B. Prabhakar, D. Shah, Randomized gossip algorithms, IEEE Transactions on Information Theory, Special issue of IEEE Transactions on Information Theory and IEEE ACM Transactions on Networking 52 (2006) 2508–2530.

[14] A.-M. Kermarrec, M. van Steen, Gossiping in distributed systems, SIGOPS Oper. Syst. Rev. 41 (2007) 2–7.

[15] A. Dimakis, A. Sarwate, M. Wainwright, Geographic gossip: Efficient averaging for sensor networks, IEEE Trans. Signal Processing 56 (3) (2008) 1205–1216.

[16] T. Aysal, M. Yildiz, A. Sarwate, A. Scaglione, Broadcast gossip algorithms for consensus, IEEE Trans. Signal Processing 57 (7) (2009) 2748 –2761.

[17] R. Karp, C. Schindelhauer, S. Shenker, B. Vocking, Randomized rumor spreading, in: Proceedings of the 41st Annual Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, USA, 2000, pp. 565–574.

[18] M. Jelasity, A. Montresor, O. Babaoglu, Gossip-based aggregation in large dynamic networks, ACM Trans. Comput. Syst. 23 (2005) 219–252.

[19] D. Shah, Gossip algorithms, Found. Trends Netw. 3 (2009) 1–125.

[20] D. Mosk-Aoyama, D. Shah, Computing separable functions via gossip, in: Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing, PODC '06, ACM, New York, NY, USA, 2006, pp. 113–122.

[21] D. Kempe, A. Dobra, J. Gehrke, Gossip-based computation of aggregate information, in: FOCS '03: Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, 2003, pp. 482–491.

[22] A. Olshevsky, J. N. Tsitsiklis, Convergence speed in distributed consensus and averaging, SIAM J. Control Optim. 48 (2009) 33–55.

[23] H. Straková, W. N. Gansterer, T. Zemen, Distributed QR factorization based on randomized algorithms, in: Proceedings of the 9th International Conference on Parallel Processing and Applied Mathematics, Part I, Vol. 7203 of Lecture Notes in Computer Science, 2012, pp. 235–244.

[24] D. Kempe, F. McSherry, A decentralized algorithm for spectral analysis, Journal of Computer and System Sciences 74 (1) (2008) 70 – 83.

[25] H. Straková, W. N. Gansterer, A distributed eigensolver based on randomized communication, Technical Report RLCTA-2012-1, University of Vienna, Research Group Theory and Applications of Algorithms, *submitted* (2012).

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

[27] P. Jesus, C. Baquero, P. S. Almeida, Fault-tolerant aggregation by flow updating, in: Proceedings of the 9th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems, DAIS '09, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 73–86.

[28] P. Jesus, C. Baquero, P. Almeida, Fault-tolerant aggregation for dynamic networks, in: 29th IEEE Symposium on Reliable Distributed Systems, 2010, pp. 37 – 43.

[29] S. Hoory, N. Linial, A. Wigderson, Expander graphs and their applications, Bull. Amer. Math. Soc. 43 (2006) 439–561.

[30] W. J. Dally, Performance analysis of k-ary n-cube interconnection networks, IEEE Trans. Comput. 39 (1990) 775–785.

[31] A. Grama, G. Karypis, V. Kumar, A. Gupta, Introduction to Parallel Computing, 2nd Edition, Addison Wesley, 2003.