Communication Efficient Checking of Big Data Operations

Lorenz Hübschle-Schneider and Peter Sanders
Institute of Theoretical Informatics
Karlsruhe Institute of Technology, Germany
{huebschle,sanders}@kit.edu

20th November 2018

Abstract

We propose fast probabilistic algorithms with low (*i.e.*, sublinear in the input size) communication volume to check the correctness of operations in Big Data processing frameworks and distributed databases. Our checkers cover many of the commonly used operations, including sum, average, median, and minimum aggregation, as well as sorting, union, merge, and zip. An experimental evaluation of our implementation in Thrill (Bingmann et al., 2016) confirms the low overhead and high failure detection rate predicted by theoretical analysis.

1 Introduction

Recently, Big Data processing frameworks like Apache Spark [1], Apache Flink [2] and Thrill [3] have surged in popularity and are widely used to process of large amounts of data. Computation and data are distributed over a large number of machines connected by a fast network, and processing is based on collective operations that transform datasets. As compute performance and memory capacity continue to grow ("Moore's Law") while simultaneously becoming ever cheaper, oftentimes all data can be held in main memory. Combined, these advances let users process enormous amounts of data quickly. However, as the number of machines increases, so does the rate of hardware failures and thus the importance of *fault tolerance*, as well as the difficulty of writing correct programs that handle all edge cases. Some frameworks, *e.g.* Apache Spark, can deal with failing machines [1], but none of the popular solutions can detect *silent errors*. These can be the result of subtle errors in the programming, but also spontaneous bitflips in memory ("soft errors"), caused for example by cosmic rays [4]—a concern that can largely be mitigated by using ECC RAM, but at the cost of increased hardware expenditure. We propose probabilistic algorithms to detect silent failures in the frameworks' operations with little overhead, which we refer to as *checkers*. These checkers verify the integrity of the computation while treating the operation as a black box, *i.e.* independent of its implementation. Our checkers cover many of the commonly used operations of current-generation data-parallel computing frameworks.

Consider the options available to a programmer who wants to increase users' confidence in the correctness of her program. Formal verification would be ideal, but very difficult for complex programs and does not address hardware errors. Testing can help the programmer to avoid mistakes, but cannot prove the absence of bugs. Lastly, she can write a small and fast program that verifies

Table 1: Our main results. Parameters: input size n; number of processing elements (PEs) p; failure probability δ; machine word size w; trade-off parameter d; communication startup cost α; communication cost per bit β

Operation	**Broadcast Result?**	**Certificate Required?**	**Checker Running Time $\mathcal{O}(\cdot)$**
Sum/Count Agg.	no	no	$\left(\frac{n}{p} + \beta d w\right) \log_d \frac{1}{\delta} + \alpha \log p$
Average Agg.	no	dist	same as above
Median Agg.	yes	yes[a]	same as above
Minimum Agg.	yes	yes	$\frac{n}{p} + \alpha \log p$
Permutation, Sort, Union, Merge, Zip, GroupBy[b], Join[b]	no	no	$\left(\frac{n}{pw} + \beta\right) \log \frac{1}{\delta} + \alpha \log p$

[a] no certificate required if input elements are distinct.
[b] invasive checker for input redistribution phase.

the output of the main program: a *checker*. The checker must be very fast to avoid large slowdowns, much faster than the main program. The probabilistic checkers we consider here make a favorable trade-off between confidence and speed.

In distributed computing, communication latency and bandwidth limitations are among the main limiting factors. Our main optimization criterion is therefore the maximum amount of data sent or received at any single processing element (PE), as the slowest PE determines overall running time. In a previous paper [5] we proposed to intensify the search for algorithms with low *bottleneck* communication volume. More precisely, consider an input of n fixed-size elements distributed over the p PEs such that every PE holds $\mathcal{O}(n/p)$ elements. Then we want *no* PE to send or receive more than $o(n/p)$ elements, or be part in more than a polylogarithmic number of data exchanges (messages).

In this paper, we adopt the terminology of Thrill [3] for operations. We also design our checkers to become part of it, and implemented them within Thrill for our experiments.

Note that the majority of our results also apply to distributed database systems. The proximity between data-parallel big data processing frameworks and distributed databases is exemplified by the SQL layer of Apache Spark [6].

Table 1 lists our main results. For each operation, it states whether the entire result needs to be available at all PEs (or a distributed result suffices), whether the checker requires a certificate—and if yes, whether a distributed certificate suffices—and the checker's running time.

2 Preliminaries

Let an operation's input data set consist of n elements, each represented by a fixed number of machine words[1], and let k denote the number of elements in the output of a particular operation, and let w be the machine word size in bits. Consider p processing elements (PEs) connected by a network, numbered $1..p$, where $a..b$ is shorthand for $\{a, \ldots, b\}$ throughout this paper. We assume that PEs can send and receive at most one message simultaneously (full-duplex, single-ported communication). Then, sending a message of size m bits takes time $\alpha + \beta m$, where α is the time to initiate a connection and β the time to send a single bit over an already-established connection. We treat α and β as variables in asymptotic analysis. Thus, a running time of $\mathcal{O}(x + \beta y + \alpha z)$ allows us to discuss *internal work* x, *communication volume* y, and the *number of messages* (or *latency*) z separately. Frequently, all three aspects are important, and combining them into a single expression for running time allows for a concise representation.

Throughout this paper, logarithms with an unspecified base are binary logarithms: $\log x := \log_2 x$.

Error Model Checkers have one-sided error: they are never allowed to reject the result of a correct computation. Only in the case of an incorrect result are they allowed to erroneously accept with a small probability, limited by the parameter $\delta > 0$.

Distribution of Input We usually assume that the n input elements are distributed over the p PEs such that each PE holds $\mathcal{O}(n/p)$ elements. This is mainly in order to simplify notation. A generalization would usually involve an additional term proportional to the maximum number of elements located on any PE. Note that we explicitly *do not* assume random distribution of the input, as that would require redistribution of the entire input in cases where this is not the case already.

Certificates Some operations become much easier to check if the result of the operation is accompanied by a *certificate* to facilitate checking. However, when an algorithm provides an output along with a certificate for said output, the certificate might also be faulty. We need to take care not to accept incorrect results because the certificate contains the same flaw, *i.e.* we need to verify the correctness of the certificate as well.

Collective Communication A *broadcast* distributes a message to all PEs. *Reduction* applies an associative operation to a sequence of k bits. In an *all-reduction*, the result is also broadcast. All of these operations can be performed in time $T_{coll}(k) := \mathcal{O}(\beta k + \alpha \log p)$ [7, 8]. In an *all-to-all* communication, every PE sends k bits to every other PE. This can be performed in $T_{all\text{-}to\text{-}all}(k) := \mathcal{O}(\beta k + \alpha p)$ using direct delivery or $\mathcal{O}(\beta k \log p + \alpha \log p)$ using hypercube indirect delivery [9]

Hashing To simplify analysis, we assume the availability of *random hash functions*, *i.e.* hash functions chosen uniformly at random from the set of all mapping between the input and output value types. Then, hash values can be treated like numbers chosen uniformly at random from the hash functions' image space. We explain separately when weaker guarantees on the randomness of the hash function suffice.

[1] Oftentimes, adaptation to variable sized objects is possible. Our focus on fixed size objects is mainly to simplify notation.

Reduction A *reduce* operation collects elements by their keys, processing all elements of the same key in an arbitrary order using an associative reduce function $f : \mathit{Value} \times \mathit{Value} \to \mathit{Value}$. This function describes how two combine two elements into one. To reduce the local elements of a PE, we use a hash table h. Process elements one-by-one, denoting the current element by $e = (k, v)$, and set $h[k] = f(v, h[k])$ or $h[k] = f(v, 0)$ if $h[k]$ does not exist. We then use a simple reduction algorithm (see "Collective Communication" above) to obtain the final result at PE 0. The total time taken is $\mathcal{O}(n/p + T_{coll}(wk)) = \mathcal{O}(n/p + \beta wk + \alpha \log p)$.

GroupBy A more general approach to aggregation is *GroupBy*, where all elements with a certain key are collected at one PE and processed by the group function $g : [\mathit{Value}] \to \mathit{Value}$. This enables the use of more powerful operators such as computing median, but requires more communication. Total running time is $\mathcal{O}(n/p + T_{all\text{-}to\text{-}all}(wn/p)) = \mathcal{O}(n/p + \beta wn + \alpha p)$.

Result Integrity When the output of an operation or a certificate is provided at all PEs rather than in distributed form, we need to ensure that all PEs received the *same* output or certificate. This can be achieved by hashing the data in question with a random hash function, and comparing the hash values of all other PEs. This can be achieved in time $\mathcal{O}(k + \alpha \log p)$ by broadcasting the hash of PE 0, which every PE can compare to its own hash, and aborting if any PE reports a difference.

3 Related Work

We were surprised to find that distributed, let alone communication efficient, probabilistic checkers appear to be a largely unstudied problem. We therefore give a short overview of techniques for increasing confidence in the output of a program.

From a theoretical standpoint, formally verifying the correctness of a program is the ultimate goal. While tremendous progress has been made in this area (e.g. [10]), verifying implementations of complex distributed algorithms in popular programming languages is still infeasible.

In practice, programs are often tested against a set of test cases of known good input/output pairs, ensuring that the program computes the correct result without crashing for all test cases. However, testing can never demonstrate the *absence* of bugs.

Blum and Kannan [11] introduce probabilistic checkers in a sequential setting. McConnell et al. [12] build upon this to design (sequential) *certifying algorithms*, stressing the importance of simplicity of the checker, and putting increased focus on certificates (also termed *witnesses*) that prove that the output is correct. While formally verifying the algorithm may be infeasible, doing so for the verifier may be within reach. As a result, one can use fast but complicated algorithms that may be beyond the programmer's competence to fully understand, while maintaining certainty in the correctness of the result.

Verification of arbitrary computations performed by a single machine or outsourced to a set of untrusted machines is a well-studied problem, dating back over 25 years and published under names such as "verifiable computing" [13], "checking computations" [14], or "delegating computations" [15]. All of these are computationally expensive beyond the limits of feasibility, despite recent efforts to make verifiable computing more practical [16]. A recent survey by Walfish and Blumberg [17] concludes that *"[t]he sobering news, of course, is these systems are basically toys"* due to orders-of-magnitude overhead. In contrast, our work focuses on checking specific operations, allowing us to develop checkers that are fast in practice.

An approach that has received significant attention for linear algebra kernels is *Algorithm-Based Fault Tolerance* (ABFT) [18, 19]. By encoding the input using checksums, and modifying the algorithms to work on encoded data, ABFT techniques detect and correct any failure on a single processor. However, this requires the algorithms to be redesigned to operate on the encoded data.

Note that while some existing systems, *e.g.* Apache Spark [1], implement some fault tolerance measures such as detecting and handling the failure of individual nodes, there are large classes of failures that no existing big data processing system appears to detect or mitigate. These include data corruption caused by *soft errors*, *i.e.* incorrectly handled edge cases in the programming, or bitflips in CPU or main memory caused by cosmic rays, leak voltage, as well as *hard errors*, *i.e.* device defects causing repeated failure at the same memory locations. A common solution to memory errors is the use of ECC RAM [20], which can correct most soft errors and detect (but often not correct) many hard errors [21]. This is orthogonal to our checkers: a checker can never detect errors introduced into the input data before the algorithm is invoked, so the use of ECC RAM remains advisable even when computations can be checked.

4 Count and Sum Aggregation

Reductions are perhaps the most important operation in the kind of Big Data systems we consider, and the paradigm they extend even carried them in their name: MapReduce. The checker we describe works not only for sum aggregation, but also other operations on integers that fulfill certain properties. We require that the reduce operator $\oplus$ be associative, commutative, and satisfy $x \oplus y \neq x$ for all $y \neq 0$, *i.e.* every element except the neutral element changes the result. Examples include count aggregation, which conceptually equals sum aggregation where the value of every element is mapped to 1, and exclusive or (*xor*). Without loss of generality, we therefore only discuss sum aggregation in this section. Several other common choices of reduce functions violating the above requirements and aggregations requiring the additional power of GroupBy are discussed in Section 6.

In sum aggregation, we are given a distributed set of (*key*, *value*)-pairs. Let K be the (unknown) set of keys in the input and $k := |K|$ its equally unknown size. The output of sum aggregation then consists of a single value for each key, which is the sum of all values associated with it in the input. In SQL, this operation would be expressed as

```
SELECT key, SUM(value) FROM table GROUP BY key.
```

Theorem 1. *Let $\oplus$ be an associative and commutative reduce function on integers satisfying $x \oplus y \neq x$ for all elements x, y with $y \neq 0$. Then $\oplus$-aggregation of n elements with failure probability at most δ can be checked in time $T_{\text{check-sum}}(n, p, \delta) := \mathcal{O}\left(\left(\frac{n}{p} + \beta dw\right)\log_d \frac{1}{\delta} + \alpha \log p\right)$, where d is a tuning parameter.*

Note that only the local work depends on the input size, and that the running time is independent of the number of keys.

To check the result of such an aggregation, we apply a naïve sum reduction algorithm to a condensed version of the input. We shall sometimes refer to this as the *minireduction* in subsequent sections. Let d from the statement of Theorem 1 be the size of the condensed key space, with $2 \leq d \ll k$. We then use a random hash function $h : K \to 1..d$ to map the original keys to the reduced key space. Let $\hat{r} \geq d$ be a modulus parameter, with r chosen uniformly at random from the half-open interval $(\hat{r}, 2\hat{r}]$. Apply a naïve sum reduction modulo r to the thus-remapped input

def checkSumAgg(v : Element[n_i], o : Element[k_i], $\delta : \mathbb{R}$) : Boolean
  $(d, \hat{r}) : \mathbb{N} \times \mathbb{N}$:= numerically determined parameters (see Table 2)
  $r : \mathbb{N}$:= random number from $\hat{r}$+1..2$\hat{r}$ −− modulus parameter
  $h : K \to 1..d$:= random hash function −− maps keys to buckets
  w_v := cRed(v, d, r, h) −− apply condensed reduction to input
  w_o := cRed(o, d, r, h) −− ...and asserted result of sum aggregation
  return $w_v = w_o$ −− significant only at PE 0

def cRed(arr : Element[], $d : \mathbb{N}$, $r : \mathbb{N}$; $h : K \to 1..d$) : Value[d]
  t : Value[d] = $\langle 0, \ldots, 0 \rangle$
  foreach $(k, v) \in arr$
    $t[h[k]] := (v + t[h[k]]) \bmod r$ −− local reduction
  Reduce(t, $+_r$, 0) −− reduce to PE 0 with addition modulo r
  return t −− significant only at PE 0

Algorithm 1: A single iteration of the sum aggregation checker. Here, v is the input to the sum aggregation operation, a sequence of n elements of which PE i holds n_i; o is the asserted result with k_i out of k elements at PE i; and δ is the maximum allowed failure rate.

and—separately—the output of the aggregation algorithm. If both produce the same result, the operation was likely conducted correctly. Pseudocode is given in Algorithm 1.

Lemma 2. *A single iteration of the above sum aggregation checker fails with probability at most* $\frac{1}{\hat{r}} + \frac{1}{d}$.

Proof. If the aggregation was performed correctly, then the per-bucket results in the reduced keyspace are the same for input and output, and thus the checker always accepts a correct result. Thus assume from now on that the output of the aggregation operation is *incorrect*, *i.e.* the checker *should* fail.

For $i \in K$, let n_i be the (correct, unknown) $\oplus$-aggregate of all values with key i, and n_i' be the asserted such value. Then, as the result is incorrect, there exists at least one i with $n_i \neq n_i'$. Let $I := \{i \in K \mid n_i \neq n_i'\}$ be the keys whose results were computed incorrectly by the operation.

Let $h : K \to 1..d$ be the random hash function used to map keys to the condensed keyspace. We use this hash function to map elements to the d buckets by their keys, and reduce the values in associated counters modulo r, where r is chosen uniformly at random from $(\hat{r}, 2\hat{r}]$. Effectively, we operate in the residue class ring $\mathbb{Z}/r\mathbb{Z}$. Thus the checker fails if

$$\forall_{j \in 1..d} : \bigoplus_{\substack{i \in I \\ h(i)=j}} n_i = \bigoplus_{\substack{i \in I \\ h(i)=j}} n_i' \mod r.$$

We prove the claimed failure probability by first considering the failure modes introduced by the modulus, and then analyzing a version of the checker without a modulus. In this second part, we injectively map each hash function h for which the checker fails to $d-1$ distinct hash functions for which it does not fail.

(1) If there is an $i \in I$ with $n_i = n_i' \mod r$, the checker cannot detect the error in this key. For a single key, this occurs with probability $r^{-1} \leq \hat{r}^{-1}$ by definition of r. However, for the checker to fail, $n_i = n_i' \mod r$ must hold for all $i \in I$. Thus, the probability of failure declines exponentially

with $|I|$, and $|I| = 1$ is the hardest case. Therefore, $\hat{r}^{-1}$ bounds the total additional failure probability introduced by the modulus.

(2) Define $F := \{h : K \to 1..d \mid \text{checker fails for } h\}$ and let $\breve{\imath} := \min I$ be the first key with mismatched aggregate value. For each $h \in F$, we define the $d-1$ hash functions $\overline{h_j}$ with

$$\forall_{j \in 1..d \setminus \{h(\breve{\imath})\},\ i \in K} : \overline{h_j}(i) = \begin{cases} h(i) & i \neq \breve{\imath}, \\ j & else \end{cases}$$

and let $\overline{F} := \left\{\overline{h_j} \mid h \in F, j \in 1..d \setminus \{h(\breve{\imath})\}\right\}$. Clearly, the checker does not fail for any $\overline{h_j} \in \overline{F}$, as exactly one element with different values is being remapped in a hash function for which it does fail (the case of $n_{\breve{\imath}} = n'_{\breve{\imath}} \mod r$ is treated in **(1)**). By the assumption $x \oplus y \neq x$ for $y \neq 0$, the result will differ in exactly two buckets, and the checker will notice.

We also need to show that the mapping is injective, *i.e.* that the new hash functions are unique and thus $|\overline{F}| = (d-1)\,|F|$. Clearly, for given h all of its $\overline{h_j}$ are different, so assume that there exists an $h' \neq h$ and j, j' so that $\overline{h_j} = \overline{h'_{j'}} \in \overline{F}$. Then, by definition, $j = \overline{h_j}(\breve{\imath}) = \overline{h'_{j'}}(\breve{\imath}) = j'$ and thus $j = j'$. For $\overline{h_j} = \overline{h'_j}$, we furthermore need

$$\forall_{i \in K} : \overline{h_j}(i) = \overline{h'_j}(i) \overset{def.}{\iff} \forall_{i \in K \setminus \{\breve{\imath}\}} : h(i) = h'(i)$$

Thus $h = h'$ if and only if $h(\breve{\imath}) = h'(\breve{\imath})$. But this must hold, for otherwise we would have $h' = \overline{h_{h'(\breve{\imath})}} \in \overline{F}$ by definition of the $\overline{h_j}$, which contradicts the assumption that $h' \in F$. Therefore, such an h' cannot exist and $|\overline{F}| = (d-1) \cdot |F|$. □

The failure probability bound in **(2)** is tight: if the only difference between the two inputs is the key of a single item, then the probability that the hash values of the new and old key are the same (and the modification thus goes unnoticed) is $1/d$. This follows from the uniformity of random hash functions.

Lemma 3. *Checking $\oplus$-reduction with $\oplus$ as in Theorem 1, n input pairs, and k keys, using d buckets, with moduli in $(\hat{r}, 2\hat{r}]$ and probability of failure at most $\delta > 0$, is possible in time*

$$\mathcal{O}\left(\left(\frac{n}{p} + \beta d \log(2\hat{r})\right) \log_{\left(\frac{1}{\hat{r}} + \frac{1}{d}\right)^{-1}} \delta^{-1} + \alpha \log p\right).$$

Proof. From the above description we can see that a single iteration of the checker requires time $\mathcal{O}(n/p)$ to hash and locally reduce the input, and $T_{coll}(d \log(2\hat{r})) = \mathcal{O}(\beta d \log(2\hat{r}) + \alpha \log p)$ for the reduction. By repeating the procedure $\left\lceil \log_{\left(\frac{1}{\hat{r}} + \frac{1}{d}\right)^{-1}} \delta^{-1} \right\rceil$ times, we can increase the probability of detecting an incorrect result to at least $1 - \delta$ by accepting only if all repetitions of the procedure declare the result to be likely correct. To keep the number of messages to a minimum, we can execute all instances of the checker simultaneously (this also means that we only have to read the input once), and perform their reductions at the same time. □

We can instantiate this checker in different ways to obtain the characteristics we wish. First, we use this to show Theorem 1:

Proof (Theorem 1). The theorem follows from Lemma 3 with $d \leq \hat{r} \leq 2^{w-1}$ and $\left(\frac{1}{\hat{r}} + \frac{1}{d}\right)^{-1} < d$. □

Table 2: Some numerically determined optimal values for bucket count d and modulus parameter $\hat{r}$ given a message size of b bits.

b	δ	d	$\hat{r}$	#its	achieved δ
1024	10^{-4}	37	2^8	3	$3.0 \cdot 10^{-5}$
1024	10^{-6}	25	2^7	5	$2.5 \cdot 10^{-7}$
1024	10^{-8}	18	2^7	7	$4.1 \cdot 10^{-9}$
1024	10^{-10}	14	2^6	10	$2.5 \cdot 10^{-11}$
1024	10^{-20}	6	2^4	32	$3.3 \cdot 10^{-21}$
4096	10^{-6}	124	2^{10}	3	$7.4 \cdot 10^{-7}$
4096	10^{-10}	68	2^9	6	$2.1 \cdot 10^{-11}$
4096	10^{-20}	32	2^8	14	$4.4 \cdot 10^{-21}$
16 384	10^{-7}	420	2^{12}	3	$1.8 \cdot 10^{-8}$
16 384	10^{-10}	273	2^{11}	5	$1.2 \cdot 10^{-12}$
16 384	10^{-20}	148	2^{10}	10	$7.6 \cdot 10^{-22}$
16 384	10^{-30}	93	2^{10}	16	$1.3 \cdot 10^{-31}$
65 536	10^{-10}	1170	2^{13}	4	$9.1 \cdot 10^{-13}$
65 536	10^{-20}	630	2^{12}	8	$1.3 \cdot 10^{-22}$
65 536	10^{-30}	420	2^{12}	12	$1.1 \cdot 10^{-31}$
65 536	10^{-40}	321	2^{11}	17	$2.9 \cdot 10^{-42}$

We can also minimize bottleneck communication volume and find that minimum at $d = 2$ buckets, $\hat{r} = 8$ for a modulus range of 9..16, and thus a minireduction result size of 8 bits with $\log_{1.6} \delta^{-1}$ repetitions. However, this high number of repetitions causes a lot of local work. Further, the practical usefulness of sending single bytes across the network is questionable. In effect, real-world interconnects have an effective minimum message size b, such that sending fewer than b bits is not measurably faster than sending a message of size b bits. Thus, our goal should be to minimize the number of iterations—$\left\lceil \log_{\frac{1}{\hat{r}}+\frac{1}{d}} \delta \right\rceil$—under the constraint that the result size be close to b bits: $d\lceil \log(2\hat{r}) \rceil \lceil \log_{\frac{1}{\hat{r}}+\frac{1}{d}} \delta \rceil \leq b$. This relation can be used to (numerically) compute optimal choices of $\hat{r}$ and d for given b. Table 2 shows such values for some interesting choices of b and δ. However, in practice, keeping local work low might be more important than these solutions to minimize δ admit, and one might prefer to trade a reduced number of iterations for a larger value of d and perhaps choose $\hat{r} = 2^{31}$.

Optimizations Multiple instances of this algorithm can be executed concurrently by using a hash function that computes $c \cdot \lceil \log d \rceil$ bits. Its value can then be interpreted as c concatenated hash values for separate instances, enabling bit-parallel implementation. It is also possible to use Single Instruction Multiple Data (SIMD) techniques to further reduce local work. Refer to Section 7 for specific implementation details.

5 Permutation and Sorting

Many approaches for permutation checking exist in the sequential case, and often directly imply communication efficient equivalents. Perhaps most elegantly and first described by Wegman and Carter [22]—albeit in a less general manner than we prove here—we can use a random hash function and compare the sum of hash values in the input and output sequences. Once we have established that a sequence is a permutation of another, verifying sortedness of the output sequence only requires that each PE receive the smallest element of its successor PE and compare its local maximum to it.

Lemma 4. *Let $E = \langle e_1, \dots, e_n \rangle$ and $O = \langle o_1, \dots, o_n \rangle$ be two sequences of n elements from a universe U. Then, for a random hash function $h : U \to 0..H-1$, define $\lambda := \sum_{i=1}^{n} h(e_i) - h(o_i)$ in $\mathbb{Z}/H\mathbb{Z}$ (*i.e.*, mod H), and*

$$\mathbf{P}[\lambda = 0] = \begin{cases} 1 & \text{if } E \text{ is a permutation of } O \\ H^{-1} & \text{otherwise.} \end{cases}$$

Proof. First define $h(X) := \sum_{x \in X} h(x)$ for any set or sequence X. If E and O are permutations of each other, then $h(E) = h(O)$ due to commutativity of addition, and thus $\lambda = 0$ by associativity. Otherwise, when interpreted as sets, E and O each consist of a set of shared and private elements: $E = S \cup P_E$, $O = S \cup P_O$, with $P_E \cap P_O = \varnothing$. By commutativity, the shared elements contribute the same value $h(S)$ to both sums of hash values: $h(E) = h(S) + h(P_E)$ and $h(O) = h(S) + h(P_O)$. Since P_E and P_O are disjoint sets and h is a random hash function, $h(P_E)$ and $h(P_O)$ are independent and uniformly distributed over $0..H-1$, and thus equal with probability $1/H$. The same thus also holds for $h(E)$ and $h(O)$. □

[**this breaks for multiple occurrences of the same element. We can either require uniqueness or drop the modulo (in practice, use 64-bit numbers to add 32-bit hash values, and use higher precision only to add the results of every block of 2^{32} elements) and adjust the proof to Peter's idea. Let e be an element that w.l.o.g. occurs more frequently in E than O, say k times in E and $k' < k$ in O (can be 0). Then $h(E) = h(E \setminus e) + k \cdot h(e)$ and $h(0) = h(O \setminus e) + k' \cdot h(e)$, which is equivalent to $h(e) = (h(O \setminus e) - h(E \setminus e))/(k - k') =: x$, which occurs with probability at most $1/H$ because x is independent of $h(e)$.**] TODO

Every PE can compute the sum for its $\mathcal{O}(n/p)$ local elements independently (see Section 2). A sum all-reduction modulo H over these values then yields s. The algorithm can thus be executed in time $\mathcal{O}(n/p + \beta \log H + \alpha \log p)$.

However, this algorithm requires confidence in the randomness of the hash function h. If we do not have access to a sufficiently trusted hash function, we can use a different approach based on constructing a polynomial from E and O, commonly attributed to R.J. Lipton, and essentially a solution to Exercise 5.5 in Mehlhorn and Sanders [23].

Lemma 5. *For two sequences $E = \langle e_1, \dots, e_n \rangle$ and $O = \langle o_1, \dots, o_n \rangle$ from a universe $0..U-1$ and any $\delta > 0$, choose a prime $r > \max(n/\delta, U-1)$ and define the polynomial*

$$q(z) := \prod_{i=1}^{n} (z - e_i) - \prod_{i=1}^{n} (z - o_i) \mod r.$$

Then for random $z \in 0..r-1$,

$$\mathbf{P}[q(z) = 0] \begin{cases} = 1 & \text{if } E \text{ is a permutation of } O \\ < \delta & \text{otherwise.} \end{cases}$$

Proof. If E is a permutation of O, then both products have the same factors and thus $q(z) = 0$ for all z by commutativity of multiplication. Otherwise, because r is larger than any e_i or o_i (this is important to ensure that no pair i, j with $e_i \equiv o_j \mod r$ exists) is prime, q is a non-zero polynomial of degree n in the field $\mathbb{F}_r$. Such a polynomial has at most n roots.[2] Thus the probability of $q(z) = 0$ is at most $\frac{n}{r} < \frac{n}{n/\delta} = \delta$. □

Instead of doing $2n$ expensive multiplication-modulo-prime operations, one could also consider using carry-less multiplication in a Galois Field $\mathrm{GF}(2^\ell)$ with an irreducible polynomial. Multiplication in Galois Fields can be implemented very efficiently, *e.g.* using Intel SIMD instructions [24].

Theorem 6. *It is possible to check whether a sequence of n elements is a permutation of another such sequence with probability at least $1 - \delta$ in time $\mathcal{O}\big((n/(pw) + \beta)\log\frac{1}{\delta} + \alpha \log p\big) =: T_{\textit{check-perm}}(n, p, \delta)$.*

Proof. We can boost the success probabilities of the algorithms from Lemmata 4 and 5 arbitrarily by executing several independent instances and accepting only if all instances do. Batching communication keeps latency to $\log p$. Choose $H = 2^w$ in Lemma 4 to obtain the claimed bound. In Lemma 5, choose $\delta = 2^{-w+1}n$. Then, r can always be chosen in $[2^{w-1}, 2^w]$ by Bertrand's postulate, and a machine word can hold the values. □

Theorem 7. *Checking whether a sequence of n elements is a sorted version of another such sequence with probability $\geq 1 - \delta$ is possible in time $T_{\textit{check-sort}}(n, p, \delta) := \mathcal{O}\big(T_{\textit{check-perm}}(n, p, \delta)\big)$.*

Proof. After verifying the permutation property using Theorem 6, it remains to be checked whether the elements are sorted. First, verify that the local data is sorted in time $\mathcal{O}(n/p)$. Then, transmit the locally smallest element to the preceding PE, and receive the smallest element of the next PE. Compare this to the locally largest element. Lastly, verify that no PE rejected using a gather operation. In total, this requires time $\mathcal{O}\big(T_{\textit{check-perm}}(n, p, \delta)\big) + \mathcal{O}(n/p) + 2\beta w + 2\alpha + T_{\textit{coll}} = \mathcal{O}\big(T_{\textit{check-perm}}(n, p, \delta)\big)$. □

6 Further Checkers

There are many operations for which we can construct checkers from the sum aggregation and permutation checker. We discuss several in the following subsections. Afterwards, we turn to some *invasive* checkers, *i.e.* checkers that do not treat the operation as a black box, and instead check only part of the operation. While less desirable than true checkers, these checkers allows us to broaden the range of covered operations.

6.1 Average Aggregation

Computing the per-key averages is impossible to express in a scalar reduction, but becomes easy when replacing (*key*, *value*)-pairs with (*key*, *value*, *count*)-triples and using the reduce function $\oplus$ with $(k_1, v_1, c_1) \oplus (k_1, v_2, c_2) := (k_1, v_1 + v_2, c_1 + c_2)$. The computation is then followed up by an output function h with $h(k_1, v_1, c_1) := (k_1, v_1/c_1)$. This avoids the use of the much more communication-expensive GroupBy function.

[2] This can easily be seen by induction. It is easy to verify for $n \leq 1$. Now let q be a polynomial of degree $n \geq 2$, and a be a root of q (if none exists, we are finished). Then $q = q' \cdot (X - a)$ for some polynomial q' of degree $n - 1$, and by the induction hypothesis, q' has at most $n - 1$ roots. For some $b \in \mathbb{F}_r$, $q(b) = q'(b) \cdot (b - a)$ is zero iff $a = b$ or $q'(b) = 0$. Thus q has at most n roots.

Checking average aggregation is easy when these per-key element counts are available in a certificate, for then we can reconstruct the result of a sum aggregation by undoing the final division—termed h above—by multiplying the average value with the count for every key. As described above, this certificate naturally arises during computation anyway, and thus does not impose overhead on the average computation. Now we can leverage the sum checker, applying it both to the input sequence and the reconstructed sums. As the product is computed component-wise, both the asserted averages and the certificate can be supplied in distributed form, as long as both values are available at the same PE for any key.

To prevent accidental mismatches when both averages and counts are scaled in a way that yields the same reconstructed sums—*e.g.* double the averages and halve the counts—, we also need to check the correctness of the counts. Therefore, we also need to apply the *count aggregation checker* to the input and the certificate. The same can also be achieved in a single step by applying the (*key*, *value*, *count*)-triple aggregation trick using the reduce function $\oplus$ described above to the checker.

Corollary 8. *For a given input sequence of n key-value pairs $\langle e_1, \ldots, e_n \rangle$ with $e_i = (k_i, v_i)$, keys $k_i \in K$, and $k := |K|$, it is possible to check whether a set of k asserted per-key averages $\langle a_1, \ldots, a_k \rangle$ is correct if the number of values associated with each key is available as a certificate $\langle c_1, \ldots, c_k \rangle$, by supplying $\langle (e_1, 1), \ldots, (e_n, 1) \rangle$ as input and $\langle (a_1 \cdot c_1, c_1), \ldots, (a_k \cdot c_k, c_k) \rangle$ as output to the sum aggregation checker of Section 4, achieving the time complexity and success probability of Theorem 1.*

6.2 Minimum and Maximum Aggregation

Now consider computing the minimum or maximum value per key (*w.l.o.g.*, we shall only consider minima henceforth). This is a surprisingly difficult operation to check. Clearly, the sum aggregation checker of Section 4 is not directly applicable, as the min function does not satisfy the requirement $\min(a, b) \neq a$ if $b \geq a$. To check min-aggregation, we need to verify for each key that (a) no elements smaller than the purported minimum exist, as well as determine whether (b) the minimum value does indeed appear in the input sequence. Both subproblems seem to require the asserted result to be known at all PEs. Let $S = \langle x_1, \ldots, x_n \rangle$ be the input sequence, and $M = \langle m_1, \ldots, m_k \rangle$ the asserted output. Given M, property (a) is easy to verify by iterating the locally present part of S and verifying that no element exists with a value smaller than the entry of its key in M.

However, property (b) is surprisingly hard to check using $o(k)$ bits of communication (it is easy to verify in time $\mathcal{O}(n/p + \beta k + \alpha \log p)$ using a bitwise or reduction—see Section 2—on a bitvector of size k specifying which keys' minima are present locally, and testing whether each bit is set in the result). A certificate in the form of the locations of the minima, available in full at every PE, remedies this. Each PE then needs to verify its set of local asserted minima. The certificate is required to be available in full at every PE so that we can ensure that all keys are covered—otherwise, a faulty algorithm might simply "forget" a key, and the checker would not be able to notice.

Theorem 9. *Checking minimum (maximum) aggregation is possible in time $\mathcal{O}(n/p + \alpha \log p)$ provided that the asserted output and a certificate specifying which PE holds the minimum (maximum) element for any key are available at all PEs.*

Proof. Follows directly from the above discussion. □

Note that this checker is not probabilistic and thus guaranteed to notice any errors in the result. We discuss possibilities for improvement and possible lower bounds in the section on future work at the end of the paper.

def checkMedian(v : Element$[n_i]$, o : Map⟨Key→Value⟩, δ : $\mathbb{R}$) : Boolean
  s : Map⟨Key → $\mathbb{Z}$⟩ using default value 0 for unknown keys, size k
  foreach $(key, value) \in v$ −− combine mapping and local reduction
    if $val < o[key]$ **then** $s[key] = s[key] - 1$
    if $val > o[key]$ **then** $s[key] = s[key] + 1$
  return checkSumAgg(s.to_array, $\langle 0, \ldots, 0 \rangle$, δ)

Algorithm 2: Pseudocode for the median checker (unique values). The function checkSumAgg is defined in Algorithm 1. Here, v is the input to the median aggregation operation, a sequence of n elements of which PE i holds n_i; o is the asserted result of size k; and δ is the maximum allowed failure rate. For simplicity, we use associative arrays indexed by key for o and s.

6.3 Median Aggregation

We use the common definition of the median of an even number of elements as the mean of the two middle elements. Thus there may not exist an element that is equal to the median, but—assuming unique values—the number of elements *smaller* than the median is always equal to the number of *larger* elements.

If the asserted medians are available at every PE, checking a median aggregation operation on unique values can be reduced to the sum aggregation problem by verifying the above property. Simply map elements smaller than their key's median to −1, and larger elements to +1. Then, the sum over all of these values must be 0 for every key. This property can be checked probabilistically using the sum aggregation checker of Theorem 1. Pseudocode of our algorithm is given in Algorithm 2.

Requiring that each value occur no more than once for each key is without loss of generality because it can be enforced by an appropriate tie breaking scheme. However, the median aggregation algorithm does not necessarily need to apply the tie breaking scheme during its execution: we only need to know which occurrence of the median value is the one with rank $n/2$, as determined by the chosen tie breaking scheme. Depending on the median aggregation algorithm used[3], this can be much simpler than using the tie breaking scheme in the entire algorithm.

Theorem 10. *Median aggregation on a sequence of n elements can be checked with failure probability at most $\delta > 0$ in time $\mathcal{O}(T_{\text{check-sum}}(n, p, \delta))$ provided the asserted result is available at every PE. For non-unique values, tie breaking information on the median values is required as a certificate.*

Proof. By definition, an element is the median of a set of unique elements if and only if the number of elements smaller than it is exactly equal to the number of elements that is larger. This is verified using the sum aggregation checker by mapping smaller values to −1 and larger values to +1. Ties are broken using an appropriate tie breaking scheme and the certificate. The claim then follows from Theorem 1. □

6.4 Zip

Zipping combines two sequences $S_1 = \langle x_1, \ldots, x_n \rangle$ and $S_2 = \langle y_1, \ldots, y_n \rangle$ of equal length n index-wise, producing as its result a sequence of pairs $S = \langle (x_1, y_1), \ldots, (x_n, y_n) \rangle$. However, since S_1 and S_2

[3]We describe a communication efficient algorithm in [25], which could be executed in parallel for every key

need not have the same data distribution over the PEs, this is nontrivial. Thus, the elements of (at least) one sequence need to be moved in the general case. Checking Zip therefore requires verifying that the order of the elements is unchanged in both sequences. For this, we require a hash function that can be evaluated in parallel on distributed data, independent of how the input is split over the PEs. One example would be the inner product of the input and a sequence of n random values, $R = \langle r_1, \ldots, r_n \rangle$, where $r_i = h'(i)$ for some high-quality hash function h' [9]. This way, R can be computed on the fly and without communication.

Theorem 11. *Checking* $\mathrm{Zip}(\mathrm{S}_1, \mathrm{S}_2)$ *with* $|S_1| = |S_2| = n$ *with false positive probability of at most* $\delta > 0$ *can be achieved in time* $\mathcal{O}\Big(\frac{n}{pw} \log \frac{1}{\delta} + \beta \log \frac{1}{\delta} + \alpha \log p\Big)$.

Proof. A single iteration of the checker works as follows. Apply the hash function to S_1 and the first part of the elements of S. Computing this hash value is possible in time $\mathcal{O}(n/p + \beta \log H + \alpha \log p)$ for the families of hash functions discussed above, where the output of the hash function is in $0..H-1$. The same is applied to S_2 and the second part of the elements of S. Accept if both pairs of hashes match. Again, the success probability can be boosted to δ by executing $\log_{H^{-1}} \frac{1}{\delta}$ instances of the checker in parallel, and the claimed running time follows for $H = 2^w$. □

6.5 Other Operations

We now briefly describe several further operations to which the permutation and sort checker can be adapted. For the latter two, we present *invasive* checkers for the element redistribution phase. The rest of the operation needs to be checked with an appropriate local checker.

6.5.1 Union

Verifying whether a multiset S is the union of two other multisets S_1 and S_2 of size n_1 and n_2, respectively, is equivalent to checking whether S is a permutation of the concatenation of S_1 and S_2. We can therefore adapt the permutation checker of Section 5 to iterate over *two* input sets.

Corollary 12. *The* $\mathrm{Union}(\mathrm{S}_1, \mathrm{S}_2)$ *operation can be checked in time* $\mathcal{O}\big(T_{\textit{check-perm}}(|S_1| + |S_2|, p, \delta)\big)$, *where* $T_{\textit{check-perm}}(n, p, \delta)$ *is the the time to check permutations of size* n *from Theorem 6.*

6.5.2 Merge

A Merge operation combines two sorted sequences S_1 and S_2 of length n_1 and n_2, respectively, into a single sorted sequence S of length $n_1 + n_2$. Checking it is thus equal to verifying that S is sorted and the union of S_1 and S_2, *c.f.* the previous subsection.

Corollary 13. *Checking* $\mathrm{Merge}(\mathrm{S}_1, \mathrm{S}_2)$ *with probability at least* $1 - \delta$ *is possible in time*

$$\mathcal{O}(T_{\textit{check-sort}}(|S_1| + |S_2|, p, \delta)) .$$

6.5.3 GroupBy

The GroupBy operation is conceptually similar to aggregation (see Section 4), but passes all elements with the same key to the *group function* $g : [\mathit{Value}] \to \mathit{Value}$. Thus, for every key, all elements associated with it need to be sent to a single PE. This stage of a GroupBy operation can therefore

be checked using the sort checker, where the order is induced by the hash function assigning keys to PEs. The group function needs to be checked separately by an appropriate *local checker*, which falls outside the scope of this paper.

Corollary 14. *Checking the redistribution phase of* GroupBy *on a sequence of n elements with probability at least $1 - \delta$ is possible in time $\mathcal{O}(T_{\text{check-sort}}(n, p, \delta))$.*

6.5.4 Join

Similarly to GroupBy, we can design an invasive checker for element redistribution in Join operations. The two common approaches to joins are the *sort-merge join* and *hash join* algorithm [26]. Note that the sort checker can be used for both approaches, because as far as data redistribution is concerned, a hash join is essentially a sort-merge join using the hashes of the keys for sorting. To further verify that the distribution of keys to PEs is the same for both input sequences, we exchange the locally largest (smallest) keys with the following (preceding) PE and check that those are larger (smaller) than the local maximum (minimum). The correctness of the element redistribution then follows from their global sortedness.

Corollary 15. *Checking the input redistribution phase of a hash or sort-merge* Join *on two sequences of n_1 and n_2 elements with success probability at least $1-\delta$ is possible in time $\mathcal{O}(T_{\text{check-sort}}(n_1 + n_2, p, \delta))$.*

7 Experiments

We developed implementations of our core checkers, sum aggregation and sorting. These were integrated into Thrill [3], an open source data-parallel big data processing framework[4] using modern C++ and developed at Karlsruhe Institute of Technology. Thrill provides a high-performance environment that allows for easy implementation and testing of our methods, while offering a convenient high-level interface to users.

Goals The aim of our experiments is twofold. First, to show that our checkers achieve the predicted detection accuracy, and second, to demonstrate their practicability by showing that very little overhead is introduced by using a checker.

Manipulators To test the efficacy of our checkers, we implemented *manipulators* that purposefully interfere with the computation and deliberately introduce faults. Manipulators are a flexible way to introduce a wide variety of classes of faults, allowing us to test our hypotheses on which kinds of faults are hardest to detect. It is easy to convince oneself that large-scale manipulation of the result is much easier to detect than subtle changes. Thus, our manipulators focus on the latter kind of change in the data. The specific manipulation techniques used with an operation are described in the respective subsections.

[4] See `http://project-thrill.org` and `https://github.com/thrill/thrill/` for details. Our code is available at `https://github.com/lorenzhs/thrill/tree/checkers`

Table 3: Configurations tested for Sum Aggregation checker.
First set was used for accuracy tests, second set for scaling tests.

Configuration ($\#\text{its} \times d$ m $\log \hat{r}$)	Table size (bits)	Failure rate (δ)	Comment
1×2 m31	64	$5 \cdot 10^{-1}$	High $\hat{r}$ is less effective than
1×4 m31	128	$2.5 \cdot 10^{-1}$	↳ multiple iterations
4×2 m4	40	$1 \cdot 10^{-1}$	Lower δ *and* size than above
4×4 m3	64	$2 \cdot 10^{-2}$	$\delta = 2\,\%$ for 64-bit table
4×4 m5	96	$6 \cdot 10^{-3}$	
4×8 m3	128	$3.9 \cdot 10^{-3}$	
4×8 m5	192	$6 \cdot 10^{-4}$	
4×8 m7	256	$3.1 \cdot 10^{-4}$	
5×16 m5	480	$7.2 \cdot 10^{-6}$	
6×32 m9	1 920	$1.3 \cdot 10^{-9}$	
8×16 m15	2 048	$2.3 \cdot 10^{-10}$	
4×256 m15	16 384	$2.4 \cdot 10^{-10}$	
5×128 m11	7 680	$3.9 \cdot 10^{-11}$	
8×256 m15	32 769	$5.8 \cdot 10^{-20}$	lower local work, larger size
16×16 m15	4 096	$5.4 \cdot 10^{-20}$	higher local work, smaller size

Table 4: Manipulators for Sum Aggregation Checker

Name	Manipulation applied
Bitflip	Flips a random bit in the input
RandKey	randomize the key of a random element
SwitchValues	switches the values of two random elements
IncKey	increments the key of a random element
$IncDec_n$	acts on $2n$ elements with distinct keys,
↳ using $n = 1$	incrementing the keys n elements and
↳ and $n = 2$	decrementing that of n other elements

Implementation Details As hash functions, we used CRC-32C, which is implemented in hardware in newer x86 processors with support for SSE 4.2 [27], and tabulation hashing [22, 28] with 256 entries per table and four or eight tables for 32 and 64-bit values, respectively. We abbreviate the hash functions with "CRC", "Tab", and "Tab64". Where needed, pseudo-random numbers are obtained from an MT19937 Mersenne Twister [29].

Platform We conducted our scaling experiments on up to 128 nodes of bwUniCluster, each of which features two 14-core Intel Xeon E5–2660 v4 processors and 128 GiB of DDR4 main memory. For our overhead and accuracy experiments, we used an AMD Ryzen 7 1800X octacore machine with 64 GiB of DDR4 RAM. The code was compiled with g++ 7.1.0 in release mode.

7.1 Sum Aggregation

We implemented the sum aggregation checker of Section 4. As workload, we chose integers distributed according to a power law distribution, with frequency $f(k; N) = 1/(kH_N)$ for the element of rank k.

Table 5: Overhead of sum aggregation checker: checker local input processing time for 10^6 pairs of 64-bit integers, 10 000 runs

Configuration (see Table 3)	Time per element [ns]	Approx. #cycles
5×16 CRC m5	4.5	16
6×32 CRC m9	4.6	17
8×16 CRC m15	5.1	18
4×256 CRC m15	3.8	14
5×128 Tab64 m11	4.7	17
8×256 Tab64 m15	7.3	26
16×16 Tab64 m15	10.0	36

Here, N is the number of possible elements and H_N is the N-th harmonic number. This distribution naturally models many workloads, *e.g.* wordcount over natural languages.

The tested configurations of the checker—number of iterations, number of buckets, hash function, and modulus parameter—are listed in Table 3. The first set of configurations was used for testing detection accuracy, the second for scaling experiments.

Implementation Details To minimize local work, our implementation employs *bit-parallelism* where possible, *e.g.*, instead of computing eight four-bit hash values, we compute one 32-bit hash value and partition it into eight groups of four bits, which we treat as the output of the hash functions. This is implemented in a generic manner to satisfy any partition of a hash value into groups. Since 64 hash bits suffice to guarantee a failure probability of nearly 10^{-20} (see Table 3), evaluating a single hash function suffices in all practically relevant configurations.

To minimize the cost of adding modulo r, we use 64-bit values for the buckets internally, add normally, and perform the expensive modulo step only if the addition would overflow. This can be detected cheaply with a jump-on-overflow instruction.

Detection Accuracy The manipulators we used are listed in Table 4. Fig. 3 demonstrates that our checkers achieve the theoretically predicted performance in practice, even when using hash functions with weaker guarantees. We can see that Lemma 2 generally overestimates the failure probability introduced by the modulus. Note that the significance of the results for configurations with low δ is limited, as the expected number of failures in 100 000 runs for *e.g.* 4×8 Tab m7 is only 31.1.

We note that CRC-32C appears to work very well for subtle manipulations, which is likely because it was designed so that small changes in the input cause many output bits to change. This makes it likely that a change in an element's key causes it to be sent to a different bucket in at least one iteration. However, the least significant bits appear to change in similar ways for different inputs, as indicated by the $IncDec_1$ measurements, causing an elevated failure rate in this setting. Tabulation hashing performs quite uniformly well across the board, which is not entirely unexpected given its high degree of independence.

Overhead We measured the sequential overhead of the sum aggregation checker on the aforementioned AMD machine, as single-node performance on bwUniCluster proved to be too inconsistent to yield useful results (this is likely due to thermal limitations). The results are shown in Table 5. The

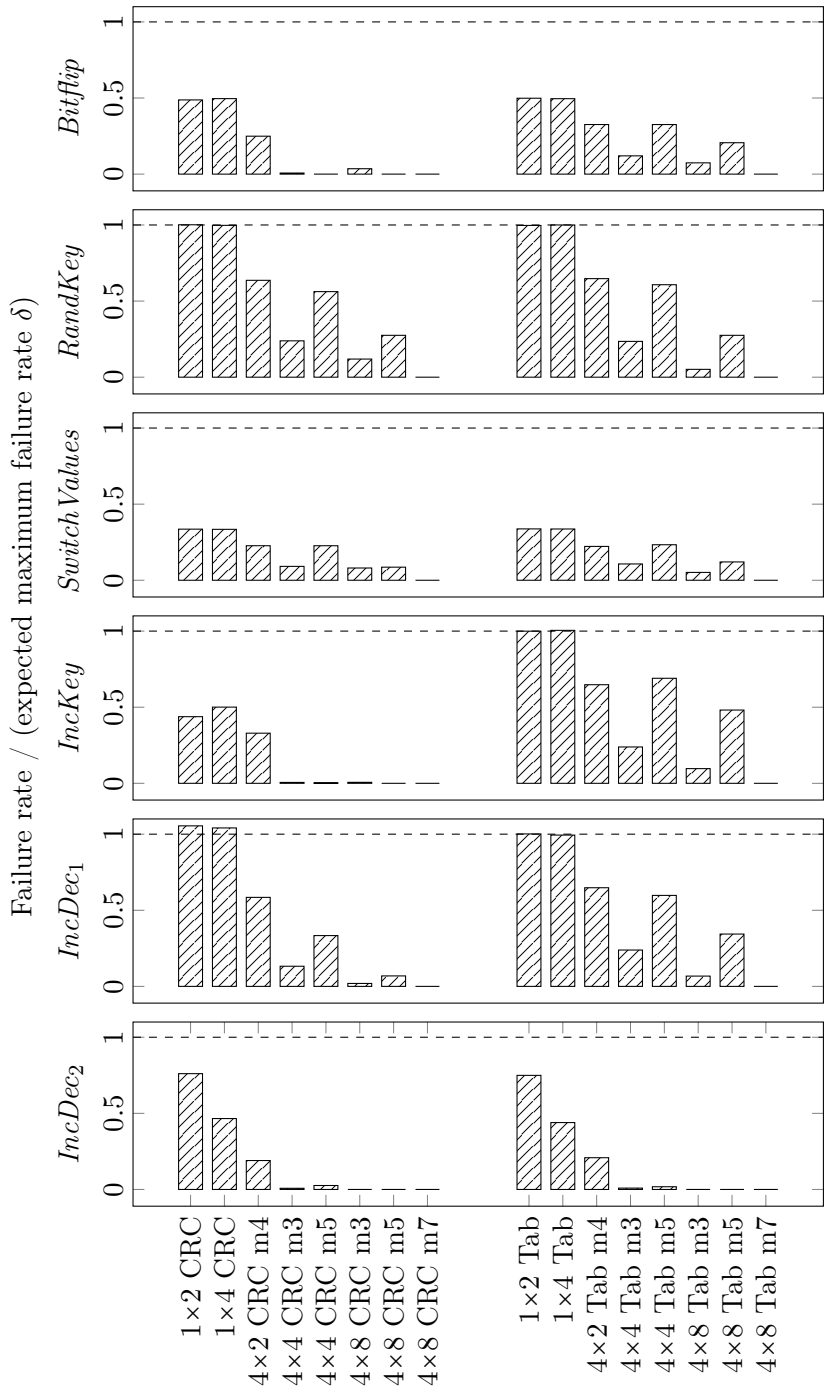

Fig. 3: Accuracy of the Sum Aggregation checker for different manipulators. The 50 000 input elements follow a power law distribution with 10^6 possible values, executed on 4 PEs and measured over 100 000 iterations.

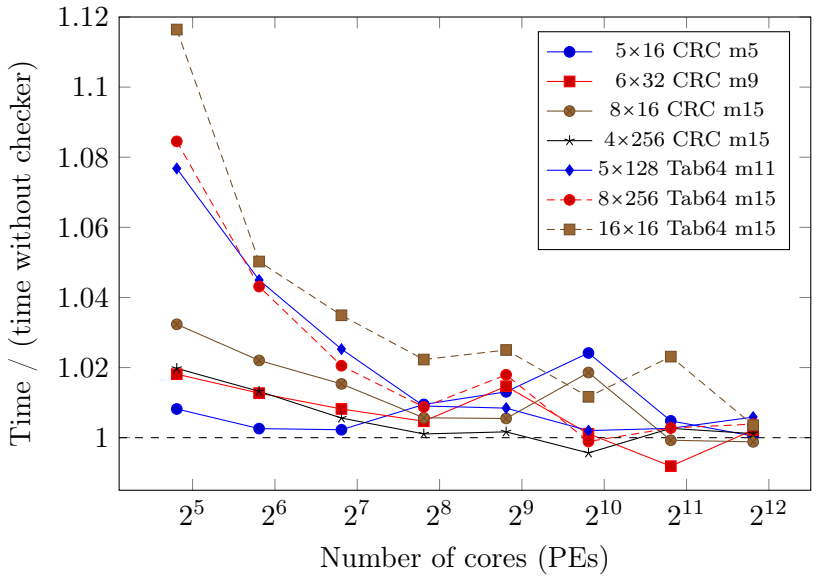

Fig. 4: Weak scaling experiment for Sum Aggregation checker with 125 000 items per PE, following a Zipf (power law) distribution. Average over 1000 runs.

configurations used provide high confidence in the correctness of the result and cover a wide range of practically relevant parameter choices. We can see that $\delta < 10^{-10}$ can be achieved with less than 5 ns per element on this 3.6 GHz machine. A configuration with very small table size (512 Bytes) and thus very little communication, with $\delta \approx 5 \cdot 10^{-20}$, can be realized with as little as 10 ns per element, approximately 36 cycles. For comparison, the main reduce operation takes approximately 88 ns per element using a single core of the same machine. This surprisingly low overhead was achieved by carefully engineering our implementation as described in Section 7.1. A naive implementation would likely cause at least one order of magnitude more overhead.

Scaling Behavior We performed weak scaling experiments, the results of which are shown in Fig. 4. These measurements suffer from a noticeable amount of noise because reduction and checker are interleaved—elements are forwarded to the checker as they are passed to the reduction. This is necessitated by the design of Thrill, as in related Big Data frameworks. As a result, we measure the entire reduce-check pipeline, the running time of which is influenced by multiple sources of variability, including the network, causing the aforementioned noise. We note that the results for a single node remain consistent with the sequential overhead measurements of the preceding paragraph.

It is clearly visible that the overhead introduced by the checkers is within the fluctuations introduced by the network and other sources of noise. Furthermore, starting with four nodes, data exchange for the reduction dominates overall running time, which becomes nearly independent from the checker's configuration and thus accuracy, although some impact of the number of iterations on running time remains visible. Nonetheless, the average overhead over all configurations is a mere 1.1 % when using more than a single node. Observe that even the slowest and most accurate configuration, "16×16 Tab64 m15" with $\delta < 6 \cdot 10^{-20}$, thus providing near-certainty in the correctness of the result, adds only 2.4 % to the average running time on two or more nodes.

Table 6: Manipulators for Sort/Permutation Checker

Name	Manipulation applied
Bitflip	Flips a random bit in the input
Increment	increment some element's value
Randomize	set some element to a random value
Reset	reset some element to the default value (0)
SetEqual	set some element equal to a different one

7.2 Permutation and Sorting

In this subsection, we evaluate the permutation and sorting checker of Section 5 with a workload of 10^6 integers chosen uniformly at random from the interval $0..10^8 - 1$. We implemented a single iteration of the hash-based permutation and sorting checker of Lemma 4, and truncate the output of the hash function to H bits for different values of H. Manipulations are applied *before* sorting in order to test the permutation checker and not the trivial sortedness check.

Detection Accuracy We measured detection accuracy of the sort checker using CRC-32C and tabulation hashing, two fast real-world hash functions with limited randomness. Our experiments in Appendix A show that tabulation hashing provides sufficient randomness to achieve the theoretically predicted detection accuracy on all tested manipulators. We further observe that CRC-32C does *not* seem to provide sufficient randomness for all of the manipulators listed in Table 6. In particular, we observed a significantly increased failure rate using the *Increment* manipulator where a single element of the input was incremented by 1. No significant deviations from expected performance were observed for the other manipulators.

Running Time On the aforementioned AMD machine, the overhead for local processing of input and output of the sorting operation was 2.0 ns per element for CRC32, and 2.8 ns when using 32-bit tabulation hashing. This corresponds to roughly 3.5 % of total running time when considering 100 000 elements. As the time spent on the hash function does not depend on how many of its output bits are used, the configuration did not have measurable impact on the running time.

We do not present scaling experiments for the permutation checker, as the only communication is a global reduction on a single integer value and one message sent and received per PE, both also containing exactly one integer value.

8 Conclusions

We have shown that probabilistic checking of many distributed big data operations is possible in a communication efficient manner. Our experiments show excellent scaling and that the running time overhead of the checkers in a distributed setting is below 5 % for sum aggregation and sorting, even when near-certainty in the correctness of the result is required. Accuracy guarantees predicted by theoretical analysis are achieved in practice even when using hash functions with limited randomness, and memory overhead is negligible. The basic methods used in our checkers are extremely simple, making manual verification of the correctness of the checker feasible. The existence of such checkers could speed up the development cycles of operations in big data processing frameworks by providing

correctness checks and allowing for graceful degradation at execution time by falling back to a simpler but slower method should a computation fail.

Future Work

Lower bounds on communication volume and latency for probabilistic distributed checkers would offer some insight into how far from an optimal solution our checkers are. Furthermore, it would be interesting to see whether more operations can be checked without the need for a certificate or requiring the purported result to be available at each PE. For example, could a probabilistic minimum aggregation checker with sublinear communication exist?

It would also be interesting to know whether the sum aggregation checker can be adapted for other data types such as floating point numbers without suffering from numerical instability issues such as catastrophic cancellation.

Acknowledgments

The authors acknowledge support by the state of Baden-Württemberg through bwHPC. We would like to thank Timo Bingmann for support with Thrill.

References

[1] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *2nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10, 2010.

[2] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl *et al.*, "The Stratosphere platform for big data analytics," *The VLDB Journal*, vol. 23, no. 6, pp. 939–964, 2014.

[3] T. Bingmann, M. Axtmann, E. Jöbstl, S. Lamm, H. C. Nguyen, A. Noe, S. Schlag, M. Stumpp, T. Sturm, and P. Sanders, "Thrill: High-Performance Algorithmic Distributed Batch Data Processing with C++," in *IEEE International Conference on Big Data*, 2016, pp. 172–183.

[4] J. F. Ziegler and W. Lanford, "Effect of cosmic rays on computer memories," *Science*, vol. 206, no. 4420, pp. 776–788, 1979.

[5] P. Sanders, S. Schlag, and I. Müller, "Communication efficient algorithms for fundamental big data problems," in *IEEE International Conference on Big Data*, 2013, pp. 15–23.

[6] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi *et al.*, "Spark SQL: Relational data processing in Spark," in *ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 1383–1394.

[7] V. Bala, J. Bruck, R. Cypher, P. Elustondo, A. Ho, C. Ho, S. Kipnis, and M. Snir, "CCL: A portable and tunable collective communication library for scalable parallel computers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no. 2, pp. 154–164, 1995.

[8] P. Sanders, J. Speck, and J. L. Träff, "Two-tree algorithms for full bandwidth broadcast, reduction and scan," *Parallel Computing*, vol. 35, no. 12, pp. 581–594, 2009.

[9] M. Dietzfelbinger, K. Mehlhorn, P. Sanders, and R. Dementiev, *Sequential and Parallel Algorithms and Data Structures – The Basic Toolbox*, 2018, manuscript in preparation.

[10] W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, and M. Ulbrich, Eds., *Deductive Software Verification - The KeY Book*, ser. Lecture Notes in Computer Science. Springer, 2016, vol. 10001.

[11] M. Blum and S. Kannan, "Designing programs that check their work," in *21st ACM Symposium on Theory of Computing*. ACM, 1989, pp. 86–97.

[12] R. M. McConnell, K. Mehlhorn, S. Näher, and P. Schweitzer, "Certifying algorithms," *Computer Science Review*, vol. 5, no. 2, pp. 119–161, 2011.

[13] R. Gennaro, C. Gentry, and B. Parno, "Non-interactive verifiable computing: Outsourcing computation to untrusted workers," in *Advances in Cryptology – 30th Annual Cryptology Conference*. Springer, 2010, pp. 465–482.

[14] L. Babai, L. Fortnow, L. A. Levin, and M. Szegedy, "Checking computations in polylogarithmic time," in *23rd ACM Symposium on Theory of Computing*, ser. STOC '91. ACM, 1991, pp. 21–32.

[15] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum, "Delegating computation: Interactive proofs for muggles," in *40th ACM Symposium on Theory of Computing*, ser. STOC '08. ACM, 2008, pp. 113–122.

[16] B. Parno, J. Howell, C. Gentry, and M. Raykova, "Pinocchio: Nearly practical verifiable computation," in *2013 IEEE Symposium on Security and Privacy*, ser. SP '13. IEEE, 2013, pp. 238–252.

[17] M. Walfish and A. Blumberg, "Verifying computations without reexecuting them," *Communications of the ACM*, vol. 58, no. 2, pp. 74–84, 2015.

[18] K.-H. Huang and J. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Transactions on Computers*, vol. 33, pp. 518–528, 1984.

[19] G. Bosilca, R. Delmas, J. Dongarra, and J. Langou, "Algorithm-based fault tolerance applied to high performance computing," *Journal of Parallel and Distributed Computing*, vol. 69, no. 4, pp. 410–416, 2009.

[20] L. A. Barroso, J. Clidaras, and U. Hölzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool, 2013, vol. 8, no. 3, Synthesis Lectures on Computer Architecture.

[21] A. A. Hwang, I. A. Stefanovici, and B. Schroeder, "Cosmic rays don't strike twice: understanding the nature of DRAM errors and the implications for system design," *ACM SIGPLAN Not.*, vol. 47, no. 4, pp. 111–122, 2012.

[22] M. N. Wegman and J. L. Carter, "New hash functions and their use in authentication and set equality," *Journal of Computer and System Sciences*, vol. 22, no. 3, pp. 265–279, 1981.

[23] K. Mehlhorn and P. Sanders, *Algorithms and Data Structures – The Basic Toolbox.* Springer, 2008.

[24] J. S. Plank, K. M. Greenan, and E. L. Miller, "Screaming fast Galois field arithmetic using Intel SIMD instructions." in *11th USENIX Conference on File and Storage Technologies*, ser. FAST '13, 2013, pp. 298–306.

[25] L. Hübschle-Schneider and P. Sanders, "Communication efficient algorithms for top-k selection problems," in *2016 IEEE International Parallel and Distributed Processing Symposium.* IEEE, 2016, pp. 659–668.

[26] H. Garcia-Molina, J. D. Ullman, and J. Widom, *Database Systems: The Complete Book*, 2nd ed. Prentice Hall Press, 2008.

[27] V. Gopal, J. Guilford, E. Ozturk, G. Wolrich, W. Feghali, J. Dixon, and D. Karakoyunlu, "Fast CRC Computation for iSCSI Polynomial using CRC32 Instruction," Intel Corporation, 2011, White Paper.

[28] M. Pătraşcu and M. Thorup, "The power of simple tabulation hashing," *Journal of the ACM*, vol. 59, no. 3, pp. 14:1–14:50, 2012.

[29] M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudo-Random Number Generator," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 8, no. 1, pp. 3–30, 1998.

A Permutation Checker Accuracy

In Figure 5 we show the results of accuracy experiments for the permutation checker as described in Section 7.2. This verifies whether the hash functions used provide sufficient randomness to detect manipulations. As discussed in Section 7.2, we can see that CRC-32C does not provide sufficient randomness to detect off-by-one errors in element values as simulated using the *Increment* manipulator (see Table 6).

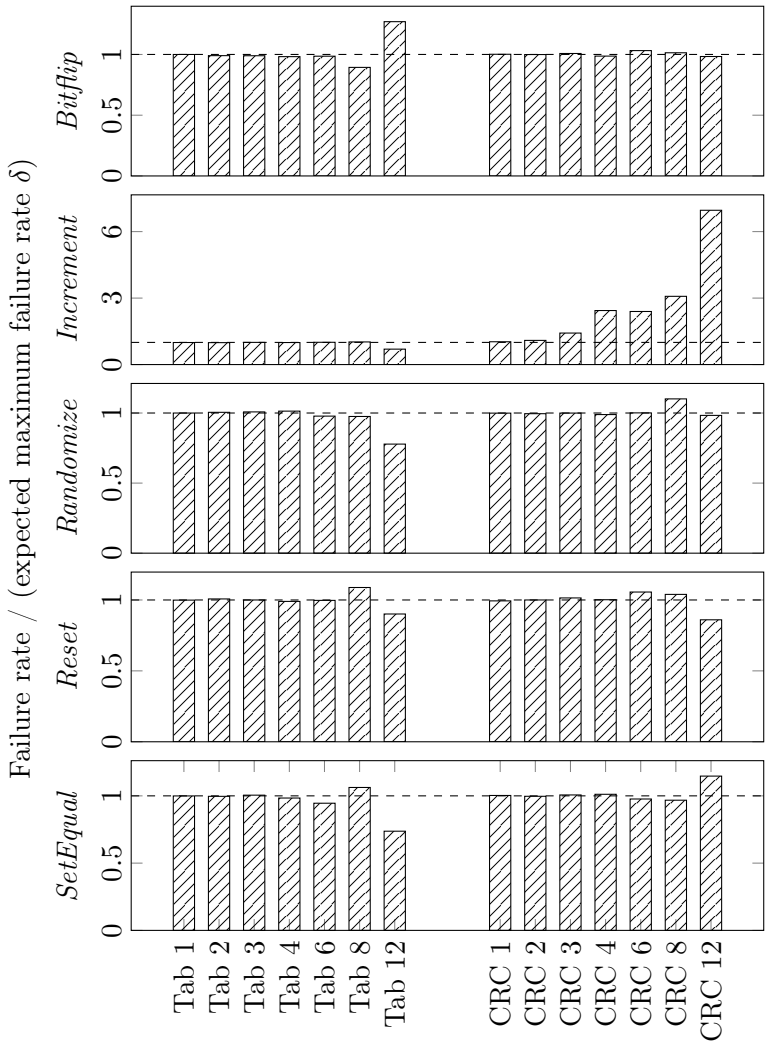

Fig. 5: Accuracy of the Permutation/Sort checker for different manipulators. Uniformly distributed input with 10^8 possible values, 10^6 input elements, 4 PEs, 100 000 iterations for each manipulator.