Algorithms for Dispersed Processing

Josef Spillner, Alexander Schill
Faculty of Computer Science
Technische Universität Dresden
01062 Dresden, Germany
Email: {josef.spillner,alexander.schill}@tu-dresden.de

***Abstract*—Highly scalable computing environments demand a parallelisation and distribution of processing tasks. Consequently, the data being processed is redundantly distributed to benefit from data locality characteristics, but also to increase safety, privacy and security objectives. Such distributed processing is coordinated by message passing or map-reduce programming styles. For partially replicated and dispersed data, however, the distributed processing poses new challenges because the required input data elements are not wholly available anymore to the processing tasks. Novel and adjusted processing algorithms which work under restricted assumptions thus become an important part of distributed infrastructures. We review, propose and analyse algorithms which align with split and dispersed data structures. Subsequently, we contribute and evaluate our implementations thereof in order to assess possible future applications on top of dispersed storage and multipath transmission of data.**

I. Problem Statement and Scope

Cloud providers typically offer data storage and transmission services along with proximity-aware processing services. Commercial providers as well as research and educational networks offer such combinations. To increase the reliability, full data replication can be used although there is a clear trend towards using partial redundancy due to decreased capacity requirements for the same availability guarantees [1], [2]. A downside of such coding schemes is that the resulting data fragments cannot easily be used in isolation for any meaningful computation. Some parts are missing and the remaining parts may be completely scrambled. This leads to a presumed necessity to retrieve the data over a network before processing it which can be both time-consuming and costly in a Cloud setting. A better alternative would be a smart local processing of fragments with limited coordination overhead.

One could argue that the expected slowdown is not worth the saved transmission cost. However, the Cloud compute services co-evolve with computing-enabled storage and networking hardware which bring processing capabilities closer to the data. Application-controlled disks are already available with network-connected drives (e.g. Ethernet HDDs, Kinetic) which alleviates the need for intermediary storage servers. It can be expected that future disks ship with programmable controllers which offer functionality beyond the current status detection and bad block trimming. Likewise, network equipment is made increasingly programmable by applications (e.g. software-defined networking, onePK) and network-attached FPGAs are increasingly used for number crunching without the need to shuffle data from and to the CPU [3].

These hardware and service trends support algorithms which explicitly benefit from the resulting fine-grained local data processing design. Nonetheless, existing algorithms in parallel and distributed computing are mostly working on coarse-grained units of data with zero-delay random access, including whole numbers and strings. Examples for popular algorithms are distributed event processing, message passing and map-reduce computations [4]. Only few approaches explicitly consider partial primary data structures or partial replication, especially in combination with privacy-preserving encryption schemes. Furthermore, most assume homogeneous grids, clusters or high-performance compute nodes rather than dynamic Cloud resources with arbitrary characteristics, including trust and resource (un)certainty [5]. Fig. 1 gives a high-level comparison of the proposed dispersed processing algorithms with conventional processing over dispersed data and other distributed processing techniques. f thereby denotes the function that would be computed locally, having the whole data as its input. m and r describe the typical map and reduce functions found in larger homogeneous cluster environments, working over distributed data. The fragment-aware mapper $m*$ are operating over different fragments of the data in a dispersed computation setting, e.g. a database for in-situ data or an event processing engine for in-transit data. Depending on the availability of compute nodes, algorithms benefit from a higher grade of distribution due to less data transmissions.

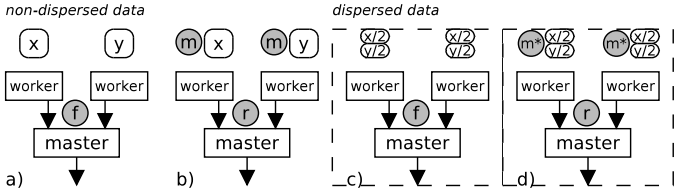

Fig. 1. Comparison of computing models: (a) central computation over distributed data after retrieval; (b) distributed mapping with subsequent reduction; (c) undesired central computation after retrieval and reconstruction; (d) dispersed mapping with subsequent reduction

Therefore, we argue for a smart splitting of small data units and variables into even smaller but still meaningful fragments and their dispersion according to the infrastructure properties. Whereas in reliable distributed data storage the coding and subsequent splitting is optimised for read and write performance, especially with erasure coding [6], we point out data processing tasks for which a type-dependent

bit splitting is crucial. Such techniques have been proposed before in application-specific contexts, for instance for bit-split intrusion detection [7] and column-oriented databases [8]. Reaching a more wide-spread use of reliable infrastructure services however requires a foundational view on type-dependent dispersion algorithms.

This paper contributes generic and type-dependent algorithms in the next section. Of special interest are search, calculation and statistical analysis algorithms. The algorithms presentation is followed by a description of implementations and practical evaluation results, a summary of limitations and finally an outlook towards future dispersed processing applications.

II. Algorithms

The literature knows about a lot of general-purpose algorithms. Among the most well-understood ones are searching and sorting [9] which apply to general-purpose data structures, matching applicable to string data types, and mathematical calculations which apply to numerical data types. Furthermore, there are graph, combinatorial, lexical and a vast number additional algorithms, some of which could be of relevance but will require future analysis [10]. There are also algorithms which do not apply to existing large data structures but instead they deterministically generate data based on initial seeds, such as the Fibonacci function and random number generators, and are therefore excluded from our analysis.

The following subsections report about precise algorithms which have successfully been applied to dispersed and potentially secured data. Compared to their holistic (non-dispersed) relatives, the run-time behaviour differences range from unnoticeable and predictable to severe and hard to estimate.

A. Search

The data type considered here is text, i.e. text characters and human-readable strings, in combination with binary character sequences. The input is a search pattern (substring) and a text body, and the output is a list of match positions or the empty list in the absence of any matches.

Existing single-pattern string searching algorithms assume the presence of the entire text: $list(positions) = search(text, pattern)$. Representatives are the naïve search, Boyer-Moore search, Knuth-Morris-Pratt search, Bitap and (as special case of the otherwise set-of-pattern) Rabin-Karp search. Some of these algorithms furthermore assume prior treatment of either the text and/or the pattern, e.g. through the creation of hashes, position tables or indexes.

In contrast, search over dispersed data occurs in parallel over k significant parts of the data: $list(positions) = search(text_1, pattern_1) \cap ... \cap search(text_k, pattern_k)$ These parts or fragments $text_i$ result from splitting the data which is a common operation in distributed storage applications. In contrast to simple chunking, which would allow the re-use of existing search algorithms save for a few corner cases like pattern wraps around the chunk ends, a per-byte or per-word split requires more sophisticated techniques.

1) Search over split data: If a text or text stream is split, the result is a set of fragments or fragment streams. Each element of the set needs to be searched separately. The search key is equally split into fragments for this technique to work. Each node performs the search function on its fragments and returns a probabilistic search result list which is guaranteed to contain all actual results, but also a number of false positives due to the information uncertainty which results from the missing bits per fragment. Eventually, the set of search results (positions) needs to be cross-correlated into an intersection set in order to eliminate the false positives and to achieve the final result.

When a greedy handling of the probabilistic search results is acceptable, follow-up functions for each result can also be offloaded to the nodes for increased performance, thus avoiding the cross-correlation bottleneck. Sometimes, the offloading even works in case a degraded, non-reconstructible state is reached. This technique is closely aligned to the working principle of Bloom filters and other probabilistic data structures. An example for such acceptable greedy functions is the aggressive deletion of cache entries which tolerates additional (false-positive) deletions apart from the risk of having to repeat retrievals. In the context of Cloud computing, another example is an optimistic reservation of resources based on presumed events, leading to improved application behaviour (no missing scale-outs and scale-ups due to false negatives) at the expense of potentially higher than necessary cost.

Fig. 2 compares the well-known naïve search algorithm with search over split data in the simple case of units of 8 bits being split into two times 4 bit fragments ($k = 2$). The notation format is *position:half-byte*, where position refers to the occurrence in the original data and half-byte specifies the higher (hi) and lower (lo) 4 bit, respectively. The algorithm gains complexity by having to look for the lower and upper 4 bit portions, respectively, in both the upper and lower 4 bits (grey arrows) in each fragment stream.

This technique can be generalised to splitting into k fragment streams with variable width b_k in bits per stream. For dispersal algorithms beyond simple splitting, for instance erasure coding or secret sharing, it is not applicable due to the lack of structure preservation. However, in order to not gain searchability at the expense of security and privacy, the splitting may be combined with homomorphic encryption per fragment which itself exhibits a structure-preserving property for a number of target algorithms [11]. Furthermore, in order to maintain the reliability of data, fault-tolerance schemes from RAID-like systems can be adopted. They take k data fragments and add m parity fragments to reconstruct data for up to (including) m simultaneous erasures [12]. Search with both encrypted and redundant data will be explained after giving a syntactic example of simple search over dispersed data.

Listing 1 contains the entire search algorithm in Python notation for the case of $k = 2$. The variable LS refers to the last state and SC to the second choice repetition. The input data are the haystack H, the needle N, as well as a start position idx and a padding indicator for texts with an odd original length.

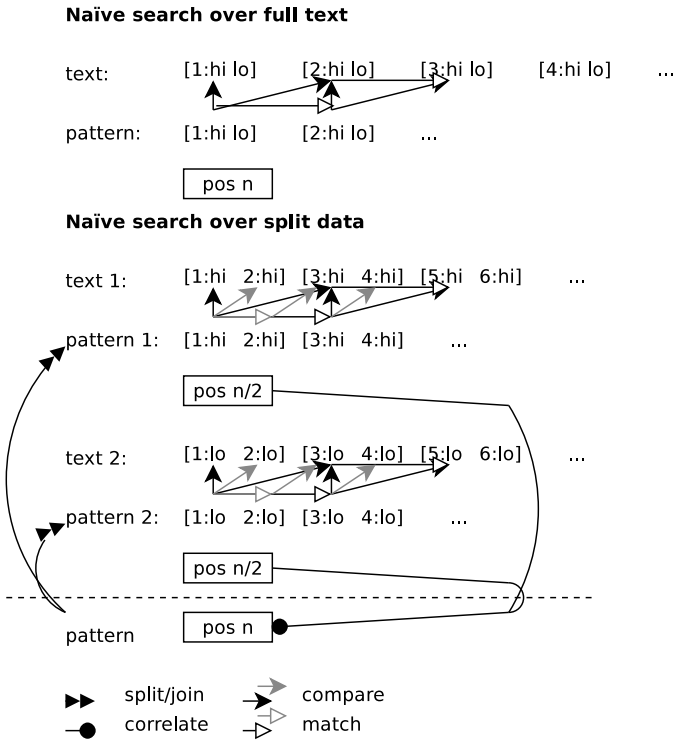

Fig. 2. Full text and split text search in comparison

Listing 1. Search over dispersed data

```
def findbits(H, N, idx, padding):
bitfilter = [0xF0, 0x0F]
for i in range(idx, len(H)):
  k = i
  j = 0
  ctr = 0
  offset = 0
  SC = False
  reset = False
  LS = -1
  while True:
    next = False
    if ctr % 2 == 0:
      if (H[k] & 0xF0) == (N[j] & 0xF0) and
          not (SC and ctr == 0) and LS != 1:
        LS = 1
      elif (H[k] & 0x0F) == ((N[j] & 0xF0) >>
          4) and LS != 0:
        LS = 0
        offset = 1
        next = True
      else:
        if not SC:
          reset = True
        else:
          break
    else:
      if (H[k] & 0x0F) == (N[j] & 0x0F) and
          not (SC and ctr == 0) and LS != 0:
        LS = 0
        next = True
      elif (H[k] & 0xF0) == ((N[j] & 0x0F) <<
          4) and LS != 1:
        LS = 1
      else:
        if not SC:
          reset = True
        else:
          break
    if reset:
      k = i
      j = 0
      ctr = 0
      offset = 0
      SC = True
      reset = False
      continue
    ctr += 1
    if next:
      k += 1
    if ctr % 2 == 0:
      j += 1
    if ctr + padding == len(N) * 2:
      return i * 2 + offset
    if j >= len(N) or k >= len(H):
      break
return -1
```

2) Search with redundant data: The aforementioned algorithm is defined for $n = k/m = 0$ schemes. In practice, however, most distributed storage strategies include redundant data for failure detection and recovery. With dominant disk array parity schemes, the redundant data is either moved to a dedicated node or interspersed diagonally. For search, the dedicated node scheme has the disadvantage of excluding this node from processing due to the nature of redundant data. Furthermore, when one node fails, the data can be recovered but the search will not be possible until the point of recovery. Diagonal schemes work well and are preferred for wear-levelling. But they lead to non-elegant processing algorithms due to the rotating combination of nodes depending on the search position. Therefore, we introduce a simple new scheme with horizontal parity rows. We call the scheme R-Code due to its similarity to X-Code, S-Code and further deviations thereof [13]. Compared to those, R-Code accepts an arbitrary number of failing nodes ($m > 0$) and does not impose restrictions on the overall number of nodes except for a reasonable minimum ($n \geq 3$). It is technically applicable to $n = 2$ although this would lead to the ability for each node to fully reconstruct the data of the respective other one. R-Code divides each chunk of data into $n - m$ stripes which are then dispersed into n fragments onto a $(n - m, n)$ matrix.

The first parity is calculated as the XOR parity over $n - m$ data values. All remaining parities are linear combinations of these $n - m$ data values with coefficients taken from the $mx(n - m)$ Vandermonde matrix, exactly as defined in [12]. A single storage node must never have a parity value that depends on a data values this node actually holds and further there must be no two parity values on this node which rely on the same data value. Having fulfilled these requirements, the vertical scheme from [12] can be transformed into a horizontal scheme having the same properties.

We suggest the notation (n, R) with $n = k$ significant nodes and $m = 0$ redundant nodes, but R redundant stripes per chunk

across all nodes. In terms of redundancy, an (n, R)-R-Code thus corresponds to a $(k = n - m, m = R)$ erasure code.

Fig. 3 compares a conventional redundant storage configuration, akin to RAID-4 with parity codes as redundant data across all nodes stored in a dedicated node, with the simplified R-Code scheme for $R = 1$. For distributed fragment search applications in the Cloud, R-Code offers a $\frac{1}{n}$ performance boost over $(k, 1)$ conventional coding. Given the same amount of data, a higher distribution will however result in more false positives which depending on the network and processing characteristics may offset this performance gain.

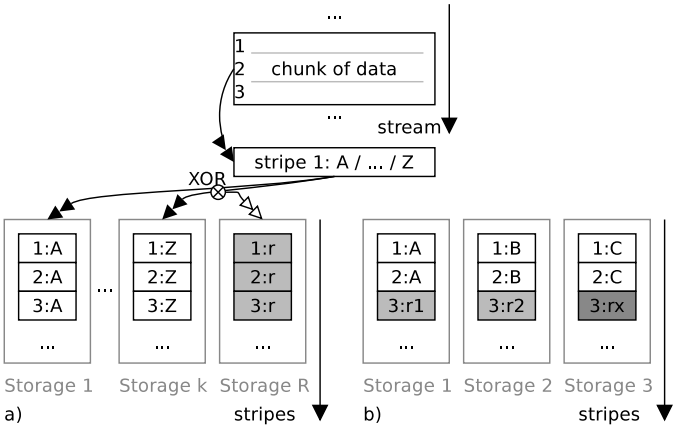

Fig. 3. Storage with redundancy: (a) parity or erasure coding in extra node for $k > 1$ and $m = 1$; (b) R-Code for $k = 2$ and $R = 1$

3) Search with encrypted data: As all k data fragments are constructed by simple splitting of the original data, confidentiality might by a concern when storing critical or personal information. Privacy preservation for stored data is typically achieved by application of some sort of encryption mechanism. The issue with classical encryption in our case is the lack of structure within the ciphertext which we need to perform our dispersed search algorithm and evaluate the offloaded functions. Therefore a structure preservation, e.g. some sort of homomorphism between the unencrypted plaintext and the encrypted ciphertext is needed. Depending on the desired functionality homomorphic encryption [14], homomorphic hashes [15], symmetric ciphers in a pseudo-one-time-pad mode (OFB or CTR mode) and order preserving encryption [16] are possible candidates. A homomorphism would therefore exist regarding addition or multiplication within a finite field, addition modulo 2 (XOR) or regarding a function evaluating the order of two elements.

B. Calculation

The data types considered here are both integer numbers and fixed-point numbers. Variable floating-point dispersion with data inspection before the splitting takes place is subject to future work. Alternatively, floating point numbers can be converted into fixed-point representations [17].

In dispersed calculation, a client node splits the numbers to be operated on. Server nodes (called processors) perform their operations on partial numbers and return partial results. The client node then reconstructs the full result. As opposed to working on chunks of data, dispersed calculation either needs to restrict or to modify the semantics of operations depending on the dispersion. This is similar to homomorphic encryption operations which work on encrypted data without distribution. Dispersed calculation is clearly an open research topic as it presents additional challenges.

A suitable classification of dispersed calculation is driven by the types of data to be worked on and by the combination of splitting with further data modification, including encryption and compression. In our work, we consider plain and encrypted split integer arithmetic, fixed-point arithmetic (which through transformation also applies to floating-point numbers), and statistical functions which are commonly used in data analysis domains.

1) Calculation on integers: Table I gives a simple example of a dispersed $71 + 19$ addition operation. Two operands are split into a higher and lower 4-bit fragment each. Both pairs of fragments can be independently summated. Afterwards, both sums are accumulated again at the client. This scheme introduces both network overhead and two additional summation function calls compared to a non-dispersed summation.

TABLE I
ADDITION OF TWO NUMBERS WITHOUT REDUNDANCY

Number (decimal)	Number (binary)	Fragments
71	01000111	a_1: 0100; b_1: 0111
19	00010011	a_2: 0001; b_2: 0011
71+19=90	01011010	
Partial results are a_1+a_2: 0101; b_1+b_2: 1010		
Computed result is (a_1+a_2 « 4) + b_1+b_2: 01011010 $\hat{=}$ 90		

For $k \geq 2$, x summands and the word size w, the scheme can be generalised into the map-reduce form $sum = \sum_{i=1}^{k}(\sum_{j=1}^{x} fragment_{ij} \ll w/k)$.

More generally, if two integers a and b are split into k fragments f_{ai} (f_{bi}) ($1 <= i <= k$) with f_{a1} (f_{b1}) being the lowest order bits and f_{ak} (f_{bk}) having the highest order, fragment i of both splits is having b_i bits. We compute the dispersed sums $s_i = f_{ai} + f_{bi}$ with s_i having again b_i bits and c_i being the carry-bit of the s_i computation and $c_0 = 0$. The overall final result s is then given by $s = \sum_{i=1}^{k}(s_i + c_{i-1}) \cdot 2^{\sum_j = 1^{i-1} b_j}$.

For multiplication, a two-node distribution does not suffice anymore. Instead, the data needs to be dispersed to four nodes at the minimum. Table II gives the example for $71 * 19$. No node has full access to either of the factors. Furthermore, this strategy introduces both network overhead, three additional multiplication function calls, and one additional summation at the client.

Both arithmetic functions are schematically shown in Fig. 4 for a second example of $23 + 24$ and $23 * 24$, respectively.

2) Calculation on encrypted integers: When the confidentiality of data is high, the need to protect numbers in remote calculation scenarios rises as well. As outlined in the

TABLE II
MULTIPLICATION OF TWO NUMBERS WITHOUT REDUNDANCY

Number (decimal)	Number (binary)	Fragments
71	01000111	a_1: 0100; b_1: 0111
19	00010011	a_2: 0001; b_2: 0011
71*19=1349	10101000101	
Partial results are $a_1 * a_2$: 4; $a_1 * b_2$: 12; $a_2 * b_1$: 7; $b_1 * b_2$: 21		
Computed result is $(a_1 * a_2 \ll 8)$ + $(a_1 * b_2 \ll 4)$ + $(a_2 * b_1 \ll 4)$ + $b_1 * b_2$: 10101000101 $\triangleq$ 1349		

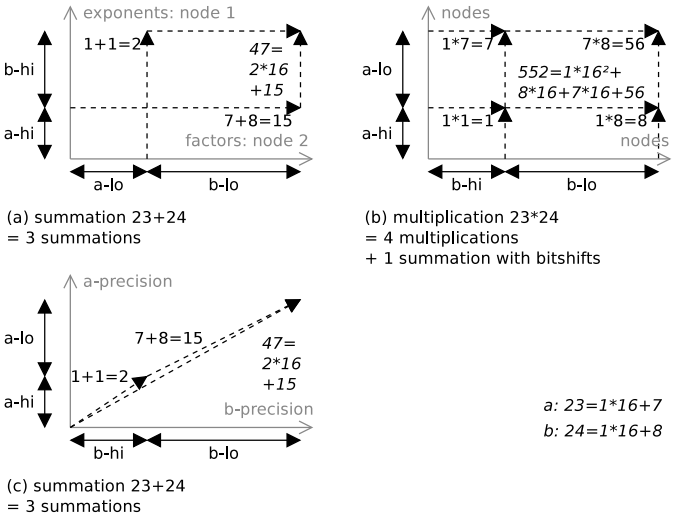

Fig. 4. Schema and example for (a) dispersed addition, (b) dispersed multiplication and (c) alternative visualisation for (a)

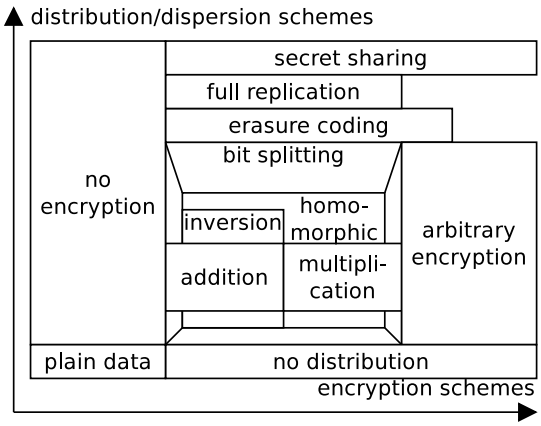

Fig. 5. Classification of dispersion and encryption options and the intersection between both for individual processing tasks

introduction, there are classes of homomorphic encryption algorithms which for a certain set of functions allow for performing these functions on encrypted input and obtaining an encrypted output [11].

In the Paillier cryptosystem, for example, additions and multiplications can be performed which combines nicely especially with the need to protect whole numbers for dispersed multiplication. In this cryptosystem, the formula $(1+1)*2 = 4$ results in, for instance, $(15351 + 8856) * 3504 = 8616$ with the 8-bit encryption key parameters $n = 143$ and $\lambda = 60$. Applying these transformations leads to about twice the amount of data compared to the non-encrypted variant and a higher computation time due to the need to encrypt and decrypt locally in the map-reduce paradigm.

Fig. 5 shows the interesting intersection point between processing over dispersed data and processing over homomorphically encrypted data.

3) Statistical functions: There are a number of statistical and data analytics functions which can be broken down into a combination of isolated simple calculation functions. Due to this relation, a number of statistical operations can be performed directly on distributed fragments without the needs to retrieve and combine them.

Counting can be performed on a single node. Summation with any number of summands, i.e. value aggregation, can be performed as generalisation of the addition and subtraction already presented. Average values can also be calculated, as shown by example in Table III, assuming the proper handling of rational numbers or the tolerance of rounding errors.

TABLE III
AVERAGE OF TWO NUMBERS WITHOUT REDUNDANCY

Number (decimal)	Number (binary)	Fragments
71	01000111	a_1: 0100; b_1: 0111
19	00010011	a_2: 0001; b_2: 0011
ø(71,19)=45	00101101	
Partial results are ø(a_1,a_2): 2.5; ø(b_1,b_2): 5		
Computed result is (ø(a_1,a_2) « 4) + ø(b_1,b_2): 00101101 $\triangleq$ 45		

On the other hand, a number of statistical moments cannot easily be determined without full access to all numbers. Among them are minimum, maximum, median and spread, which can only be determined among the higher-bits fragments with a certain likelihood and fuzziness. For them, using a handover protocol is necessary.

4) Calculation on fixed-point numbers: Smart data coding and splitting strategies beyond just bit and byte positions are useful for special-purpose calculations. For instance, a fixed-point number can be split at the point in order to tolerate the unavailability of a particular fragment - the fractional part - at the cost of losing the precision, which is still preferred over losing the number entirely. Fuzzy calculation and estimation with error tolerance can be performed with imprecise numbers resulting from large volumes of data dispersed across multiple nodes.

Fig. 6 shows the mapping from data fragment availability to numerical precision for a $k = 2, m = 1$ dispersion. The three fragments are called (i)nteger, (f)ractional and (r)edundant, respectively.

This splitting strategy can be applied to any combination of multiple-of-2 k and m. The fractional part range is (0..1). It is mapped to the range of the fragment capacity (e.g. 8 or 16 bit) and thus turned into a floating point number whose integer part is used as fragment. The resulting deviation occurs even when all fragments are available. Using a reasonable fragment

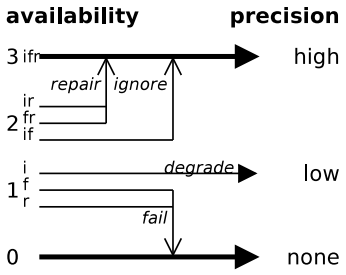

Fig. 6. Numerical precision depending on the availability of data fragments

size prevents a significant loss of precision.

C. Handover algorithms

The previous calculation schemes all assume a full isolation of all nodes, which in a Cloud context is often a desired property due to privacy concerns. On the other hand, a number of algorithms depend on an underlying *handover protocol* in which the master node extends the *map-reduce* pattern into *map-carry-reduce* by taking some result from a node and passing them as additional data to the subsequent one. Handover protocols are required for bit shifting (left and right carry bits), sorting, as well as many statistical moments including the minimum, maximum, median and spread. Furthermore, they are required for strictly fixed-size numbers. One intrinsic design property is that they are iterative rather than parallel; a second one is that they lower the confidentiality of data to some degree.

The general handover protocol is shown in Fig. 7. At the first node, a set of candidate fragments is determined. Together with their positions, the resulting list of tuples of cardinality $|\alpha|$ is aiding the calculation at the second node. Its result is a subset of α and its cardinality $|\beta|$ is equal to or less than $|\alpha|$. The final list of fragments is $\hat{X}$ which represents complete numbers. Usually $1 \leq |\hat{X}| \leq 2$ for statistical moments.

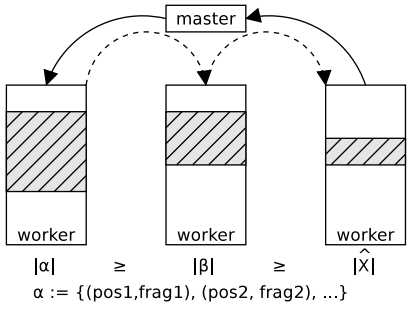

Fig. 7. Scheme for the map-carry-reduce handover protocol over dispersed fragments

III. Implementation Results

Our splitting, searching, calculation and statistical analysis implementations are structured into modules, both figuratively (as logical collection of functions) and literally (as Python modules and optimised C libraries). All implementations and associated measurement scripts, results and plots are publicly available from the Dispersed Algorithms Git repository[1]. In this section, we concentrate on the local calculation performance; the repository also contains fully networked examples.

A. Splitter

We have developed a Python splitter for a fixed fragment size of 4 bits each ($k = 2$) and a corresponding C splitter which reimplements the Python splitter but also allows for different configurations. It offers a byte-aligned bit splitting over streams of data both with redundancy-less schemes, e.g. 3/3/2 bits ($k = 3$), and with single parity redundancy ($m = 1$).

Fig. 8 compares the splitting performance by measuring the throughput. All measurements were conducted on an Intel Core i7 M620 CPU with 4 cores @ 2.67GHz and 6 GB of free memory. In order to exclude the disk performance, 1 GB was reserved for a RAM disk (via tmpfs) on which a 382 MB video file was stored and then split into $k = 2$ parts with and without redundancy. One can clearly see the miserable performance of the Python implementation, which in addition also caused a larger than planned consumption of memory, requiring us to interpolate the Python results with half of the video files to avoid an out-of-memory error. Our implementations are compared against Jerasure 1.1A codecs for both ReedSolVan and CauchyGood coding.

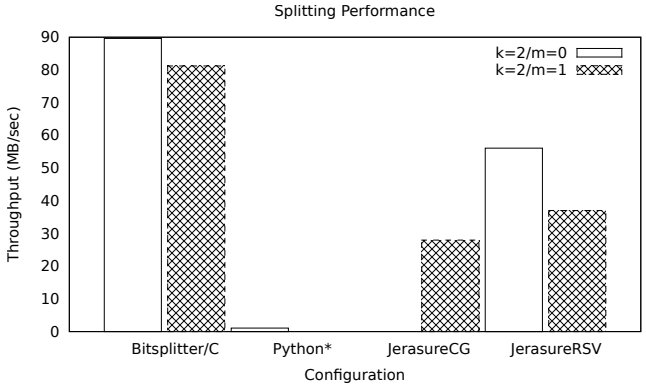

Fig. 8. Data splitting performance comparison

B. Search module

Similar to the splitting, we have implemented distributed search over multiple fragment streams in both Python and C. The two search invocations involved a book text, Jules Verne's Marco Polo, and looking for both a unique single-occurrence word (Cai-ping-fu) with 0% false positives due to its length and a rather common word (Polo) with 167 occurrences and a medium false positive rate of 46.98%. The performance comparison is shown in Fig. 8. One can clearly see the order of magnitude penalty from the non-dispersed search function in Python (which is actually implemented in C) to our C implementation, and the even higher relative penalty from the C search to the dispersed Python search. Interestingly, the

[1]Dispersed Algorithms: git://serviceplatform.org/git/dispersedalgorithms

deviation between the search times of two words is minimal in the C implementation, which makes it a good candidate for random search tasks with predictable performance.

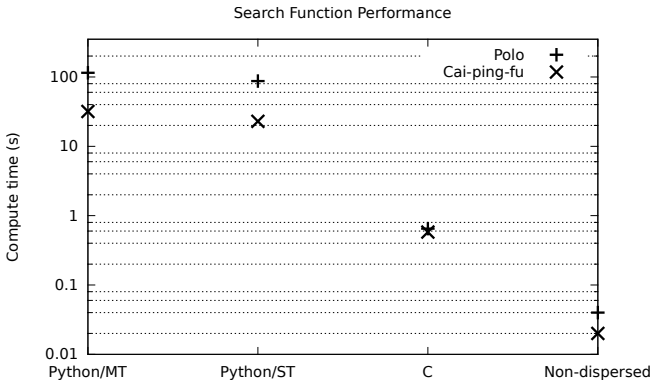

Fig. 9. Data searching performance comparison

C. Calculation module

The calculation experiment compares a native addition of one million randomly generated 8-bit numbers with the equivalent of k dispersed additions ($2 \leq k \leq 9$) followed by a central aggregation of all k fragment sums. Fig. 10 shows how the overall single-thread performance is severely affected by the introduction of dispersion but how it degrades linearly with every additional node. In practice, multi-threaded implementations and a real parallelisation across several nodes will regain much of this degradation which is shown as a convergence towards the optimum boundary in the figure.

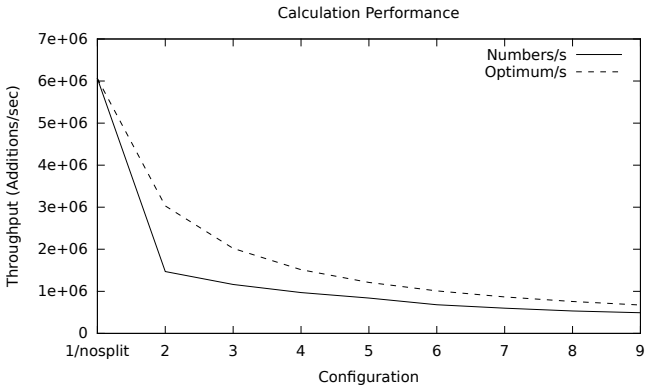

Fig. 10. Integer addition performance comparison

In addition, we were interested in the practical implications of using fixed-point arithmetics. Fig. 11 is the result of comparing 10000 randomly generated floating point numbers in the range $[0..1]$ with their (1,1)-fixed-point equivalents, i.e. one byte for the integer and another byte for the fractional part. The deviation is in per mille relative to the numerical space of 256 distinct values. One can clearly see that for practical purposes, the deviation can often be ignored even when choosing a compact encoding.

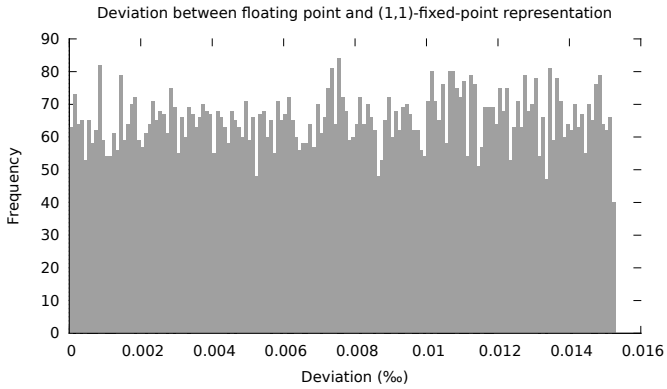

Fig. 11. Distribution function of (1,1)-fixed-point precision deviations

D. Handover module

We have simulated 1000 additions of dispersed 32-bit integers with an R script and the general multiple precision arithmetic library (GMP). The average number of rounds for the carry-bit propagation remains relatively low, not more than two, even when using 16 fragments. The associated risk for carry-bit errors rises up to 20%, but is generally less than half of this for the first round. Fig. 12 shows the simulation results.

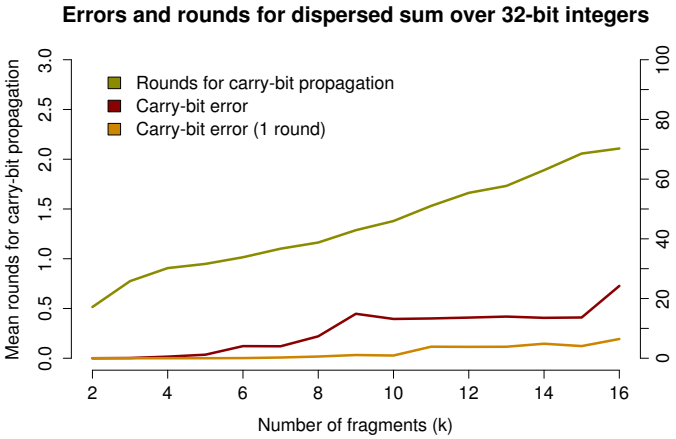

Fig. 12. Number of rounds and mean errors for the carry-bit propagation for addition

Fig. 13 visualises the performance of another simulation of statistical moment calculation which requires the handover protocol. One million numbers in a small range (sr, 0-100) and a large range (lr, 0-1000000) were produced randomly, producing a normal distribution. Of each set, the minimum, maximum and median values were determined. Up until 6 nodes there is no noticeable performance drawback. With higher node numbers, the drawback grows significantly.

IV. Limitations

In this section, the limitations across all algorithms shall be summarised and consolidated.

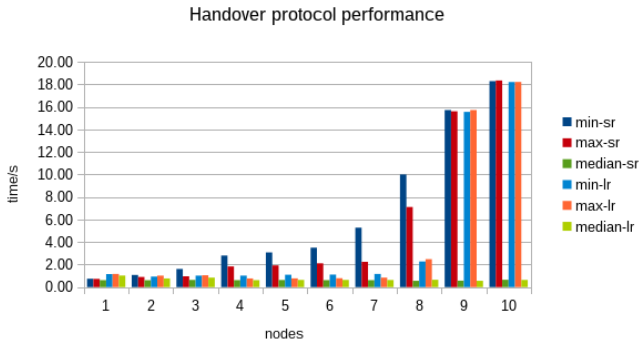

Fig. 13. Performance measurement of the handover algorithm for three statistical moments

According to our findings, some algorithms are apparently non-implementable under the isolated dispersed processing scheme. One prominent example is sorting. If fragments change their position relative to other fragments on one node, the resulting order has no relationship with sorted fragments on another node. Some of these algorithms can be piggy-backed by a handover protocol, but more work is needed to fully explore its usefulness and its downsides concerning trust and privacy.

Some algorithms benefit from a non-uniform distribution of data across heterogeneous nodes. For instance, in the R-Code scheme, the last parity fragment is involved in any write operation and therefore should be placed on a RAM disk or another write-tolerant storage target. Involving knowledge about the non-functional properties of the nodes will be a key aspect to achieve well-balanced dispersed processing.

V. Summary and Research Outlook

Applications processing generic, text, numerical and multimedia data benefit from special algorithms which are aware of the degree of dispersion of data fragments. In many cases, such algorithms can work very close to the data assuming an appropriate hardware or service infrastructure in order to avoid costly network transmissions. Furthermore, awareness of the availability and encryption of fragments leads to adaptive applications which let programmers worry less about failures and privacy issues in distributed systems. We have introduced algorithms with several examples on text searching, calculation and statistical analysis over data with structure-preserving dispersion for higher parallelisation, redundancy for higher availability and homomorphic encryption for higher protection to demonstrate these claims. Our work has shown that there are clear performance drawbacks but there is also a potential for transmission savings which may make up the drawbacks depending on the connection capacity and pricing. In the future, we are going to extend the work with controlled quality degradations over dispersed numeric and multimedia data.

Acknowledgements

This work has been partially funded by the German Research Foundation (DFG) under project agreement SCHI 402/11-1. We would like to thank Martin Beck for contributing ideas to the search over R-Code data and to homomorphic encryption.

References

[1] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur, "XORing Elephants: Novel Erasure Codes for Big Data," *The 39th International Conference on Very Large Data Bases (VLDB)*, vol. 6, no. 5, pp. 325–336, August 2013.

[2] B. Fan, W. Tantisiriroj, L. Xiao, and G. Gibson, "DiskReduce: Replication as a Prelude to Erasure Coding in Data-Intensive Scalable Computing," Carnegie Mellon University Parallel Data Laboratory, Tech. Rep. CMU-PDL-11-112, October 2011.

[3] O. Knodel, T. B. Preußer, and R. G. Spallek, "Next-Generation Massively Parallel Short-Read Mapping on FPGAs," in *22nd IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, Santa Monica, California, USA, September 2011, pp. 195–201.

[4] A. Celesti, N. Peditto, F. Verboso, M. Villari, and A. Puliafito, "DRACO PaaS: A Distributed Resilient Adaptable Cloud Oriented Platform," in *IEEE 27th International Symposium on Parallel & Distributed Processing Workshops and PhD Forum (IPDPSW)*, Cambridge, Massachusetts, USA, 2013, pp. 1490–1497.

[5] D. Warneke and O. Kao, "Exploiting Dynamic Resource Allocation for Efficient Parallel Data Processing in the Cloud," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 6, pp. 985–997, June 2011.

[6] J. L. Gonzalez, V. Sosa-Sosa, B. Bergua, L. M. Sanchez, and J. Carretero, "Fault-Tolerant Middleware Based on Multistream Pipeline for Private Storage Services," in *7th International Conference for Internet Technology and Secured Transactions (ICITST)*, London, UK, December 2012, pp. 548–555.

[7] L. Tan and T. Sherwood, "Architectures for Bit-Split String Scanning in Intrusion Detection," *IEEE Micro*, vol. 26, no. 1, pp. 110–117, January/February 2006.

[8] D. J. Abadi, P. A. Boncz, and S. Harizopoulos, "Column-oriented Database Systems," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1664–1665, August 2009.

[9] D. E. Knuth, *The Art of Computer Programming*. Addison-Wesley, 1998, vol. 3 - Sorting and Searching, 2nd Edition.

[10] J. Lin and M. Schatz, "Design Patterns for Efficient Graph Algorithms in MapReduce," in *Proceedings of the Eighth Workshop on Mining and Learning with Graphs (MLG)*, Washington D.C., USA, August 2010, pp. 78–85.

[11] M. Mani, "Enabling Secure Query Processing in the Cloud using Fully Homomorphic Encryption," in *Proceedings of the Second Workshop on Data Analytics in the Cloud (DanaC)*, New York City, New York, USA, June 2013, pp. 36–40.

[12] J. Plank, "A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems," *Software, Practice & Experience*, vol. 27, no. 9, pp. 995–1012, September 1997.

[13] L. Xu and J. Bruck, "X-Code: MDS Array Codes with Optimal Encoding," *IEEE Transactions on Information Theory*, vol. 45, no. 1, pp. 272–276, January 1999.

[14] C. Gentry, "Fully Homomorphic Encryption Using Ideal Lattices," in *Proceedings of the 41st Annual ACM Symposium on Theory of Computing (STOC)*, Bethesda, Maryland, USA, June 2009, pp. 169–178.

[15] V. Lyubashevsky, D. Micciancio, C. Peikert, and A. Rosen, "SWIFFT: A Modest Proposal for FFT Hashing," in *Fast Software Encryption: 15th International Workshop FSE, Revised Selected Papers*, ser. LNCS, 2008, vol. 5086, pp. 54–72.

[16] A. Boldyreva, N. Chenette, and A. O'Neill, "Order-Preserving Encryption Revisited: Improved Security Analysis and Alternative Solutions," in *Proceedings of the 31st Annual Conference on Advances in Cryptology (CRYPTO)*, ser. LNCS, 2011, vol. 6841, pp. 578–595.

[17] M. Franz, B. Deiseroth, K. Hamacher, S. Jha, S. Katzenbeisser, and H. Schröder, "Secure Computations on Non-Integer Values," in *2nd IEEE International Workshop on Information Forensics and Security (WIFS)*, Seattle, Washington, USA, December 2010, pp. 1–6.