2016 16th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing

Towards Memory-Optimized Data Shuffling Patterns for Big Data Analytics

Bogdan Nicolae*, Carlos Costa†, Claudia Misale†, Kostas Katrinis*, Yoonho Park†
*IBM Research, Ireland
{bogdan.nicolae, katrinisk}@ie.ibm.com
†IBM T. J. Watson Research Center, USA
{chcost, cmisale, yoonho}@us.ibm.com

***Abstract*—Big data analytics is an indispensable tool in transforming science, engineering, medicine, healthcare, finance and ultimately business itself. With the explosion of data sizes and need for shorter time-to-solution, in-memory platforms such as Apache Spark gain increasing popularity. However, this introduces important challenges, among which data shuffling is particularly difficult: on one hand it is a key part of the computation that has a major impact on the overall performance and scalability so its efficiency is paramount, while on the other hand it needs to operate with scarce memory in order to leave as much memory available for data caching. In this context, efficient scheduling of data transfers such that it addresses both dimensions of the problem simultaneously is non-trivial. State-of-the-art solutions often rely on simple approaches that yield sub-optimal performance and resource usage. This paper contributes a novel shuffle data transfer strategy that dynamically adapts to the computation with minimal memory utilization, which we briefly underline as a series of design principles.**

***Index Terms*—big data analytics, data shuffling, memory-efficient I/O, elastic buffering**

I. Introduction

Data is the new natural resource. Its ingestion and processing leads to valuable insight that is transformative in all aspects of our world [1]. Science employs data-driven approaches to understanding nature and developing prompt answers to fundamental scientific and societal questions across domains and disciplines. In all industries and sectors, data science has been a fast growing value generator, leveraging the abundance and growth of data availability due to various contemporary factors (e.g. connectivity, mobile, social media) and coming up with deeper insight solutions to prompt business cases.

As Big Data Analytics starts becoming essential across value chains, there is a natural need for shortened time-to-insight and improved economy of scale. In this regard, data-oriented programming models that separate the computation from its parallelization gained rapid popularity beginning with the MapReduce [2] paradigm. However, as the user has to worry less about parallelization, the runtime becomes increasingly complex. One major contribution in this context was the *data-locality centered design*: the storage layer is co-located with the compute elements and exposes the data locations such that the computation can be scheduled close to the data. Using this approach, data movements over the network are drastically reduced, which improves performance and scalability. However, the push for performance prompted the need for better integration between the data flow and the computational flow, in order to avoid important overheads due to the storage layer. To this end, a new generation of in-memory big data analytics frameworks is increasingly gaining popularity over MapReduce, such as *Apache Spark* [3]. By making heavy use of in-memory data caching, Spark minimizes the interactions with the storage layer, which further reduces I/O bottlenecks due to slow local disks, extra copies and serialization issues.

Memory, however, is a precious resource: prices have stopped dropping for the past years yet the number of cores keeps growing. Thus, memory per core gets smaller and smaller yet there is no evidence that the bytes/flop requirements will be dropping. As a consequence, the "preciousness" of main memory as a system component is highly likely to be increasing; more so when considering the in-memory computing trend of big data analytics, which is necessary to achieve the desired time-to-solution. Furthermore, it is important to note that many users rely on utility computing models (e.g. clouds) to run big data analytics, which use resource consumption as the decisive factor in pricing. Thus, efficient memory utilization is important from this perspective as well.

Therefore, modern big data analytics have to reconcile several trade-offs: support user-friendly programming models that deliver high performance and scalability with minimal memory utilization. One difficult challenge in this context is *data shuffling*. It is a fundamental pattern that facilitates the implementation of a large family of collaborative data aggregation primitives (e.g. reduce, join, groupby). With respect to performance and scalability, data shuffling is challenging because it involves complex all-to-all communication patterns over the network. In fact it is the one of the main factors that could potentially limit the overall effectiveness of exploiting data locality and in-memory caching. To address this issue, it is important to use a fast interconnect and an asynchronous I/O model that overlaps the computation with the data transfers to hide communication latencies as much as possible. However, doing so may lead to an explosion of memory utilization required for buffering, which is in addition to the memory

978-1-5090-2453-7/16 $31.00 © 2016 IEEE
DOI 10.1109/CCGrid.2016.85

needed for user data.

In this context, shuffle strategies that minimize the auxiliary memory utilization are paramount, since memory is expensive and better used for actual data caching. This paper contributes a novel data transfer strategy that is specifically designed to operate efficiently under tight memory constraints. We summarize our contributions as follows:

- We formulate the problem of data shuffling under tight memory constraints and identify the associated challenges (Section II).
- We underline the design principles for an adaptive, memory-efficient data transfer strategy that addresses these challenges (Section IV).

II. Problem definition and challenges

Data shuffling is a fundamental data management primitive. In a broad sense, it refers to a set of n processes, each of which has a local dataset D_i partitioned into n pieces: $D_{i,1}, D_{i,2}...D_{i,n}$, each of which in turn is supposed to be accessed by another process. This can happen either in a pull mode (i.e., each process i fetches $D_{j,i}$, from each other process j) or in a push mode (i.e., each process i sends $D_{i,j}$ to each other process j). This pattern naturally appears in a broad range of data operations: parallel joins, aggregations, sorts, etc [4], [5].

In big data analytics, data shuffling is a key component of large-scale data aggregations. One widely-know example is MapReduce, in which mapper tasks shuffle the data to reducer tasks [2]. The newer generation of big data analytics frameworks, out of which the most representative is *Spark* [3], offers a rich set of data manipulation primitives in addition to *reduce*: *groupByKey*, *repartition*, *coalesce*, *cogroup*, *join*. All these operations rely on data shuffling and are leveraged by an entire higher level ecosystem of libraries: machine learning, graph processing, SQL query processing, etc. For simplification purposes, in this paper we abuse the term "reduce" operation to include the whole family of operations and primitives that rely on data shuffling, as applied to any framework used (e.g. Spark). By extension, the processes that are involved in the data shuffling and consume the shuffle blocks are referred to as "reducers".

At large scale, data shuffling typically involves huge amounts of data. In this context, it is not feasible to gather all the shuffle data before it is consumed, both because the data transfers would take a long time to complete and because a large amount of memory and local storage would be needed to cache it. As a consequence, a producer-consumer model is typically adopted, where the data transfers asynchronously accumulate shuffle blocks that are consumed by the computation. Even so, since data transfers are performed concurrently, this creates a complex all-to-all parallel communication pattern that puts a significant burden on the networking infrastructure. This in turn can cause I/O bottlenecks, which can lead to situations where the computation is blocked waiting for fresh shuffle blocks to arrive.

Besides the issue of waiting for shuffle blocks, the opposite also creates an important challenge: how to deal with the accumulation of shuffle blocks. When data transfers are fast, shuffle blocks may accumulate faster than they can be consumed, leading to an explosion of memory utilization. Using local memory for buffering intermediate shuffle data is not desired, because it is an expensive resource that otherwise could be used to cache actual user data. For example, aggressive caching is a key feature that Spark relies upon to deliver high performance. To mitigate this problem, each reducer typically uses a limited amount of memory that can be used to accumulate shuffle blocks from remote nodes. For example, Spark implements this limit as an upper bound on the amount of shuffle data that can be in transit from other remote nodes at any point in time. This limit is independently applied to each reducer. Since in the worst case (i.e. when shuffle blocks are not consumed) only up to the maximum amount of permitted in-flight shuffle data can accumulate, this simultaneously represents an upper bound on the memory used to accumulate the shuffle blocks. For the rest of this paper, we refer to this upper bound as the *in-flight reducer limit*. It is important to note that with an increasing number of cores per node, the number of reducers increases as well, which may lead to an explosion of overall memory utilization.

Given this context, we are faced with a difficult trade-off: on one hand it is desirable to accumulate as many shuffle blocks as possible in the background for each reducer, because this lowers the risk of blocking the computation. However, on the other hand it is important to minimize the memory utilization by placing tight in-flight limits on the reducers. In this paper, it is precisely this trade-off that we address. Our goal is to design a memory-efficient shuffle strategy that delivers high performance and is highly scalable, while reducing the memory utilization as much as possible within the hard upper bound given by the in-flight limit. For simplicity, we assume that all co-located reducers on the same node can trivially access each other's shuffle blocks (e.g. using shared memory or disks). Thus, the actual problem we focus on is how to transfer the remote shuffle blocks over the network.

III. Related work

How to optimize the performance of big data applications has been extensively studied in the context of MapReduce. Vertical scalability issued are explored in [6]. Overlapping the map phase with the reduce phase efficiently such that reducers do not lock out resources when idle is explored in [7]. Some studies show that CPU also can become a major bottleneck in big data analytics [8].

With respect to the storage layer and user data, improved concurrency control through multi-versioning can improve I/O data throughput significantly under concurrency compared with HDFS, as demonstrated by BlobSeer [9]. In-

memory caching as an additional layer on top of the storage layer is also gaining increasing attention recently [10].

With respect to data shuffling itself, the problem has been explored from multiple perspectives. Theoretical consideration was given in [11], where the authors present upper and lower bounds on the parallel I/O complexity of the shuffle phase. Low-level optimizations of the networking layer where data shuffling is explored in the context of high performance interconnects such as InfiniBand exist both for MapReduce [12] and Spark [13]. Furthermore, optimizations that are orthogonal to data transfers can be an effective complement: compression [14], [15], natural data redundancy [16], shuffle file consolidation [14].

IV. An Adaptive Memory-Efficient Shuffle Block Transfer Proposal

This section details our proposal for an adaptive shuffle block transfer strategy that addresses the challenges introduced in Section II. We focus on two aspects: (1) how to select a remote node where to get the shuffle blocks from; (2) when and how many shuffle blocks to fetch from the selected node.

A. Shuffle block source selection

We use three criteria for selection, detailed below.

Load balancing of data transfers using node-level coordination: In a large scale all-to-all parallel communication pattern, I/O bottlenecks are unavoidable due to the interference between the data transfers and potential load imbalances. As a result, it is important to coordinate the reducers in such way that they are aware of each other's intent, which enables better planning of the data transfers to avoid I/O bottlenecks. However, doing so is not without drawbacks, as this introduces an additional synchronization overhead that is necessary to facilitate collaboration. Since from a computational perspective the reducers can progress independently, there is no way to leverage an already existing synchronization point context to exchange such additional information (which is often the case for example in bulk-synchronous applications that use barriers). To address this trade-off, we propose to coordinate all reducers at node-level using shared in-memory data structures that keep track for each remote node of the total amount of in-flight data generated by all co-located reducers. This local view of the in-flight data can be used by the reducer to prioritize the node with the minimal load in order to improve load balancing. While this may not be globally optimal, it is an effective compromise given the large number of co-located reducers and the negligible performance overhead.

Prioritization based on node-level responsiveness: Load balancing alone is not enough to mitigate I/O bottlenecks: even if a node does not need to serve a lot of requests, it can still be less responsive than a heavier loaded node. This can happen because of multiple reasons: starvation due to unfair allocation of I/O bandwidth, high CPU utilization during the computation, etc. Furthermore, since load is measured from a local perspective, lack of responsiveness can simply happen because at the global level the node is in fact heavily loaded but this was not detected at local level. Thus, we propose to measure for all remote nodes how much time the reducers block waiting for its shuffle blocks. Based on this information, a moving average can be calculated for each remote node based on the most recent shuffle blocks. This effectively creates a measure of how responsive each remote node is from a local perspective, which enables the selection strategy to *adapt to the particular situation of each node independently* (e.g., a remote node may look unresponsive to a node with fast reducers but could be considered responsive otherwise).

Static circular load-balancing of the initial data transfers: One important standing issue is how to assign the initial fetch requests to the nodes: since there is no historical information about waiting times and no node has any in-flight pending requests, it is not possible to apply the previous two principles right from the beginning. To this end, we propose a third selection criteria that works as follows: lacking any additional information, each reducer prefers the remote node that is the closest successor to its local node that actually has shuffle blocks still to be fetched. Any predefined circular ordering of the nodes can be used, as long as all reducers from all nodes agree on it. Using this approach, reducers hosted on different nodes will prefer different remote nodes with high probability, which reduces the risk for I/O bottlenecks without any synchronization overhead.

B. Data transfer planning

In a limitless configuration where shuffle blocks can accumulate indefinitely before being consumed, the transfer algorithm adopted by a reducer is trivial: send an initial request that includes all shuffle blocks to each remote node and start collecting the results. When there is an in-flight limit in place, a request for a new shuffle block can be issued only if the size of the request is smaller than the in-flight limit. It is at this point when the selection strategy of the remote node becomes important. Using the selection strategy mentioned above, a trivial data transfer planning strategy would simply issue a new request whenever the in-flight size is below the in-flight limit. However, there are several important disadvantages when adopting such a trivial strategy, which we address below.

Shuffle block aggregation and request dispersal based on in-flight increment: First, issuing a separate request for each shuffle block can have a high overhead, especially if the size of the shuffle blocks is very small. Thus, it may pay off to wait until a request can be formulated for multiple blocks at once. Furthermore, it is also important to spread the requests among multiple nodes in order to reduce the risk of fluctuations and unresponsiveness that may happen during the data transfers that cannot be anticipated by the selection strategy. To this end, we define the *in-flight increment* as the minimum size that a request needs to have

in order to be issued. A request is never issued if it is not larger than the in-flight increment unless there are no more shuffle blocks that can be grouped together. Furthermore, a request is not allowed to be much larger than the in-flight increment. Specifically, whenever the in-flight size is below the in-flight limit minus the in-flight increment, new fetch requests are scheduled repeatedly based on the criteria introduced in Section IV-A until the in-flight limit is filled. Since the in-flight size of the remote node changes after each invocation, a different node is returned each time with high probability. This enables multiple parallel requests to different remote nodes, which reduces the risk of I/O bottlenecks.

Memory-efficient elastic in-flight reducer limit: Second, filling the whole capacity of the reducer up to its hard in-flight limit constantly may be sub-optimal, especially if the computation consumes the shuffle blocks at a slower rate. Thus, it is important to adapt the data transfer rate to the computation in such a way that it accumulates as few shuffle blocks as possible without causing waits, which minimizes the memory utilization. To this end, we introduce an elastic scheme that works as follows: initially, all reducers issue requests until they fill their hard in-flight limit. However, once shuffle blocks start accumulating, each reducer monitors the computation and records its average wait time. If the wait time is much smaller than the average, then its in-flight limit shrinks by the in-flight increment (but cannot shrink to less than the in-flight increment itself), otherwise it grows by the in-flight increment (without surpassing the hard in-flight limit). This elastic in-flight limit replaces the hard in-flight limit in all decisions.

Note that the in-flight increment can be optimized based on the networking infrastructure (latency, throughput, protocol, etc.). In combination with the elastic in-flight limit, each reducer effectively has a mechanism to adapt to its own computation and optimize its own memory utilization independently of the other reducers.

V. Discussion

We have formed the design principles presented above into a prototype shuffle data transfer strategy. We chose Spark as the framework to illustrate this strategy on, due to the large traction it received in recent production analytics platforms and solutions. The implementation details and experimental evaluations that demonstrate the benefits of this proposal for real-life Spark applications are outside the scope of this short paper and will be released in a subsequent publication.

Furthermore, we believe our proposal can be also extended in two other directions. First, we decided to avoid synchronization across nodes due to extra overhead. However, if this overhead can be masked by piggy-backing extra information on top of regular shuffle block transfers, then this could potentially be leveraged asynchronously for better selection and transfer planning. Second, we did not explore the interference between independent shuffles that run concurrently or the result that shows better stability of in-flight data. There are multiple interesting aspects to explore in this context, such as how to co-optimize independent shuffles or minimize interference with other (Spark or non-Spark) workloads.

References

[1] T. Hey, S. Tansley, and K. M. Tolle, Eds., *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, 2009.

[2] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[3] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI'12: The 9th USENIX Symposium on Networked Systems Design and Implementation*, San Jose, USA, 2012, pp. 15–28.

[4] G. Graefe, "Encapsulation of parallelism in the volcano query processing system," in *SIGMOD '90: The 1990 ACM SIGMOD International Conference on Management of Data*. Atlantic City,USA: ACM, 1990, pp. 102–111.

[5] C. Baru and G. Fecteau, "An overview of db2 parallel edition," *SIGMOD Rec.*, vol. 24, no. 2, pp. 460–462, May 1995.

[6] B. Nicolae, "Understanding Vertical Scalability of I/O Virtualization for MapReduce Workloads: Challenges and Opportunities," in *BigDataCloud '13: 2nd Workshop on Big Data Management in Clouds (held in conjunction with EuroPar'13)*, Aachen, Germany, 2013.

[7] J. Tan, A. Chin, Z. Z. Hu, Y. Hu, S. Meng, X. Meng, and L. Zhang, "Dynmr: Dynamic mapreduce with reducetask interleaving and maptask backfilling," in *EuroSys '14: Proceedings of the Ninth European Conference on Computer Systems*. Amsterdam, The Netherlands: ACM, 2014, pp. 2:1–2:14.

[8] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, "Making sense of performance in data analytics frameworks," in *NSDI'15: The 12th USENIX Conference on Networked Systems Design and Implementation*, Oakland, USA, 2015, pp. 293–307.

[9] B. Nicolae, D. Moise, G. Antoniu, L. Bougé, and M. Dorier, "BlobSeer: Bringing high throughput under heavy concurrency to Hadoop Map/Reduce applications," in *IPDPS '10: Proc. 24th IEEE International Parallel and Distributed Processing Symposium*, Atlanta, USA, 2010, pp. 1–12.

[10] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks," in *SOCC '14: Proceedings of the ACM Symposium on Cloud Computing*, Seattle, USA, 2014, pp. 6:1–6:15.

[11] G. Greiner and R. Jacob, "The efficiency of mapreduce in parallel external memory," in *LATIN'12: Proceedings of the 10th Latin American International Conference on Theoretical Informatics*, Arequipa, Peru, 2012, pp. 433–445.

[12] M. W.-u. Rahman, X. Lu, N. S. Islam, and D. K. D. Panda, "HOMR: A Hybrid Approach to Exploit Maximum Overlapping in MapReduce over High Performance Interconnects," in *ICS '14: Proceedings of the 28th ACM International Conference on Supercomputing*, Munich, Germany, 2014, pp. 33–42.

[13] X. Lu, M. W. U. Rahman, N. Islam, D. Shankar, and D. K. Panda, "Accelerating Spark with RDMA for Big Data Processing: Early Experiences," in *HOTI'14: IEEE 22nd Annual Symposium on High-Performance Interconnects*, Mountain View, USA, 2014, pp. 9–16.

[14] A. Davidson and A. Or, "Optimizing shuffle performance in spark," University of California, Berkeley - Department of Electrical Engineering and Computer Sciences, Tech. Rep., 2013.

[15] B. Nicolae, "On the benefits of transparent compression for cost-effective cloud data storage," *Transactions on Large-Scale Data- and Knowledge-Centered Systems*, vol. 3, pp. 167–184, 2011.

[16] ——, "Leveraging naturally distributed data redundancy to reduce collective I/O replication overhead," in *IPDPS '15: 29th IEEE International Parallel and Distributed Processing Symposium*, Hyderabad, India, 2015, pp. 1023–1032.